Constraint driven operation assignment for retargetable VLIW compilers

Constraint Driven Operation Assignment for Retargetable VLIW Compilers

Marco Bekooij

Constraint Driven Operation Assignment for Retargetable VLIW Compilers

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 maandag 12 januari 2004 om 16.00 uur

door

Marco Jan Gerrit Bekooij

geboren te Doorn

Dit proefschrift is goedgekeurd door de promotoren:

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

Cover: Layout of a VLIW processor which can execute 41 operations in parallel. Many of the techniques applied in the compiler for this processor are described in this thesis. The VLIW processor and the compiler are products of Silicon Hive.

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Bekooij, Marco Jan Gerrit

Constraint Driven Operation Assignment for Retargetable VLIW Compilers
Marco Bekooij.-
Eindhoven: Eindhoven University of Technology
Thesis Eindhoven. - With ref. - With summary in Dutch
ISBN 90-74445-60-8
Subject heading: constraint analysis, constraint satisfaction, compiler technology, code generation, signal processor architectures, VLIW-processors.

The work described in this thesis has been carried out at the Philips Research Laboratories in Eindhoven, The Netherlands, as part of the Philips Research programme.

Preface

The thesis in front of you is a result of my efforts at the Philips Research Laboratories. In this unique laboratory I have had the possibility to do for 4 years research on the same industrial relevant problem. This thesis would not be there without the contribution of fellow-researchers which I like to thank personally.

The major part of the research has been done in the Core Compiler Codesign (COCOON) project which was headed by Jeroen Leijten. I am grateful to him that he gave me the opportunity to work on constraint driven operation assignment techniques despite the large uncertainty about their applicability and the potential benefits of this approach. Exchange of valuable insights developed in the COCOON project have made a significant contribution to the approach presented in this thesis.

Without Jef van Meerbergen I wouldn't be able to obtain the same research success. He established the right boundary conditions and put my research in a broader context. He also urged me, often earlier then I wished, to apply the developed techniques on real-life design cases.

I also like to thank professor Jochen Jess for given me the change to obtain my Ph.D. degree from the Eindhoven University of Technology. He pushed me from the beginning in the right direction with his remark that a Ph.D. thesis should preferably be about a single small in-depth worked out problem. I would also like to thank Albert van der Werf for the opportunity he gave to continue the research on constraint driven code generation in the Monarch project.

The presented operation assignment techniques are an essential extension of the constraint driven scheduling techniques which were introduces by Adwin Timmer approximately 10 years ago. So called value lifetime serialization techniques were added by Bart Mesman. Koen van Eijk implemented these scheduling and serialization techniques in the research tool FACTS. This tool has been extended with an implementation of the in this thesis described operation techniques.

I am also thankful for the development version of A|RT-designer that was provided by the company Adelante Technologies. With this processor synthesis and compilation tool suite I was able to do an extensive evaluation of the developed operation assignment techniques. The animated discussions with Koen van Nieuwenhoven, André Collignon, Ivan Indinge and Kurt Du Pont of Adelante Technologies were experienced as very valuable.

Finally, I would like to thank Renée and her mother Clara Jacobs for their pleasant distraction

during writing of this thesis and I would like to thank my father for supporting me through the years.

Contents

Chapter 1

Introduction

This thesis describes constraint driven operation assignment techniques. These techniques are intended for application in a retargetable compiler that generates code for application domain specific Very Long Instruction Word (VLIW) processors. Multiple register files and an incomplete communication network in the data path are applied in these processors to improve their power-efficiency and scalability. However the use of more than one register file and an incomplete communication network make the assignment of operations to the functional units in the processor an important but difficult subtask in the compiler.

To derive a proper operation assignment, we apply constraint analysis techniques. These analysis techniques take explicitly the interconnect in the data path of the processor into account and remove infeasible operation assignment options from the assignment search-space. This way assignment decisions that would inevitably lead to violation of resource or timing constraints are prevented.

The organization of the remaining part of this chapter is as follows. In Section 1.1 we motivate why we focus on compilation techniques for application domain specific VLIWs. The retargetable compiler flow is presented in Section 1.2. Why operation assignment is an important but difficult subtask in the compiler is described in Section 1.3. The major contributions of this work are listed in Section 1.4. In Section 1.5 the differences between this work and operation assignment techniques known from the literature are indicated. Finally, Section 1.6 describes the organization of the thesis.

1.1 Application domain specific VLIWs

Today's embedded systems typically contain several processors, memories and peripherals that communicate with each other via an on chip network. Power dissipation usually limits the functionality that can be offered by these systems. Approximately 100 mW is seen as an acceptable power dissipation for battery supplied mobile systems, while the use of cheap plastic IC-packages limits the power dissipation to approximately 1 W for wired systems.

Specialization of processors towards their task can significantly improve their power-efficiency [HMV]. Therefore event driven and control dominated tasks are usually executed on general purpose processors such as offered by ARM [Fur96] or MIPS [MIP], while signal processing tasks are often carried out on classical Digital Signal Processors (DSPs) like the TI C54x [Lea97] or the R.E.A.L. [KLMW98].

Programming of general purpose processors is usually done in a high level language like C which is translated by a compiler into micro-code. On the other hand classical DSPs are often manually programmed in assembly code because the micro-code delivered by a compiler is often of an insufficient quality for these processors [Leu97]. However, programming in assembly code is a time-consuming and error-prone task which requires detailed knowledge of the DSP's instruction set.

For classical DSPs efficient code generation with a compiler is prohibited by the irregularities in the data path of these processors and the non-orthogonality of their instruction sets [ZVSM94]. Contrarily, for VLIW processors a compiler can generate high quality code because these processors have a regular data path and an orthogonal instruction set. However these processors have, for embedded applications, an often unacceptably low power-efficiency and micro-code density.

The code generation techniques described in this thesis are intended for the next generation VLIW processors. Unlike the first generation VLIW processors, exhibiting a single register file, the targetted VLIWs may contain several register files, an incomplete communication network and a refined instruction set optimized for digital signal processing. These processors are more difficult compiler targets but are more power-efficient than the first generation VLIW processors. On the other hand they are typically less power-efficient, yet significantly more compiler friendly compared to classical DSPs.

embedded processor type	performance			compiler friendliness			
	ILP	m. RF	rate.	ILP	c. RF	o. enc	rate.
general purpose	low	no	-	low	yes	yes	++
classical DSP	medium	yes	+	medium	no	no	- -
first generation VLIW	medium	no	+	medium	yes	yes	+
second generation application domain specific VLIW	high	yes	++	high	no	yes	-

Table 1.1: Characteristics of processors in the case they are applied for DSP applications. In this table stands the abbreviation "ILP" for instruction level parallelism, "m. RF" for multiple register files, "c. RF" for central register file, "rate." for rating and "o. enc" for orthogonal instruction encoding.

Another important characteristic of VLIW processors with multiple register files is that the number of parallel computation units in the data path is better scalable than the data path of processors with a central register file. The data path of a processor is well scalable if an increase of the number of parallel computation units in the data path does not result in a more than proportional increase in silicon area or in a significant decrease in the power-efficiency of

embedded processor type	power efficiency				code size		
	ILP	m. RF	Iset	rate.	i. enc	m. RF	rate.
general purpose	low	no	RISC	- -	medium	no	-
classical DSPs	medium	yes	CISC	+	high	yes	++
first generation VLIW	medium	no	RISC	-	none	no	- -
second generation application domain specific VLIW	high	yes	CISC	++	low	yes	+

Table 1.2: Continuation of Table 1.1 with characteristics of processors in the case they are applied for DSP applications. In this table stands the abbreviation “ILP” for instruction level parallelism, “m. RF” for multiple register files, “RISC” for Reduced instruction set, “CISC” for complex instruction set, “Iset” for instruction set, “rate.” for rating and “i. enc” for instruction encoding.

the processor. In processors with multiple register files the number of register fields and ports per file is typically lower than in processors with a central register file. Register files with a smaller number of ports and register fields are faster and more power-efficient than register files with a large number of ports and register fields. Therefore, processors with multiple register files are potentially more power-efficient and able to deliver a significantly higher computational performance than processors with a central register file. The characteristics of the processor types discussed above are summarized in Table 1.1 and Table 1.2.

1.2 Compiler flow

Our work is based on the retargetable compiler flow as shown in Figure 1.1. The algorithmic description of the application, for example in the C++ language, is the input to the compiler frontend. The frontend translates the description into an intermediate representation. During this translation the functionality offered by the data path of the processor, is taken into account. The intermediate representation together with a description of the target processor and a specification of the time constraints are the inputs of the code generator. The code generator produces an executable program. The operation assignment techniques, that are described in this thesis, are essential entities of the code generator.

1.3 Operation assignment challenges

Operation assignment is a difficult task in the case that processors are targeted with more than one register file and an incomplete network. In this case, a functional unit can access only a subset of the register files for reading or writing. Therefore, operations must be assigned to the functional units in such a way that these units are able to access the Register Files (RFs) in which their input values are stored. This is illustrated with a small example below.

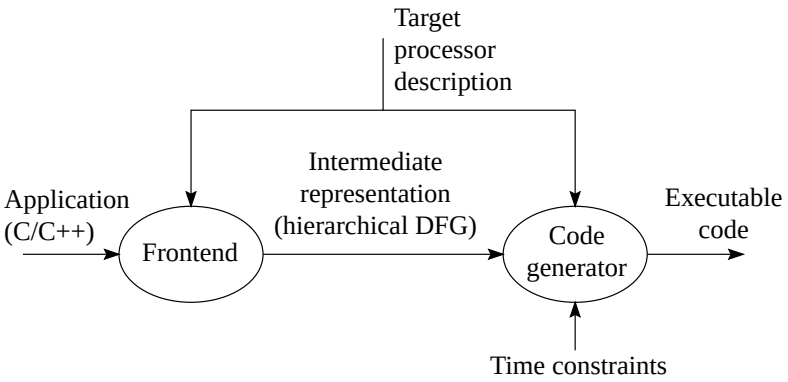


Figure 1.1: The applied compiler flow.

In Figure 1.2 a Data Flow Graph (DFG) and a data path of a processor are shown. The resources in the data path are the Functional Units (FUs), the registers in the register files, the interconnect in the connection network, and the input ports and output ports of those building blocks. All these resources are controlled by VLIW-instruction bits and can be used in parallel.

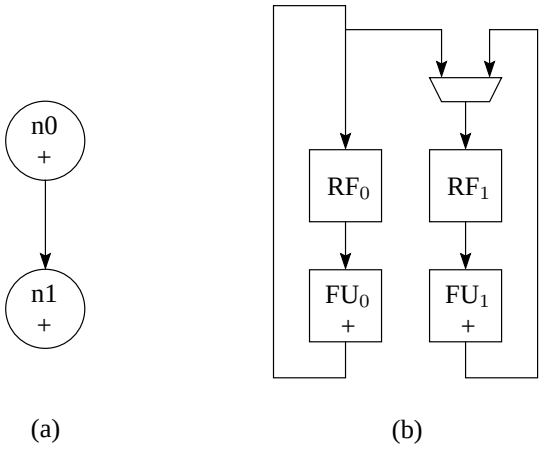


Figure 1.2: A DFG (a) and a data path instance (b).

A communication path is a path from the output port of a functional unit to the input port of a functional unit. This path traverses the connection network and a register file. Assignment of the operations in the DFG to the functional units of the target processor should be such that the required communication paths exist. For the DFG and the architecture of Figure 1.2, the requirement that a communication path from the producing FU to the consuming FU, has consequences for the assignment of the operations $n0$ and $n1$ to the functional units. For example, if operation $n0$ is assigned to FU_1 then also operation $n1$ must be assigned to FU_1.

In the case that the DFG is folded then operation $n0$ and $n1$ of succeeding loop iterations are executed in the same clock cycle. In this case those operations must be assigned to different FUs. There must also be a communication path from the FU that executes operation $n0$ to the FU that executes operation $n1$. These requirements can only be satisfied by assigning operation $n0$ to FU_0 and operation $n1$ to FU_1.

This example illustrates that the assignment of operations is complicated by the use of an incomplete communication network in the processor. Besides that the assignment should be such that there is a communication path between the functional unit that produces a result and the functional unit that consumes this result, this communication path should also be available. The communication path should be available, that is, the same path should not be used for the transfer of another result in the same clock cycle.

In the case that more than one register file is applied in the data path then the intermediate results produced by a functional unit must also be assigned to a register file. A register file has a limited capacity determined by the number of register fields it contains. The register file assignment should be such that the data to be held in a file does not exceed its capacity. The register file assignment is usually a result of the assignment of the operations and vice versa. Therefore, these two problems are tightly interrelated.

Multi-casting allows a result of a functional unit to be stored in more than one register file. With multi-casting it is often possible to generate an optimal schedule despite the use of many small register files in the data path. Multi-casting complicates operation assignment, because the number of used register file write ports depends on the assignment of the operations. The in this thesis described assignment techniques are intended for processors that support multi-casting.

Loop folding [Lam88] [Goo89] - which is also known as loop pipelining, or software pipelining - is supported because time critical parts in signal processing application are often encoded as for-loops. In folded loops successive for-loop iterations are executed simultaneously such that more instruction level parallelism can be exploited. An important challenge for code-generators is that loop folding requires that the application can be represented in a cyclic data flow graph.

We assume an hierarchical data flow graph [KMN+92] as the intermediate representation of the application. Our techniques cover entire applications ranging over sets of basic blocks.

Another issue is that operations without a resource conflict may use the same hardware resource in the same cycle. This is called operation merging. An example of the absence of a resource conflict is established by memory read operations that access the same memory location. An often occurring case is a constant being fetched by multiple read operations from the same memory location.

1.4 Main contributions

In this thesis constraint driven operation assignment techniques are described that effectively cope with multiple register files and an incomplete network in the data path of VLIW pro-

cessors. A tight coupling between operation assignment and scheduling (time assignment) is obtained by making use of constraint analysis techniques [EMAP+00].

We introduce in this thesis a conflict graph model of the assignment search-space. The consequences of an assignment decision are derived by pruning algorithms that operate on the conflict graph. Pruning of the conflict graph prevents unfortunate assignment decisions that could lead to a violation of resource or time constraints.

The assignment of the operations to the functional units was assumed as given, in previous work on constraint driven code generation for processors with an incomplete network [Tim95] [Mes01]. Given this assumption accurate bounds of the earliest as possible and the as late as possible start times of the operations could be derived with analysis techniques. These bounds are also needed to derive an operation assignment for which there is a schedule with a predefined schedule length or throughput. However, during the operation assignment process, some of the operations are assigned and others not. Given this partial assignment, similar analysis techniques are used to derived bounds on the start times of the operations. For run-time efficiency reasons we introduce operations of which the type can be adapted during the operation assignment process. After an operation is assigned, its type is adapted which reflects that this operation can only be executed on a specific functional unit.

A global write-back bus has been introduced in the network of the processor in order to guarantee that there exists at least one connection from an output of any functional unit to the input of any other functional unit. The bus eliminates the need for copy operations which simplifies the assignment problem drastically. The process of storing values in the background memory in case a register file capacity is exceeded is called “spilling”. The global write-back bus also guarantees that every intermediate value can be spilled.

The global write-back bus also makes the assignment of operations in a basic block to a large extend independent of the assignment of the operations in surrounding basic blocks. The reason is that the bus always provides a communication path from a producing operation in a surrounding basic block to a consuming operation in the inner basic block.

A so called block operation has been introduced which models a scheduled basic block. Block operations are used during the assignment of operations that belong to a basic block that surrounds other basic blocks. Value lifetime serialization of block operations and ordinary operations is possible because the number of registers per register file that are used inside a scheduled basic block is included in a block operation.

Most of the operation assignment techniques that are presented in this thesis have been implemented in our research tool FACTS. Interfaces with the A|RT Designer VLIW-DSP development environment [Ade] have been accomplished. This has allowed us to verify the correctness of the assignment techniques and has enabled the evaluation of the assignment techniques on industrially relevant examples.

1.5 Related work

A number of combined operation assignment and scheduling techniques are described in literature. An technique often used is list scheduling [Muc97] in which a priority function is used to select an operation to be assigned to a functional unit and a clock cycle. Sometimes "un-scheduling" of operations is applied [Goo89]. Close to optimal results are reported for processor architectures with a regular data path and an orthogonal instruction encoding. Contrarily a poor schedule quality is reported [Leu97] for processors with irregular data paths or a non-orthogonal instruction encoding.

Other assignment and scheduling techniques have been developed that generate high quality code at the expense of long compilation times. Examples are [HD98] which uses branch and bound techniques, [WLH00] which transforms the scheduling in an integer linear programming formulation, [Bea91] which solves the scheduling problem by means of genetic algorithms and [GFO92] which employs simulated annealing. Due to their time complexity, these techniques cannot ensure optimality given a limited time budget, and are only applicable to small code fragments and situations where long compilation times are acceptable.

The assignment technique described in this thesis is intended for VLIW processors with an orthogonal instruction set and almost homogeneous data path. In a homogeneous data path all functional units support the same data types. The described assignment techniques are intended for processors with more than one register file and an incomplete communication network.

Rather than optimizing cost according to a cost function, our assignment technique tries to satisfy the constraints imposed by a completely specified data path and the specified timing constraints. The technique searches for an assignment of operations that satisfies these constraints. The search-space is reduced by applying pruning algorithms that eliminate infeasible cases. This makes it possible to derive an assignment and a schedule or to detect infeasibility for DFG instances of a reasonable size (< 100 operations) and data paths with up to 50 FUs in typically less than a minute compilation time.

A difference with the operation assignment techniques described in the literature as compared to our techniques is that most other techniques insert so-called copy operations into the DFG. Copy operations are used to copy a value from a register file into another register file such that it is accessible by the appropriate functional unit. Usually the data path is designed in such a way that a valid mapping of the DFG can always be obtained by insertion of copy operations into the DFG. A disadvantage of the use of copy operations is that they can introduce a significant number of additional cycles to execute the application on a processor with multiple registers files compared to a processor with a single register file. It is also a difficult problem to decide during code generation where to insert copy operations into the DFG such that the performance penalty is minimized.

Our operation assignment techniques exploit multi-casting. Multi-casting implies storing of an intermediate value produced by a functional unit in more than one register file simultaneously. By making use of multi-casting there is usually hardly any performance penalty for the use of multiple register files in the processor. Also important is that the DFG does not change during operation assignment if multi-casting is applied. This enables the use of pow-

erful assignment search-space pruning algorithms. It also enables a computationally efficient incremental update of the schedule search-space.

Table 1.3 gives an overview of relevant properties of the assignment techniques for VLIW processors as described in the literature. The name of the first author of the article is in the first column. In the second column it is indicated whether assignment is done in an independent pre-processing step before scheduling (Y) or integrated with scheduling (N). A disadvantage of an independent pre-processing step is that the effect of the assignment of operations on the schedule cannot be taken into account, a situation that is avoided with our techniques. The third column contains an 'Y' if the technique is able to insert copy operations in the DFG. Some evidence is provided in Chapter 2 that copy operations typically impair the quality of the schedule. If the technique is intended for loops, an 'Y' can be found in the fourth column. The fifth column contains an 'Y' if the assignment technique takes the interconnect in the data path of the processor into account. Overall, a 'N' is preferable in column 2 and 3, while a 'Y' is the entry to be preferred in columns 4 and 5.

Author	Pre-proc.	Insert copy ops.	Loops	Inter-con.
Mattson [Ban98]	N	Y	Y	N
Banerjia [Ban98]	N	Y	N	Y
Bashford [BL99]	N	Y	N	Y
Desoli [Des98]	Y	Y	N	N
Ellis [Ell86]	Y	Y	Y	N
Kock [Koc95]	Y	Y	Y	Y
Nystorm [NE98]	Y	Y	Y	Y
this work	N	N	Y	Y

Table 1.3: Characteristics of the operation assignment technique.

This thesis describes operation assignment techniques which make use of constraint analysis. Therefore, the techniques match seamlessly with the other constraint analysis based code generation techniques [EMAP$^+$00] which are implemented in our research tool FACTS. The use of constraint analysis distinguishes this assignment technique from the operation assignment techniques described in the literature. The premise is that with constraint analysis, high quality schedules can be generated for processors that can execute many (> 10) operations in parallel. The data paths of these processors may contain many small register files and a sparsely connected network.

Processors with these characteristics are difficult compiler targets for which, as far as we know, all other code generation techniques often produce inferior schedules. However, the attractiveness of these processors is their superior power-efficiency.

Our code generation techniques can be re-targeted such that their effectiveness can be easily evaluated for a variety of processors. Retargetability requires that the techniques are, to a large extend, processor independent. Because the code generation techniques are intended for DSP processors they should support complex operations which are operations with more than two inputs and one output and can have several (pipeline) registers. An example of a complex operation is the Multiply ACcumulate (MAC) operation. This operation has 3 inputs

of which one input is typically sampled one cycle after the other two. The accumulation register can be considered as holding the state of the MAC-unit.

The described assignment techniques are intended to be applied on loops of which the schedule has a large impact on the performance. Loop folding is applied to obtain an increase in parallelism and performance at the cost of a slight increase in code size.

1.6 Thesis organisation

This thesis is organized as follows. Operation assignment implies that the operations of a data flow graph are assigned to functional units in the processor. The data flow graph and the target processors are defined in Chapter 2. In the same section the operation assignment problem is formulated. The data flow graph and a description of the target processor are the inputs of the code generator. The tasks performed by the code generation are described in Chapter 3. These code generation tasks are based on the constraint analysis strategy. This strategy requires that all possible assignments as well as the constraints are modeled in a representation which is called the “assignment search-space”. The assignment search-space is described in Chapter 4. It is pruned with the rules described in Chapter 5. Pruning prevents decisions that lead to infeasibility and provides this way a kind of look-ahead. Multi-casting is a very efficient way to distribute intermediate results into more than one register file. This essential feature of the presented operation assignment techniques is described in Chapter 6. How hierarchical dataflow graphs and merging of operations is supported is described in Chapter 7. These features make it possible to test the operation assignment techniques on processors and hierarchical data flow graphs generated with the A|RT-designer tool-suite. The quantitative results obtained in this environment are presented in Chapter 8. Finally, conclusions are drawn in Chapter 9.

Chapter 2

Inputs of the Code Generator

The inputs of the retargetable code generator are the intermediate representation of the application, a description of the data path of the target VLIW processor and the timing constraints. These three inputs are defined more precisely in this chapter.

The organization of this chapter is as follows. In Section 2.1 the characteristics of the class of target VLIW-processors dealt with in this thesis are described. Section 2.2 describes the intermediate representation of the application that will be mapped on the VLIW processor. Section 2.3 describes the timing constraint which can be specified by the user or derived automatically. Given the description of the inputs of the code generator the addressed operation assignment problem is restated in Section 2.4.

2.1 Target processor

The characteristics of the VLIW-processors, for which our operation assignment techniques are intended, are described in this section. The organization of this section is as follows. First, the characteristics of the processor's data path are given in the form of a data path template. This is followed by the motivation of the choice for processors with a distributed register file architecture. Then the connection network of the processor is defined. Finally, the applied instruction encoding is described.

2.1.1 Data path template

The template of the data path of the target VLIW processors is shown in Figure 2.1. Processors that adhere to this template have the following characteristics:

- Multiple Register Files (RFs). These register files can have more than one Write Port (WP) and more than one Read Port (RP),

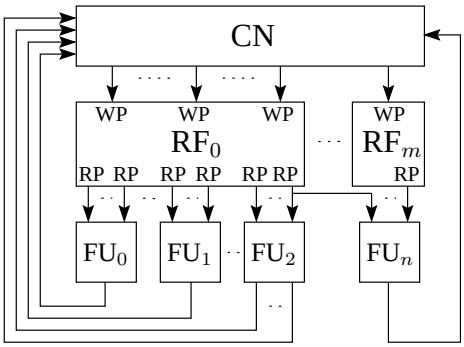


Figure 2.1: VLIW-processor data path template.

- Multiple Functional Units (FUs). These functional units can differ in the type of the operations that they can execute. Examples of operation types are addition, multiplication and memory load/store,
- A Connection Network (CN) between the outputs of the functional units and the write ports of the register files. This connection network could be incomplete and may contain shared busses. An example of a shared bus is the so-called "global communication bus" which will be introduced in Section 3.2.1,
- An orthogonal instruction encoding. An instruction encoding is orthogonal if the encoding of the instructions allows all combinations of operations to be executed in parallel [Wul81].

The data path template does not prescribe the register file architecture nor does it specify the internals of the connection network or the instruction encoding. The next subsection describe the register file architecture, connection network and the instruction encoding scheme for which our operation assignment technique is intended.

2.1.2 Register file architecture

From the data path template described in the previous section, processors with a central, a clustered and a distributed register file architecture can be derived. In this section we motivate the distributed register file architecture as being the most suitable architecture for our objectives.

For a processor with a central register file (see Figure 2.2) operation assignment is almost trivial because every functional unit can access all register fields in the central register file. A drawback of processors with a central register file is however the severely limited scalability of the data path of such a processor. The reason is that if the number of functional units in the data path is increased, in order to increase the peak performance of the processor, then the number of register file ports must be increased. It is likely that the number of intermediate

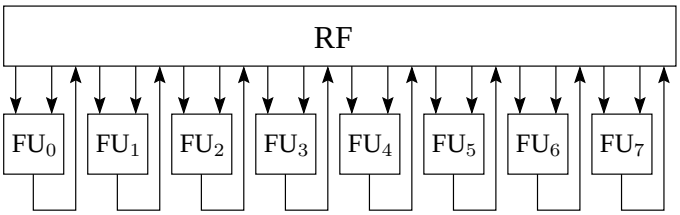


Figure 2.2: A VLIW-processor data path instance with a central register file architecture.

values that must be stored in the register file will also increase. Therefore, the number of register fields in the register file must be increased. In each register field an intermediate value can be stored. A larger number of ports and fields makes the register file slower and decreases its power-efficiency [RDK+00]. Therefore, it is likely that additional functional units in the data path reduce the power-efficiency and the maximal clock frequency of a processor with a central register file. A reduction of the clock frequency decreases the processors performance and makes the added functional units less effective.

This is confirmed by the results of our own experiments which are shown in Table 2.1. This table presents the critical path length and the logic cell area obtained with logic synthesis of register files with a varying number of data input and output ports and a varying number of register fields of 32 bits in an CMOS 0.18 μm technology at 1.5V and 125 C^o. Given that the total number of registers in the processor is kept the same, these experiments confirm that it is likely that the use of register files with a larger number of ports will have a negative effect on the processor's clock frequency. The reduction in clock frequency becomes more significant if it is taken into account that the actual critical path length after layout is approximately twice as long. This increase in critical path length is due the fact that the estimate of the critical path length made by the synthesis tool is based on a typical wire density in standard cell blocks while the wire density in register files is typically much higher. If we assume the validity of the rule of thumb that the power dissipation is proportional with the area, then the results also indicate that the power-efficiency of the processors is reduced if the number of register file ports is increased.

data ports	register fields/file	critical path ns	logic cell area μm^2	total logic cell area (32 fields) μm^2
2	4	4.5	12550	100400
4	8	4.3	39326	157304
8	16	7.0	122065	244130
16	32	8.3	205005	205005

Table 2.1: Characteristics of synthesized register files with 32 bit/field in 0.18 μm CMOS.

Multiple register files are applied in processors with a clustered register file architecture (see Figure 2.3) in order to improve the scalability of the data path. In such an architecture a register file together with the functional units that can read input data from this file and can

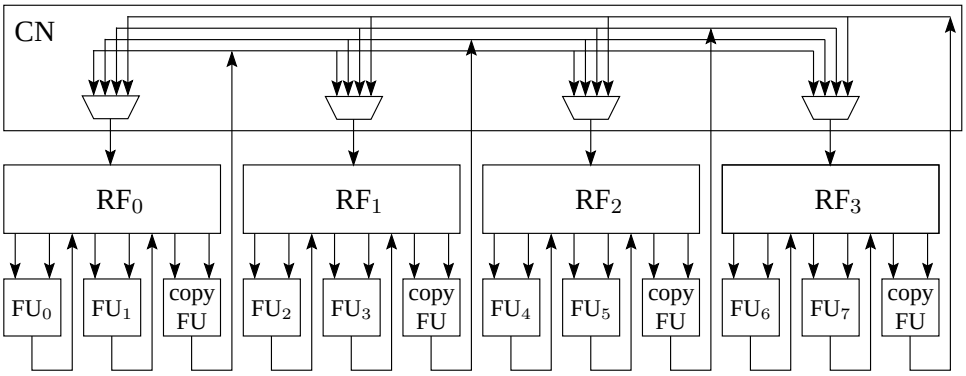


Figure 2.3: A VLIW-processor data path instance with a clustered register file architecture.

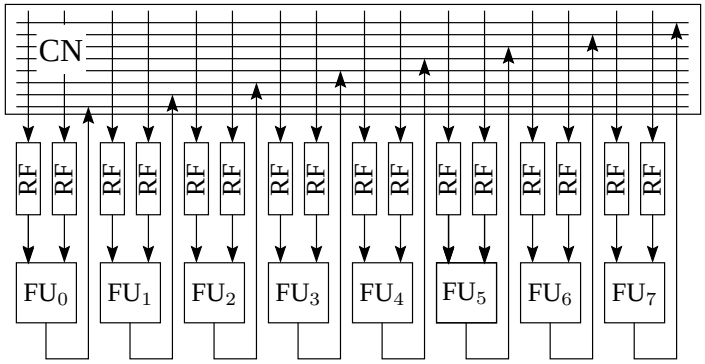


Figure 2.4: A VLIW-processor data path instance with a distributed register file architecture.

write the results to this file is called a cluster. The communication of data between clusters is performed by copy operations that are executed on the copy functional units which are indicated as “copy FU” in Figure 2.3 .

A major disadvantage of the clustered register file architecture, compared to a processor with the same functional units but a central register file, is the performance loss caused by the copy operations. The performance loss reported by Faraboschi et.al. [FDF98] is 15-20% in terms of clock cycles for a data path with 2 clusters and 25-30% in terms of clock cycles for a data path with 4 clusters. Similar results are reported by Gageldonk [Gag01]. These results were obtained for processors with a sufficient number of fields in the register files and a modest utilization of the copy units. Therefore, these results indicate that the performance loss is primarily caused by the latency of the copy operations. Because the performance loss increases significantly with the number of clusters we conclude that the scalability of the data path of a clustered architecture is limited.

A processor with a distributed register file architecture (see Figure 2.4) can become more

favorable in the case a large number of functional units is applied in the data path of the processor. By applying multi-casting it is possible to store an intermediate value produced by a functional unit in more than one register file without any additional operations for communication purposes. A performance loss of only 2% for processors with a distributed register file architecture that support multi-casting compared to a processor with a central register file is reported by Mattson et.al. [MDR+00]. We have obtained similar results with A|RT designer.

Also the width of the VLIW instruction word is affected by the register file architecture of the processor. The reason is that, with a central register file, every functional unit can access all the registers in the processor. Therefore, the number of bits per source or destination field in the VLIW instruction word equals $\lceil {}^2log(fields) \rceil$. However, the number of fields per file tend to be much smaller in a processor with a distributed register file architecture and thus also the number of bits per source and destination field. This reduction in bits is partially compensated by the fact that besides the register field also the producing functional unit output port and the destination register file must be specified in the destination field. If the connection network in the data path is sparsely connected then only a few functional unit output ports are connected to a register file write port. In this case only a few additional bits are needed in the destination field to specify one of these ports. For these processors it is plausible that the instruction width is smaller than for a processor with a central register file. This is supported by the figures in Table 2.2. This table contains the instruction width and the number of instructions for three applications that are mapped on an application specific VLIW processors with a central or a distributed register file architecture. The same functional units are applied in the processors with a central register file as in the processors with the distributed register file architecture. The processors were generated and programmed with A|RT designer.

design	central RF		distributed RF	
	instruction width (bits)	number of instructions	instruction width (bits)	number of instructions
fir	37	19	30	16
fft	72	61	64	60
ray-tracing	619	50	420	51

Table 2.2: Instruction code size of VLIW processors with a central and a distributed register file architecture.

It is likely that the instruction word size for a distributed register file architecture with a dedicated register file per input port is sub-optimal. The reason is that there are approximately twice the number of write ports as there are functional unit output ports in this architecture. A reduction in the number of write ports could possibly reduce the number of destinations fields in the VLIW instruction word. A reduction of the number of write ports can be achieved by merging several register files in one register file.

Our operation assignment techniques are intended for processors with a distributed register architecture because multi-casting can be relatively easily supported by our constraint analysis based code generation techniques while insertion of copy operation seems to be virtually impossible. Also the data path of a processors with a distributed register file architecture scales well and the usage of a large number of small register file reduces the instruction word

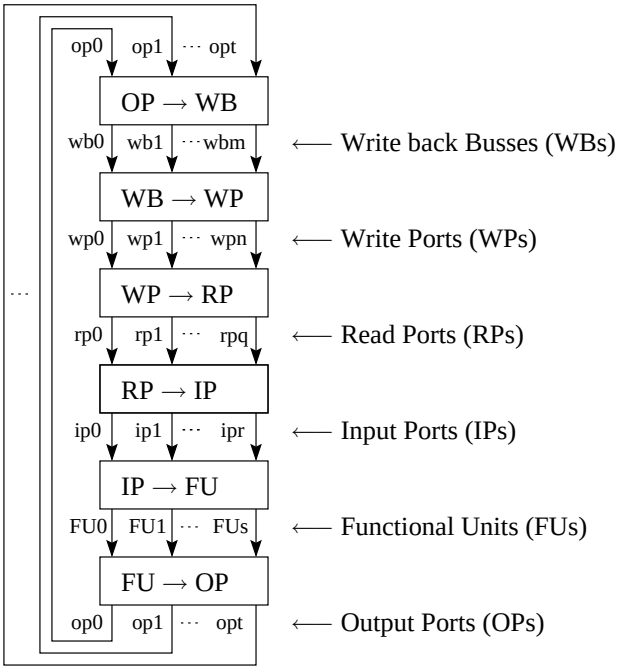


Figure 2.5: Generic network model of the data path template of Figure 2.1.

width and the code size.

2.1.3 Network model of the data path

There can be several communication paths in a processor from an output port of a functional unit, to an input port of a functional unit, through the connection network and a register file. All possible paths through the data path template of Figure 2.1 are captured in a network model which is shown in Figure 2.5. In Chapter 4 it is described how the assignment search-space is constructed by making use of this network model.

The network model of Figure 2.5 contains several sub-networks which are represented by boxes. These sub-networks are non-blocking in the sense of [Clo53]. That is, simultaneous data transport through this network from idle network input ports to idle network output ports can be performed if a connection can been made between these ports in the network. Within these networks not all possible connections may actually exist.

Every sub-network output port is in Figure 2.5 connected to one input port of another sub-network. Also, every sub-network input port is connected to one network output port. The input port of one sub-network has the same label as the output port of another sub-network to which it is connected.

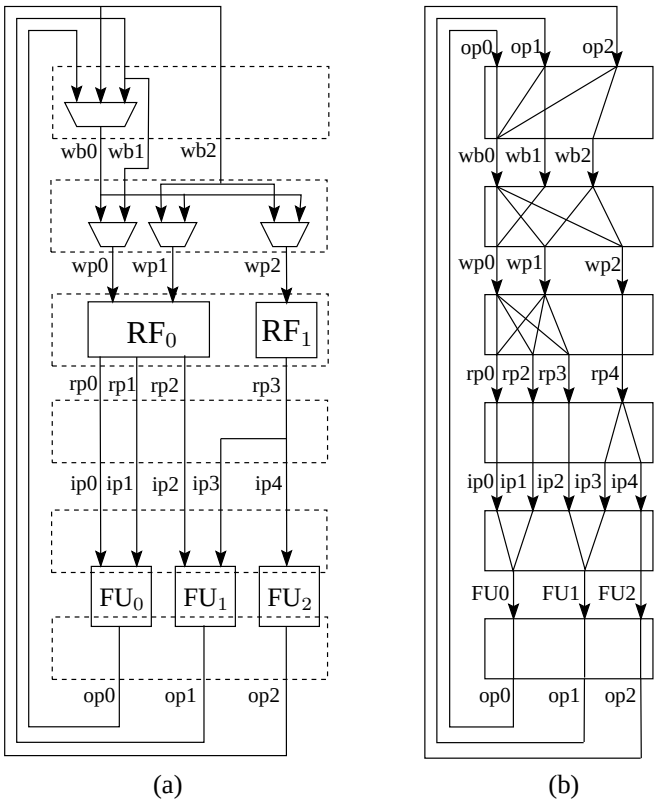


Figure 2.6: Processor data path (a) and corresponding network model (b).

An example of a network model of a data path is shown in Figure 2.6. This example illustrates how the shared bus wb0 can be included in the network model. This is a shared bus inside the connection network of the processor. Figure 2.6 also illustrates the possibility that more than one functional unit can obtain it's input data via the same register file read port.

The sub-network "WP→RP" is incomplete if more than one register file is applied in the data path. Therefore routing problems and network port conflicts should be taken into account during code generation even if all other networks are fully connected. These routing and port conflicts complicate operation assignment.

2.1.4 Instruction encoding

In this section we introduce "data-stationary instruction encoding" and "time-stationary instruction encoding".

In the data-stationary encoding [LBSL94] every instruction controls a complete sequence of operations that have to be executed on a specific data item, as it traverses the data pipeline. To

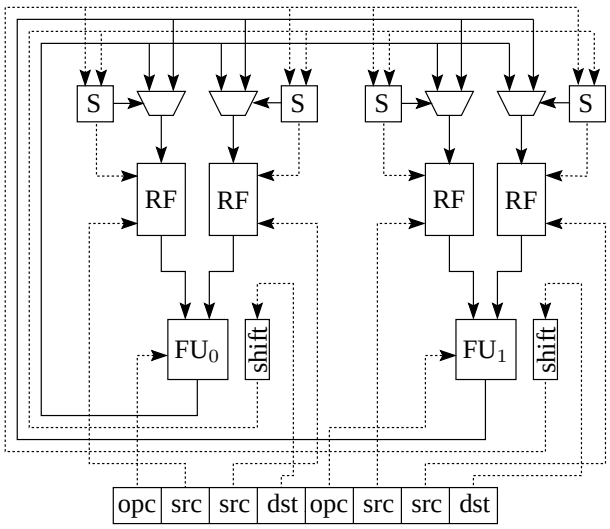


Figure 2.7: Control of the data path by the fields in a data-stationary encoded instruction.

achieve this, the opcode, which specifies the operation type to be executed, and the register source and destination addresses are located in the same instruction word. Once the instruction word has been fetched from program memory and decoded, the controller makes sure that the destination addresses become available in the correct clock cycle. Therefore, if an operation has a latency of two cycles then the destination address needs to be available one clock cycle later.

The data path and instruction word of a processor that uses a data stationary instruction encoding is shown in Figure 2.7. The dashed lines in these figures indicate how the bits in the fields are connected to the data path elements. In the destination field the destination register and the destination register file are specified. The content of this field is send to a shift register. If the latency of the functional unit is N cycles then the destination is delayed N-1 cycles by the shift register. After the destination field is delayed, it is send to the select units (S). These units determine whether the result should be stored in the register file and extract from the destination the control bits for the switches in the network as well a the destination register field.

An important disadvantage of the data-stationary instruction-encoding is that no efficient encoding scheme is known that supports multi-casting. The reason is that multi-casting requires the option to specify a variable number of destination addresses per opcode. If the length of the instruction word is fixed then the worst-case number of destinations, which equals the number of register file write ports in the data path, must be specified for every opcode. This increases the instruction word size more than necessary because in the typical case there is only one destination per opcode. For the same reason it is virtually impossible to encode efficiently, operations that consume in a time multiplexed fashion several input values via the same input port or that produce multiple output values. Modeling of this complex in-

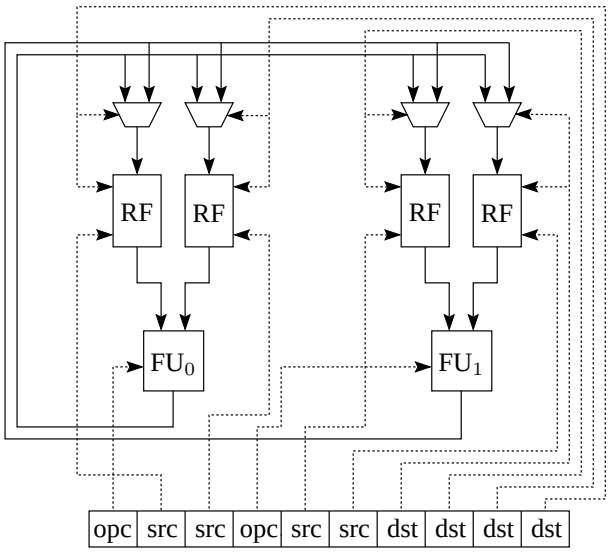


Figure 2.8: Control of the data path by the fields of a time-stationary encoded instruction.

put/output behavior with ordinary operation such that traditional scheduling techniques can be applied, is described in [BWB00]. Another drawback of the data-stationary instruction-encoding is that operations with a long latency require a lot of memory bits in the data path because the destinations are needed several cycles after the start of the operation and are delayed by a shift register in the processor. Multi-casting and operations with complex time-shapes and multiple destinations can be efficiently supported by applying a time-stationary instruction encoding [GPL+97]. This encoding scheme is described below.

Figure 2.8 shows how the data path is controlled by the fields of a time-stationary encoded instruction. The opcode (opc) field in the instruction determines the operation type that is executed on a functional unit. The source (src) fields determine the registers, from which the operands are retrieved. A destination field (dst) is associated with every register file write port. The destination field determines in which register a result is stored and from which functional unit output port this result is read. The dashed lines in Figure 2.8 indicate which data path elements are controlled by a field in the VLIW instruction. The bits in a field of a VLIW instruction word control these data path elements directly. Directly means that only wires are used to connect the bits to the right data path elements and that the content of the fields are not temporarily stored. Therefore, if an operation has a latency larger than one then the register destination is specified in a instruction word different from the one holding the corresponding opcode.

For time-stationary instruction encoded processors it is highly desirable that the scheduler schedules beyond *basic block* boundaries. A basic block is a maximal sequence of instructions that can be entered only at the first of them and exited only from the last of them [Muc97]. Without scheduling beyond basic block boundaries the pipeline registers

in the functional units must be empty at the end of a basic block. With deeply pipelined functional units this could lead to many extra instructions at the end of a basic block.

The operation assignment techniques described in this report are able to exploit multi-casting and are therefore intended for time-stationary encoded processors with a distributed register file architecture. The performance and code size penalty due to the time-stationary instruction encoding is in our case negligible because most of the functional units in the processors that we target are not deeply pipelined. Another reason is that the time critical for-loops in DSP applications are typically folded which is a form of scheduling of operations beyond basic block boundaries.

A data transport from a functional unit output port to a register file write port is encoded with a mix of bus-programming and socket-programming [Cor95]. Per write port it is encoded from which bus the data should be read (socket-programming) and per bus it is encoded from which functional unit output port the data should be read (bus-programming).

2.2 Intermediate representation of the application program

The second input of the code generator is a representation of the application program. This representation describes one basic block, which is a fragment of the application, in the form of a data flow graph. The complete application is represented in a hierarchical data flow graph. Executable code for complete applications is generated, by applying the operation assignment and scheduling techniques on all the basic blocks in the hierarchical data flow graph.

The organization of this section is as follows. First, the characteristics of the data flow graph representation of a basic block are described. Then, the characteristics of a flow graph and a hierarchical data flow graph representation of an application are described. For the case of processors with multiple register files it is motivated why a hierarchical data flow graph representation is likely to be a more suitable then a flow graph.

2.2.1 Data flow graph

Operation assignment is the assignment of operations in a data flow graph to resources in the processor. The data flow graph [KM92] represents the operations and the data and sequence dependencies of a basic block. Our data flow graph is defined as follows:

Definition 2.1 (Data Flow Graph.)
The tuple $(V, E_d \cup E_l \cup E_s, w_e, \mathrm{id}, t_i, t_o)$ defines a Data Flow Graph (DFG), where

- V is the set of nodes (operations),
- $E_d \subseteq V \times V$ is the set of data dependency edges,
- $E_l \subseteq V \times V$ is the set of loop carried data dependency edges,

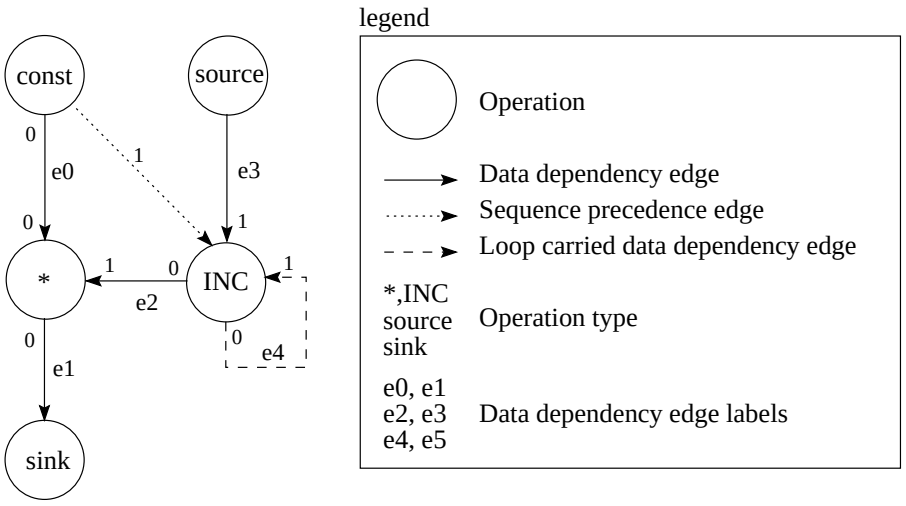


Figure 2.9: An arbitrary Data Flow Graph (DFG).

- $E_s \subseteq V \times V$ is the set of sequence precedence edges,
- $w_e : E_s \cup E_d \cup E_l \rightarrow \mathbb{Z}$ is a function describing the timing delay (in clock cycles) associated with each sequence precedence edge and data dependency edge. The timing delay w_e associated with an edge $e \in E_d$ is 1 and the timing delay w_e associated with an edge $e \in E_l$ is 0.
- id : $E_s \cup E_d \cup E_l \rightarrow \{0, 1\}$ is a function describing the iteration distance. The iteration distance of a loop carried data dependency edge $e \in E_l$ is 1. The iteration distance of the other edges is 0,
- $t_i : E_d \rightarrow \mathbb{N}$ is a function describing the operation input port number of the succeeding node, and
- $t_o : E_d \rightarrow \mathbb{N}$ is a function describing the operation ouput port number of the preceding node.

A DFG example is shown in Figure 2.9. An operation in this DFG has a type and can be executed on data path resources that support this type. Examples of data path resources are functional units and network ports. When an operation is executed it uses a data path resource.

Operations in a basic block can consume values produced by operations outside the basic block. The operations outside the basic block are represented in the DFG by source operations. Conversely, a value produced in a basic block can be consumed by an operation that does not belong to the basic block. These consuming operations are represented in the DFG by sink operations.

The data dependency edges in the DFG specify that an operation can only be executed after its input values are produced. Data edges might be labeled with a unique label. These labels are used to refer to a specific data edge. The input and output port associated with a data dependency edge are respectively specified at the head and the tail of the edge. Source and sink operations have dummy ports. All input and output ports, except dummy ports, are respectively mapped on input and output ports of a functional unit. The convention is used that operation input port 0 is mapped on the left most functional unit input port. Operation input port 1 is mapped on the next functional unit input port to the right, and so on. Operation output ports are mapped in a similar way to functional unit output ports. If the mapping of operation ports is irrelevant or trivial for a problem that is discussed then the port labels are left implicit.

If the DFG is executed repeatedly then a value produced in one iteration can be consumed by an operation executed in the next iteration. This type of data dependency is called a loop carried data dependency and is represented in Figure 2.9 with a dashed edge.

Multiple data precedence edges leave the same output port of an operation if this operation produces a result that is consumed by multiple operations. However, typically only one data edge enters an input port of an operation. Only in the case that the execution of producing operations is mutually exclusive then the data dependency edges that leave these mutual exclusive operations can enter the same input port of an operation.

Sequence precedence edges are depicted in Figure 2.9 with dotted edges. A timing delay w_e is specified in the middle of a sequence precedence edge.

A data dependency edge has a weigth of 1. Operations with a latency larger than one clock cycle are modelled as a chain of operations in the DFG. Sequence precedence edges are used to enforce that operations in the chain must start in consecutive clock cycles. This way of modeling typically results in modest increase of the run-time of the scheduler.

The iteration distance id defines a weight of an edge in multiples of the initiation interval (II) of the schedule. The total weight of an edge $e \in E_s \cup E_d \cup E_l$ is $w(e) = -\mathrm{II} \times \mathrm{id}(e) + w_e(e)$. The iteration distance $\mathrm{id}(e)$ is 1 for loop carried data dependency edges and 0 for the other edges.

The task of the scheduler is to assign each operation $v \in V$ a start time $s(v)$. The start times are constrained by the available resources and the data dependencies and sequence precedences. A data dependency edge or sequence precedence edge (v_i, v_j) states that:

$$s(v_j) \geq s(v_i) + w(v_i, v_j) \tag{2.1}$$

The weight w of an edge can be negative. In this case equation 2.1 can be rewritten in the form of equation 2.2 to make the interpretation easier. Equation 2.2 states that if $w(v_i, v_j) < 0$ then v_i may not start more than $|w(v_i, v_j)|$ cycles after the start of $s(v_j)$. Negative sequence edges are used for example to model pipelined operations. An example of a pipelined operation is a multiply accumulate operation in which the addition must start one cycle after the multiplication. A fragment of a DFG with a multiply accumulate operation is shown in Figure 2.10.

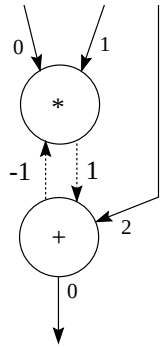


Figure 2.10: A fragment of a DFG with a multiply accumulate operation.

$$s(v_i) \leq s(v_j) + |w(v_i, v_j)| \qquad \text{if } w(v_i, v_j) < 0 \tag{2.2}$$

Before we define the distance between two operation, the notion of a path is introduced:

Definition 2.2 (Path)
A path P of length d from operation v_i to operation v_j is a chain of precedences $v_i \rightarrow v_k \rightarrow ... \rightarrow v_l \rightarrow v_j$ that implies $s(v_j) \geq s(v_i) + d$ with $d = \sum_{e \in P} w(e)$.

Definition 2.3 (distance)
The distance $d(v_i, v_j) \in \mathbb{Z} \cup -\infty$ from operation v_i to v_j is the length of the longest path from v_i to v_j. If there is no path from v_i to v_j then the distance $d(v_i, v_j) = -\infty$. The distance $d(v_i, v_i)$ is per definition 0.

2.2.2 Flow graph representations

Application programs can be represented in a flow graph [ASU86], in which edges indicate the flow of control and nodes represent basic blocks. In these graphs there is a directed edge from basic block B1 to block B2 if B2 can immediate follow B1 in some execution sequence, that is if:

1. there is a conditional or unconditional jump from the last statement of B1 to the first statement of B2, or
2. B2 immediate follows B1 in the order of the program, and B1 does not end in an unconditional jump.

A C-source code fragment is shown in Figure 2.11. The corresponding flow graph is shown in Figure 2.12.

```
t=0;
for(i=0; i<10; i++){
  t+=a[i]*b[i];
}
x[0]=t;
```

Figure 2.11: C-source code fragment.

The values that are alive at the end of a basic block are typically transferred into background memory. In order to reduce the number of store and corresponding load operations, it is possible to save some of the values in registers. Because these registers hold values across basic block boundaries they are called global registers and the values saved in these registers are called global values.

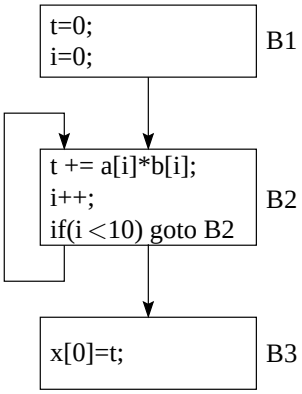


Figure 2.12: Flow graph of a C-source code fragment.

The assignment of a fixed number of global values to the global registers is typically performed by the compiler frontend. This approach has a couple of drawbacks. First, the available number of global registers is not always the best. It can, for example, be desirable to use fewer global registers because a larger number of registers is required to hold the local intermediate values of the basic block. Another drawback of this approach is that unfortunate global variable assignments made by the compiler frontend could lead to the situation that global values must first be copied into another register file before they can be accessed by the appropriate functional unit. It is likely that these copies will significantly impair the schedule quality in the case of processors with a large number of register files.

An alternative representation of an application is a hierarchical data flow graph [Lam88]. In this hierarchical data flow graph, control-flow primitives such as branching and iteration are modeled by means of the hierarchy [Mic94]. The edges in such a graph represent data dependencies or sequence precedence constraints and the nodes represent ordinary operations or block operations. Block operations can be expanded in operations and precedence edges.

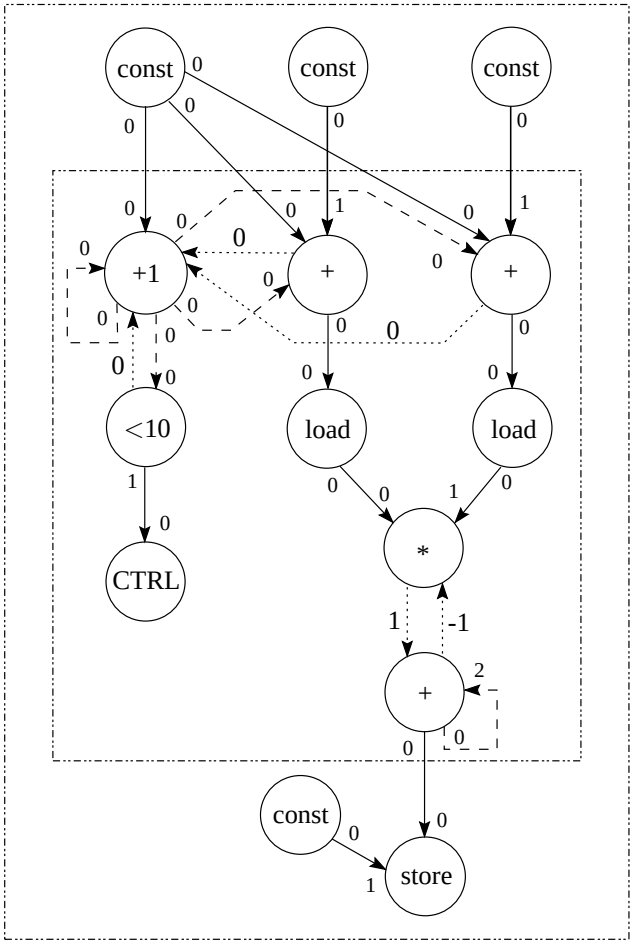


Figure 2.13: Hierarchical data flow graph of the C-source code fragment in Figure 2.11.

A hierarchical data flow graph can be scheduled by making use of hierarchical reduction [Lam88]. In this technique, the operations in the deepest nested block operation are scheduled first. Then this block operation is scheduled together with the other operations of the surrounding block operation. This is repeated until all operations in the block operations are scheduled.

An example of a hierarchical data flow graph is shown in Figure 2.13. This graph is the intermediate representation of the C-source code fragment shown in Figure 2.11. In this graph the comparison operation produces an output value which signals the controller whether the loop should be terminated.

During the hierarchical reduction all values are implicitly assigned to register files during operation assignment. There is no difference between values that are local to one basic block

and values that cross basic block boundaries. The operation assignment routines take the communication paths in the network into account and can be made aware of the register file pressure. Also complete basic blocks, which are represented as block operations, and other operations can be serialized [Mes01] in order to adapt the lifetimes of variables such that the values that are alive across basic blocks fit in the register files. Therefore hierarchical data flow graphs seem to be a more suitable intermediate representation for our purpose.

2.3 Timing constraints

The timing constraints are specified as the latency and the initiation interval of the schedule. The Latency (L) of a schedule is the number of clock cycles after which all the operations in the DFG are executed. For loops an initiation interval can be specified. The Initiation Interval (II) of a schedule is the number of clock cycles after which the next execution of the same DFG is started. If II $< L$ then the loop is folded. The user can specify these timing constraints or a shell around the code generator can be used to determine sharp lower bounds of these timing constraints.

The steps performed in this shell, are shown in Figure 2.14. In the first step, a lower-bound estimate of the initiation interval is calculated. A lower bound on II can be derived from the number of available resources of a certain type and the number of operations of the same type in the DFG. This resource lower bound II_{res} ignores the data dependencies and sequence precedences in the DFG. Another lower bound on II can be derived from the length of the loops in the DFG. This lower bound II_{loop} ignores the number of available resources. A more accurate lower bound on II is the maximum of II_{res} and II_{loop}, so

$$\text{II} = max(\text{II}_{res}, \text{II}_{loop}) \tag{2.3}$$

With equation 2.4 the resource lower bound is derived. In this equation rsu is the number of operations of type τ_i and rsa the number of available resources of that type.

$$\text{II}_{res} = \max_{\tau_i \in \tau} \lceil \frac{\text{rsu}(\tau_i)}{\text{rsa}(\tau_i)} \rceil \tag{2.4}$$

According to definition 2.2 implies a path P in the DFG from operation v_i to operation v_j that:

$$s(v_j) \geq s(v_i) + \sum_{e \in P} w_e(e) - \text{II} \times \sum_{e \in P} \text{id}(e) \tag{2.5}$$

For a loop c in the DFG, for which $s(v_j) = s(v_i)$, equation 2.5 can be rewritten as:

$$\text{II} \geq \frac{\sum_{e \in c} w_e(e)}{\sum_{e \in c} \text{id}(e)} \tag{2.6}$$

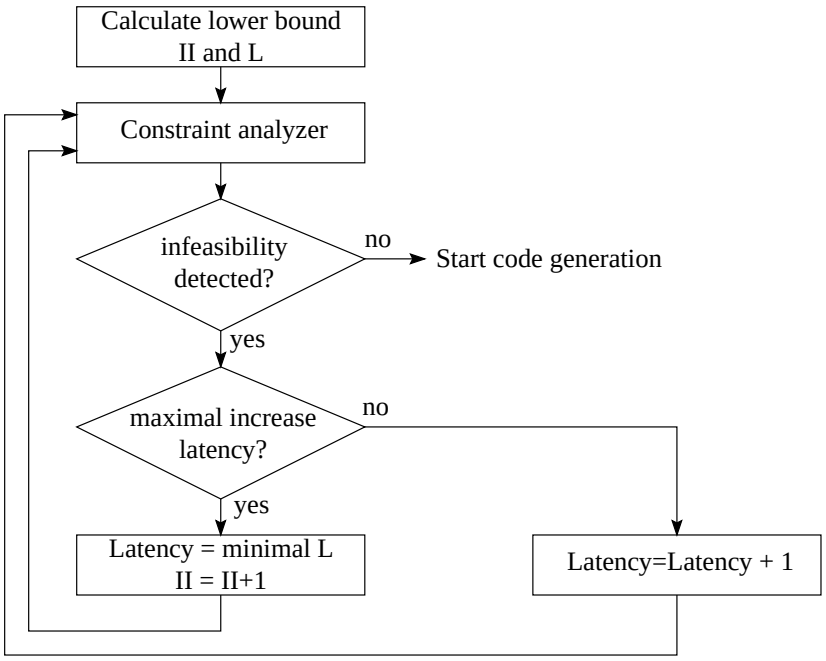


Figure 2.14: Procedure to determine sharp lower bounds for the Latency (L) and Initiation Interval (II) constraint of the schedule.

Because this must hold for every loop $c \in C$ in the DFG, a lower bound on II due to loops in the DFG becomes:

$$\mathrm{II}_{loop} = \max_{c \in C} \lceil \frac{\sum_{e \in c} w_e(e)}{\sum_{e \in c} \mathrm{id}(e)} \rceil \tag{2.7}$$

The longest path in the DFG from which the loop carried data dependencies are removed is used as the lower bound on the latency of the schedule.

Given the initial timing constraints the schedule search-space is pruned by the constraint analyzer which is described in more detail in Section 3.1.1. If the schedule search-space is empty after pruning then infeasibility is reported and the latency constraint is increased with 1 cycle. If the latency constraint is increased a predefined number of times and still infeasibility is reported then the initiation interval is increased with 1 cycle and the latency constraint is reset to the longest path in the DFG. This process is repeated until constraint analysis stops reporting infeasibility. In this process the latency is increased before the initiation interval is increased because the initiation interval determines to a large extend the execution time of a folded loop.

During the process, which is described in the previous paragraph, the timing constraints are relaxed. Another approach would be to start from a situation with a feasible schedule and then tightening the timing constraints. This approach was not chosen because our benchmark

results indicate (see Chapter 8) that often solutions can be found close to the lower bound timing constraints. Therefore, it takes less compilation time when the search process is started from the lower bound timing constraints.

The lower bound timing constraints are derived given a DFG and a target processor. These lower bounds are used as initial timing constraints. In the case a schedule is found that respects these initial timing constraints, then this schedule is also an (performance wise) optimal schedule. However in the case the compiler frontend generates for the same C-program another DFG then it could be that for this DFG a schedule exists for even tighter timing constraints. Another example is that in the compiler frontend, the selection of the operation input ports is performed. If this input port selection is left undefined for commutative operations then this additional freedom could be exploited in order to obtain a better schedule. Our assignment techniques do not exploit this freedom.

2.4 Problem statement

Given the definition of the target processor and the intermediate representation of the application, the operation assignment problem can be stated as follows:

Problem 1: Given a DFG, find an assignment of operations to functional units and intermediate values to register files such that a schedule can be established which satisfies the specified initiation interval and latency constraints.

If the initiation interval is smaller than the latency then loop-folding should be applied. It is assumed that the assignment of operation ports to functional unit ports is specified beforehand. It is also assumed that the assignment of memory read and write operations to a memory is specified. Each memory is modeled as a functional unit in the data path. Scheduling of operations together with block operations should be possible. It is assumed that the VLIW-processors have a distributed register file architecture, a potentially incomplete communication network, and that they support multi-casting.

Chapter 3

Code Generation by Traversing the Search-Space

During code generation a DFG is mapped on the data path of a target VLIW processor. The code generation task is split is several subtasks which are performed after each other in order to obtain high quality schedules in a reasonable compilation time. These subtasks, which are also called phases in literature, are described in Section 3.1. The phases are mutually dependent, that is, decisions in one phase potentially affect the decisions in another phase. How dependencies between phases are handled, is described in Section 3.2. All code generation phases are based on constraint analysis. The basic principles of constraint analysis are described in more general terms in the last section of this chapter.

3.1 Code generation phases

Our code generator gets as input a DFG, an abstract description of the target processor and timing constraints and produces executable code (see Figure 1.1). The code generation task is partitioned in subtasks, which are called phases because they are successively executed. These subtasks are supposed to be easier to handle individually. This procedure should lead to a better result than solving the problem in one piece. In every phase, only one type of decision is made by an algorithm, which is special for this phase. An example of a decision type is the assignment of operations to functional units.

In the code generator there is an operation assignment phase, a value lifetime serialization phase, a scheduling phase, and finally a register binding phase (see Figure 3.1). During the operation assignment phase, operations are assigned to functional units. During the lifetime serialization phase, operations are ordered in time in such a way that a valid register binding exists after the scheduling phase. During the scheduling phase, the start times of operations are determined. After scheduling, the register binder selects a register for every intermediate value.

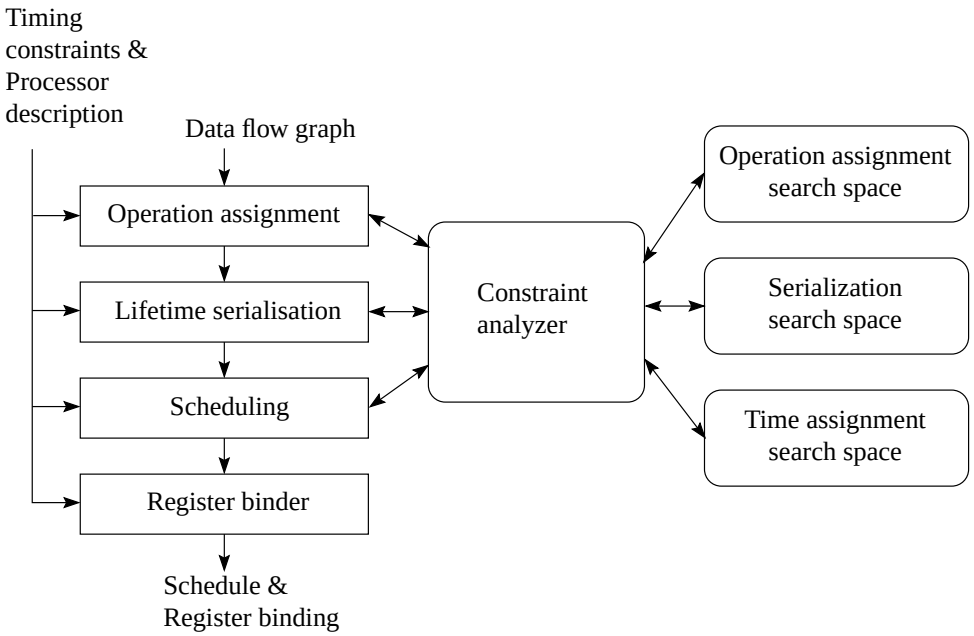


Figure 3.1: Code generation phases.

The assignment of operations to functional units determines implicitly the register files in which the input values of these operations must be stored. The assignment of values to register files is called the register file binding. Register file binding is performed before lifetime serialization because the lifetime serialization techniques require that the register file binding is defined. Lifetime serialization takes place before scheduling because this allows the schedule freedom to be used to satisfy the register file capacity constraints.

The decisions that are taken in the different phases are interdependent. Unfortunate decisions can remove all feasible options in a succeeding phase. Therefore, the code generation phases interact with the constraint analyzer. The constraint analyzer queries and updates the search-space representations of the phases. In the search-space of a phase all feasible values of the decision variables are represented as well as the most relevant constraints for that phase. The most relevant constraints for the operation assignment phase are the routing constraints. However, also the known resource conflicts are modeled in this search-space. The most relevant constraints for the serialization phase are the number of registers per register file. For the scheduling phase are the precedence constraints the dominant constraints.

The consequences of a decision in a phase are evaluated by pruning the search-space representation of that phase. After pruning, additional constraints are derived from this search-space that are incorporated in the search-spaces of other phases. The pruning proceeds consecutively through the subsequent search-spaces. This pruning process is continued until pruning does not result in any new constraints. This way the effects of a decision on future decisions in succeeding phases is taken into account. In other words, the pruning techniques are applied in order to achieve a tight coupling of the phases.

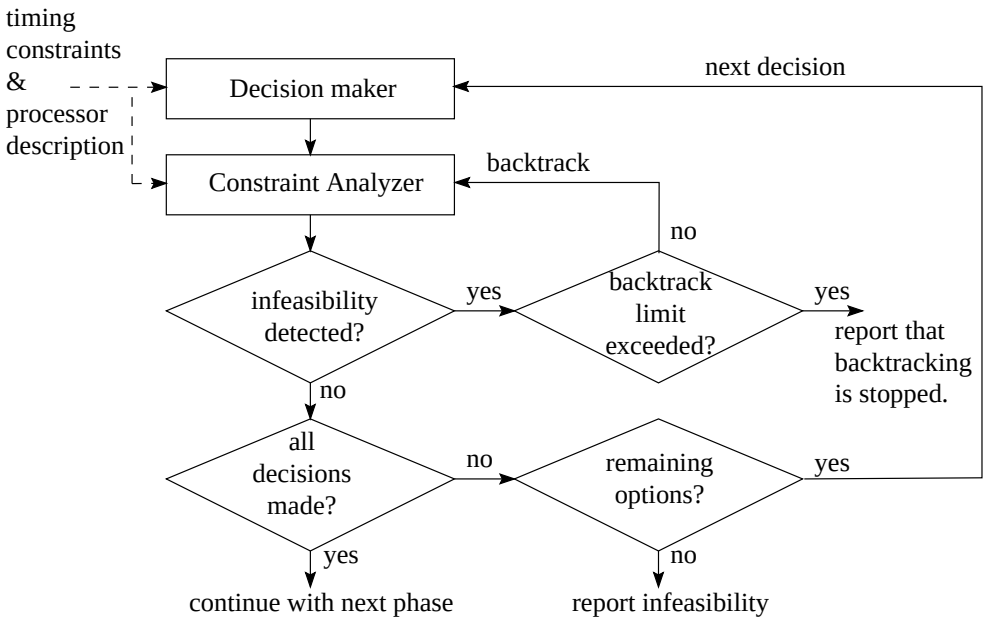


Figure 3.2: Flow-diagram of the steps taken in a phase.

The entire process of code generation is essentially a backtracking scheme proceeding through the search-spaces. The steps taken in any one of the phases is shown in the flow diagram of Figure 3.2. The first step in this diagram is performed by the decision-maker, which decides to add a certain additional constraint based on some priority function. An example of such a constraint is that a particular operation must be executed on a specific functional unit out of a set of equivalent functional units.

After a decision is taken, the consequences on the search-space are evaluated by the constraint analyzer. In case the constraint analyzer detects that the search-space is empty after pruning, then infeasibility is reported. If infeasibility is reported, then a backup step is performed. This step removes the last decision and recovers the situation as it was just before the last decision. The backtracking process is aborted if the number of backtracks, which is equal to the number of backup steps, exceeds a predefined value. In this case, it is reported to the user that no solution was found but that it is not guaranteed that no solution exists. In the case that the backtrack limit is not exceeded and that there are still variables to which more than one value can be assigned then the next decision is taken. In the case that all options are examined without detecting a feasible solution then infeasibility it is reported to the user.

The coupling of the phases is an essential property of our code generator. Some basic knowledge about the schedule search-space representation and the most essential schedule search-space pruning rule makes it possible to understand the techniques used to achieve this coupling. The schedule search-space representation and the pruning rule are described in the next subsection. The phase coupling techniques are described in Section 3.2.

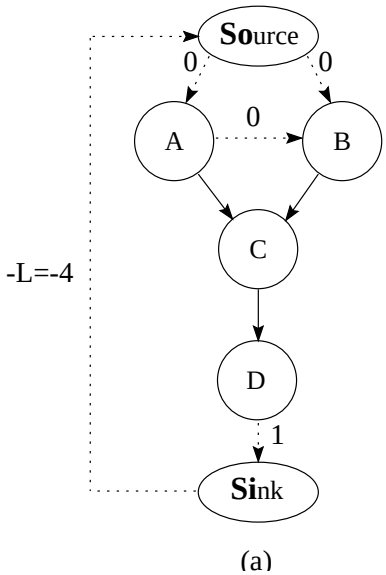


Figure 3.3: An example DFG of which the distance matrix before and after pruning is shown in respectively Figure 3.4 and Figure 3.5.

3.1.1 Schedule search-space pruning

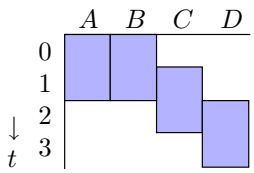
The schedule search-space is represented in a distance matrix [Mes01] in which the distance is stored between every pair of operations in the DFG. These distances are derived with an all-pairs shortest path algorithm [CLR90] as is described in [Mes01]. The distance matrix for the DFG in Figure 3.3 is shown in Figure 3.4a. The latency (L) of the schedule is assumed to be constrained to 4 clock cycles. This constraint is captured with the sequence edge from the sink to the source. The initiation interval (II) is constrained to be 5 clock cycles. The distance $d(v_i, v_j)$ is stored in location row i, column j of the distance matrix. A schedule interval representation can be derived from the distance matrix. The boundaries of an interval are the distances $d(\text{source}, v_i)$ and $-d(v_i, \text{source})$. For a feasible schedule the start time $s(v_i)$ of operation v_i must be in the interval $d(\text{source}, v_i) \leq s(v_i) \leq -d(v_i, \text{source})$.

A pair of operations may have a “soft” or a “hard” resource conflict. They have a “soft” conflict if they claim the same resource. They have a “hard” resource conflict if they both use a resource of the same type in the same clock cycle. Soft conflicts can be resolved by scheduling the operations in different clock cycles. Hard conflicts can be resolved by assigning the operations to different resources.

If we assume that operation A and B in Figure 3.3 have a soft resource conflict then rule 1 from [MSTM97], which is for convience repeated below, derives that operation B must be scheduled after operation A. This is the case because the sequence precedence edge between operation A and B in the DFG prevents that operation B can be scheduled before operation A and the soft resource conflict forbids to schedule them in the same cycle.

	So	A	B	C	D	Si
So	0	0	0	1	2	3
A	−1	0	0	1	2	3
B	−1	−1	0	1	2	3
C	−2	−2	−2	0	1	2
D	−3	−3	−3	−2	0	1
Si	−4	−4	−4	−3	−2	0

(a)

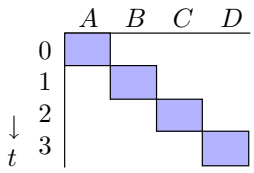


(b)

Figure 3.4: The distance matrix (a) and schedule intervals (b) for the DFG in Figure 3.3 before applying rule 1.

	So	A	B	C	D	Si
So	0	0	1	2	3	4
A	0	0	1	2	3	4
B	−1	−1	0	1	2	3
C	−2	−2	−1	0	1	2
D	−3	−3	−2	−1	0	1
Si	−4	−4	−3	−2	−1	0

(a)



(b)

Figure 3.5: Distance matrix (a) and schedule intervals after applying rule 1.

Rule 1 If $d(v_i, v_j)$ *mod* II $= 0$ and there is a soft resource conflict between operation v_i and v_j then a sequence edge (v_i, v_j) with weight $d(v_i, v_j) + 1$ is added.

The updated distance matrix and schedule intervals, after the sequence edge with a weigth $w_e(v_A, v_B) = 1$ is introduced by rule 1, are shown in Figure 3.5.

Backtracking of the last assignment decision occures in case infeasibility is detected by the constraint analyzer. A computationally efficient way to detect infeasibility during the assignment phase is described in the next subsection. This approach is explained in more detail in the rest of this thesis.

3.1.2 Operation assignment infeasibility detection

The assignment search-space is updated by the constraint analyzer after every assignment decision made in the assignment phase. In this search-space all assignment options, routing constraints and the hard resource conflicts are modeled. The assignment search-space is modeled in a conflict graph. The details of this model are described in Chapter 4.

The assignment search-space allows to select an operation and to determine it's assignment freedom, represented by the set of functional units on which it can be executed. For such an operation it is decided to which functional unit it will be assigned. These decisions are

modeled as additional constraints in the conflict graph. Infeasibility can be detected by an attempt to color this conflict graph with a selected set of colors. The success of this attempt indicates feasibility, while failure entails backtracking. The legalization of this procedure is supplied by the assertion that this graph can be colored with a selected set of colors if and only if there exists an assignment that satisfies the routing constraints as well as the known hard resource conflicts.

After every assignment decision an updated conservative estimate of the execution times of operations is calculated by the constraint analyzer and therefore new hard resource conflicts might pop-up which will be incorporated in the conflict graph. Detection whether the last assignment decision is feasible or not would require that the updated conflict graph is colored after every operation assignment decision. However, because coloring of the conflict graph is to computationally intensive, it is replaced by pruning. During pruning infeasible operation assignment options are detected and removed from the search-space with algorithms that have a polynomial-time computation complexity. The pruning algorithms are described in Chapter 5. The improvement in compilation time, by making use of pruning, comes at the cost of weaker guarantees. With pruning the guarantee is restricted to, that given the assignment of *all* operations infeasibility will be detected if a routing constraint or a hard resource conflict is violated. Given this weaker guarantee it can occur that the violation of a routing constraint, as a consequence of an assignment decision, is detected after many subsequent assignment decisions which leads to a lot of backtracking. However, by making use of pruning it is still possible to guarantee that all routing constraints are respected when the assignment phase is finished and all operations are assigned.

One of the pruning rules in Section 5.1, operates directly on the conflict graph. Because it operates on the conflict graph it takes all routing constraints as well as the hard resource conflicts into account. However with this pruning rule we can not guarantee that given the assignment of *all* operations, infeasibility will be detected, if a routing constraint is violated. This must be guaranteed because lifetime serialization and scheduling can not resolve routing constraint violations. Therefore, another pruning algorithm is applied which gives this guarantee. This algorithm is described in Section 5.2. It operates on its own internal model but the results are incorporated in the conflict graph. This algorithm takes only routing constraints into account and does not account for the hard resource conflicts, therefore the pruning rules in Section 5.1 and 5.2 are repetitively applied till no additional infeasible assignment options are detected by these rules.

3.2 Phase coupling

In this section the mutual dependency between the assignment and the schedule phase is illustrated with an example. This is followed by a description of the technique used to achieve coupling of the assignment and the schedule phase.

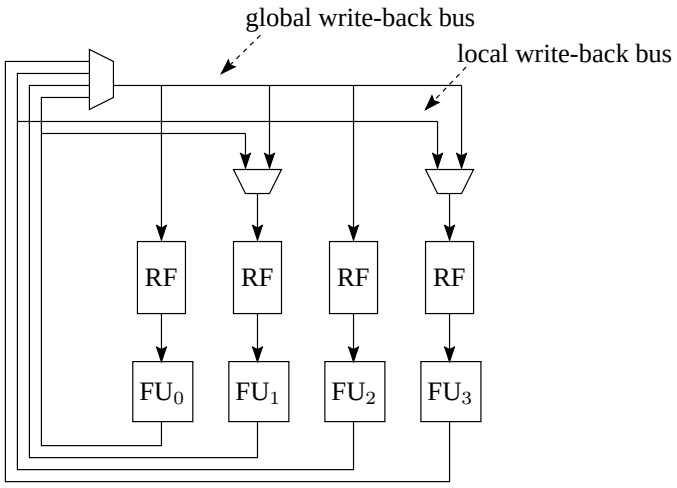


Figure 3.6: Data path used to illustrate the mutual dependency between operation assignment and scheduling.

3.2.1 Phase coupling example

The data path in Figure 3.6 and the DFG in Figure 3.7 are used to illustrate the effect of the assignment of the operations on the schedule. In this data path a global write-back bus is applied that provides a connection from every functional unit output port to every register file.

Assume that the objective is to map the data flow graph of Figure 3.7 on the data path of Figure 3.6 such that the initiation interval II=1. All intermediate values can be transported via the local write-back busses if the operations $n0$, $n1$, $n2$, and $n3$ are assigned to the functional units FU_0, FU_1, FU_2, and FU_3 respectively. Alternatively, if the operations $n0$, $n1$, $n2$ and $n3$ are respectively assigned to functional units FU_1, FU_0, FU_3, and FU_2 then the global write-back bus should transport two intermediate values. However, the global write-back bus can transport only one intermediate value per clock cycle. Therefore, an initiation interval of 1 clock cycle can not be obtained given this assignment.

After every assignment decision we evaluate the use of resources. If the number of required resources in a cycle exceeds the number of available resources then the last assignment decision is undone and another assignment is examined.

3.2.2 Schedule search-space pruning given a partial assignment of operations

After pruning of the assignment search-space, there potentially remain less functional units on which an operation can be executed. If after pruning it becomes clear that an operation

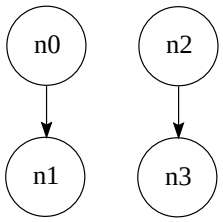
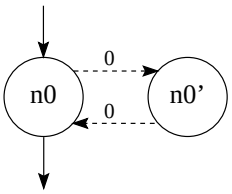


Figure 3.7: DFG used to illustrate that it is desirable that during operation assignment the usage of the global write-back bus is taken into account.



Given:
$n0$ is of type MULT
$n0'$ is of type MULT1

Assumed resource constraints:
4 resources MULT
1 resource MULT1

Figure 3.8: The resource usage of operation $n0$ is specialized by adding operation $n0'$.

must be executed on a particular unit then this knowledge is used to restrict the schedule search-space.

For example if after pruning it is known that operation $n0$ must be executed on multiplier 1 (MULT1) then a virtual operation is added that models this additional constraint in the DFG. This operation is virtual because it is introduced merely for modeling purposes. Only one resource is made available that has the same type as the added operation. Because a new operation that uses a resource was added during assignment, it is called a dynamic resource usage.

How the virtual operation is added in a DFG, is shown in Figure 3.8. The sequence edges enforce that operation $n0$ and the virtual operation $n0'$ are scheduled in the same cycle. Therefore is it possible to use the same entries in the distance matrix for $n0$ as well as for $n0'$ which results in a smaller distance matrix.

The selection of a resource out of a set of equivalent resources, has been given the name "resource usage specialization". Which resource is selected is modeled in the DFG with a virtual operation. Specialization of a resource restricts the schedule search-space. A restriction of the schedule search-space is implemented with an run-time efficient algorithm that performs an incremental update of the distance matrix (see [Mes01]).

The resource usage of the added virtual resources is evaluated by the constraint analyzer. This constraint analyzer applies the in Section 3.1.1 described rule 1 which potentially results in a restriction of the schedule search-space.

In a similar way the resource usage of network ports and functional unit ports is specialized.

Additional constraints are also derived from the schedule search-space and modeled in the assignment search-space. The rule used to derive these constraints is described in Section 4.3.2.

3.3 Constraint analysis strategy

All code generation phases in our scheduler are based on the constraint analysis strategy. This section describes the constraint analysis strategy in more general terms. This provides insight in the strengths and weaknesses of the strategy. It also indicates the applicability of this strategy for problems other than code generation.

In subsection 3.3.1 we describe the relation between combinatorial optimization, search and decision problems and constraint satisfaction problems. The operation assignment problem is a combinatorial search problem which is formulated as a constraint satisfaction problem. In subsection 3.3.2 we describe the applied constraint analysis strategy used to find a solution of this constraint satisfaction problem. We prove in subsection 3.3.4 that the operation assignment problem is $\mathcal{NP}$-hard.

3.3.1 The relation between combinatorial problems

Many abstract problems can be formulated as combinatorial optimization problems [CCPS98]. An instance of a combinatorial optimization problem can be defined as the problem of finding a solution that is optimal with respect to a given objective function f, in a finite set of solutions S. The *objective function* $f : S \rightarrow \mathbb{R}$ determines the *cost* of a solution. An optimization problem is either a minimization or a maximization problem. Without loss of generality we restrict ourselves to treating minimization problems which are formally defined as:

Definition 3.1
An instance of a *combinatorial minimization problem* is a pair (S, f), where S is a finite set of *candidate solutions* and $f : S \rightarrow \mathbb{R}$ is the *objective function*. One is asked to find a solution $s \in S$ for which $f(s)$ is minimal, i.e., $f(s) = min_{s' \in S} f(s')$

The objective of a combinatorial search problem is to find a solution $s \in S$ for which a condition addressed as the goal predicate is true. A combinatorial search problem is formally defined as:

Definition 3.2
An instance of a *combinatorial search problem* is a pair (S, g), where S is a finite set of *candidate solutions* and $g : S \rightarrow \{true, false\}$ is the *goal predicate*. One is asked to find a *solution* $s \in S$, if one exists, for which $g(s)$ is true.

Combinatorial minimization problems with an integral objective function $f : S \rightarrow \mathbb{N}$
can be solved by making use of a corresponding combinatorial search problem. In this corresponding search problem the cost is defined in advance and is checked by the goal predicate.

A solution for the combinatorial minimization problem is then obtained by solving the combinatorial search problem for a cost of 0. This cost is increased with one until a solution $s \in S$ is found for which $g(s)$ is true.

An instance of a combinatorial decision problem can be associated with an instance of a combinatorial search problem. A combinatorial decision problem is defined as:

Definition 3.3
An instance of a *combinatorial decision problem* is a pair (S, h), where S is a finite set of *candidate solutions* and $h : S \rightarrow \{true, false\}$ is the *goal predicate*. One is asked if there exists a solution *solution* $s \in S$ for which $h(s)$ is true.

In a combinatorial decision problem the question is answered whether there is a solution while in combinatorial search problem the solution is also derived if it exists. Therefore a combinatorial search problem is at least as difficult as the associated combinatorial decision problem.

The theory of $\mathcal{NP}$-completeness [GJ79] formalizes the difference between "hard" and "easy" decision problems. "Easy" decision problems are solvable in polynomial time. For the "hard" problems, there are no polynomial time algorithms known that solve these problems despite considerable research effort.

The most difficult decision problems belong to the class of $\mathcal{NP}$-complete problems. If the decision variant of a combinatorial search problem is $\mathcal{NP}$-complete then the search problem is said to be $\mathcal{NP}$-hard. In Section 3.3.4 it will be shown that our operation assignment problem belongs to the class of $\mathcal{NP}$-hard problems. Many combinatorial search problems, including our operation assignment problem, can be formulated as Constraint Satisfaction Problems (CSP) [Mon74]. A constraint satisfaction problem consists of the following:

- A set of n variables $\{x_1, ..., x_n\}$ in the discrete, finite domains $D_1, ..., D_n$
- A set of m constraints $\{c_1, ..., c_m\}$ which are predicates $c_k(x_i, ..., x_j)$ defined on the *search-space* which is the Cartesian product $D_i \times ... \times D_j$. If c_k is TRUE, then the assignment of values to the variables is said to be consistent with respect to c_k.

The question is to find an assignment a of values to the variables such that all constraints are satisfied, i.e., $\forall_{1 \leq i \leq m} c_i(a)$. The assignment a specifies for each variable x_i a value v_i from the domain D_i.

In the next section a variant of the backtracking algorithm is described that is used to find a solution for our combinatorial search problem which is formulated as a constraint satisfaction problem.

3.3.2 Search algorithm based on constraint analysis

As described in the previous section, the constraint analysis strategy can be applied if the original optimization problem is rewritten as a combinatorial search problem which can be

formulated as a constraint satisfaction problem. In order to solve the combinatorial search problem a search-space is constructed. This search-space contains the discrete, finite set of options that are considered as potential solutions to the decision problem. All these options are considered one by one. In the case an option is found that satisfies all the constraints then the search process is stopped and a solution is returned. In the case all options are considered but there is no one that satisfies the constraints then obviously there is no solution.

In practical implementations, the result of the combinatorial search problem should be derived within a finite amount of time. Because we often have to deal with $\mathcal{NP}$-hard problems, the run time can be exceptionally long. Therefore the answer "May be a solution exists" is returned in the cases when the maximum run-time is exceeded and the search is aborted. If this occurs, searching for a solution is continued for relaxed constraints.

Exploration of all options in a search-space can be performed with a backtracking algorithm [LP98]. A backtracking algorithm is shown in Figure 3.9. In line 2 the problem P is split in the subproblems $p_1, ..., p_r$. These subproblems can be created by adding an additional constraint. An additional constraint can for example be included by selecting a value $v_i \in D_i$ for variable x_i. Each of these subproblems $p_i \in P_b$ is tested in line 5 of the backtracking algorithm. The $test(p_i)$ function can return OK or $Continue$. If the test replies that the subproblem is a solution then $test(p_i)$ returns OK and the backtracking algorithm is halted and returns the value OK which indicates that a solution was found. If test(p_i) equals $Continue$ then the subproblem is included in the pending set of problems P.

```
BACKTRACK(P)
 1  branch out of P in P_b = {p_1, p_2, ..., p_r}
 2  if P_b = ∅
 3    then return OK
 4    else for each p_i ∈ P_b
 5         do testResult = test(p_i)
 6            if testResult = OK
 7              then return OK
 8              else if testResult = Continue
 9                     then if Backtrack(p_i) = OK
10                            then return OK
11  return Infeasible
```

Figure 3.9: Backtracking algorithm.

The backtracking algorithm splits a problem in smaller subproblems till it detects that one of the subproblems is a feasible solution. In this case the Backtrack(P) algorithm returns OK. If all subproblems $p_i \in P_b$ are infeasible then no solution exists and the algorithm returns in this case $Infeasible$. In the constraint analysis strategy several techniques are applied in order to improve the average run-time of this algorithm. The applied techniques are pruning, symmetry detection and bottleneck identification.

Pruning techniques remove infeasible cases from the search-space that can easily be derived. Easily means in this case that the infeasible cases are identified with algorithms exhibiting a

polynomial-time computational complexity. These infeasible cases do often not satisfy the constraints of a relaxation of the original problem.

Another technique that reduces the search-space is symmetry analysis [EMT99]. This analysis derives equivalent options. Because the options are equivalent, all options except one can be removed from the search-space.

Bottleneck identification determines the order in which the options in the search-space are selected. In other words, bottleneck identification is a heuristic which selects the most "critical" variable x_i and the value v_i for variable x_i for which it is "most likely" that this choice does not result in infeasibility. Bottleneck identification is applied after every decision because then an updated representation of the search-space is obtained in which new bottlenecks might pop-up. The bottleneck identification heuristics used during lifetime serialization are extensively described in [Mes01] and in [AP02]. The heuristic used during operation assignment is described in Section 7.3.

The structure of an algorithm which is based on the constraint analysis strategy is shown in Figure 3.10. In line 1 of this algorithm, the problem P is split into a set of subproblems P_b. The problem P is split in subproblems by selecting with a bottleneck identification heuristic a value v_i for variable x_i. If the problem cannot be split in subproblems then the algorithm CA returns *OK* otherwise the actual search-space given the selected subproblem p_i is pruned by the function *analyze()* in line 5. This function returns *Infeasible* if the search-space is empty after pruning. If there remain options in the search-space after pruning then the subproblem p_i is split in new subproblems in a recursive call of algorithm CA in line 7. If infeasibility is detected after pruning or the recursive call of the algorithm CA in line 7 did not return *OK* then there are two options depending on whether the backtrack limit is exceeded. In case the backtrack limit is not exceeded then the search-space before selecting p_i in line 4 is calculated by the $recoverSearchSpace()$ function. After the search-space is recovered another subproblem in P_b is evaluated. If all the subproblems in P_b are infeasible then *Infeasibility* is returned by the algorithm CA. In the other case, in which the backtrack limit is exceeded, the value *Continue* is returned by the CA algorithm. This value indicates that no solution was found by the CA algorithm but that it cannot be excluded that a solution exists.

A nice feature of constraint analysis is that tight constraints often improve pruning and help this way to find a solution. A drawback of constraint analysis is that the implementation is usually more complex and slower than a greedy algorithm. The additional complexity must be justified by the quality of the solution derived by the constraint analysis algorithm compared to the quality of the solutions derived with a greedy algorithm given a fixed run-time budget.

3.3.3 Releated work on constraint analysis

Time and Resource Constraint Scheduling Problems (TRCSP) are solved as special cases of a Constraint Satisfaction Problem (CSP) in [Nui94]. The objective of a time and resource constraint scheduling problem is the derivation of the start time and the set of resources used by the operations such that all constraints are satisfied. The CSP problem is solved with a tree search algorithm which is to a large extend similar to the search algorithm in Fig-

```
CA(P)
 1  branch out of P in P_b = {p_1, p_2, ...p_r}
 2  if P_b = ∅
 3    then return OK
 4    else for each p_i ∈ P_b
 5         do analyseResult = analyze()
 6            if analyseResult ≠ Infeasible
 7              then caResult = CA(p_i)
 8                   if caResult = OK
 9                     then return OK
10            if (analyseResult = Infeasible ) ∨ (caResult ≠ OK )
11              then if backtracks ≤ backtrackLimit
12                     then backtracks = backtracks + 1
13                          recoverSearchSpace()
14                     else return Continue
15  return Infeasible
```

Figure 3.10: Search algorithm based on constraint analysis.

ure 3.10. Different terminology is used for pruning and bottleneck identification. Pruning is called consistency checking and bottleneck identification is called variable and value selection. An important difference with our work is that we focus on the operation assignment and scheduling for VLIW processors. Our restricted scope enables exploitation of problem specific knowledge which results in problem specific pruning and bottleneck identification algorithms. An example are the constraints which are a result of an incomplete network. In the TRCSP problem formulation these constraints are treated as additional constraints. However we model routing constraints in a problem specific search-space representation. This representation is pruned with a generic pruning algorithm as well as pruning rules which are based on the fact that the constraints are a consequence of an incomplete communication network.

In [BL99] the Constraint Logic Programming (CLP) platform ECLIPSe [WNS97] is used to solve the operation assignment and scheduling problem for classical DSPs. The problem is described as a minimization problem with as cost function the latency of the schedule. The applied tree search technique is based on the branch and bound strategy. A basic technique in CLP is constraint propagation which is a different name for pruning. Despite that no problem specific pruning rules for code generation were applied, the same code quality was produced as in the case of the hand written code for some examples of the DSPStone [ZVSM94] benchmark set. No result were reported for DFGs with more than 20 operations.

3.3.4 Computational complexity of the assignment problem

In this section we will prove that the decision variant of the operation assignment problem belongs to the class of $\mathcal{NP}$-complete problems and thus the operation assignment search problem is $\mathcal{NP}$-hard. The proof is based on a reduction from the $\mathcal{NP}$-complete subgraph

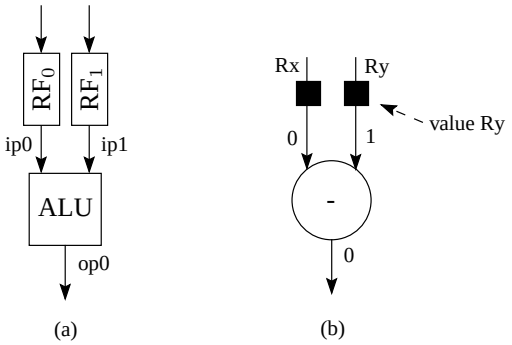


Figure 3.11: Functional unit with input and output ports (a) and an operation with input and ouput ports (b).

isomorphism problem. First some observations about the our operation assignment problem are made which make the proof easier to understand.

The first observation is that an operation input port label and operation output port label is associated with respectively every head and tail of a data edge in the DFG. Also the ports of the functional units in our target VLIW architecture are labelled. These ports are introduced and labelled because the input ports of a functional unit do not need to be equivalent as is for example the case if the functional unit can perform a mathematical operation which is not commutative. Another reason is that the used functional unit input port determines in which register file the input value will be stored. For example if the minus operation in Figure 3.11b is executed on the function unit in Figure 3.11a then the result depends on the assignment of the operation input ports to the functional unit input ports. Also the register file binding of the values Rx and Ry depend on this port assignment. However for architectures in which all values that are input values of a functional unit are fetched from the same register file then the port assignment can be adapted as required after the assignment of all the operations in the DFG to the functional units.

The data path, shown in Figure 3.12 can be represented as a graph shown in Figure 3.13 in which vertices represent functional units and every edge a communication path from an output port of a functional unit through the network and a register file to an input port of a functional unit. Labels are associated with the head and tail of these edges which indicate respectively the input and output ports of the functional unit.

All the communication paths between the functional units in a time interval can be captured in a similar graph. The graph in Figure 3.14 captures all communication paths between the functional units given a time interval of 2 clock cycles.

The operations and edges of the DFG in Figure 3.15 can be mapped to the vertices and edges of a subgraph of the graph in Figure 3.14. This mapping must be such that an operation input port label x of an edge in the DFG corresponds to the label ip$x$ of an edge in the graph of Figure 3.14. Such a mapping is shown in Figure 3.16. This mapping correspond to a valid assignment of the operations in the DFG, given a latency constraint of the schedule of

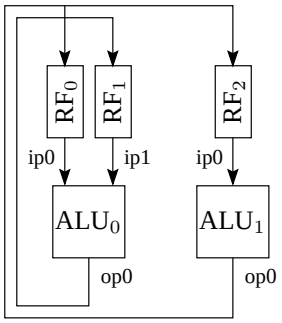


Figure 3.12: An example of a data path.

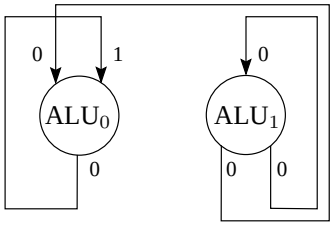


Figure 3.13: Graph representation of the communication paths in the data path of Figure 3.12.

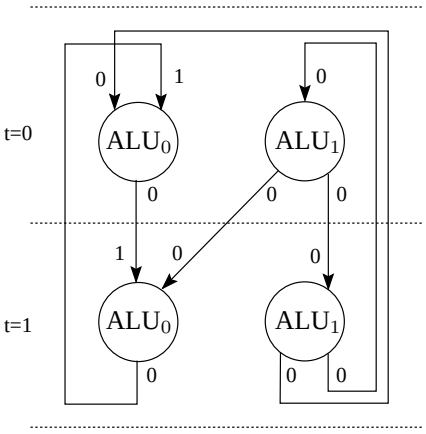


Figure 3.14: Graph representation of the available communication paths during 2 clock cycles in the data path of Figure 3.12.

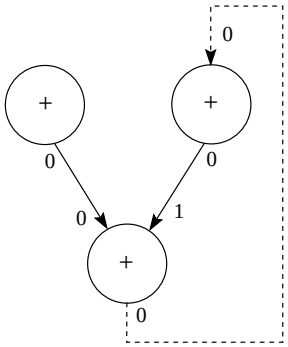


Figure 3.15: DFG which is a subgraph of the graph in Figure 3.14.

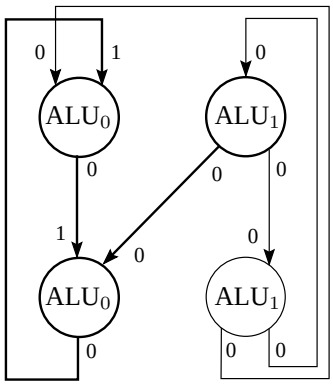


Figure 3.16: Graph representation of the available communication paths during 2 clock cycles. The fat lines indicate a subgraph which is equal to the DFG in Figure 3.15.

2 cycles. This mapping respects the constraints which are a result of the incomplete network. Also the resource constraints are respected because the mapping corresponds to the situation that maximal one operation per clock cycles is executed on a functional unit.

Also the operations and edges of the DFG in Figure 3.17 can be mapped to the vertices and edges of the graph in Figure 3.18. In this case the DFG is folded given an initiation interval of 1 clock cycle.

In the proof of Theorem 3.1 an algorithm is applied that runs in polynomial-time and computes a function f. This function f maps every instance i of the subgraph isomorphism problem (π) to an instance i' of the operation assignment decision problem (π'). This algorithm is hopefully easier to understand given the observations made in the previous paragraphs.

Theorem 3.1
The problem of deriving an assignment of operations in a DFG to functional units such that

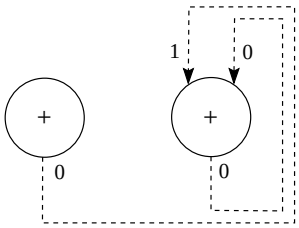


Figure 3.17: DFG with only loop-carried data dependencies which is a subgraph of the graph in Figure 3.14.

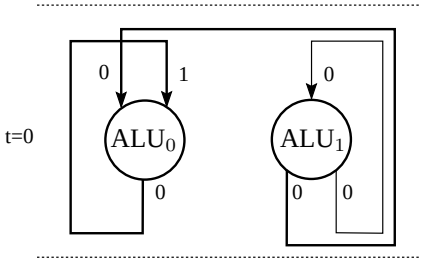


Figure 3.18: Graph representation of the available communication paths during 1 clock cycles. The fat lines indicate a subgraph which is equal to the folded version of the DFG in Figure 3.17.

the required communication paths are available in the data path is $\mathcal{NP}$-hard, even if all functional units in the data path are equivalent.

Proof:

For an given assignment of operations to functional units we can verify in polynomial time whether the required communication paths between the functional units can be made in the data path. Hence, the problem is in $\mathcal{NP}$.

We will describe a polynomial time algorithm that computes a function f that maps instances i of the subgraph isomorphism problem π in instances i' of the decision variant of the operation assignment problem π'. That the function f exists proves that the decision variant of the operation assignment is $\mathcal{NP}$-complete because the subgraph isomorphism decision problem belongs to the class of $\mathcal{NP}$-complete problems [GJ79]. Therefore the operation assignment search problem is $\mathcal{NP}$-hard.

Let the directed graph $G'(V'_g, E'_g)$ representing a DFG instance and the graph $H'(V'_h, E'_h)$ an architecture instance of the problem π'. Let also the directed graph $G(V_g, E_g)$ and the directed graph a $H(V_h, E_h)$ instances of the problem π.

The algorithm that computes the function f is defined as follows. Let $V'_h = V_h$ and $E'_h = E_h$. Let all the edges $e'_h \in E'_h$ be loop carried data dependencies. With every tail of an edge $e'_h \in E'_h$ the output port op0 is associated. With every head of an edge $e'_h \in E'_h$ incident to a node $v'_h \in V'_h$ an unique input port ip_x with a consecutive numbering starting from 0 is associated. All vertices $v'_h \in V'_h$ correspond to operations of the same type. Similar procedure is applied for the graph G_h, i.e., $V'_g = V_g$ and $E'_g = E_g$, Also with every tail of an edge $e'_g \in E'_g$ the output port mumber 0 is associated and with every head of an edge $e'_g \in E'_g$ incident to a node $v'_g \in V'_g$ an unique input port with number x is associated. The input ports v'_g are consecutive numbered starting from 0. All vertices $v'_g \in V'_g$ correspond to a functional unit which can execute operation of an arbitrary type.

The algorithm that computes the function f belongs to the class of polynomial time algorithms because this algorithm adds $|E_g|$ functional unit output port labels and adds $|E_h|$ operation output port labels. There are also $|E_g|$ functional unit input port labels and $|E_h|$ operation input port labels added.

Now we prove that if instance i of π is a 'yes' instance then i' is a 'yes' instance. With the function f a valid instance of a DFG and a data path can be derived from i. This data path graph corresponds to instances of our template in which all input operands are read from the same register file via register read ports which are not shared. This register file has the same number of write ports as read port such that no resource conflicts occur. The register files in the data path also have a sufficient number of registers. Also for every $e_g \in E_g$ there is a conflict free communication path in the processor. Now H' is subgraph in G' because adding labels to the edges in H and G does not change the topology of H and G. This H' subgraph in G' correspond to a valid operation assignment given a timing constraint that specifies an initiation interval of 1 cycle.

Now we prove if instance i' of π' is a 'yes' instance then i of π is a 'yes' instance. If i' is 'yes' instance then the H' is a subgraph in G' and G' corresponds to a data path in which all

input operands are read from the same register file. If we drop all input port and output port labels from the edges the we obtain the graph H and the graph G. The graph H is a subgraph of G because the removal of labels does not change the topology of a graph. □

Chapter 4

Assignment Search-Space Representation

All combinations in which operations can be assigned to functional units and the constraints due to an incomplete communication network and resource conflicts are modelled in an assignment search-space representation. Some combinations of operation assignments are not feasible as a consequence of the incomplete network in the processor or as a consequence of resource conflicts. During the assignment decision process some of the infeasible operation assignment options are removed from this search-space by pruning algorithms.

This chapter describes the operation assignment search-space representation. The outline of this section is as follows. First a conflict graph and the conflict graph coloring problem is defined in Section 4.1. Then in Section 4.2 it is described how an incomplete network can be modeled in such a conflict graph. In Section 4.3 the Assignment Conflict Graph (ASCG) is introduced in which all operation assignment options and the constraints imposed by the network are modelled. How the number of nodes and edges in the ASCG can be reduced, is described in Section 4.3.4

4.1 Conflict graph concepts

In this section we define a conflict graph, an annotated conflict graph and the coloring of these graphs. Subsequent sections describe the modeling of the assignment search-space in an annotated conflict graph.

A conflict graph is defined as follows:

Definition 4.1 (Conflict Graph.)
A Conflict Graph (CG) is an undirected graph represented by a tuple (V, E), where

- V is the set of vertices
- $E \subseteq V \times V$ is the set of edges. An edge $e \in E$ indicates a conflict.

Colors are assigned to the vertices during coloring of a conflict graph. Every vertex is colored with only one color during coloring. A *proper coloring* [Gri94] of CG occurs when we color the vertices of CG such the vertices adjacent to the edge (a, b) are colored with different colors. The *chromatic number* $\chi(CG)$ is the minimum number of colors needed to properly color CG. An *exact coloring* is a proper coloring that uses $\chi(CG)$ colors.

In the assignment search-space model, a conflict graph is used in which a color out of a restricted set of colors can be assigned to a node. We have given such a conflict graph the name "annotated conflict graph". The annotated conflict graph is defined as follows:

Definition 4.2 (Annotated Conflict Graph.)
An Annotated Conflict Graph (ACG) is an undirected graph represented by a tuple (V, E, Z), where

- V is the set of vertices
- $E \subseteq V \times V$ is the set of edges.
- Z is the set of colors. To every vertex $v \in V$ a set of colors is associated: colors$(v) \subseteq Z$. Such a set is given the name *Color Set*.

The annotated conflict graph can be transformed into an ordinary conflict graph with the same properties. This is illustrated with an example in Figure 4.1. As this figure shows, after conversion a graph is obtained in which a clique of nodes is added to the original conflict graph. The size of the clique is the cardinality of the set Z. Edges incident to the nodes in this clique impose the same restrictions as the color sets in the annotated conflict graph if this graph is colored with $|Z|$ colors. Because an annotated conflict graph can be transformed in an ordinary conflict graph, it can be seen as a short hand notation of the conflict graph obtained after transformation.

The conflict graph of Figure 4.1b models the same restrictions as the annotated conflict graph in Figure 4.1a. This can be seen as follows: Assume that node c0, c1 and c2 in Figure 4.1b are colored with respectively color 0, 1 and 2. The edge in this figure between node c1 and n2 guarantees that n2 cannot be colored with color 1. This edge is introduced to model that color 1 is a color that is not present in the color set of node n2 in Figure 4.1a. The edge between node n0 and node c2 and the edge between node n1 and node c2 in Figure 4.1b models in

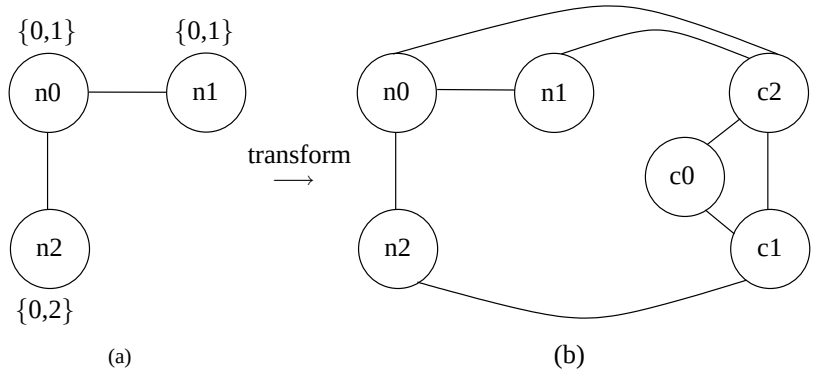


Figure 4.1: Transformation of an annotated conflict graph (a) into a ordinary conflict graph (b).

a similar way that color 2 is not present in the color sets of the node n0 and n1. Therefore given that node c0, c1 and c2 in Figure 4.1b are colored with respectively 0, 1 and 2 then the same combination of colors can be assigned to node n1, n2 and n3 in Figure 4.1b as it is the case for the nodes n1, n2, n3 in Figure 4.1a. In the case that other colors then 0, 1 and 2 are assigned by the coloring algorithm to respectively node c0, c1, c2 then after coloring the colors can be renamed such that the desired color assignment of these nodes is obtained. After renaming of the colors, also a correct coloring of the other nodes is obtained.

The graph K-colorability decision problem [GJ79] answers the question wheter a conflict graph can be colored with maximal K colors, where $K \leq |V|$ and $K \geq 3$. This decision problem is NP-complete and therefore no algorithm is known with a polynomial time complexity that derives a proper coloring for every possible conflict graph. In other words, coloring of a conflict graph can be computational intensive as a consequence of the exponential increase of the complexity of the coloring problem with the size of the conflict graph. Coloring of the graph may therefore take a long time. However Coudert [Cou97] found that for many practical conflict graph instances, an exact coloring can be computed in a reasonable amount of time. For the moment the assumption is made that the conflict graph instances considered during assignment have similar properties as the instances that where considered by Coudert and can therefore be colored in a reasonable amount of time.

4.2 Network model

All communication paths in the data path of a processor can be captured in a network model, as is described in Section 2.1.3. In Figure 2.6 of that section a data path instance is shown along with the corresponding network model. In this network model every output of a sub-network is connected to only one input of another network. The selection of an input port of a sub-network determines the network output ports through which a result can be routed. The other way around is also true, that is, the selection of an output port of a sub-network

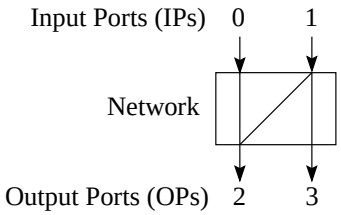


Figure 4.2: Network without a connection between input port 0 and output port 3.

through which a result must be routed determines through which ports the input data can be provided. These mutual dependencies between input ports and output ports can be modeled in a conflict graph.

Figure 4.2 depicts a network in which there is no connection between input port 0 and output port 3. If output port 3 is selected then input port 0 can not serve as input port. This restriction is modeled in the conflict graph shown in Figure 4.3. The IP and OP node in this graph are associated with respectively the input ports and the output ports of the network. The numbers in the sets next to a node indicate the colors that can be assigned to that node. These colors correspond one to one to the network ports. The third node in the graph, which is labeled with CN, enforces the restriction that if the IP node is colored with color 0 that then the OP node cannot be colored with color 3. All other combinations of colors can be assigned by a coloring algorithm. The restriction on the colors that can be assigned to the IP and OP node is enforced by the CN node in the following way. In the case, the IP node is given color 0 then the edge between the IP and the CN node enforces that color 0 cannot be assigned to the CN node. Because only color 0 and 3 can be assigned to the CN node, the CN node must be colored with 3. If the CN node is colored with 3 then the edge between the CN node and the OP node enforces that color 2 is the only color that can be assigned to the OP node. In a similar way the assignment of color 3 to the OP node has as consequence that, color 0 cannot be assigned to the IP node. The CN nodes can be used in a similar way to model in a conflict graph restrictions imposed by networks with an arbitrary interconnection pattern between input ports and output ports. A CN node is needed for every combination of an input port and an output port for which there is no connection in the network. Therefore, the worst-case number of CN nodes in the conflict graph per network equals #IP$\times$#OP where #IP and #OP are respectively the number of input ports and output ports of the network.

Formally a network and the ACG which capture the routing constraints are defined as:

Definition 4.3 (Network)
A network can be represented by a tuple (IP, OP, k), where

- IP is the set representing the network input ports,
- OP is the set representing the network output ports,
- $k : IP \times OP \rightarrow \{true, false\}$ is a function. It defines the connections that can be

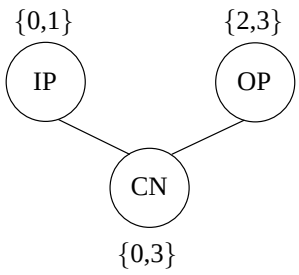


Figure 4.3: Conflict graph in which the restrictions of the network of Figure 4.2 are modeled.

made between input and output ports. If ip $\in IP$ can be connected with op $\in OP$ then $k(\text{ip}, \text{op})$ is true. Otherwise $k(\text{ip}, \text{op})$ is false.

Definition 4.4 (network ACG)
A network Annotated Conflict Graph (network-ACG) can be represented by a tuple $(v_{IP}, v_{OP}, V_{CN}, E_{CN})$, where

- v_{IP} is a vertex representing the set of network input ports IP. With the vertex v_{IP} a set colors(v_{IP}) is associated. Every color in the set colors(v_{IP}) corresponds one to one to an input port ip $\in IP$,

- v_{OP} is a vertex representing the set of network output ports OP. With vertex v_{OP} a set colors(v_{OP}) is associated. Every color in the set colors(v_{OP}) corresponds one to one to an output port op $\in OP$. The colors in the set colors(v_{OP}) are different from the colors in the set colors(v_{IP}), i.e., colors$(v_{OP}) \cap$ colors$(v_{IP}) = \emptyset$,

- V_{CN} is a set of vertices. There is a vertex $v_{cn} \in V_{CN}$ for every input port ip $\in IP$ and output port op $\in OP$ pair for which $k(\text{ip}, \text{op})$ is false. A set colors(v_{CN}) is associated with every $v_{cn} \in V_{CN}$ that contains two colors. One color correspond to the input port ip and one to the output port op,

- E_{CN} is a set of undirected edges. There is an edge $e_1 \in E_{CN}$ with $e_1 = (v_{IP}, v_{CN})$ and an edge $e_2 \in E_{CN}$ with $e_2 = (v_{OP}, v_{CN})$ for every vertex $v_{CN} \in V_{CN}$

The network model of the data path, which is described in Section 2.1.3, consist of 6 non-blocking sub networks. Each sub network is represented by a network ACG. The network ACG of a sub network with for example write ports (WP) as input ports and read ports (RP) as output ports is denoted in the next sections by G_{WPRP}. The function that defines the connections that can be made between input and output ports of this sub network is denoted by $k_{WPRP}(\text{ip}, \text{op})$.

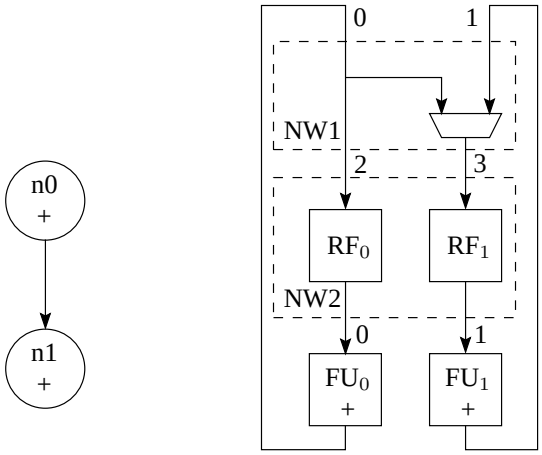


Figure 4.4: A DFG example (a) and a data path instance (b) to illustrate modeling of the assignment search-space in a conflict graph.

4.3 Assignment search-space model

In Section 4.3.1 it is described how the constraints imposed by the incomplete networks in the data path together with the data dependencies of a DFG can be modeled in a so called ASsignment Conflict Graph (ASCG). Section 4.3.2 describes how the usage of resources can be modeled in the same ASCG. The ASCG is formally defined in Section 4.3.3.

4.3.1 Modeling of interconnect constraints

The assignment search-space for the DFG and data path shown in Figure 4.4 is modeled in the conflict graph of Figure 4.5. This conflict graph is called the ASsignment Conflict Graph (ASCG). The nodes $n0$ and $n1$ in the ASCG correspond to operations $n0$ and $n1$ in the DFG. The numbers in the sets depicted above these nodes correspond to the output ports of network NW2 in Figure 4.4. Because an output port of NW2 is connected to only one functional unit, these numbers correspond one to one to the functional units FU_0 and FU_1. The data produced by operation $n0$ should, according to the data edge in the DFG, be consumed by operation $n1$. A value produced by the functional unit which executes $n0$ should pass network NW1 to reach one of the write ports 2 or 3 of register files RF_0 and RF_1. In the conflict graph there is a node labeled "WP" that corresponds with these write ports. The CN nodes in the conflict graph model the inhibited connections in the networks. The CN nodes that model the inhibited connections of one network are surrounded by a dashed box.

An important property of the ASCG is that a valid coloring of this graph exists, if and only if there exists an assignment that satisfies the constraints imposed by the interconnect in the data path. From a valid coloring of the graph, the assignment of the operations can be derived.

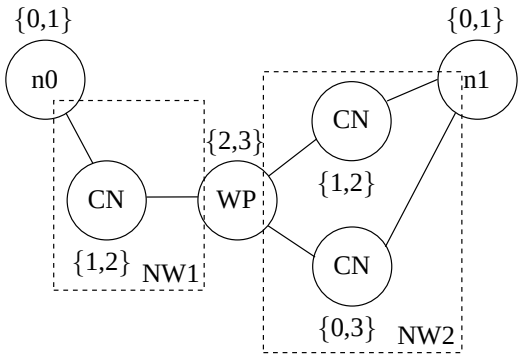


Figure 4.5: Assignment conflict graph in which the communication paths in the data path of the target processor are modeled.

A color assigned to a node, except for the CN nodes, corresponds to the usage of a specific data path resource.

4.3.2 Modeling of hard resource conflicts

A conservative estimate of the distances between every pair $< v_i, v_j >$ of operations in the DFG is calculated by the timing constraint analysis techniques. Two operations are executed in the same cycle if the distance between these operations satisfies equation 4.1. If these operations have the same type and are executed in the same cycle then they are not allowed to use the same resource. This can be modeled with an edge between the nodes in the ASCG that correspond to operation v_i and v_j. This edge assures that these nodes will obtain a different color when the graph is colored. The different colors correspond to an assignment of operations to different resources.

$$d(v_i, v_j) = -d(v_j, v_i) \quad and \quad d(v_i, v_j)\ mod\ \mathrm{II} = 0 \tag{4.1}$$

In the case the DFG in Figure 4.4 is the body of a loop then $n0$ and $n1$ are executed in the same cycle if the initiation interval is 1 cycle. If II=1 then the operations that are executed in the same cycle are from two successive iterations of the loop. Because they are executed in the same cycle an edge is added in the ASCG between node $n0$ and $n1$ as shown in Figure 4.6. In the only feasible coloring of this graph the colors 0, 1 and 3 are respecitively assigned to node $n0$, $n1$ and WP in the ASCG. This correspond to the assignment of operation $n0$ and $n1$ of the DFG to respectively functional unit FU_0 and FU_1 and the usage of WP number 3.

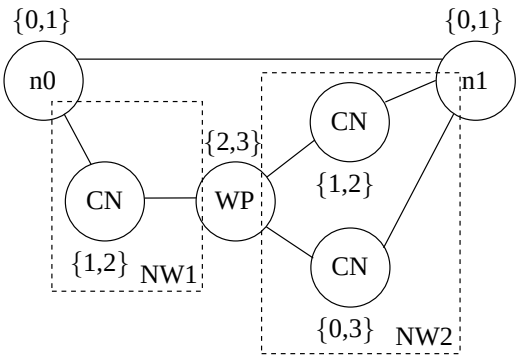


Figure 4.6: Assignment conflict graph in which a resource conflict is modeled.

4.3.3 The assignment conflict graph

In the previous paragraph, a description is given how interconnect and resource constraints can be modeled in the ASCG. It is now time to give a formal definition of the assignment conflict graph. The tuple (V_a, E_a) defines the ASCG. This graph is defined given a DFG $(V, E_d \cup E_l \cup E_s, w_e, \text{id}, t_i, t_o)$ with the set of data edges $E_{de} = E_d \cup E_l$.

Definition 4.5 (Immediate predecessors)
pred$(e) = \{v \mid e = (v, w)\}$

Definition 4.6 (Immediate successors)
succ$(e) = \{w \mid e = (v, w)\}$

Definition 4.7 (mapping of DFG operation on functional units vertex)
$FU : V \rightarrow V_a$ is the function $FU(v) = v_{FU}$ that gives for each DFG operation v the one to one corresponding vertex v_{FU}.

Definition 4.8 (mapping a data edge on a output port vertex)
$OP : E_{de} \rightarrow V_a$ is the function $OP(e) = v_{OP}$ gives for each data edge e a vertex v_{OP}. There is a set colors(v_{OP}) associated with v_{OP}. Every color in the set corresponds one to one to a functional unit output port in the data path that can be used to transport the result.

Definition 4.9 (mapping a data edge on a write back port vertex)
$WB : E_{de} \rightarrow V_a$ is the function $WB(e) = v_{WB}$ that gives for each data edge e the one to one corresponding vertex v_{WB}. There is a set $colors(v_{WB})$ associated with v_{WB}. Every color in this set corresponds one to one to a write back bus in the data path. Every write back bus is represented by a color.

Definition 4.10 (mapping a data edge on a write port vertex)
$WP : E_{de} \rightarrow V_a$ is the function $WP(e) = v_{WP}$ that gives for each data edge e the one to one corresponding vertex v_{WP}. There is a set $colors(v_{WP})$ associated with v_{WP}. Every color in this set corresponds one to one to a write port in the data path. Every write port is represented by a color.

Definition 4.11 (mapping a data edge on a read port vertex)
$RP : E_{de} \rightarrow V_a$ is the function $RP(e) = v_{RP}$ that gives for each data edge e the one to one corresponding vertex v_{RP}. There is a set $colors(v_{RP})$ associated with v_{RP}. Every color in this set corresponds one to one to a read port in the data path. Every read port is represented by a color.

Definition 4.12 (mapping a data edge on a write port vertex)
$IP : E_{de} \rightarrow V_a$ is the function $IP(e) = v_{IP}$ that gives for each data edge e a vertex v_{IP}. There is a set $colors(v_{IP})$ associated with v_{IP}. Every color in the set corresponds one to one to a functional unit input port in the data path that can be used to transport the result.

Definition 4.13 (Network port type)
$TYPE(v) =$ type gives the port type for each ASCG vertex v. The port types of the network model of the data path are defined in Section 2.1.3. Examples of a port types are read port and functional unit input port.

Definition 4.14 (Operation)
$OPR : V_a \rightarrow V$ is the function $OPR(v_a) = v$ that gives for an ASCG vertex v_a the corresponding operation v.

Definition 4.15 (Subgraph inclusion function)
$\chi(v_I, v_O, G_{NW}) = (V_{CN1}, E_{CN1})$ with $G_{NW} = \{V_{IP}, V_{OP}, V_{CN}, E_{CN}\}$ where
$E_{CN1} = \{(v_I, v_{CN}), (v_{CN}, v_O) \mid \exists v_{CN}\ (v_{IP}, v_{CN}) \in E_{CN} \wedge (v_{CN}, v_{OP}) \in E_{CN}\}$,
$V_{CN1} = V_{CN}$.

Definition 4.16 (Data edge to ASCG function)
$\chi' : E_{de} \rightarrow V_a \times V_a \times E_a$ is the function $\chi'(e) = (V1, V_{CN}, E)$ where
$n1 =$ pred(e)
$v_{FU1} = FU(n1)$
$v_{OP} = OP(e)$
$v_{WB} = WB(e)$
$v_{WP} = WP(e)$
$v_{RP} = RP(e)$
$v_{IP} = IP(e)$
$n2 =$ succ(e)
$v_{FU2} = FU(n2)$
$(V_{CN1}, E1) = \chi(v_{FU1}, v_{OP}, G_{FUOP})$
$(V_{CN2}, E2) = \chi(v_{OP}, v_{WB}, G_{OPWB})$
$(V_{CN3}, E3) = \chi(v_{WB}, v_{WP}, G_{WBWP})$
$(V_{CN4}, E4) = \chi(v_{WP}, v_{RP}, G_{WPRP})$
$(V_{CN5}, E5) = \chi(v_{RP}, v_{IP}, G_{RPIP})$
$(V_{CN6}, E6) = \chi(v_{IP}, v_{FU2}, G_{IPFU})$

$V_{CN} = V_{CN1} \cup V_{CN2} \cup V_{CN3} \cup V_{CN4} \cup V_{CN5} \cup V_{CN6}$
$V1 = v_{FU1} \cup v_{OP} \cup v_{WB} \cup v_{WP} \cup v_{RP} \cup v_{IP} \cup v_{FU2}$
$E = E1 \cup E2 \cup E3 \cup E4 \cup E5 \cup E6$

Definition 4.17 (ASCG)
The assignment conflict graph for the DFG and data path is:
$\chi''(E_{de}) = (V_a, E_a)$ where
$V1 = \{V1 \mid e \in E_{de}, (V1, C1, E1) = \chi'(e)\}$
$V2 = \{C1 \mid e \in E_{de}, (V1, C1, E1) = \chi'(e)\}$
$V_a = V1 \cup V2$
$E1 = \{E1 \mid e \in E_{de}, (V1, C1, E1) = \chi'(e)\}$
$E2 = \{(v1, v2) \mid v1 \in V1, v2 \in V1, v1 \neq v2 \ \wedge \ TYPE(v1) = TYPE(v2) \ \wedge$
$d(OPR(v1), OPR(v2)) = -d(OPR(v1), OPR(v2)) \ \wedge$
$d(OPR(v1), OPR(v2)) \ mod \ \mathrm{II} = 0\}$. (see equation 4.1).
$E_a = E1 \cup E2$

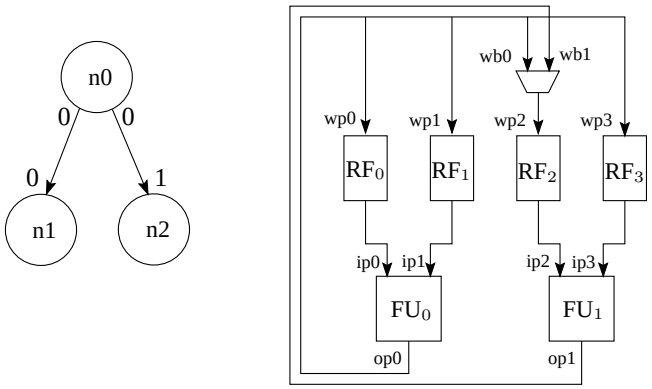
An example of an ASCG which is constructed according definition 4.17 for the data path and DFG in Figure 4.7 in the case that there aren't any hard resource conflicts (condition expressed by equation 4.1 is not satisfied) is shown in Figure 4.9. The network model of the data path is shown in Figure 4.8. The nodes labelled with "OP" and "WB" in these ASCG correspond respectively with the functional unit output port and a write-back bus via which the result of operation n0 is conveyed. The node "IP" at the left side in Figure 4.9, corresponds to the functional unit input port number 0 via which the result is consumed by operation n1. The ports of the functional units are numbered from left to right in increasing order. Therefore input port number 0 corresponds to the ports which are labelled with ip0 and ip2 in the network model. The node "IP" at the right side in Figure 4.9 corresponds to the functional unit input port number 1 via which the result is consumed by operation n1 as is specified in the DFG. Input port number 1 of the functional units is labelled with ip1 and ip3 in the network model.

This ASCG has the following useful property:

Theorem 4.1
An assignment of operations to resources which respects the interconnect constraints of the data path corresponds to a feasible coloring of the ASCG and vice versa.

Proof:

Given the construction of an ASCG which complies definition 4.17 then every feasible assignment of operations to functional units which respects the interconnect constraints of the data path can be represented with colors that are assigned to ASCG nodes. Because the interconnect constraints are respected, one of the two colors in the color set of a CN node is not assigned to one of the two adjacent nodes. This color can be assigned to the CN node. The colors assigned to the nodes in the ASCG is a feasible coloring because every adjacent node is assigned a different color.



(a) (b)

Figure 4.7: A data flow graph (a) and a data path (b) for which the ASCG is shown in Figure 4.9.

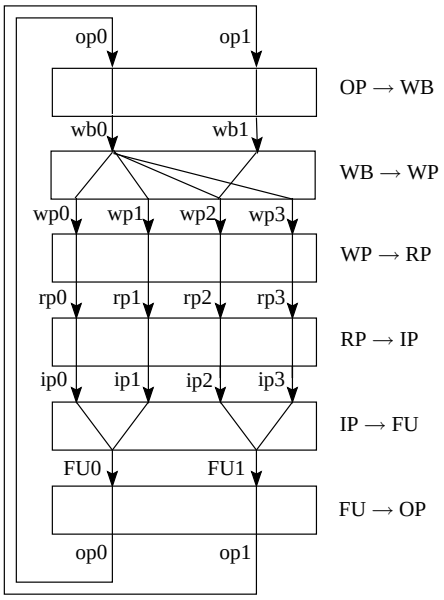


Figure 4.8: Network model of the data path shown in Figure 4.7b.

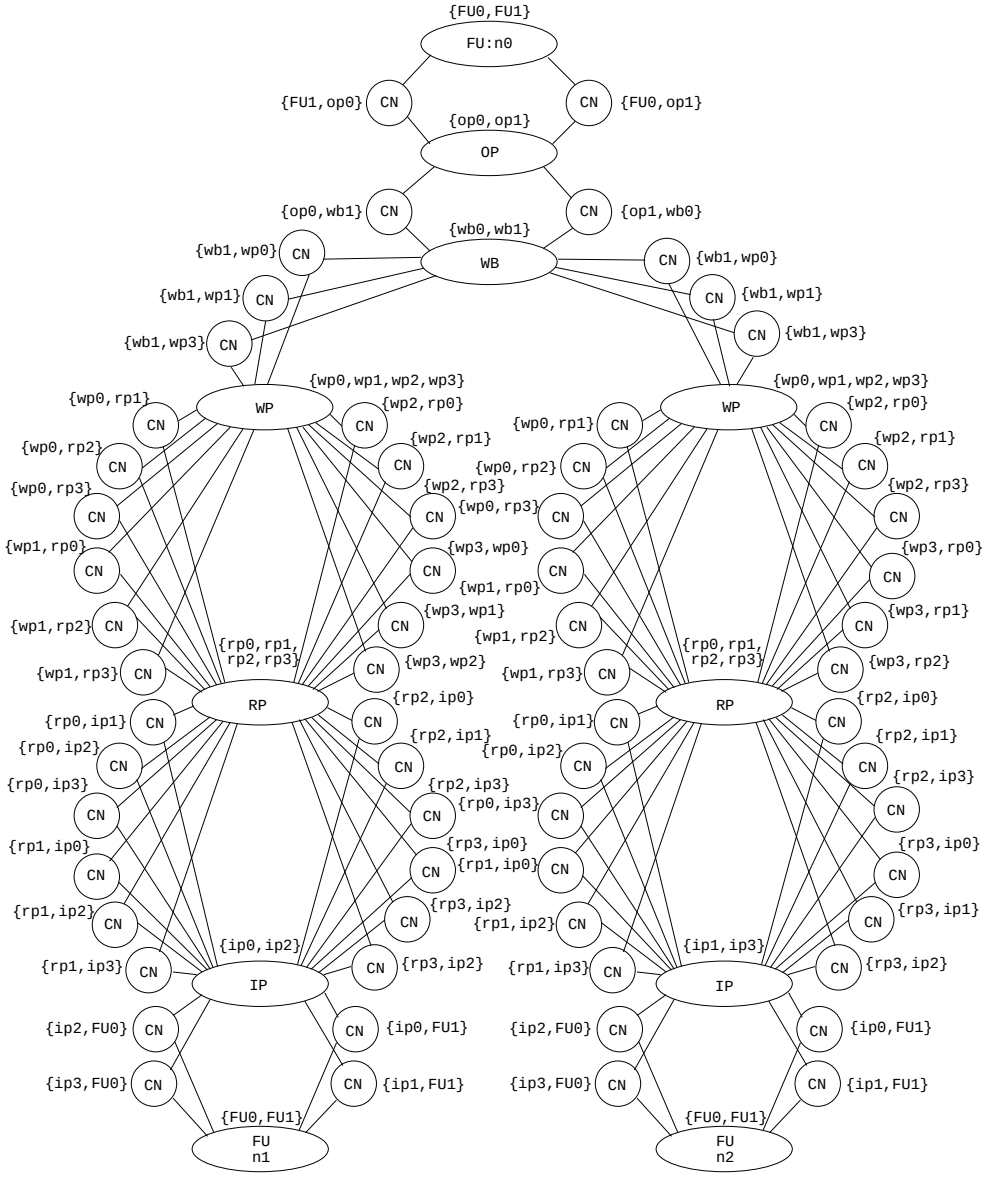


Figure 4.9: Assignment conflict graph for the DFG and data path in Figure 4.7.

But also the other way around is true, that is, every feasible coloring of the ASCG represents an assignment of operations and ports to functional units and network ports. The required connections can be made in the data path because the CN nodes exclude combinations of colors that correspond to ports of a network that cannot be connected. □

If the time assignment of the all operations is known then all the resource conflicts can be modeled with edges in the ASCG. This ASCG has the following property:

Theorem 4.2
Given the time assignment of all operations in the DFG then a coloring of the ASCG corresponds to a valid assignment of operations to resources which respects the resource and interconnect constraints of the data path.

Proof:

In the case the time assignment is known then it is also known which pairs of operations are scheduled in the same clock cycle. Between the pair of vertices in the ASCG that correspond to this pair of operations there is an edge. This edge prevents that these vertices will be colored with the same color. Different colors correspond to the assignment to different resources. At the same time the interconnect constraints are also respected because Theorem 4.1 still holds. □

4.3.4 Redundancy in the assignment conflict graph

A disadvantage of the modeling a network in the way as described in Section 4.3.1 is that the number of nodes and edges in the assignment conflict graph rapidly grows with the number of date edges in the DFG and the number of functional units in the processor. The worst case growing factor of the number of nodes in the conflict graph is $O(\#\text{IPs} \times \#\text{OPs} \times |E_{de}|)$ with $\#$IPs and $\#$OPs the largest number of input and output ports of a network in the data path and $|E_{de}| = |E_d \cup E_l|$ the number of data edges in the DFG. The size of the conflict graph determines to a large extend the memory usage during compilation. The number of nodes and edges of the conflict graph also affect the time it takes to prune the colors in the conflict graph. In practice this has as a consequence that the excessive long compilation time makes it impractical to handle DFGs with more than 100 nodes and target processors with more than 20 functional units. In order to improve the compilation time it is desirable to reduce the size of the assignment conflict graph. A reduction can be achieved by removing nodes and edges that are redundant.

If an assignment conflict graph is constructed according definition 4.17 then every connection that cannot been made between a network input port and a network output port is modeled with a CN node and two edges in this graph. This CN node and these edges are redundant if the network input or output port is excluded by other constraints. For example, according to the DFG in Figure 4.7a consumes operation $n1$ a result via input port 0. From the interconnect in the data path of Figure 4.7b it follows directly that this result can never be transported via the ports ip1, ip3, rp1, rp3. Therefore these ports can be removed from the sets of the nodes “RP” and “WP” at the left side in Figure 4.9. Now the nodes labelled with CN that could disable one of these ports are redundant. The ASCG after removal of redundant CN nodes and edges is shown in Figure 4.10.

An alternative representation for the assignment conflict graph is a constraint graph representation which is presented in Appendix A. We have put this description in an appendix because it was discovered when this thesis was almost completed and the use of this model influences

the results. It is likely that the pruning algorithms that operate on a constraint graph have a lower run-time because the constraint graph is typically a more compact representation of the assignment search space than a conflict graph. Because, additional information is explicit in the constraint graph also the number infeasible assignment options that is detected will be different, which results also in a lower run-time.

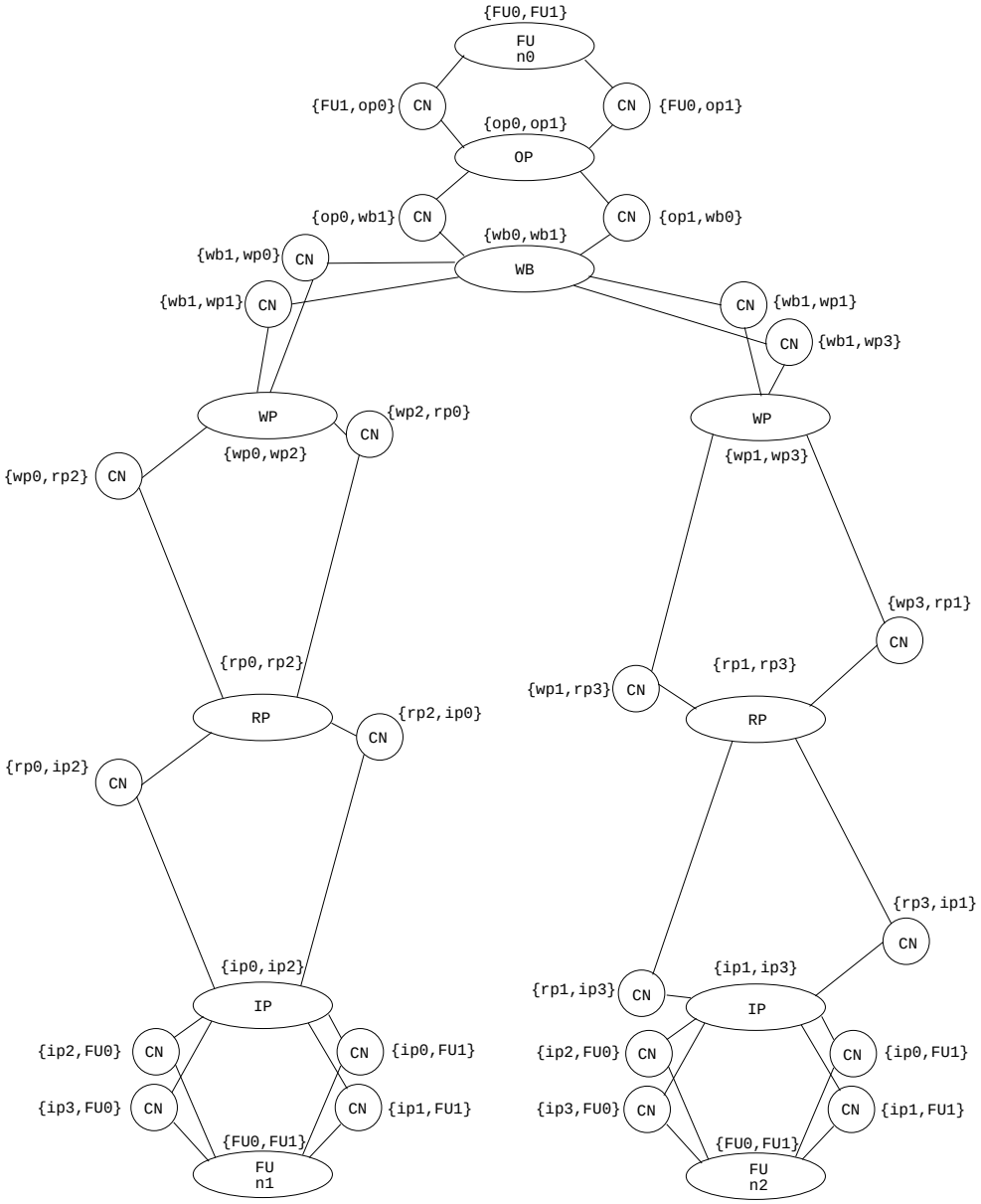


Figure 4.10: Assignment conflict graph after removal of unreachable ports and redundant CN nodes.

Chapter 5

Assignment Search-Space Pruning

Infeasible operation assignment options are removed from the assignment search-space by pruning algorithms. These algorithms must have a low computational complexity because pruning is repeated after every assignment decision. Therefore, the pruning algorithms remove only infeasible assignment options that can be easily identified. In this chapter, two assignment search-space pruning algorithms are presented. The first pruning algorithm, which is described in Section 5.1, is based on the fact that different colors must be assigned to nodes of a clique in the assignment conflict graph. The other pruning algorithm derives infeasible operation assignment options by explicitly taking the connections in the network of the processor into account. This algorithm is described in Section 5.2. At the end of this section the guarantee that can be given by this algorithm is stated.

5.1 Pruning of colors of nodes in a clique.

A different color must be assigned to every node in a clique in a conflict graph. The colors that can not be assigned to a certain node because they must be assigned to adjacent nodes can be removed. A bipartite graph is used to derive the colors that can be removed. This graph has been given the name "bipartite clique pruning graph" and is defined as follows:

Definition 5.1 (Bipartite Clique Pruning Graph.)
Consider a clique of nodes $V \subseteq V_a$ in the ASCG. Then the Bipartite Clique Pruning Graph (BCPG) is represented by the tuple (N,A), where:

- $N = W \cup R$ is the set of nodes with the node $w \in W$ corresponding one to one to a node $v \in V_a$ in the clique in the ASCG. The node $r \in R$ corresponds one to one to a color in the union of the color sets ($\cup_{v \in V_a} \text{colors}(v)$) of the nodes in the clique.

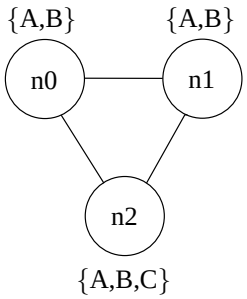


Figure 5.1: A clique of the ASCG used to illustrate pruning of colors with a Bipartite Clique Pruning Graph (BCPG). With the BCPG it can be derived that color A and B can be removed from the color set of $n2$.

- $A = W \times R$ is the set of undirected edges; there is an edge $(w, r) \in A$ if the color represented by the node $r \in R$ is in the color set of the ASCG node represented by the node $w \in W$.

Assume that the edge (w, r) does not belong to the set of irreducible components of this bipartite graph. In this case corresponds the node r to a color that can not be assigned to node w because there is a one to one correspondence between the set of proper colorings and the set of complete matchings. This color can be removed from the allowed color set of the node w in the annotated conflict graph. This pruning method is given the name "Clique-Pruning". The computational intensive part of this method is the derivation of the irreducible components. The irreducible components can be derived in $O(\mid N \mid^{1/2} \cdot \mid A \mid)$ [SV76].

In Figure 5.1 a clique and in Figure 5.2 the corresponding Bipartite Clique Pruning Graph is shown. The solid lines in the bipartite graph correspond to edges that belong to the set of irreducible components of this graph. The dashed lines correspond to the edges that do not belong to the set of irreducible components. These dashed edges indicate that color A and B can not be assigned to node n2. These colors can therefore be removed from the color set of node n2.

The pruning is likely to be more successful for larger cliques. However deriving a covering of the edges in a graph with a minimal number of cliques is the so called "covering by cliques" problem which is NP-hard [GJ79]. Therefore, the pruning algorithm is only applied for cliques that can easily be identified.

Cliques with 2 nodes are easily identified because they correspond to adjacent nodes in the conflict graph. For adjacent nodes the above described pruning algorithm reduces to rule 2, which is stated as follows:

Rule 2 Let u and v be adjacent nodes in a ASCG and c be the only color in the color set of node u then c can be removed from the color set of node v.

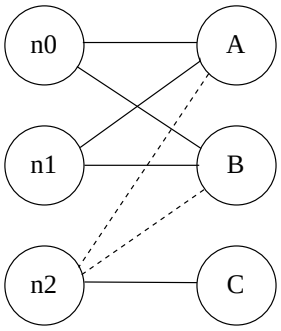


Figure 5.2: A Bipartite Clique Pruning Graph (BCPG) used to prune the ASCG in Figure 5.1.

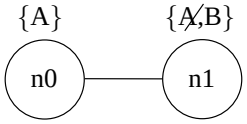
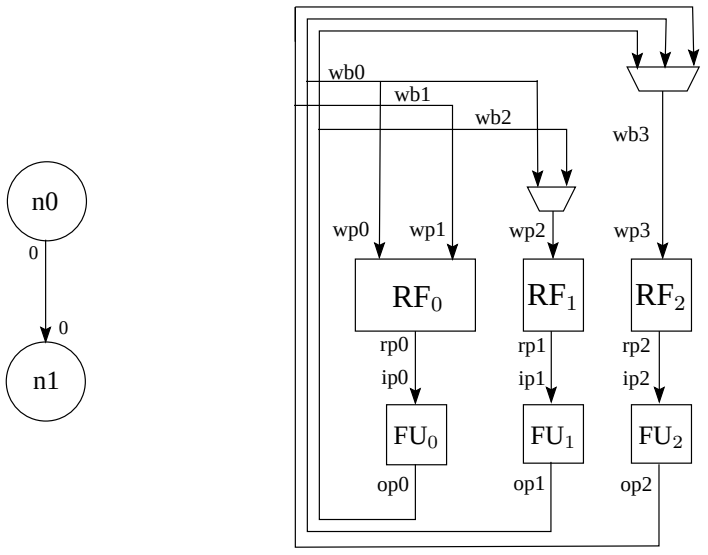


Figure 5.3: Pruning of colors of adjacent nodes.

An example of an assignment conflict graph that illustrates pruning with rule 2 is shown in Figure 5.3. Color A can be removed from the color set of node n1 because color A is the only color that can be assigned to the adjacent node n0.

This pruning rule is embedded in the Prune-Adjacent algorithm, which is shown below. In line 2 of this algorithm, every node in the graph is checked for the number of colors in its color set. If only one color is in the color set then this color is removed from the color sets of the adjacent nodes of u in the Prune-Neighbors function. If after removal only one color remains in the color set of an adjacent node v then this color is removed from the color sets of the nodes that are adjacent to v. Line 2 in the Prune-Neighbors function checks if indeed a color is removed and prevents an endless recursive call of the Prune-Neighbors function. The Prune-Neighbors function is called recursively such that all colors that can be removed according to rule 2, are removed from the color sets in the ASCG after applying the Prune-Adjacent algorithm. The computational complexity of this algorithm is $O(|Z| \cdot |V|^2)$ because ever node has at most V neighbors and is visited at most $|Z|$ times. With $|Z|$ the number of different colors in the union of the color sets in the ASCG.



(a) (b)

Figure 5.4: An example DFG (a) and a data path (b) used to illustrate a limitation of the Prune-Adjacent algorithm.

PRUNE-ADJACENT($ASCG$)
1 **for** each vertex $u \in V[ASCG]$
2 **do if** $|colors(u)| = 1$
3 **then** Prune-Neighbors $(u, colors(u))$

PRUNE-NEIGHBORS(u, c)
1 **for** each $v \in Adj[u]$
2 **do if** $c \in colors(v)$
3 **then** $colors(v) = colors(v) \backslash c$
4 **if** $|colors(v)| = 1$
5 **then** Prune-Neighbors (v, C)

Pruning of the ASCG with the Prune-Adjacent algorithm does not guarantee that it derives all infeasible network ports even if all operations are assigned to a functional unit. An unsuitable backtrack criterium during operation assignment is therefore the detection of an empty color set after pruning with the Prune-Adjacent algorithm.

That the Prune-Adjacent algorithm does not allways derive all infeasible network ports is illustrated with the DFG and data path that are shown in Figure 5.4. The network model of this data path is shown in Figure 5.5. In this example it is assumed that operation n0 and n1 are both assigned to FU_0. The ASCG for this network and data path after the Prune-Adjacent algorithm is applied is shown in Figure 5.4. In this case the Prune-Adjacent algorithm didn't

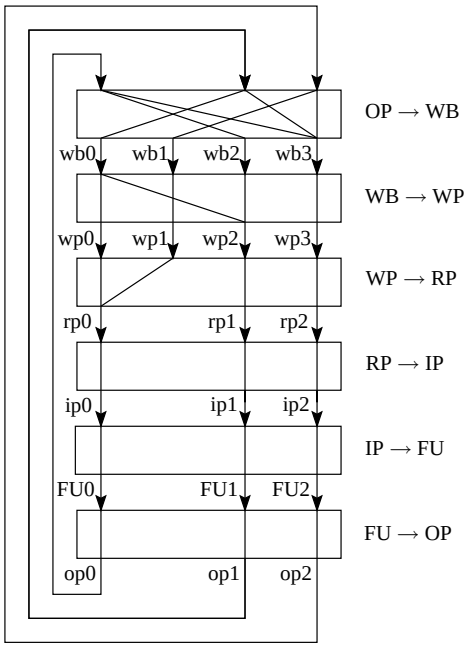


Figure 5.5: The network model of the data path in Figure 5.4.

detect that the assignment of color wp0 or wp1 to the WP node and the color wb2 or wb3 to the WB node is infeasible. The reason is as follows. The assignment of the color op0 to the OP node has as a consequence that the color wb2 as well as wb3 are an option for the WB node. Because there is more than one option for the WB node the Prune-Adjacent algorithm is unable to detect colors that can not be assigned to the WP node. On the other hand, the assignment of the color rp0 to the RP node has as a consequence that the color wp0 as well as wp1 are an option for the WP node. Because more than one color can be assigned to the WP node the Prune-Adjacent algorithm is unable to detect colors that can not be assigned to the WB node.

In the next section another pruning algorithm is described which is able to derive all infeasible network ports if all operations are assigned to a functional unit.

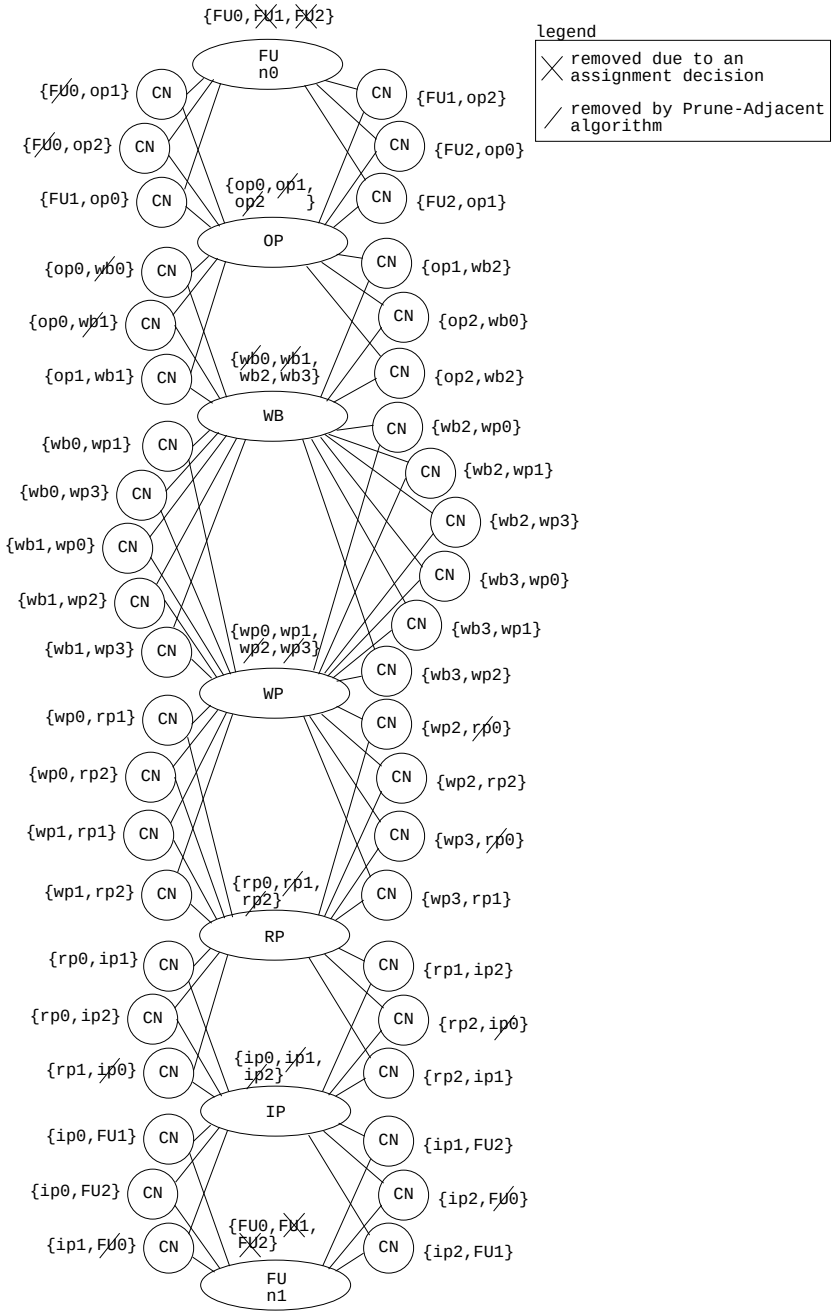


Figure 5.6: ASCG for the DFG and data path in Figure 5.4.

5.2 Connectivety driven pruning

This section describes a pruning algorithm which derives infeasible operation assignment options which are only a consequence of the use of an incomplete network in the data path. The pruning algorithm is explained in the following steps. First a simple example that illustrates how the interconnect in the networks affect the assignment of operations is described in Section 5.2.1. Then in Section 5.2.2 a graph is introduced in which the communication paths between functional units can be modeled that execute the operations in the data flow graph. This graph is the input of a pruning algorithm which is described in Section 5.2.3. How cyclic data flow graphs can be handled by this pruning algorithm is described in Section 5.2.4. In Section 5.2.5, Section 5.2.6 and Section 5.2.7 several extensions of the pruning algorithm are described that can be used to derive infeasible operation assignment options which are a consequence of a combination of constraints.

5.2.1 Pruning example

The data path and DFG in Figure 5.7 and the corresponding data path network model in Figure 5.8 are used to illustrate how operation assignment is affected by the use of an incomplete network in the data path. Assume that it is decided that operation $n0$ is assigned to functional unit FU_0. This decision has as a consequence that the result of operation $n0$ can not be stored in RF_0 because there is no communication path through the $OP \rightarrow WB$ and the $WB \rightarrow WP$ network to this register file. This result must therefore be stored in RF_1. A result stored in RF_1 is not accessible by FU_0 and FU_1. Therefore the assignment of operation $n1$ to FU_0 or FU_1 is infeasible and the corresponding colors can be removed from the color set of the ASCG node $FU(n0)$ (see definition 4.7).

5.2.2 Communication path graph

In this section a graph is introduced in which the communication paths between functional units are modeled that potentially execute the operations in the data flow graph. This graph is the input of a pruning algorithm which is described in Section 5.2.3.

In the network model Figure 5.8 the functional units are represented as output ports of the IP $\rightarrow$ FU network. This makes it possible to use the same pruning algorithm for the derivation of infeasible operation assignment options as well as for the derivation of infeasible network ports and functional unit ports.

The infeasible operation assignment options are derived with a pruning algorithm that operates on a directed graph. The graph has been given the name “Communication Path Graph (CPG)”. In this graph all possible communication paths between a producing and a consuming functional unit are modeled.

The CPG for the data path and the DFG shown in respectively Figure 5.7 is depicted in Figure 5.9. This graph is constructed as follows. For every operation in the DFG the same number of nodes as there are functional units are created in the CPG. Therefore there are

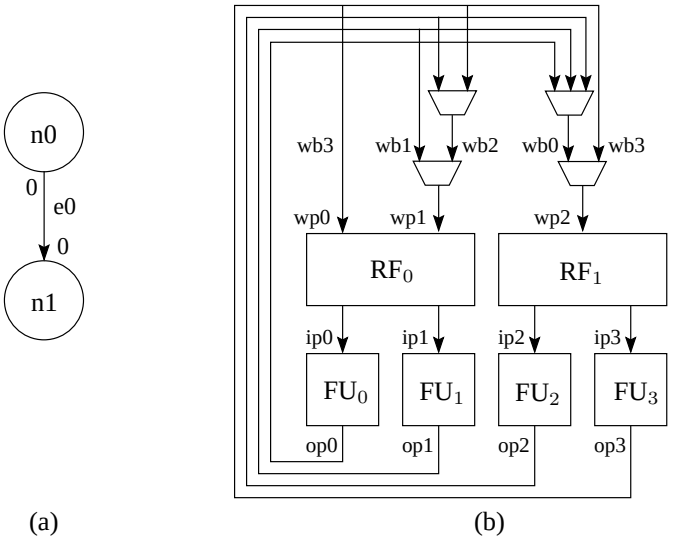


Figure 5.7: A data flow graph (a) and a data path (b) which are used to illustrate that the interconnect in the networks affect the assignment of operations to the functional units.

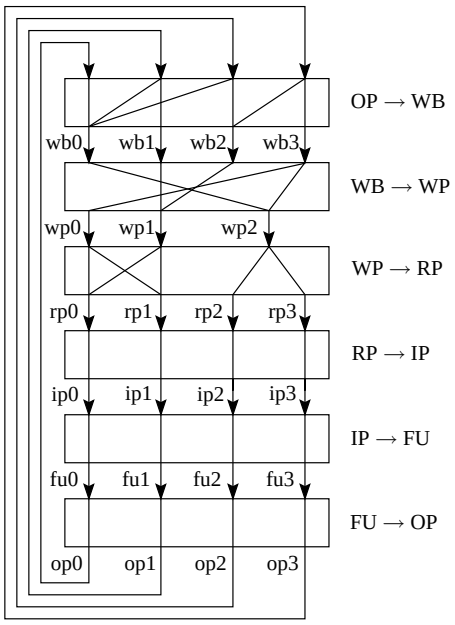


Figure 5.8: A data path network model in which all communication networks and ports of the data path in Figure 5.7b are made explicit and labeled.

four nodes created in the CPG in Figure 5.9 for operation n0 as well as for operation n1. For every data edge in the DFG there are as many nodes created in the CPG as there are network ports in the data path. Every node that is created corresponds to one specific network port. Therefore there are four nodes created in the CPG for respectively the output ports (ops), the write-back busses (wbs) the read ports (rps) and the input ports (ips). Because there are three write ports in the data path there are also only three nodes created in the CPG. Directed edges are added between the nodes in the CPG if there exist a connection in the network in the data path between the corresponding network input port and network output port. In addition, a source and a sink node are included in the CPG. Directed edges are added from the source node to all nodes in the CPG that correspond to functional units that execute operations that do not have any incoming data edge in the DFG. There are also directed edges to the sink node added. These edges leave the nodes in the CPG that correspond to functional units that potentially execute operations that do not have outgoing data edges.

The communication path graph, which is described by the tuple (W, E), is defined below. This graph is defined given a DFG $(V, E_d \cup E_l \cup E_s, w_e, \text{id}, t_i, t_o)$ (see Definition 2.1) with the set of data edges $E_{de} = E_d \cup E_l$. In the definition of the communication path graph the function $k(\text{ip}, \text{op})$ is used. This function describes a network in the data path and is defined in Section 4.2.

Definition 5.2 (mapping of DFG operation on functional unit vertices)
$FUs : V \rightarrow W$ is the function $FUs(v) = V_{FU}$ that gives for a DFG operation $v \in V$ a set of vertices V_{FU}. There is for every functional unit in the data path a one to one corresponding vertex $v_{FU} \in V_{FU}$.

Definition 5.3 (mapping of a DFG operation on functional unit output port vertices)
$OPs : E_{de} \rightarrow W$ is the function $OPs(e) = V_{op}$ that gives for a data edge e a set of vertices V_{op}. There is for every functional unit output port in the data path a one to one corresponding vertex $v_{op} \in V_{op}$.

Definition 5.4 (mapping of a DFG operation on write back bus vertices)
$WBs : E_{de} \rightarrow W$ is the function $WBs(e) = V_{wb}$ that gives for a data edge e a set of vertices V_{wb}. There is for every write back bus in the data path a one to one corresponding vertex $v_{wb} \in V_{wb}$.

Definition 5.5 (mapping of a DFG operation on write port vertices)
$WPs : E_{de} \rightarrow W$ is the function $WPs(e) = V_{wp}$ that gives for a data edge e a set of vertices V_{wp}. There is for every write port in the data path a one to one corresponding vertex $v_{wp} \in V_{wp}$.

Definition 5.6 (mapping of a DFG operation on read port vertices)
$RPs : E_{de} \rightarrow W$ is the function $RPs(e) = V_{rp}$ that gives for a data edge e a set of vertices V_{rp}. There is for every read port in the data path a one to one corresponding vertex $v_{rp} \in V_{rp}$.

Definition 5.7 (mapping of a DFG operation on input port vertices)
$IPs : E_{de} \rightarrow W$ is the function $IPs(e) = V_{ip}$ that gives for a data edge e a set of vertices V_{ip}. There is for every input port in the data path a one to one corresponding vertex $v_{ip} \in V_{ip}$.

Definition 5.8 (Network port)
$P(v) = port$ gives for a CPG vertex the one to one corresponding network port in the data path network model.

Definition 5.9 (Source vertex)
source() $= v_{source}$ gives the source vertex of the CPG.

Definition 5.10 (Sink vertex)
sink() $= v_{sink}$ gives the sink vertex of the CPG.

Definition 5.11 (Communication Path Graph for a data edge)
$\psi : E_{de} \rightarrow W \times W \times W \times E$ is the function $\psi(e) = (V_{FUp}, V_p, V_{FUc}, E)$ where
$n1 =$ pred(e)
$V_{FUp} = FUs(n1)$
$V_{op} = OPs(e)$
$V_{wb} = WBs(e)$
$V_{wp} = WPs(e)$
$V_{rp} = RPs(e)$
$V_{ip} = IPs(e)$
$n2 =$ succ(e)
$V_{FUc} = FUs(n2)$
$E1 = \{(v_{FU}, v_{op}) \mid v_{FU} \in V_{FUp}, v_{op} \in V_{OP}, k_{FUOP}(P(v_{FU}), P(v_{op})) = 1\}$
$E2 = \{(v_{op}, v_{wb}) \mid v_{op} \in V_{op}, v_{wb} \in V_{WB}, k_{OPWB}(P(v_{op}), P(v_{wb})) = 1\}$
$E3 = \{(v_{wb}, v_{wp}) \mid v_{wb} \in V_{wb}, v_{wp} \in V_{WP}, k_{WBWP}(P(v_{wb}), P(v_{wp})) = 1\}$
$E4 = \{(v_{wp}, v_{ip}) \mid v_{wp} \in V_{wp}, v_{ip} \in V_{IP}, k_{WPIP}(P(v_{wp}), P(v_{ip})) = 1\}$
$E5 = \{(v_{ip}, v_{FU}) \mid v_{ip} \in V_{ip}, v_{FU} \in V_{FUc}, k_{IPFU}(P(v_{ip}), P(v_{FU})) = 1\}$
$E = E1 \cup E2 \cup E3 \cup E4 \cup E5$
$V_p = V_{op} \cup V_{wb} \cup V_{wp} \cup V_{rp} \cup V_{ip}$

Definition 5.12 (Communication Path Graph.)
Given a data path of a processor and a DFG. Then the Communication Path Graph (CPG) is represented by the directed graph (W, E), where:
$\psi'(E_{de}) = (W, E)$
$v_{source} =$ source()
$v_{sink} =$ sink()
$W1 = \{Vp \mid e \in E_{de}, (Vp, Va, Vc, Ea) = \psi(e)\}$
$W2 = \{Va \mid e \in E_{de}, (Vp, Va, Vc, Ea) = \psi(e)\}$
$W3 = \{Vc \mid e \in E_{de}, (Vp, Va, Vc, Ea) = \psi(e)\}$
$E1 = \{Ea \mid e \in E_{de}, (Vp, Va, Vc, Ea) = \psi(e)\}$
$E2 = \{(v_{source}, v) \mid e \in E_{de}, (Vp, Va, Vc, Ea) = \psi(e), v \in Vp, pred(e) = \emptyset\}$
$E2 = \{(v, v_{sink}) \mid e \in E_{de}, (Vp, Va, Vc, Ea) = \psi(e), v \in Vc, succ(e) = \emptyset\}$
$W = W1 \cup W2 \cup W3 \cup v_{source} \cup v_{sink}$
$E = E1 \cup E2 \cup E3$

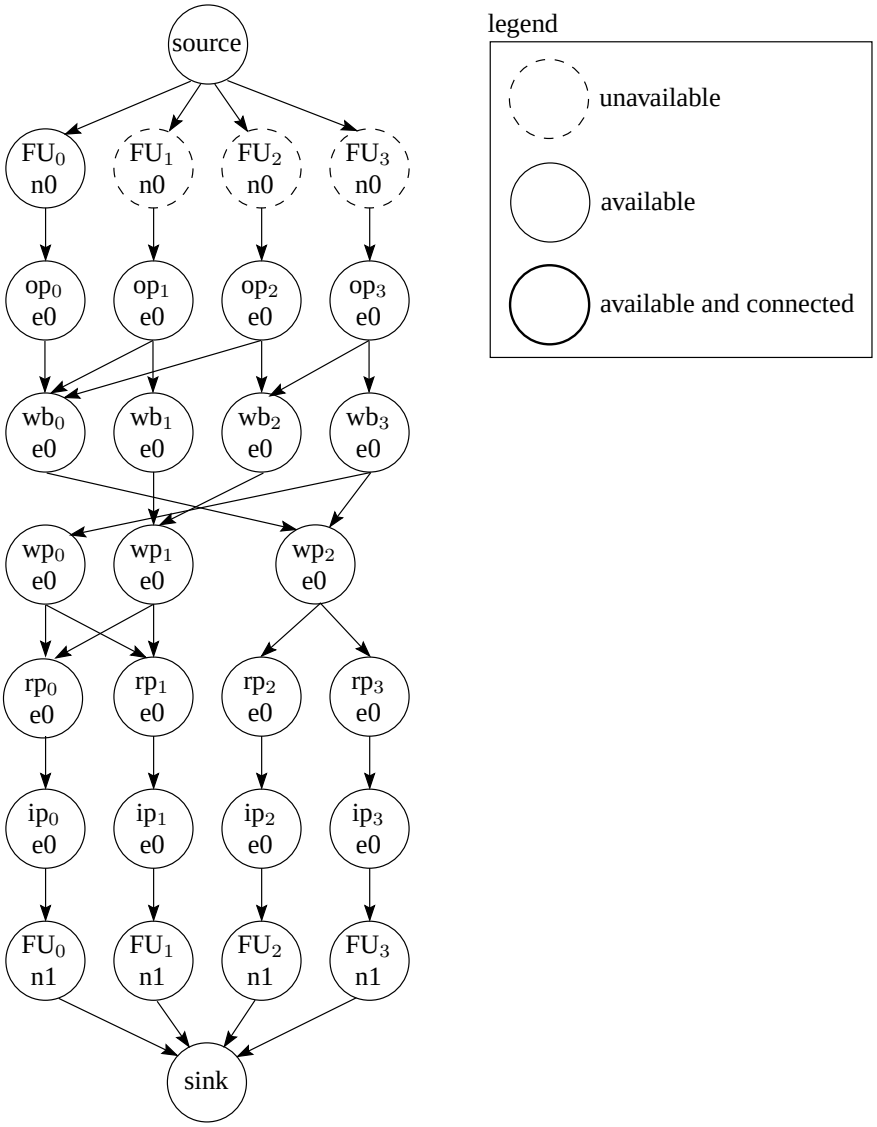


Figure 5.9: The Communication Path Graph (CPG) for the DFG and data path shown in Figure 5.7. This graph is used to derive infeasible operation and port assignment options.

5.2.3 Communication path graph pruning algorithm

The pruning algorithm that operates on the CPG has been given the name “Prune-CPG”. This algorithm is described below.

All the nodes in the CPG that correspond to ports and functional units that can be used for

the execution of the operations are marked as available before the Prune-CPG algorithm is applied. A functional unit is unavailable if the type of the operation is different from the type the functional unit can execute. Another reason is that a decision has been made to assign the operation to another functional unit. In addition, the colors in the ASCG can indicate that a functional unit or a port can not be used. Because, in this example we assume that the decision was made that operation n0 is executed on FU_0 all nodes in the CPG in Figure 5.9 will be marked as available except the nodes that are labeled with "FU_1 n0" and "FU_2 n0" and "FU_3 n0". The nodes which are unavailable are drawn with dashed lines in Figure 5.9.

The Prune-CPG algorithm is used to derive infeasible operation assignment options. In order to derive these infeasible assignment options, it derives the nodes in the CPG that lie on paths from the source node to the sink node through nodes that are marked as available. All nodes that are not on a path from the source to the sink node correspond to ports or functional units that are infeasible assignment options because they do not lie on a communication path from a functional unit that can execute the producing operation to a functional unit that can execute the consuming operation. The CPG for the DFG and data path in Figure 5.7 is shown in Figure 5.9. The nodes and edges that lie on a path from source to sink through nodes that are marked as available and connected are drawn with fat lines in Figure 5.10.

The Prune-CPG algorithm, which is shown below, finds the nodes on the paths from the source node to the sink node by traversing the CPG graph in a depth-first-search manner from the source to the sink node. Before the traversal starts the nodes are marked in line 2 of the Prune-CPG algorithm as not visited ($visited[u] = false$) and in line 3 as not connected to the sink ($connected[u] = false$). In line 4 the $source$ node is taken as a starting point of the depth-first-search traversal. If during the traversal a node is discovered which is available and not already visited then this node is marked as visited in line 7. All other neighbor nodes are visited in order to derive whether they are on a path to the sink. If during the traversal the $sink$ node is discovered then all the nodes on its way back to the source node are marked as nodes that are connected to the $sink$ node ($connected[u] = true$). In the case a node is discovered that was already visited then it is known if this node is connected to the $sink$. The reason for this is that the CPG graph is acyclic. Because the CPG is acyclic a node in the CPG can only be visited again after all nodes that are "further away" from the $source$ node and "closer" to the $sink$ node have been explored. In the case that a visited node is discovered that is marked as connected then all the nodes on its way back to the source node are marked as nodes that are connected to the $sink$ node.

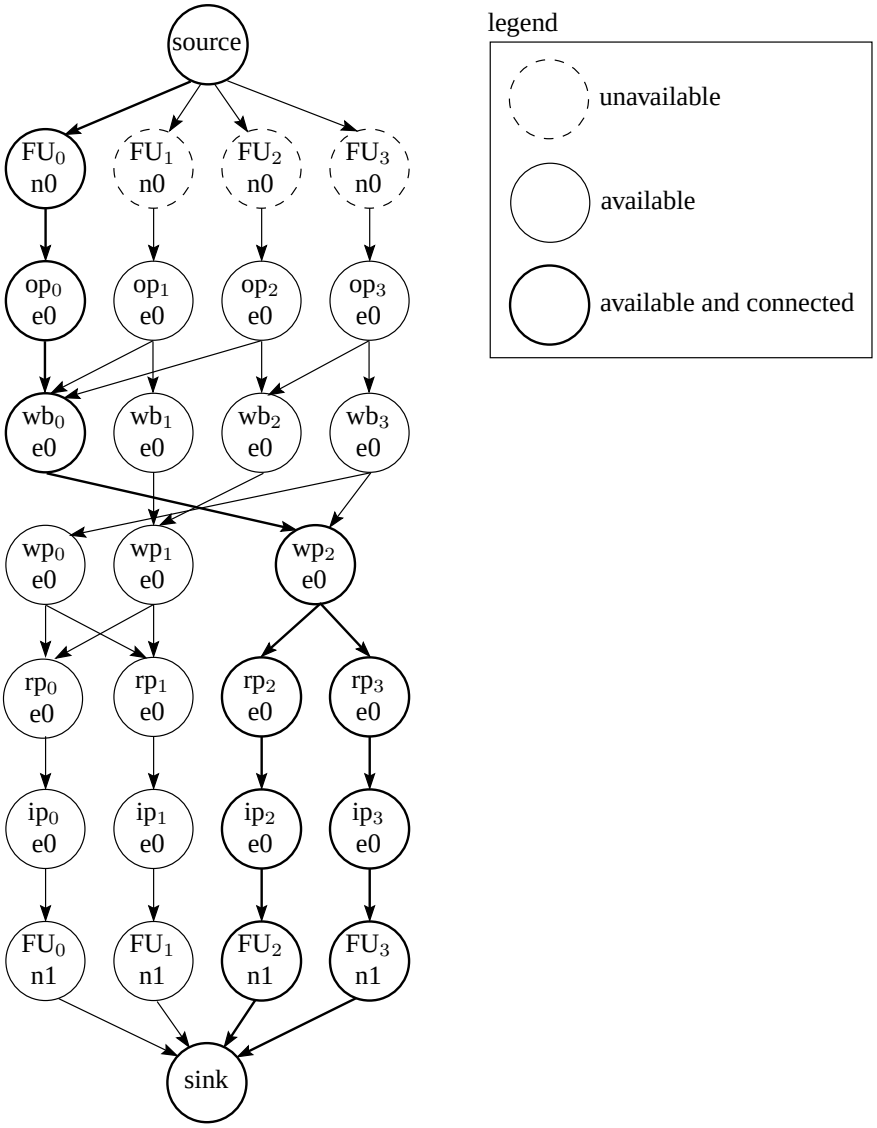


Figure 5.10: A communication path graph after the Prune-CPG algorithm has been applied.

```
PRUNE-CPG(CPG)
 1  for each vertex u ∈ V[CPG]
 2  do visited[u] ← false
 3     connected[u] ← false
 4  Visit(source)

VISIT(u)
 1  if u = sink
 2     then return true
 3  if available[u] = false
 4     then return false
 5  if visited[u] = true
 6     then return connected[u]
 7  visited[u] ← true
 8  retval ← false
 9  for each v ∈ Adj[u]
10  do if Visit(v) = true
11        then connected[u] ← true
12             retval ← true
13
14  return retval
```

The Prune-CPG algorithm is a adapted version of a depth-first-search algorithm which is described in for example [CLR90]. The Prune-CPG algorithm has a computational complexity of $O(\mid V \mid + \mid E \mid)$ because it visits every node and edge in the graph maximal once.

5.2.4 Communication path graphs for cyclic data flow graphs

This section describes how cyclic DFG can be modeled in the CPG such that the Prune-CPG algorithm can be used for the derivation of infeasible assignment options.

A DFG with a loop-carried data precedence edge is shown in Figure 5.11. This loop-carried data precedence edge is replaced by an ordinary data precedence edge by introducing a dummy operation for every operation that has an incoming loop-carried data precedence edge. This results in the DFG' shown in Figure 5.12 in which the dummy operation n0' is introduced which has an incoming data precedence edge from operation n0. Because DFG' is acyclic a CPG can be constructed according to definition 5.12.

A constraint that can not be modeled in the CPG is that the operations n0 and n0' must be assigned to the same functional unit. For example if after applying the Prune-CPG it is detected that a functional unit that is an option for the execution of operation n0 is not marked connected, then also the same functional unit for the execution of operation n0' should be marked as unavailable. After an update of the CPG the Prune-CPG algorithm is applied once again.

With the above described procedure, some but not all infeasible operation assignment op-

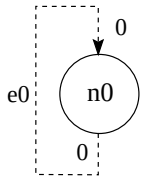


Figure 5.11: Example DFG with a loop-carried data dependency.

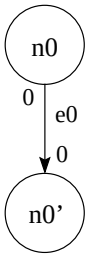


Figure 5.12: DFG' in which the loop-carried data dependency of the DFG in Figure 5.11 is removed.

tions can be derived by making use of the Prune-CPG algorithm. A data path is shown in Figure 5.13 for which there does not exist a feasible assignment of the operations in the DFG of Figure 5.11. The corresponding CPG after pruning is shown in Figure 5.15. All functional units lie on a path from the source to the sink and are marked as connected. Thus, they are marked as potentially feasible assignment options while there is no feasible operation assignment possible.

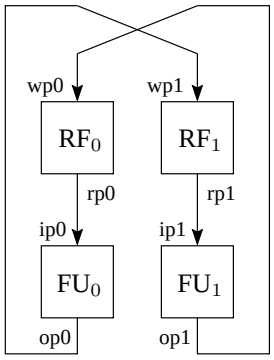


Figure 5.13: A data path used to illustrate that the Prune-CPG does not derive all infeasible operation assignment options for the cyclic DFG in Figure 5.11.

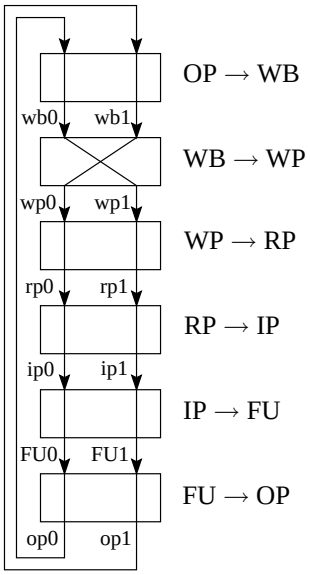


Figure 5.14: A data path network model of the data path in Figure 5.13.

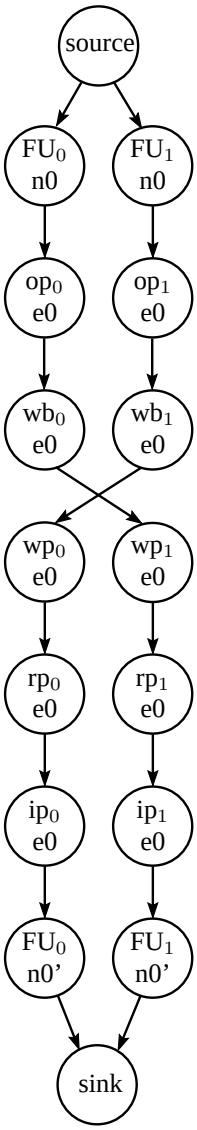


Figure 5.15: CPG for the acyclic DFG' shown in Figure 5.12 and the data path of Figure 5.13.

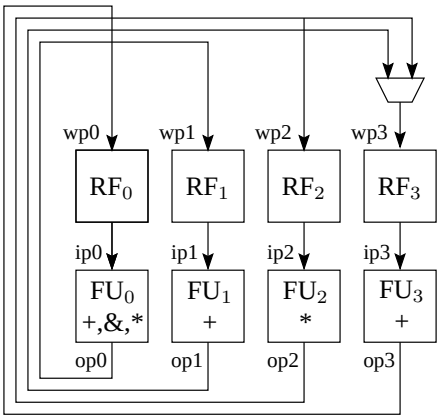


Figure 5.16: Data path used to illustrate that a combination of interconnect constraint can make operation assignment options infeasible.

So far, CPGs are described for two operations with a data precedence edge between them. These CPGs were used to prune the assignment search-space based on interconnect constraints between functional units that can executed these operations. However, the pruning techniques should be extended to derive infeasible assignment options based on a combinations of interconnect constraints. These extensions are described in the next sections.

5.2.5 Data dependency chains

In typical DFGs most of the operations consume values that are produced by other operations and produce results that are the input values of other operations. Therefore, the assignment of an operation to a functional unit must be such that a communication path exists from the functional units that produce the input values for this functional unit. At the same time a communication path must exist from this functional unit to the functional units that consumes its result. An example data path and DFG is used to explain the derivation of infeasible operation assignment options with the Prune-CPG algorithm for DFGs with more than two operations that are data dependent. The data path and DFG are shown in respectively Figure 5.16 and Figure 5.18. The data path network model is depicted in Figure 5.17.

The infeasible assignment options can be derived by making use of the CPG in Figure 5.19. The fat line in this CPG is the only path from source to sink through the nodes that are marked as available. All the functional units and ports which are not on this path are infeasible assignment options.

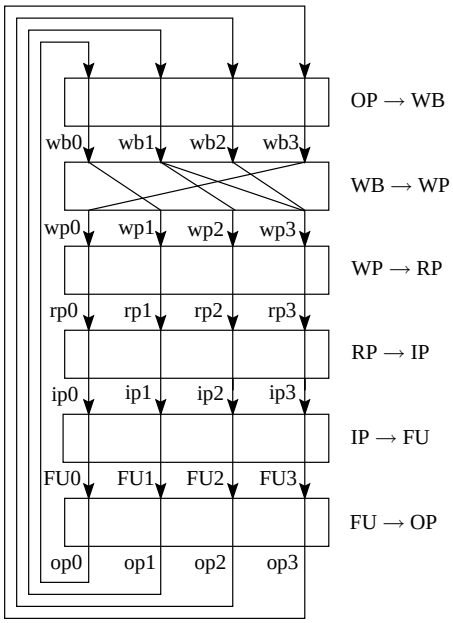


Figure 5.17: The network model of the data path in Figure 5.16.

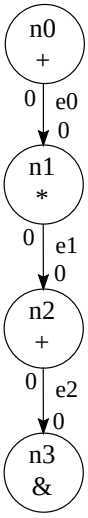


Figure 5.18: Example DFG of which the operations are assigned to the functional units in the data path of Figure 5.16.

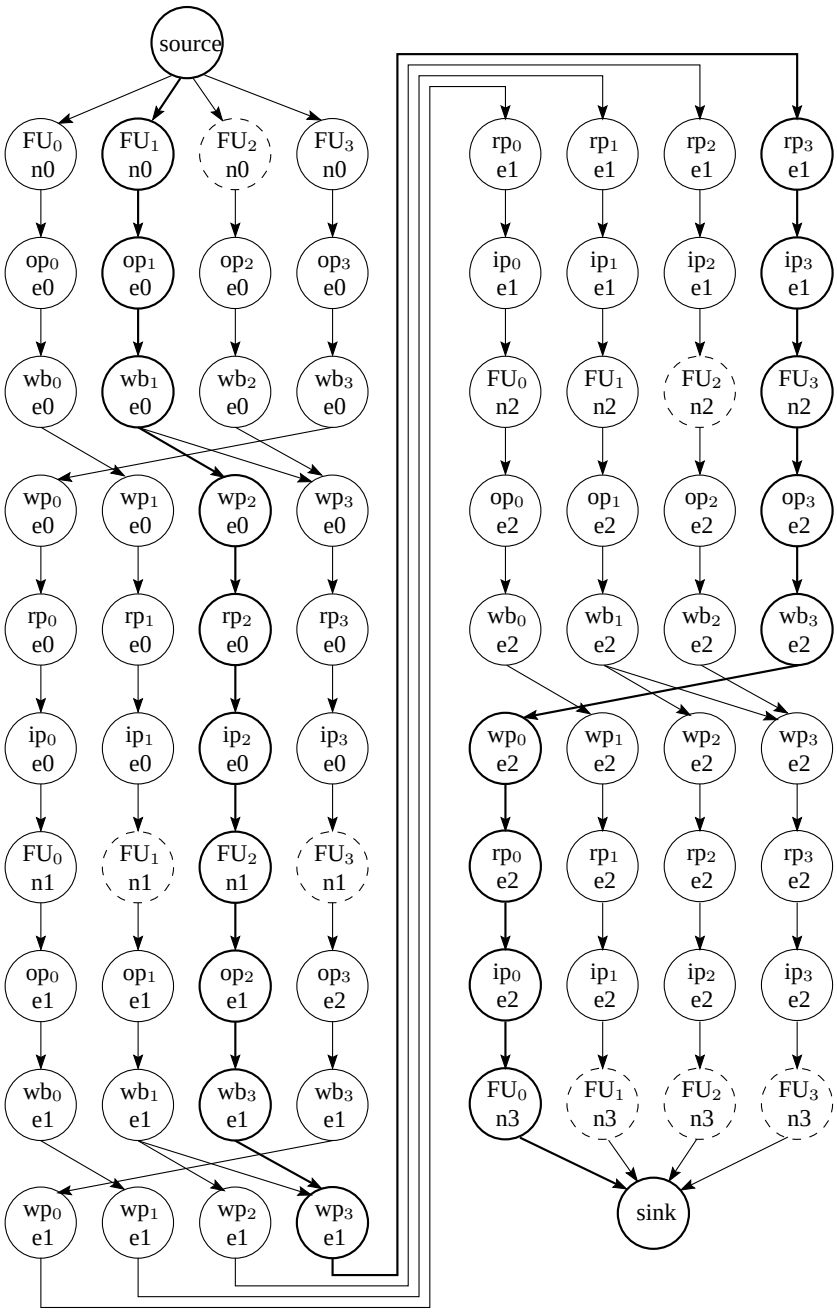


Figure 5.19: CPG for the data path in Figure 5.16 and the DFG in Figure 5.18.

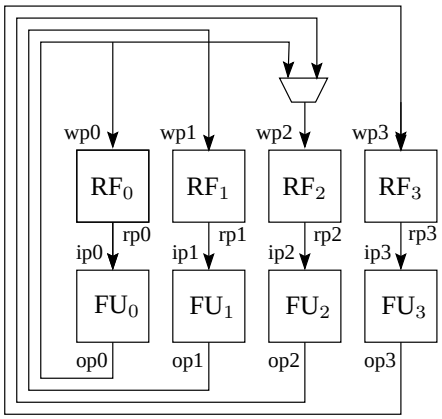


Figure 5.20: Data path used to illustrate pruning caused by a combination of connectivity constraints.

5.2.6 Diverging data dependencies

In the previous example the assignment was derived for a DFG with operations that consume at most one input value and produce at most one output value. In this case, infeasible operation assignment options were derived by applying the Prune-CPG algorithm on the corresponding CPG. This section describes the derivation of infeasible options for the case that a result of an operation is consumed by several operations. In this case, not one communication path but several paths must exist from the producing functional unit output port to several consuming functional unit input ports.

A data path and a DFG are shown in respectively Figure 5.20 and Figure 5.22. Operation $n0$ in the DFG produces a value that is consumed by two other operations. The assignment of these operations must be such that a communication paths exist to the functional units that execute the consuming operations. In this example we will assume that operation $n1$ can be executed on FU_0 or FU_3 and operation $n2$ can only be executed on FU_2. Notice that under these assumptions the only feasible assignment option for operation $n0$ is FU_0.

The corresponding CPG, after applying the Prune-CPG algorithm once, is shown in Figure 5.23. All the nodes that are draw with thin lines correspond to functional units or ports that are infeasible assignment options. However the nodes labeled with "FU_0 n0" and "FU_3 n0" are drawn with fat lines because they lie on a path from the source to the sink node. This indicates that both units are marked as potentially feasible assignment options for operation n0 while only FU_0 is indeed feasible.

To be able to detect that FU_3 is infeasible it should be taken into account that nodes in the CPG can correspond to the same functional unit or the same port in the data path. If one of these nodes in the CPG are not marked as available and connected by the Prune-CPG algorithm then the other nodes that correspond to the same functional unit or port should be marked as unavailable.

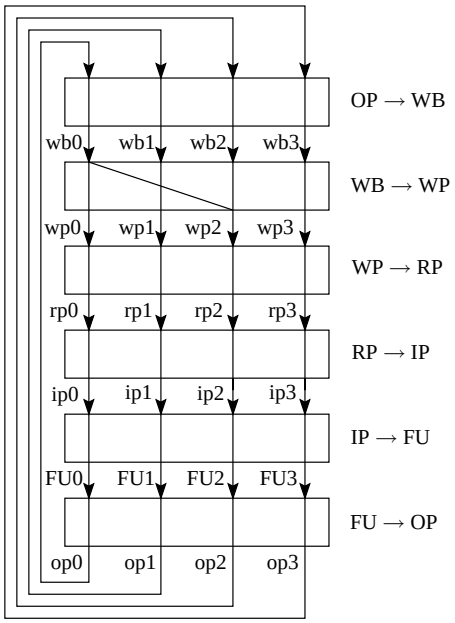


Figure 5.21: Network model of the data path of Figure 5.20.

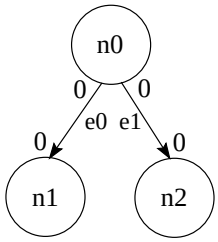


Figure 5.22: DFG used to illustrate pruning.

In the CPG of Figure 5.23 the nodes labeled with "op_3 e0" and "op_3 e1" correspond to exactly the same functional unit output port. The node "op_3 e1" is not marked as connected and therefore the node labeled with "op_3 e0" should be marked as unavailable. The resulting CPG, after applying the Prune-CPG algorithm a second time, is shown in Figure 5.24. In this CPG the node labelled with "FU_3 n0" is not marked as connected which indicates that it is indeed an infeasible assignment option for operation n0.

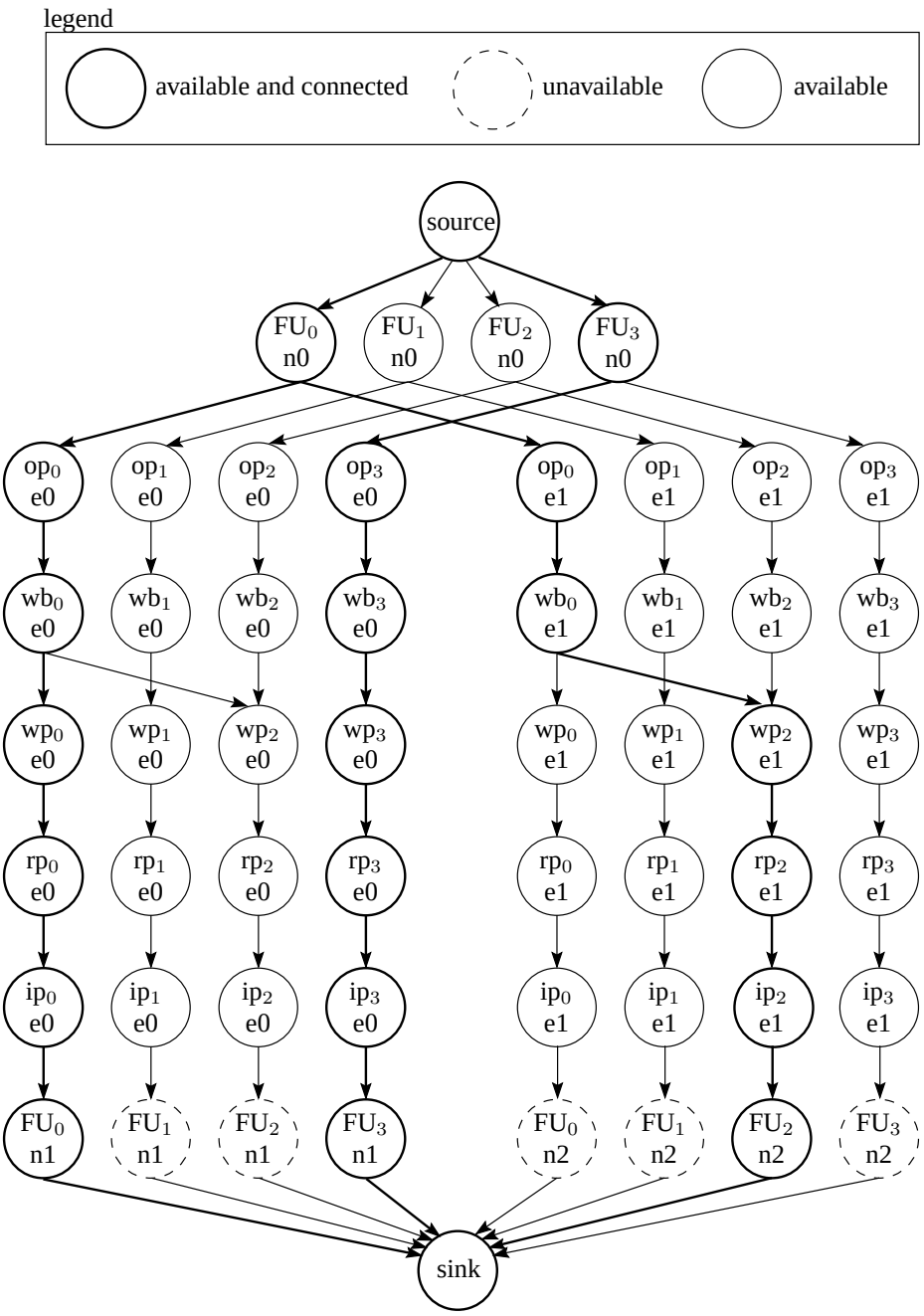


Figure 5.23: The Communication Path Graph (CPG) for the data path in Figure 5.20 and the DFG in Figure 5.22 on which the Prune-CPG algorithm has been applied.

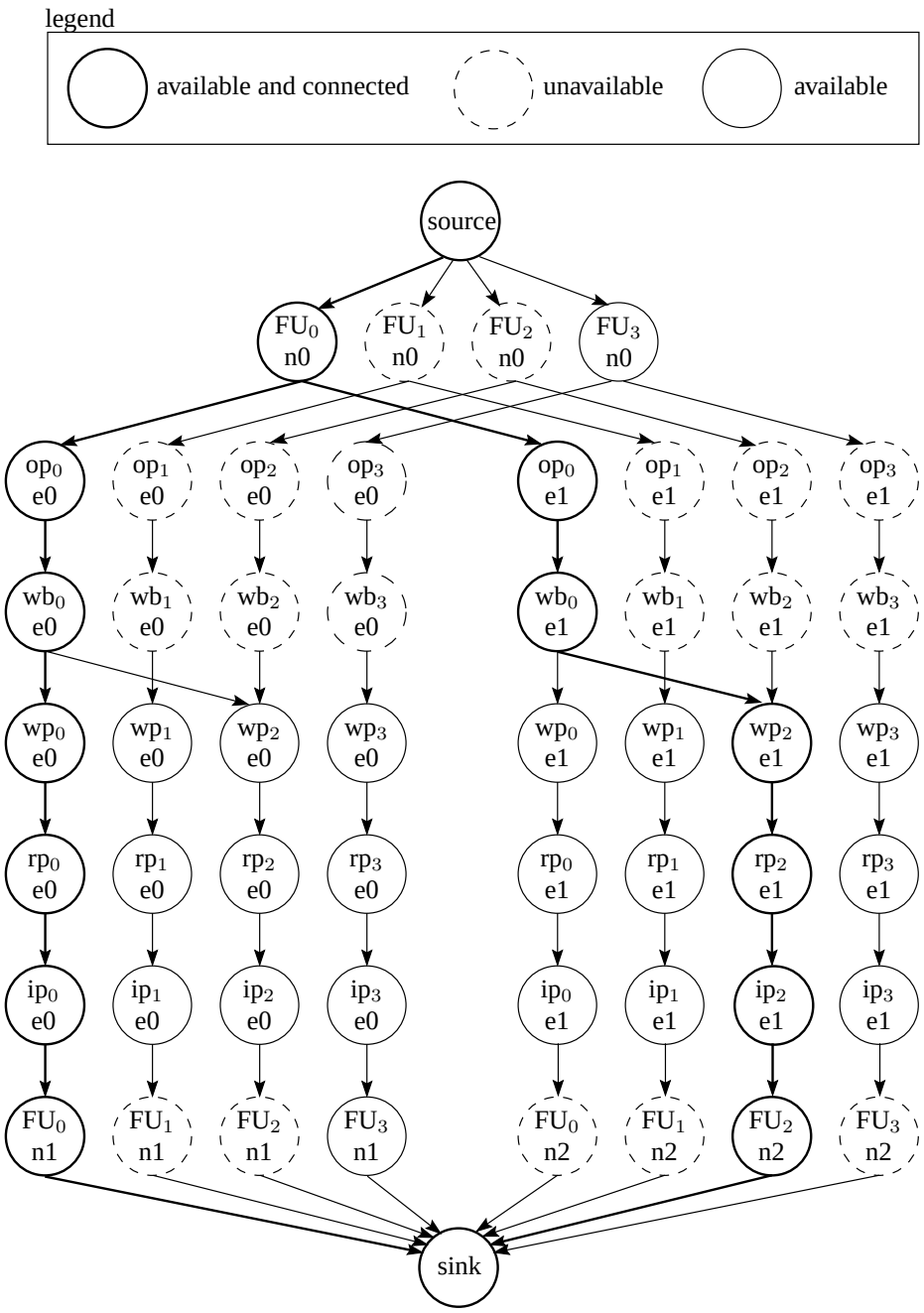


Figure 5.24: The Communication Path Graph (CPG) after the CPG algorithm has been applied twice.

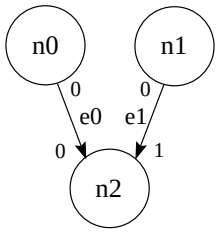


Figure 5.25: DFG used to explain how the CPG pruning techniques should be extended such that it can be derived which functional units can not read the required input values.

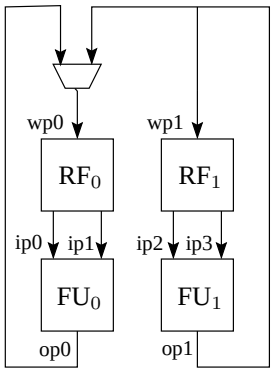


Figure 5.26: Data path used to explain how the pruning techniques should be extended such that it can be derived which functional units can not read the required input values.

5.2.7 Converging data dependencies

In this section the problem is addressed that the assignment of operations to functional units must be such that communication paths exist from several producing operations to an operation that consumes the results. The DFG and data path used to explain the addressed problem is shown in Figure 5.25 and Figure 5.26 respectively. The corresponding data path network model is shown in Figure 5.27. In this example the assumption is made that operation n0 is executed on FU_0 and that operation n1 is executed on FU_1. Notice that in this case operation n2 can only be executed on FU_0.

The CPG that corresponds to the DFG and the data path of respectively Figure 5.25 and Figure 5.26 is shown in Figure 5.28. The nodes labeled with "FU_0 n2" and "FU_1 n2" are marked as connected and therefore as an option for execution of operation n2. However to be able to execute an operation on a functional unit there must exist a communication path to all functional unit input ports that are used. In the example this is not the case for FU_1 that must consume a value produced by n0 through functional unit input port ip_2. Because "ip_2 e0" is not marked as connected we must mark the node "ip_3 e1" as unavailable. After updating the

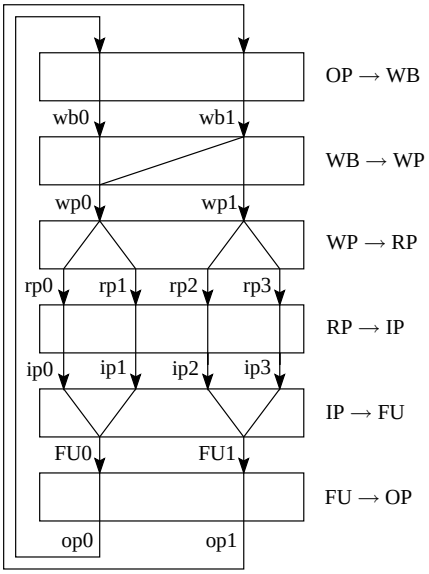


Figure 5.27: Network model of the data path in Figure 5.26.

CPG the Prune-CPG algorithm is applied again. The resulting CPG is shown in Figure 5.29 in which node "FU_1 n2" is not marked as connected.

5.2.8 Guarantees after pruning

Pruning algorithms typically remove some but usually not all infeasible options from a search-space. However, the applied pruning algorithms give some guarantees about which infeasible assignment options will certainly be removed. Why these guarantees are essential is described in this section.

After the operation assignment phase all operations should be assigned to a functional unit such that the required communication paths can be made in the data path of the processor. The CPG-algorithm is an essential pruning algorithm because it has the following property:

Theorem 5.1
Given that every operation in the DFG is assigned to a functional unit then infeasibility is detected with the prune-CPG algorithm if a required communication path can not be made in the data path of the processor.

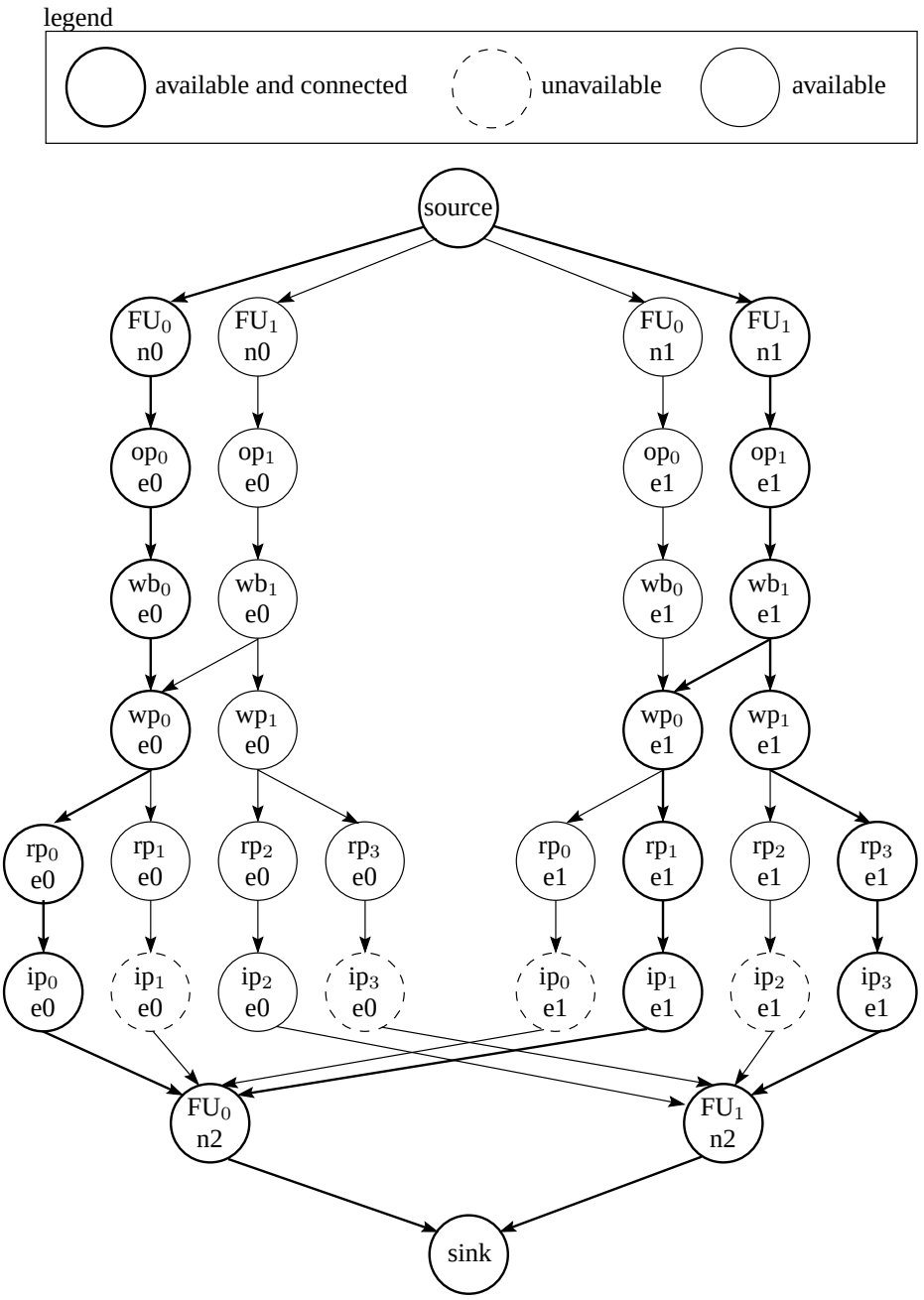


Figure 5.28: The Communication Path Graph (CPG) used to detect the functional units and ports which are infeasible operation assignment options.

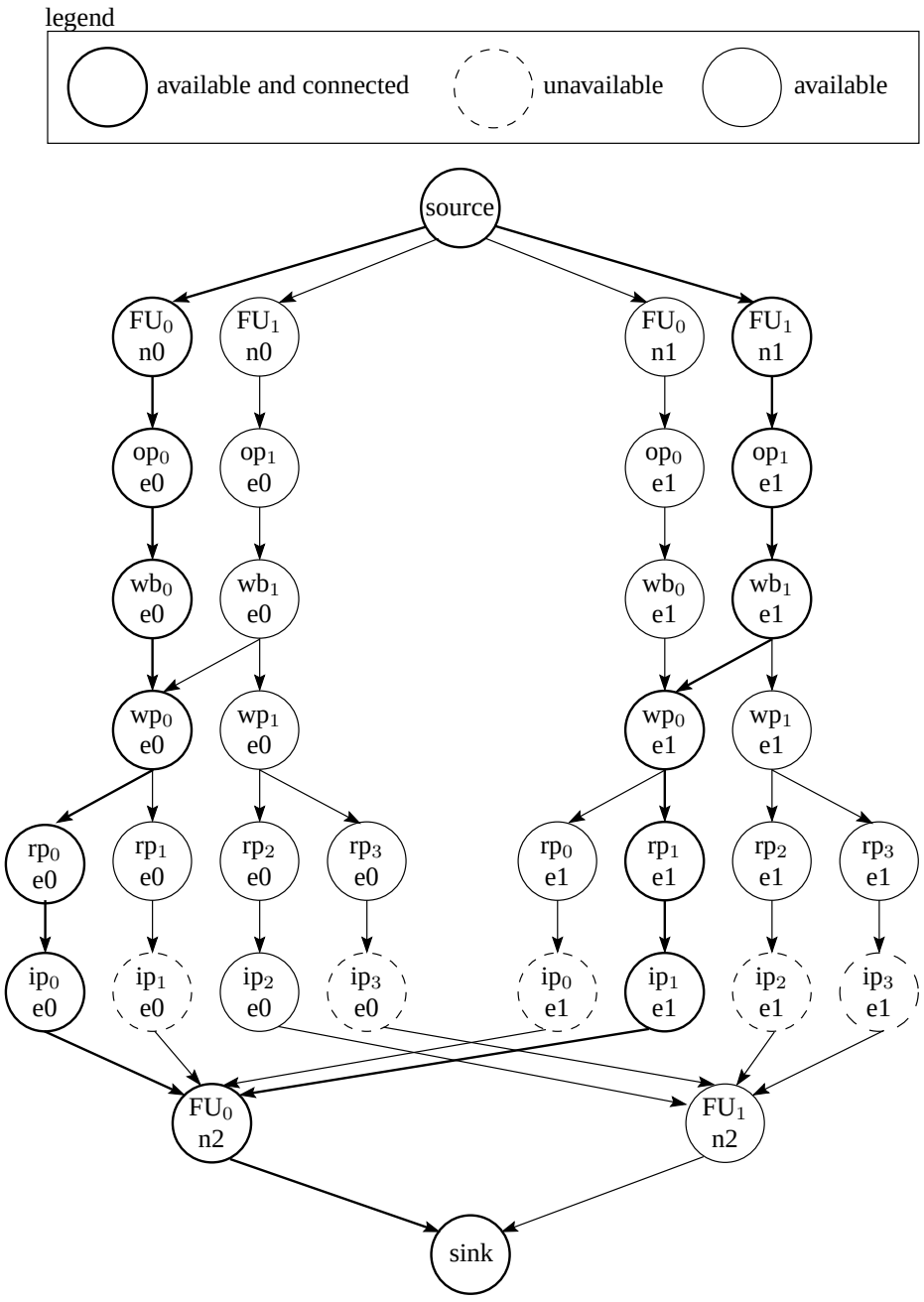


Figure 5.29: The same Communication Path Graph (CPG) as in Figure 5.28 but after the CPG is updated and CPG algorithm is applied a second time.

proof:

A data edge in the DFG implies data transport of a result from a producing FU to the FU that consumes the results. All possible communication paths in the data path between these FUs are modeled in the CPG because for a single data edge there is a one to one mapping of FUs and network ports in the data path and nodes in the CPG. There is also a one to one mapping of connections between network input ports and network output ports and directed edges between the corresponding nodes in the CPG. Therefore, if and only if there is a communication path in the data path from the producing FU to the consuming FU then there is also a corresponding path in the CPG. If there is such a path in the CPG it will be detected with the CPG-algorithm because the CPG-algorithm traverses all paths from the source to the sink node. The situation that is described in Section 5.2.4, in which there is a path in the CPG while there is no feasible communication path in the data path, can not occure because all operations are assigned. □

The CPG-algorithm can replace coloring of the ASCG after every operation assignment decision because the CPG-algorithm can according to Theorem 5.1 be used to detect that a required communication path does not exist. Pruning is prefered above coloring of the ASCG because the pruning algorithm has a polynomial time complexity.

After the operation assignment phase not only the operations should be assigned but it should also be known which network ports are used. This makes is possible to take all the resource conflicts into account during serialization and scheduling.

That the ports that are used are a consequence of the operation to functional unit assignment can be seen as follows. The used functional unit input ports and output ports follow directly from the assignment of the operations and the functional unit port specification in the DFG. For every functional unit input port a unique register file read port can be derived because according to the VLIW data path template there is only one register file read port connected to every functional unit input port. If a functional unit output port is connected to a local write-back bus and this bus is part of the required communication path then this bus is selected otherwise the global write-back bus is selected. The used read port determines in which register file the intermediate value must be stored. The used write port of this register file can be derived from the selected write-back bus if every register file write port is connected to to maximal one write-back bus. We restrict ourselves to this type of data paths.

Chapter 6

Multi-casting

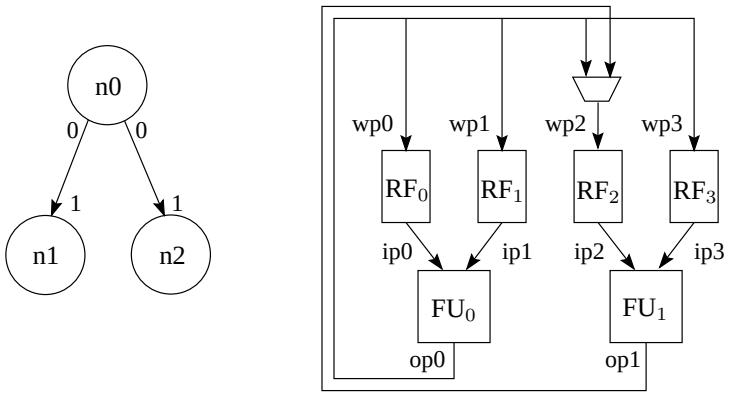
The processors that are considered in this report support multi-casting. Multi-casting allows a result of a functional unit to be stored in multiple register files. This chapter describes the issues related to the exploitation of multi-casting by our operation assignment techniques.

The outline of this chapter is as follows. Section 6.1 explains in more detail what multi-casting is and why it is useful. Section 6.2 explains the issues related to modeling of multicasting in the ASCG. Copy operations that must be inserted in the DFG before it can be mapped on a data path, is the topic of Section 6.3. Modification of the data flow graph is highly undesirable because it usually requires a computational expensive recalculation of the distance matrix. Section 6.4 describes the issues related to the use of global write-back bus in the data path. The use of a global write-back bus in combination with multi-casting makes it always possible to map the DFG on the data path without copy operations. If a global write-back bus is applied then there can be several communication paths from a functional unit output port to a register file write port. How the proper write-back busses are selected during operation assignment, is described in Section 6.4.1. The last section of this chapter describes how the scalability of the data path can be enhanced by making use of pipeline registers in the global write-back busses.

6.1 Multi-casting concept

In Figure 6.1a a DFG is shown in which an intermediate result is consumed by two operations. If these operations are executed on functional units that do not have access to the same register file then the intermediate result must be stored in two register files. A processor is said to support multi-casting if an intermediate result can be stored in more than one register file in the same clock cycle.

Multi-casting becomes a difficult problem if the number of register files in which an intermediate result must be stored depends on the assignment of the consuming operations. In this case the number of used register file write ports depends on the assignment of the consuming



(a) (b)

Figure 6.1: Example DFG (a) and data path (b) used to illustrate modeling of multicating in the ASCG.

operations to the functional units. The use of a larger number of write ports can increase the latency and the initiation interval of the schedule. Also, the number of intermediate values and the lifetimes of the values in the registers can be affected, which can have an effect on the schedule.

For example, if the consuming operations $n1$ and $n2$ in Figure 6.1a are assigned to respectively functional unit FU_0 and FU_1 in Figure 6.1b then multi-casting must be applied and two write ports must be used to store the intermediate result in a register in register file RF_1 and RF_3. This differs from the case that both consuming operations are assigned to one functional unit. In that case the intermediate result is stored in only one register file and only one write port is used. How the dependency between the assignment of operations and the write ports that must be used is modeled in the ASCG is explained in the next section.

6.2 Modeling of multi-casting in the ASCG.

Multi-casting is handled by our operation assignment technique as follows. In the case a result is consumed by N operations then N nodes of the type WP are introduced in the ASCG. These WP nodes are potentially colored with the same color. If the nodes have the same color then the same register file write port is used. The same write port can be used in the same cycle because one intermediate value is stored in the register file.

Figure 6.2 shows the ASCG for the DFG and data path in Figure 6.1. If operation $n1$ and $n2$ are both executed on FU_1 then it is case that the color "wp3" must be assigned to both WP nodes in this ASCG. In this case the result of operation $n0$ is stored in register file RF_3. If operation $n1$ and $n2$ are respespectively assigned to FU_0 and FU_1 then the CN nodes in

the ASCG will enforce that the only valid colors for left and right WP node in Figure 6.2 are "wp1" and "wp3" respectively. In this case the result of operation n0 is stored in register file RF_1 as well as in RF_3.

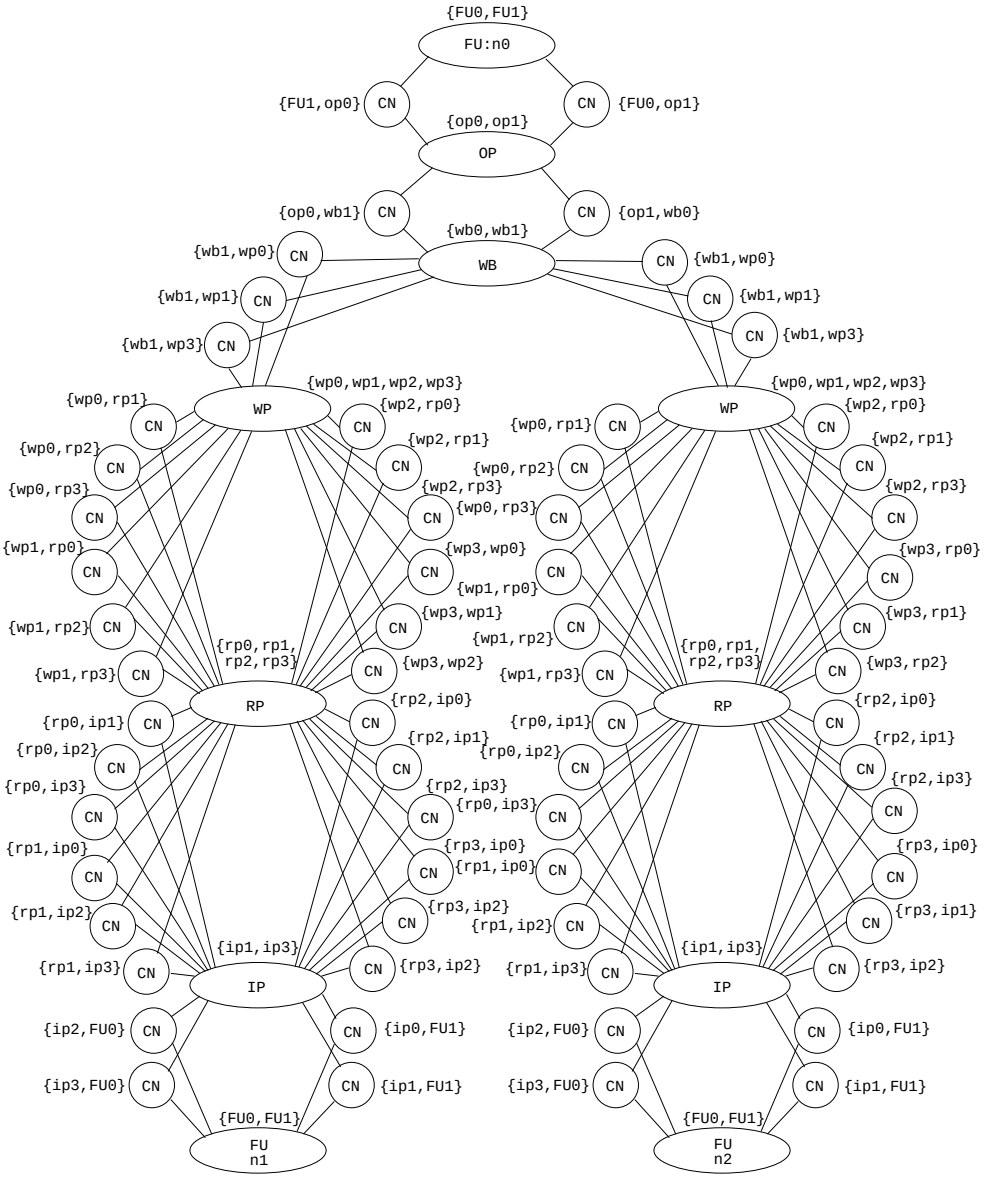


Figure 6.2: Assignment conflict graph for the DFG and data path in Figure 6.1.

The number of write ports that are used, depend on the assignment of the operations. Therefore, the schedule search-space is first pruned given the best-case assumption that there is

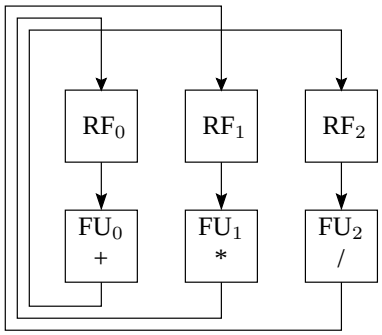


Figure 6.3: Data path example used to demonstrate the necessity of copy operation.

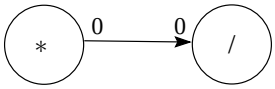


Figure 6.4: DFG which can not be executed on the data path shown Figure 6.3 because there is no connection between the output of functional unit FU_1 and the write port of register file RF_2.

one write port used per intermediate result. If after some operation assignment decisions it is detected that multi-casting must be applied then the usage of the additional write ports is taken into account during pruning of the schedule search-space.

6.3 Copy operations

In Figure 6.3 a data path is shown with an incomplete network. The DFG shown in Figure 6.4 can not be executed on this data path. The reason is the following: in this example the multiply operation can only be executed on functional unit FU_1 and the division operation must be executed on functional unit FU_2. Therefore, the result produced by functional unit FU_1 must be stored in register file RF_2 such that functional unit FU_2 can read the result. However, the network misses the appropiate connection.

The DFG shown in Figure 6.5 results in the same behavior as the DFG shown in Figure 6.4 but can be executed on the data path of Figure 6.3. A so-called copy operation is used to copy a value from a register file into another register file.

Copy operations are in the literature also called move or pass operations. We think that the name copy operation is more appropriate because it is possible that the value remains alive in the register file from which the value is copied. Opcode space can be saved by implementing copy operations with operations effectively just passing the input value. For example, the copy operation in the DFG of Figure 6.3 must be implemented as an operation that adds zero

to the input value.

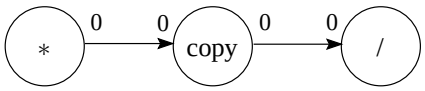


Figure 6.5: The DFG of Figure 6.4 but now with a copy operation such that this DFG can be executed on the data path shown in Figure 6.3.

A disadvantage of the use of copy operation is that they introduce an additional latency of at least one clock cycle and also occupy a functional unit during a clock cycle. This can lead to a schedule with a lower throughput or a longer latency. The next section describes an alternative for the copy operations.

6.4 Global write-back busses

An alternative for the use of copy operations is the use of a global write-back bus. A global write-back bus is a bus that provides a path from every functional unit output port to every register file. A data path with such a bus is shown in Figure 6.6. The combination of multi-casting and the use of a global write-back bus makes it always possible to store a result which is produced by a functional unit in *all* the register files from which the result is read.

Care should be taken that the global write-back bus does not become the critical resource. Therefore, a network should be applied that contains the appropriate local write-back busses. A local write-back bus is a bus that provides a path for only one functional output port to at least one register file write port.

6.4.1 Write-back bus assignment

During operation assignment, it is decided on which functional units the operations are executed. The assignment of operations determines implicitly which communication paths can be used for the transportation of the intermediate values from the functional unit output ports to the register file write ports. If there are several equivalent paths available then it must be decided which bus will be used. This decision problem is given the name *write-back bus assignment*. Write-back bus assignment is in general a difficult problem because the bus assignment determines the used resources, which potentially affects the schedule.

If there is only one global write-back bus in the data path then this bus must be used, if there is no communication path via a local write-back bus. In all other cases, a local write-back bus should be used. It is therefore simple to derive the communication bus assignment from the assignment of operations in the case only one global write-back bus is applied in the data path of a processor. Only this option is supported by assignment techniques that are implemented in FACTS.

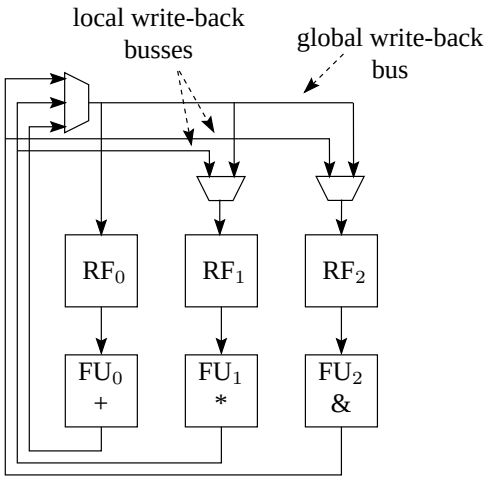


Figure 6.6: Processor with a global write-back bus. This bus guarantees that a communication path exists from every functional unit output port to every functional unit input port.

If there are several global write-back busses in the data path then also a choice must be made which global write-back bus should be used. Because all global write-back busses provide the same functionality, the write-back bus assignment can be postponed till the assignment of the operation to functional units is completed.

These write-back busses are not equivalent if a different subsets of the functional units can put a result on these busses. The subsets considered contain more than one functional unit and less than all the functional units in the data path. Write-back bus assignment becomes a difficult problem if there are several busses in the data path, which are not equivalent. In this case, several communication paths via different write-back busses can exist from a functional unit output port to a functional unit input port. In this case, a decision must be made about which bus will be used. An unfortunate decision can prohibit that another result is send in the same cycle to the appropriate register file. Therefore, unfortunate write-back bus assignment-decisions can cause violation of the timing constraints.

6.4.2 Scalability considerations

A potentially important disadvantage of the use of global write-back busses is that it limits the scalability of the data path of a processor. If the number of functional units in the data path is increased then at a certain point the delay of the global write-back bus will become dominant and will, to a large extend, determine the maximum clock frequency of the processor. The scalability of the data path can be enhanced by introducing a pipeline register into the global write-back busses. In this case, it should be taken into account that a result will be stored one cycle later in the register file if a global write-back bus is used. This communication latency can be modeled during operation assignment with sequence precedence edges. These

sequence precedence edges can be derived with the following rules:

Rule 3 If a global write-back bus must be used for the transport of a result from operation v_i to operation v_j then a sequence precedence edge (v_i, v_j) must be added with weight 2.

The following rules can be used for pruning of the assignment search-space:

Rule 4 If distance $-d(v_j, v_i)$ is 1 then a local bus must be used for the communication of a result from v_i to v_j.

Pipelined write-back busses are currently not supported in our implementation.

Chapter 7

Hierarchy, Operation Merging and the Decision Heuristic

In the previous paragraphs, only the techniques for the assignment of operations of inner basic blocks of a hierarchical DFG have been described. That more than one operation can use the same resource, if they do not have a resource conflict, has also not been considered. The applied heuristics, which determine which operation is assigned first during the assignment decision process, have not been described in the previous chapters. These issues are the topic of this chapter.

The organization of this chapter is as follows. First, in Section 7.1 we describe how hierarchical data flow graphs are handled. The techniques used to support operations that have the same type but do not have a resource conflict is described in Section 7.2. The applied decision heuristics are described in the last section of this chapter.

7.1 Hierarchical data flow graphs

Hierarchical reduction [Lam88] is the technique that is used to handle hierarchical data flow graphs. In this technique, the inner basic blocks are scheduled first. Then each inner basic block is represented as a block operation in the surrounding basic block. A block operation is an object similar to an operation in a basic block. The operations in the surrounding basic blocks are scheduled together with the block operations. This is repeated until all blocks have been collapsed in one block operation.

To be able to support hierarchical reduction a compact representation of a scheduled basic block as a block operation is needed. The applied representation is illustrated with the C-program in Figure 7.1. A data flow graph that represents the behavior of this C-program is shown in Figure 7.2. The edges drawn with dashed lines are loop carried data dependencies. A loop carried data dependency enters the same operation input port as an initialization value that is produced by an operation that belongs to a surrounding basic block.

```
void macloop(int x[100], int y[10], int z[10]){
  loopj:for(int j=0; j<10; j++){
    loopi:for(int i=0; i<100; i++){
      z[j]+=y[j]*x[i];
    }
  }
}
```

Figure 7.1: C program which is used to illustrate the representation of a scheduled inner basic block.

The operations that belong to a basic block are surrounded by a dashed box in Figure 7.2. The inner basic block BB1 corresponds to “loopi” in the C program. This basic block receives several initialization values from the surrounding basic block BB2. Basic block BB2 corresponds to “loopj” in the C program. After execution of “loopi” the result is stored in memory by the operation with the type “ST”.

After basic block BB1 is scheduled it can be collapsed in a block operation which is labeled in Figure 7.3 with “BB1”. This block operation should represent all constraints that should be taken into account during operation assignment, serialization and scheduling of the surrounding basic block BB2. These constraints are modeled with several dummy operations. The dummy operations occupy all the functional units in the data path during one cycle such that other operations can not be scheduled in parallel. The data edges which were connected to the block operation are connected to the dummy operations in such a way that the interconnect constraints are correctly modeled. Figure 7.4 shows a data flow graph in which the block operation of Figure 7.3 is replaced by dummy operations.

During the execution of the operations of an inner basic block, intermediate values are stored in registers. The number of registers that are used can not be captured in the model of a block operation that is shown in Figure 7.4. Figure 7.5 shows an example DFG in which the block operation is expanded in 2 times 4 dummy operations. This model allows that the usage of registers can be captured with data edges between dummy operations that are executed in two consecutive cycles. The 3 data edges between the dummy operations in this example model that 3 values are used during the execution of the basic block BB1. Two values are stored in register file RF1 and 1 value in register file RF2. This modeling technique enables reuse of the value lifetime serialization techniques. Serialization of an ordinary operation and a dummy operation in a block operation is equivalent to the serialization of an ordinary operation and all operations in the same nested basic block.

In the case dummy operations are used to model the inner loop then the schedule of the outer loop is adapted after scheduling. A gap is created at the place were the dummy operations of a block operation are located in the schedule. The size of this gap is equal to the number of *potentials* of the schedule of the block operation. This number of potentials is equal to the initiation interval of the schedule of the block operation if the schedule is folded and otherwise equal to the latency of this schedule.

A gap of the right size can be created in the schedule if all sequence edges do not have a

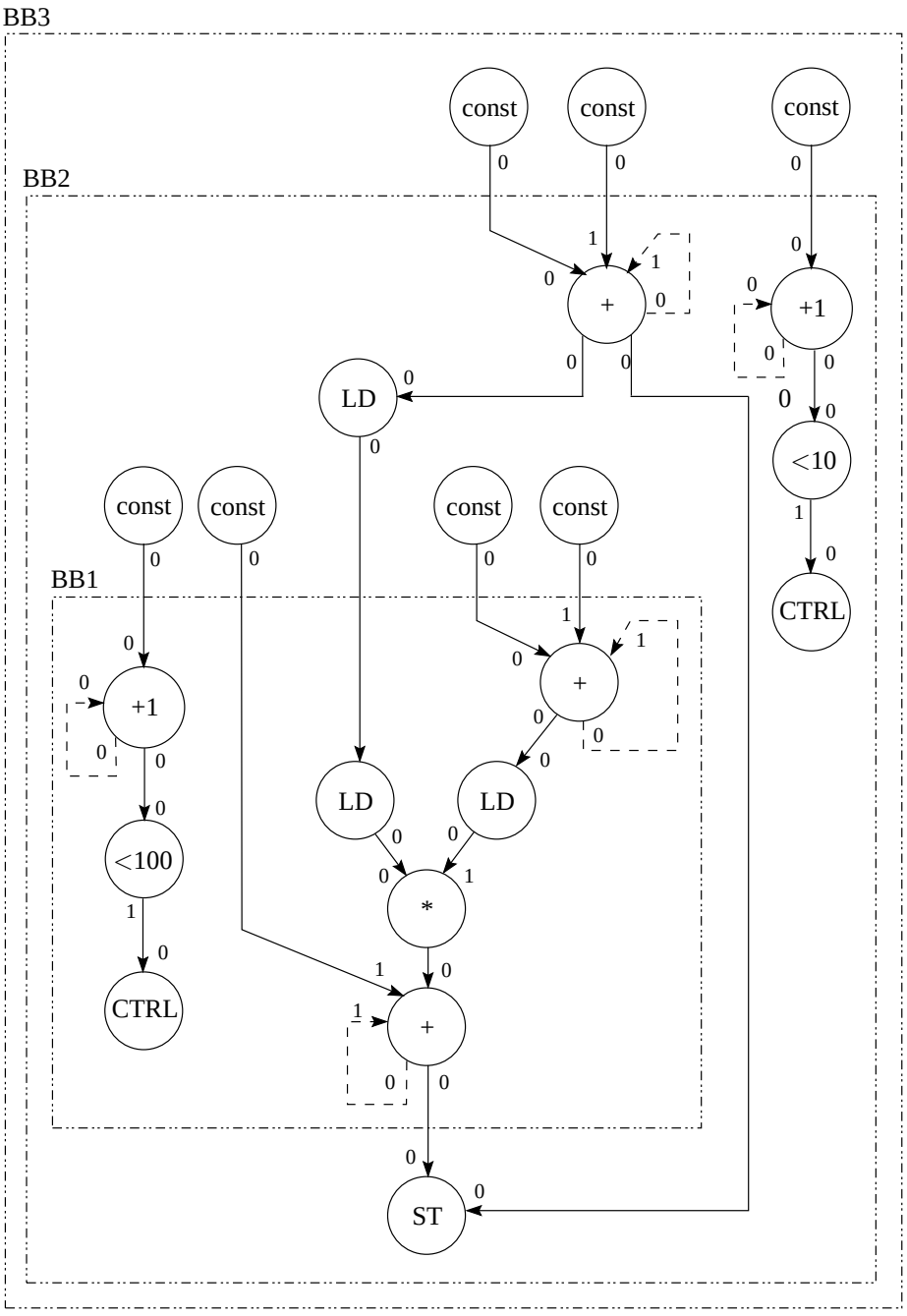


Figure 7.2: Hierarchy of nested basic blocks in a hierarchical data flow graph.

weight less then -1. In this case, a pair of operations that is constraint by a sequence edge with a negative weight is scheduled before or after the dummy operations. In this case the negative sequence constraints can never be violated by stretching of the schedule at the place of the dummy operations. Also sequence edges with a weight larger than one are not allowed because this could lead to violation of a sequence constraint if the schedule is shortened at the place of the dummy operations.

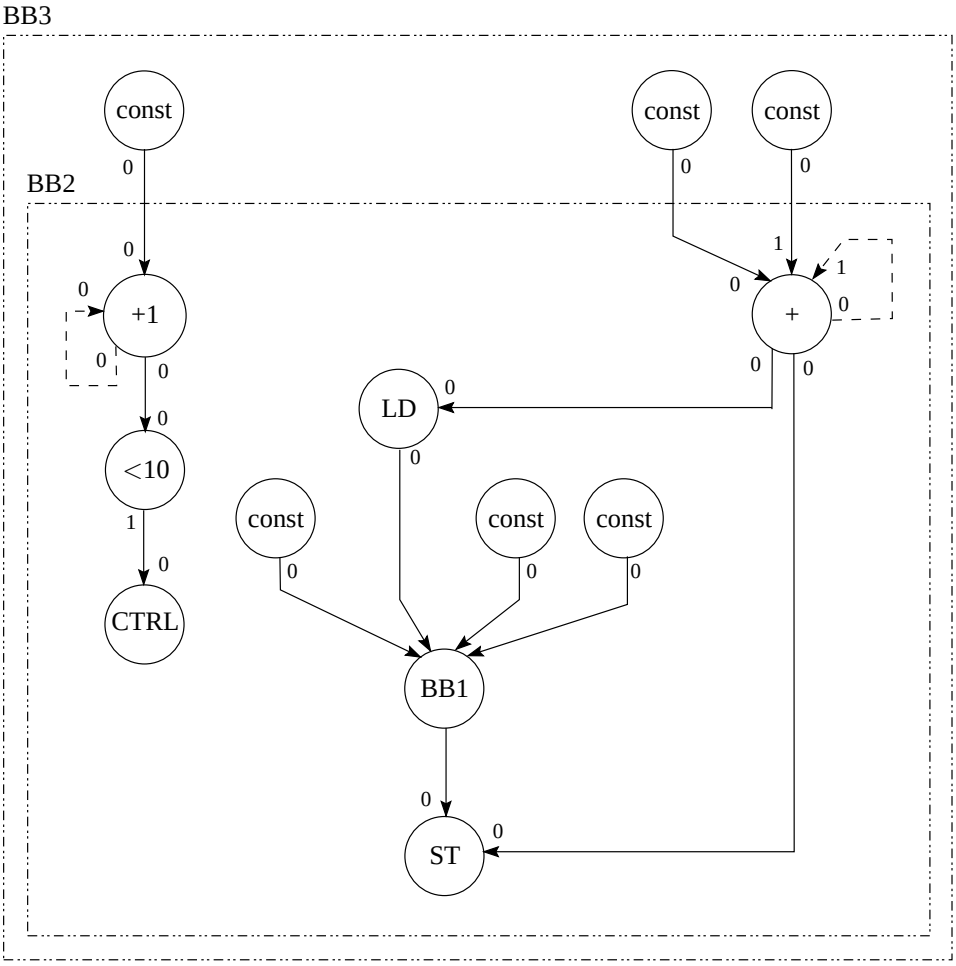


Figure 7.3: Hierarchy of nested basic blocks in which a basic block (BB1) is represented as a block operation.

The use of an incomplete network without a global write-back bus complicates operation assignment. It complicates assignment because during operation assignment it must be guaranteed that there is a communication path from a producing operation that belongs to a surrounding basic block to a consuming operation that belongs to a nested basic block. However, if the operations of the nested basic block are assigned and scheduled before the operations of the outer basic block then it is impossible to guarantee that the operations of the nested basic block are assigned such that the required communication paths exist. A similar problem occurs if the operations of a surrounding basic block are assigned before the operations are assigned of a nested basic block.

This problem is circumvented by the use of a global write-back bus in the connection network. This bus guarantees that at least one communication path exists in the data path from a producing operation in a surrounding basic block to a consuming operation in a nested basic block. This way unfortunate assignment decisions do not lead to infeasibility but only lead to a suboptimal schedule.

A similar to the above described problem occurs for the values that are produced by operations that belong to a nested basic block and are consumed by operations that belong to a surrounding basic block. Therefore, during the assignment of operations that belong to a nested basic block it is taken into account that the results of this nested basic block is stored in register files of which the read ports are connected to the input ports of functional units that executed the consuming operations. The existence as well as the availability of the used communication paths is taken into account during the assignment of the operations of the nested basic block. This is achieved by assigning the operations that belong to a surrounding basic block but consume results that are produced by operations that belong to the nested basic block together with the operations that belong to nested basis block. For example, for the assignment of the operations of basic block BB1 in Figure 7.2 a DFG is used that also includes the store operation (ST) that belongs to basic block BB2. Also the data edge from the plus operation to the store operation is included in this DFG.

An operation that belongs to a surrounding basic block can consume several results which are potentially produced by operations that belong to different nested basic blocks. Such an operation is assigned together with the operations of *one* of these nested blocks. During this assignment, only the existence and availability of a communication paths from the producing operations in one of the nested basic blocks are taken into account. However, the global write-back bus in the processor guarantees that there is at least one connection to the input ports of a functional unit on which the operation of the surrounding basic block is executed. Therefore, there is always a valid assignment of the operations of the other nested basic blocks.

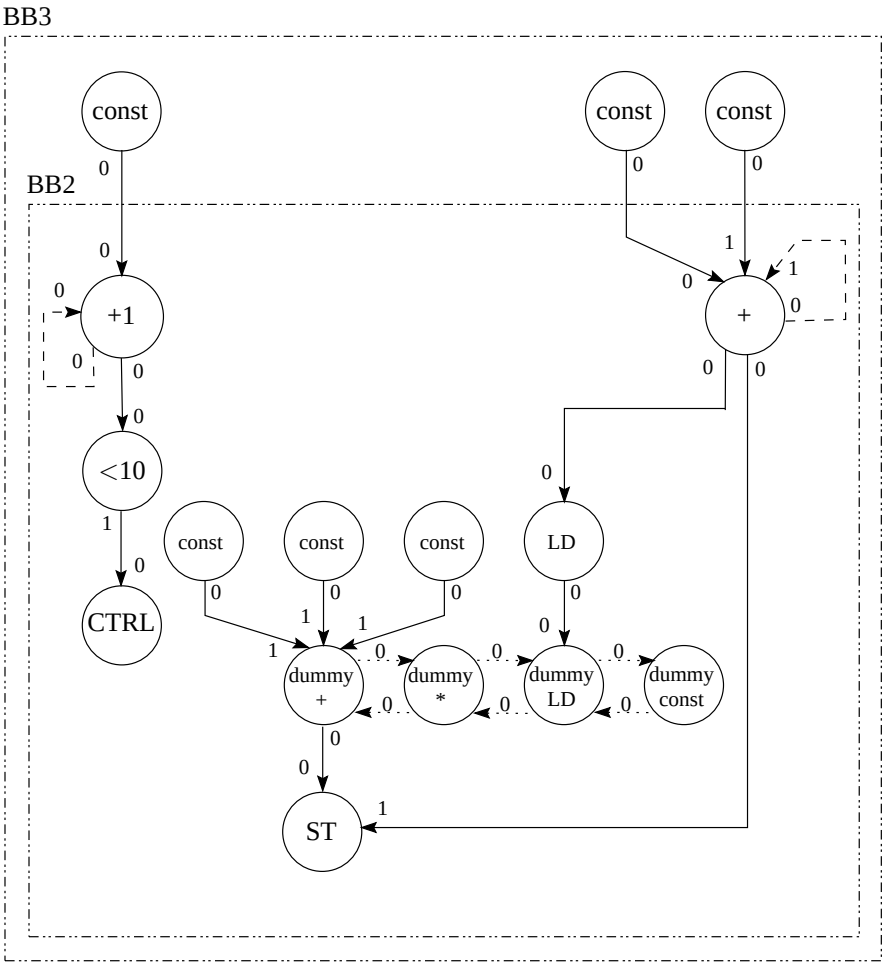


Figure 7.4: Use of several operations instead of a block operation to model basic block (BB1).

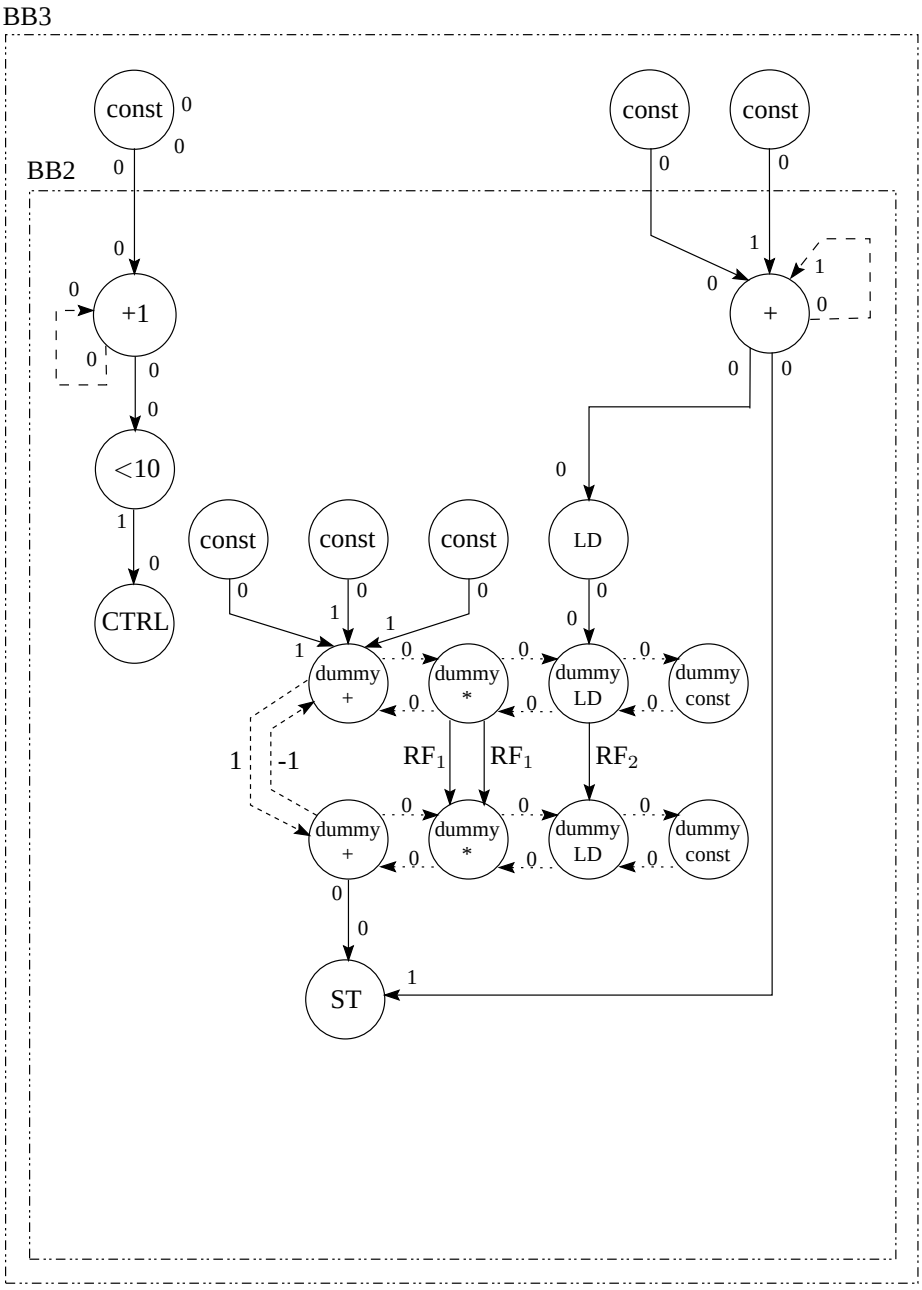


Figure 7.5: Modeling of the register usage in an inner basic block (BB1) with data dependency edges between dummy operations.

7.2 Operation merging

Operations are merged if they are executed in the same clock cycle on the same resource. Operations can be merged if they do not have a soft resource conflict. Merging of operation can reduce the schedule length or initiation interval. An example of operations that do not have a resource conflict are two read operations that read the same value from the same memory address. Another example is that two operations fetch the same value from a register file via the same register file read port.

Operation merging is handled with three different techniques. Each technique is suitable for a particular situation. These techniques are:

1. derivation of soft resource conflicts before operation assignment.
2. addition of virtual operations in the DFG.
3. adaptation of the type of the operations during operation assignment.

Soft resource conflicts between pairs of operations can be derived before operation assignment in the case that these operations must be executed on one specific functional unit in the data path. The derived soft resource conflicts are used by pruning rule 1 which is described in Section 3.1.1. The soft resource conflicts that we derive before assignment are the conflicts between the memory read operations that are executed on the same functional unit. These conflicts can be derived because the assignment of memory read operations is defined before operation assignment. All pairs of read operations that access the same addresses and are executed on the same functional unit do not have a soft resource conflicts.

The assignment of memory read as well as memory write operations is defined before operation assignment because the assignment of variables like arrays to memory banks must be defined before a DFG can be extracted from the C-description of the algorithm. A different assignment of an array to a memory bank can result in a different DFG because the operations needed to calculate the memory addresses can change. The assignment of an array to a memory bank forces usually that a set of memory read and write operations from different basic blocks accesses that particular memory bank.

In some cases it is possible to model resource conflicts by including virtual operations in the DFG before operation assignment, as is described in [Bra99] [Tim95] [ECR99] and [ZBJ+01]. This is possible only in the case that the assignment of operations to functional units can be permuted after scheduling because the connections to and from the functional units are equivalent. This is often not the case for processors with a distributed register file architecture and an incomplete network. Another necessary condition is that the existence of a resource conflict between a pair of operations may not depend on the assignment of the operations. An example in which this condition is not respected is shown in Figure 7.6. In this example a potential write port resource conflict prohibits that both ALU operations can be executed in parallel, if both RAM operations, which are executed in a different cycle, are assigned to the same RAM unit.

Another situation in which operation merging can occur, is illustrated with the DFG and the data path in Figure 7.7. The register file read port is used once if both operations in the

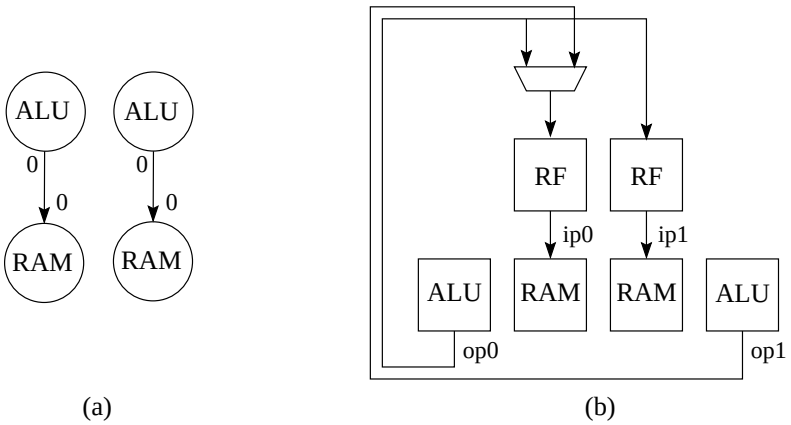


Figure 7.6: A DFG (a) and a data path snipped (b). If the RAM operations of the DFG are assigned to the same RAM unit in the data path then a soft write port resource conflict prohibits that both ALU operations can be executed in parallel.

DFG are executed in the same cycle, otherwise the read port is used twice. After operation assignment it is known which operations make use of the same read port. These potential resource conflicts are modeled by adding virtual operations after operation assignment in the DFG.

If multicasting is supported then the number of write ports that are used depend on the assignment of the operations as is illustrated with the DFG and the data path in Figure 7.8. A larger number of used write ports can result in additional resource conflicts. These write port resource conflicts cannot be modeled in advance with virtual resources because whether there are resource conflicts depend on the assignment of the operations. Merging of write port operations is supported by adapting the types of the operations during operation assignment as is described in Section 3.2.2. Adaptation of the type of the operations is possible because the number of used write ports depend only on the assignment of the operations and not on the schedule.

7.3 Decision heuristic applied during operation assignment

Operation assignment decisions are made during the operation assignment phase. Assignment decisions are taken for every operation for which there is after pruning more than one assignment option left.

During the operation assignment decision process the operations with the highest number of input ports and output port are assigned first. The motivation for this criterion is that these operations are relatively difficult to assign because they require the availability of the largest number of communication paths.

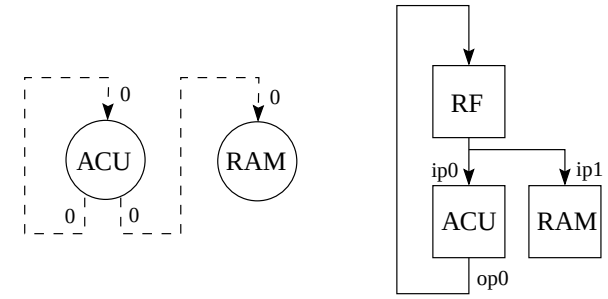


Figure 7.7: A DFG (a) and a data path snipped (b). If the ACU and RAM operation of the DFG read their input value in the same clock cycle then the read port is used only once, otherwise it is used twice.

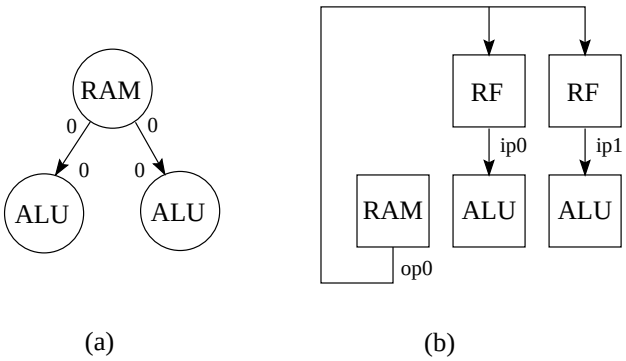


Figure 7.8: A DFG (a) and a data path snipped (b). If the ALU operations of the DFG are assigned to different ALUs then the result value of the RAM operation is stored in two register files and two write ports are used. If the ALU operations are assigned to the same ALU then one write port is used.

During the operation assignment decision process the functional units that are first selected are the units with the minimal number of operations already assigned to it. The motivation for this criterion is that this balances the computational load of the functional units. This is desirable because the initiation interval and the latency is equal or larger than the maximal number of operations that are assigned to the same functional unit.

Chapter 8

Quantitative Evaluation

In this chapter the results that are obtained with our code generator FACTS are presented. The outline of this chapter is as follows. In Section 8.1 the experimental compiler flow is presented. With this flow, the operation assignment techniques in FACTS were tested on realistic applications and processor data path instances. The results can be found in Section 8.2. The results that indicate whether given an operation assignment also a serialization and a schedule can be found, are described in Section 8.3. The schedules generated with FACTS are compared with the schedules generated with the A|RT-scheduler in Section 8.4. These results indicate the quality of the schedules that can be obtained after operation assignment and lifetime serialization.

8.1 Experimental compiler flow

The experiments were carried out with the compiler flow shown in Figure 8.1. This flow has been set up around the A|RT-designer tool suite of Adelante Technologies [Ade]. In this flow, C++-code is compiled into micro-code for a predefined VLIW processor. This flow provides a user-friendly way to generate test cases for FACTS. The flow also provides means to verify the results produced by FACTS.

In this flow C++ code according to the System-C [Sys] standard is translate by the A|RT compiler frontend into a hierarchical DFG which is stored in a file with the name “alg.rtg”. Another input of the A|RT compiler frontend is the “architecture.pra” file. This file contains a description of the data path of the VLIW processor. This description is also put by the compiler frontend into the “alg.rtg” file.

The “alg.rtg” is read in by a tool called “RT2DFG” which extracts a basic block. The extracted basic block has the name “basic block name”. The operations of this basic block together with the (loop-carried) data dependencies and sequence precedence edges are modeled in a DFG. This DFG is stored in the file “rt2dfg.dfg”.

A tool called RT2MD extracts from the “alg.rtg” file a description of the processor and writes

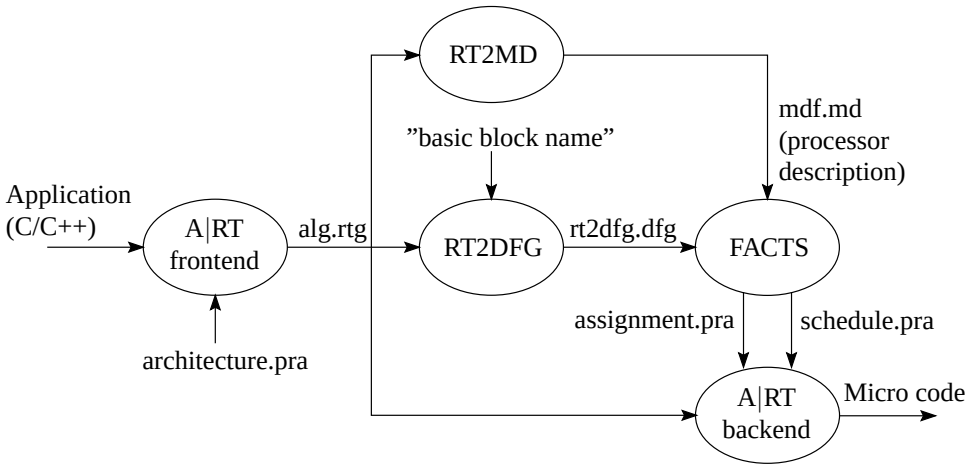


Figure 8.1: Experimental compiler flow.

it in the machine description file “mdf.md”.

The DFG in the “rt2dfg.dfg” file together with a description of the target processor in the “mdf.md” is read in by FACTS. An operation assignment and a schedule is generated by FACTS and stored in two files. In the “assignment.pra” file the assignment of operations is stored and in the “schedule.pra” file the start moments of the operations relative to the start of the basic block are stored. The register binding can be computed in a straightforward way from the assignment and the schedule in the A|RT backend.

The “assignment.pra” and “schedule.pra” file together with the “alg.rtg” are processed by the A|RT backend. In the A|RT backend the operations are assignment as specified in the “assignment.pra” file and the basic block is scheduled as specified in the “schedule.pra” file.

8.2 Evaluation of the operation assignment techniques

The operation assignment techniques have been tested on various algorithms and architectures. Before the quantitative results are presented in Section 8.2.2, two operation assignment experiments are described in more detail. The focus in the first experiments is on assignment in combination with hierarchy in the data flow graph. The second example illustrates that operation assignment can even be difficult for processors with a fully connected network and a distributed register file architecture.

8.2.1 Operation assignment examples

The C++ program in Figure 8.2 describes an algorithm that computes an Inverse Modified Discrete Cosine Transform (IMDCT) [Nol97]. This algorithm is translated by the compiler

frontend into a hierarchical DFG and mapped on the data path that is depicted in Figure 8.3.

The data path of Figure 8.3 contains 20 functional units of which 3 are equivalent ALUs, 4 are equivalent MAC units, and 5 are equivalent ACUs. The incomplete connected network was obtained by synthesis with A|RT-designer. The global write-back bus was added after synthesis was completed.

The objective of the experiment was to test whether an assignment and a schedule could be found for this data path with an incomplete network. Therefore all information about the assignment of the operations, which was specified by the programmer, was deleted afer the processor was synthesized. The throughput of the schedule should at least be the same as the throughput of schedule obtained with A|RT-designer.

During code generation, the operations of the deepest nested basic block in the hierarchical DFG are assigned and scheduled before operations of surrounding basic blocks are assigned and scheduled. In the C++ program the deepest nested basic block is the for-loop with the label “inner”. During the assignment of the operations of this loop care is taken that the results t0, t1, s0, s1 are stored in a register file such that these results can be, without any additional copy operations, used outside the inner basic block. This is achieved by assigning the operations that implement the statements r0 = s0 + t0 and the r1 = s1 + t1, together with the other operations of the loop “inner”.

After the inner basic block is assigned and scheduled, the operations of the surrounding basic block “outer” are assigned and scheduled. Then the operations of the function “imdct36” are assigned and scheduled.

The resulting schedule is shown in Figure 8.4. The for-loop “inner” is folded with an initiation interval of 2 cycles. An initiation interval of one cycle is not possible because some of the functional units, like for example inport_1, must be used twice. This load diagram also shows that the global write-back bus was used two times in the for-loop “inner”. Therefore this assignment is different than the assignment used during synthesis of the processor because this bus is added after synthesis. The outer-loop and the imdct function are not folded in the schedule of Figure 8.4.

In the second example a Viterbi algorithm is mapped on a data path with a fully connected network which is shown in Figure 8.5. Such a fully connected network is also assumed during processor synthesis in A|RT-designer. There are 4 equivalent Add Compare Select (ACS) functional units in the data path. Each ACS-unit has four output ports.

The obtained schedule is shown in Figure 8.6. All ASCs units are fully utilized during the inner-loop. During operation assignment there were 24 assignment decisions and 5 backtracks made. That 5 backtracks were needed indicates that even for a processor with a fully connected network operation assignment is difficult. The backtracks can not be a result of a missing connection in the network but must a result of resource conflicts, like register file write port conflicts, which were detected by constraint analysis.

```
#define N 36
#define M 18

typedef Fix<32, 28> T_IN;
typedef Fix<64, 56> T_OUT;

T_IN imdct_cos0[N*M], imdct_cos1[N*M], imdct_cos2[N*M],
     imdct_cos3[N*M];

void imdct36(const T_IN in0[M], const T_IN in1[M],
             T_IN out[N])
{
  int i0 = 0, i1 = 18*M, i2 = 1, i3 = i1+i2;
  int p0 = 0, p1 = 18, p2 = 17, p3 = 35;

  outer: for (int l1 = 0; l1 < 9; l1++)
  {
    int m0 = 0, m1 = 9;
    T_OUT s0 = 0, s1 = 0, t0 = 0, t1 = 0;
    T_IN x0, x1;

    inner: for (int l0 = 0; l0 < 9; l0++)
    {
      x0 = in0[m0];
      T_IN c1=imdct_cos0[i0];
      T_IN c2=imdct_cos1[i1];
      s0 += x0*c1; s1 += x0*c2;

      x1 = in1[m1];
      T_IN c3=imdct_cos2[i2];
      T_IN c4=imdct_cos3[i3];
      t0 += x1*c3; t1 += x1*c4;

      m0++; i0++; i1++; m1++; i2++; i3++;
    }

    T_IN r0, r1;
    r0 = s0 + t0;
    r1 = s1 + t1;
    T_IN r2 = -r0;

    out[p0] = r0; out[p1] = r1; out[p2] = r2; out[p3] = r1;
    p0++; p1++; p2--; p3--;
  }
}
```

Figure 8.2: C++ description of an IMDCT algorithm.

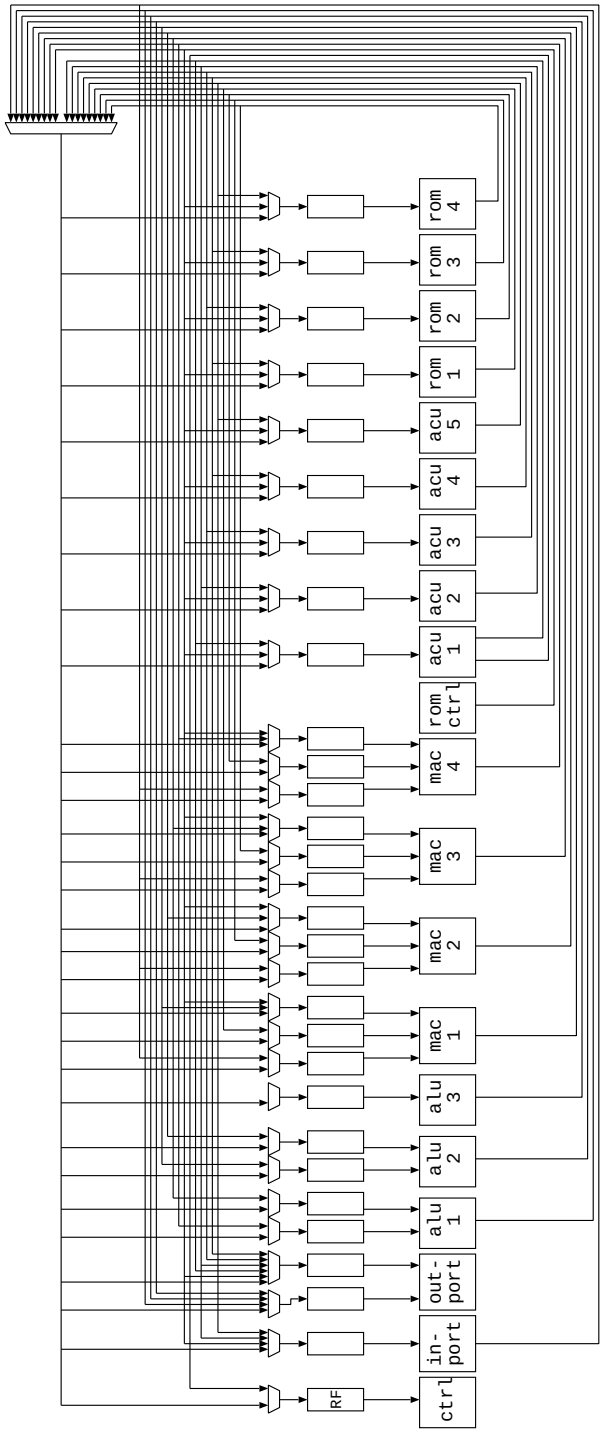


Figure 8.3: Data path of a VLIW-processor optimized for the IMDCT function.

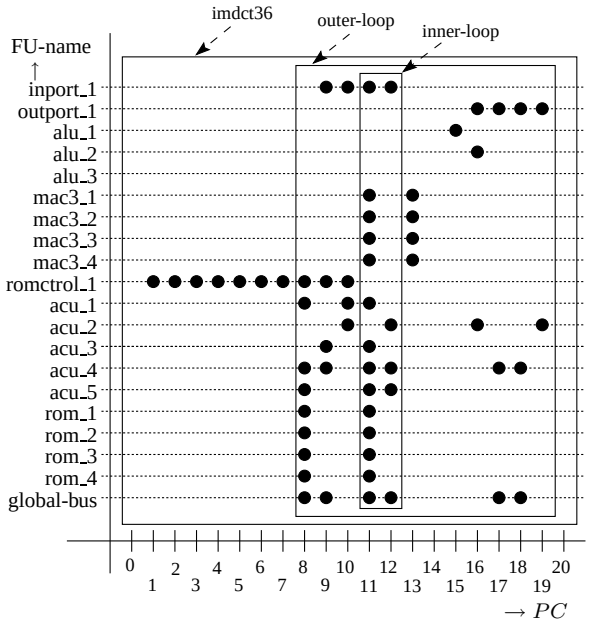


Figure 8.4: Schedule of the IMDCT algorithm of Figure 8.2 mapped on the data path of Figure 8.3.

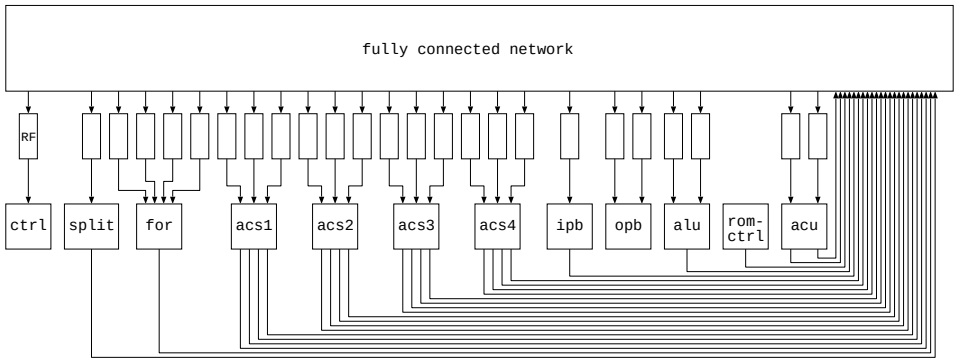


Figure 8.5: Data path of an VLIW processor which is optimized for a Viterbi algorithm.

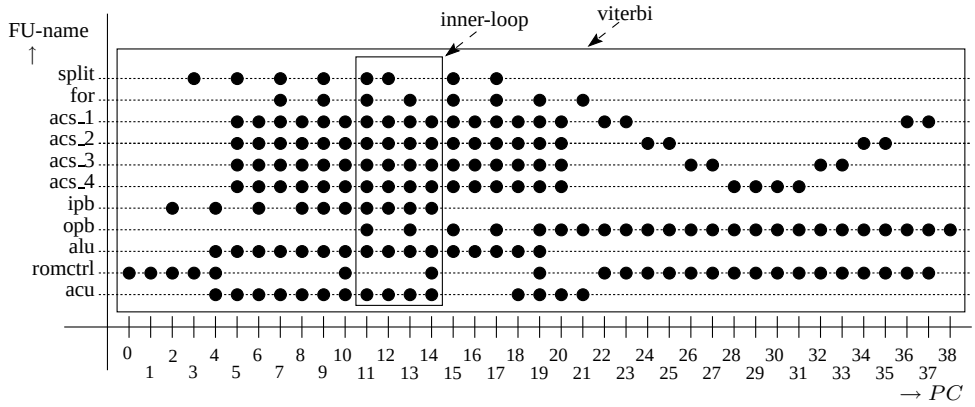


Figure 8.6: Schedule of the Viterbi algorithm.

8.2.2 Assignment results

Quantitative results of operation assignment experiments are presented in this section. The results of these experiments can be found in Table 8.1, Table 8.2 and Table 8.3. For each algorithm in Table 8.1 the number of operations (V) and data edges (E_d) of an inner basic block are presented. Each inner basic block is a loop that is mapped on a processor that was synthesized for the same algorithm with A|RT-Designer. These processors contain an incomplete network and do not have a global write-back bus. In all cases an assignment and a schedule were found with an initiation interval (II) equal to the lower bound of the initiation interval (l.b. II). Less than 8 decisions (d.) and 1 backtrack (b.) were needed to obtain an assignment of all the operations in a basic block. The time needed to obtain these assignments was for all test cases less than 30 seconds on a 500 MHz Pentium III processor.

These results indicate that with the constraint driven operation assignment techniques high quality schedules can be obtained. The run time indicates that construction and pruning of the assignment conflict graph which has potentially a large number of nodes (> 1000) does not lead to excessive run times.

The results in Table 8.2 show how many assignment decisions and backtracks were needed to map the same DFGs of Table 8.1 on similar data paths with the same functional units but with a fully connected network. For convenience the number of operations (V) and data edges (E_d) in the DFGs, and the number of decisions and backtracks in the case of an incomplete network are repeated in this table.

The results in Table 8.2 indicate that fewer operation assignment decisions and backtracks are needed in the case a processor with an incomplete network instead of a complete network is targeted. From this is can be concluded that there is less assignment freedom in the case processors with an incomplete network are targetted and that pruning removes in this case a larger number of infeasible operation assignment options from the operation assignment search-space.

no.	algoritm name	$\mid V \mid$	$\mid E_d \mid$	II l.b.	II	d.	b.	time s
1	andblock	5	6	2	2	0	0	0.02
2	ASU-ex	7	8	1	1	0	0	0.01
3	fir2	8	10	2	2	0	0	0.08
4	mac-loop0	10	12	2	2	0	0	0.1
5	mac-loop2	14	18	2	2	4	0	0.5
6	biquad	14	19	7	7	0	0	0.3
7	bit-packer	14	22	1	1	0	0	0.1
8	imdct	23	41	4	4	0	0	0.9
9	viterbi	49	117	4	4	3	1	28
10	FFT-radix4	53	56	8	8	8	0	5

Table 8.1: Operation assignment results for data paths with an incomplete connection network without global write-back bus.

The operation assignment results in Table 8.3 are for processors with an incomplete network with a global write-back bus. Note that this global write-back bus is a fallback. It should only be used as little as possible.

The first 3 basic blocks in the table are from the IMDCT program which is described in Section 8.2.1 and are mapped on the data path of Figure 8.3. The other basic blocks are from an MPEG2 layer III audio encoding algorithm. This algorithm is mapped on a VLIW processor with 34 functional units including 6 MAC-units, 6 dual port memories, 4 Address Calculation Units (ACUs) and 2 Arithmetic and Logic Units (ALUs). All functional unit consume 32 bit integer values.

From Table 8.3 it can be concluded that basic blocks with up to 287 operations ($\mid V \mid$) can be assigned on a VLIW with a highly parallel data path in less 300 seconds run time. Pruning seems to be quite effective because the number of backtracks is small (< 15) and the number of assignment decisions is usually much less than the number of operations in the basic blocks. A schedule can be obtained, in all test cases, with a Initialization Interval (II) equal to the lower bound (l.b.) II and a Latency (L) which is maximally 3 cycles longer than the lower bound latency L.

The in Table 8.1, 8.2 and 8.3 presented results are for the case that after operation assignment the schedule search-space is updated with information from the assignment search-space and vice versa. Table 8.4 presents also results for the situation that operation assignment and scheduling are performed completely independent of each other.

The results in the Table 8.4 indicate that the number of assignment decisions and backtracks increases in the case that the phases are treated independently of each other. In this case, for three out of four benchmarks no schedule was found. An explanation for the results of test cases number 2 an 3 in Table 8.4 is that an initiation interval of 1 cycle implies that all operations are scheduled in the same clock cycle. In the case the assignment and schedule phases are coupled then an edge (see Section 4.3.2) is included in the ASCG between all pairs of nodes that correspond to operations of the same type. These edges improve the pruning of

no.	algorithm name	$\mid V \mid$	$\mid E_d \mid$	d. i.n.	b. i.n.	d. f.c.n	b. f.c.n
1	andblock	5	6	0	0	0	0
2	ASU-ex	7	8	0	0	0	0
3	fir2	8	10	0	0	3	0
4	mac-loop0	10	12	0	0	0	0
5	mac-loop2	14	18	4	0	4	0
6	biquad	14	19	0	0	0	0
7	bit-packer	14	22	0	0	3	1
8	imdct	23	41	0	0	14	0
9	viterbi	49	117	3	1	24	5
10	FFT-radix4	53	56	8	0	8	0

Table 8.2: Comparison of the number of operation assignment decisions and backtracks for data paths with an incomplete network (i.n.) and for data paths with a fully connected network (f.c.n.).

the assignment search-space significantly. An explanation for the fact that no schedule could be obtained after operation assignment for test case number 4 in Table 8.4 is that the applied bound on the latency is usually too tight.

no.	basic block name	$\mid V \mid$	d.	b.	II l.b.	L l.b.	II	L	run time (s)
1	inner	24	12	4	2	6	2	6	9
2	outer	82	5	0	-	12	-	12	11
3	wrapper	49	0	0	-	10	-	10	0
4	sbai_unroll1_fold	22	8	7	1	3	1	3	27
5	sba_idct_loop	147	15	13	-	16	-	16	269
6	sba_butterfly_loop	20	7	5	-	5	-	7	10
7	sbaw_unroll1_fold	31	5	0	1	5	1	5	12
8	enc_sbaw_outer	193	8	6	-	16	-	16	177
9	_loop2	43	3	5	-	6	-	6	6
10	_if6	287	6	0	-	36	-	36	55
11	_if6	2	0	0	-	2	-	2	0
12	_if5	72	0	0	-	7	-	7	0
13	mdct_writeback_fold	18	0	0	1	2	1	2	0
14	_loop1	27	8	5	-	8	-	8	22
15	mdct_alias_reduction	70	8	2	-	8	-	8	2
16	mdctm_unroll1_fold	30	15	0	3	3	3	5	34
17	mdct_main_loop	153	17	15	-	15	-	15	197
18	mdct_prepro_loop	22	13	0	-	9	-	9	85
19	_if5	285	8	0	-	26	-	29	65
20	_if4	68	0	0	-	7	-	7	0
21	_loop0	21	7	0	-	7	-	7	0
22	_if4	64	0	0	-	6	-	6	1
23	cop	65	0	0	-	8	-	8	0

Table 8.3: Operation assignment results for target processors with a global write-back bus.

no.	basic block name	II	L	d. c.	b. c.	run time c. (s)	d. dec.	b. dec.	run time dec. (s)	schedule
1	inner	2	1000	12	7	10	34	108	17	Y
2	sbai	1	1000	2	2	16	16	54	68	N
3	sbaw	1	1000	1	0	6	14	34	52	N
4	sba	-	16	13	3	64	13	0	46	N

Table 8.4: Comparison of operation assignment decisions (d.) and backtracks (b.) for the case that operation assignment is performed coupled (c.) with or decoupled (dec.) from the schedule phase.

8.3 Lifetime serialization after operation assignment

The assignment of an operation to a functional unit determines implicitly the register file read ports through which its input values are read. Because these read ports belong to a register file also the register file binding is implicitly determined by the assignment of the operations. This register file binding determines which values are stored in the register file. An unfortunate register file binding can prohibit that a serialization of the value lifetimes is found which guarantees a correct value to register binding after scheduling. Therefore experiments were carried out, with the objective to test how often sufficient schedule freedom remains for lifetime serialization given the operation assignment and register file binding.

The results of the lifetime serialization experiments are presented in Table 8.5. The data path of the target processors, including the number of registers in each register file, were obtained by synthesizing a processor for the same algorithm with A|RT designer. For each algorithm, the number of operation (V) and data edges (E_d) of the tested inner loop are specified. Also the timing constraints are specified in the form of the initiation interval (II) and the latency (L) given this initiation interval. Whether a lifetime serialization and a schedule given the assignment of the operations was found, is specified in the 7th and 8th column in the table. The sum of the registers in all the register files needed after processor synthesis with A|RT is given in the 9th column. The total number of registers needed after lifetime serialization is given 10th column. The number of serialization decisions and backtracks and the run-time of the serialization algorithm are given in the last 3 columns of the table.

The results in this Table 8.5 indicate that often, but not always, a serialization and a schedule could be found. In the case no serialization and schedule can be found, some values must be temporarily saved in the background memory by spill and restore operations. A strategy for the insertion of spill and restore operations, as described in for example [Cha82] and [BDEO97], has not yet been implemented in FACTS.

In example no 11 of Table 8.5 a serialization of the lifetimes was derived but no schedule could be found. A reason could be that the conservative estimate of the schedule, which is used during serialization, was not accurate enough. In this case it could be that there does not exist a schedule given the sequence constraints introduced by serialization.

The results in Table 8.5 also indicate that usually the same number of registers after serialization as after synthesis were needed. An explanation is that the capacity constraints obtained with synthesis are tight. Another explanation is that the objective of serialization is satisfaction of the register file capacities and not minimization of the number of used registers. Therefore no additional sequence edges are introduced by serialization as soon as it is guaranteed that the register file capacities will be respected after scheduling.

no.	algo. name	$\mid V \mid$	$\mid E_d \mid$	II	L	ser.	sch.	regs A|RT	regs FACTS	d.	b.	time (s)
1	andblock	5	6	2	2	Y	Y	7	7	0	0	0.0
2	ASU-ex	7	8	1	3	Y	Y	9	9	0	0	0.0
3	fir2	8	10	2	2	Y	Y	12	12	0	0	0.0
4	macloop	10	12	2	2	Y	Y	10	10	1	0	0.1
5	biquad	14	19	7	9	Y	Y	19	18	0	0	0.0
6	bitpack	14	22	1	1	Y	Y	11	11	0	0	0.0
7	adapt	25	30	8	8	Y	Y	20	20	2	0	0.2
8	crashL0	29	29	14	15	Y	Y	7	7	34	23	27
9	viterbi	49	117	4	4	Y	Y	56	56	2	0	2.1
10	FFT-R4-l0	53	56	4	14	Y	Y	33	33	8	0	5.0
11	FFT-R4-l3	83	105	8	19	Y	N	58		15	0	16

Table 8.5: Register serialization results for register file capacities derived during synthesis with A|RT-designer.

8.4 Scheduling results

In this section the throughput or latency of the schedules obtained by FACTS are compared with the schedules obtained with the scheduler in A|RT-Designer. The results give an indication of the quality of the schedules that can be obtained after operation assignment and lifetime serialization with the scheduler in FACTS.

The comparison is made with the scheduler in A|RT which operates as a list scheduler, with some backtracking capabilities in the case the latency is minimized. The scheduler makes use of an iterative folding technique [Goo89] in the case the initiation interval is minimized.

Schedules were generated for 7 signal processing applications that were mapped on an application domain specific VLIW processors. The signal processing applications are an MPEG2 layer III audio encoder and decoder, a 256 point FFT, a Viterbi error corrector, a Turbo decoder, and a bitpacker for MPEG2 video encoding. These applications are highly parallel, and have an average ILP of 32 operations in the inner loops. However, there are on average only 3 operations scheduled in a VLIW instruction in the basic blocks of these applications. The throughput of the with the FACTS scheduler folded inner loops of the 7 applications in the benchmark set is in 88% the same as the throughput obtained with the A|RT-scheduler. The initiation interval of the other 12% of the folded loops is 1 cycle shorter. The latency of the schedules generated with FACTS is in 16% of the cases a few cycles shorter and 11% of the cases a few cycles longer than the schedules generated with A|RT.

The presented scheduling results are for the basic blocks in a hierarchical DFG representation of the application. The assignment of the operations to the functional units was assumed as given and an infinite register file capacity was assumed. Lower bound timing estimates

were used as initial timing constraints. These timing constraints were relaxed in the case the FACTS scheduler couldn't find a valid schedule within 100 scheduling decision backtracks.

Operations that read the same memory location can be scheduled in the same cycle on the same resource by the FACTS scheduler. All other operations have a resource conflict in FACTS. The A|RT scheduler may schedule also other operations in the same cycle on the same resource if this does not cause a resource conflict. An example is that the same values are added by two addition operations in the DFG. In this case the add operations might be scheduled in the same cycle by the A|RT scheduler. The FACTS as well as the A|RT scheduler support that a value can be retrieved from a register file and used as an input in the same cycle by more than one functional unit.

That all precedence and resource constraints were respected in the FACTS schedules is verified by importing the schedule as specified in the "schedule.pra" file in A|RT.

The scheduling results of all the basic blocks of the MPEG2 layer III decoder application are presented in Table 8.6. This table contains for each basic block of the decoder application it's name, whether it is folded or not, the number of operation in the basic block, the runtime of the FACTS scheduler, and the obtained initiation interval (II) in the case folding is applied and otherwise the latency (L) of the schedule of the FACTS and the A|RT scheduler. The indentation of the basic block names in the table reflects the nesting of the basic blocks. The number of operations include ordinary operations which are executed on a functional unit and block operations that represent nested basic block. The reported latency includes the latency of nested block operations.

basic block name	folded	$\mid V \mid$	runtime (s)	II/L FACTS	II/L A|RT
cop	N	14	7	85	81
_switch0	N	6	6	74	74
_switch0	N	318	274	58	59
imdct_loop_outer	N	156	47	10	11
imdct_loop_inner	Y	50	7	2	2
_switch0	N	44	6	72	72
_switch0	N	4	5	1	1
_switch0	N	60	10	70	70
_loop1	N	58	9	61	61
_loop2	N	325	409	55	55
sbs_dct_outer	N	142	142	8	8
sbs_dct_inner	Y	42	15	1	1
sbs_win_outer	N	169	7	11	11
sbs_win_inner	Y	51	59	1	1
sbs_wb	N	47	7	7	7
_switch0	N	6	7	9	7
_loop0	N	14	6	3	3

Table 8.6: Scheduling results of the FACTS scheduler and the A|RT scheduler for an MPEG2 layer III decoder application.

The average runtime as function of number of operations in the basic block is presented in Table 8.7. The results in this table indicate that the runtime of the FACTS scheduler grows rapidly with the size of a basic block.

Number of operations in basic block	Average runtime (s)
0 .. 49	4
50 .. 99	15
100 .. 199	125
200 .. 400	341

Table 8.7: Average run time of the FACTS scheduler for different basic block sizes.

Chapter 9

Conclusion

This thesis describes code generation techniques -and in particular operation assignment techniques- for embedded application domain specific VLIW processors. The characteristics of the data paths of these VLIW processors are defined by means of a data path template. Processors with multiple register files and an incomplete communication network are supported because these processors are power efficient as well as scalable.

Our code generation techniques make use of constraint analysis. Constraint analysis is preferred because traditional code generation techniques require too much help of the programmer to satisfy all relevant constraints simultaneously. By using the constraints to prune the search-space of the code generator, the consequences of a decision for future decisions are made visible and provide a kind of look-ahead. Pruning is based on relatively simple rules. Accumulation and interaction of these rules can result in a large reduction of the search-space. After every decision, new constraints are added which potentially lead to additional pruning. The search-space is, for run-time efficiency reasons, incrementally updated by the pruning algorithms instead of being recomputed from scratch.

Most of the described techniques have been integrated in the research code generation tool FACTS [EMT99]. FACTS has been interfaced with the commercial VLIW processor synthesis and programming tool suite A|RT-designer from Adelante Technologies [Ade]. This coupling enabled evaluation of the code generation techniques on industrially relevant benchmarks.

The main focus of this thesis is on operation assignment techniques. Operation assignment is a difficult task in the code generator because an incomplete communication network and multiple register files are supported by the VLIW-processors template. A global write-back bus has emerged as an indispensable item. This bus connects every functional unit output port with every register file. Also multi-casting must be supported, which allows distribution of a result value into multiple register files. The global write-back bus in combination with multi-casting eliminates the need for operations that copy a value from one register file into another register file. This simplifies operation assignment significantly and is essential for the described operation assignment techniques.

The entire set of operation assignment options is modeled in one conflict graph. If a coloring of this conflict graph with a predetermined number of colors exists then the required connections are available in the network of the processor. Colors which can not be assigned to conflict graph nodes because of a combinations of constraints, are removed by pruning rules.

Code generation is split in an operation assignment phase, a lifetime serialization phase and a schedule phase. A tight coupling between these phases is obtained by pruning the schedule search-space after every operation assignment decision and after every serialization decision. The effect of an assignment decision on the schedule search-space is taken into account with dynamic resources. The type of these resources is potentially adapted after every assignment decision. Adaptation of a resource type can result in a reduction of the schedule search-space.

A machine description file contains an abstract description of the target VLIW-processor. This file constitutes the first input to the code generator. The other input of the code generator is the description of the application in the form of a hierarchical data flow graph.

Complete applications can be handled with the described code generation techniques. The behavior of these applications is expressed in a hierarchical data flow graph. After the operations of a nested basis block are assigned and scheduled this basic block is represented as a block operation in the surrounding basic block. The applied block operation model represents all the data routing, timing and storage constraints that should be satisfied during the assignment and scheduling of the operations of the surrounding basic block.

The results obtained showed, for our benchmark sets, that in most cases an assignment, a lifetime serialization and a schedule were obtained in a few minutes. Some cases failed because of the imperfect coupling of the code generation phases. An insufficient number of registers in a register file was another important cause of failure. This should be solved in the future by “spilling” some values temporarily in the background memory.

With the emergence of constraint driven code generation techniques the scheduling results have become more predictable and easier to analyze. We expect therefore that it will become more rewarding for the programmer to rewrite his program in such a way that the schedule quality improves. An example of such a program transformation is the removal of unnecessarily long value life times which prevent more aggressive folding of inner loops. In the future we expect that also these behavior preserving program transformations will be automatically performed by the compiler frontend.

Currently, techniques to cope with instruction encoding constraints are under development at Eindhoven University of Technology and will be integrated into the FACTS tool. The FACTS tool has already been used to program VLIW processors in the ERC and Champ project in Philips Research. The FACTS tool will be reengineerd in a new organizational entity in Philips Research which will hopefully become a new company. Support of a global write-back bus, which is proposed in this thesis, has already been added to the A|RT-designer tool suite.

Appendix A

Constraint Graph Representation

An annotated conflict graph can be represented in a constraint graph [Mac77] [DP88]. Therefore, an assignment conflict graph which is a special case of an annotated conflict can also be represented in a constraint graph. This constraint graph representation can be made extremely compact compared to the original assignment conflict graph by making use of a special properties of the assignment conflict graph. Besides that, the constraint graph representation also enables additional pruning. The constraint graph representation is presented in an appendix because the thesis was almost finished when this representation was found in literature.

The structure of this appendix is as follows. First, the constraint graph is defined and it is presented how an annotated conflict graph can be modeled in a constraint graph. Then it is shown how the number of nodes and edges of a constraint graph can be reduced while preserving the essential information in the constraint graph. Finally, the addition pruning rule is described.

A constraint graph is defined in as follows:

Definition A.1 (Constraint Graph.)
A constraint graph is an undirected graph represented by a tuple (V_{cg}, E_{cg}, R), where

- V_{cg} is a set of vertices which represent variables. With every vertex $v_p \in V$ a finite list L_p of values is associated which is the domain of the variable.
- $E_{cg} \subseteq V_{cg} \times V_{cg}$ is a set of edges.
- R is a set of relations. With every edge $e = (v_q, v_s) \in E_{cg}$ a relation $R_{q,s}$ is associated. The relation $R_{q,s}^{k,l} = 1$ iff the kth value of L_q for variable q and the lth value of L_s for variable s can be selected simultaneously. In this case, it is said that the kth value of L_q for variable q is consistent with the lth value of L_s for variable s. Otherwise $R_{q,s}^{k,l} = 0$. A relation can be represented in a matrix with dimensions $|L_q| \times |L_s|$.

In Figure A.1 a conflict between two nodes in a conflict graph is represented in a constraint

$$R_{v1,v2} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} = R^T_{v2,v1}$$

Figure A.1: Constraints of a conflict graph (a) modelled in a contraint graph (b).

graph. Assignment of the same value to variable $v1$ as well as to $v2$ is prevented by the zeros in the relation $R_{v1,v2}$.

Intersection of two relations allows only pairs that are allowed by both. Intersection of two relations is defined as: $R_{q,s} = R'_{q,s}$ & $R''_{q,s}$ with $R^{k,l}_{q,s} = {R'}^{k,l}_{q,s} \wedge {R''}^{k,l}_{q,s}$. The $\wedge$ operator denotes the boolean AND function.

Composition of two relations, $R_{q,r}$ and $R_{r,s}$, induces a relation $R_{q,s}$ as follows: A pair $(x1, x3)$ is allowed by $R_{q,s}$ if there is at least one value x_2 such that $(x1, x2) \in R_{q,r}$ and $(x2, x3) \in R_{r,s}$. In matrix notation the induced relation $R_{q,s}$ can be obtained by a bitwise matrix multiplication $R_{q,s} = R_{q,r} \cdot R_{r,s}$ with:

$$R^{k,l}_{q,s} = (R^{k,0}_{q,r} \wedge R^{0,l}_{r,s}) \vee (R^{k,1}_{q,r} \wedge R^{1,l}_{r,s}) \vee ... \vee (R^{|L_r|,l}_{q,r} \wedge R^{|L_r|,l}_{r,s}) \tag{A.1}$$

The $\vee$ operator in equation A.1 denotes the boolean OR function.

The intersection and composition operations can be applied to the relations in a constraint graph, in order to reduce the number of nodes in this graph. This is illustrated with the assignment conflict graph representation shown in Figure A.3 for the network in Figure A.2. The assignment conflict graph of Figure A.3 is converted in the constraint graph of Figure A.4. From this constraint graph the CN1 and CN2 nodes are removed by applying the composition operation twice. This results in the constraint graph of Figure A.5. The compact constraint graph of Figure A.6 is obtained by applying the intersection operation.

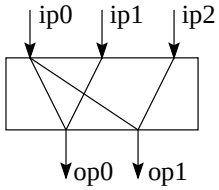


Figure A.2: Communication network example.

In Chapter 2 it was shown that all communication paths for data through the data path of a VLIW processor can be represented with 5 non-blocking networks. These networks can be

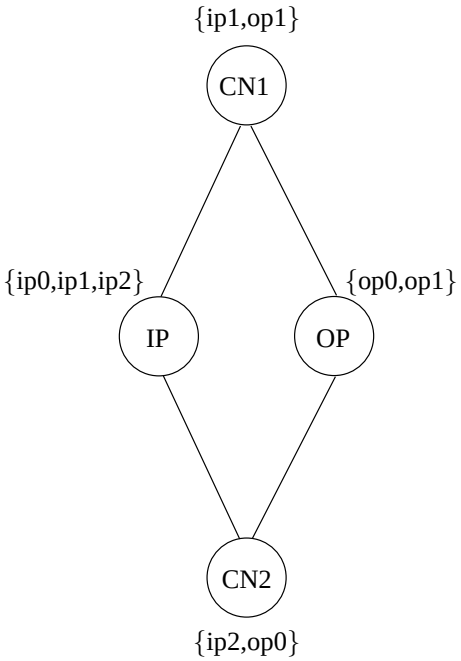


Figure A.3: An assignment conflict graph model of the network in Figure A.2.

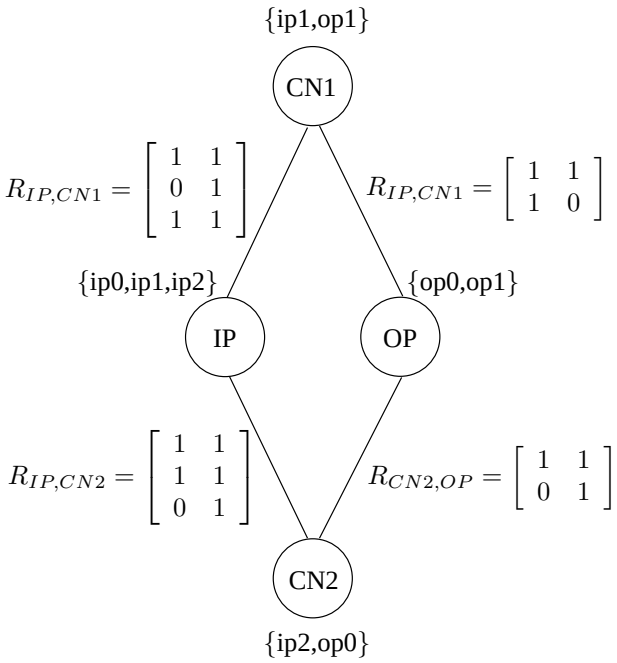
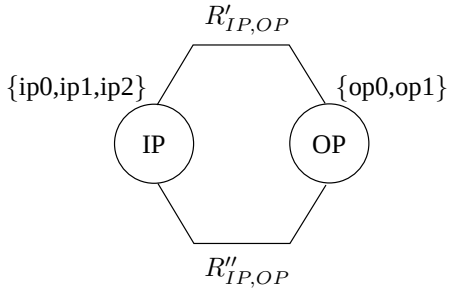


Figure A.4: A onstraint graph graph representation of the conflict graph in Figure A.3.

$$R'_{IP,OP} = R_{IP,CN1} \cdot R_{CN1,OP} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$



$$R'_{IP,OP} = R_{IP,CN1} \cdot R_{CN1,OP} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}$$

Figure A.5: The contraint graph after the composition operation have been applied twice on the relations of the constraint graph of Figure A.4.

$$R_{IP,OP} = R'_{IP,OP} \,\&\, R''_{IP,OP} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Figure A.6: The contraint graph after the intersection operation have been applied on the relations of the the constraint graph of Figure A.5.

modelled with 5 different relation matrices. Because there is also one relation matrix needed to model a resource conflict there are in total 6 different relations matrices present in a compact constraint graph representation of an assignment conflict graph. In our implementation only these 6 relation matrices are stored in memory and pointers are used to associate an edge with a relation matrices. The pruning rules are defined such that the content of the relation matrices remains unchanged.

The additional pruning rule is illustrated with the network instance in Figure A.7. If we assume that the input port $ip2$ of this network cannot be used then the output port $op2$ cannot be reached. The network of Figure A.7 is modelled in the assignment conflict graph of Figure A.8. The pruning rules described in Section 5.1 are not able to derive that $op2$ should be removed from the domain of the OP node. However, that $op2$ is only an option for the variable OP if $ip2$ is an option for the variable IP can easily be derived from the third column of the $R_{IP,OP}$ relation in Figure A.9. In other words, in the relation matrix $R_{x,y}$ it is explicitely visible which values of a domain of variable x can be removed if a value of the domain of variable y is removed. These infeasible case can be derived with an arc consistency algorithm [Mac77] which has a computational complexity of O($|L|^3 * |V_{cg}| * |E_{cg}|$) [MF85] with $L = max(|L_q|, |L_s|)$.

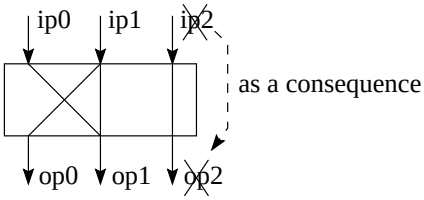


Figure A.7: In this network there is only one connection possible from an input port to output port $op3$. Therefore if input port $ip2$ is not available in this network then output port $op3$ is not reachable.

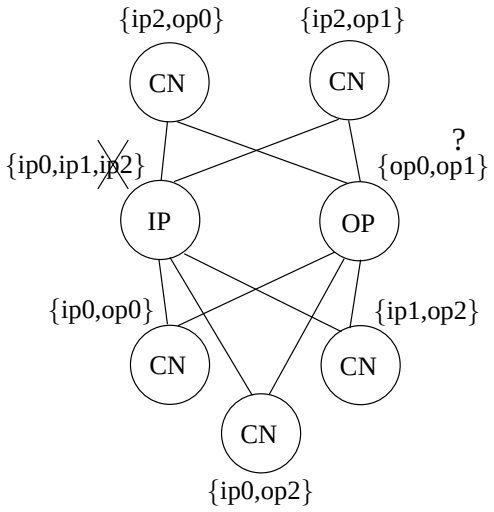


Figure A.8: Assignment conflict graph in which the network of Figure A.7 is modelled.

$$R_{IP,OP} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

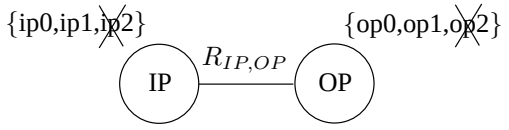


Figure A.9: Constraint graph in which the network of Figure A.7 is modelled.

Bibliography

[Ade] *http://www.adelantetech.com*.

[AP02] C.A. Alba Pinto, *Storage constraint satisfaction for embedded processor compilers*, Ph.D. thesis, Eindhoven University of Technology, 2002.

[ASU86] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers principles, techniques, and tools*, Addison-Wesley, 1986.

[Ban98] S. Banerjia, *Instruction scheduling and fetch mechanisms for clustered VLIW processors*, Ph.D. thesis, North Carolina State University, 1998.

[BDEO97] P. Bergner, P. Dahl, D. Engebretsen and M. O'Keefe, *Spill code minimization via interference region spilling*, Proceedings of the 1997 ACM SIGPLAN conference on Programming language design and implementation, May 1997, pp. 287–295.

[Bea91] S.J. Beaty, *Genetic algorithms and instruction scheduling*, (Albuquerque, New Mexico), Proc. of the 24th Annual International Workshop on Microprogramming, 1991, pp. 206–211.

[BL99] S. Bashford and R. Leupers, *Constraint driven code selection for fixed-point DSPs*, (New Orleans, USA), Proceedings Design Automation Conference, 1999, pp. 817–22.

[Bra99] R. Braspenning, *Modeling issue slot constraints with resources*, Tech. report, Eindhoven University of Technology, 1999.

[BWB00] N. Busá, A. van der Werf and M. Bekooij, *Scheduling coarse grain operations for VLIW processors*, (Madrid, Spain), Poc. Int. Symp. on System Synthesis, 2000.

[CCPS98] W.J. Cook, W.H. Cunningham, W.R. Pulleyblank and A. Schrijver, *Combinatorial optimization*, John Wiley & Sons, 1998.

[Cha82] C. Chaitin, *Register allocation & spilling via graph coloring*, Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, June 1982, pp. 98–105.

[Clo53] C. Clos, *A study of non-blocking switching networks*, Bell System Technical Journal **32** (1953), no. 2, 406–424.

[CLR90] T. Cormen, C. Leierson and R. Rivest, *Introduction to algorithms*, MIT Press, 1990.

[Cor95] H Corporaal, *Transport triggered architectures: Design and evaluation*, Ph.D. thesis, Delft University of Technology, 1995.

[Cou97] O. Coudert, *Exact coloring for real-life graphs is easy*, Proceedings of the 34th ACM/IEEE Design Automation Conference., 1997.

[Des98] G. Desoli, *Instruction assignment for clustered VLIW DSP compilers: A new approach*, Tech. report, HP-98-13, 1998.

[DP88] R. Dechter and J. Pearl, *Network-based heuristics for constraint-satisfaction problems*, Artificial Intelligence (1988), no. 34, 1–38.

[ECR99] C. Eisenbeis, Z. Chamski and E. Rohou, *Flexible issue slot assignment for VLIW architectures*, In 4th International Workshop on Software and Compilers for Embedded Systems, St. Goar, Germany, 1999.

[Ell86] J.R. Ellis, *Bulldog: A compiler for VLIW architectures*, ACM Doctor Dissertation Awards, MIT, Cambridge, 1986.

[EMAP+00] C.A.J. van Eijk, B. Mesman, C. Alba-Pinto, Q. Zhao, M. Bekooij, J. van Meerbergen and J. Jess, *Constraint analysis for code generation: Basic techniques and applications in FACTS*, ACM Transactions on Design Automation of Electronic Systems, vol. 5, 2000, pp. 774–793.

[EMT99] C.A.J. van Eijk, B. Mesman and A. Timmer, *Identification and exploitation of symmetry in DSP algorithms*, Proc. IEEE conf. on Design Automation and Test in Europe, 1999, pp. 602–608.

[FDF98] P. Faraboschi, G. Desoli and J.A. Fisher, *Clustered instruction-level parallel processors*, Tech. report, HPL-98-13, 1998.

[Fur96] Steve Furber, *ARM system architecture*, Addison-Wesley, 1996.

[Gag01] J.S.H. van Gageldonk, *Instruction scheduling for COCOON*, Technical Note 194, Philips Research, July 2001.

[GFO92] A. De Gloria, P. Faraboschi and M. Olivieri, *A non-deterministic scheduler for a software pipelining compiler*, Proceedings of the 25th Annual International Symposium on Microarchitecture, 1992.

[GJ79] M.R. Garey and D.S. Johnson, *Computers and intractability - a guide to the theory of NP-completeness*, W.H. Freeman and company, 1979.

[Goo89] G. Goossens, *Optimisation techniques for automated synthesis of application-specific signal-processing architectures*, Ph.D. thesis, Katholieke Universiteit Leuven, 1989.

[GPL+97] G. Goossens, J. van Praet, D. Lanneer, W. Geurts, A. Kifli, C. Liem and P. Paulin, *Embedded software in real-time signal processing systems: Design technologies*, Proceedings of the IEEE, vol. 85, March 1997, pp. 436–454.

[Gri94] R.P. Grimaldy, Discrete and Combinatorial Mathematics, ch. 11, pp. 588–598, Addison Wesley, 1994, pp. 588–598.

[HD98] S. Hanono and S. Devadas, *Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator*, Proc. Design Automation Conference, ACM Press, 1998, pp. 510–515.

[HMV] D. Hwang, C. Mittelsteadt and I. Verbauwhede, *Low power showdown: Comparision of five DSP platforms implementing a LPC speech codec*.

[KLMW98] P. Kievits, E. Lambers, C. Moerman and R. Woudsma, *R.E.A.L. DSP technology for telecom baseband processing*, CDROM Proc. International Conference on Signal Processing Applications & Technology, 1998.

[KM92] D.C. Ku and G. De Micheli, *High-level synthesis of ASICs under timing and synchronization constraints*, Kluwer Academic Publishers, 1992.

[KMN+92] T. Krol, J. van Meerbergen, C. Niessen, W. Smits and J. Huisken, *The SPRITE input language: An intermediate format for high level synthesis*, European Design Automation Conference (EDAC) (Brussels, Belgium), March 1992.

[Koc95] E. de Kock, *Video signal processor mapping*, Ph.D. thesis, Eindhoven University of Technology, 1995.

[Lam88] M. Lam, *Software pipelining: An effective scheduling technique for VLIW machines*, Proceedings of the SIGPLAN, June 1988, pp. 318–328.

[LBSL94] P. Lapsley, J. Bier, A. Shaham and E.A. Lee, *DSP processor fundamentals*, Berkeley Design Technology, Inc., 1994.

[Lea97] W. Lee et al., *A 1-v programmable DSP for wireless communications*, IEEE Journal of Solid-State circuits **32** (1997), no. 11, 1766–1776.

[Leu97] R. Leupers, *Retargetable code generation for digital signal processors*, Kluwer Academic Publishers, 1997.

[LP98] H.R. Lewis and C.H. Papadimitriou, *Elements of the theory of computation*, Prentice Hall, 1998.

[Mac77] A.K. Mackworth, *Consistency in networks of relations*, Artificial Intelligence (1977), no. 8, 99–118.

[MDR+00] P. Mattson, W. Dally, S. Rixner, U. Kapasi and J. Owens, *Communication scheduling*, Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, November 2000.

[Mes01] B. Mesman, *Constraint analysis for DSP code generation*, Ph.D. thesis, Eindhoven University of Technology, 2001.

[MF85] A.K. Mackworth and E.C. Freuder, *The complexity of some polynomial network consistency algorithms for constraint satisfaction problems*, Artificial Intelligence (1985), no. 25, 65–74.

[Mic94] G. De Micheli, *Synthesis and optimization of digital circuits*, McGraw-Hill, 1994.

[MIP] *http://www.mips.com*.

[Mon74] U. Montanari, *Networks of constraints: Fundamental properties and applications to picture processing.*, Information Sciences **7** (1974), 95–132.

[MSTM97] B. Mesman, M. Strik, A. Timmer and J. van Meerbergen, *Constraint analysis for DSP code generation*, Proc. Int. Symp. on System Synthesis, 1997, pp. 33–40.

[Muc97] S. Muchnick, *Advanced compiler design and implementation*, Morgan Kaufmann Publishers, 1997.

[NE98] E. Nystrom and A. Eichenberger, *Effective cluster assignment for modulo scheduling*, 31st Annual ACM/IEEE International Symposium on Microarchitecture, Dallas, USA, 1998, pp. 103–14.

[Nol97] P. Noll, *ISO/MPEG audio coding*, International Journal of High Speed Electronics and Systems **8** (1997), no. 1, 69–118.

[Nui94] W.P.M. Nuijten, *Time and resource constrained scheduling*, Ph.D. thesis, Eindhoven University of Technology, 1994.

[RDK+00] S. Rixner, W.J. Dally, B. Khailany, P Mattson, U.J. Kapasi and J.D. Ownes, *Register organization for media processing*, Proc. of the 6th International Symposium on High-Performance Computer Architecture, 2000, pp. 375–386.

[SV76] A. Sangiovanni-Vincentelli, *A note on bipartite graphs and pivot selection in sparse matrices*, vol. 23, IEEE Transactions on Circuits & Systems, 1976.

[Sys] *http://www.systemc.org*.

[Tim95] A. Timmer, *From design space exploration to code generation*, Ph.D. thesis, Eindhoven University of Technology, 1995.

[WLH00] K. Wilken, J. Liu and M. Heffernan, *Optimal instruction scheduling using integer programming*, 2000, pp. 121–133.

[WNS97] M. Wallace, S. Novello and J. Schimpf, *ECLiPSe: A platform for constraint logic programming*, Tech. report, IC-Parc, Imperial College, London, 1997.

[Wul81] W.A. Wulf, *Compilers and computer architecture*, IEEE Computer **14** (1981), no. 7, 41–47.

[ZBJ+01] Q. Zhao, T. Basten, J. Jess, B. Mesman and C.A.J. van Eijk, *Static resource models of instruction sets*, (Montréal, Canada), Proc. Int. Symp. on System Synthesis, September 2001, pp. 159–164.

[ZVSM94] V. Zivojnovic, J.M. Velarde, C. Schlager and H. Meyr, *DSPSTONE:a DSP-oriented benchmarking methodology*, CDROM Proc. International Conference on Signal Processing Applications & Technology, 1994.

Samenvatting

In veel consumenten elektronica producten worden processoren toegepast voor het bewerken van gedigitaliseerde signalen. Deze processoren zijn gewoonlijk ingebed in een systeem en moeten wat rekenkracht, vermogensverbruik en fabricage kosten aan stringente eisen voldoen.

Door het optimaliseren van een processor voor een specifieke taak, of een kleine verzameling van taken, kan er aan strengere eisen worden voldaan. Deze specialisatie heeft een grotere diversiteit aan processor types tot gevolg. Door het toepassen van geautomatiseerde processor ontwerp en programmeer systemen wordt er getracht om de ontwikkelkosten in de hand te houden.

Een processor kan onder andere geoptimaliseerd worden door het toepassen van een incompleet communicatie netwerk in de processor. Daarnaast is het wenselijk om meerdere register files toe te passen in een processor met een groot aantal parallelle bewerkingseenheden. Deze optimalisaties hebben tot gevolg dat er veel hulp en expertise van programmeur nodig is om hoogwaardige microcode te genereren met behulp van traditionele code generatie technieken in een compiler. Met de in dit proefschrift beschreven code generatie methode is het in veel gevallen wel mogelijk om hoogwaardige microcode volledig automatisch te genereren.

Het toepassen van een incompleet netwerk in de processor maakt het toekennen van basis bewerkingen aan bewerkingseenheden een moeilijke taak voor de code generator. Een toekenning moet namelijk zo plaatsvinden dat voor iedere bewerking die uitgevoerd wordt op een bewerkingseenheid er een kanaal in het netwerk van de processor is, dat gebruikt kan worden om het resultaat naar de bewerkingseenheid toe te sturen die de resultaat consumerende bewerking uitvoerd. Dit communicatiekanaal en de bewerkingseenheid moeten tevens op het gewenste tijdstip beschikbaar zijn.

In de voorgestelde code generatie methode wordt er gezocht naar een oplossing. Na het nemen van een bewerkings toekenningsbelissing wordt er geanalyseerd welke toekomstige beslissings opties niet tot een oplossing kunnen behoren gegeven de reeds gemaakte beslissingen. Deze gevallen worden verwijderd uit de zoekruimte zodat tijdens toekomstige beslissingen andere toekenningsbeslissingen zullen worden geprobeerd. Indien er gedetecteerd wordt dat er gegeven de gemaakt beslissingen geen oplossing bestaat, dan worden er beslissingen ongedaan gemaakt en andere opties geprobeerd. Het verwijderen van zoveel mogelijk beslissings opties die niet tot een oplossing behoren, verminderd het aantal keer dat er op een beslissing terug gekomen moet worden en de tijd die nodig is om een oplossing te vinden.

Voor het bewerking aan bewerkingseenheid toekenings probleem wordt er een conflict graaf opgesteld waarin alle opties en combinatie van niet toegestane opties gerepresenteerd worden. Gevallen die zeker niet tot een oplossing behoren worden gevonden met algoritmes die rekentijd efficiënt zijn. Indien door analyse wordt vastgesteld dat twee bewerkingen op hetzelfde tijdstip uitgevoerd moeten worden dan wordt er een kant in de conflict graaf toegevoegd. Deze kant sluit uit dat deze beide bewerkingen aan dezelfde bewerkingseenheid wordt toegekend. Indien er wordt vast gesteld dat een bewerking op een specifieke bewerkingseenheid moet worden uitgevoerd dan wordt deze informatie gebruikt om nauwkeuriger het tijdsinterval te bepalen waarin de operatie uitgevoerd kan worden.

De voorgestelde toekenningstechnieken zijn ge-implementeerd in een prototype codegenerator FACTS. Deze code generator is gekoppeld aan de processor synthese omgeving A|RT-designer. Door het koppelen van FACTS aan A|RT-designer kunnen processoren, die bevroren zijn na synthese, hergeprogrammeerd worden. Deze omgeving is gebruikt om de codegeneratie technieken in FACTS te evalueren voor industrieel relevante applicatie domein specifieke processor ontwerpen. De resultaten tonen aan dat er met deze technieken in veel gevallen microcode gegenereerd kan worden die de opslag capaciteit van de register files en de beschikbare verbindingen in de VLIW-processor respecteert en aan stringente eisen wat betreft de rekentijd voldoet.

Curriculum Vitae

Marco Bekooij was born on March 26, 1968 in Doorn, The Netherlands. From 1989 he studied Electrical Engineering at the Hogeschool Utrecht. After his graduation in 1992 he continued his study in Electrical Engineering at the Twente University of Technology. In 1995 he graduated on the subject of verification of digital circuits.

Since May 1995, Marco Bekooij is working as a research scientist at the Philips Research Laboratories Eindhoven. From December 1998, he has been working towards a Ph.D. degree. He has published several papers on various aspects of digital signal processing, hardware-software codesign, and code generation for re-programmable VLIW processors.