A THEORY FOR AUTOMATED SYNTHESIS OF ARCHITECTURES FOR REPETITIVE MULTI-RATE ALGORITHMS

Sanjay R. Deshpande

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-90-28 August 1990

To my parents,

Nisha and Raghunath Deshpande

A THEORY FOR AUTOMATED SYNTHESIS OF ARCHITECTURES FOR REPETITIVE MULTI-RATE ALGORITHMS

by

SANJAY R. DESHPANDE, B.TECH., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May, 1990

Acknowledgments

I would like to thank my supervisor, Prof. J. C. Browne, for his constant support and encouragement. My deep gratitude goes to Dr. P. P. Jain for his role as my co-supervisor, support, help, and especially for making available his valuable time for the innumerable discussions we had throughout this research. I would like to thank Prof. R. M. Jenevein for his support throughout, the many valuable discussions during the early stages of this research, and his help during the preparation of the final draft of this dissertation. Thanks are also due to Professors G. J. Lipovski, D. S. Fussell, H. G. Cragon, and R. H. Flake for serving on my dissertation committee.

I would like to extend my appreciation to Cheng-Liang Lin and Mukund Belliappa for implementing the DDGTool. Thanks are due to Peter Newton and Steve Sobek for many helpful suggestions. Thanks also go to Karen Nordby for her help in preparing this document.

There are some others that I would like to mention here. Their presence made it all seem a little easier.

I will like to thank Roy Jenevein again for being a dear friend. Special thanks go to Lucille Jenevein for her loving friendship and for making me feel a part of the family. Thanks to Matt Sejnowski for being a dear and close friend. Our post-tennis discussions over Conan's pizza were always intellectually invigorating. Thanks go to Ravi Rao for willingly tolerating my lay curiosity in Physics, which often lead to long illuminating discussions, and for the after-

noon cappuccino. Last, but not least, I would like to thank the members of my family — my parents, my sister Nanda, brother-in-law Anil Pradhan, nephew Nilay, and niece Madhumita. Their love, support, and encouragement saw me through many a dim day.

Sanjay R. Deshpande

The University of Texas at Austin

May, 1990

A THEORY FOR AUTOMATED SYNTHESIS OF ARCHITECTURES FOR REPETITIVE MULTI-RATE ALGORITHMS

Publication No. _____

Sanjay R. Deshpande, Ph.D.
The University of Texas at Austin, 1990

Supervisors: James C. Browne and Prem P. Jain

A theoretical framework is developed to achieve automated architectural synthesis for data-independent, repetitive, multi-rate algorithms from their behavioral specifications.

Multi-rate functions are formally defined. It is shown that systolic architectures for algorithms incorporating multi-rate functions make inefficient use of hardware components and that a multi-clock design style can produce more efficient architectures.

A graph-oriented language, called Data Dependency Graphs (DDGs), is introduced to facilitate the specification of multi-rate computations. Computational semantics suitable for multi-rate computations are associated with

the nodes and edges of the DDG. A compatible model for function execution by hardware components is proposed. A bus-based architectural scheme is also proposed.

The synthesis process is seen as translation from DDGs to a architectures. Analytic techniques are introduced to extract design information from the DDGs. Synthesis problems are formulated, and heuristic approaches are suggested for their solution. An implementation of a heuristic synthesis system is described. The implementation is evaluated via experiments.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

A continuing trend towards miniaturization of feature sizes and a move towards design automation has fueled the growth of the Application-Specific IC (ASIC) industry. Over the last few years, the progress of VLSI technology has made it less expensive to invest in a dedicated hardware system and thus obtain very high speed-ups.

The interest in ASICs, and even board-level application-specific computing engines in general, is engendered from several quarters. First and foremost, are the algorithms in signal processing. In the past, typically, these algorithms were executed off-line on large and powerful main-frame computers. More and more, however, for applications such as radar and sonar, and on remote computing platforms such as in avionics or space-based instrumentation, the processing of signals is done on-line and in real-time. Moreover, increasingly, digital signal processing is also finding its way into systems which heretofore were the realm of analog processing, such as television and radio. The on-line execution of these extremely compute-intensive algorithms demands very high processing rates. These considerations, in effect, disqualify main-frame based solutions, and, on the other hand, promote special purpose systems as solutions. Similar considerations also hold for vision and control applications in robotics and in real-time graphics processing, where computational geometry replaces signal processing. The last but not the least impetus is received from

the desire and feasibility of embedding algorithms, previously executed on general purpose processors, in hardware, at low cost, thereby gaining substantial speed advantage over the conventional solution.

All integrated circuits benefit from the speed gains resulting from faster device technologies and reduction in the feature sizes, but ASICs derive their speed advantage by employing only those hardware functions that are used by the algorithm, and by customizing their interconnections. But ASICs, since they are typically manufactured in smaller volumes, are beset by the problem of high design overhead per unit. Therefore, an increased level of design automation is imperative to make them economically attractive.

Automated design of *algorithm-specific circuits* has often been called *silicon compilation* [27, 11], in which, the computation to be performed is described in some suitably abstract machine-readable form. This specification is then automatically translated into a low level description of a circuit, and ultimately layout, that implements the computation.

Over the last decade, a significant amount of research has gone into advancing the state-of-the-art in silicon compilation. The effort has ranged from defining new design description languages, VHDL, ZEUS, and ISP [53, 19, 6], to cite a few, to the construction of complete design environments, MAHA, Yorktown Silicon Compiler, Cathedral I, II, and III, and HAL [45, 7, 25, 13, 41, 47], for example. Table 1.1 gives a more comprehensive list of synthesis systems described in the literature.

In [11] Gajski and Thomas characterize different aspects of silicon compilation using the *Y-chart*, reproduced here in Figure 1.1. The three axes of the Y-chart represent three different domains of design description: *behavioral, structural, and physical-form.* The chart has the characteristic that the

System	Site	Reference
Cathedral I	IMEC	[25]
Cathedral II	IMEC	[13]
Cathedral III	IMEC	[41]
BUD-DAA	AT&T	[38]
Elf	Audesyn Inc.	[20]
Emerald/Facet	CMU	[59]
EMUCS	CMU	[22]
Flamel	Stanford	[58]
HAL	Carlton U.	[47]
Hercules	Stanford	[39]
MAHA	USC	[45]
SAW	CMU	[57]
SLICER	U. Illinois	[42]
YSC	IBM (Yorktown)	[7]

Table 1.1: Some high level synthesis systems.

descriptions farther from the center are more abstract than those closer to it. Thus the innermost level corresponds to the lowest level VLSI circuit primitives, and the outermost level corresponds to system-level descriptions.

Silicon compilation can usually be divided into two steps: 1) translation from behavior to structure, and 2) translation of structure to geometric form for layout. By convention, the former step is called *synthesis*, and the latter is comprised of *floor-planning, placement, and routing.*

The research reported herein falls under the category of synthesis. In particular, the research deals with high-level architectural synthesis. In it, the goal is to describe algorithms in a suitable high level programming language and to produce an architecture using pre-designed building blocks, such as ALUs.

In the following section, an overview of the architectural synthesis

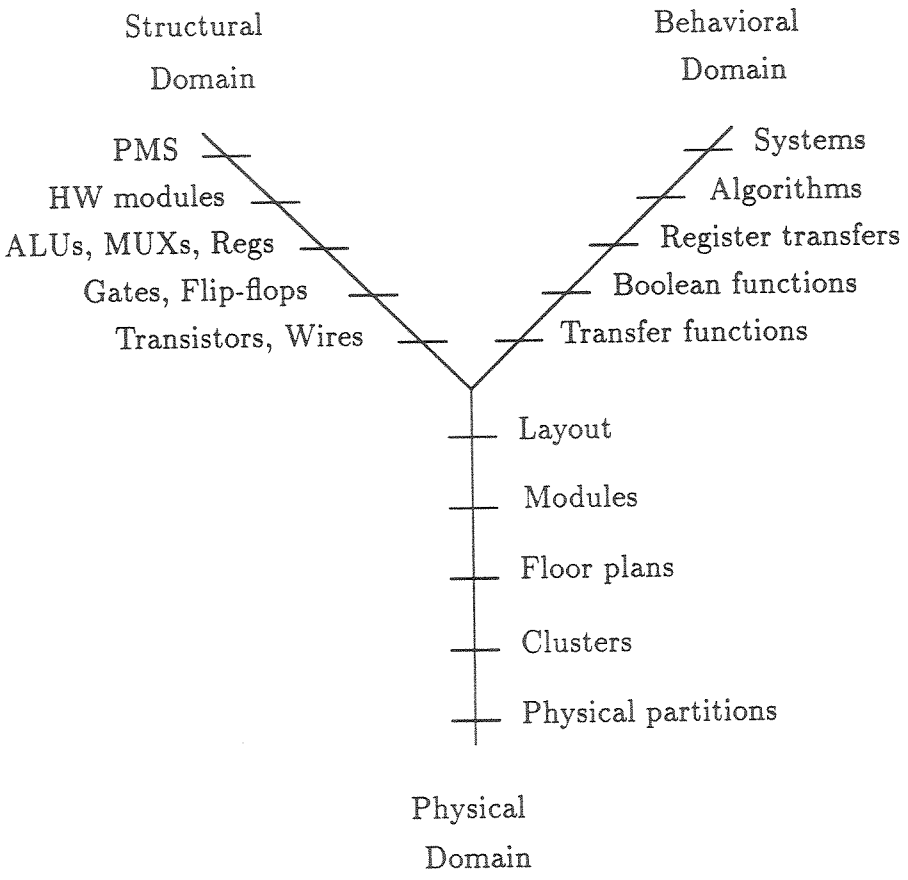


Figure 1.1: The Y-chart for silicon compilation.

process is given. In Section 1.2, a survey of related work is given. In Section 1.3 the goals of this research are stated. Section 1.4 contains an outline of the approach developed in this research. Section 1.5 contains the outline of the remaining chapters of this dissertation. Finally, in Section 1.6 the contributions of this research are briefly previewed.

1.1 Overview of the synthesis process

High level synthesis starts with a specification of the computation in an abstract programming language which obscures the details of the hardware

components that will eventually implement the operations. Typically, before an architecture is designed, this specification goes through translation phases involving, perhaps one or both of:

- *Behavior level transformations:* Certain parts of the computation may be transformed into operations that are implementable in hardware. Loops may be unrolled creating a loop-free computation. Temporary variables may be inserted or removed to simplify the computation.

- *Specification language transformation:* The translation process may use an intermediate form of specification that is easier to analyze. Most often, textual input specifications are translated into directed graphs. For example, many synthesis systems employ *Value Trace*, a directed acyclic graph representation developed at CMU [54], as an intermediate form.

Subsequent to the above steps, decisions are made about architectural issues. These include,

- *Choice of computational primitives:* The computational operations in the algorithm are *bound* to types of components. For example, both additions and subtractions may be executed on a common ALU. Similarly, divide-by-2 operation may be done using a "right-shift" component.

- *Choice of communication primitives:* The data transfers among the computational components have to be effected via a communication structure. It is essential to determine the components such as busses, multiplexers, cross-bar switches, etc., out of which this structure will be constructed.

- *Choice of clocking scheme:* All or almost all synthesis systems produce designs which are synchronous, and therefore must decide the nature of

the clocking scheme based on an underlying global clock. Apart from the decisions related to technology, there can be three basic choices:

1. all registers in the architecture are clocked with the edges of the global clock (e.g. in a systolic design),
2. each registers is clocked by one of a small number of inter-related clocks derived, prior to synthesis, from the global clock, or
3. clocks for individual registers in the buffer are specified by the final design.

Characteristics of an architecture produced by the synthesis system greatly depend on the decisions made regarding the above issues. These decisions act as constraints, and in effect define its *component domain* (C), subject to which the synthesis process must design. These, in turn, help partly define its *algorithmic domain* (A), the types of operation that can be "faithfully" translated from the input language, and consequently, the types of synthesizable algorithm.

(By faithful is meant, without any intermediate modification at the algorithm specification level. For example, if a complex-multiplication operation is specified in the algorithm, the synthesis system may either choose to resolve it into component operations and implement them via adders and multipliers, or it may utilize a complex-multiply component. The latter is considered a faithful translation.)

In this view, synthesis is seen to be a *mapping* from components and algorithms to architectural designs (D), $(C \times A) \longrightarrow D$. The mapping implied in the synthesis process is different from the mapping of an algorithm onto an

architecture in the traditional sense: in the latter, the architecture is predefined, whereas in the former, there exists an extra degree of freedom and the architecture has to be "computed" from the demands of the algorithm.

The synthesis process must produce designs to achieve one or more objectives including cost, utilization, and speed of computation. These objectives along with the abovementioned constraints guide the formulation of algorithms employed in the synthesis process.

The term *synthesis methodology* is used to mean collectively, the component domain of the synthesis process, its design objectives, and the collection of analytic techniques and algorithms it embodies.

1.2 Background and related work

In this section a brief look is taken at other approaches to synthesis of architectures from behavioral specifications. The salient aspects of these approaches are indicated.

There have been many successful efforts at obtaining a systolic architecture for a given algorithm. These methods are based on the principle of *retiming* presented in [35], in which the algorithm is modified such that the algorithm execution is pipelined and each unit of computation has a delay less than or equal to the cycle time of the global clock. Some examples of such efforts are [50, 9, 49, 24]. However, as this research has shown (see Chapter 2), for multi-rate algorithms, systolization does not produce efficient architectures.

All synthesis approaches are characterized by the absence of a predetermined number of components in the architecture and the connectivity between them. Thus any design approach which does not make assumptions about these two aspects can be considered a synthesis approach.

Renfors and Neuvo in [51] built upon the transform techniques of Fettweis [17] to obtain a minimum latency architecture for recursive digital filters under hardware delay constraints. However, they did not consider the effects of multi-rate operations or components.

Schwartz in [52] showed the existence of *cyclo-static schedules* to execute computations defined by signal flow graphs on synchronous homogeneous multiprocessor systems. The cyclo-static schedules are found to require the minimum number of processors determined from the computational requirements of the algorithm under given input latency. Schwartz also used an extended form of the transform technique of Fettweis, in addition to some other techniques such as *blocking*, to obtain the schedules. However, the approach developed in [52] is not extensible to real-time, on-line, multi-rate algorithms and the use of heterogeneous multi-rate hardware components.

Lee and Messerschmitt, in [34], presented a technique for scheduling multi-rate computations on a uniprocessor. They also extended the technique to schedule the algorithm on a shared-memory multiprocessor. For the latter, they assumed that all the processors were identical. Since they also made an assumption about the number of processors in the architecture as well as their connectivity instead of computing it, their results are not applicable for architectural synthesis. Nonetheless, they introduced *synchronous data flow graphs* of which the *Data Dependency Graphs*, used here are, an extension.

The Sehwa pipeline synthesis system was one of the first pipeline synthesis systems to be described in the design automation literature [44]. Sehwa is part of the MAHA synthesis system described in [45], and is based on the graph-partitioning concept. The single rate algorithm to be implemented is input as a directed acyclic graph. The synthesis system partitions the graph,

a partition being executed within a single clock cycle. An edge that spans neighboring partitions results in the insertion of a buffer and adds to the cost and delay of the architecture. The edges internal to a partition get merged into multiplexers. The computational components are not *shared* within a clock cycle, but can be shared during different clock cycles, leading to a reduction in cost. The MAHA system uses the concept of *freedom*, the difference between the ALAP and ASAP scheduling times of an operation, and uses the heuristic of scheduling the operation with least freedom first. The Sehwa pipeline synthesis algorithm is examined in greater detail in Chapter 3.

The HAL synthesis system, described in [46], is also oriented towards synthesis of pipelines for repetitive single rate algorithms. The algorithm to be implemented is input as a data flow graph, and the system produces an architecture with minimum number of components. The significant contribution of the HAL system is the *steepest-descent* heuristic-based, global load balancing scheduling algorithm used to minimize the concurrency of use of individual types of components [47].

In [7] Bryton and others describe the Yorktown Silicon Compiler. The system consists of an array of languages and tools for behavioral synthesis from algorithm specification to layout synthesis. The algorithm to be implemented is specified in a language called Yorktown Intermediate Format. The synthesis itself is divided into the three steps of: structural synthesis, logic synthesis and layout design. The structural synthesis refers to the architectural synthesis being considered herein. The central scheduling procedure in structural synthesis is called *control state splitting*; it is used to allocate increasing number of clock cycles for long operations.

System Architect's Workbench (SAW) from Carnegie Mellon Univer-

sity ([57]) is tuned specifically to automating microprocessor design. The algorithm being implemented is input in Instruction-Set Processor Specification (ISPS) and translated into a *value trace*, a directed acyclic graph [54]. The value trace is then scheduled by the system's CSTEP module using a modified list scheduling technique.

The Slicer silicon compiler ([42]) accepts the behavioral description in a Pascal-like input language and converts it into a *control/data flow graph* (CDFG). It subsequently uses critical path analysis of the graph to schedule the operations by giving the operation with the least amount of slack (called *mobility* in Slicer terminology) highest priority.

1.3 Research goal

This section discusses the objectives of this research. It starts by discussing the motivations behind the current research.

1.3.1 Motivations

The current research was aimed at extending the applicability of the synthesis process in two directions: 1) handling of multi-rate algorithms and, 2) optimization of communication hardware along with computational hardware. Motivations for seeking these extensions are as follows:

Multi-rate algorithms

High levels of integration has made it attractive to embed large algorithms in hardware. Many such algorithms involve operations that result in changes in data rates. These operations accept a certain number of input operands and produce a certain different number of output operands per exe-

cution. For example, a decimation-by-four operation takes four data items as input and produces one data item at the output.

Operations such as decimation, that cause change in data rates, are herein called *multi-rate functions*, and the algorithms incorporating them are referred to as *multi-rate algorithms*. Instances of multi-rate algorithms appear in diverse application areas such as:

- Signal processing: e.g., modulation/demodulation, sonar beam-forming, and phased array radar.
- Adaptive control.
- Graphics: e.g., interpolation, anti-aliasing computations.
- Coding theory.
- Data encryption and recovery.

In the above application areas, certain algorithms are inherently multi-rate by virtue of computational definition. However, multi-rate behavior may also occur in common hardware components which multiplex data over their ports. Such components are said to *share* their ports, and are referred to as *shared port* components.

As the level of integration increases, the number of available external connections do not grow as fast as the area of the circuit. In fact, it has been observed (Rent's Rule, [40]) that the number of connections grows only as $4g^{0.6}$, where g is the number of gates in a VLSI circuit. The implication is that, the larger the circuit is, the more likely it is to share external ports and behave

as a multi-rate function. A hierarchical design style, often adopted for large designs, furthers this necessity of sharing ports.

Traditionally, multi-rate algorithms are divided into smaller sub-computations which have constant data rates. These sub-computations are synthesized into separate circuit components, which are subsequently interconnected via manually designed interfaces.

In addition to being slow and expensive, this traditional approach is not well defined and can often result in inefficient designs. A single step synthesis process is required to produce efficient architectures. Such a single step process must support abstractions to adequately model the multi-rate behaviors of functions.

Optimization of communication hardware

Another extension was aimed at optimizing communication hardware. Most of the synthesis methodologies described in the literature relegate to the optimization of communication hardware a secondary status. It is done *a posteriori* to the optimization of the computational components and storage. However, when the number of communication components in the architecture is large, and when each of these connects together several computational components, it is important that optimization of communication hardware be given a serious consideration.

It is true that communication requirements can be traded against storage requirements, since, for every postponed data-transfer, a buffer must be included to hold that datum. However, because the area cost of a communication component, say, a bus, connecting multiple components is larger than that for buffer registers, the trade-off is considered worthwhile. This can be

expected to be especially true for architectures with large number of computational components.

1.3.2 Research goal

The overall three-part goal of this research effort has been:

1. Develop a framework to
   - Express repetitive, multi-rate algorithms with the following constraints:
     - Repetition rate is constant
     - No data-dependent computations are allowed
     - Operations are data-value independent
     - Operations always operate on the same numbers of data items
   - Express behaviors of multi-rate functions
   - Formulate synthesis problems as optimization problems
   - Propose solution methods to these problems
2. Develop solutions techniques for the synthesis problems
3. Demonstrate the feasibility of a synthesis methodology based on the above framework, the solution methods, and known scheduling heuristics.

It is important to emphasize here that the techniques and methods developed during this research, and presented herein, were oriented towards repetitive algorithms. Different techniques would be required for non-repetitive algorithms.

The result has been a theoretical framework supporting an architectural synthesis process for repetitive, multi-rate algorithms, and a methodology based on this framework. In Section 1.4, the approach taken in this research is outlined. Before that, other related work is reviewed.

1.4 The new synthesis approach and problem definition

This section gives a brief overview of the new synthesis approach developed in this dissertation.

The new approach (i.e. the approach developed herein) to the problem of developing a methodology for synthesis of repetitive multi-rate algorithms, using multi-rate components, involves the development of a theoretical framework to support expression of multi-rate algorithms and their translation to efficient architectures. The framework can be divided into the following aspects:

- A representation basis for expressing algorithms
- A new model of execution for hardware components
- Definition of an architectural schema
- Formulation of the synthesis problems as constrained transformation from algorithms expressed in the representation basis to an architecture.
- Development of certain analytic techniques to assist in the synthesis process.

In the following paragraphs, these aspects are touched upon in greater detail.

The representation basis chosen is called the Data Dependency Graph (DDG). A DDG is a data flow graph augmented to provide a mechanism for specifying multi-rate operations. The computational model is assumed to be *data-driven data flow.*

The user specifies an algorithm by drawing a DDG and providing the necessary specifications for its nodes and edges. Such direct graphical representation eliminates the need for a procedural language to express algorithms and the translation step of transforming the input specification into an intermediate graphical form. It also has the advantage of being able to express the available parallelism.

The user specifies the *repetition rate* of the algorithm by indicating the inverse quantity called the *input latency.*

It is assumed that each operation in the algorithm is executed by a unique type of hardware component. This mapping of computational operations to components is provided by the user. On the other hand, throughout this work it is further assumed that the communication operations are executed via busses.

A DDG, for which the input latency is specified and an operation-to-component mapping has been provided, is referred to as an *annotated-DDG.*

The new model of execution for hardware components is a deterministic, multiple clock-cycle model, with phases for input, operation execution, and output, which may perhaps overlap each other. This model supports combinational and internally sequential components, and provides a satisfactory model for multi-rate components as well. Furthermore, it is compatible with the computation model of multi-rate functions and can thus be used for their analysis.

It is assumed that target architectures are synchronous (i.e. using a global system-wide clock), and are controlled via a ROM-based control program. The control program is obtained statically by the synthesis process. The control ROM is addressed via a counter.

Using the DDG-specification of the algorithm, the execution model of the components, and the user-provided input latency, analysis is carried out to determine the number of executions of each node of the algorithm, and in the case of acyclic DDGs, also the numbers of components required in the architecture. Following this the synthesis is carried out, which involves the steps of *scheduling* and *assignment of operations to hardware components.*

The last two steps, namely analysis and synthesis, are part of the automatic synthesis system.

The overall synthesis process appears as shown in Figure 1.2.

1.4.1 Synthesis problem for multi-rate algorithms

In the light of the framework outlined in the preceding, the synthesis problem can be stated as follows:

For a given multi-rate repetitive algorithm, given that

- *each operation in the algorithm is data-independent,*
- *each operation is executed on a unique component, and*
- *communication operations are executed via busses.*

Assuming that the cost of the architecture is computed in terms of the total number of computational components and busses, and the connectiv-

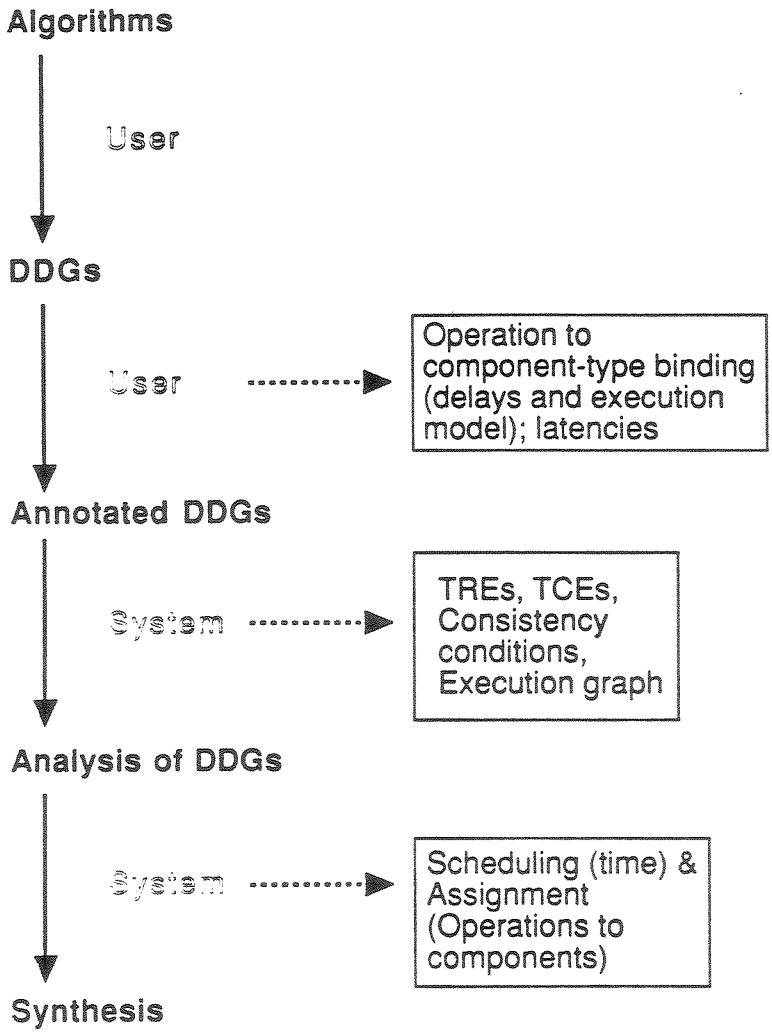


Figure 1.2: The synthesis process

ity of the architecture (i.e. number of connections between the computational components and the busses),

- *for a cyclic DDG, establish the lower bound on the input latency (minimum input latency).*
- *given the input latency for the algorithm, find a minimum cost architecture*

In the next section is given a brief outline of the other chapters of this dissertation. The section following it contains a summary of the contributions of this research.

1.5 Outline of the dissertation

Chapter 2 contains a formal definition of multi-rate functions and algorithms. It further explains the concept of a *shared port* and shows the similarities between the implementation of a multi-rate function and a multi-rate component using shared ports. It contains motivational arguments for choosing a multi-clock scheme over the systolic scheme for efficient synthesis for multi-rate algorithms.

The following chapter, Chapter 3, contains the definition of the Data Dependency Graphs, the algorithm representation basis of the new approach. It also contains the definition of the associated computational model, the definition of the execution model for hardware components, and the specification of the general structural characteristics of the architectures generated by the synthesis process. After these are discussed, the stage is set to compare the new approach with some other approaches with respect to their effectiveness and the general architectural characteristics.

Chapter 4 describes a number of analytic techniques to which an algorithm specification can be subjected, to obtain design information, and to assist in the synthesis processes of scheduling and assignment. Specifically, the techniques include the *token rate equations*, the *token count equations*, the *execution graph* of the DDG, and the *equivalence transformations*. The chapter also contains a discussion of initial tokens in directed loops of a DDG. A necessary and sufficient condition for guaranteeing the absence of deadlock in a directed loop is given there.

Chapter 5 addresses the two synthesis problems for multi-rate algorithms enunciated in the last section. In particular, the synthesis for cyclic DDGs is resolved into multiple sub-cases, and each sub-case is considered separately. The general case is a DDG with multiple interconnected loops. For this general case, the problem is known to be NP-hard and is formulated as an integer programming problem. The special case of an acyclic graph is solvable in polynomial time, and a scheme is presented to achieve the same.

For the special case of DDGs with *independent* loops, the complexity of the synthesis processes can be reduced by a *divide and conquer* heuristic. In it, the scheduling of a loop is done independently of that of the rest of the DDG, albeit at the risk of forsaking the global optimal solution. In the case of the DDGs with *nested loops* a similar heuristic approach is possible, in which a loop is scheduled without regard to the loops nesting it and the loops it nests. However, this is only possible if the loops satisfy the condition of *uniform criticality*. This condition is stated in Section 5.5, and there it is shown that the nested loops that satisfy this condition, can be correctly scheduled in the manner stated above.

The special case of a DDG with an outer loop is equivalent to the DDG

without the loop, but on which *graph latency* constraints have been imposed. This problem is NP-hard, and for it an iterative algorithm is proposed.

Chapter 6 contains description of the implementation of a fast, heuristic synthesizer for acyclic DDGs. Specifically, the chapter contains discussions of the various heuristics and their performance. It also describes the extensions made to the basic synthesizer to make available a limited ability to explore the architectural solution space. The chapter also contains a demonstration example of an elliptical filter.

Chapter 7 contains illustrations of some desirable extensions to the present methodology. The contents of the chapter are intended to point to future directions in which the present research may be carried.

Finally, Chapter 8 summarizes the findings of this research and contains concluding remarks.

1.6 Research contributions

This research has focused on the development of a theoretical framework to support the synthesis of multi-rate algorithms and the use of multi-rate components, and on the development of a methodology based on this framework. Following salient contributions are reported herein:

- **Formal definition of multi-rate functions and algorithms.** Also, it is shown that a non-systolic multi-clock architectural scheme is more efficient for synthesis of multi-rate algorithms.

- **Shared port as a source of multi-rate behavior.** It is recognized that the increasing levels of integration lead to shared port components, and therefore multi-rate behavior.

- **Unified representation for multi-rate behavior.** It is recognized that multi-rate components can be abstracted as multi-rate functions as well, and that both can be analyzed formally in an identical manner.

- **Data Dependency Graphs (DDGs).** The DDGs are used as a single *unified* means for specifying a multi-rate algorithm. The DDGs also permit the explicit inclusion of data communication operations. A data-reordering communication primitive called the SR-communication node is introduced to facilitate the specification of multi-rate behavior of components in a DDG.

- **Execution model for hardware components.** An execution model suitable for multi-rate components is formulated. This model can capture the behavior of computational components as well as communication components. These components may have execution delays that are single or multiple clock cycles long, and may be combinational or sequential.

- **Analytic techniques to assist the synthesis process.** The techniques include:

  - **Token rate equations.**
  - **Token count equations.**
  - **Execution graph.**
  - **Equivalence transformations.**

- **Formulation and solution of synthesis problems stated in Section 1.4.1.**

- **Extensions.** Extensions are proposed to synthesize architectures for

1. Algorithms incorporating data-dependent computations.
2. Multiple algorithms combined via *count dependencies.*

- **DDGTool and a heuristic synthesis system.** The synthesis methodology based on the above concepts is demonstrated by implementing a graphical front-end, called DDGTool, to draw DDGs, and a heuristic synthesis system. The heuristics are introduced for scheduling of operations and for assignment of pre-scheduled operations to components to minimize the connectivity of the architecture.

Chapter 2

Multi-rate algorithms

As seen in the previous chapter there has been an intense interest in the area of high level synthesis of hardware pipelines, and increasingly more complex algorithms are being implemented in hardware. The systolic design approach [32] and the design approach of Sehwa [44] represent two systematic and formal approaches to the area of synthesis of efficient pipelined architectures. Although these two approaches have been applied to diverse application areas, as was indicated in the previous chapter, there are algorithms which fall outside their domain. In this chapter, this extended domain of application is specified; its implications for the design process are also discussed.

2.1 Multi-rate functions and algorithms

Algorithms that are executed repeatedly fall into two broad categories: single rate, and what will be defined here as multi-rate. The first can be informally described as one in which the data rates remain the same along any path in the computation graph of the algorithm. The latter can be described as one in which the data rate may change along a path in the algorithm graph.

In multi-rate algorithms, the change in data rate may be functional in origin, meaning that it is part of the specification of the computation. As indicated in the last chapter, multi-rate computations occur in various application areas such as signal processing, adaptive controls, graphics, etc. For example,

in decimation filtering — a well-known signal processing algorithm, the amount of data to be processed is reduced by eliminating part of the input data, and thus the reduction in data rate is part of the computational specification.

On the other hand, the data-rate variation may have an implementational origin, meaning that it may be dictated by its hardware design. In either case, the effect is the same, the candidate algorithm available for the design process becomes multi-rate.

In this section multi-rate functions and algorithms will be formally defined. The concept of a *shared port* is presented as the source of multi-rate behavior in hardware components.

2.1.1 Definitions

Formal definition of a multi-rate function is given first. For the present, attention will only be on computations with single input and output domains. But as will be seen later, the concepts are readily extensible to those with multiple input and output domains.

Single input and output domain functions

Definition 1 *A relation* R *over sets A and B (from A to B, denoted as* $A \longrightarrow B$*) is a set of ordered pairs* (a, b)*, where a and b are sets and* $a \in A$ *and* $b \in B$*.*

Since the interest here is in the computational aspects of algorithms, the sets a and b will be assumed to be sets of primitive data items. Note that a and b are sets of primitive elements. Often, in the literature, a and b

are themselves primitive elements. The definition given here reduces to the traditional one when sets a and b each include just a single element.

Definition 2 *A function* F *is a relation which satisfies the property:* $\forall a$, $\forall b_1, \forall b_2\ [(a, b_1) \in \mathrm{F} \wedge (a, b_2) \in \mathrm{F} \Longrightarrow b_1 = b_2]$.

Definition 3 *A function* F *from* $A \longrightarrow B$ *for which* $\forall (a, b) \in \mathrm{F}$, $|a| = m, |b| = n$, *is denoted as* $\mathrm{F}^{m;n}$.

Definition 4 *A function* $\mathrm{F}^{m;n}$ *is* p-input-unique *if* $\forall a_i, a_j \in Domain(\mathrm{F}^{m;n})$, $|a_i - \bigcup_{j \neq i} a_j| = p,\ p \geq 1$.

Definition 5 *A function* $\mathrm{F}^{m;n}$ *is* q-output-unique *if* $\forall b_i, b_j \in Range(\mathrm{F}^{m;n})$, $|b_i - \bigcup_{j \neq i} b_j| = q,\ q \geq 1$.

A p-input-unique, q-output-unique $\mathrm{F}^{m;n}$ is denoted by $\mathrm{F}^{m;n}_{p;q}$. Here consideration will only be given to functions which have m=p and n=q. For brevity these functions are denoted as simply $\mathrm{F}_{p;q}$. These functions effectively operate on unique sets of input data (a_i's), such that the individual data values are not shared between the domain sets a_i. Similarly, the output images b_is do not share data elements.

In general, p and q may be different, although it is certainly possible to have p = q.

To define multi-rate functions, the concept of time must be introduced into the last definition. This is done by asserting the following:

For $\mathrm{F}_{p;q}$, *each element of input set* a_i *is accessed at distinct time instant; the same is true for elements of the output set* b_j.

In other words, each operand belonging to a domain is accessed at a distinct time instant. Note that the above requirement does not rule out a concurrent access of an element of set a_i and an element of set b_j.

It will be convenient to assume that consecutive time instants are separated by a fixed *time-interval*, and that a primitive element of a domain is accessed one per time-interval. Thus, for $\mathrm{F}_{p;q}$, the input set occupies p time-intervals, and the output set occupies q time-intervals.

Single rate and multi-rate functions can now be formally defined:

Definition 6 *An* $\mathrm{F}_{p;q}$, *where* $p = q$, *is called a rate-invariant function.*

Definition 7 *An* $\mathrm{F}_{p;q}$, *where* $p = q = 1$, *is called a single rate function.*

Definition 8 *An* $\mathrm{F}_{p;q}$, *where* $p \neq q$, *is referred to as a rate-varying function.*

Definition 9 *An* $\mathrm{F}_{p;q}$, *where* $p = q$, *or* $p \neq q$, *is called a multi-rate function.*

Notice that one may define a multi-rate rate-invariant function as one for which $\mathrm{p} = \mathrm{q} \neq 1$ — rate-invariant functions leave the data rate unchanged.

Single and multi-rate algorithms can also be defined.

Definition 10 *An algorithm in which all functions are single-rate is called a single-rate algorithm.*

Definition 11 *An algorithm is called a multi-rate algorithm only if it is not a single rate algorithm.*

Notice that the definition of multi-rate functions allows p to be equal, or unequal, to q. However, of more importance in the next section will be the latter case of p≠q, i.e. the rate-varying functions. Thus, although the term "multi-rate" subsumes both rate-invariant and rate-varying functions, throughout the rest of this dissertation, unless explicitly stated, it will imply a function of the latter type.

Multiple input and output domains

A generalization to multiple input and output domains is straightforward.

A multi-rate function operating on multiple input and output domains is denoted by $F_{p_1,p_2,\ldots;\ q_1,q_2,\ldots}$. Here, p_1, p_2, ... are cardinalities of input sets a_1, a_2, ... from domains A_1, A_2, ... respectively, and q_1, q_2, ... are cardinalities of output sets b_1, b_2, ... from codomains B_1, B_2, ... respectively.

As in the case of single domain functions, for multi-domain functions the uniqueness of time of access is required only for inputs belonging to same domain.

A function $F_{p_1,p_2,\ldots;\ q_1,q_2,\ldots}$, is defined to be single rate if $\forall\ i, j, p_i = q_j = 1$.

A function $F_{p_1,p_2,\ldots;\ q_1,q_2,\ldots}$, is defined to be rate-varying if $\exists\ i, j, p_i \neq q_j$; if $\exists\ i, j, p_i \neq p_j$; or if $\exists\ i, j, q_i \neq q_j$.

Single and multi-rate algorithms have the same definitions as before.

In Chapter 3 a multi-rate function will be represented by a node in a directed graph. The input domain will be represented an input edge and the output domain will be represented by an output edge. And the integers p and

q will be associated with the input and output edges respectively.

2.1.2 Sources of multi-rate

There are two sources via which multi-rate behavior may become part of the hardware implementation of a function:

1. computational definition, and
2. implementational constraint.

Regardless of the source, the behavior of the implementation can be formally described with the same abstraction: $\mathrm{F}_{p_1,p_2,\ldots;\ q_1,q_2,\ldots}$. This unified representation permits the synthesis methodology to deal formally at once with algorithms that are computationally multi-rate (such as decimation), and with algorithms that are made to take on multi-rate behavior through the incorporation of predefined hardware-implemented functions.

Computational definition

A function may inherently be defined in terms of a time-series of data, as is the case for many signal processing functions. For these the inputs and outputs are defined as functions of time instants (e.g. sampled signals). A hardware implementation of this abstract definition of the function receives input data values over a number of time-intervals, and similarly, produces output values over multiple time-intervals. Now, if the cardinality of input set is unequal to that of the output set, the result is a multi-rate component.

This occurs, for example, for functions such as decimation and interpolation, and in serial-parallel converters, and are therefore characterized

as multi-rate functions. In these particular cases, the input and output data elements are also unequal in number; i.e., $p \neq q$. For instance, an 8-to-1 serial to parallel converter collects eight bits of data and produces a single byte. Such a converter can be described as $F_{8;1}$.

Implementational constraint: Shared port component

A port of a computational component is called a *shared port* when the constraint of sharing electrical conductors, in the form of pins, wires, or pads, imposes the requirement of *time-multiplexing* more than one distinct logical operand over it. When so used, the individual operands are transferred over different time-intervals, thereby imparting the component a multi-rate behavior.

As the VLSI technology moves toward higher levels of integration, there is a tendency for the computations cells to become increasingly complex and thus large. However, as the area of a cell increases, its input-output capacity increases approximately only as its area's square root — a consequence of its 2-dimensional implementation. Thus there is a growing necessity to share a port for multiple operands, for both input and output. The same phenomenon also occurs during hierarchical design of systems.

As an example of a component with shared-port induced multi-rate behavior, consider a multiplier implemented in VLSI which inputs two 16-bit quantities and produces a 32-bit output. Assume that the input and output pads are distinct. If for the sake of economy of pads, the two halves of the output are multiplexed on a set of 16 pads, a shared port component results. Such a multiplier behaves as a multi-rate function which can be described as $F_{1,1;2}$. If the two inputs were to be multiplexed as well, the component can be

specified as $F_{2;2}$, a rate-invariant function.

2.2 Why multi-clock design?

In the last section, a precise meaning was given to the term *multi-rate function*. In this section, reasons will be presented as to why, for multi-rate algorithms, formulation of synthesis techniques different from the ones used for single rate algorithms are required to obtain efficient architectures.

2.2.1 Need for multi-clock implementation

Functions, according to their mathematical definition, are memoryless, and thus would correspond to purely combinational circuits. In a combinational implementation of a function, an output in the range of the function is obtained by inserting all corresponding input data values at once. However, for multi-rate functions, $F_{p;q}$, the inputs arrive during distinct time-intervals. This requires internal storage and therefore sequential implementation which may be either synchronous or asynchronous.

It will be assumed throughout the rest of this dissertation that a synchronous style of design is selected for the sequential implementation.

The sequential style implies a global clock that is supplied to the component. The time-intervals, introduced earlier for defining multi-rate functions, can now be identified with individual cycles of this global clock.

Single-clock-rate-efficiency

Consider a synchronous implementation of $F_{p;q}$. Without any loss of generality, assume that the input arrives sequentially, at most one element per

clock cycle. The same is assumed to be true for the output. Note that some cycles are allowed, during which no elements may be input or output, and also during which the function is computing the image set b_i for an input set a_i.

Definition 12 *An implementation of* $F_{p;q}$ *is* single-clock-rate-efficient *(SCRE), if after a finite initial delay, it can accept a valid input element and produce a valid output element during every clock cycle. Otherwise the function is* single-clock-rate-inefficient *(SCRI).*

Notice the word "valid" in the above definition. The word is added to the definition to disallow inclusion of *null* (or *don't-care*) elements into the data stream.

A systolic system is a Moore machine which uses a single global clock to clock all its storage elements (registers) [36]. Each clock cycle, a systolic system inputs new valid values and, computes and generates new valid outputs. So it can be concluded,

Lemma 1 *A systolic system is SCRE.*

Consider a finite storage synchronous implementation of a multi-rate functions $F_{p;q}$. Assume that $F_{p;q}$ is SCRE. The implementation will be further assumed to be causal in the sense that $\forall a_i, a_j$, if a_i is input before a_j, b_i is output before b_j.

First assume that p<q. It takes p cycles to input an a_i and it takes q cycles to output the corresponding image b_i. Since $\forall a_i a_j, i \neq j \Rightarrow a_i \cap a_j = \emptyset$, a b_i can be computed as soon as the corresponding a_i is input, and may be output after a finite latency; and the a_i can subsequently be discarded. So storage requirements for only the b_is may be analyzed.

Consider the p·q clock cycles starting with the cycle during which the first element of an a_i is input. During these cycles q a_is are input, and this results in q b_is. Now consider the p·q cycles starting with the cycle during which the first element of the first b_i so produced is output. Only p b_i elements are output during this interval. The remaining q-p b_is have to be stored and output during subsequent cycles, increasing the delay after which the next sequence of q b_is are output. Thus there is a net build-up of delayed b_i elements requiring unbounded storage and latency, contradicting the finite storage and finite latency assumption.

If p>q, then similar analysis requires a diminishing latency for every subsequent b_i. Since latency is bounded below by zero (due to causality property of a synchronous circuit), the implementation is forced to introduce cycles during which no output takes place.

The above arguments may be expressed as the following observation:

A synchronous implementation of an $F_{p,q}$ *when* $p \neq q$ *is SCRI.*

The above observation is important because, when considered along with Lemma 1, it implies the following proposition:

Proposition 1 *A multi-rate function,* $F_{p,q}$, $p \neq q$, *is not efficiently systolizable.*

The proposition implies that if $F_{p,q}$, $p \neq q$ is implemented as a systolic system, the hardware will not be utilized efficiently. Or equivalently, if it is implemented as a systolic system, the execution will entail insertions of null elements in the data streams. An illustration of this proposition is presented in Section 3.3.

Typically, systolic designs are *efficient* in terms of utilization of hardware components, since during each clock cycle, every component does useful work. However, if an SCRI component is connected to a systolic sub-system, or, in general, an SCRE sub-system, the latter will be forced to be spend idle cycles, and therefore become inefficient. This leads to,

Proposition 2 *An architecture for an algorithm involving multi-rate functions, if synthesized using the same clock to clock all its storage elements at every clock cycle, does not utilize its hardware efficiently.*

The above proposition predicts that it will be inefficient to synthesize an architecture for the entire multi-rate algorithm as a single systolic array. The question therefore is: what alternative techniques may be used to obtain efficient designs?

The proposition hints that a non-systolic approach, using multiple clocks, might be more appropriate. Such a synthesis approach will be called a *multi-clock design.*

In the next chapter is introduced a suitable execution model for hardware components. There, abstract representations of hardware components are proposed and the semantics of these representations are discussed. The abstractions will be used to specify algorithms.

In this chapter the algebraic notation of $\mathrm{F}_{p,q}$ was used to represent multi-rate functions. In the subsequent chapters, an equivalent graphical representation is used instead. This representation is explained in the next chapter.

Chapter 3

Data Dependency Graphs, Architectural Models and Comparisons

The process of high level synthesis starts with a specification of the desired computation. A special type of graph, the Data Dependency Graph (DDG), is employed as a medium for this purpose. This chapter contains a detailed discussion about DDGs and their component parts.

The translation of DDGs to architectures requires a precise definition of the characteristics of the target architecture, the specification of primitives, and the model of execution assumed for hardware components, which will actually implement the operations of the algorithm. The architecture is envisioned to be synchronous, containing multiple computational components interconnected via register files to a communication structure comprising of multiple busses, and with the execution of the intended algorithm mediated by a static schedule stored in a control ROM. All these aspects, along with an example of a multiplier, illustrating the modelling of its behavior as a DDG node, are also covered in detail in the chapter. As will be seen, there exists a dual relationship between a DDG node and its counterpart in hardware — the former is an algorithmic abstraction of the latter, and the latter is an implementation satisfying the behavioral abstraction depicted by the former.

This chapter also contains a comparison with other high level synthesis paradigms, high-lighting the differences in the underlying execution and architectural models and the results of the methodologies based on these paradigms.

3.1 Data Dependency Graphs

A Data Dependency Graph (DDG) is used to express algorithms.

A DDG is a connected, directed, labeled, and specified graph, where nodes define computation or communication operations and directed edges represent data dependencies between pairs of nodes.

The use of directed graphs to specify computations for the purpose of automated design of architectures, and more generally for silicon compilation, is a familiar practice [16, 30, 26, 54]. In the area of automated synthesis, specifying computations directly in graphical form has a practical advantage over specifying them in textual form: one of avoiding the intermediate step of translation from a textual language to an internal graphical form, common in most synthesis approaches. As will be seen later in Chapter 4, the DDGs provide a further advantage of enhanced representation power consistent with the analytic techniques used for architectural synthesis.

The node at the tail of an edge is the source of data and the one at its head is the destination of that data. These nodes are called *parent* and *child* nodes, or alternatively *from-* and *to-* nodes, respectively. The edge represents an *input dependency* of the child- or to- node and an *output dependency* of the parent- or from- node. A node may have multiple input and output dependencies.

The nodes are classified as *start*, *stop*, and *intermediate*. The start nodes do not have any input dependencies and the stop nodes are without any output dependencies; the rest are the intermediate nodes. Although this definition allows a DDG to have multiple start and stop nodes, the discussion here will be restricted only to single-start—single-stop DDGs.

The nodes are presumed to have a data-independent behavior associated with them and so the DDGs, in general, also represent data-independent computations. Later in Chapter 7, however, there is described uses of DDGs to represent non-data dependencies as well as certain types of data-dependent computations.

The execution of the algorithm represented by a DDG is carried out in a manner similar to that of the traditional data-flow model of computation [28, 14]. The nodes execute via the *firing* of the corresponding hardware component, absorbing data elements or *tokens* from their input dependencies, and producing tokens over their output dependencies. Each execution of a node is referred to as an *invocation.* The tokens flow from the parent node to the child node over the dependency edge between them.

Although DDGs can, in principle, represent a computation which is executed once, it will be employed to specify a repetitive computation wherein the entire graph is repeated infinite number of times at some regular interval of time. This notion of regularity is refined in a later section.

3.1.1 DDG nodes

In a DDG, a node may represent either a computation or a communication operation. One of the key contributions of this research was to recognize the need for, and the possibility of, giving an explicit representation for communication operations in the form of communication nodes. Semantics are introduced into the communication primitives at the representation level so as to have expressive power adequate for describing hardware behavior, and yet be similar in form to computational operations. By so representing all data communication explicitly at the algorithmic level, it is possible to analyze all

types of operations, including communication, uniformly. This leads to better utilization of communication hardware in the final architecture.

A fully specified DDG node has associated with it a 4-tuple <o, I, O, F>, components of which are:

- o: Operation executed by the node
- I: The set of input token markings
- O: The set of output token markings
- F: Firing discipline of the node

Each of these components is now discussed.

Operation

The association of an operation with a node is specified by the user as part of the specification of the algorithm. A DDG may have two classes of node: computational and communication. Computational nodes represent arithmetic, logic, serialization/parallelization, and decimation/interpolation operations etc., and communication nodes represent communication operations between nodes of the first type. A communication operation is simply looked upon as a special type of computation in which there is no transformation of data. Throughout the rest of this dissertation, wherever relevant, computational nodes will be depicted by larger circles and communication nodes by smaller circles.

The operation represented by a node is treated as being *atomic*, irrespective of its internal complexity. That is, although an operation internally might involve a multi-step computation incorporating sub-operations more

primitive, when expressed as a node at the DDG level, it is treated as being non-interruptible, and irresolvable further into simpler sub-operations. This restriction in the semantic definition of a DDG node justifies the solution approach that each DDG node is mapped to a hardware component.

The operation is also assumed to be independent of the values of the operands. Besides being atomic, the operations represented by the nodes are assumed to not carry any state information that is relevant to the synthesis process from one invocation to the next. Thus an operation is also independent of its previous invocations.

Input and output token markings

An integer associated with each dependency denotes the number of tokens absorbed (in the case of input dependency) or produced (in the case of output dependency) during every invocation of the node. The integers associated with the input dependencies are called *input token markings* and those associated with the output dependencies are called *output token markings*. This definition of the behavior of a DDG node is similar to that of a node in the parallel computation model of Karp and Miller [28] and is the same as synchronous data flow graphs of Lee and Messerschmidt [34]. However, the semantics of the nodes is extended beyond those in [34] by the introduction of the concept of *firing discipline* discussed below.

As will be discussed shortly, each dependency has a *port-id* associated with it. The token-markings corresponding to dependencies bearing the same port-id's and related to similar but distinct nodes, must be the same.

All tokens corresponding to a given dependency are of the same size (or, bit-width), but tokens corresponding to two dependencies may have differ-

ent sizes.

Clearly, such a nodal specification of operations is suitable for representing multi-rate functions defined in the previous chapter. The integers assigned to the dependencies correspond to the subscripts p_i and q_j of a function $F_{p_1,p_2,\ldots;\ q_1,q_2,\ldots}$.

Firing disciplines

A *firing discipline* is a rule that states the conditions on input operands under which a node is allowed to fire. Two firing disciplines are defined: *AND* and *SR*. The formal semantic specification of the two disciplines appears in Figure 3.1.

AND discipline

The AND firing discipline requires that a node not fire until each of its input dependency carries tokens which in number are equal to or more than its token marking. All computational nodes are assumed to use the AND firing discipline.

A communication node using the AND firing discipline has only one input dependency and one or more output dependencies with token markings each of 1; such a node represents a broadcast communication.

SR discipline

The presence of multi-rate functions, arising either from computational definition or from implementational constraint, may make it necessary at times to specify a particular ordering of operands. For example, consider a component for the *modulus* operation. For the sake of illustration, assume that

e_{I_i}	Input edge i
e_{O_j}	Output edge j
e_{I_s}	Selected input edge number
e_{O_s}	Selected output edge number
$e_{I_{max}}$	Maximum number of input edges
$e_{O_{max}}$	Maximum number of output edges
$t^a_{e_{I_i}}$	# of tokens available on e_{I_i}
$t^m_{e_{I_i}}$	token marking on e_{I_i}
$t^m_{e_{O_j}}$	token marking on e_{O_j}
$t^t_{e_{I_i}}$	# of tokens transferred from e_{I_i}
$t^t_{e_{O_j}}$	# of tokens transferred to e_{O_j}

AND

forever do

$\square$ $\forall$ i $t^a_{e_{I_i}} \geq t^m_{e_{I_i}} \longrightarrow$

$t^a_{e_{I_i}} -= t^m_{e_{I_i}}$;
Execute function;
$\forall$ j put $t^m_{e_{O_j}}$ tokens on e_{O_j};

od

SR

e_{I_s}=e_{O_s}=0; $\forall$ i $t^t_{e_{I_i}} = 0$; $\forall$ j $t^t_{e_{O_j}} = 0$;

forever do

$\square$ $t^a_{e_{I_s}} > 0 \longrightarrow$

$t^a_{e_{I_s}}$–;
Transfer token to e_{O_s};
$t^t_{e_{I_s}}$++; $t^t_{e_{O_s}}$++;
$\square$ $t^t_{e_{I_s}} = t^m_{e_{I_s}} \rightarrow (e_{I_s}$++) mod $e_{I_{max}}$;
$\square$ $t^t_{e_{O_s}} = t^m_{e_{O_s}} \rightarrow (e_{O_s}$++) mod $e_{O_{max}}$;

od

Figure 3.1: Semantic specification of the AND and SR firing disciplines.

it shares the input port for transferring divisor and the dividend and uses a separate port to output the modulus. If the order in which the two operands are input determines which one is treated as the divisor and which the dividend, it is necessary to specify the order in which the operands should be fed to the component. At the DDG level, this ordering is specified using a communication node with the SR firing discipline.

Only a communication node may be defined to have the SR firing discipline. The SR (standing for *serializing/reordering*) firing discipline, requires a node to "act" each time there is a token on an input dependency, selected according to a mechanism described below. The "action" is carried out by a firing of a communication component; i.e., a bus. Upon firing, the token is passed to the selected output dependency.

The selection of the input and output dependencies is ordered by their port-id's. Thus, at first, both input and output dependencies with port-id's of 1 are selected; then those with port-id's of 2 are selected; and so on. Each dependency remains selected for a number of firings equal to its token marking, before the next dependency in the port-id order is selected.

The token markings for a node with the SR firing discipline satisfy the condition that the sum of token-markings of its input dependencies equals the sum of token-markings of its output dependencies.

The semantic specification of the SR discipline is formally expressed in Figure 3.1.

It is important to note that, in functionality, the SR communication node is a generalization of the behavior achievable via an arbitrary interconnection of the *switch* and *merge* (or *distributor* and *selector*) data-flow language nodes with pre-determined control inputs [5, 14, 12, 37].

Components modeled by an SR-communication node

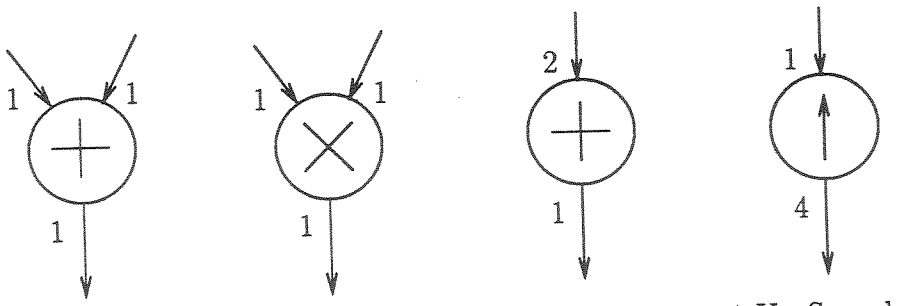
Throughout this dissertation, and in the implementation of a heuristic scheduler described herein (Chapter 6), it is assumed that the type of communication components used in the architecture is a *bus* (discussed in Section 3.2.1). In the context of a bus, the SR-communication node indicates ordering of data transfers over the bus. However, the choice of the communication component need not be limited to a bus; multiplexers and demultiplexers may also be used. An SR-communication node, with multiple input dependencies and a single output dependency, also accurately models the behavior of a multiplexer; a demultiplexer can be modeled by an SR-communication node with single input dependency and multiple output dependencies.

Figure 3.2 contains examples of some types of nodes that might be used in a DDG. The figure contains two types of "+" nodes, one with two input dependencies and the other with only one input dependency but which accepts two input tokens. Thus the choice of the representation used in a DDG is dictated by the component chosen for implementation, and vice versa. The figure also contains examples of the two types of communication nodes, with SR and AND firing disciplines just described.

3.1.2 DDG Edges

The edges of a DDG represent data-dependencies between pairs of nodes. The transfer of data along an edge is constituted by the *first in first out* (FIFO) flow of tokens over it. Thus when the *from* node fires, the tokens produced by it queue up behind the tokens already present on the edge, and when the *to* node fires, it absorbs tokens from the front of the queue.

Since nodes themselves do not carry any state information, it is con-



Some Computation Nodes

Communication Primitives

Figure 3.2: Examples Of Nodes

tained wholly in the values of the tokens present on the edges, the firing of the nodes causing the state transitions.

As mentioned earlier, a node may have multiple input and output dependencies. Each input dependency is semantically identified by assigning it a unique positive integer called the *input port-id.* The input dependencies bearing the same semantic relationship to two distinct DDG nodes representing an identical operation on different data, must have the same input port-id's associated with them. The reverse is also true. By convention, port-id's are unique positive consecutive integers This requirement also holds for output dependencies; the corresponding ports are given *output port-id's.*

The above semantics are meaningful in the light of the fact that a DDG node is mapped to a hardware component, that each of its dependency corresponds to the data accessed via a port of the component, that a port is identified by its distinct port-id, and that the dependency is ordered accordingly and has an associated port-id.

Edges may be assigned initial tokens by associating a *non-negative* integer with them. The integer denotes the number of tokens available at the start of the algorithm for the particular to-node to absorb. (In the graphical form, these tokens are shown alongside the edge.)

The values of these tokens are not important to the synthesis process; their number is of consequence, however. Later, it will be shown that for cyclic DDGs, the number of initial tokens present on the edges of a directed loop will determine whether the loop will deadlock or not, and that they also determine the minimum possible value of the input latency.

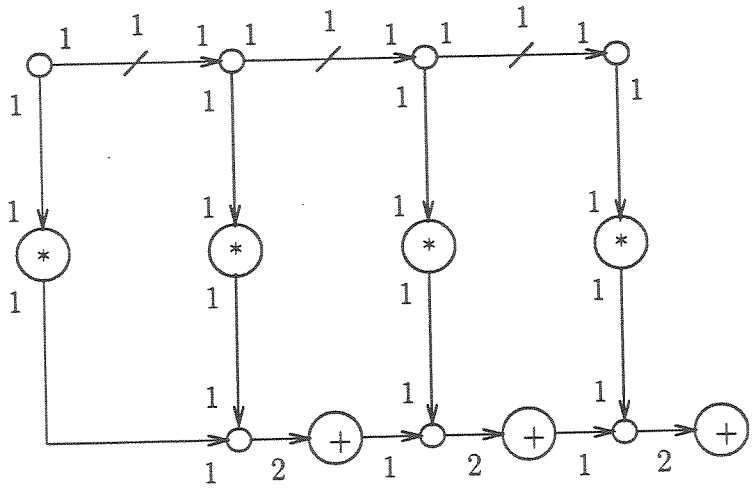


Figure 3.3: DDG for a 4-point FIR

3.1.3 An Example of a DDG

Figure 3.3 contains a DDG for a 4-point FIR computation. There are several interesting features to the DDG. Firstly, the "Add" nodes have only one input dependency on which both input operands flow. These are an example of shared-port components. Secondly, as discussed above, these necessitate the use of SR communication nodes. Also notice that every two computational operations are interposed by a communication operation. This is imperative since a uniform treatment to communication and computation nodes is intended, and as such must explicitly specify all data communication operations between computational operations. Further notice that, unlike the traditional signal flow graphs popular in signal processing literature, there are no sample delays, instead there are initial tokens on some of the edges. The multiplers have a single input dependency which inputs a single token — the other operands, the coefficients, are assumed to be fixed and not explicitly shown.

3.2 Architectural Models

The DDGs define the domain of the synthesis process. In this section the co-domain of the synthesis process, the expected parameters of the final architecture, is discussed. With a parametric definition of the domain and co-domain in hand, the synthesis process (i.e., the translation from a DDG to an architecture) can be formulated.

This section is divided into two parts. In the first, the overall system characteristics are discussed, and in the second an execution model for hardware components is proposed. The latter deals with modeling of behaviors of hardware components as multi-rate functions and contains the illustration of a shared output port multiplier.

3.2.1 Architectural Characteristics

Synchroneity of architectures and global clock

The architectures generated will be *synchronous*. That is, there will be a single time-base in relation to which all events in the system can be specified. Clearly, self-timed sequential circuits, or asynchronous communicating circuits such as arbiters and phase locked loops, cannot be considered. The synchroneity of the architecture will be achieved via the presence of a single, (possibly, non-overlapping two phase,) fine-granularity *global clock*. All events will be scheduled to occur at the "ticks" of this clock.

The architecture will contain combinational and sequential circuits. And by construction, an occurrence of a purely combinational loop will be prevented by introducing hardware buffers at the input of every hardware component, computation or communication. This is not an artificial measure, but, on

the contrary, is in conformance with the computation model of a node which requires that it not fire until *after* the necessary input tokens are available on the input edge.

Buffers

The buffers to hold the input tokens are in the form of edge-sensitive registers which may be clocked by clock edges derived from, and coinciding with, the edges of the above-mentioned global clock. These registers will be assumed to have a high-impedance state at their outputs when the output is not *enabled.* An MSI example of such registers is the 74374 circuit [56]. However, multiple of these registers are grouped together into *register-files* with an addressing mechanism to select only one of the group. The inputs and outputs of all the registers in a single register file are respectively connected together; thus at any given time only one of the registers of a register-file can be written or read. But a read and a write of the same register or different registers of a single register-file may proceed simultaneously.

It will be assumed that the bit-width of the registers is equal to the bit-width of the data it stores; i.e., each register stores exactly one token. But since tokens corresponding to different dependencies may have different sizes, the corresponding registers may have different sizes. So by convention, the grouping of the registers will be such that all the registers in a register-file will have the same size. Since all tokens flowing over a given edge have the same size, the registers may be grouped by the edge they correspond to.

As said previously, buffers are inserted at the input of every hardware component. Since every edge is input to some component, this is equivalent to saying, every edge corresponds to set of buffers (i.e., a register file). It is logical

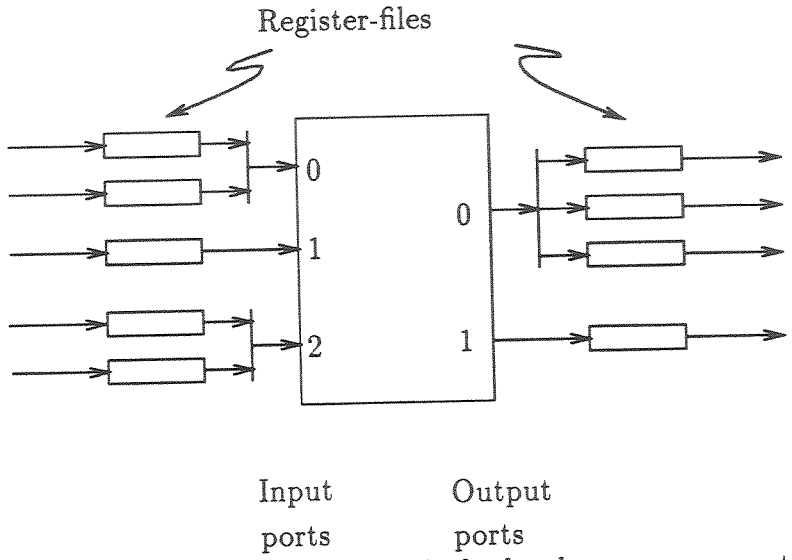


Figure 3.4: Connectivity model of a hardware component.

that every edge be translated to a set of buffers because it is on the edges that the tokens constituting the state of the computation reside and storage devices are required to store them.

In general, an edge gets translated into wire connections, via one or more associated register files, to an input port of the respective computational or communication to-node. However, it is more convenient to associate register-files with the ports of only the computational components, each register-file corresponding to a unique pair of a port and a communication component. This leads to the connection model of a component shown schematically in Figure 3.4. This model will be used later in Chapter 6 while dealing with the problem of minimizing total connectivity in the architecture for a given schedule.

Components

The architecture, in general, will contain multiple instances of a component type, including the communication components, which are shared by

the operations of the algorithm. Each operation of the algorithm is mapped one-to-one to a type of physical component that can perform it, and as discussed earlier, this mapping is provided by the user at the DDG level as part of the specification (o) of the node. Also, a given component type implements exactly one type of operation.

A firing of a node in the DDG corresponds to an execution of its operation by a hardware component. This execution is assumed to require non-zero time (or *delay*) which is independent of the values of the operands (or alternatively, data elements or tokens) involved.

The DDG may refer to multiple types of communication operations, and hence the architecture may have multiple types of communication components. However, the current research has focused on a single type of well understood and easily sharable communication component — a *bus*[1], and on the communication capabilities afforded by it. As discussed in Section 3.1.1, both AND and SR types of communication operations are generally needed, and these a bus can support.

The DDG may refer to multiple types of busses, different busses distinguished by their bit-widths. The bit-width of a bus is assumed to be the same as the bit-width of the tokens it transfers. Thus each token takes a single invocation of the respective communication operation. Different busses may be used to transfer different sized tokens and although all examples cited herein refer to a single bus size and a single token size, it is not an inherent limitation.

A bus is assumed to be a passive device used to carry signals from a

[1]As noted earlier, the SR-communication node can also represent the behavior of multiplexors and demultiplexors.

source to one or more destinations. Typically, a bus will be in the form of a group of wire-connections from the source buffers with high impedance outputs which might be transferring tokens on it and with input connections to the destination buffers. Any other technological requirements such as terminations and drivers will be of no concern here. A communication operation is executed over a bus by enabling the source buffer for output and the destination buffer(s) for input. A communication operation may, in general, occupy more than one cycle of the global clock, although in all the examples used in this dissertation it is always assumed that every communication operation can be completed within a single clock cycle.

Pipelining and latencies

As stated in the previous section, the computation represented by a DDG is assumed to be repetitive and consequently is repeated at regular intervals. If the execution time for a single invocation of the DDG computation is longer than the *period* of its invocation, operations corresponding to more than one invocation execute simultaneously necessitating a *pipelined* execution of the computation. However, unlike in the traditional pipelined or systolic systems, the architectures obtained will not contain registers clocked by the same clock, every clock cycle. Instead, they are clocked at different times to match the times of generation of the held tokens.

In a repetitive computation, the period of invocation of its start node, called the *input latency*, is specified. Or, instead, the inverse of this value, called the *sampling rate*, may be specified. As will be seen in the next chapter, for single-rate computations, such as signal flow graphs, the sampling rate also refers to the rate of invocation of every node in the graph. But, for multi-

rate computations, in general, this may not be the case, since the nodes may each execute different numbers of times. Hence, two distinct terms, L_I and L_S, which determine the periods of invocation of the input node and the entire DDG respectively, are defined below.

Input Latency (L_I): The definition of input latency is consistent with that used in the context of signal flow graphs: It is the period of invocation of the start node of a DDG and defines the interval, in terms of number of clock cycles, between the onset of two consecutive invocations of the start node.

Schedule Latency (L_S): The start node of a multi-rate DDG must be invoked certain minimum number of times (see Chapter 4) to complete a single invocation of the DDG computation. This minimum number is denoted by $N_{S_{min}}$. Schedule latency is defined to be $L_S = N_{S_{min}} \cdot L_I$. This definition makes L_S the period of initiation of a DDG computation.

Schedule Cycle Length (L_{Sch}): It may be desirable to compute a schedule for more than one initiation of a DDG. This may correspond to N_S invocations of the start node, instead of the $N_{S_{min}}$ above. (As shown in Chapter 4, it turns out that N_S must be an integer multiple of $N_{S_{min}}$.) The *schedule cycle length*, L_{Sch}, is defined to be $N_S \cdot L_I$.

The repetitive schedule implemented in the architecture is periodic with L_{Sch}. That is, from one period of L_{Sch} clock cycles to the next, the times of execution for invocations of nodes are identical modulo-L_{Sch}. The times are not required to be periodic with respect to any smaller period, however. Thus, although DDG computations are initiated periodically every L_S clock cycles, the overall schedule is not periodic with it as the period. Which means,

although the invocations of the start node are periodic with a period of L_I, (and, therefore, of course, L_S,) invocations of other nodes might not be. Consequently, the invocations of the output node of the DDG might not be periodic with a period of L_I either. If this aperiodicity of the output is undesirable from a practical point of view, however, additional buffers can easily be inserted to achieve the periodicity.

ROM based static schedule

It has been assumed that the computation represented by a DDG is data-independent. Under this assumption, it is possible to compute *statically* (i.e., pre-compute) the execution schedule for the DDG. Such a schedule will specify the starting times of executions of individual invocations of all computational as well as communication operations. Further, given that the operation of a component is independent of data values, the sequence of micro-orders to control its operation can be statically specified. Thus, the schedule control information can be combined with the sequences of micro-orders for various components to obtain times at which each micro-order is issued. The result is a micro-program which has no data-dependent branches. This program, therefore, can be stored in a ROM and accessed by merely sequencing through its addresses in consecutive order. Each address corresponds to a cycle of the global clock and contains micro-orders that are to be issued during that cycle. The size of the ROM is L_{Sch} locations.

Figure 3.5 shows a schematic representation of the architecture which will result from the design process. The box labeled "Components" forms the execution hardware structure for the algorithm. The output of the control ROM asserts or negates the various control points. Here, one bit per control

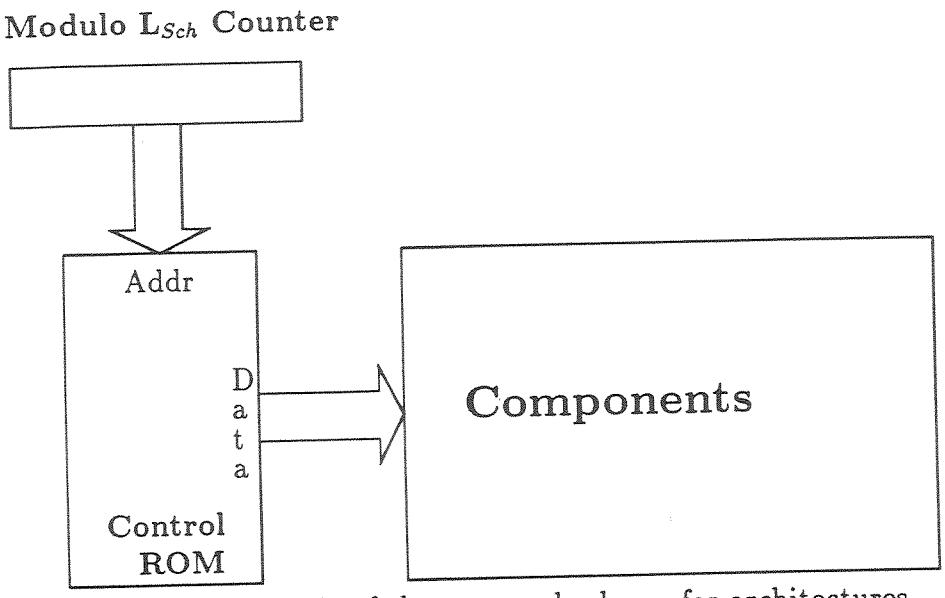


Figure 3.5: Schematic of the proposed scheme for architectures.

point has been assumed to produce a completely horizontal micro-program structure. A modulo-L_{Sch} counter is used drive the address lines of the ROM.

3.2.2 Execution models of hardware components

This sub-section introduces an execution model for hardware components. The execution model is applicable to both combinational and internally sequential single-rate and multi-rate components. The description of the execution model is followed by an illustration where an off-the-shelf component is modeled as a multi-rate function and the dual aspect of applying appropriate control signals to the component to make it behave according to the model is illustrated.

Execution model for hardware components

The choice of an execution model for hardware components is motivated by the following considerations:

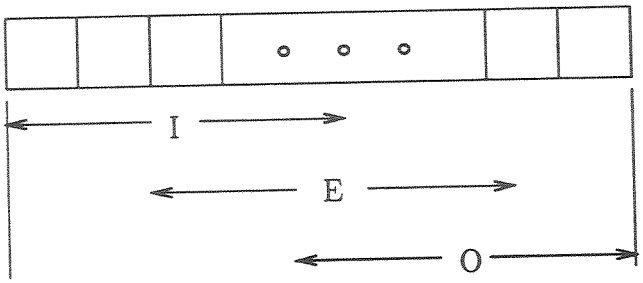


Figure 3.6: The proposed execution model for a hardware component.

- The model must be deterministic, statically defined, and uniform for all components.
- The model must be multi-cycle to accommodate multi-rate functions and components (See Chapter 2).
- The model must be compatible with the abstract model of computation of node. The abstract model of the firing of a node in a DDG expects a node to fire only upon the availability of all input tokens, upon firing to absorb *all* input tokens, and subsequently to produce *all* output tokens.

Any model that satisfies these requirements is acceptable.

The proposed execution of a hardware component is comprised of three phases: *input* (I), *execution* (E), and *output* (O). These phases may overlap as shown Figure 3.6. The execution begins with the start of the I-phase and ends with the completion of the O-phase. The phases always occupy one or more consecutive cycles of the global clock.

For an internally sequential component, a token is input over an input dependency every clock cycle of the input phase until all the necessary tokens are input, and a token is output over an output dependency every cycle of the output phase until the necessary tokens have been output. Thus, the length of

the input phase in clock cycles equals the maximum number of tokens input on any input dependency, and the length of the output phase equals the maximum number of tokens output over any output dependency.

It is easy to see how the above model is adequate for the three types of components mentioned previously. For combinational components, all three phases completely overlap each other. This model covers both single and multiple cycle execution delays. For a single-rate internally sequential component, both input and output phases are single clock cycle long. For multi-rate components, the input and output phases have different lengths.

The *execution delay* of a DDG node is defined to be that of the execution of its operation by a component. The latter is defined to be the number of cycles from the start of the input phase of the execution to the end of its output phase.

For any computational node or an AND communication node this definition poses no problem, since, as discussed in an earlier section, a communication node behaves like a combinational computation node. But, for an SR communication node, a single invocation of the node involves multiple firings, one for each token transferred, with idle clock cycles between them. Nevertheless, a definition for the delay of an SR communication node that is consistent with the interpretation of a single invocation of the node is necessary. Therefore, the execution delay of an SR communication node is defined to be: *the product of the delay for a single firing and the sum of the token markings on its input (or output) dependencies.* Clearly, because of the intermediate idle cycles, an invocation may extend over more cycles than those defined by its delay, but it will never occupy fewer cycles.

Example

Two related aspects of multi-rate abstraction of hardware components are now discussed with the help of an illustration.

The first aspect of abstraction has to do with being able to describe hardware components as multi-rate components with a timing behavior model of Figure 3.6. The second aspect is of controlling a hardware component so that it behaves according to the execution model of a multi-rate component just described. The discussion is carried with the help of an example of an off-the-shelf multiplier chip, VL2044, from VLSI Technology Inc. The chip is illustrated in two different multi-rate configurations. Although the example used is one of a predesigned component, the same considerations can go into designing a new component, in which input and output signals are multiplexed over the scarce external connectors.

VL2044 is a single chip multiply-accumulator circuit with flexible input and output controls. (The block diagram of the circuit is displayed below in Figure 3.7.) There are two parallel input ports, X and Y, for injecting two 16-bit quantities and a 32-bit port for the output. The inputs can be clocked independently using two separate clocks CLKX and CLKY respectively. There is a separate clock, CLKP, for the output. There are other inputs to control the functionality of the circuit, but since these inputs are not relevant here, they will be ignored — it will be assumed that these inputs will be set so that the circuit performs a 16-bit$\times$16-bit multiplication to produce a 32-bit product.

In the next figure, Figure 3.8, are shown timing relationships between different signals. The set-up times for the two inputs are 25 ns (min.), so a system clock time of 40 ns is chosen. This will provide for other signal propagation delays as well. The clock period of 40 ns corresponds to a clock

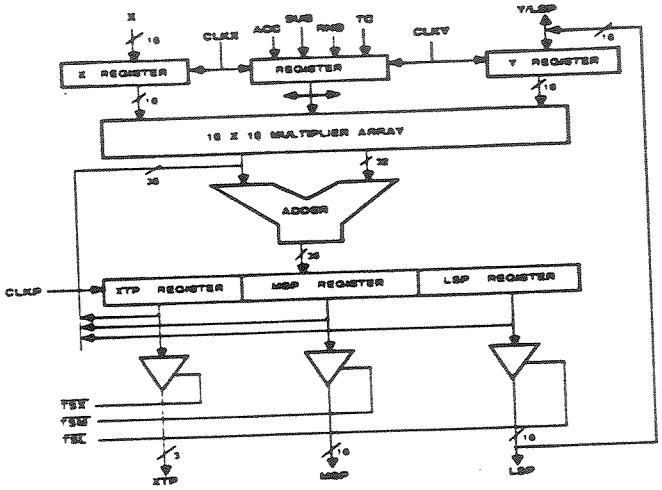


Figure 3.7: VL2044 Schematic.

frequency of 25 MHz, an easily achievable frequency for today's technology.

For the sake of this illustration, a multiplication time of 90 ns is assumed.

Let both configurations have a single output port. From specifications, it is found that the least significant 16 bits of the output are physically connected to the same pins over which the Y input goes in. The most signifi-

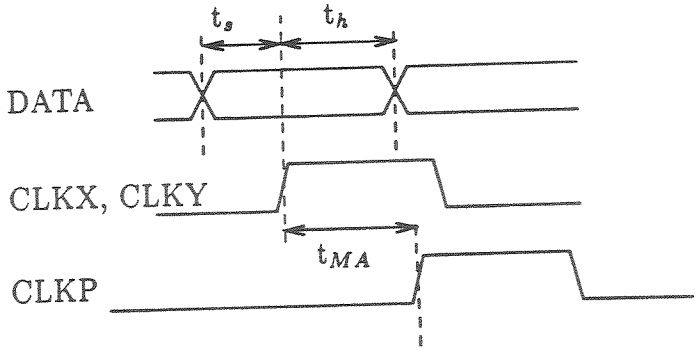


Figure 3.8: VL2044 Timing diagram.

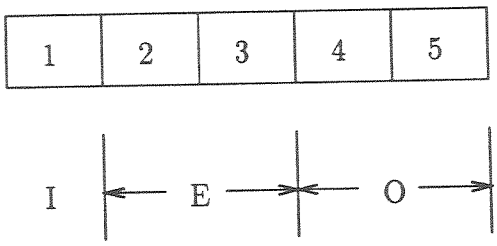


Figure 3.9: Execution model for the first configuration.

cant part of the input, however, has separate output path. By choosing to have a single output port of 16 bits, a choice is made to multiplex both 16-bit halves of the output over the same port as one of the inputs, specifically, the Y input. This choice, of course, does not pose a serious problem in terms of abstracting the multiplier, and it is also compatible with the abstract connectivity model of the multiplier.

Why choose to multiplex the two halves of the output when the chip already has two separate ports? The reason a designer might want to make such a choice is because it would reduce the number of busses to which the multiplier has to be connected — that is, assuming only 16-bit busses are available — and this would mean less area for communication. This is a reasonable choice, if the time penalty of multiplexing does not outweigh the area penalty otherwise, and if the multiplier is not to be used in a pipelined manner. It will be assumed here that both of these conditions are true.

Let the first configuration be the one in which both the inputs, X and Y, are fed simultaneously (and in parallel). Given this choice, the abstract behavior of the multiplier will be as shown in Figure 3.9. The behavior has an input phase of a single clock cycle, followed by an execution phase of two clock cycles, followed by an output phase of two clock cycles.

To make the multiplier behave in the fashion of Figure 3.9, the fol-

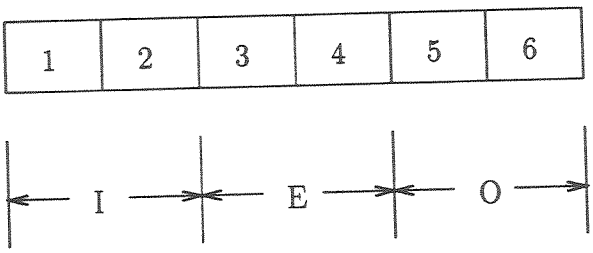


Figure 3.10: Execution model for the second configuration.

lowing control signals must be asserted.

End of cycle 1: CLKX and CLKY

End of cycle 3: CLKP

During cycle 4: TSL

During cycle 5: TSM

For the second configuration, the two inputs are multiplexed as well. The corresponding abstract behavior is shown in 3.10. The only difference in the two behaviors is that in the second configuration, the input phase is two cycles long. The corresponding control signals will be:

End of cycle 1: CLKX

End of cycle 2: CLKY

End of cycle 4: CLKP

During cycle 5: TSL

During cycle 6: TSM

Notice that in defining the control signals, certain signals are specified to be asserted throughout the cycle while certain others, only at the end of the cycles. This is a common practice in hardware design and an equivalent effect is easily achieved in VLSI using two-phase clocks.

3.3 Comparison with other synthesis systems

In the last chapter synthesis approaches of other related research efforts were reviewed. This is an appropriate place to compare the new approach to some of these approaches.

As will be seen in the following, none of the methodologies considered deal explicitly and formally with the concept of multi-rate functions and shared port components. Often, the assumed model of execution of hardware components is limited to single clock cycle execution, which rules out both internally sequential components and combinational components with multi-cycle delays. Most effort in these systems is in the area of providing a language to express algorithms and an environment to automatically or semi-automatically translate the algorithm into an architecture. The languages provided are often procedural and textual. The latter is an important difference between the present approach and these other approaches in that the input language used by the present approach is graphical. The major task in these systems is the translation of the algorithm specified textually in the input language into an intermediate graph-oriented form, which is then scheduled and its operations are assigned to hardware components.

Here, a comparison is made of the present approach and of the characteristics of architectures generated by it with other hardware design paradigms. Of theoretical interest is the comparison with the systolic approach and of interest in terms of efficiency of design is the comparison with the synthesis approach of Sehwa [44]. Finally, Cathedral I [25], II [13] and III [41] are compared from methodological point of view. An overall methodological comparison is presented in Table 3.1.

Parameter	Systolic	Sehwa	Cathedral I	Cathedral III	New
Component Delay Model	1 Clock cycle	$\leq$ 1 Clock cycle	1 Clock cycle	1 Clock cycle	Multiple clock cycles
Multi-rate Components	No	No	No	No	Yes
Multi-rate Computations	No	No	No	Ad hoc	Yes
Treatment of Communication hardware	Dedicated hardware	Handled separately	Handled separately	Handled separately	Treated uniformly
Component Sharing	None	Limited	None	Ad hoc	Full

Table 3.1: Comparison with other pipeline design methodologies.

3.3.1 Systolic approach

Strictly speaking, the systolic approach is not a synthesis methodology, but a algorithm-transformation methodology. Yet its design process, and the architectures generated by it, can be compared with those of the present approach.

The systolic methodology is used to design spatially and temporally parallel special-purpose architectures employing the pipelining concept. The process of systolization involves *introducing* buffer registers such that, if two registers are connected via a combinational data-path, its delay is less than or equal to the clock cycle. This is required to ensure correct operation under the constraint that every register is clocked every cycle of a single global clock.

In the following, the data-paths, between the buffers introduced by a methodology, are called *primitive operations*, or simply operations. Thus, the above is equivalent to saying that the primitive operations in a systolic architecture all have the same delay (or unit delay), and those allowed by the present methodology may have longer delays.

There are three fundamental differences between the systolic approach and the new approach presented here: the synthesis process, the treatment of individual operations, and input latency. The new approach differs from the systolic in that, whereas the input to the systolization process is a single sequential circuit which is transformed into a "systolic" circuit via retiming [36], the transformation being based solely on the delay criterion just stated, the present approach partitions the combinational data-paths on the basis of functionality and does not require the delay of the path to be less than or equal to the basic clock cycle.

Thus, the systolic approach does not treat the operations as being

atomic and registers may be introduced within these operations. In the present approach, on the other hand, the operations specified in the algorithm are treated as being atomic and retained as such in the architecture.

Another characteristic of systolic architectures is that the input latency is also equal to the unit delay defined by the global clock. Clearly, in this respect, systolic architectures are seen to be a special case of architectures producible by the new approach.

There are other major architectural differences between the two design styles. Many arise from the above differences.

In systolic architectures hardware components are not shared among primitive operations, either computational or communication, whereas in the new approach operations share components whenever possible. As pointed earlier, this capability of component sharing is advantageous in presence of multi-rate operations. This concept is demonstrated via the following example of a decimation filter with two FIR blocks flanking the decimation block. In presence of a rate change operation of decimation of order 2, the subsequent computation is *folded* and mapped onto an array half in size.

Consider the two stage filter shown in Figure 3.11. The filter is comprised of two FIR filter blocks of orders K1 and K2 with a decimation block in the middle. Assume, for the purpose of illustration, that the decimation factor is 2 and that K2 is an even number greater than or equal to 4.

An FIR filter of order K is described by the following recurrence relation:

$$y_n = a_0 x_n + a_1 x_{n-1} + a_2 x_{n-2} + \cdots + a_{K-1} x_{n-K+1} \tag{3.1}$$

$$= \sum_{i=0}^{K-1} a_i x_{n-i} \cdot \tag{3.2}$$

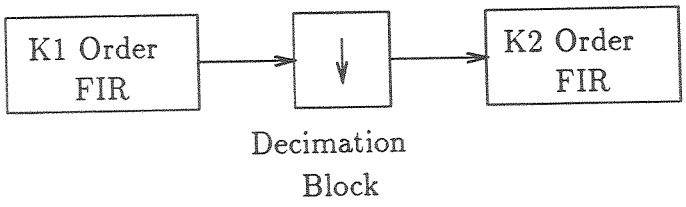


Figure 3.11: A two stage FIR

One possible systolic implementation of an FIR filter presented in [4] uses a cell structure shown in Figure 3.12. Using this cell structure the systolic implementation uses a single chain of K cells to implement a K-order filter. Thus using a single clock, K1+K2 cells are needed to implement the two stage filter.

If the existence of the rate change operation between the two FIR blocks is taken into account, however, a filter that is lower in cost can be obtained.

In the alternative design, three clocks C1, C2 and C3 are used. Clock C1 has twice the frequency as C2 and C3, and C3 has the phase opposite to C2. The K1 order FIR is implemented, as before, using the C1 clock. The K2 order FIR can be implemented as shown in Figure 3.13. This design can be looked upon as being *piecewise systolic* or *folded systolic.*

In this design there are only half as many systolic cells as in the single clock design, but they continue to be clocked by C1. However, during alternate cycles, the cells compute different halves of two different sums which are eventually combined to produce the correct output. To match the speeds of the x and y inputs of each cell so that the cells collect appropriate terms, an extra register has to be added per cell in the y path.

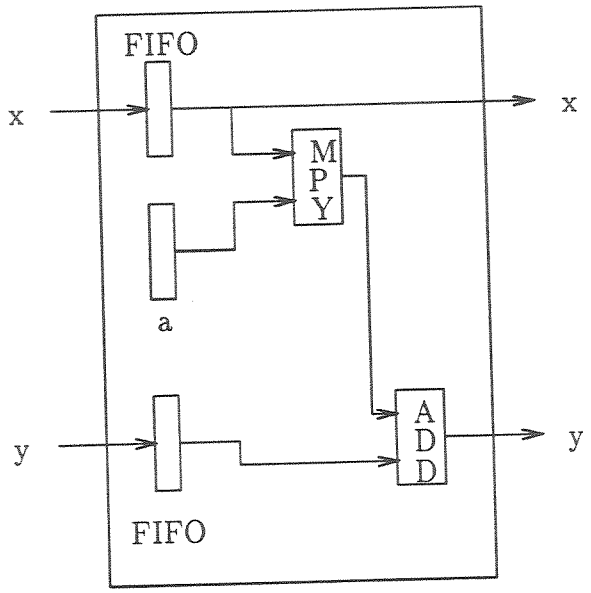


Figure 3.12: A systolic cell for FIR computation

To compute the cost differential between the two approaches assume the following:

A Cost of an adder.

M Cost of a multiplier.

R Cost of a register.

m Cost of a multiplexer.

c Number of registers per systolic cell ($\mathbf{c} \geq 3$).

The cost differential between the two designs is given by,

$$\Delta Cost = \frac{K2}{2}(A + M + cR) - \frac{K2}{2}(m + 2R) - A - 2R \quad (3.3)$$

$$= \frac{K2}{2}(A + M + (c-2)R - m) - A - 2R \quad (3.4)$$

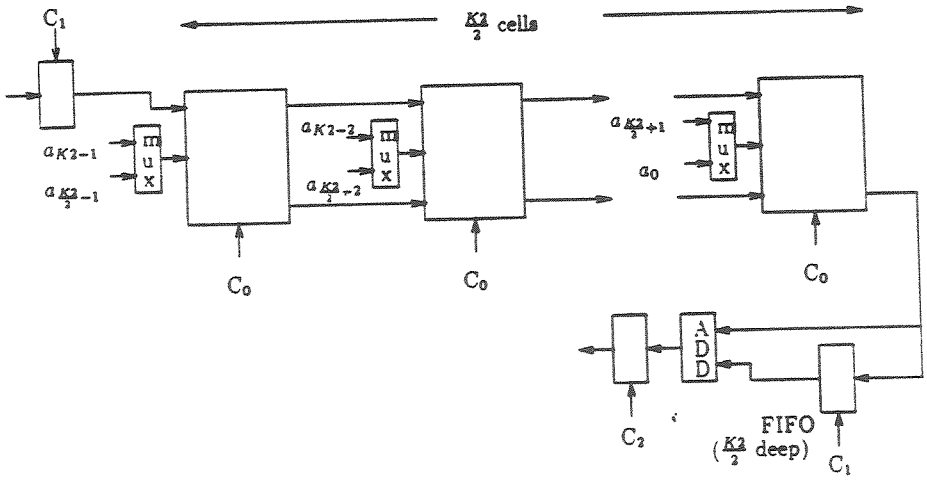


Figure 3.13: A multi-clock implementation of an FIR

The $\Delta Cost$ is positive if A + M + (c–2)R > m + $\frac{2}{K2} \cdot (A+2R)$, that is, if (1 - $\frac{2}{K2}$)A + M + $(c-2-\frac{4}{K2})$R > m, which is generally true in practice since M $\geq$ m.

In the above implementation, the same clock rate (clock C1) was used to drive the array and the computation was multiplexed over a half-sized array. Instead, the same sized array but a clock half as fast could be used, or in other words, clock C2, could be to drive the array, allowing the use of slower but less expensive components.

3.3.2 Sehwa

There are several differences between the Sehwa methodology and the new one.

Sehwa starts with an acyclic directed graph denoting the computation, and partitions it into multiple subgraphs using cut-lines (imaginary lines that cut across directed cutsets of the original graph), each subgraph corresponding to a time-step in the pipeline execution. The edges of these cutsets are the sites of latches which hold results of the stage computation at the end of the clock cycle. The critical path of a stage determines the clock-cycle of the pipeline.

One of the major differences between Sehwa and the current formulation is that Sehwa computes the clock's period, and this determines the partitioning of the computation graph: the length of a critical path in every subgraph is no longer than the clock cycle. In the case of the new approach, there exists a high frequency clock in terms of whose period all delays are counted, and no attempt is made to partition the graph.

Implicit in the objective of the Sehwa methodology is the absence of any clock other than the one used to latch inter-stage results. Thus, in

Sehwa, the basic building block is a *combinational* hardware component, and its methodology disallows internally sequential components. This is a serious restriction in view of the stated goals of this dissertation. Clearly, multi-rate functions and algorithms and use of shared port components are also outside the application domain of the Sehwa methodology. Furthermore, since Sehwa requires that the length of a critical path in a subgraph be no longer than the clock cycle, Sehwa implicitly disallows multi-cycle combinational components as well.

The fact that Sehwa puts latches only at the boundaries of the subgraphs and not on the edges within it makes the Sehwa methodology superior to the new methodology in terms of number of registers used. But insertion of input registers for each hardware component allows sharing within a Sehwa clock-cycle and is therefore cost-advantageous.

For example, consider a subgraph which has a path with a multiplication and an addition and a separate path with only an addition operation. (See Figure 3.14.) It is clear that Sehwa will require the clock-cycle to be of a minimum period equal to the sum of the delays of a multiplier and an adder, and will disallow the sharing of a single adder for the two additions. The new methodology will, however, allow the such a sharing of an adder.

To illustrate the above with an example, consider the sixteen point FIR filter in Figure 3.15 presented in [44]. Figure 3.16 displays a DDG for the same filter with the explicit inclusion of communication operations. Assuming that the execution times for adders and multipliers to be 40 ns and 80 ns respectively, as assumed in [44], communication operations to have a delay of 20 ns each, and input latency of 300 ns, again, as assumed in [44], with a stage time of 100 ns, Sehwa requires 5 adders and 3 multipliers to produce a

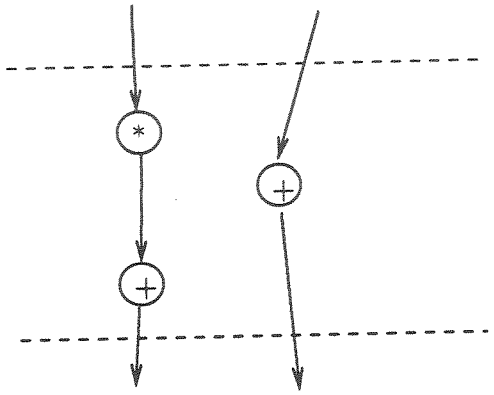


Figure 3.14: A critical path and a non-critical path within the same time-step.

pipeline with graph latency of 600 ns. Figure 3.17 presents a feasible schedule, producible via the new approach presented herein, with the same graph latency of 600 ns (actually 580 ns) and yet which requires 2 fewer adders. In that figure the vertical lines denote the clock ticks of a global high frequency clock with a clock cycle of 20 ns. The transfers of data over busses are also assumed to take 20 ns each, which is the same as the latch time for Sehwa.

Furthermore, Figure 3.18 shows a feasible schedule which further reduces the number of adders required by 1 at the expense of increase in graph latency to 820 ns and with a schedule latency of 600 ns. Since, for pipelined architectures, the graph latency is not of primary importance, this may be a more attractive solution.

The above difference is further amplified by the following exercise. For the sake of the ensuing comparison, let the graph latency be 900 ns. Clearly, the schedule need not change for the new approach. For Sehwa, assuming, as before, a stage time of 100 ns, this will translate to a graph latency of 9 time-steps. This implies that Sehwa will introduce 8 cut-lines in the graph to partition it. Given that the input latency is of 300 ns or 3 time steps, each time step will correspond to the overlapped execution of 3 partitions.

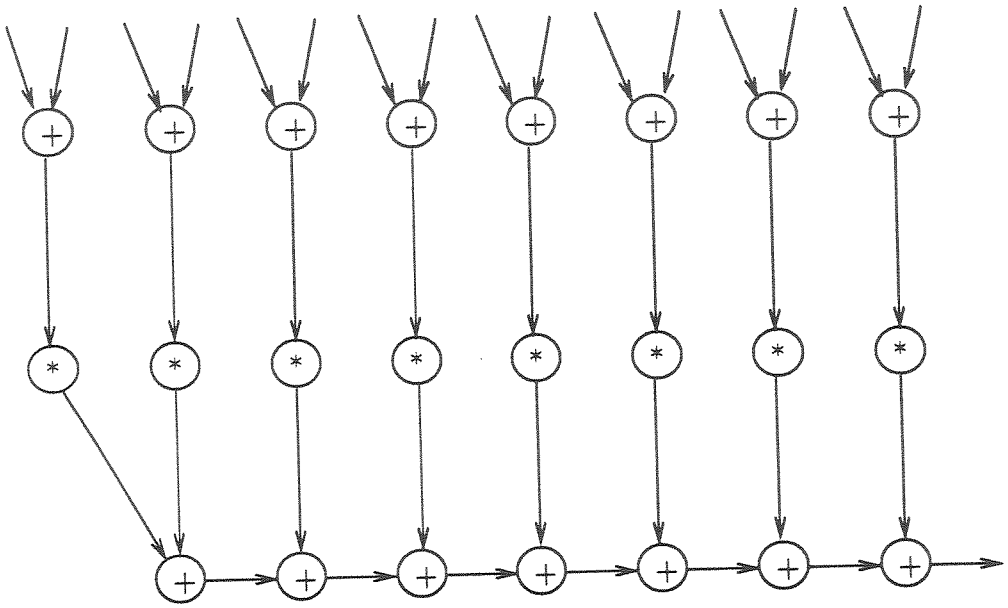


Figure 3.15: Flow graph of the 16 point FIR filter.

Since Sehwa does not share components within a time step, the minimum number of adders required by Sehwa is the maximum number of additions appearing in any of the time-steps. Thus to reduce the number of adders necessary in a Sehwa architecture, attempt must be made to minimize the maximum number of additions per time-step. The best one can do, (although, it cannot be guaranteed that it is always possible to partition the graph to achieve this,) is to distribute the additions uniformly such that each partition has at most one less than the maximum. The maximum in this case is given by $\lceil \frac{number\ of\ additions}{3} \rceil$ $= \lceil \frac{15}{3} \rceil = 5$, which is still the same as before. Thus Sehwa does not reduce the number of adders required even when the graph latency is allowed to increase by 50% of its minimum. In fact, no matter how large a graph latency is allowed, as long as the stage time remains 100 ns and the input latency remains 300 ns, the same requirements will result; the Sehwa methodology is insensitive to changes in graph latency.

Table 3.2 summarizes the above comparisons.

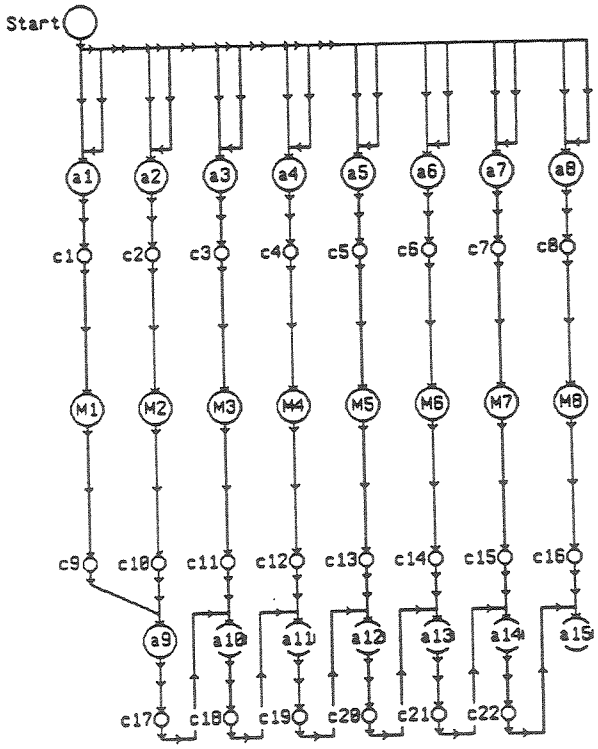


Figure 3.16: The DDG representing the FIR filter.

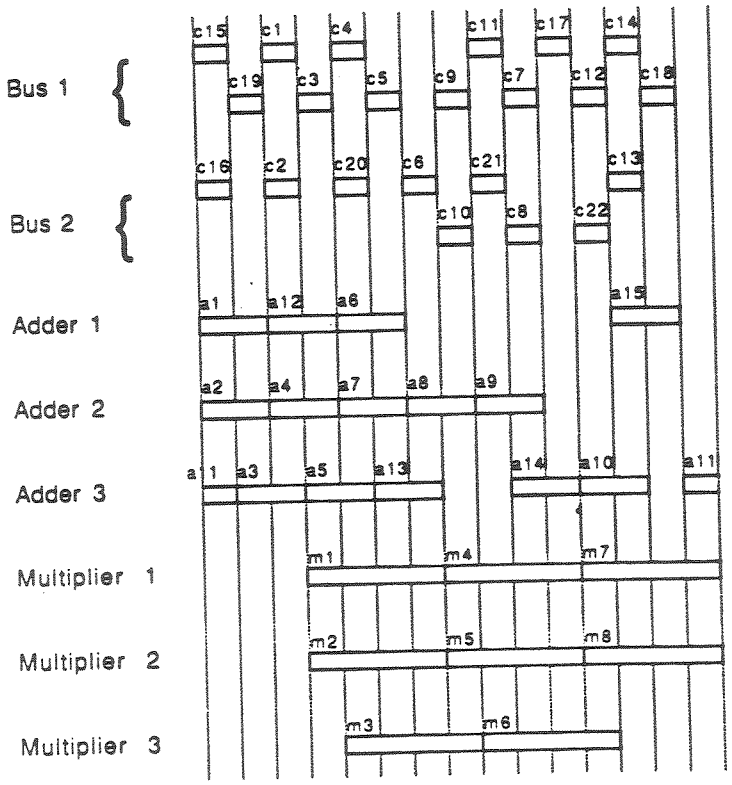


Figure 3.17: The schedule with graph latency of 600 ns obtainable by the new approach.

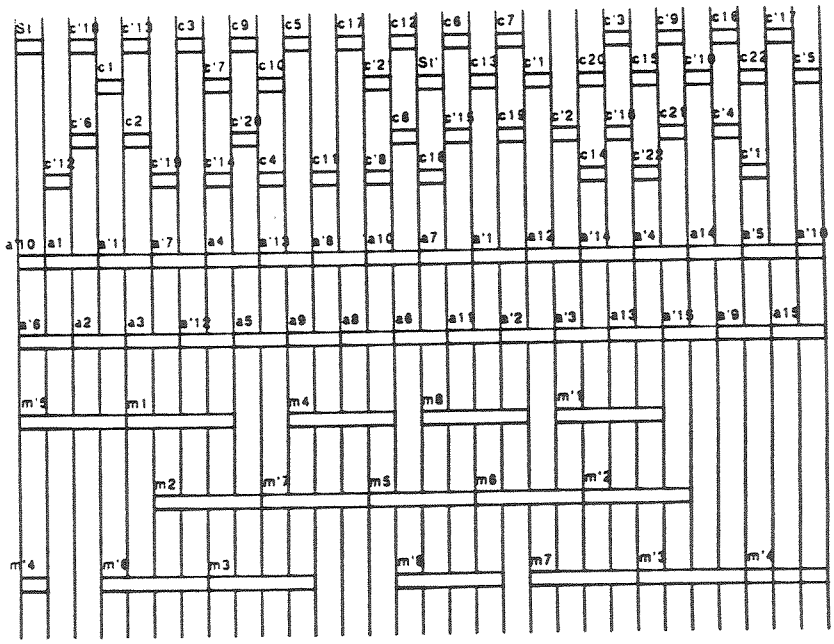


Figure 3.18: The schedule with graph latency of 820 ns obtainable by the new approach.

Parameter	Sehwa		Present	
Input Latency	300 ns	300 ns	300 ns	300 ns
Graph Latency	600 ns	900 ns	600 ns	900 ns
# of Multipliers	3	3	3	3
# of Adders	5	5	3	2
# of Busses	Muxes	Muxes	2	2

Table 3.2: Comparison between Sehwa and the present approach.

Note that the timing results obtained by the present approach are actually better than asserted in the foregoing, as they are biased adversely by the time taken for the communication operations. In Sehwa, on the other hand, the time taken for communication between components within a control step is not explicitly accounted for and so reflect as shorter execution times.

Of note is also another advantage of using the present approach: the explicit accounting of communication operations. By following the premise of inserting a buffer in front of every hardware component, it is possible to give computation and communication operations a *uniform* treatment. The benefit gained is that it is possible to analyze *a priori* the communication requirements of the architecture and provide a well defined semantic structure to the communication architecture using busses.

The advantage of this can be seen in terms of the number of busses in the architecture. Although [44] does not report on the number of busses in the architecture, [21] reports the number of busses for the same algorithm to be 6.

It is important to point out, however, that communication resources can be directly traded against buffers. Thus one must expect more buffer registers in a design obtained using the new approach. This issue is discussed further in Chapter 6.

3.3.3 Cathedral systems

Cathedral I, II, and III are three, high level, automated synthesis systems directed toward three different design styles. Cathedral I and III are better suited for signal processing applications and design of algorithm-specific pipelines, whereas Cathedral II is more general purpose. Cathedral I is oriented toward bit-serial architectures while Cathedral III handles bit-slice architectures. Following paragraphs will compare and contrast each of these with the new approach on the bases of models of computation and architectural characteristics.

None of Cathedral I, II and III has formally defined execution models for hardware components which the respective methodologies use. However, all three use predefined library components in terms of which all operations are implemented. All three assume combinational single rate components, and so multi-cycle and multi-rate components are outside the scope of the three methodologies. Cathedral II and III, however, do handle multi-rate algorithms involving decimation and interpolation, but they do so in an ad hoc and semi-automatic manner involving designer participation, the designer providing strategies for sharing components, as necessitated by the presence of multi-rate operations. Furthermore, the multi-rate operations are restricted only to those of decimation and interpolation, and serialization and parallelization; the general methodology to handle multi-rate components is not part of the Cathedral

systems.

Unlike in the new approach, the three Cathedral systems treat communication operations separately from computation. Cathedral I and III assume dedicated wire paths between computation elements with multiplexers used to switch between data sources. Cathedral II, on the other hand, attempts to combine communication operations into bus structures after the scheduling process is complete. Cathedral II may, therefore, be expected to require more busses than the minimum required by the new approach.

Of the three Cathedral systems, only Cathedral II incorporates a general component sharing scheme. Cathedral I and III inherently do not support sharing of hardware.

Similar to the new approach, the primitive operations in Cathedral I and II are atomic, but Cathedral III uses the concept of retiming and allows insertion of registers within a component circuit to ensure that a data path between any two registers is no longer than the clock cycle.

Cathedral II uses a textual, applicative language called SILAGE as the input medium to express algorithms and translates it to the intermediate graphical form of Value Trace [54].

Chapter 4

Analyses of DDGs

As a first step in the automated synthesis process, a multi-rate algorithm specified as a DDG must be examined for *correctness*. The correctness here implies the DDG's amenability to obtaining a feasible architecture. Next, the DDG is analyzed to extract design information such as number of components, their shareability, the minimum possible input latency, and the schedule latency to be later used for scheduling and assignment.

In this chapter, analytic techniques to assist in these steps are presented. These include the *token rate equations*, the *token count equations*, and the *equivalence transformations*. A *rate consistency* condition, presented in Section 4.1, must be satisfied by a DDG for a feasible architecture and execution schedule to exist. A directed loop in a DDG may deadlock if sufficient number of initial tokens are not present on its edges. The condition for the freedom from deadlock in a loop is derived in Section 4.6. An extended graph called the *execution graph* of a DDG is used to establish the lower bound on the input latency and to schedule the computation. The construction of this structure is covered in Section 4.4. The technique of *subgraph-collapse* is useful for defining a multi-rate component that executes a multi-rate sub-algorithm. This is described in Section 4.3.

4.1 Token Rate Equations

In this section DDGs are analyzed to obtain architectural design information such as number of hardware elements required in the architecture. To do this, the concept of *token rate equations* is introduced.

Consider nodes i and j connected by a dependency as shown in Figure 4.1. Since there is no loss of data during transfer between the two nodes, the average rate of production of tokens (i.e., data items) by node i must equal the rate of absorption of tokens by node j. Assuming that p_i and p_j are average numbers of elements of types i and j respectively, and D_i and D_j their respective execution delays, the following relation must hold:

$$p_i \cdot \frac{O_e}{D_i} = p_j \cdot \frac{I_e}{D_j}$$

In asserting the above equation, a notion of steady-state balance of rates of in-flow and out-flow of tokens has been used. This notion is similar to the concept of *local-balance* used by Chandy in [8] for analyzing steady-state lengths of queues in a queueing network. The analogy is clear if the edge is looked upon as a FIFO queue.

Each dependency in a DDG gives rise to a *token rate equation* (TRE) of the form shown above. (Note that the equation remains unchanged even if the direction of the dependency is reversed.) Thus the average behavior of a DDG can be expressed in terms of |E| TREs, where E is the set of edges of the DDG. For the pipeline to function properly, the set of TREs must be simultaneously satisfied by the vector $< p_i >$.

In general, a solution of the TREs, $< p_i >$, is a vector of rational numbers. However, the actual number of elements of type *i* in an architecture

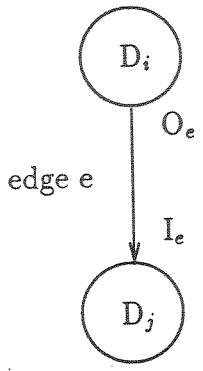


Figure 4.1: A Dependency

must be an integer. One may choose $n_i = \lceil p_i \rceil$, or any larger integer. Given $n_i \geq p_i$, $u_i = \frac{p_i}{n_i}$ can be interpreted as the average utilization of n_i elements of type i.

The above equation can be transformed to read

$$\frac{O_e}{D_i} \cdot p_i - \frac{I_e}{D_j} \cdot p_j = 0$$

In the matrix form, the above equation can be cast as:

$$\begin{bmatrix} \frac{O_e}{D_i} & -\frac{I_e}{D_j} \end{bmatrix} \begin{bmatrix} p_i \\ p_j \end{bmatrix} = 0$$

or equivalently,

$$\begin{bmatrix} O_e & -I_e \end{bmatrix} \begin{bmatrix} \frac{1}{D_i} & 0 \\ 0 & \frac{1}{D_j} \end{bmatrix} \begin{bmatrix} p_i \\ p_j \end{bmatrix} = 0$$

Thus, the system of TREs is of the form $\mathbf{R} \cdot \mathbf{p} = \mathbf{0}$, in which matrix $\mathbf{R}$ can be factored into two matrices $\mathbf{T}$ and $\mathbf{D^{-1}}$ such that, matrix $\mathbf{T}$ involves only token markings O_e and I_e, the subscripts refering to the corresponding edge, and $\mathbf{D^{-1}}$ is the inverse of the diagonal delay matrix $\mathbf{D}$ of the execution

delays of operations. The T matrix is of size |E| × |V|, there being a row per edge of the DDG.

Clearly, since a DDG is a graph with |V| nodes, the rank of its **R** matrix must be ≤ |V|, the number of variables. However, for the set of TREs to have a non-trivial solution (the trivial solution is $\mathbf{p} = \mathbf{0}$), the rank of **R** must be less than |V| [55]. It will be shown that the condition stated in Definition 14, if satisfied by a DDG, makes the rank of its **R** matrix |V|–1 and therefore guaranties a non-trivial solution.

But before the above can be shown, it is necessary to establish that a tree-form DDG always has a non-trivial solution.

Definition 13 *An undirected graph obtained by ignoring the directions on edges of a DDG is called an Undirected DDG (UDDG).*

According to the definition, the dependencies of a DDG become edges in the UDDG. The TREs for the edges in a UDDG are those for the corresponding dependencies in the DDG. Thus the set of TREs describing the UDDG is the same as that of the original DDG and the solution to one is the solution to the other.

Notation: Path-product-ratio

Consider a path s from some node p to some other node q within the UDDG of a given DDG. (See Figure 4.2.) A *path-product-ratio* ρ_s is formed by traversing the path s. Starting at p and initially with $\rho_s = 1$, upon exiting a node, the denominator of ρ_s is multiplied by the token marking on the edge leaving the node, and upon entering a node, the numerator of ρ_s is multiplied by the token marking on the edge entering that node. For convenience, the factors

Figure 4.2: A Path s From p To q

multiplied into the numerator are denoted as as O_j and those multiplied into the denominator are denoted as I_j. Thus,

$$\rho_s = \frac{\prod_s O_j}{\prod_s I_j}.$$

If the path contains no edges, i.e., if the start and end nodes of the path are the same and it does not traverse any edge, then $\rho_s = 1$ by definition.

Consider the set of TREs corresponding to the dependencies along the path s. Assume that a path s from node p to q is formed by edges 1,2, ... ,i as shown in Figure 4.2. The intermediate nodes are assigned the same labels as their respective input edges.

If the set $<p_i>$ is chosen such that the set of TREs is simultaneously satisfied, then the following must hold true:

$$\begin{aligned}
p_p \cdot \frac{O_1}{D_p} &= p_1 \cdot \frac{I_1}{D_1} \\
p_1 \cdot \frac{O_2}{D_1} &= p_2 \cdot \frac{I_2}{D_2} \\
&\vdots \\
p_i \cdot \frac{O_i}{D_i} &= p_q \cdot \frac{I_i}{D_q}
\end{aligned}$$

Back substituting, the following relationship is obtained:

$$p_q = p_p \cdot \frac{D_q}{D_p} \cdot \frac{\prod_{j=1}^{i} O_j}{\prod_{j=1}^{i} I_j}.$$

When p_p and p_q satisfy the above relationship, they are said to satisfy the set of TREs along the path s.

Notice that,

$$\frac{\prod_{1}^{p} O_j}{\prod_{q}^{i} I_j} = \rho_s .$$

Thus, the above relation simplifies to $p_q = p_p \cdot \frac{D_q}{D_p} \cdot \rho_s$. This is put in the form of a lemma:

Lemma 2 *If there is a connected path s between nodes p and q of a DDG, and if p_p and p_q satisfy the set of TREs along path p in its UDDG, then $p_q = p_p \cdot \frac{D_q}{D_p} \cdot \rho_s$.*

Theorem 1 *: A DDG in the form of a tree has a non-trivial TRE solution.*

Proof: Since a TRE is insensitive to the direction of the dependency, consider the UDDG of the DDG. A solution to the TREs of the UDDG will also be a solution to the TREs of the DDG.

Assume that the nodes in the UDDG are relabeled uniquely, 1 through m, with the root being assigned the label '1'. (The choice of the root node is quite arbitrary; any node may be made the root. See Figure 4.3). For each edge e, the token marking associated with the end closer to the root is denoted

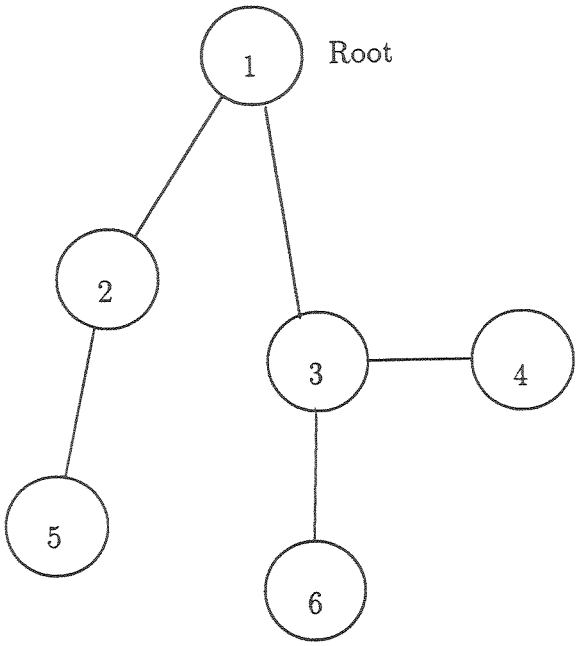


Figure 4.3: A Tree-Form UDDG

by O_e, and that associated with its other end is denoted by I_e. Also, of the two nodes an edge connects, the one that is nearer the root is considered the parent of the other.

Since the UDDG is a tree, there is a unique path from the root to every node in the graph. The following values can be chosen as solutions to the set of TREs for the UDDG. For a node i ($i \neq 1$), assign

$$p_i = \frac{D_i}{D_1} \cdot p_1 \cdot \rho_i \ ,$$

where ρ_i is the product as defined earlier taken along the (unique) path from the root (node 1) to node i. p_1 can be chosen to be any rational number other than zero to obtain a non-trivial solution. □

To see that these assignments do satisfy the set of TREs, consider a pair of nodes p and q connected by an edge e such that p is the parent of q. Since the UDDG has the form of a tree, the path from the root to node q is

unique and node p and edge e are in it. Therefore, $\rho_q = \rho_p \cdot \frac{O_e}{I_e}$. It must be verified that the assignments to p_p and p_q satisfy the TRE for the edge e.

$$\begin{aligned} p_q &= \frac{D_q}{D_1} \cdot p_1 \rho_q \\ &= \frac{D_q}{D_1} \cdot p_1 \cdot \rho_p \cdot \frac{O_e}{I_e} \\ &= \frac{D_q}{D_1} \cdot \frac{D_p}{D_p} \cdot p_1 \cdot \rho_p \cdot \frac{O_e}{I_e} \\ &= \frac{D_q}{D_p} \cdot \frac{D_p}{D_1} \cdot p_1 \cdot \rho_p \cdot \frac{O_e}{I_e} \\ &= \frac{D_q}{D_p} \cdot p_p \cdot \frac{O_e}{I_e} \end{aligned}$$

That is,

$$p_q \cdot \frac{I_e}{D_q} = p_p \cdot \frac{O_e}{D_p}$$

Notice that the solution $< p_i >$ is uniquely determined by the choice of p_1. That is, the solution has exactly one free variable, and not more than one variable can be independently chosen. Put in matrix terminology, the corresponding **R** matrix has the rank of one less than the number of variables. Therefore, the following lemma can be stated:

Lemma 3 *The* **R** *matrix of a tree-form DDG with V nodes has the rank of* $|V|-1$.

Since any DDG contains its spanning tree with all its nodes, it can be concluded that,

Lemma 4 *The* **R** *matrix of a DDG with* $|V|$ *nodes has the rank* $\geq |V|-1$.

Consistent DDGs

Definition 14 : *A DDG is said to be a Consistent DDG (CDDG), if and only if, it satisfies the rate consistency condition: for every cycle* l *in its UDDG,* $\rho_l = 1$.

Whether a DDG satisfies the rate consistency condition for a DDG can verified by using the following algorithm.

Algorithm 1 : Algorithm to check for a CDDG.

- *Obtain the UDDG from the given DDG.*
- *Use a Depth First Search algorithm*[1].
- *Assign* $\rho_{root} = 1$.
- *On traversing an edge (u, v) from node* u *to node* v, *compute* $\rho_v = \rho_u \cdot \frac{O_u}{I_v}$, *where* O_u *and* I_v *are markings on the edge (u, v) at* u *and* v *respectively.*
- *For each back edge (u, v), verify that* $\rho_v = 1$.

Consider a dependency loop similar to the one shown in Figure 4.4, which may be part of a CDDG. It consists of a path a from node p to node q and an edge b from q to p. If edge b is removed from the loop, only path a is left and the sub-graph becomes acyclic. The following can be demonstrated:

[1] As described in [1]

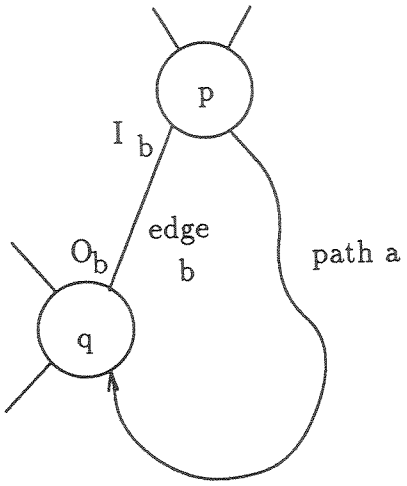


Figure 4.4: A Cycle Within A CDDG

Lemma 5 : *If a set of p_is satisfies the set of TREs for path formed by all but one edge of a directed loop of a CDDG, they also satisfy the TRE for the remaining edge of the loop.*

Proof: The proof is developed with reference to Figure 4.4, in which the path is labeled by the letter "a", and the edge by "b".

Let $p_p, \ldots, p_q$, be the set of values satisfying the TREs along path a. Then,

$$p_q = p_p \cdot \frac{D_q}{D_p} \cdot \rho_a.$$

By condition 1 in Definition 14 above,

$$\rho_l = \rho_a \cdot \frac{O_b}{I_b} = 1.$$

Thus,

$$p_q = p_p \cdot \frac{D_q}{D_p} \cdot \frac{I_b}{O_b}.$$

□

This lemma implies that in order to obtain a solution for the set of TREs representing dependencies that form a cycle in a UCDDG, all but one of the dependencies need to be considered. The solution obtained will satisfy the TRE for the remaining dependency. In algebraic terms, it means that the TRE for the last dependency is linearly dependent on the equations for the other dependencies forming the cycle. Or, in other words, if there are m dependencies which form a cycle, then at most m–1 of the TREs are linearly independent. Put differently, if the set of TREs for a cycle are expressed in a matrix form, since the rank of a matrix is the number of independent rows, the corresponding **R** matrix has the rank of m–1. This can be stated in the form of the following corollary.

Corollary 1 *The* **R** *matrix corresponding to a set of TREs for the edges of an m-node loop of a UCDDG has the rank of m–1; each TRE is linearly dependent on the other m–1 TREs.*

Theorem 2 *The* **R** *matrix of any CDDG with* $|V|$ *nodes has the rank of* $|V|-1$.

Proof: It has already been shown that the rank of the **R** matrix is greater than or equal to |V|–1 and that it cannot be greater than |V|.

Consider the UCDDG of the CDDG being analyzed. Let the edges of the UCDDG be divided into two disjoint sets E_t and E_l, such that E_t is the set of edges in a spanning tree of the UCDDG and E_l is the set of remaining edges of the UCDDG. Thus $E_t \cup E_l$ is the set of all edges in the UCDDG. The rank of the **R** matrix corresponding to the edges in E_t is |V|–1 (by Lemma 3).

By the definition of a spanning tree, every edge e in E_l creates a simple cycle when added to the set E_t. But by Lemma 1, the edge e does not add to

the rank of the **R** matrix. This is true for every edge in E_l. Thus the rank of the **R** matrix of the UCDDG, and therefore of CDDG, is the rank of the **R** matrix corresponding to the edges in E_t, which is |V|–1. □

The consequence of the above theorem is that,

Theorem 3 : *The set of TREs for a CDDG has a non-trivial solution.*

Before going on to the next section, the ranks of the two matrices T and D^{-1}, into which matrix R can be factored, are established: The D^{-1} matrix is a diagonal matrix of size |V|, and therefore has a rank of |V|. However, as was proved earlier that the R matrix for a CDDG has a rank of |V|–1. It must therefore be concluded,

Theorem 4 *The rank of the topology matrix T of a CDDG with |V| nodes is |V|–1.*

The last result could have been obtained directly, had the delays of the nodes been ignored. In the next section the number of invocations of each node to make up a complete single invocation of the DDG is obtained using this result. (Note that Lee and Messerschmidt in their paper ([34]) require the same result to be true for a PASS schedule to exist.)

Uniqueness of the rate solution

The solutions to the set of equations $\mathbf{R} \cdot \mathbf{p} = \mathbf{0}$ form the null-space of matrix **R** and are called its null vectors. This null-space has the same dimension as the number of free variables. Since the rank of **R** of a CDDG is |V|–1, the non-trivial solution has one free parameter — call it θ — to which any rational

value may be assigned. Thus given a solution vector $\mathbf{p}$, $\theta \cdot \mathbf{p}$ is also a solution, where θ is a rational scalar. Typically, however, the free variable will be bound by the input latency — which is often determined externally by the application, and the number of input components — which will be assumed to be fixed at 1. The result is: given the input latency, the rate solution is unique for a given DDG.

4.2 Token Count Equations

The execution of the computation specified by a DDG is realized via the invocations of its nodes. Since a statically defined repetitive execution schedule for the DDG is desired, the numbers of times the nodes of the DDG must be invoked, per single invocation of the DDG computation, have to be established. To this end, the DDG will be analyzed in terms of *token count equations* (TCEs) described in this section.

Consider nodes i and j connected by a dependency as shown in Figure 4.1. The number of times node j is invoked is determined by the number of tokens available on the edge e connecting the two nodes. Each invocation of node i produces O_e tokens, whereas each invocation of node j absorbs I_e tokens. There may be some initial tokens available on the edge. However, since each repetition of the schedule contains the same number of invocations of each node, the number of tokens produced by invocations of node i must equal the number of tokens absorbed by those of node j, so that at the end of every invocation of the schedule, the initial tokens are restored to their original number. Assuming that N_i and N_j are numbers of invocations of nodes i and j respectively per invocation of the schedule, the choice of N_i and N_j must satisfy the following token count equation (TCE):

$$N_i \cdot O_e = N_j \cdot I_e$$

which may also be written as,

$$N_i \cdot O_e - N_j \cdot I_e = 0$$

This can be re-written in the matrix form as,

$$\begin{bmatrix} O_e & -I_e \end{bmatrix} \begin{bmatrix} N_i \\ N_j \end{bmatrix} = 0$$

Every dependency in a DDG gives rise to a *token count equation* (TCE) of the form shown above. Notice that the above TCE, like the corresponding TRE, is independent of the direction of the dependency. Thus the numbers of invocations of the nodes of the DDG can be expressed in terms of |E| TCEs, where E is the set of edges of the DDG. For a repetitive schedule, the set of TCEs must be simultaneously satisfied by the set $< N_i >$.

The system of TCEs is of the form $\mathbf{T} \cdot \mathbf{N} = \mathbf{0}$, where T is the topology matrix encountered in the last section and N is a vector of size |V|. From Theorem 4 the rank of T is known to be |V|–1, and thus,

Theorem 5 : *The set of TCEs for a CDDG has a non-trivial solution.*

In fact, all the results in the last section have counterparts for TCEs. Two of them are stated here for completeness and future use.

Lemma 6 *If there is a connected path s between nodes p and q of a DDG, and if N_p and N_q satisfy the set of TCEs along path p in its UDDG, then $N_q = N_p \cdot \rho_s$.*

Lemma 7 : *If for a loop in a CDDG, a set of N_is satisfies the set of TCEs for path a, they also satisfy the TCE for edge b.*

The solution vectors **N** define the null-space of the topology matrix T. As before, there is one free parameter, say ϕ, which may be chosen at will. Thus if $<N_i>$ is a solution, so is $\phi \cdot <N_i>$. Unlike in the case of TREs, however, no external constraints fix the value of ϕ, and any solution may be chosen. In the next chapter, this freedom will be used to choose the right solution to obtain an architecture with minimum resource requirement for an acyclic CDDG.

(Throughout this work only CDDGs, which represent meaningful computations, will be of concern. Therefore, henceforth, the term DDG will imply that it satisfies the consistency condition.)

Minimum token count solutions of DDGs

All the entries in the topology matrix **T** are integers, and so in general the solution to TCEs can be integers or rationals. Because the solution to TCEs represents numbers of invocations of the nodes, integer solutions are sought. Of course, as mentioned above, given an integer solution $<N_i>$, and an integer ϕ, another integer solution $\phi \cdot <N_i>$ can be obtained. Because every node in the DDG is expected to fire at least once, no element in a solution vector $<N_i>$ can be zero.

Of interest is the *minimum integer solution* to the set of TCEs. Once this is obtained, every other integer solution can be found by choosing an appropriate integer multiplier ϕ. Also, given an integer solution $<N_i>$, it is easy to obtain the minimum integer solution by dividing the *greatest common*

divisor of the individual N_i's into the original solution. The following definition will be used for the *minimum token count solution* of a DDG:

Definition 15 *The minimum token count solution of a DDG is the minimum integer solution to its set of TCEs.*

Clearly,

Lemma 8 *The minimum token count solution for a DDG is unique.*

Proof: If $<N_i^1>$ and $<N_i^2>$ are two distinct minimum token count solutions of the DDG. Since they are solutions to the same set of TCEs, they must be related by an integer multiple ϕ. Let $<N_i^2> = \phi \cdot <N_i^1>$, $\phi \neq 1$. But then by definition, $<N_i^2>$ is not a minimum solution. □

Given the minimum token count solution of a DDG, another token count solution is obtained by multiplying it by an appropriate integer value for ϕ. Since the minimum token count solution is unique, it is obvious that,

Lemma 9 *Any given token count solution implies, and is implied by, a unique value of ϕ.*

4.3 Subgraph collapse and hardware composition

In graph-theoretic terminology, any graph, created by removing a (possibly empty) subset of nodes, and all edges incident upon the nodes of this subset or a (possibly empty) subset of edges of the original graph, is called a sub-graph. In this section, only certain restricted sub-cases of this definition of a sub-graph will be used.

Definition 16 *A sub-DDG is a sub-graph created inside a simple closed curve which intersects the original DDG only in edges and which includes at least one node. The set of edges intersecting the curve is called the cutset of the sub-DDG.*

The nodes of the subgraph upon which the edges entering the enclosed region are incident are the input nodes of the sub-DDG. The nodes of the subgraph upon which the edges leaving the enclosed region are incident are the output nodes of the sub-DDG. Notice that a node of the sub-DDG may be both, an input and an output node. If a sub-DDG is acyclic, then some of the input nodes will have no other input dependencies, and are referred to as the *primary input nodes*; and some of the output nodes will have no other output dependencies, and are referred to as the *primary output nodes*. If, on the other hand, the sub-DDG is cyclic, there might be no primary input and/or output nodes.

Before proceeding with the concept of sub-DDG collapse, note the following rather obvious lemma in passing:

Lemma 10 *: A sub-DDG of a CDDG satisfies the consistency condition.*

Proof: If the sub-DDG is not consistent, it would mean that one of the loops of the corresponding undirected sub-graph violates Definition 14. But this loop is part of the original DDG as well, which cannot be a CDDG either, as a result. This is in contradiction to the original premise. Hence this lemma. □

4.3.1 Sub-DDG Collapse

In this section, sub-DDGs are analyzed and a mechanism is formulated, by which a sub-DDG may be implemented on a single composite hard-

ware module. There are three main motivations for having this capability. One is to use a single programmable device for implementing part of an algorithm and using that device in the overall architecture. Another is the motivation provided by the standard cell approach to develop library cells for common computations. Yet another is the ability to do hierarchical design, in which larger sub-DDGs are implemented using circuits of component sub-DDGs.

The following definition for a subgraph of a directed graph will be used.

Definition 17 *A connected subgraph is a subgraph for which none of its subgraphs has an empty cutset*[2].

Consider the connected sub-DDG G′ shown in Figure 4.5. The input nodes A_1, ..., A_p, and output nodes B_1, ..., B_q, of G′ are as shown in the figure. Without any loss of generality, assume in the following analysis that the two sets are disjoint. Let the edges input to the sub-DDG be numbered a_1, ..., a_r and the edges output from the sub-DDG be numbered b_1, ..., b_s, as shown there. The set of input edges, $\{a_1, \ldots, a_r\}$, can be divided into p disjoint subsets such that, each subset i has all input edges corresponding to node A_i. Similarly the set of output edges, $\{b_1, \ldots, b_s\}$, can be divided into q disjoint subsets such that, each subset j corresponds to the output node B_j.

Let $\mathbf{T}_{G'} \cdot \mathbf{N}_{G'} = \mathbf{0}$ be the set of TCEs for the sub-DDG, where $\mathbf{T}_{G'}$ is its topology matrix.

In Lemma 10 it was proved that a sub-DDG of a CDDG is a CDDG. Therefore, integer solutions exist for the above set of equations. Let $< N_i >$ be

[2]Alternatively, a connected subgraph is a subgraph which is weakly connected.

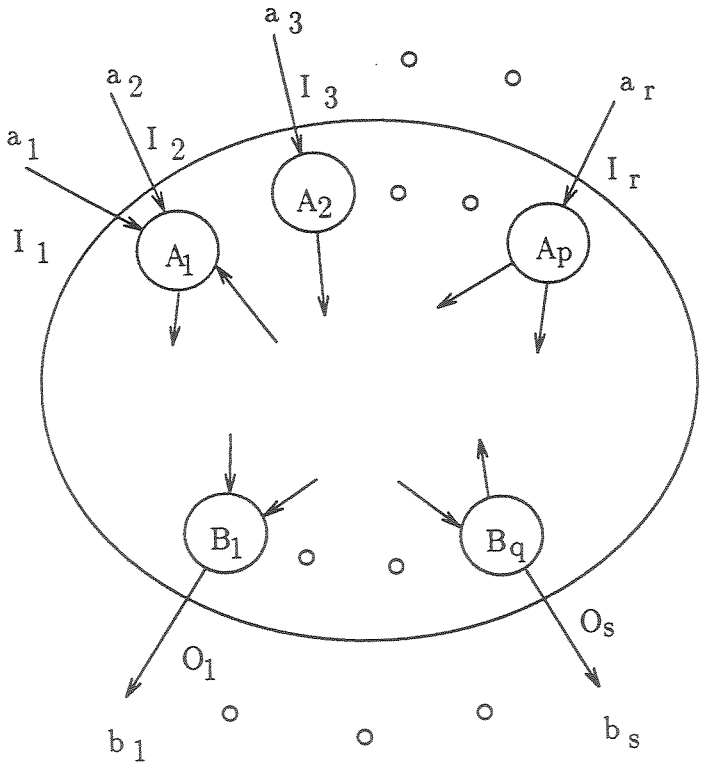


Figure 4.5: A Subgraph

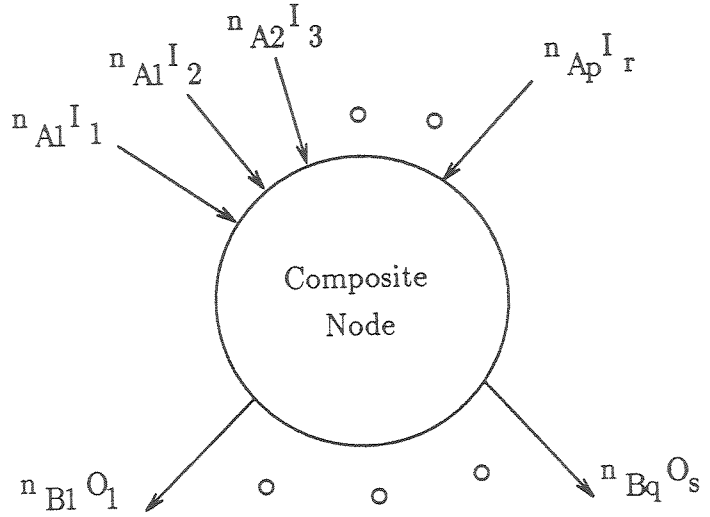


Figure 4.6: Single Node Equivalent Of The Subgraph

an integer solution to the above set of equations. Now consider the *composite* node shown in Figure 4.6. Notice that the number of tokens associated with each input and output edge equals the corresponding number in the sub-DDG times N_i, where N_i is the solution value for node i to which the particular edge belongs in Figure 4.5. This transformation is referred to as the *Subgraph-Collapse Transformation.*

It will now be proved that,

Theorem 6 *: A CDDG preserves its consistency property under the Subgraph-Collapse Transformation.*

Proof: Before proving the theorem itself, it will be shown that the consistency condition is satisfied in presence of the node representing the collapsed sub-DDG.

Consider Figure 4.7. The figure shows a loop in the UDDG corresponding to the DDG under consideration. The loop lies partly inside — path

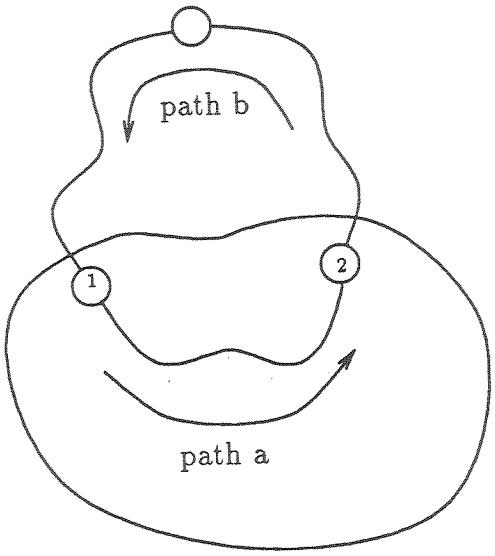


Figure 4.7: A loop in the UDDG lying partly inside the subgraph.

a, and partly outside the sub-DDG — path b. It is already known, $\rho_a \cdot \rho_b = 1$. According to the figure, the token markings I_1 and O_2 are included in the product-ratio ρ_b.

Under collapse, only the part of original loop that is also a part of the sub-DDG — path a, is affected; instead of path a, there is now a composite node with the shown dependency markings. Figure 4.7 shows the case where a path passes through the sub-DDG, in via node 1 and out via node 2. Since N_1 and N_2 satisfy the set of TCEs for the sub-DDG, they also satisfy the set of TCEs along path a. Thus, by Lemma 6, $N_2 = N_1 \cdot \rho_a$.

The product-ratio for the transformed loop is equal to

$$(\rho_b \frac{I_1}{O_2}) \cdot \frac{N_2 O_2}{N_1 I_1} = \rho_b \cdot \rho_a = 1.$$

Thus the new loop satisfies the consistency condition.

If a cycle is entirely contained within the sub-DDG, then of course the resultant CDDG does not have that cycle after the transformation.

Therefore, it can be concluded that the transformed DDG satisfies all conditions of a CDDG. Hence the theorem. □

Although no restrictions have been stated so far on how a sub-DDG is chosen to form a composite node, in the subsequent it will be assumed that the candidate sub-DDG will not contain a directed loop of the original DDG partially; a directed loop will either be contained wholly or it will lie completely outside. This is because, if a directed loop is partially contained within the collapsed sub-DDG, as will be seen later, it may not be possible to satisfy the token markings for the corresponding input and output dependencies of the composite node in a manner compatible with the intended semantics of its execution as implied in the lemma below.

The composite node implements the token count solution of the sub-DDG. It is worthwhile to notice that the number of tokens defined on the input dependencies of the composite node do indeed permit the execution of the chosen token count solution of the sub-DDG, and as such the output dependencies will indeed generate the appropriate number of tokens on the output dependencies. Stating this as lemma:

Lemma 11 *If the composite node obtained via a subgraph-collapse transformation of an acyclic sub-DDG is presented with tokens equal in number to the markings on its input dependencies, it can execute the corresponding token count solution of the sub-DDG and produce tokens equal in number to the markings on its output dependencies as a result.*

Proof: Since the sub-DDG is acyclic, its nodes can be sorted topologically so that each node in the sorted list depends only on the nodes before itself and/or on input dependencies of the sub-DDG. All the primary input

nodes appear at the head of such a list. These nodes can execute immediately at the begining since all their input dependencies have tokens on them. Each of these primary input nodes can fire the number of times determined by the token count solution used to form the composite node and its input and output dependency markings. They, therefore, generate tokens on their output dependencies so that their children nodes can fire the correct number of times as determined by the token count solution. And so on down the topologically sorted list. □

Definition 18 *Freedom from deadlock of a sub-DDG is defined as the ability to completely execute any of its token count solutions, given that tokens in an appropriate numbers are supplied over its input dependencies.*

That is, each node can be validly invoked the number of times stipulated by the solution. Here "validly" implies the satisfaction of the requirements implied by the node's firing discipline. Clearly then, from above,

Lemma 12 *An acyclic sub-DDG is free from deadlock.*

As will be illustrated with an example in a later section, presence of directed loops may make a sub-DDG (or a DDG) susceptible to deadlock, if sufficient number of initial tokens are not present on the edges forming the loop. Assuming that the given sub-DDG is deadlock-free, Lemma 11 can be extended to conclude,

Lemma 13 *If the composite node, obtained via a subgraph-collapse transformation of a deadlock-free sub-DDG, is presented with tokens equal in number to the markings on its input dependencies, it can execute the corresponding token*

count solution of the sub-DDG; it thereby produces tokens equal in number to the markings on its output dependencies.

In the foregoing, a token count solution of the sub-DDG was used while replacing it with a composite node. The composite node is assumed to behave like other nodes in the DDG. Thus it has the property of atomicity and behaves pursuant to the model described in Section 3.2.2. As a result, the composite node will wait until the numbers of tokens at its input dependencies equal the corresponding to the token markings on them. The token markings are proportional to the solution chosen to the TCEs of the sub-DDG. The larger the solution, the larger the token markings. And the larger the token markings, larger the waiting time before the composite node can fire. Besides, larger solution also implies larger computation and may imply larger execution time and/or more hardware to implement it. It would, therefore, make sense to usually choose the minimum solution to implement the composite node. This will tend to produce the least expensive functionally complete component with the fastest possible response time.

Often, the purpose of subgraph-collapse transformation will be to design a composite node in hardware. In such cases, it may also be desired that the number of ports in the component be lower than the number of dependencies for the sub-DDG it implements. That is, some of the dependencies will share a port in the component. This effect is easily specified at the DDG level by the use of an SR communication node to force the dependencies to share a port in the hardware realization. These ideas are made concrete via an example:

Example of composition

Figure 4.8a shows a sub-DDG that computes the Cartesian distance between two points on the x-y plane. In Figure 4.8a, the subgraph has four input dependencies. Suppose that the composite node is to have a single port and therefore the sub-DDG should have a single input dependency. A modified sub-DDG suitable for this purpose is shown in Figure 4.8b. Notice that the input node to the sub-DDG is an SR communication node.

The minimum token count solution of the sub-DDG is shown in parentheses next to each node in Figure 4.8b. The composite node is shown in Figure 4.8c. The composite node has only one input dependency and one output dependency. The token marking on the input dependency is 4 — equal to the product of the token marking on the SR communication node of Figure 4.8b, 2, and its token count solution value.

4.4 Execution graph of a DDG

For the purpose of scheduling a DDG, an important step in synthesis, and for defining the smallest possible input latency for a DDG under hardware delay constraints, there will be a need to define its execution behavior, in which it will be necessary to specify precisely which tokens are transferred from which source invocation to which destination invocation. A DDG's *execution graph* will be used to describe this information. The construction of an execution graph from an annotated DDG is explained in this section.

4.4.1 Construction of the execution graph

For a multi-rate DDG, a complete execution is specified by the the solution of its TCEs. Each invocation of an AND node in the DDG is represented by a node in the execution graph and each invocation of an SR node is

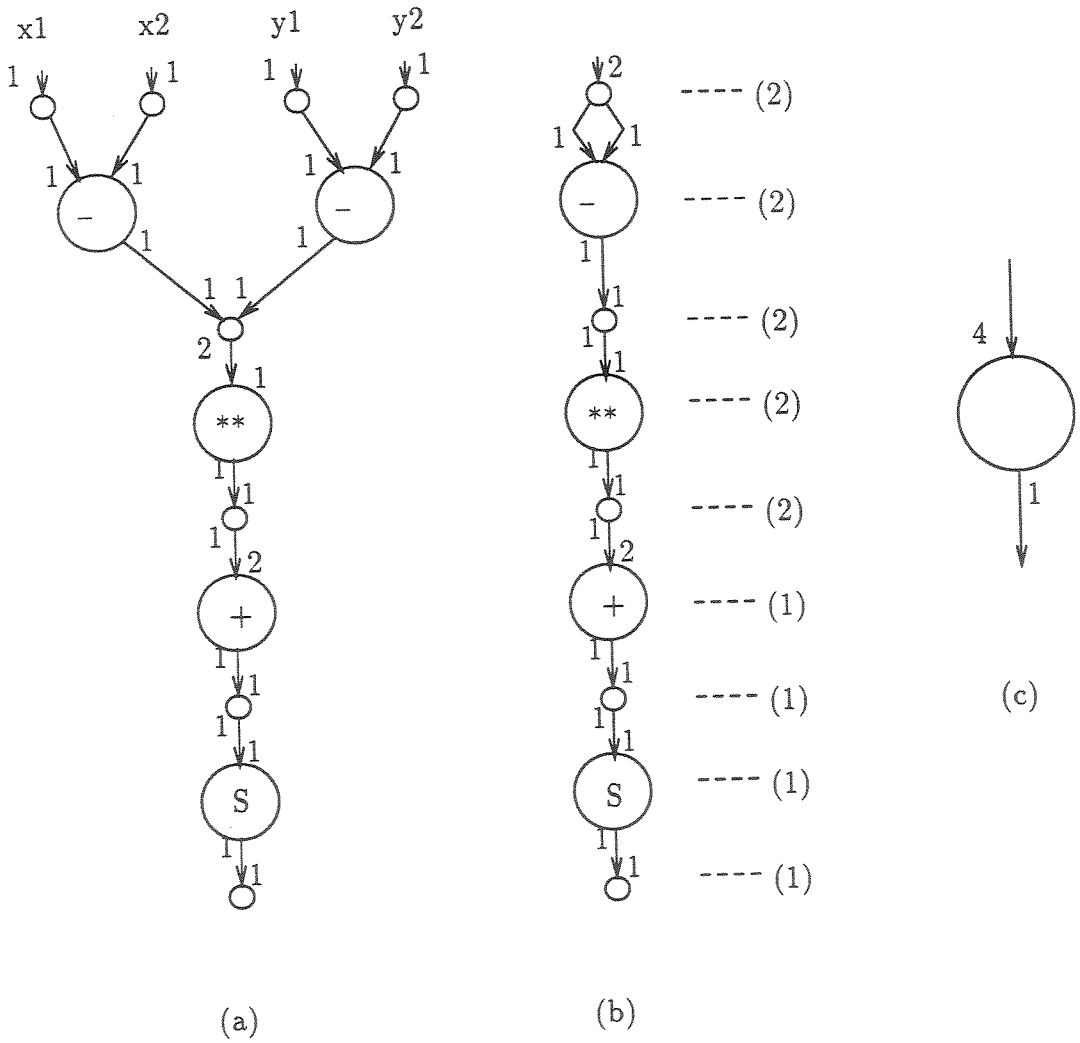
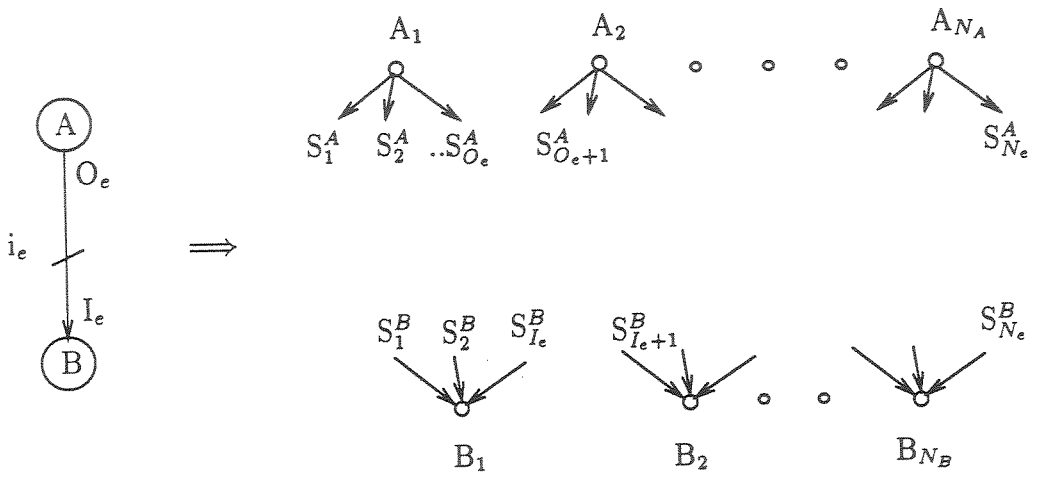


Figure 4.8: An example of a composition



(a) (b)

Figure 4.9: Translation of an edge in a DDG to a bipartite graph.

represented by a set of nodes, each node corresponding to a token the SR node transports. These nodes are now connected as follows:

Consider two nodes A and B in the DDG connected by an edge e. See Figure 4.9a. Let the token markings at the two extremities of the edge be O_e and I_e respectively, and let the initial tokens present on the edge be i_e, as shown in the figure. Thus the number of tokens emitted by an invocation of node A is O_e, and that absorbed by an invocation of node B is I_e. Let each token emitted by an invocation of A be given an *id*, the first token being given an id of 1 and the last, O_e. (These id's will be referred to again in Chapter 5.) The execution graph has nodes corresponding to the invocations of the two nodes. Let the numbers of invocations according to the token count solution be N_A and N_B resectively. By the semantics of the token count solution, the combined number of tokens produced by invocations of A equals $N_A \cdot O_e$, the combined number of tokens absorbed by invocations of B equals $N_B \cdot I_e$, and $N_A \cdot O_e = N_B \cdot I_e$. Let this number be denoted by N_e.

Now N_e directed edges are added between invocations of A and B to form a bipartite multigraph. Each of the N_e edges corresponds to a single token that will pass from an invocation of A to an invocation of B during a single invocation of the DDG, or equivalently, assuming a static schedule, during each schedule cycle. The edges are drawn as described below.

If either of A or B is of an SR type, then the corresponding nodes in the execution graph have a single edge incident on them. For nodes of type AND, each corresponding node in the execution graph gets either O_e or I_e edges depending whether it is an A or a B invocation respectively.

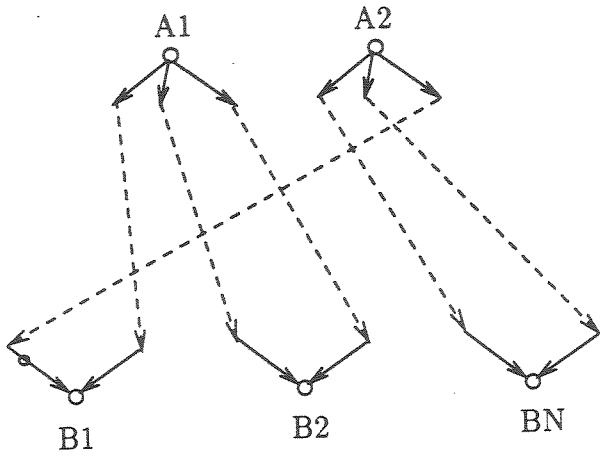
Let, for convenience, and without loss of generality, both A and B be of type AND. Label the A nodes in the execution graph, A_1 through A_{N_A}, and the B nodes, B_1 through B_{N_B}. Further assume that they are drawn in order from left to right.

Now *stubs* are drawn for each node on which the incident edges will terminate. Each node of type A has O_e stubs and each node of type B has I_e stubs. Label these stubs as S_1^A through $S_{N_e}^A$ on the A side and S_1^B through $S_{N_e}^B$ on the B side. (See Figure 4.9.)

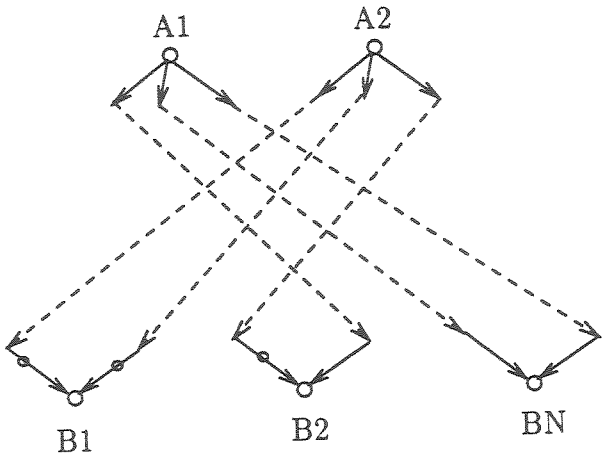
If the initial number of tokens, i_e, on the edge e in the DDG is zero, then the edges are drawn between S_j^A and S_j^B, $1 \leq j \leq N_e$, otherwise stub S_j^A is connected to stub $S_{(j+i_e) \bmod N_e}^B$, $1 \leq j \leq N_e$. Examples for $N_A = 2$ and $N_B = 3$ for i_e values of 1 and 3 are shown in Figure 4.10.

Initial tokens i_e are distributed to the edges of stubs starting with S_1^B, one per edge at a time and repeating if necessary.

Note that if either A or B is of type "SR", each invocation would correspond to O_e or I_e nodes respectively in the execution graph, each of which



(a) $i_e = 1$



(b) $i_e = 3$

Figure 4.10: Examples of edge translations.

would have a single stub associated with it. There will still be the same number of stubs, namely N_e, on each side.

The above construction is repeated for every edge of the DDG.

The execution of the nodes in the execution graph follows the "AND" convention. That is, each node now executes strictly when there is a token present on each of the incoming edge. And further, each node executes exactly once every complete invocation of the DDG. Since the entire execution graph is invoked once every schedule cycle, each node in the graph must execute exactly once every schedule cycle. Also, the entire execution graph, including the token assignment just described, is restored to its previous state at the end of the schedule cycle.

To see why the above distribution of tokens has the correct semantics, recall that the invocations of the nodes as specified in the token count solution were numbered starting from 1, which is the order in which the executions will occur. The initial tokens on the edges signify the tokens which are ready to be absorbed. The first I_e tokens will be absorbed by the first invocation of B, the next I_e tokens by the second, and so on. The above construction, along with the semantic that a node in the execution graph can be executed when each input edge has a token on it, achieves exactly this effect

Notice also that after each node in the execution graph has executed, each of the edges drawn above receives exactly one additional token, which is also the number of tokens absorbed from it. The edges which received a distribution of the initial tokens are thus restored their original number of tokens, preparing the execution graph for the next repetition of the DDG computation.

4.5 Equivalence transformations

Two equivalence transformations for DDGs are introduced in this section; one for delays and another for tokens. These transformations are in the same spirit as those used for signal flow graphs [17] and for systolic circuits [36].

The *delay-transformation* is used while analyzing the execution graphs, whereas the *token-transformation* is used to draw conclusions about properties of the DDGs themselves. It will be shown later that when tokens are interpreted as delays, the two transformations become equivalent.

4.5.1 Equivalence transformation for delays

Figure 4.11 shows two possible transformations of delays across a node of an execution graph. It shows that configuration a is equivalent to configuration b, and configuration c is equivalent to configuration d. The transformation a$\longrightarrow$b is called the *forward* delay-transformation and the other, c$\longrightarrow$d, is called the *backward* delay-transformation.

In the pair a$\longrightarrow$b, a unit delay at an input edge is transported across the node to the output edges by subtracting it from each input edge and adding it to each output edge. In the inverse operation, c$\longrightarrow$d, a unit delay is transported across the node from an output edge to the input edges by subtracting it from every output edge and adding it to each input edge.

The first *delay-transformation* implies that if it is *possible* to schedule an invocation of a node a unit delay (e.g., a clock cycle) earlier, then it is permissible to do so in terms of *correctness of the operation* and the schedule. In the signal flow terminology, the delay-transformation is often referred to as

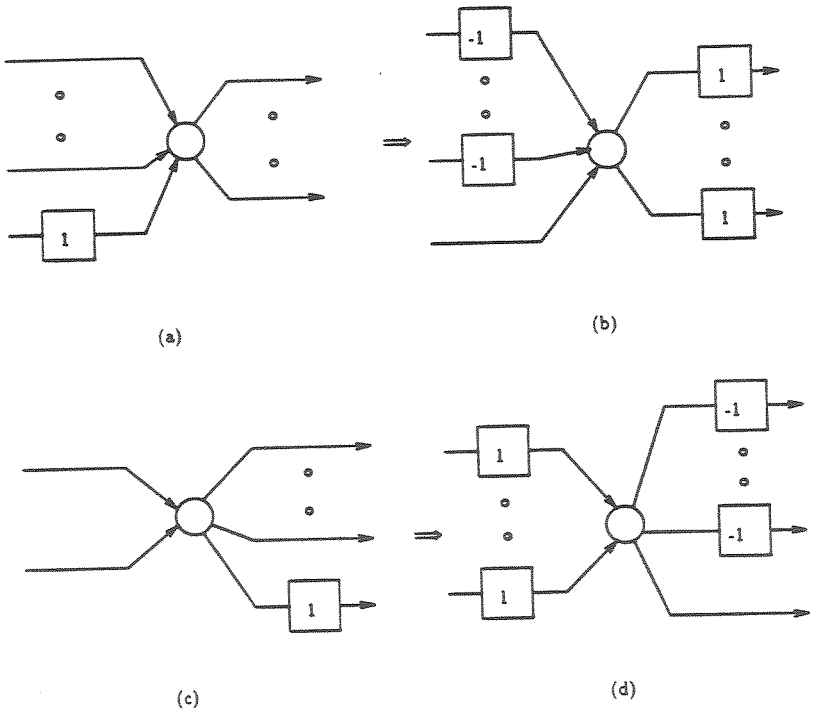


Figure 4.11: Delay transformations.

the *shift-invariance* of computation. For the execution model of a hardware component proposed in the last chapter, it is intuitive that this is indeed so. The subtraction of a unit delay over one of the input dependencies will force the input phase to start a unit delay earlier and this will be reflected on all other input dependencies as well. Since the delay of the operation is independent of the schedule, the output phase also will consequently terminate earlier by a unit delay. The second delay-transformation implies a reverse operation of postponement of an invocation and is equally intuitive.

It should be clear that the delay-transformations may be applied repeatedly to a node to transfer multiples of unit delays across a node.

The concept of delay-transformation is generalized to a subgraph as shown in Figure 4.12. As in the previous case, these transformations imply shifting in time the cumulative schedule of the entire subgraph. Since this transformation implies that the relative times of execution for invocations within the subgraph be kept constant, it follows that none of the data dependencies are violated within the subgraph.

In the above, it was said that the delay-transformation is permissible if it is *possible*. The delay-transformation is possible if the necessary delays are *available* on the edges. For example, for the first delay-transformation, there must be unit delays available on every input edge of the node, and for the second, a unit delay must be present on every output edge. Since a start node of an acyclic graph has no input edges, the first transformation may be applied to it any number of times; and since its stop node has no output edges, the latter transformation may be applied to it any number of time. Similar arguments also hold for subgraphs with unidirectional cutsets, such as cutsets that naturally occur for subgraphs of acyclic graphs, and which include either

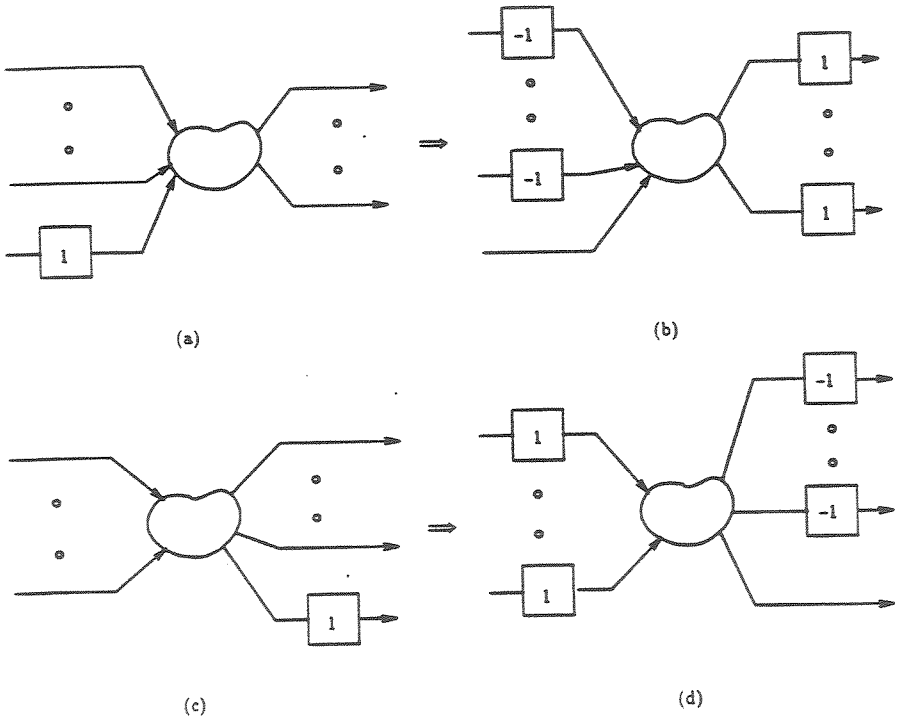


Figure 4.12: Delay transformations for subgraphs.

the start node or the stop node of the graph. These transformations can be invoked to justify insertion of arbitrary amount of *slack* in the necessary edges. This fact will be assumed in the next chapter, where scheduling of acyclic DDGs is discussed.

4.5.2 Equivalence transformation for tokens

Figure 4.13 shows two possible transformations of tokens across a node of a DDG. Note that the tokens are transformed in terms of the token markings

of corresponding edges. As in the case of delay-transformations, the two *token-transformations* are inversely related. As with the delay-transformations, the transformation a$\longrightarrow$b is called the *forward* token-transformation, and the other, c$\longrightarrow$d, is called the *backward* token-transformation.

The token-transformations imply a change of state of a computation as a result of the execution of an invocation. The first token-transformation corresponds to the change of the state of the computation after an invocation of the node, whereas the second produces the state prior to the execution of the invocation.

Since the number of tokens on an edge can never be negative (see Section 3.1.2), the first token-transformation also implies the firing requirement of an AND node. That is, for a permissible transformation, each input dependency must have number of tokens no less than the corresponding token-marking. Despite the differences in the execution semantics, it is easy to see that the transformation is equally valid for an SR communication node.

If there are no input dependencies for a node, the first token-transformation is always valid. Thus for the input node of the DDG, the token-transformation can be applied at any time. Similarly, the second token-transformation may be applied to the output node at any time.

The concept of token-transformations is extended to a subgraph. This extension is along the same lines as the subgraph-collapse transformation introduced in the last section, and in fact mimics an execution, and undoing of an execution, of a token count solution of the sub-DDG under consideration:

The single node shown for the nodal token-transformations is replaced by the boundary defining the subgraph, and in place of the dependencies of the node, the dependencies crossing the boundary of the subgraph are used. The

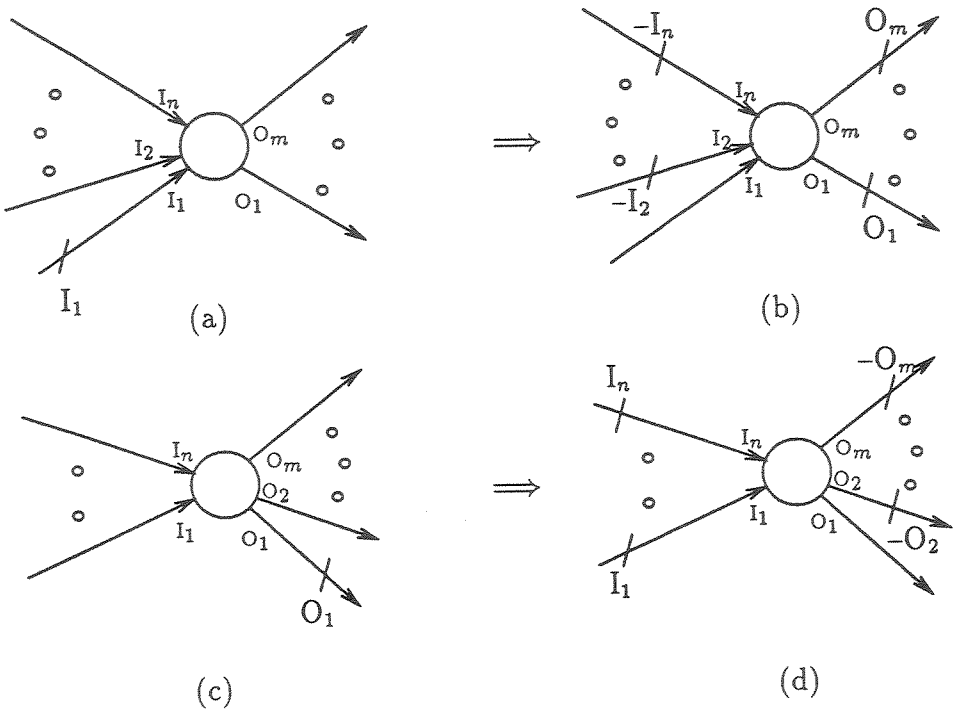


Figure 4.13: Token transformations.

token markings now correspond to the number of tokens transferred over the dependencies for an execution of the minimum token count solution of the sub-DDG. The representations of the two token-transformations applicable to sub-DDGs are shown in Figure 4.14.

Lemma 14 *Token-transformations are valid for non-deadlocking subgraphs.*

Proof: Since token-transformations for individual nodes are equivalent to complete executions of their invocations, Lemma 13 implies this lemma. □

Note that the token-transformations for sub-DDGs can be applied repeatedly to absorb and generate multiples of the token markings referred above.

As was the case for the input node of a DDG, any non-deadlocking sub-DDG containing the input node with a cutset directed out from it can

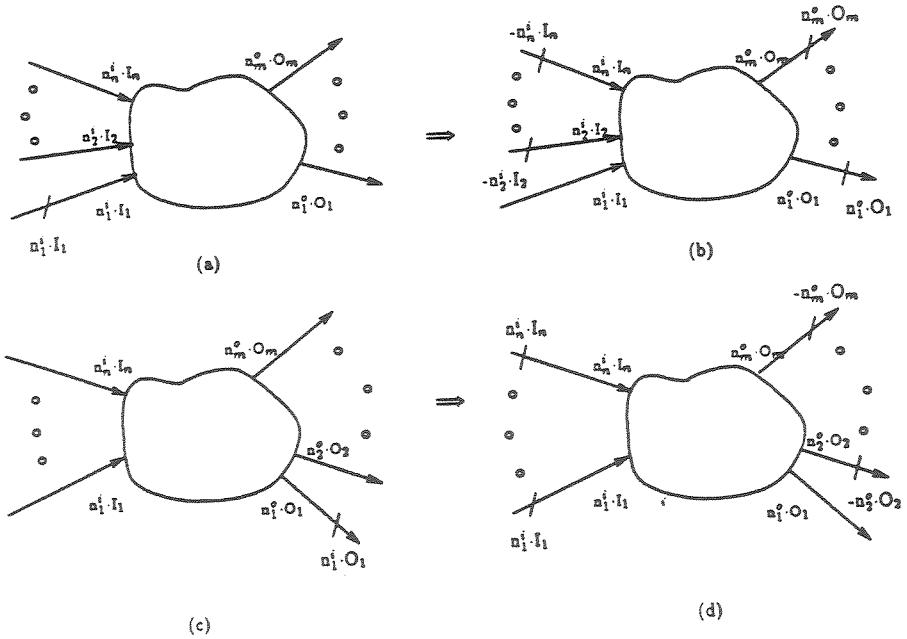


Figure 4.14: Token transformations for subgraphs.

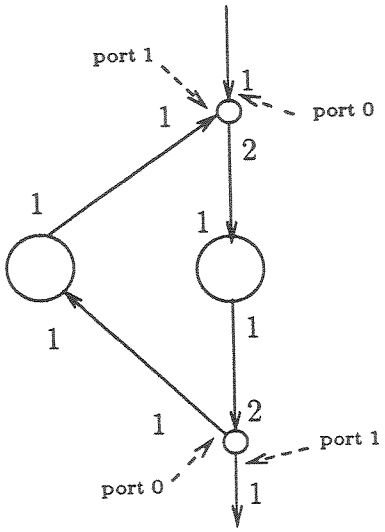


Figure 4.15: A DDG loop with zero initial tokens and no deadlock.

be freely transformed to produce tokens over its out-going dependencies. The numbers of these tokens are determined by the token count solution used for the sub-DDG. Of course, since the set of edges of a sub-DDG are a subset of that of a DDG, a token count solution of the DDG also contains a multiple of the minimum token count solution of the sub-DDG. This multiple may also be used for the token-transformation.

4.6 Initial tokens in loops

Figure 4.15 shows an example in which the use of SR communication nodes avoids the deadlock despite the fact that the edges of the directed loop have zero initial tokens. This is because the execution graph of the DDG does not contain a directed loop. In general, however, directed loops in a DDG will result in directed loops in its execution graph and absence of tokens on the edges will result in a deadlocked execution.

For a single rate DDG, such as a signal flow graph, a single token in

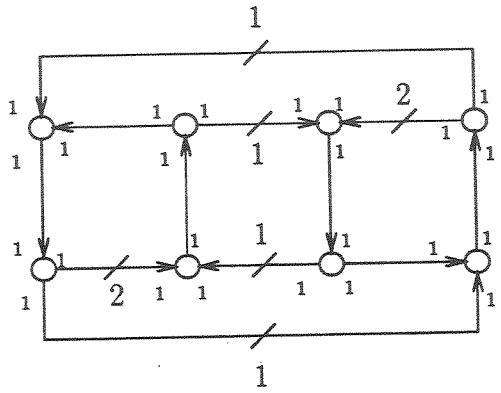


Figure 4.16: Interlocked loops with distributed initial tokens.

any of the edges of the loop is sufficient to guarantee freedom from deadlock. But for a general, multi-rate DDG the number of tokens may have to be more than one. In this section, the sufficient condition, in terms of number of tokens that must be present on the edges of a DDG loop, to guarantee freedom from deadlock, is stated. Analysis here is restricted to the case of a simple loop with initial tokens distributed such that only one of its edges has sufficient number of tokens to enable the invocation of its to-node. As will be seen later, certain types of loops, such as *independent loops* (defined in the next chapter), which may have initial tokens distributed over more than one edge can be reduced to the "single token-carrying edge" case using token-transformations. For more general loops, which share nodes with more than one loop, it may not be possible to move all tokens to a single edge of the loop. Figure 4.16, where all nodes are assumed to have AND firing discipline, shows one example of this last case.

4.6.1 Conditions for deadlock freedom in loops

Consider the case of a loop that does not share its nodes with any other loops. The nodes of this loop may have incident on them edges other

than the loop edges. All such output dependencies may be ignored, since they cannot affect the firing of the nodes of the loop. Input dependencies other than those of the loop may be ignored as well for the following reasons: These input dependencies go to the ancestor nodes. Since all nodes of a DDG are reachable from its start node, a sub-DDG consisting of the start node and all the ancestors of the loop nodes can be identified. Assuming that this sub-DDG is non-deadlocking, Lemma 14 allows the assumption that adequate tokens can be inserted in the said input dependencies to let the corresponding loop nodes to fire the number of times given by the token count solution of the DDG. Note that none of the output dependencies deleted earlier could go to any of the ancestor nodes lest the loop would not be independent of other loops. So, only the loop and its edges remain, and the effect of initial tokens along these edges alone need be analyzed. This leads to the following important conclusion:

Lemma 15 *An independent loop of a non-deadlocking DDG can be independently scheduled.*

Proof: As was just seen, a sub-DDG comprising of the start node of the DDG and all the ancestors of the nodes of the loop can be identified. Tokens can be introduced on the input dependencies of the nodes of the loop (other than those dependencies that are part of the loop itself, of course) such that each node may execute all its invocations corresponding to the token count solution chosen for computing its schedule. Thus the computation of the schedule for the nodes of the loop can be made completely independent of the schedule of other nodes of the DDG. □

The above lemma should not be taken to mean that the schedule of an independent loop must be obtained independently. It is stated merely to

establish feasibility. In fact, scheduling the loop independently will generally lead to greatly increased response times since the scheduling of the loop will have to wait until all of the requisite tokens are available on the said input dependencies. Nonetheless, the lemma is a useful result to know since it permits the application of the divide-and-conquer technique to reduce the complexity of DDG scheduling.

In an independent loop, at any given time, there has to be at least one node ready to fire, or else the loop is deadlocked. For the execution to continue forever this condition must remain true at all times. If ever after finite time, the above condition becomes false, the loop will enter dead-lock, and will remain so forever afterwards.

Consider selecting one of the edges of the loop. Via token transformations, all initial tokens can be collected on this one edge such that, only the node on which the edge terminates can fire, and none other. This process may leave some tokens on the other edges, but these are not enough to allow the corresponding node to fire; otherwise, of course, the token-transformation process is applied again. Before going on, this is formalized as,

Lemma 16 *For an independent directed loop of a non-deadlocking DDG, with initial tokens on more that one edges, token-transformations can always be used to obtain a configuration in which the initial tokens in the loop enable the firing of only one of the nodes; all other nodes are disabled and cannot fire.*

So now the following scenario exists: There is a directed loop in which all but one nodes are disabled and multiple invocations of the enabled node may be fired at once.

Lemma 17 *The sufficient condition for an independent loop to be deadlock-free is that the enabling arc carry number of initial tokens equal to or more than that necessary to complete the execution of minimum token count solution of the loop.*

Proof:

To prove the sufficiency condition, notice the following. Since the original DDG is consistent, the initial tokens are restored to their original numbers after the execution of invocations determined by the minimum solution for all the nodes of the loop and hence another execution of the minimum solution can be initiated. This can be repeated any integral number of times to execute any solution of the loop.

The token count solution of the DDG implies an integer multiple of the same for any of its sub-DDG, and therefore for the loop under consideration. This leads to the conclusion that the loop will not deadlock during the execution of the token count solution of the DDG. □

It was assumed in the foregoing that the candidate loop is independent of other loops. The implication was that it is possible to apply token-transformation to move the tokens to one of the edge so that only its *to* node is enabled. However, this transformation was merely used to simplify the proof; it is not a necessary condition. As illustrated in Figure 4.16, for non-independent loops, it might be impossible to isolate all enabling tokens on a single edge. Yet if the tokens in the loop allow a complete execution of its minimum token count solution, it is certain that that particular loop will not enter deadlock on its own account, since its complete execution restores all initial tokens back to their original values. Thus the scope of Lemma 17 is broadened thus:

Lemma 18 *A DDG loop is free from deadlock if, and only if, the initial tokens on its edges allow a complete execution of its minimum token count solution.*

It is now easy to see why for a single rate DDG such as one represented by a signal flow graph, a single token per loop is sufficient to guarantee freedom from deadlock. This is because, each node in such a DDG executes exactly once during the invocation of the DDG and absorbs and produces exactly one token during that execution.

Chapter 5

Synthesis of Architectures

There are three important steps to the synthesis of architectures. First, a decision must be made on how to implement the operations of the DDG. As explained in Chapter 3, in the new methodology being described, this decision is made already by the choice the designer makes about the component-type for each node. The synthesis process must still compute the *number of each component-type* that will be present in the architecture.

Second step is to *schedule* the individual operations of the DDG so that the sequencing relationships are preserved. The scheduling is guided by the execution graph (discussed in Chapter 4), which defines the necessary precedence relationship between invocations of the nodes. Computation of a schedule involves definition of time instant at which an invocation begins its execution and the sequence of control micro-orders to be issued to the component executing it. However, as was seen in Chapter 3, the execution model of a component is known beforehand, and so, once the execution start times are computed, the sequence of micro-orders is easy to generate.

Third, the invocations have to be *assigned* to individual instances of the hardware components. This assignment determines the *connectivity* of the architecture, i.e., the information about connections between ports of components and busses. The connectivity also provides information about the number of register-files in the final architecture (see Chapter 3).

Often, the scheduling and assignment steps of the synthesis process

cannot be separated. But in the case of acyclic DDGs, (Section 6.3,) the two parts can be handled independently, the scheduling process preceding the assignment process. When the two parts are so separated, the output of the scheduling process affects the input to the assignment process.

The first section of this chapter lists various design objectives used in the architectural synthesis for repetitive computations.

5.1 Optimization criteria

Architectures for repetitive computations are synthesized subject to the following three independent parameters:

- Input latency.
- Graph latency.
- Hardware resource requirements (i.e., number of computation and communication components).

The actual optimization criterion chosen may be a combination of the above three parameters.

Input latency:

Input latency was defined in Chapter 3. Whenever a DDG and its execution graph have directed loops in them, there is an upper bound placed on how often the DDG may be invoked; the bound is imposed by the delays of the components implementing the operations. Since one important objective in designing a pipelined architecture is to obtain a high computational throughput, a question arises: what is the highest possible throughput? Or

equivalently, what is the minimum input latency? Often, however, the input latency is pre-determined by the application.

Graph Latency:

For single rate DDGs, graph latency is defined to be the time delay between the beginning of execution of the *start* node of the graph to the end of execution of its *stop* node. A definition of graph latency for multi-rate DDGs is more complicated and will be further expanded upon in Section 5.5.

Graph latency, also sometimes referred to as the response time, or simply, latency, of the pipeline, is another design criterion of importance. Many real-time applications with hard deadlines require response to an input to appear within a pre-specified time period, which implies that the invocation of the DDG-computation for a given input must be completed within that time period. (The effect of input latency on the schedule of a directed loop has an equivalent effect on its execution graph.)

Resource requirements

The synthesis process is always aimed at minimizing the number of hardware components used in the architecture and thus utilizing them to the maximum. In the presence of graph latency constraint, however, the complexity of optimizing this parameter is prohibitively high (NP hard), and, often, a sub-optimal solution is acceptable.

5.1.1 Current scope

Section 1.3 stated the scope of the current research. To repeat, the synthesis problems studied here are:

For a given multi-rate repetitive algorithm, given that

- *each operation in the algorithm is executed on a unique component-type,*
- *communication operations are executed via busses, and*
- *the cost of the architecture is measured in terms of numbers of computation and communication components and the connectivity of the architecture,*

1. *for a cyclic DDG, find the lower bound on the input latency (minimum input latency).*
2. *given the input latency for the algorithm, find a minimum cost architecture*

Section 5.2 contains the development of the theory to solve the first part. It also indicates briefly a method to obtain an architecture which realizes the minimum input latency.

The general architectural component minimization problem for cyclic multi-rate DDGs with pre-specified input latency is formulated next in Section 5.3. The general problem is NP-hard, and so the subsequent interest has been in special cases which can be solved more readily.

The exact solution for acyclic, multi-rate DDGs, with unconstrained graph latency, using a greedy, heuristic scheduling method is considered in Section 5.4.

Following that, in Section 5.5, a cyclic DDG with a single outside loop is shown to be equivalent to a DDG with a graph latency constraint. This problem is known to be NP-hard, and so an iterative algorithm, which

presumes the existence of a resource-constraint scheduler, to achieve a graph latency-constraint scheduler, is presented. Its termination is also proved.

Finally, two special cases of multi-rate DDGs with loops, DDGs with *independent* or *nested* loops, are studied to develop divide-and-conquer scheduling heuristics to lower the complexity of the scheduling process. The effort here has been to find a condition under which the heuristic can be applied and to prove that the heuristic will yield a correct synthesis.

5.2 Minimum latency scheduling

In the last chapter the construction of the execution graph for a multi-rate DDG was described. Here, appropriate timing semantics, which facilitate the computation of the lower bound on input latency under hardware delay constraints, are introduced into the execution graph.

5.2.1 Delay semantics in execution graphs

In the original DDG the delays are associated with its nodes, and edges have no delays associated with them. In the execution graph, however, it is preferable to associate appropriate delays with the edges, and not with the nodes, in anticipation to the transformation to be used.

Recall that the execution of a node has three, possibly overlapping, phases, namely input, computation, and output. The execution starts with the input phase when all necessary tokens are available in the input buffers for firing, and it ends with the termination of the output phase. The length of the output phase is determined by the maximum number of tokens output per execution on any of the node's output edges.

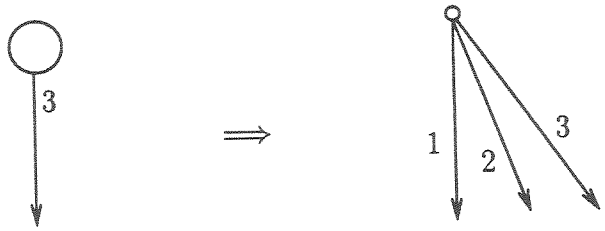


Figure 5.1: Distribution of delay over edges in the execution graph.

A delay equal to the total execution delay of a node minus the length of the output phase is assigned to each of the output edges of the node. Additional delays are assigned to the output edges in accordance with the order in which tokens are released. The first edge to receive the token is assigned an extra delay of one time unit, the second, two time units, and so on. Figure 5.1 shows an example of such distribution in the execution graph for a single edge of the DDG.

Delay semantics of initial tokens in the execution graph

Now, delay semantic are assigned to the initial tokens in the execution graph. In fact, the delay semantic being introduced is valid for any token. Since each token in the execution graph is replaced every schedule cycle, a delay of $-L_{Sch}$ (i.e., schedule cycle length; see Chapter 3) is associated with each token. This delay is assigned to the edge on which the token resides; the presence of token is henceforth ignored. Notice that this delay is of the sign opposite to that of the execution delays assigned above, since it represents the time *available* for execution as opposed to the time that is *spent* during execution.

Consider how this assignment of delays to the tokens makes the delay and token transformations introduced in Section 4.5 equivalent. Here only the forward transformation will be explained; explanation for the backward transformation are similar.

Recall that in the forward token-transformation, the edges lose or gain tokens equal in number to their token markings, and each application of the token-transformation implies the execution of an invocation. In the execution graph, the firing of an invocation results in absorbing a token from every input edge and putting a token on every output edge. Thus, it can be concluded, that each application of the forward token transformation to a DDG node corresponds in the execution graph to absorption of a token on each input edge of the respective node and putting out of a token on each of its output edge.

The forward delay-transformation, on the other hand, requires that equal delay be absorbed from each input edge and the same delay be inserted in the output edges. Given that each token is equivalent to a delay of L_{Sch}, this exactly what the token transformation achieves.

In fact, as long as all tokens are assigned the same delay, the two transformations are mutually equivalent. The choice of $-L_{Sch}$ is dictated by the fact that a particular invocation in the execution graph is fired exactly once every L_{Sch} units of time, after which the tokens it absorbed during the previous firing will have been restored.

Minimum input latency computation

Using the construction described in Section 4.4, the original multi-rate DDG is converted into an execution graph, in which the nodes fire upon receiving one token on each of their input edges. Further, tokens are translated to delays and these are assigned to the respective edges. In effect, a simple directed graph, with special delay semantics, just described, associated with its edges, is obtained. The lower bound on the input latency, $L_{I_{min}}$, can now be computed.

The arguments used here are essentially the same as those of Renfors and Neuvo [51]. For multi-rate DDGs, the analysis is presented below.

The choice of input latency is dictated by the loops in the graph since they impose a self dependency for the operations in them. The value of the input latency must be such that in every loop, the cumulative delay of execution of operations must be less than or equal to the delay afforded by the presence of initial tokens.

As seen previously, the presence of initial tokens in a loop of the execution graph indicates time available to complete the execution of all the operations belonging to it. Each token corresponds to L_{Sch} $(= N_{Sch} \cdot L_I)$ units of available time. For each loop in the execution graph, the available time must equal or exceed the sum of execution delays. The following makes this idea formal.

Two execution graphs are *essentially equivalent* if they are obtained by applying a sequence of delay-transformations described in the previous chapter.

Consider a *spanning tree* of an execution graph. The edges that belong to the tree will be called *tree-edges*, and the remaining are called the *link-edges*. Adding a link edge to the spanning tree causes a loop to be formed, if the directions of the edges were disregarded. These loops are called a *complete set of fundamental loops* [17].

For every tree-edge of the spanning tree, a cut-set which includes it and no other tree-edges can be found. By applying the delay-transformation for subgraphs to this cutset, the entire delay on the tree-edge can be transferred to the link-edges. This can be done for every tree-edge, and thus delays from all the tree-edges can be transferred to the link-edges.

If d_e is the delay on the tree-edge e, this delay is transferred to a link-edge as either d_e or $-d_e$ depending on whether the orientation of the link-edge coincides with that of the loop — the orientation of the loop matches that of the tree-edge.

It can be shown easily that each link-edge receives the signed sum of delays of the edges of the fundamental loop it forms. The delay of a tree-edge *adds* to the delay of the link-edge if the two edges have the same orientation with respect to the fundamental loop they are parts of. For a directed fundamental loop, the corresponding link-edge receives all additive delay contributions; for the non-directed fundamental loops, the contributions are not all additive.

The requirement that the available delays in the directed loops be greater than or equal to the cumulative execution delays of their respective operations can be restated in terms of delays on the links following the aforementioned transformations: *every link-edge corresponding to a directed loop carry a non-positive delay.*

The non-positivity of the link-edge delays allows the insertion of extra delays, (*or slacks, or shimming delays,*) along the link-edges to make the resultant sum to be zero, which is the indication that the execution delays equal the available delays.

The non-positivity of the link-edge delays, in general, depends on two factors: 1) the choice of L_{Sch}, and, 2) the choice of the spanning tree. It will be seen, however, that the choice of L_{Sch} makes the post-transformation link-edge delays for directed loops non-positive independent of the spanning tree, and that, given such an L_{Sch}, a spanning tree can always be chosen such that for it, in particular, the non-directed loop link-edges also carry non-positive post-transformation delays. The selection of such a spanning tree and the

addition of the necessary positive slacks along the link-edges ensures that all data dependencies are satisfied, which offers a method for obtaining a feasible minimum-latency schedule.

Each directed fundamental loop, l, of the execution graph has a delay of the form $(\nu_l \cdot (-L_{Sch}) + D_l)$ where D_l is the cumulative delay of operations in the loop assigned to edges and ν_l is the number of token delays in the loop. Thus for each loop, l, the following must be true:

$$(\nu_l \cdot (-L_{Sch}) + D_l) < 0$$

That is

$$\nu_l \cdot L_{Sch} > D_l$$

This inequality is satisfied for all loops if

$$L_{Sch} = \max l\{\frac{D_l}{\nu_l}\}$$

or, since L_{Sch} can be expressed as $L_I \cdot N_s^{Sch}$, where N_s^{Sch} is the number of invocations of the start node,

$$\nu_l \cdot N_s^{Sch} \cdot L_I > D_l$$

L_I must be chosen such that,

$$L_I = max\{\frac{D_l}{\nu_l \cdot N_s^{Sch}}\}$$

The loops for which this maximum holds are called *critical loops.*

To see that this choice of L_I, satisfies the non-positivity requirement for any spanning tree, it merely needs to be observed that for any spanning tree, each directed loop will contribute one of its edges as a link-edge, and for this

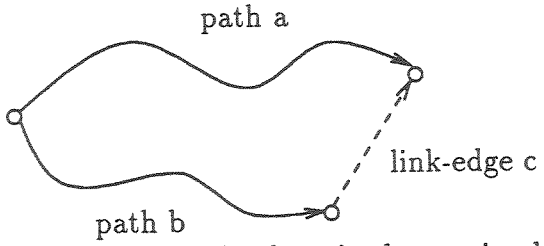


Figure 5.2: Delay configuration for a link-edge c in the maximal delay spanning tree.

edge the delay sum always remains the same ($n_l \cdot (-L_{Sch}) + D_l$), independent of the link-edge, and therefore the spanning tree.

If this value of L_{Sch} is substituted in the execution graph delays and a *maximal delay* spanning tree is obtained, the delays for all of its link-edges will be non-positive. To see this, consider Figure 5.2. Let D_a and D_b be the delays along paths a and b, respectively. Assuming that $D_a \geq D_b + D_c$, paths a and b are parts of the maximal spanning tree. Edge c, with delay D_c, does not belong to the tree and is, therefore, a link-edge. After the transformations, the final delay carried by edge c will be $D_c - D_a + D_b \leq 0$.

The selection of the maximal spanning tree following the selection of input latency and the subsequent insertion of slacks in the appropriate link edges results in the computation of a feasible minimal input latency schedule. The time instants of the start of execution of different invocations are obtained by adding up the delays to the corresponding nodes, along the tree, beginning with the start node, and computing their moduli with respect to L_{Sch}.

The time complexity of finding the most critical loop can be evaluated by evaluating the complexity of the constituent steps: The number of loops in a directed graph is bounded to be $O(|E|)$, where E is the number of edges in the execution graph. The complexity of finding all the loops in the execution graph and evaluation of their delays is the same as that of labeling all these loops,

which can be achieved in $O(|E|^2)$ time using *depth first search* traversal of the graph and using a *stack*-oriented data structure. Finding the most critical of these loops can be achieved in time bounded by the number of loops in the graph, which is $O(|E|)$. Computation of the maximal spanning tree is $O(|E|)$ in time complexity. So, the overall complexity of finding the minimum input latency schedule is bounded to be $O(|E|^2)$.

It should be noted, however, that although the time complexity of finding a minimum input latency schedule is polynomial in terms of the number of edges in the execution graph, such a schedule may judiciously utilizes the hardware resources of the architecture. In the next section, the resource minimization is formulated as an integer programming problem.

5.3 Integer programming formulation

In this section the problem of finding a *minimum resource requirement schedule* for a cyclic, multi-rate DDG is formulated as an integer programming problem. The resource requirement is interpreted here as the total number of hardware components, both communication and computation. (The cost of buffers and connectivity are ignored in this formulation.) The execution graph of the DDG is used for defining the problem, and it is assumed that it corresponds to the appropriate token count solution of the DDG.

As has been assumed throughout the preceding, all delays and times are measured in terms of numbers of clock cycles of the global system clock. All references to invocations of the nodes and the dependency edges are in relation to the execution graph. To begin with, the following notation, used in formulating the problem, is introduced.

Notation

N_s^{min}	Number of invocations of the start node as determined by the minimum integer solution of the TCEs of the DDG.
N_s	Number of invocations of the start node in the schedule = $\phi \cdot N_s^{min}$, where ϕ is an integer ≥ 1 which is the number of times the minimum solution is repeated to compute the schedule (see Section 4.2).
N_i	Number of invocations of node i in the schedule.
L_I	Input latency.
L_S	Schedule latency = $N_s^{min} \cdot L_I$.
L_{Sch}	Schedule cycle length = $\phi \cdot L_S$.
$< i, j >$	j^{th} invocation of node i.
$t_{i,j}$	Time of start of execution of $< i, j >$.
$s_{i,j,k,l}$	Slack inserted between $< i, j >$ and $< k, l >$.
$\tau^e_{i,j,k,l}$	Maximum token id emitted by $< i, j >$ corresponding to dependency edge e and transferred to $< k, l >$ (see Section 4.4).
O_i	Operation represented by node i.
D_i	Execution delay of O_i.
o_i	Length of the output phase of O_i.
δ_i	= D_i - o_i.
$\nu^e_{i,j,k,l}$	Minimum number of initial tokens distributed to an execution graph edge e between $< i, j >$ and $< k, l >$.

$\mathbf{n}_{comp}$ Number of types of components.

$\mathcal{N}_k$ Number of components of type k required in the architecture.

To compute the cost of the architecture, the following notation is used. The contribution to cost by invocation $< i, j >$, is denoted by $c^{\tau}_{i,j}$, during cycle τ, $0 \leq \tau < L_{Sch}$, of the schedule. The contribution reflects the requirement for a hardware component to execute invocation $< i, j >$.

$$c^{\tau}_{i,j} = \begin{cases} 1 & \text{if } 0 \leq l < D_i, \\ & (t_{i,j} + l) \bmod L_{Sch} = \tau \\ 0 & \text{otherwise} \end{cases}$$

The number of components of type k, $\mathcal{N}^{\tau}_k$, required during cycle τ, $0 \leq \tau < L_{Sch}$, of the schedule is given by

$$\mathcal{N}^{\tau}_k = \sum_i \sum_j c^{\tau}_{i,j}, \text{ such that } O_i = k.$$

The number of components of type k required in the architecture, $\mathcal{N}_k$, is given by

$$\max_{0 \leq \tau < L_{Sch}} \mathcal{N}^{\tau}_k$$

The cost of the architecture is defined by

$$\mathbf{C}_A = \sum_{k=1}^{\mathbf{n}_{comp}} C_k \cdot \mathcal{N}_k$$

where C_k's are constants.

The constrained optimization problem for a given DDG and a given input latency, L_I, can now be stated thus:

Given a ϕ, the multiple of the minimum token count solution, and given the following constraints,

- $\forall$ i, j, $t_{i,j} \geq 0$,
- $t_{s,0} = 0$, and $0 \leq j \leq N_s-2$, $t_{s,j+1}-t_{s,j} = L_I$,
- $\forall$ i, $0 \leq j \leq N_i-2$, $t_{i,j+1}-t_{i,j} \geq 0$,
- $\forall$ i, j, k, l, such that $\exists$ an edge from $< i,j >$ to $< k,l >$, $s_{i,j,k,l} \geq 0$,
- $\forall$ i, j, k, l, such that $\exists$ a dependency edge e from $< i,j >$ to $< k,l >$, $t_{k,l}-t_{i,j}-\delta_i-\tau^e_{i,j,k,l} = s_{i,j,k,l}-L_S \cdot \nu^e_{i,j,k,l}$,

minimize C_A.

It has been assumed that there is at least one loop in the execution graph. If the DDG is acyclic, however, and the problem will become unconstrained. It is also assumed that there are no response-time or *graph latency* constraints on the schedule, but one may also be imposed. The presence of graph latency constraint can be incorporated into the above formulation by supplementing the set of constraints by constraints on the scheduling times for invocations of the stop node. These constraints typically take the form of relative delays with respect to the scheduling times of invocations of the start node. The problem of scheduling under the constraint of graph latency is examined in a subsequent section.

In the foregoing the general case of scheduling a cyclic DDG was formulated as an integer programming problem. As explained there, the problem combines two known problems into one: one, of multiprocessor scheduling with deadlines, imposed by the loops in the execution graph, and the other of scheduling to minimize the cumulative cost. The former is a well-known NP-hard combinatorial optimization problem. The combined problem, thus, in general, is NP-hard to solve exactly [18], and so heuristics are often employed

to solve it near-optimally. Several heuristic techniques have been presented in the literature [43, 23].

Note that the above formulation does not account for the contribution to cost of the architecture from buffers. Usually, to certain extent, buffers can be traded-off against communication components. This trade-off is further expanded upon in Chapter 6.

In the following sections, some special cases of DDGs are considered and scheduling methods for them are proposed.

5.4 Acyclic DDGs

In the last section, it was commented that the resource minimization problem for an acyclic DDG is one of unconstrained optimization, if graph latency constraints are not imposed. This problem is studied in greater detail in this section. Of interest will be the possibility that the resource requirements in terms of number of communication and computation components can be determined before the start of the scheduling process, and the fact that the schedule can be computed in polynomial time.

For repetitive algorithms, which have been the subject of interest, the primary design objective is high throughput. The graph latency is often not of importance, and it can be traded in favor of lower architectural cost. This trade-off will be pursued here. It will be shown that an architecture can be constructed for an acyclic DDG at the lower-bound cost implied by the set of TREs.

If for an operation of type i, p_{ij}'s are the TRE-solution values of the nodes j representing the operation, then the implied lower bound on the number of components is: $\lceil \sum_j p_{ij} \rceil$. This resource requirement is henceforth

referred to as the *Minimum Resource Requirement* (MRR). In this section it will be shown that for acyclic DDGs, the MRR is simultaneously realizable for all types of operations.

Consider an operation of type i. Let $\sum_j \phi N_{ij}$ be the total number of operations of type i in the token count solution used to schedule the DDG, where ϕ is the multiple of the minimum token count solution discussed previously. Let L_{Sch} be the schedule cycle length. Given that a static schedule is to be computed and the total execution delay of the operation is D_i, each component of type i can execute and perform $\lfloor \frac{L_{Sch}}{D_i} \rfloor$ operations during a schedule cycle. It follows then, that at least

$$\left\lceil \frac{\sum_j \phi N_{ij}}{\lfloor \frac{L_{Sch}}{D_i} \rfloor} \right\rceil = \left\lceil \frac{\phi \sum_j N_{ij}}{\lfloor \frac{L_{Sch}}{D_i} \rfloor} \right\rceil$$

components of type i are needed in the architecture to guarantee that the number of operations performed by the components equal or exceed that in the execution graph.

Consider the case where L_{Sch} is an integer multiple of D_i. The number of components required is given by,

$$\left\lceil \frac{\phi \sum_j N_{ij}}{\frac{L_{Sch}}{D_i}} \right\rceil$$

By recalling Lemma 6, the above expression becomes:

$$\left\lceil \frac{\phi \sum_j N_s^{min} \rho_{si}}{\frac{L_{Sch}}{D_i}} \right\rceil = \left\lceil \frac{\sum_j N_s \rho_{si}}{\frac{L_{Sch}}{D_i}} \right\rceil = \left\lceil \frac{N_s \sum_j \rho_{si}}{\frac{L_{Sch}}{D_i}} \right\rceil$$

where ρ_{si}'s are path-product-ratios from the start node to the nodes representing operation i and N_s is the token count solution for the start node.

Notice, by definition, N_s and L_{Sch} are related by the following relationship: $L_{Sch} = N_s \cdot L_I$. And since the start node fires exactly once every L_I

time units, if D_s is its execution delay, $\frac{N_s}{L_{Sch}}$ can be substituted by $\frac{p_s}{D_s}$. The above expression therefore resolves to

$$\left\lceil \frac{p_s D_i \sum_j \rho_{si}}{D_s} \right\rceil = \left\lceil \sum_j \frac{p_s D_i \rho_{si}}{D_s} \right\rceil$$

Lemma 2 says that the above expression is the same as $\lceil \sum_j p_{ij} \rceil$, which is simply the sum of requirements predicted by the TREs of the DDG, the lower bound on the resource requirement.

If L_{Sch} is chosen to be a suitable common multiple of all D_i, then the resource requirement for every component will equal its MRR. Thus,

Lemma 19 *There exists an L_{Sch} for a given DDG with which resource requirement for each component can be made to equal its MRR.*

Before proceeding further, it is important to notice that the only requirement implied by the above lemma is that the number of operations that can be performed by components in the architecture equal or exceed the number of operations in the execution graph of the DDG. It is therefore possible that a smaller value of L_{Sch} than that stated by the above lemma may be found to satisfy this requirement. This is especially true if an MRR has a very small value less than 1.

Consider the following strategy:

The MRR is computed for each component, and choose a suitable L_{Sch} such that the resource requirement equals the MRR for each component. Then exact times or *slots* are preassigned at which each component will start executing. In so doing, care is taken to ensure that the components for a given operation cumulatively fire at least as many times as the number of operations

of that type in the execution graph. Following this a schedule is computed and an operation is assigned to a free component.

Observe here that an invocation of an operation may be postponed by any amount of time after all of its data dependencies have been satisfied. This may require inserting slack on some input edges of the invocation. From the discussion in Section 4.5 about delay-transformation for acyclic graphs, it is known that it is possible to insert this slack without violating the correctness of the computation, as long as graph latency constraints are not imposed.

So whenever the operation is ready to execute, the next available execution slot is found, to which to assign the operation. Since it is already known that the number of available slots are more than or equal to the number of operations in the execution graph, it is certain that an unused slot can always be found to assign a given invocation.

Thus the following important theorem can be concluded:

Theorem 7 *It is possible to schedule an acyclic DDG and realize the MRR for each type of operation by choosing an appropriate L_{Sch}, if graph latency constraints are relaxed.*

As part of this research, an MRR scheduler was implemented for acyclic DDGs. This scheduler will be discussed in detail in the next chapter.

The choice of L_{Sch} depends on the values of D_i, and in the worst case, if the D_i's happen to be mutually prime, the smallest value of L_{Sch} can be very large. The effect of this is that the control ROM which stores the schedule information becomes very large in size. This in itself is not particularly pernicious as long as the size is within acceptable limits, beyond which the

access delays of the ROM begin to affect the requirements on the clock period. Yet, again, the control ROM can itself be partitioned and interleaved to offset this effect.

5.5 DDG with a single outer loop

The last section dwelled on acyclic DDGs and concluded with an observation that an acyclic DDG without graph latency bounds can be scheduled to realize MRR's for all operations. Whenever a graph latency bound is placed on the schedule, however, the MRR might not be achievable; the actual number of components required might exceed that suggested by the MRR. Yet, a schedule with a minimum number of components is still desirable.

In this section, a special case of DDGs is discussed. The special case is of a DDG with a single feedback path going from the stop node back to the start node. Existence of such a feedback path effectively imposes a graph latency constraint on the DDG.

As noted earlier in Section 5.3, the problem of optimally scheduling an acyclic DDG with graph latency constraint is NP-hard [18]. There have been suggested in the literature, algorithms which, given the component-delay information, heuristically schedule acyclic graphs to obtain a near-optimal solutions in terms of numbers of components. Most approaches use some variation of ASAP or ALAP and critical path analysis based scheduling techniques. Most are based on local analysis such as: *scheduling the most critical operation first.* However, some scheduling techniques based on global analysis of hardware requirements have also been recently proposed. Paulin [47], for example, has proposed the *force-directed scheduling* technique to obtain a schedule for (*single-rate*) computation graphs expressed as a single *do-loop*, given the max-

imum execution delay for the loop. This technique uses the *steepest-descent* optimization heuristic and is found to give good results.

The above-mentioned heuristic techniques can be applied with certain necessary modifications to schedule multi-rate DDGs. Notwithstanding these techniques, an iterative heuristic algorithm is presented here, and the proof of its correctness is given. In the section following this, there will be an opportunity to consider other special cases of cyclic DDGs, and the iterative algorithm about to be presented will be applicable to those cases as well. This iterative algorithm is based on a resource-constraint scheduler, the availability of which is assumed.

5.5.1 An iterative heuristic algorithm

Throughout the discussion so far, the *start* node of a DDG has been defined as being one without any input dependency and the *stop* node as one without any output dependency. For this section these definitions will be relaxed and the start and the stop nodes will be allowed to have one dependency between them representing the feedback path. It will be further required that all initial tokens in the graph be specified for this dependency and none other. Furthermore, the number of tokens so available are restricted to be an integer multiple of the number absorbed by the start node invocations, the number of invocations being equal in number to the minimum token count solution for the start node. Let this multiple be denoted by $\mathbf{m}$. Note that, in the case of a single rate DDG, $\mathbf{m}$ will, in fact, be the number of tokens available.

As before, it will be assumed that the input latency is the period of execution of the start node.

Graph latency for an acyclic DDG

In the case of single rate graphs, it is intuitive what the graph latency implies: it is the time delay between the invocation of the start node and the completion of execution of the stop node. However, in the case of multi-rate DDG, there are, in general, multiple invocations of the start and stop nodes. It is possible to imply by graph latency of an acyclic graph, the time duration between the first invocation of the start node and the last invocation of the stop node, the first and last here are with reference to the token count solution of the DDG. It is then possible to state a graph latency bound as a bound on this time duration without imposing any restriction on the first and intermediate invocations of the stop node. The times of these intermediate invocations are dependent on the DDG and its token markings and cannot be defined uniquely *a priori*. However, as will be seen next, there is a natural implication for the graph latency in the case of a repetitive computation.

The semantics for the start node have already been defined: it fires every input latency. If N_s^{min} is the number of invocations of the start node in the minimum token count solution for the DDG, then $N_s^{min} \cdot L_I$ is the schedule latency, L_S.

The semantics of the bounds on graph latency have to be consistent with the definitions of input and schedule latencies. In the case of the class of DDGs under consideration there is a natural interpretation for the graph latency constraint: *the start node of the DDG must always be schedulable every input latency.*

A few things implied by this definition of the graph latency bound can be observed. Firstly, it implies that the time delay from the beginning of execution of the first invocation of the start node to the end of the first

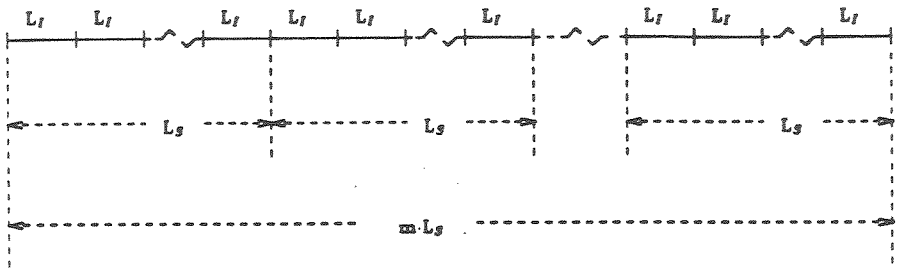


Figure 5.3: Relationships between L_I, L_S and $m \cdot L_S$.

invocation of the stop node must be $m \cdot L_s$. Similarly, the time delay between the start of the last invocation of the start node and the end of the last invocation of the stop node must also be $m \cdot L_s$. The delay bounds on other invocations of the stop node are DDG-dependent. These relationships are illustrated in Figure 5.3.

Note that the above definition is consistent with the definitions for the familiar single rate DDGs. The requirements reduce to the familiar requirement for the single rate DDG: For example, if $m = 1$ and the DDG is single rate, the graph latency requirement reduces to the graph latency being bounded by the input latency.

The iterative algorithm

In the preceding the semantics for graph latency for multi-rate DDGs were specified. An iterative heuristic algorithm to obtain a schedule under graph latency constraint starting with a scheduling algorithm that schedules

under resource constraints can now be specified. The following is assumed about the resource-constraint scheduler:

Assumption: The resource-constraint scheduler will not insert any slack for an operation, if a component is available to execute it.

Algorithm 2

1. *Apply resource-constraint scheduling algorithm and obtain an MRR schedule.*

2. Loop 1: *Do the following until the graph latency conditions stated above are met:*

   - Loop 2: *Do the following for k=1...* $\mathbf{n}_{comp}$*:*
     - *Compute cumulative slack for each type of operation.* (The cumulative slack is computed by adding together slack introduced for every operation of the given type.)
     - *Order the types by decreasing cumulative slack.*
     - Loop 3: *For each combination of k component types, do the following:*
       * *Temporarily increment by 1 the number of available components for these types, if their number is less than the number of operations of the same type in the execution graph.*
       * *Apply resource-constraint scheduling algorithm.*
       * *Verify the graph latency conditions stated above. If the graph latency has decreased, fix the number of components to the new values and break out of this loop.*

- *If the latency has decreased, break out of this loop. If it has not, then increment k. If* $k \geq \mathbf{n}_{comp}$, *increment the number of available components for all types if their number is less than the number of operations of the same type in the execution graph.*

- *If the graph latency conditions are met, terminate, otherwise continue.*

The above algorithm starts with an MRR schedule, and then systematically increments the number of each type of component up to the corresponding number of operations appearing in the execution graph. At each step, it computes the schedule and checks the graph latency requirements.

It will now be proved that the above algorithm computes a schedule that satisfies the graph latency requirements. The following lemma is proved first.

Lemma 20 *Graph latency is greater than or equal to the length of the critical path in the execution graph.*

Proof: By definition of the critical path, graph latency less that the length of the critical path is not feasible. □

Lemma 21 *The algorithm will increment the number of components of each type to the maximum value of the number of operations of the same type.*

Proof: Loop 3 increments temporarily the number of components and if this reduces the graph latency these new values are fixed as starting values for the next iteration. If the graph latency does not decrease, the last step of loop 2 increments the number of components of every type until they are equal to the number of corresponding operations. □

Theorem 8 *Algorithm 2 terminates and finds a solution.*

Proof: From the above lemma it follows that the algorithm will increment the numbers of component types until they equal the number of operations of respective type appearing in the execution graph. At these numbers of components, the resource-constrained scheduler will find a distinct component for each operation and will not insert any slack at all and produce an ASAP solution. The graph latency in this case will be equal to the length of the critical path of the execution graph, which is less than or equal to the graph latency requirements that can be imposed on the graph. Thus the algorithm must satisfy the graph latency requirement and, therefore, must terminate. □

The above algorithm assumes an existence of a resource-constrained scheduler. The quality of the graph latency constrained schedule clearly depends on it. A better resource-constrained scheduler may be able to find a schedule, less expensive in terms of required numbers of components.

The above algorithm was developed with the heuristic scheduler to be presented in the next chapter in mind. That heuristic scheduler could be used as a resource-constrained scheduler for the above algorithm. However, the scheduler creates predefined slots during which the hardware components are assumed to be available for scheduling. This scheme might introduce *dead* times when the components of a given type are not available to the scheduling process. Such a scheduler, in general, will not meet the lowest graph latency constraints, such as ones defined by the critical path within the acyclic DDG. If, however, it is assumed that either the delays of the components are such that there are no dead times, or the graph latency is sufficiently large to cover any such dead times that might accumulate, the heuristic scheduler may still be used.

5.6 Scheduling of multi-rate DDGs with independent and nested loops

The complexity of scheduling is a major concern in synthesis. As was seen earlier, the problem is NP-hard in the most general case of cyclic graphs. The exhaustive search techniques which methodically search all possible solutions for the best one are very expensive even for a DDG with a few tens of nodes so as to be prohibitive. And so, there is an interest in approximate solutions and ways to expedite the scheduling process. If cyclic DDGs satisfy certain structural conditions, a *divide-and-conquer* heuristic can be applied to reduce the complexity. This can be achieved by breaking up the overall scheduling problem to a number of smaller sub-problems. Albeit, the optimality of the solution is traded off to achieve it more quickly.

In this section, a structural condition will be enunciated for the loops of a cyclic DDG. This condition will allow a "correct" application of the divide-and-conquer heuristic. Throughout this section it will be assumed that all loops of the DDG are non-deadlocking.

5.6.1 DDG with independent loops

In this sub-section DDGs in which the loops are completely independent are examined. The independence of loops is defined as follows:

Definition 19 *Two loops, l_1 and l_2, are considered to be independent if the node-sets $\{n_i\}_{l_1}$ and $\{n_i\}_{l_2}$ corresponding to the two loops respectively are such that there are no common nodes, i.e., $\{n_i\}_{l_1} \cap \{n_i\}_{l_2} = \phi$.*

Figure 5.4 displays an example of a DDG with independent loops. The loop *start* nodes of the loops are also shown in the figure; these are defined

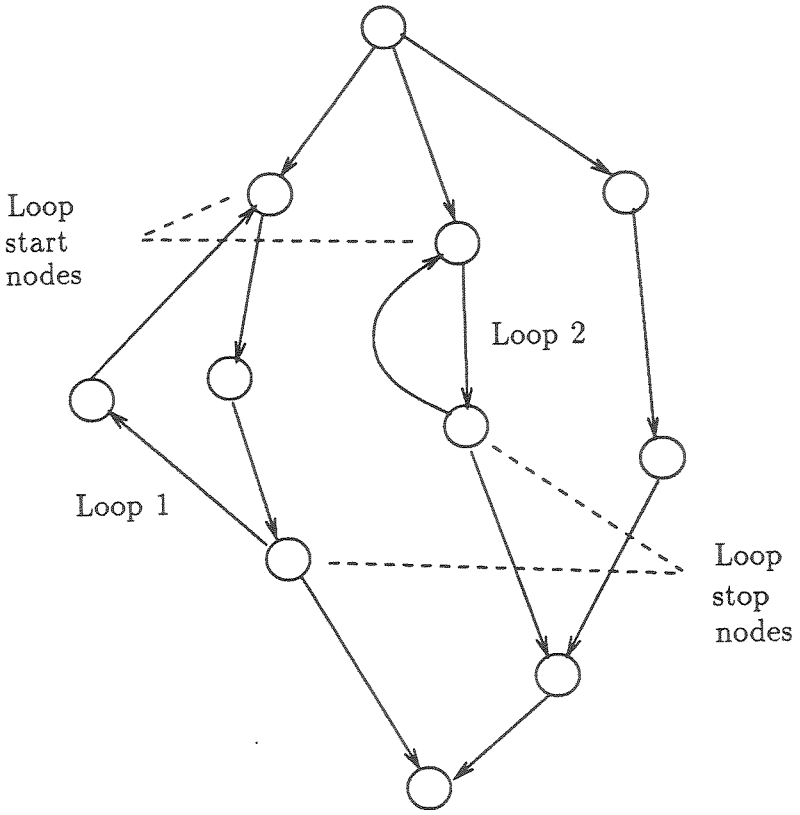


Figure 5.4: A DDG with independent loops.

to be the first nodes of the loops to be encountered during a depth-first traversal of the DDG starting with its start node.

For the case where all loops are independent, the following divide and conquer heuristic can be used:

The cyclic portion of the DDG is scheduled first, one directed loop at a time. Once all the directed loops have been scheduled, the non-cyclic portions can be scheduled assuming the schedule computed for the nodes that belong to the directed loops. The overall algorithm looks as follows:

Algorithm 3 Heuristic algorithm to schedule independent loops.

1. *Obtain a spanning tree of the DDG.*

2. *Obtain a set of links that form directed loops when added to the spanning tree obtained in step 1.*

3. *For each link obtained in step 2, schedule the corresponding directed loop independently of the rest of the graph.*

4. *Schedule the remaining acyclic portion of the graph without modifying the schedules obtained in step 3.*

Theorem 9 *Algorithm 3 correctly schedules a DDG with independent loops.*

Proof: The fact that each independent loop can be independently scheduled without introducing any scheduling errors is guaranteed by Lemma 15. In step 4, the acyclic portion of the DDG is scheduled, which results in the introduction of slack which merely postpones entirely the loop schedules without affecting the relative times of firings of nodes within the loops. □

As mentioned above, the scheduling of individual loops will be treated in more depth later.

5.6.2 DDGs with nested loops

This section contains analysis of DDGs with *nested directed loops*, related by the concept of *uniform criticality* (to be introduced in this section), and with initial tokens only on the feedback edges. These types of DDGs are encountered while representing in graphical form nested loops expressed in programming languages.

For the set of nested loops satisfying the condition of uniform criticality, a divide-and-conquer algorithm is suggested to heuristically schedule them.

The heuristic is aimed at reducing the complexity of the scheduling process. The major effort here will be to prove the correctness of the heuristic.

As in the previous section, the approach will be to schedule the loops first and then to schedule the acyclic portion of the DDG. Here, the focus will only be on the scheduling of loops and not the acyclic portion.

For the purpose of the analysis to be carried out in this sub-section, it will be assumed that for every loop, exactly one edge carries the initial tokens. The edge carrying the initial tokens is called the *feedback edge*. The node at the head of the feedback edge is the loop *start node* and the one at the tail is the loop *stop node*. The remaining set of edges of the loop are called the *feed-forward edges*, and form the *feed-forward path* of the loop.

Only a set of directed loops which satisfy the following criterion will be considered.

Definition 20 *Two loops, l_1 and l_2, are considered nested if*

1. *they do not share a feedback edge,*
2. *the node-sets $\{n_i\}_{l_1}$ and $\{n_i\}_{l_2}$ corresponding to the two loops respectively are related by subset relationship, i.e., either $\{n_i\}_{l_1} \subseteq \{n_i\}_{l_2}$ if l_2 is nested within l_1, or $\{n_i\}_{l_2} \subseteq \{n_i\}_{l_1}$ if l_1 is nested within l_2, and*
3. *they share a path defined over the node-set common to the two loops, i.e., over $\{n_i\}_{l_1} \cap \{n_i\}_{l_2}$.*

Lemma 22 *If loop l_2 is nested within loop l_1, then l_1 contains all edges of l_2 except its feedback edge, i.e., l_1 contains l_2's feed-forward path, which is also a part of its own feed-forward path.*

Proof: Since l_2 is nested within l_1, $\{n_i\}_{l_1} \subseteq \{n_i\}_{l_2}$. Thus $\{n_i\}_{l_1} \cap \{n_i\}_{l_2} = \{n_i\}_{l_2}$. The two loops therefore share a path defined over $\{n_i\}_{l_2}$, the nodes of l_2. Since l_2 has two parts, the feed-forward path defined over its nodes and a feedback edge and since the two loops l_1 and l_2 do not share their feedback edges the path shared must be the feed-forward path of l_2.

Furthermore, since l_1 cannot have additional paths defined over the nodes of l_2, it must be a part of l_1's own feed-forward path. □

It is easy to generalize this via the following lemma.

Lemma 23 *Let $\{l_1 \cdots l_l\}$ be a set of nested loops such that loop l_i nests all loops l_{i+1} through l_l. Then l_i contains feed-forward paths of loops l_{i+1} through l_l and which are also parts of its feed-forward path.*

Definition 21 *A loop is contained within a DDG or a sub-DDG, if it contains all of loop's nodes and edges.*

Lemma 24 *If loop l_2 is nested within loop l_1, then (l_1 ∪ feedback edge of l_2) contains l_2.*

Proof: Since l_2 is nested within loop l_1, $\{n_i\}_{l_1} \subseteq \{n_i\}_{l_2}$ and it has been shown above that l_1 contains all the feed-forward edges of l_2. □

This can be generalized to conclude,

Lemma 25 *Let $\{l_1 \cdots l_l\}$ be a set of nested loops such that loop l_i nests all loops l_{i+1} through l_l. Then (l_i ∪ {feedback edges of $l_{i+1} \cdots l_l$}) contains loops l_{i+1} through l_l.*

Also,

Lemma 26 *If loop l_2 is nested within loop l_1, then there exists a directed path from the start node of l_1 to the start node of l_2 which does not involve any other node in l_2.*

Proof: If the start node of l_1 is the same as the start node of l_2, then the result is trivialy true. But if it is not then, since loop l_2 is nested within loop l_1, l_1 contains all the nodes of l_2 including its start node. Thus it is reachable by a directed path from the start node of l_1 using its feed-forward path. Assume that this path includes a node of l_2 other than its start node. This implies that the directed path reaches this node before it reaches the start node. But by definition of the start node this must then also include the feedback edge of l_2, which is not a part of l_1. □

Using analogous arguments the following could be proved:

Lemma 27 *If loop l_2 is nested within loop l_1, then there exists a directed path from the stop node of l_2 to the stop node of l_1 which does not involve any other node in l_2.*

Figure 5.5 contains an example of three nested loops, l_1, l_2 and l_3. In drawing these loop the following convention is used: the outermost loop is labeled l_1, the one immediately inside it is labeled l_2, and so on, the subscript implying the nesting level. The figure also shows the initial tokens, I_1, I_2, and I_3, assigned to the three feedback edges. According to Lemma 17, the number of the initial tokens must equal or exceed that necessary to enable a complete execution of the minimum solution for the loop. The dotted lines indicate paths formed over zero or more nodes using only the feed-forward edges.

Lemma 28 *If loop l_2 is nested within loop l_1, then a token count solution for l_1 contains a multiple of the minimum token count solution for l_2.*

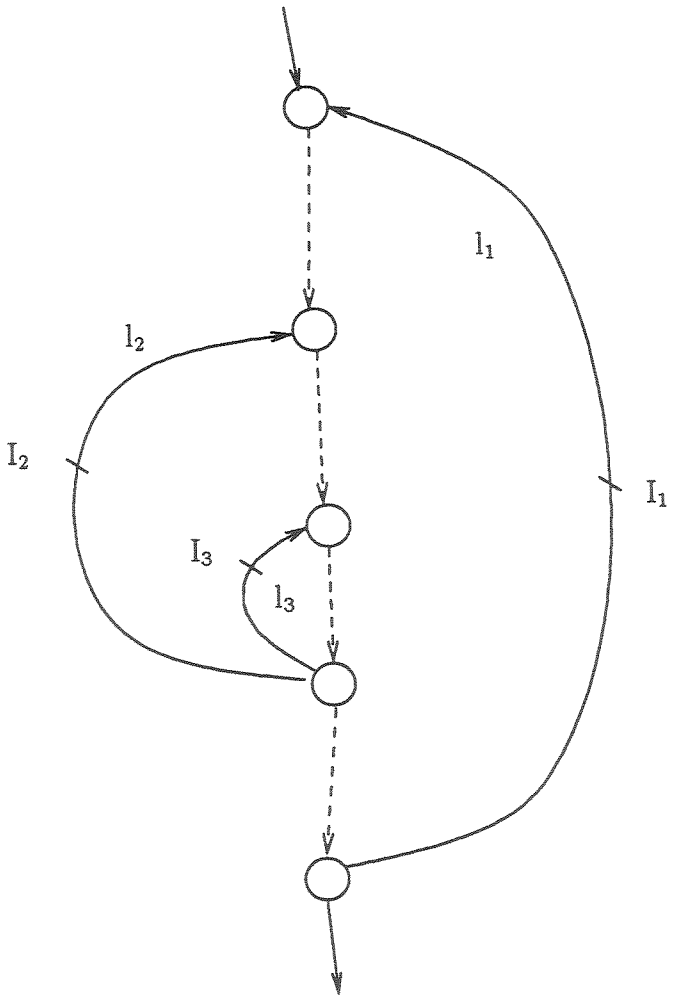


Figure 5.5: A set of nested loops.

Proof: By Lemma 22, l_1 contains all edges of l_2 except one. Thus a token count solution of l_1 satisfies the token count equations of all the edges of l_2 except one. But by Lemma 7 the token count solution of a loop is the same as that of its edge-set shy by any one edge. Combining this with Lemma 9, the required conclusion is reached. □

This can easily be extended to state that,

Lemma 29 *Let $\{l_1 \cdots l_l\}$ be a set of nested loops such that loop l_i nests all loops l_{i+1} through l_l. Then a token count solution of l_i contains multiples of the minimum token count solutions for loops l_{i+1} through l_l.*

To schedule these loops, *appropriate* integer multiples of their individual minimum token count solutions are used, such that the number of invocations of each of their nodes is equal to that obtained from the token count solution of the overall DDG. Because of Lemma 9, these integer multiples are uniquely determined by the token count solution chosen for the DDG. This fact and the last lemma leads to the conclusion:

Theorem 10 *If $\{l_1 \cdots l_l\}$ are the set of nested loops such that loop l_i nests all loops l_{i+1} through l_l, then the appropriate token count solution for l_i contains appropriate token count solutions of loops l_{i+1} through l_l.*

This theorem gives the confidence that when a loop is scheduled using the part of the solution that refers only to its nodes and edges, the loops nested within it are also scheduled with the correct solutions. In the subsequent, this fact will be tacitly assumed. Above, the term "appropriate solution" was used to refer to the solution that would result from Lemma 9. Henceforth, unless otherwise specified, a solution will imply the "appropriate solution".

An execution sub-graph is an execution graph that is obtained from the edges of a sub-DDG but by using the numbers of invocations of its nodes to equal those specified by the token count solution of the entire DDG.

If loop l_i nests all loops l_{i+1} through l_l, then the execution graph corresponding to the solution for the sub-DDG $\{l_i \cup \{$feedback edges of loops $l_{i+1} \cdots l_l\}\}$ are considered. The feedback edges of loops l_i through l_l translate to edges in the execution graph where they may become feed-forward edges or feedback edges.

As seen in Section 5.2, the feedback edges in the execution graph put upper bounds on the time of execution of the operations in the loops of which they are parts. These timing constraints must be satisfied for a feasible schedule.

Uniform Criticality

The relation of *uniform criticality* is now defined.

In Section 5.2 a loop l in the execution graph is called *critical* if for it $\nu_l \cdot N_s \cdot L_{I_{min}} - D_l$, where ν_l is the number of tokens in the loop, D_l is the delay in the loop, N_s is the number of invocations of the start node of the DDG and $L_{I_{min}}$ is the minimum input latency, is the smallest. A similar quantity, $\nu_l \cdot N_s \cdot L_I - D_l$, is defined for a given input latency L_I and is denoted by σ_l. σ_l represents the maximum amount of slack that may be introduced in the schedule of loop l without violating the timing constraints.

A given loop l_i in a DDG is transformed into a number of loops, l_{ij}, in the execution graph, for each of which $\sigma_{l_{ij}}$ can be determined. The maximum and minimum values of $\sigma_{l_{ij}}$ for all loops corresponding to l_i can be determined and are denoted as $\sigma_{l_i max}$ and $\sigma_{l_i min}$.

Definition 22 *A loop l_i is uniformly more critical than another loop l_j if, and only if,*

- *if ($l_j \cup$ feedback edge of l_i) contains l_i, then if there are paths in the execution graph of the DDG from two distinct invocations of the start node of l_i to an invocation of any node in l_i such that, one path traverses only the nodes of l_i and the other traverses nodes of l_j not common to the two loops, the delay associated with the first path is more than or equal to that associated with the second.*

- $\sigma_{l_i max} \leq \sigma_{l_j min}$.

This relationship is denoted by $l_i \lhd l_j$.

An example of the first condition above is shown in Figure 5.6.

In a multi-rate DDG, a path between two nodes in the DDG gets translated to multiple paths of unequal lengths between the corresponding sets of nodes in the execution graph, and it is necessary to ascertain that all these paths satisfy the criticality criterion. As will be found, the relation of uniform criticality is sufficient to make the scheduling of nested loops independent of each other.

Notice that the above definition expects a total ordering on the candidate loops. *It is important to bear in mind that the subsequent analysis is valid for only those nested loops that can be ordered by $\lhd$.*

Some notation is introduced now:

Notation: Edges are denoted by the letter e. When the edge refers to the DDG, it will be denoted by e^{DDG}, and if it refers to the execution graph,

it will be denoted by e^{EG}. If it is a feed-forward edge, it will carry a subscript f; if it is a feedback edge, it will carry a subscript b. The loop it belongs to will form its second subscript. If both feed-forward and feedback edges are implied, only the subscript indicating the loop will be used.

Some preliminary results are proved first. These will later lead to a heuristic algorithm to schedule multiple nested loops.

Lemma 30 *If loop l_1 nests loop l_2, then every directed path in the execution graph of the DDG starting with an invocation of the start node of l_2 and formed by only the edges from l_2, is part of a path starting with an invocation of the start node of l_1 in the execution sub-graph defined over (l_1 ∪ feedback edge of l_2).*

Proof: (l_1 ∪ feedback edge of l_2) contains l_2. Therefore a path formed by only the edges of l_2 is also a path in the execution sub-graph defined over (l_1 ∪ feedback edge of l_2). Moreover, as has been shown in the foregoing that there is a directed path in l_1 from its start node to the start node of l_2 which does not encounter any other nodes of l_2. □

Lemma 31 *Every directed path in the execution sub-graph of loop is also a part of some directed loop in the subgraph.*

Proof: If there is a path which is not part of a loop, then the terminal invocation in the path must not have any output edges corresponding to some edge belonging to the loop, which is not the case. □

From the above two lemmas it can be concluded that,

Lemma 32 *If loop l_1 nests loop l_2, then every directed path in the execution graph of the DDG starting with an invocation of the start node of l_2, formed*

by only the $e^{EG}_{l_2}$ edges, and ending with an invocation of the stop node of l_2, is part of a loop in the execution sub-graph of (l_1 $\cup$ feedback edge of l_2), defined using an e^{EG}_{b,l_1} edge.

Consider a loop formed in the execution sub-graph of l_2. If the e^{EG}_{b,l_2} of this loop is deleted, it forms a directed path using only the edges $e^{EG}_{l_2}$, including of course some corresponding to e^{DDG}_{b,l_2}. The above lemma states that such a path is also a part of a loop defined using a e^{EG}_{b,l_1}.

Theorem 11 *If loop l_1 nests loop l_2, and if $l_1 \lhd l_2$, then it is possible to schedule correctly the execution sub-graph for l_1 by considering only the timing constraints imposed by loop created by the edges e^{EG}_{b,l_1}, and opening loops by deleting all the e^{EG}_{b,l_2} edges.*

Proof: It was shown above that every path created by opening a loop within the execution sub-graph of l_2 is part of a loop in the execution sub-graph of (l_1 $\cup$ feedback edge of l_2) using a e^{EG}_{b,l_1} edge. But since $l_1 \lhd l_2$, the latter loop has less available slack than the former. Thus scheduling of the latter without consideration to the timing constraints of the former will not introduce a slack that is more than that acceptable for the former. Therefore, satisfying the timing constraint of the latter loop implies satisfaction of that of the former loop. □

Theorem 12 *If loop l_2 nests loop l_1, and if $l_1 \overset{<}{uc} l_2$, then it is possible to schedule correctly the execution sub-graph for l_1 by considering only the timing constraints imposed by loop created in it by the edges e^{EG}_{b,l_1}.*

Proof: All invocations of all the nodes of l_1 except its start node can be scheduled independently of invocations of all nodes that don't belong to l_1, if the

invocations of its start node can be shown to be independent of them So consider the scheduling of an invocation of the start node of l_1.

Each invocation of the start node receives tokens on two dependencies, one, the e^{DDG}_{b,l_1} edges, and the other, the e^{DDG}_{f,l_2} edges. By Lemma 18, initially, a certain number of invocations of the start node must be enabled for scheduling. These invocations can be scheduled immediately. Consider the first invocation of the start node of l_1 which cannot be so scheduled. As before it has two types of dependencies. The one corresponding to the e^{DDG}_{b,l_1}, which come directly from invocations of the stop node of l_1, and the other which also involves edges and nodes in l_2 that do not belong to l_1. If there are initial tokens along the latter (implying that the start nodes of l_1 and l_2 are identical), then the invocation of the start node of l_2 being considered can be scheduled as soon as the invocations of the stop nodes of l_2 produce sufficient number of tokens satisfying the former — which is again independent of invocations of the nodes not belonging to l_1. However, if it does not have the presumed initial tokens, then it must depend on invocations of its parent node belonging to l_2, which in turn must depend on some non-initial tokens generated by invocations of their parent nodes, and so on, up to the start node of l_2. These in their turn must depend on the non-initial tokens they receive from the feedback dependency e^{DDG}_{b,l_2} and therefore, the invocations of l_1's stop nodes and therefore some earlier invocations of the stop nodes, and hence also the start nodes, of l_1. So now there exist two sets of paths, as shown in Figure 5.6, from the invocations of the start node of l_1 to another invocation of its start node. The first set traverses only the edges belonging to l_1, $e^{EG}_{l_1}$, and the other also traverses edges $e^{EG}_{l_2}$ which are not part of $e^{EG}_{l_1}$. However, by the definition of uniform criticality, if loop l_2 nests loop l_1, and if $l_1 \lhd l_2$, the delays associated with the first set of paths are more than or equal to those associated with the second. Thus again, the invocation under

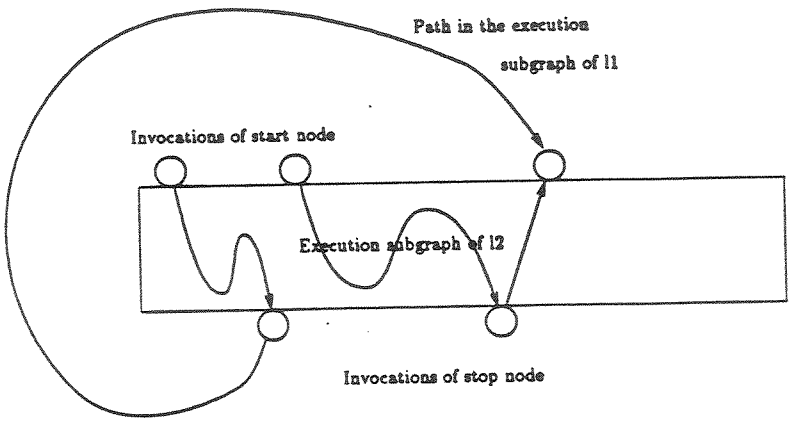


Figure 5.6: Two paths from invocations of the loop start node.

consideration can be scheduled as soon as the invocations of the stop node of l_1 corresponding to the former set has completed execution, and therefore can be made independent of execution times of invocations of nodes that don't belong to l_1.

Notice also that, by the virtue of the fact that $l_1 \lhd l_2$, the slack introduced in any loop of l_1 does not violate the timing constraint of a loop in l_2. □

The above two theorems are combined in the following single theorem:

Theorem 13 *If l_1 and l_2 is a pair of nested loops, and if $l_1 \lhd l_2$, then l_1 can be scheduled independently of l_2 without violating l_2's timing constraints.*

This theorem can be obviously generalized to state,

Theorem 14 *If $\{l_1 \cdots l_l\}$ is a set of nested loops such that loop l_i nests all loops l_{i+1} through l_l, and if l_i is more uniformly critical than all other loops, then l_i can be scheduled independently of other loops and without violating the timing constraints of the other loops.*

Now, a divide and conquer algorithm is proposed to schedule a set of uniformly critical nested loops:

Algorithm 4

1. *Order the set of nested loops $\{l_1 \cdots l_l\}$, where l_i nests all loops l_{i+1} through l_l, according to the criterion of uniform criticality.*
2. *Schedule the most critical loop l_i independent of all other loops, using if any, previously computed schedule of a nested loop.*
3. *Delete from the set, loops l_i through l_l.*
4. *Repeat steps 1 through 3 until l_1 is scheduled.*

Theorem 15 *Algorithm 4 schedules the loops correctly.*

Proof: Follows directly from Theorem 14.

The concepts involved in the above are illustrated with the help of the following example.

Consider the computation:

$$y_n = x + y_{n-1} + 0.5y_{n-2} + 0.5y_{n-3}$$

Figure 5.7a shows the DDG for the computation drawn in the tradition of a signal flow graph. Figure 5.7b shows a transformed DDG in which

the loops are shown as being nested. The coefficients were chosen in such a way that only one multiplication is necessary. The simplified DDG is shown in Figure 5.7c. It is easy to see that loops in Figure 5.7c are related by the criterion of uniform criticality. Loop b is the most critical and can, therefore, be scheduled independently of the other two. Since loop a is contained within loop b, it need not be scheduled separately. Loop c is scheduled last without modifying the schedule of loop b. Because of uniform criticality, it is certain that the final schedule satisfies the timing constraints of all three loops.

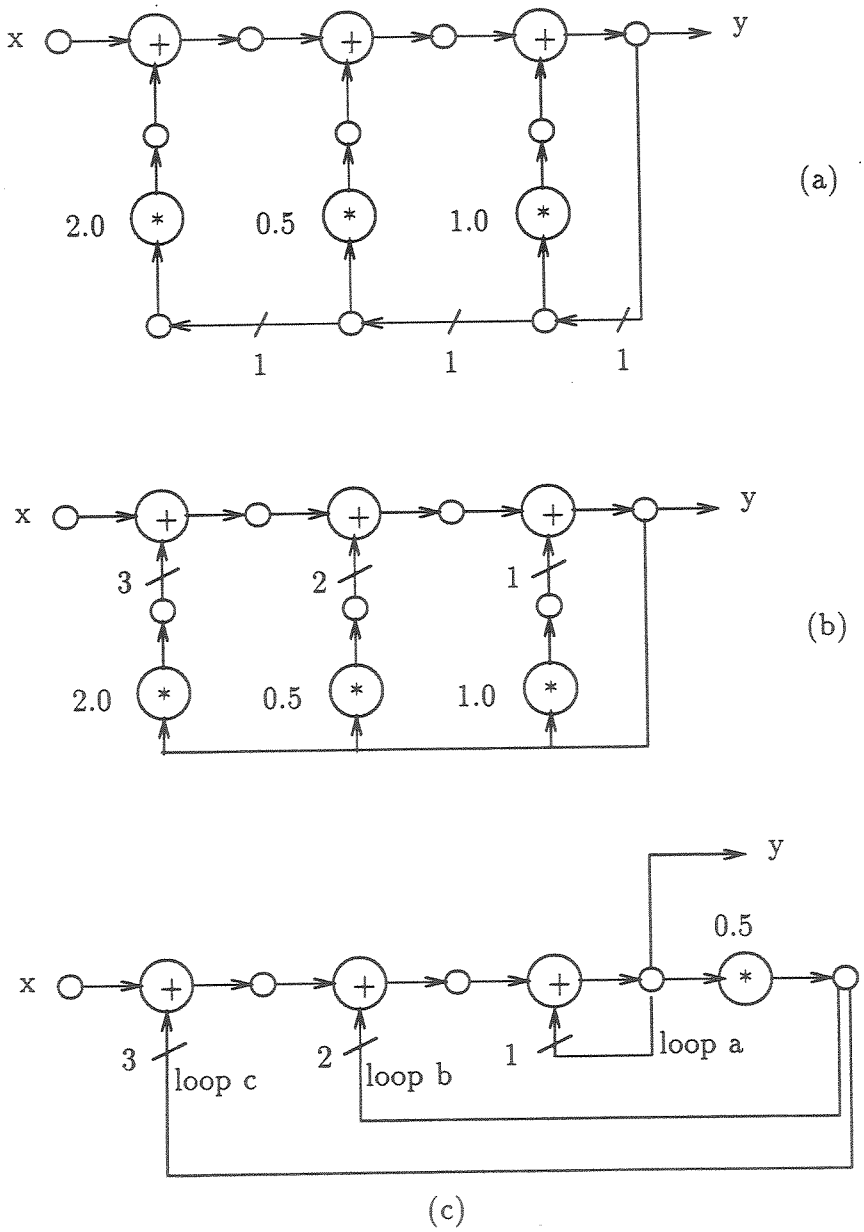


Figure 5.7: DDGs for the example illustrating uniform-criticality.

Chapter 6

Implementation of a heuristic scheduler

In the previous chapter, theoretical considerations were given to the problems of pipeline synthesis. In this chapter an implementation of a heuristic multi-rate scheduler, primarily designed to obtain an MRR schedule for an acyclic DDG without graph latency bounds, is described. As before, MRR (minimum resource requirement) is assumed to imply minimum number of computation and communication components. Number of registers and drivers are not incorporated directly in the minimization criterion.

The scheduler has been extended to allow an interactive implementation of the iterative algorithm for a DDG with a single outer loop, as described in the last chapter. A graphical front-end called DDGTool, used to allow specification of an annotated DDG, is described herein as well. Along with the DDGTool, the scheduler forms a complete pipeline synthesis system for multi-rate repetitive algorithms.

The heuristics chosen for the scheduler facilitate the independence of the two parts of the architectural design process: 1) time-scheduling and 2) assignment of operations to components. Performance of these heuristics will be examined with the help of example DDGs and strategies for improvements will be discussed. A synthesis methodology is enunciated first.

6.1 Synthesis methodology

The foundations for the methodology about to be outlined have already been laid in the previous chapters. Here those concepts are brought together and formulate a stepwise procedure to translate DDGs to hardware architectures. In later sections individual components of the methodology are discussed.

Steps of the methodology The high level synthesis methodology broadly consists of four steps.

1. **Express algorithms as DDGs.** The first step in the synthesis process is to express a computation in the form of a DDG.

2. **Annotate the DDG.** The user makes a choice about which operation will be executed on which type of hardware component and thereby binds the operations to component types. This choice determines the delays of operations, and their firing disciplines, used later during the analysis of the DDG. Token markings are also specified in this step.

3. **Choose input latency and/or schedule cycle length.** As seen in Chapters 4 and 5, these two define the free variables for the TCEs and TREs for the DDG respectively. These two variables need to be bound before an architectural solution could be found.

4. **Design the solution architecture.** With the completion of the first three steps, the *automated* synthesis procedure can take over. The synthesis procedure for acyclic DDGs is comprised of the following sub-steps:

   (a) Solve TREs.

(b) Solve TCEs.

(c) Compute the number of hardware components necessary.

(d) Construct the execution graph.

(e) Schedule the operations of the execution graph.

(f) Assign operations to components to minimize connectivity.

The first five sub-steps of the last step in earlier chapters have already been encountered. Of particular interest in this chapter is the specific implementation of the fifth and the sixth sub-steps. The implementation carries the synthesis process to the point of defining the start times of invocations and assigning them to instances of hardware components. Given that the exact sequence of control signals to be applied to the individual hardware components is known beforehand, the complete encoding of a control ROM can be mechanically obtained thereafter. To define the connectivity of the architecture, the implementation computes port-to-bus connectivity for each component.

The implementation of the above steps of the methodology are divided into two separate software modules. The last two steps are part of the multirate scheduler, whereas the first two are part of *DDGTool*, which is discussed next.

6.2 DDGTool

DDGTool is a graphical front-end to facilitate interactive drawing of DDGs and producing graph specification files that can later be read by the scheduler. It is based on *Guide*, a proprietary graph drawing software package

from *Scientific and Enginnering Software Inc.*, and is implemented[1] to execute within the *Sunviews* window based programming environment available on the Sun Microsystems Inc. workstations.

In this section, various features and capabilities of DDGTool will be briefly described. For detailed information, the reader is referred to the DDG-Tool User Manual [15].

6.2.1 Graphical capabilities

The graphical capabilities of DDGTool are referred to as *tools*, and the current capability being invoked is indicated by the corresponding unique cursor. Specifically, there are seven distinct tools and the following are their respective functions:

1. **Create:** This tool is used to draw nodes and edges on the screen, and is the default tool. A DDG is drawn on a *canvas*, only a part of which is displayed within the window on the screen at any given time.

2. **Travel:** DDGTool has a *heirarchical* drawing capability. A special type of node symbol, sub-DDG node (with concentric circles), signifies a sub-DDG. When such a sub-DDG is to be displayed or drawn, the *Travel tool* is used on the selected sub-DDG node. The sub-DDG denoted by a sub-DDG node is treated like a *macro* definition and is expanded at the time of producing output files.

[1] DDGTool was conceived and specified by the author and implemented by Mr. Cheng-Liang Lin and Mr. Mukund Belliappa of the Department of Computer Science, University of Texas at Austin.

3. **Erase:** This tool is used to delete graphical symbols of nodes and edges from a DDG.

4. **Open:** This tool is used to annotate the DDG by specifying properties associated with nodes and edges in the graph. The tool opens appropriate *forms* for the user to fill in the necessary information.

5. **Move:** As the name suggests, the *Move* tool, is used to move symbols around the screen.

6. **Square:** This tool is used to *beautify* the DDG by making its edges rectilinear.

7. **Copy:** The *Copy tool* is used to duplicate nodes and sub-DDG nodes.

In addition to the above graphical capabilities, the DDGTool allows the user to scroll over the canvas, to print the screen, and to access other graph files. These functions are invoked via *buttons* displayed in the window header banner.

6.2.2 Specification of DDGs using DDGTool

DDGs are specified by drawing them on the screen using icons provided by the DDGTool. DDGTool provides icons for nodes, and allows drawing directed arcs connecting the ports of these nodes.

DDGTool has distinct icons for computation, communication, and sub-DDG nodes. There are two more icons that are displayed but not available for the user to draw; these are the *Enter* and *Exit* nodes of a sub-DDG, which are automatically drawn by the tool when a "travel" is made to a sub-DDG. The node icons are shown in Figure 6.2. In the figure, the icons in the top

row represent communication operations, the ones in the middle row are used to represent computational operations, and the icon with concentric circles represents a sub-DDG.

Each node icon has up to two ports, an input and an output. Icons representing start and stop nodes of a DDG have only output or input ports, respectively. Each port of an icon is capable of representing multiple ports for an operation and the corresponding component, however. This binding of a port of an icon to a port of a component is done via edges, as will be explained below.

Specification of operations

The specification of operations is divided into two parts. First, all the operations used by the DDG-encoded algorithm are *declared* globally by opening the form corresponding to the *Definitions* button (see Figure 6.1). Here, declarations are made about the mnemonics used to refer to the operations, and their respective delays and *default* firing disciplines. The mnemonics declared here are visible for assignment to the nodes. As indicated in Chapter 3, the default firing discipline of all computational operations is AND, but that of the communication operation is chosen to be SR. The delays are positive integers, denoting multiples of the period of the global clock.

Second part of declaration of operations deals with individual nodes. For each node, information is provided indicating its label, type of operation — referring to one of the mnemonics declared earlier, and its *actual* firing discipline, which for all nodes except the communication, is the same as the common default for their respective operation, the AND. This specification is provided using a form invoked via the Open-tool. (See Figure 6.3.)

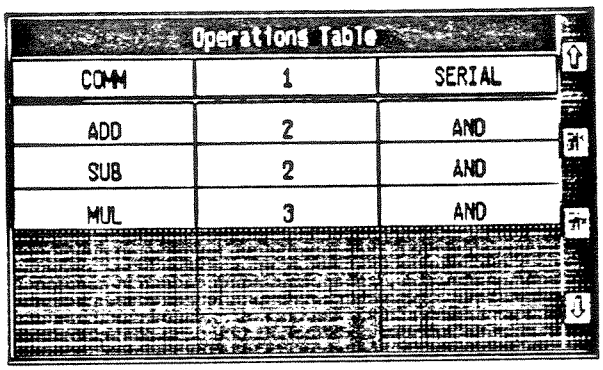


Figure 6.1: The *Definitions* form in DDGTool.

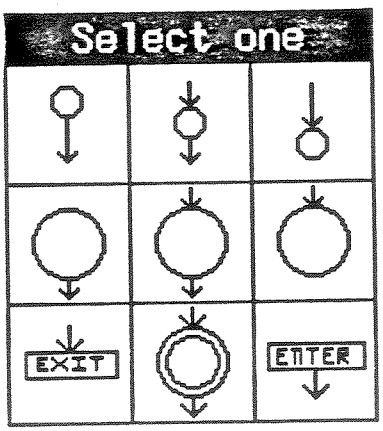


Figure 6.2: Node icons.

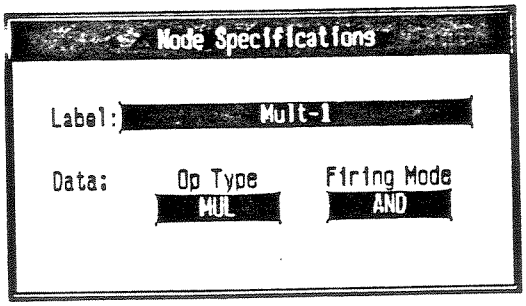


Figure 6.3: The form used to specify a node.

Specifications of data dependencies To minimize duplication of information, data dependencies are specified using forms associated with edges. For each edge, the specification includes, an optional label, initial tokens, the *port-ids* of the *from* and *to* nodes, and the corresponding token-markings. The form is displayed in Figure 6.4.

6.2.3 Scheduler input

DDGTool provides the capability to produce the specification of the DDG in the form expected by the scheduler. The *Output* button on the window banner invokes the translation function that produces three ASCII files which together contain the entire specification of the DDG. The following are their contents.

- The first file contains specification of the types of components to be used in the synthesis process. This information is derived from operation declaration from the *Definitions* form discussed above, and contains an entry for each type of component. Each entry has a unique integer id for the

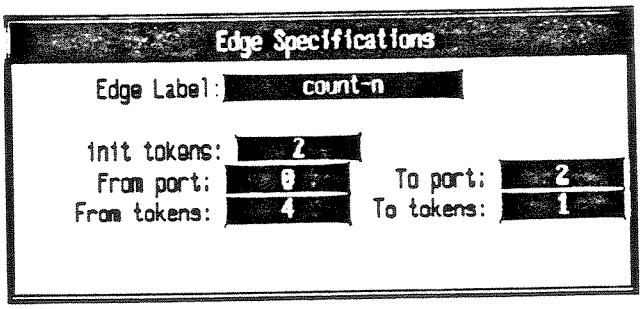


Figure 6.4: The form used in DDGTool to specify dependency information.

type of component, its firing discipline, and its delay.

- The second file contains specification of nodes. An entry for each node contains a unique integer id, an integer denoting the operation it represents, and its actual firing discipline. Again, as explained above, the actual firing discipline may be different from the default for the corresponding operation only in the case of communication nodes.

- The third file contains specification of edges, containing an entry for each edge of the DDG. The entry contains, a unique integer id of the edge, for each of its *from* and *to* nodes, its unique id, port id, and token marking, and the number of initial tokens present on the edge.

6.3 Design of the heuristic scheduler

To design a MRR architectures for acyclic DDGs, the scheduler traces the steps outlined in Section 6.1. The scheduler begins by reading in the files specifying the DDG, which are generated using the DDGTool described in the last section. It then requests the user for input latency. Given the

input latency and the DDG specification, it computes the necessary schedule cycle length to achieve the MRR (see Section 5.4). It does this by solving the TREs for the DDG. The scheduler then sets up and solves the TCEs for the data dependencies of the DDG. Following the solution of the set of TCEs, it constructs the execution graph. Finally, and importantly, it schedules and assigns the operations in the execution graph in a two-step process.

In this section the scheduling heuristics will be examined and their performance will be discussed. At first, a look is taken at the heuristic ordering among operations — the node ordering heuristic. The next heuristic is the one which allows the separation of time-scheduling of operations and their assignment to components — the slotting heuristic. This separation gives rise to an assignment problem. This problem is formulated and a heuristic is proposed to solve it — the assignment heuristic.

6.3.1 Node ordering heuristic

To the extent that the scheduler is intended to synthesize MRR architectures for acyclic DDGs, it obtains an optimal solution, this being guaranteed by Theorem 7. It trades graph latency to achieve that optimal solution. Although graph latency is not of primary concern, it is desirable to keep it as low as possible without violating the MRR constraint. In this sub-section, is presented an adaptation of a list scheduling heuristic originally proposed by Coffman [10]. This heuristic is used to *rank* operations in the execution graph when there arises a contention between similar operations for the same execution time-slot. The contention arises because of the limited number of components, which are shared by the operations. The operation with lower rank has a lower priority, and is likely to be postponed to the next available

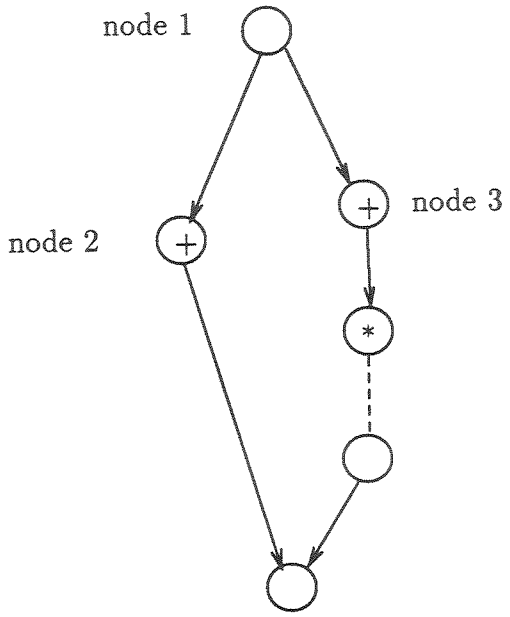


Figure 6.5: An execution subgraph.

slot.

Consider the execution subgraph shown in Figure 6.5. Assume that the MRR architecture for the DDG requires only one component of each type. After the execution of node 1, nodes 2 and 3 can execute, and thus contend for the single adder in the architecture. If node 2 is chosen in preference to node 3, the graph latency for the subgraph increases by the delay of an addition. However, if node 3 is ranked above node 2 and is given a priority of execution over it, then the graph latency remains unaffected. This is due to the fact that node 3 is on the more *critical path* of the subgraph; node 3 is *more critical.*

Therefore a scheme is needed by which the nodes are ranked in the execution graph so that a more critical operation is given a higher rank. The heuristic about to be presented was proposed by Coffman in [10] for a scheduling algorithm intended for obtaining optimal nonpreemptive schedules of acyclic graphs executing on two identical processors. A similar heuristic was pro-

posed by Kaufman in [29] for near-optimal multiprocessor scheduling of a tree-structured graph.

The heuristic, which is a modification of the *latest start time* ordering (LST-ordering) scheme, is applied to the DDG and not its execution graph. It imposes a total ordering on the nodes of the DDG. The current heuristic ignores the multi-rate characteristic of the DDG, but does take into account the delays of the operations.

The basic LST-ordering heuristic ranks DDG nodes using the following algorithm:

The algorithm maintains for each node of the DDG a list of immediate successors — all nodes which can be reached from the node by traversing a single dependency. It also maintains a list of nodes by the latest time at which they may start executing. A node remains in the list for the duration of its delay and is inserted into this list after, and only after, all its successors have exited the list. A node is assigned a rank when it exits the list. The rank assigned is the value of a monotonically increasing integer; the integer is incremented after each rank assignment. If two nodes leave the list at the same time, they are ranked in an arbitrary order.

The stop node of the DDG is inserted in the list first and assigned rank 1.

In the LST-ordering, the more critical operations appear later in the list and are therefore assigned a higher rank. But the scheme assigns ranks to equally critical nodes in an arbitrary order. Coffman's modification to this basic algorithm alleviates this problem by *anticipating* the potential criticality of a node. It uses the following criterion to break the tie between two nodes exiting the list.

For any two nodes exiting the list at the same time, their successors' ranks are compared. The one with a higher ranked successor is given the higher rank. If for one node the set of its successors is a subset of those of the other, the tie is broken in favor of the one with more successors. If the set of successors is identical for the two nodes, the tie is broken arbitrarily.

The ranks assigned to nodes are used by the scheduler to determine which of the contending invocations in the execution graph gets scheduling priority. The invocation whose corresponding node in the DDG has a higher rank is given higher priority. If two invocations of the same node contend for a slot, the one enabled earlier (i.e., one which has lower invocation number) gets higher priority.

The scheduling algorithm uses the execution graph of the DDG in the following manner:

Begin by scheduling the first invocation of the start node at time t=0, and schedule its remaining invocations once every L_I cycles of the global clock.

For all other nodes, schedule an invocation if

1. there is a token on each of its input edges, and
2. there is a component available to start the execution during that cycle.

Else, postpone the execution of the invocation to the next available slot.

Use the multi-rate execution model introduced in Chapter 3 to schedule the output phase of the execution, and deliver the tokens to appropriate output edges.

6.3.2 Slotting heuristic

At the end of Section 5.4, a strategy was described to schedule an acyclic graph with number of resources equal to the MRR of the DDG. In it, execution times were predefined for components, and operations were assigned to them as part of the scheduling process. This strategy guarantees a feasible assignment. That is, the scheduling of no operation violates the condition that there will eventually be a component free to execute a given operation, either already scheduled or to be scheduled.

Such a strategy is necessary to give the ability to compute a static schedule *locally* for each operation. Consider for a moment that such a strategy were not used. Instead, an operation is scheduled whenever all its dependencies are satisfied and independent of other similar operations. This strategy will also tend to immediately assign the operation to a component. The static nature of the schedule would then imply that the component may not be used for any other operation during all times which have the same values modulo L_{Sch}. Thus a component is available for assignment if, and only if, the component is unassigned for the entire duration of the operation. It is now conceivable that a component may earn assignments such that it is idle long enough between the planned executions, enough to keep it from being able to take all the assignments necessary for the satisfaction of the MRR criterion. Such fragmentation of the execution cycle of a component due to "inopportune" scheduling can only be overcome by *a priori* decision about the execution slots.

Short of a strategy equivalent to the one being considered, a global evaluation of the scheduling requirements has to be made. In fact, the *force-directed scheduling* heuristic of Paulin [47] is such a global analysis technique to obtain a feasible schedule.

The next question to be answered is: What are the "appropriate" positions of the slots?

The question is important because "inappropriate" positions can have the adverse effect of excessively increasing the graph latency of the schedule. The best arrangement of slots is the one that minimizes this adverse effect.

Clearly, since the lower bound on the adverse effect is 0, there must exist a best arrangement. Yet, finding the best arrangement of slots would require determining the schedule without inserting slacks and then determining the slots so that the increase in graph latency is minimized. The presence of precedence between operations, however, makes this approach combinatorial in complexity.

The other option is to choose an *a priori* arrangement, which is the choice made for the current implementation.

In the current implementation of the scheduler, the slots of all components are abutted to the first cycle of the schedule cycle, and there are no idle cycles between consecutive slots; all idle cycles are at the end of the schedule cycle. This scheme was primarily chosen for its simplicity and execution speed of the implementation.

Obvious variations to this basic scheme are certainly possible, and could be incorporated in the future versions of the scheduler. For example, instead of inserting all idle cycles at the end, they could be distributed evenly throughout the schedule cycle. Or, not all components are slotted aligned to the first cycle. Or, the slotting pattern could be input externally.

Apart from simplicity and speed, one other significant practical advantage accrues from this choice: An architecture contains multiple components

of the same type. Since because of the choice of the arrangement of the slots the components of a given type execute in lock-step synchrony, the control for all components of a given type can be derived from the same bits in the control ROM, thereby reducing the size, and therefore the cost, of the ROM.

Consider, for example, the DDG corresponding to the FIR filter presented in Chapter 3. The DDG is reproduced here again in Figure 6.6. Assuming the same input latency as before (i.e., of 300 ns.), and delays of 20 ns., 40 ns., and 80 ns. for a bus, an adder, and a multiplier respectively, the scheduler finds a schedule which has schedule cycle length of 600 ns. and graph latency of 1800 ns.

The quality of MRR schedules produced by the scheduler is dependent on the interplay of

- heuristic used to rank invocations,
- slotting heuristic, and
- *relative primeness* of the input latency and the delays of the components.

If these three factors are badly matched, the graph latency can be adversely affected. For example, in the above FIR example, the graph latency is more than twice the length of the MRR schedule presented in Chapter 3. The adverse effect can also be in the form of large number of components when graph latency constraints are imposed. In fact, slots determined *a priori* might even disallow satisfaction of graph latency constraints by forcing the introduction of otherwise unnecessary slacks.

On the other hand, improving the factors can lead to better performance. For instance, if the last of the factors is improved in the FIR example

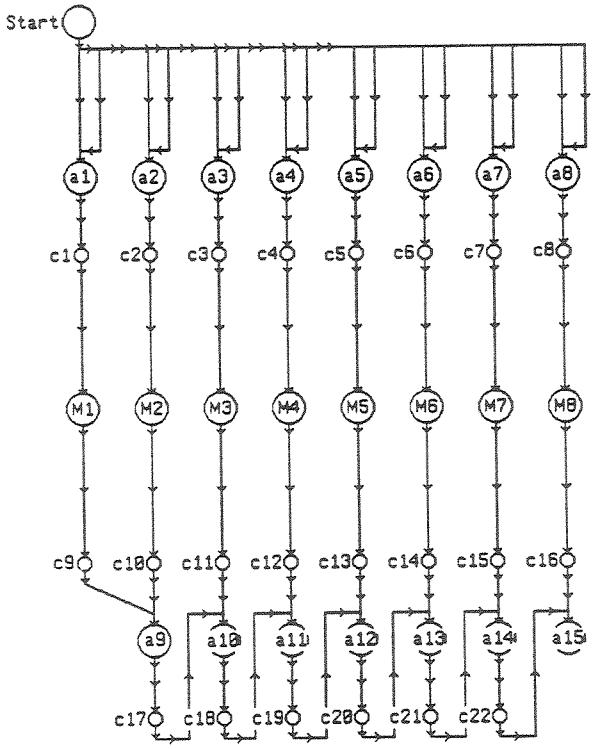


Figure 6.6: DDG representing a 16-point FIR filter.

by choosing the input latency to be 320 ns., the schedule latency drops to 320 ns. as well, and the scheduler obtains an MRR schedule which is merely 1180 ns. long.

The last factor of relative primeness affects in two ways.

The relative primeness of the component delays and the input latency tends to increase the schedule cycle length. A schedule cycle length which is a large multiple of the input latency implies a combined schedule for multiple invocations of the DDG. This in turn means that more invocations are likely to contend for any given slot, and therefore it is more likely that slack will be introduced for any given invocation. The result of introducing slack is cumulative and there is a potential risk of increasing the graph latency enormously.

This pernicious effect can be alleviated by modifying heuristics affecting the first factor. For example, one may introduce heuristics involving critical path analysis, where criticality of unscheduled operations is reevaluated every time an operation is scheduled. This problem can be handled by fixing the schedule cycle length externally, as discussed further in Section 6.4.

Secondly, the factor of relative primeness may interact with the slotting scheme:

The major shortcoming of any *a priori* slotting arrangement is that it does not take into account the topology and the precedence relations in the execution graph, and thus risks being a bad choice. This phenomenon is more likely to be manifested when the topology of the execution graph is regular and when operations with relatively prime delays follow each other. DDGs with irregular topology can be less prone to this adverse effect. For example, for the DDG shown in Figure 6.7, the current scheduler produces a graph latency no worse than any other MRR scheduler.

For the DDG of Figure 6.7 and the component delays shown there, for which the lower bound on the graph latency is 500 ns., the scheduler generates schedules with graph latencies of 660 ns., 540 ns., 580 ns., and 560 ns., for input latencies of 200 ns., 240 ns., 280 ns., and 320 ns. respectively. In fact, it can be shown that, in the case of input latency of 240 ns, no scheduler which incorporates the operation ordering heuristic of the previous section can produce a schedule better than the current one.

This particular effect of the regularity of the DDG topology can be avoided by using variations on the slotting scheme, suggested previously. For example, the skewing of slots among adders helped in the manually obtained MRR schedules of Chapter 3.

Separation of scheduling and assignment

In the slotting heuristic presented above, all components of a type have the same execution time slots. If an operation is to be assigned an execution time, it is assigned to an execution time slot. But at the time of scheduling the operation need not be assigned to a particular component, since it could be assigned to any of its type of component; the actual assignment of operations to particular components is postponed until after the scheduling process is over and handled separately.

There are advantages to this kind of separation, since independent optimization criteria can be applied to the assignment phase. In the next subsection one such optimization criterion is stated and a heuristic to solve the related assignment problem is introduced.

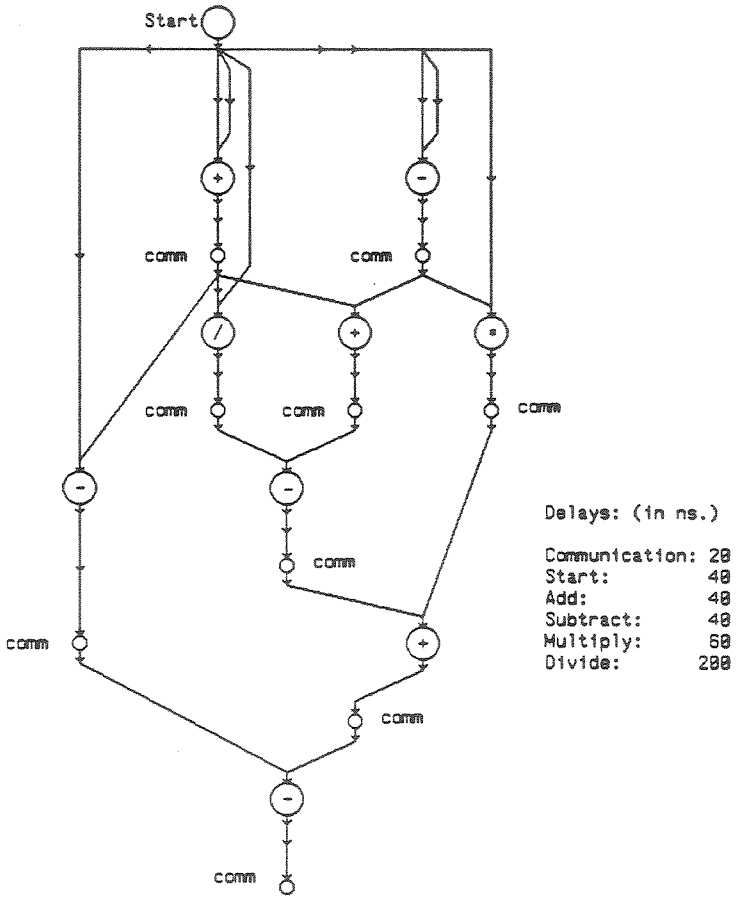


Figure 6.7: DDG with an irregular topology.

6.3.3 The assignment problem

The assignment of invocations to physical components is the final step in the architectural synthesis process. In many methodologies this step of binding the operations to physical components is performed as part of the scheduling process. As discussed in the last section, the two steps were separated in order to: 1) simplify the scheduling process, and 2) allow different optimization criteria to be developed for the assignment process. The optimization criterion is: *minimization of connections to physical components.*

The connectivity model of a physical component was introduced in Chapter 3. Every port of the physical component is connected to one or more communication components, i.e. busses, over which tokens arrive to, or depart from, the port. A *connection* is therefore identified by an ordered pair <port, bus>. Since a *register file* interposes between a port and a bus it is connected to, minimizing connections in the architecture, is equivalent to minimizing the number of register files in the architecture.

NP-hardness of the assignment problem

The assignment of operations to components to minimize the number of connection is NP-hard. To be convinced of this fact, consider a very simplified problem:

Concentrate on only one type of operation in the DDG and the corresponding type of component. Focus further on one of its ports and draw the following *conflict* graph.

Each invocation corresponds to a node in the graph. Two nodes in the graph are connected by an undirected edge, if and only if, transfer of a token corresponding to the port under consideration for one of the invocations

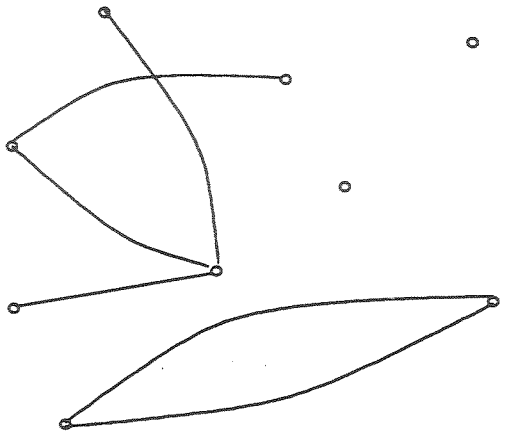


Figure 6.8: Graph-theoretic formulation of the simplified assignment problem.

is concurrent with that of the other invocation. See the example shown in Figure 6.8.

If invocations are assigned to a component such that tokens corresponding to them are transferred concurrently, the component's port must be connected to two or more busses, as simultaneous transfers must be accomplished over more than one separate busses.

For this simplified case, the problem of minimizing connections is reduced to the following graph-theoretic problem:

Assuming that n invocations have to be assigned per component, if there are k components, then the task is to find k sets of n nodes each, such that the total number of edges spanning intra-set nodes are minimized. Of course, since only the invocations that are non-concurrent may be assigned to a component, a set may not contain pairs of nodes which represent concurrent invocations. The nodes and their incident edges are deleted from the graph once they are selected to form one of these above sets.

The best possible solution is one in which for each of the k sets, there are no intra-set edges. But this corresponds to searching for successive *anti-*

cliques, which is an NP-complete problem. Hence the conclusion:

Lemma 33 *The assignment problem of minimizing connectivity is NP-hard.*

The above problem is similar to the classic graph-partitioning problem enunciated by Kernighan and Lin [31], although the objective of the above problem is the inverse of the graph-partitioning problem; in the graph partitioning problem, the objective is to minimize the inter-partition edges, while in the above the attempt is to select the sets so that the edges are preferably between them, than within them.

Assignment heuristic

To solve the assignment problem at hand with the optimization criterion of minimization of connectivity, a two-step heuristic is proposed, which is used in the current implementation of the scheduler. The first step chooses a port to optimize at a time, and the second step optimizes the connectivity for it.

Choice of port

The first step is motivated by the observation that assignment of a port, and therefore of corresponding operations to the respective components, results in reservation of busses as well. This reservation leaves the later port assignments facing worse constraints. So the following heuristic is used:

Assign most busy port first.

An operation and the corresponding type of component may have one of three possible states, depending on the state of its ports: 1) *unassigned*,

2) *partially assigned*, or 3) *assigned*. An invocation all of whose ports have been assigned, has the state *assigned*; ones which have only some of its ports assigned carry a state *partially assigned*; operations with no ports assigned are *unassigned*. For example, an addition operation has three ports, two input and one output. Initially, the operation of addition is unassigned. Let the output port be assigned first, after which the addition operation is declared to be partially assigned. When both the input ports of the operation are assigned, it becomes assigned.

During the assignment of ports, *partially assigned* operations are given higher priority that the *unassigned* ones. This is motivated by the consideration that the components which have already been bound to invocations are better assigned earlier, for the freedom of assigning them to minimize the token transfer conflict no more exists.

To find the busiest port, the maximum number of invocations that could potentially be assigned to a physical component is computed. For each such physical component, the number of tokens to be transferred over each port are calculated by taking the product of maximum number of invocations just mentioned, and number of tokens transferred over the port. The busiest port is the one that has the highest product.

The process of assigning a port involves assignment of all of the referenced operations to physical components, as well as assignment of the transfers of tokens accessed over that port to busses.

It is important to point out that above is only an approximation to the busiest port, which cannot really be decided prior to the assignment. Observe that it is not necessary that a physical component has to be assigned the maximum number of invocations. For example, suppose there are four slots per

schedule cycle and there are nine invocations to be assigned, three components are needed. Suppose further that the arrangement of the invocations in the schedule cycle is:

$$\begin{array}{cccc} i_1 & i_2 & i_3 & i_4 \\ i_5 & i_6 & & i_7 \\ i_8 & i_9 & & \end{array}$$

It is easy to see that invocations i_1, i_2, and i_3 can be mapped to one of the components, i_5, i_6, and i_4 can be mapped to the second, and the remaining can be mapped to the last. This assignment leaves each component idle for one cycle although the maximum potential number of invocations per component is four.

Assignment of invocations

The second part of the heuristic, deals with assignment of invocations to components to minimize connectivity. This problem was expressed in graph theoretic terms in the last sub-section. As stated there, this problem is NP-hard and a simple heuristic of NlogN time complexity is used.

The heuristic used for minimizing connectivity for a port is based on the observation that *the potential number of busses that a port may have to be connected to, is bounded above by the highest degree of any node in the selected set of nodes.* It is therefore reasonable to minimize highest degree chosen. The easiest way to ensure this is to:

Sort the nodes in ascending order by their degree and choose the first n nodes that are non-concurrent to form a set. Delete these nodes and repeat the procedure on the remaining graph.

Notice that this is a heuristic and not an exact approach. It suffices to point out that the number of busses are not *equal* to the maximum degree,

but only *bounded above* by it. It is entirely possible to find a set that does not minimize the highest degree contained, but is optimal nonetheless.

Notice also that the heuristic selects an invocation only once, and there is no facility for backtracking.

The above heuristic attempts to choose n nodes every iteration, instead of choosing an "average" number of invocations per component. This is not always necessary, as can be easily found out in the following example. However, it is sufficient to avoid the problem that could arise in the arrangement below, given that a single pass is made over the list of invocations:

$$\begin{array}{cccc} i_1 & i_2 & i_3 & i_4 \\ i_5 & i_6 & i_7 & i_8 \\ & & & i_9 \end{array}$$

Here, the subscripts reflect the order of the invocations in the sorted list. It is possible to assign three invocations per component. However, without an ability to backtrack, it is likely that the algorithm will assign the invocations i_1, i_2, and i_3 to the first component, the invocations i_4, i_5, and i_6 to the second, and be trapped into an infeasible assignment for i_7, i_8, and i_9.

It is easy to see that the following is true about the heuristic:

Lemma 34 *The invocation assignment heuristic finds a feasible schedule.*

Proof: Consider the state of a slot described in term of the count of invocations scheduled to execute in it but yet to be assigned to a physical component. Thus, a slot is either *empty* or *non-empty*, depending on the number of unassigned invocations is zero or not. Whenever an invocation belonging to a slot is assigned, the count is decremented by one.

The assignment heuristic assigns a component one invocation per slot, for every non-empty slot. Thus for each new physical component being assigned, for every non-empty slot the count of unassigned invocations is decreased by one.

But the scheduling algorithm guarantees that the maximum number of invocations per slot is no greater than the number of components available. That is, for each slot, the maximum value of the count is less than or equal to the number of components. Thus at the end of the assignment process, the count of each slot is equal to zero and all invocations are assigned. Also, no component is assigned more than one invocation per slot. □

The heuristic was evaluated by executing it on the DDG shown in Figure 7.1. (The DDG is discussed in greater detail in Section 7.1.) Varying sizes and topologies of the execution graph result when input latencies are varied.

When the input latency is fixed at 20 cycles, the architecture requires exactly 1 component of each type, and the number connections necessary are minimum and should equal the number of ports of the components, i.e. 16. The scheduler requires 18 connections: 16 corresponding to the connections between the ports and the bus, and two extra to connect a staging buffer required to accommodate consecutive transfers over the bus, forced into the schedule by the dependency spanning two communication nodes.

When the input latency is changed to 10 cycles, the architecture requires 2 busses, 2 divide units, and 1 component of each of the other types. The minimum number of connections that would be required is 18 and the worst-case requirement corresponding to each port being connected to each bus would result in 36 connections. The actual requirement lies between these

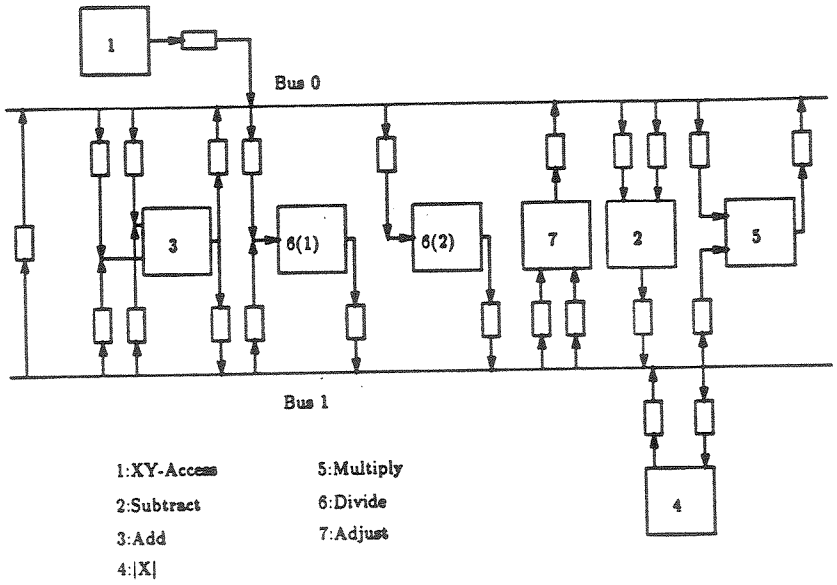


Figure 6.9: Architecture for the convex hull algorithm for input latency of 10 cycles

two bounds. The heuristic results in a requirement of 24 connections. The resultant structural diagram is displayed in Figure 6.9.

When the input latency is further reduced to 5 cycles, the architecture requires 3 busses, 4 divide units, and one unit each of the remaining types. This corresponds to the minimum requirement of 22 connections and a worst-case requirement of 66 connections. On the other hand, the heuristic results in 35 connections.

6.4 Extensions to the heuristic scheduler

Primary objective of the scheduler is to obtain an MRR schedule for an acyclic multi-rate DDG. However, other capabilities greatly enhance the

utility of the scheduler as a design tool. This section contains brief descriptions of the extensions made to the scheduler.

Schedule cycle length

The scheduler described herein accepts an input latency value and determines the minimum schedule cycle length (L_{Sch}) to satisfy the MRR as predicted from the TREs. In a previous section it was pointed out that obtaining an MRR schedule may require a very large L_{Sch} value, and that this implies a large control ROM and might also imply large graph latency. It would be desirable to allow the user to choose a value for L_{Sch}, and thereby exert direct control over the size of the control ROM and, less directly, control the graph latency.

A capability of accepting a user-specified L_{Sch} and computing a schedule based on this value has been added to the basic MRR scheduler. The scheduler computes the numbers of components necessary to guarantee a feasible schedule, numbers that may be larger than the MRR. The scheduling and assignment heuristics used are the same as before.

This capability is illustrated in Section 6.6.

Number of components

The MRR schedule defines a single point in the solution space of the desired architecture and therefore represents a particular trade-off between hardware and performance. As explained in the last chapter, Section 5.5, a resource constrained scheduler can be used to iteratively implement graph latency constrained scheduler, and in so doing, explore the solution space by

varying the quantity of hardware. For this purpose, the scheduler must provide the capability of specifying the number of hardware components in the architecture and of computing a corresponding schedule.

The scheduler is equipped with a capability of accepting the numbers of components and computing a schedule based on this resource availability. Since it uses the same underlying scheduling and assignment heuristics as the MRR scheduler, however, it does not satisfy the assumption in Section 5.5.1. It nonetheless does provide the capability for exploring the hardware – graph latency trade-off.

This capability is also used in the examples of Section 6.6.

ASAP schedule

Before exploring the solution space under graph latency constraint, it is vital to establish the lower bound on it. This bound can be found by assuming no hardware constraints, and scheduling invocations as soon as they are ready. This type of scheduling is called *as soon as possible* (ASAP) scheduling. This capability has also been added to the scheduler.

6.5 Storage considerations

Throughout this work, storage components have been left out of the cost formulation. The reason for this choice has been that their contribution to the cost cannot be given a deterministic formulation prior to actual scheduling of the other operations, and their sharing can only be determined *a posteriori* to component assignments via life-time analysis of data stored.

In this section, attention is given to storage elements and the trade-off that exists between them and communication requirements. This trade-off also

has an implication on the choice of the global clock speed, which is discussed in greater detail in the next section.

Trade-off between communication and storage components

The model of computation used herein for an operation represented by a DDG node requires that all tokens be available prior to its scheduling. This implies that there be placed registers at the input ports of the components to hold the tokens. No such requirement exists for output tokens. So the output tokens could be transferred immediately to their destinations.

However, in the present approach, the communication components are treated like other components and *must* wait for tokens to be available at their input before firing. As a result, input registers must be provided for these tokens as well. Recall from Chapter 3, that these input registers for communication components are better modeled as output registers of computational components. And thus the connectivity model of a component shown in Figure 3.4 has register-files at input as well as output ports.

Since communication components are scheduled independently, the output register-files of the other components have to hold data until a communication component becomes free to transfer the data. Thus, the more the number of communication components, the smaller the output buffers should become.

The sizes of register-files is determined from the life-time analysis of the data they hold. Chapter 3 describes the behavioral model of a buffer. According to it, a register of a register-file may be read from and written into in the same clock cycle. Using this model, the size of a register-file is given by

the maximum number of tokens residing in it during any cycle, since tokens that do not reside in the register-file concurrently can share the same register.

The current approach would prevent the output buffers from vanishing altogether, however. This is due again to the model of a communication component being same as that of a computational component, according to which, the input token must be present *prior* to the transfer. Thus, the fidelity to the computation model prohibits the trade-off between communication components and output buffers to be carried to its extreme.

Notwithstanding the requirement of the model, this inefficiency can be corrected heuristically under certain timing conditions as discussed below.

The look-ahead heuristic

An example is the best vehicle to present this heuristic and the supporting arguments:

Assume that an algorithm uses only two types of computational components: adder and multiplier. Assume that their delays are 20 and 60 ns. respectively. Assume further that the communication delay over a bus is 15 ns. The global clock cycle is chosen to be 40 ns.

Under these assumptions, the delays of the adder and the bus are specified as 1 cycle each, and that of the multiplier is 2 cycles.

But, notice here that, in fact, the addition followed by the transfer of its result can be achieved within a single clock cycle, and similarly, the multiply and the transfer of its result can be achieved within two clock cycles. In effect, the subsequent communication operation *can* be scheduled overlapped with either an addition or multiplication.

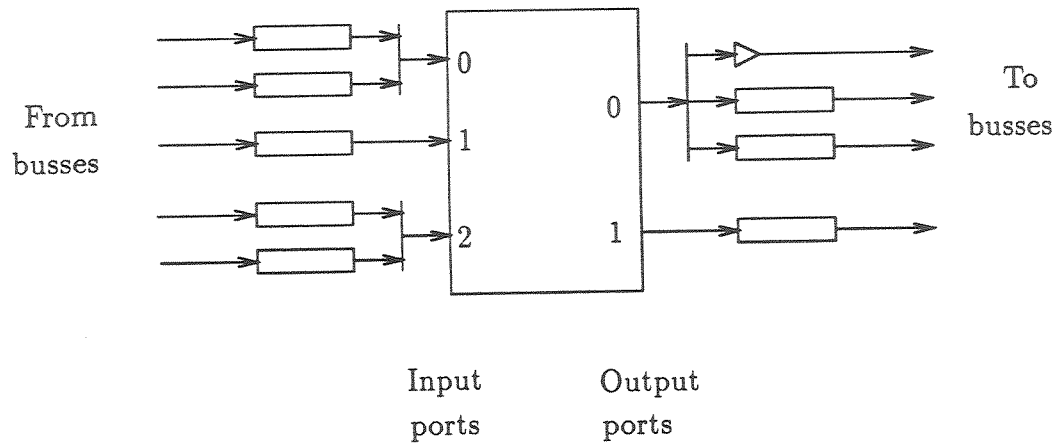


Figure 6.10: Modified connectivity model under the look-ahead heuristic.

This possibility can be incorporated into the scheduling process by explicitly introducing the *look-ahead* heuristic. According to this heuristic, the communication operation to transfer a token output by an operation is permitted to be scheduled concurrently with the token's production. Of course, the transfer is postponed in the usual way if a communication component is not available at the time.

Under this heuristic, it is easy to see, output buffers can be completely eliminated by increasing the number of communication components. The corresponding connectivity model for a component is shown in Figure 6.10.

Other methods to reduce storage requirements The components in an architecture produced by the new synthesis approach will have registers at the input ports. It is possible to attempt to reduce their number by applying a technique analogous to the look-ahead heuristic proposed above. Using such a technique, the execution of an operation will be scheduled such that the tokens are transferred directly into a component without being held temporarily in an input buffer. But this involves making stronger assumptions about the set-up

and hold times for input phases of computational components. For multi-cycle combinational components, this method still does not help, for the incoming tokens must also be held in a buffer to make them available throughout the execution phase.

Another method for reducing the number of registers is to do a global life-time analysis of tokens and to share registers among tokens whose life-times are non-concurrent. But whenever sharing of registers in done in this fashion in which registers are shared between multiple components, a need arises to insert multiplexers, demultiplexers, and auxiliary busses, thereby introducing extra hardware in the architecture.

6.6 Examples

In this section are presented two large examples which illustrate the capabilities of the synthesis tools described in this chapter.

6.6.1 Fifth order elliptical filter

Here, synthesis is carried out for a fifth order elliptical filter described in [33]. This example was chosen as a bench mark for the 1988 ACM/IEEE Workshop on High-Level Synthesis. Figures 6.11 a and b show the DDG of the filter adopted from the Value Trace shown in [60].

The DDG shown in the figure is an acyclic graph obtained by clipping open the loops of the original filter graph, in which all directed loops have a single token. As explained in Chapter 5, this acyclic graph must now be executed under graph latency constraint which is same as the input latency of the filter.

Parameter	SPAID	HAL	Present		
L_I	19	19	19	25	30
+	2	2	3	2	1
*	1^1	1^1	2^2	1^2	1^2
Busses	5	6	3	2	2
Connections	14	45	34	15	12

[1]: Pipelined multiplier. [2]: Non-pipelined multiplier.

Table 6.1: Costs of architectures for fifth order elliptical filter.

For this example, it was assumed that the delays of adders and multipliers are 1 and 2 clock cycles respectively, and that the look-ahead heuristic can be used. Table 6.1 shows architectural costs in terms of numbers of adders, multipliers, busses, and connections for various input latencies. For comparison, designs from two other synthesis systems are listed as well for the input latency of 19 cycles [21]. Notice that the scheduler described herein needs fewer busses. But it requires one extra adder to achieve the necessary graph latency. The latter is the effect of the *a priori* slotting scheme as anticipated in Section 6.3.

6.6.2 Phase modulator

This sub-section describes the synthesis of a phase modulation circuit shown schematically in Figure 6.12. The DDG for the algorithm is shown in Figures 6.13 (a, b, c, and d) drawn using the hierarchical specification capability of the DDGTool.

The specification of the phase modulation was adopted from [3], with a modification introduced in the form of the 4-point "pre-filter" FIR to prepro-

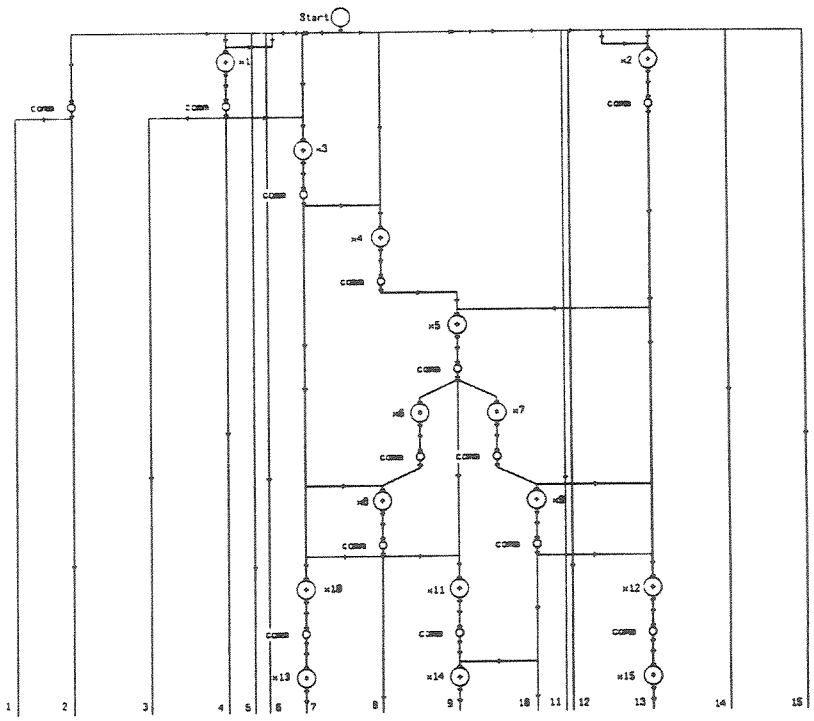


Figure 6.11: (a): DDG for the fifth order elliptical filter.

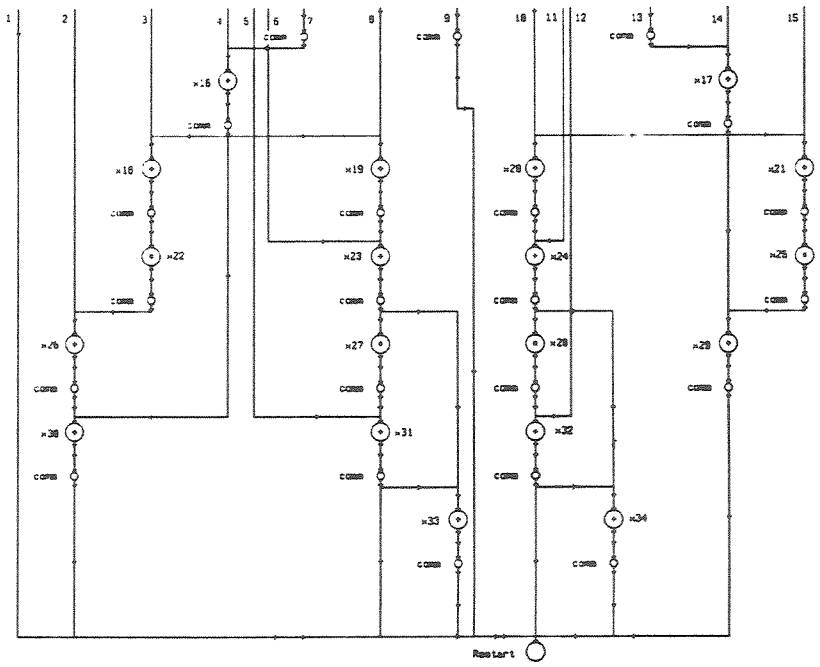


Figure 6.11: (b): DDG for the fifth order elliptical filter (continued).

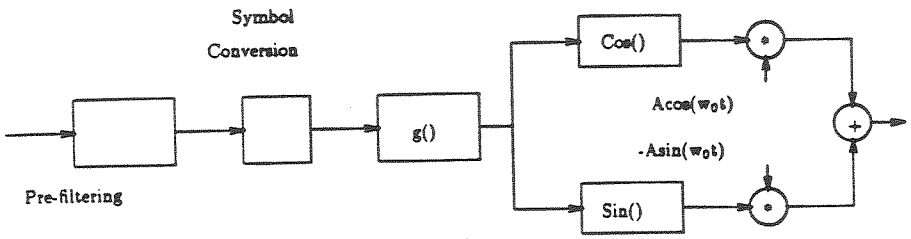


Figure 6.12: Block diagram of a phase modulator.

cess the input signal before modulating the carrier phase with it. The carrier frequency is assumed to be 32 times that of the input signal. Its specification is found in Figure 6.13(b).

The phase modulator shown in Figure 6.13(a) is a multi-rate circuit, computationally specified changes in data rate occurring at the node "8-4×2-serializer" and inside the sub-DDG "g", at the node "Repeater".

Sub-DDG "g" in Figure 6.13(a) represents the modulating convolution implemented as an 8-point FIR. Any of the several modulations described in [3] can be applied by employing suitable coefficients in "g" (see Figure 6.13(c)).

Finally, Figure 6.13(d) shows the specification of the actual carrier phase modulation computation.

The following delay values, in terms of clock cycles, were used:

Parameter	Arch. Chars.		
L_I	200	500	1000
L_{Sch}	200	500	1000
Graph Latency	875	1720	2701
Input device	1	1	1
+	2	2	1
*	7	4	3
Repeater	1	1	1
Serializer	1	1	1
w0*t	1	1	1
Sine	3	2	1
Cosine	3	2	1
Busses	6	3	2
Connections	226	69	36

Table 6.2: Architectural characteristics for the phase modulation circuits.

Communication	1	*Input*	1
+	2	*	5
Repeater	9	*Serializer*	5
$w0 * t$	3	*Sine*	8
Cosine	8		

Architectural characteristics for three different input latencies are shown in Table 6.2. The latencies input to the scheduler are also specified in terms of number of clock cycles.

6.7 Timing constraints

In the last section, an example was given to illustrate the concept of the look-ahead heuristic for scheduling communication operations in order to reduce the number of register files. The example showed how certain constraints

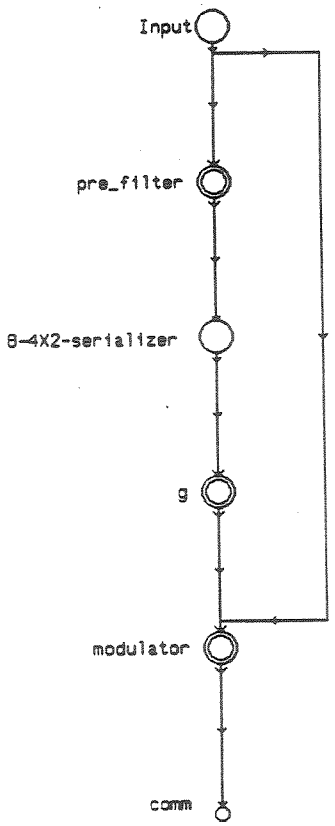


Figure 6.13: (a): Top level DDG for the phase modulation computation.

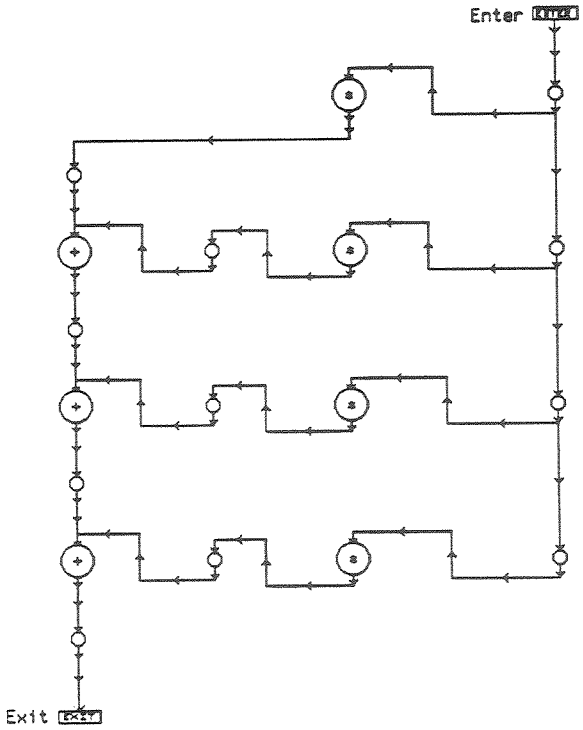


Figure 6.13: (b): Sub-DDG for the "pre-filter".

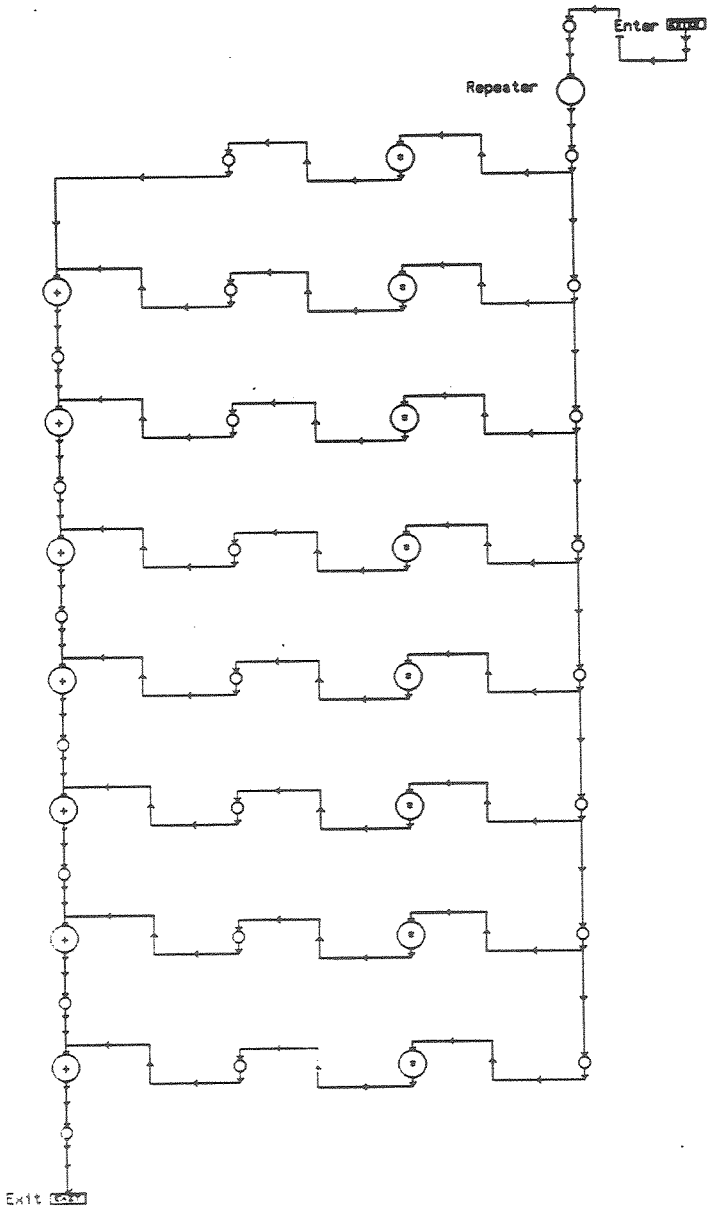


Figure 6.13: (c): Sub-DDG for "g".

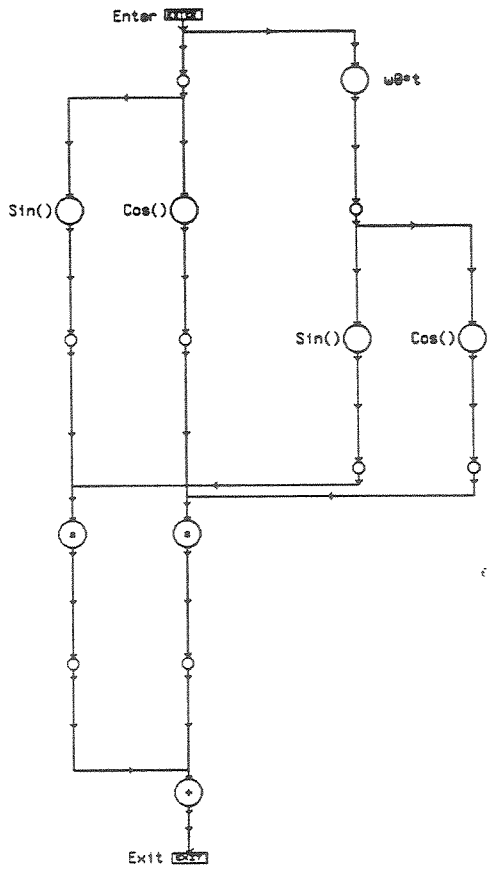


Figure 6.13: (d): Sub-DDG for the modulation computation.

on delays of computational and communication components led to reduction of register-files. In this section, the discussion on the use of the new synthesis approach to meet other timing constraints is carried further.

The ability to accommodate multiple communication component types also offers a solution to a problem that often plagues automated design systems. The problem particularly arises when the design involves several VLSI circuits laid out on a printed circuit board. In this situation, the communication delays between the VLSI circuits are much larger than those within them, and using the same delay values for both types of communication can lead to either pessimistic or erroneous designs.

The above problem is handled in a straight-forward manner in the present approach: All the busses, both internal and external to the VLSI circuits, can be assigned equal worst-case delays, corresponding to those characteristic of the external busses. The fact that these communication delays are several clock cycles long does not affect the scheduling process.

But this method of assigning the same delays for internal and external busses makes inefficient use of the higher bandwidth available for internal busses. A better approach is the following:

The DDG can be partitioned into sub-DDGs, each sub-DDG corresponding to a distinct VLSI circuit. These sub-DDGs are connected via communication nodes which are mapped to external busses. The external busses may now have delays which are much longer (i.e., many cycles longer) than those of the internal busses of the circuits. The synthesis process will automatically schedule "external" communication nodes on external busses and "internal" ones on the faster busses internal to the circuits.

The current implementation of the scheduler, however, will tend to

combine the faster busses of different circuits. Also it will attempt to map other operations belonging to different partitions to common components. It is therefore necessary to modify the assignment process to allow specification of sets of nodes that may share components.

Chapter 7

Further extensions and future directions

So far in this document a basis for high level synthesis of hardware pipelines has been discussed. This basis represents a suitable step for extending the range of high level synthesis to algorithms which have variations in data rates, and extending its domain to multi-rate components. Yet, there are certain features that the current basis lacks. Some of these features are: 1) handling hierarchical design, 2) handling internally pipelined components, and 3) handling data dependent computations. In this chapter, these and other problems as well as some possible solutions will be discussed. The intention is to indicate some future directions in which to carry forward this research.

7.1 Data dependent to data independent structure transformation

Most of the pipeline design methodologies, as also the one outlined herein, disallow data-dependent branching in algorithms. The reasons for this restriction are: when there exist data dependent branches, it is hard to obtain a "good" prediction for hardware resource requirements, and it is impossible to obtain a static schedule.

The usual approach taken in the case of data-dependent computations is to allow for the worst case requirements. For example, Sehwa ([44]), uses the technique of "Conditional Assignment" to optimize on resource requirements. Similarly, Cathedral II uses the "Value Trace" technique to iteratively estimate

hardware requirements [13].

Here, two techniques are illustrated by which a data dependent computation is transformed into a DDG whose structure is data independent. Essentially, a control dependency is transformed into a data dependency by choosing an appropriate set of operands, which is further transformed into a data independent structure. Techniques that achieve the first step by using *guards* have been suggested in [2] to facilitate the vectorization of FORTRAN programs. However, the second step proposed here makes it possible to obtain a static schedule.

Convex hull computation

Consider the following computation which is a part of an algorithm called the Jarvis March, used to compute the convex hull of a set of points in a two-dimensional euclidean plane. The computation is conducted for each point i in the set except the seed point represented by coordinates x_s and y_s.

$dx = x_1 - x_s;$

$dy = y_1 - y_s;$

$\theta = dy/(|dx| + |dy|);$

if $(dx < 0)$ $\theta = 2 - \theta;$

else

    if $(dy < 0)$ $\theta = \theta + 4;$

Assume that the points are all distinct, and therefore $|dx| + |dy| \neq 0$.

As presented above, the algorithm contains a data dependent computation, which depends on the signs of dx and dy.

However, the computation can be expressed in terms of two operands A and B which take on different values depending on the values of dx and dy: $\theta = A + B\theta$, where A takes on the values 2, 4, or 0, and B takes on the values –1, 1, and 1 correspondingly. The three conditions which generate these pairs of values are: 1) $dx < 0$; 2) $dx \geq 0$ and $dy < 0$; and 3) dx and $dy \geq 0$; respectively.

A ROM based component can easily be designed which can sense the above conditions on dx and dy, and generate the appropriate pair of operands at the output. In this particular case, the component shares its output port for both operands, and thus is a multi-rate component. If the availability of such a component is assumed, the algorithm may be expressed as a *data independent* DDG shown in Figure 7.1. The node labeled "Adjust" is the node which generates data dependent values of A and B.

Since B only takes values of –1 and 1, a simple 2's complement unit can be configured to achieve the multiplication by choosing it to complement the operand or not.

In the particular example being considered, this transformation has made it necessary to introduce into the architecture an extra component-type for the multiplication. However, in general, this may no be the case, and so such a transformation might still be justified.

Subsequently, the DDG containing the transformation is analyzed and an architecture synthesized for it, without paying attention to the dependence of the computation on data. A consequent practical advantage is, all the design algorithms developed for data independent DDGs can be used without modification. The more important benefit of improved architectural performance derives from the facts that to preserve the data dependent aspect of the

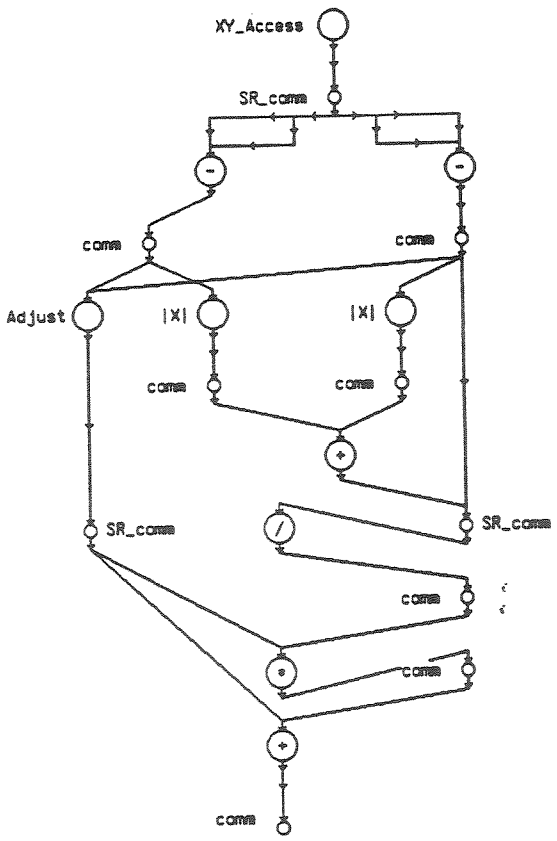


Figure 7.1: DDG to compute convex hull.

computation in the architecture requires the architecture to support dynamic scheduling of operations by employing the mechanism of a data-sensitive state machine, and that it is impossible to include the related components in a static analysis of their utilization and heuristics to improve it.

Operation reordering to obtain common computational form

As in the above, the approach to be illustrated here is based on using expressive power of DDGs to convert a data-dependent computation involving a single variable into a data-independent computation. A technique similar in spirit, yet quite different, was presented by Lin and Messerschmitt in [37]. Their technique depended on the insertion of *nil* tokens and the use of worst case computational latencies, whereas *operation reordering* is used here to obtain results that are more lucrative for hardware synthesis. The motivation in both cases is to obtain a data-independent computation that can be statically scheduled, but there is a further interest here in obtaining a data independent computational structure for the DDG.

The following computation represents two possible executions depending on the outcome of the boolean test:

if (boolean expression)

$$y = ((a_1 + x) * a_2) - a_3;$$

else

$$y = (a_4 * x) + a_5;$$

where all a_i's are constants.

Notice, however, that the two computations can be transformed into a common form as follows. The computation executed if the boolean expression

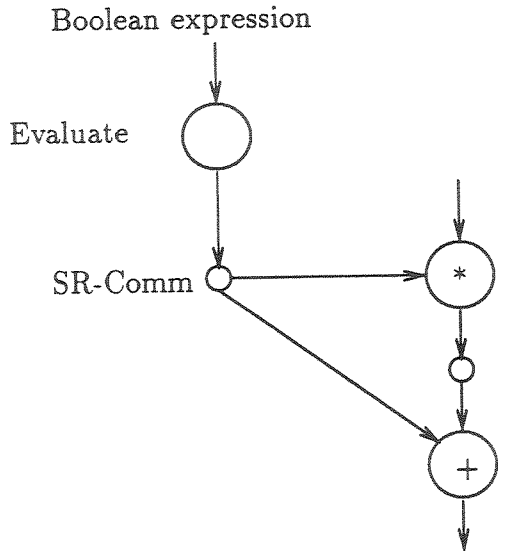


Figure 7.2: A data-independent DDG.

is FALSE is left in the same form as before, but the other computation is transformed to read a_2 * x + $(a_2 * a_1 - a_3)$. Notice that the order and number of operations are now the same in both cases. A node, similar in concept to the "Adjust" node of the last section can be defined, which evaluates the boolean expression, and produces at the output two tokens carrying values a_2 and $(a_2 * a_1 - a_3)$, or a_4 and a_5. The structure of the DDG, shown in Figure 7.2, now has a data-independent path of execution.

In addition to the advantages listed above, an architectural benefit drawn from the transformation, over the architecture employing a dynamic run-time scheduling alternative is that, since in the former the two computational path have been reduced to one, the control ROM needs fewer locations.

It may seem that the method suggested above ought to be generalized by always obtaining a "canonical form" of, say, *sum-of-products*. But such a form might not always be the better form. For example, consider

if (boolean expression)

$y = (x_1 + a_1) * x_2;$

else

$y = x_3 * a_2 + a_3;$

where x_i's are variables and a_i's are constants. For this particular example, it is better to transform the latter computation to $((x_3 + \frac{a_3}{a_2}) * a_2)$, which needs only one each of addition and multiplication. The sum-of-product arrangement, on the other hand, will necessitate more operations.

The above illustration dealt with a very restricted type of computation — one involving only one or two variables. Tacitly, it also used concepts of an identity element, existence of an inverse, and commutativity, associativity, and distributivity among the arithmetic operations to obtain the final structure. Extensions to more than two variables and exploitation of other properties of operations await further investigation.

7.2 Synthesis for multiple DDGs

Thus far the focus of this work has been on the synthesis for a single DDG, in which nodes are exclusively related by data dependencies. In such DDGs, by convention, communication nodes interpose between every pair of computational nodes to depict the transfer of data; two computational nodes are never connected directly with an edge.

Herein is proposed an extension to the *semantics* of an edge without changing its *syntax*. An edge spanning two computational nodes will be assumed to imply a control dependency rather than a data dependency. Let this form of dependency be referred to as a *count dependency*. Such a need to express the algorithm in terms of count dependencies may occur in many

cases. For example, a DDG for a computation with two independent input sources, since it must use a single start node, must contain count dependencies connecting the start node and the two nodes representing the input sources. Another example is in expressing algorithms in which, one part of an algorithm executes a certain number of times as often as some other. This latter application of count dependencies is of interest, since it provides a convenient means of obtaining a combined architecture for multiple algorithms. These concepts are demonstrated using a computation that is part of a time-domain beam-forming algorithm [48], a common algorithm in sonar signal processing.

In the beam-forming algorithm, a set of hydrophones samples sonic signals at regular intervals. The samples are identifiable by a hydrophone-time coordinate system, and the computation of a beam involves a set of such hydrophone-time samples. By choosing the appropriate time-coordinates of the signal samples for each hydrophone, the sonar is able to "listen" in a particular direction. A set of samples acquired at a particular time may be used for multiple beams, each corresponding to a distinct listening direction.

During each sampling period, a computation is carried out for each beam direction. Each computation involves multiplying each of the selected set of samples by a *shading coefficient*, and accumulating these to form a beam sample which may be further processed. A DDG for the computation, comprising of only data dependencies, is shown in Figure7.3. It is assumed there that each beam sample is composed of 50 hydrophone-time samples.

One design of the computation involves the use of a RAM to store all the data values. The presence of this RAM forms the external constraint, subject to which the algorithm must be specified.

The computation involved is now divided into the following two dis-

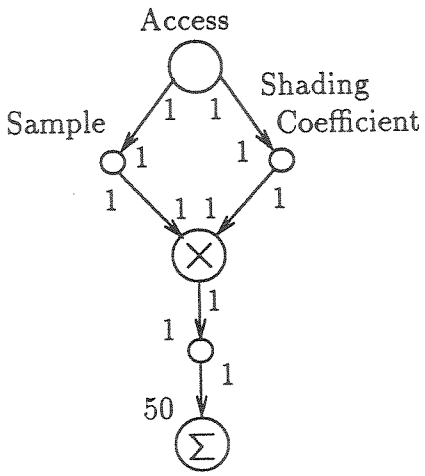


Figure 7.3: A DDG for the beam forming computation using only data dependencies.

connected yet related parts. These two parts can be represented as independent DDGs, but must cooperate to produce the correct output.

In one, an address generator unit produces RAM addresses at its output, choosing the appropriate hydrophone-time samples for a given beam sample. These samples are passed on to the multiply-accumulate unit which computes a running value of the beam sample, also at a location in the RAM.

The second part of the algorithm, outputs the final value of the beam sample after it is computed by accessing the corresponding RAM location.

The DDGs for the two parts are shown in Figure 7.4. The edges with broken lines are the count dependencies. Part A of the algorithm executes 50 times as often as part B.

7.3 Hierarchical design and internally pipelined components

It is most desirable for a synthesis methodology to be hierarchical. This way one may design small custom hardware blocks and use them to con-

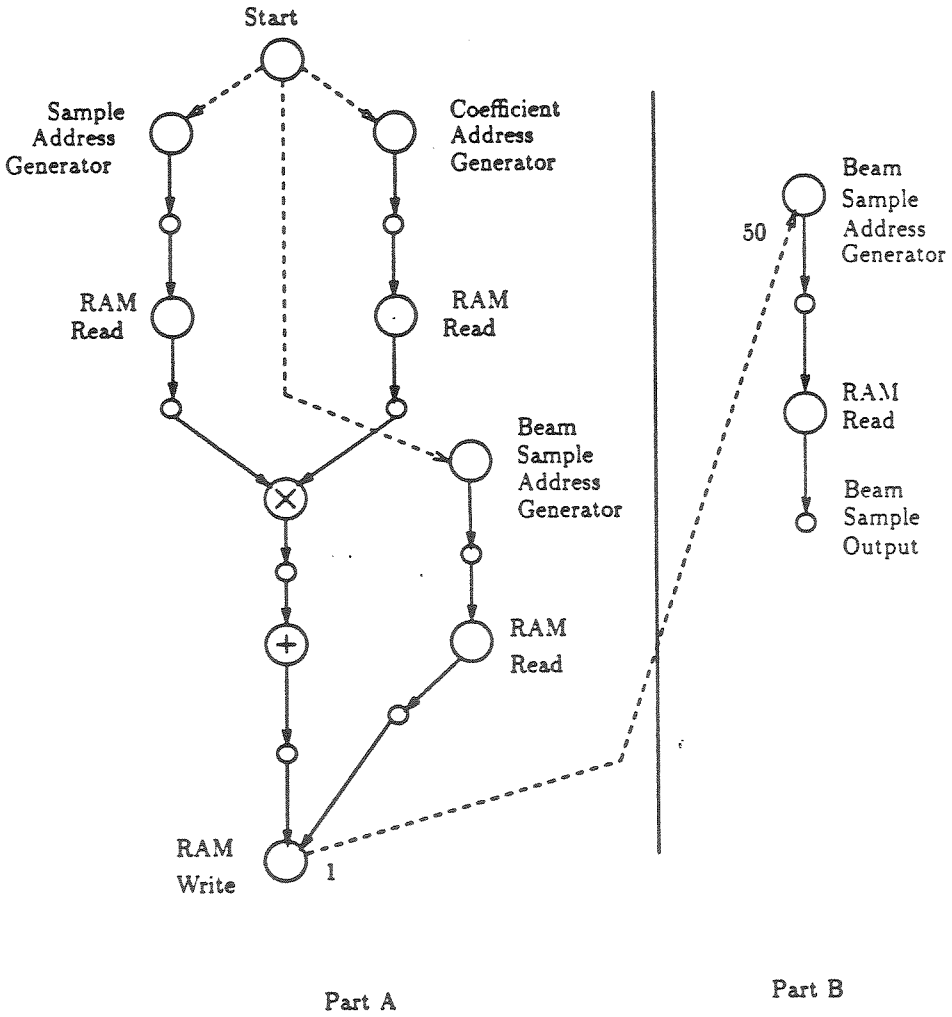


Figure 7.4: The two DDGs for the Beam-former example.

struct larger circuits. These circuits in turn form the building blocks of an even larger system. One major methodological advantage obtained in a hierarchical design is, the design complexity at each level of the hierarchy can be made independent of the size of the system being designed. The new methodology is only partially hierarchical, and making it completely so is an important future goal.

The fundamental requirement for a synthesis methodology to be hierarchical is that it must produce a design that executes according to the execution model of a component. More formally, the methodology must be *closed* with respect to the component execution model.

The new methodology is not closed with respect to its component execution model. In the current execution model, the input and output phases are comprised of a number of *consecutive* clock cycles. On the other hand, in the pipeline design it produces, invocations of the input node are separated by the input latency. Thus the behavior of the resultant architecture does not satisfy the component execution model.

Another, and equivalent, way of expressing the reason for the inability of the new methodology to support hierarchical design is to say that the current execution model is not adequate for modeling the behavior of multi-rate internally pipelined components. However, it is important to note, the model is adequate to describe the behavior of externally single rate pipelined components, which can receive inputs and produce outputs every clock cycle.

One way to support hierarchy within the current methodology is to use the hardware composition technique described in Section 4.7. The internal pipelined characteristic can be shrouded by enforcing that all inputs are accepted consecutively and stored in a buffer. The stored values can then be

fed to the internal pipeline as specified by its execution schedule.

Another avenue is to develop a more general execution model to uniformly support both pipelined and non-pipelined components. Although the general model is not provided here, the following criteria must be satisfied by any candidate model:

- The model must be compatible with the definition of a DDG node.
- Delay definitions in the model must be consistent with those in the current model so that the methodology can set up the TREs, TCEs, and the execution graph to obtain design information.
- The model must contain adequate information to allow its sharing.

Chapter 8

Summary and conclusions

The decreasing cost of hardware has given a boost to the application, design, and production of ASICs. This has given rise to a demand for high level synthesis systems, often referred to as silicon compilers. Many such silicon compilers have be developed in industry and academia.

The growing level of integration naturally leads to design of components which share input and output ports. These components are behaviorally equivalent to multi-rate functions, such as decimation and interpolation, which inherently refer to changes in data rates. Past high-level synthesis approaches have not treated these formally. Neither have communication components been treated formally.

This work has resulted in a theoretical framework to extend the synthesis methodology to multi-rate components, multi-rate functions, and to communication components, and to treat them formally.

A formal definition was given for multi-rate functions and algorithms. Under these definitions, it was shown that systolic implementations of multi-rate algorithms result in inefficient architectures, and that the multi-clock implementation led to efficiency in the architecture. The multi-clock implementation was shown to produce better architectures than the single-clock methodology of Sehwa.

The theoretical framework consists of the representation language of

the DDGs for expressing algorithms, an execution model for multi-rate hardware components — both computational and communication, that is compatible and consistent with the abstract behavior model of the multi-rate functions in a DDG, the structural model for architectures, the analytic techniques, and the synthesis heuristics.

The analytic techniques include the token rate equations, the token count equations, the token and delay equivalence transformations, and the execution graph of the DDG. Using these techniques, the DDG can be analyzed to find the minimum allowable input latency and presence of deadlocks. The token count solutions for sub-DDGs also suggest a method of hierarchically composing multi-rate components.

Several synthesis problems were analyzed and the implication on them of presence of multi-rate was elucidated. In particular, the general problem of obtaining an architecture, with minimum number of computation and communication components, was formulated for a cyclic DDG with a given input latency, as an integer programming problem. This being an NP-hard problem, special cases were further investigated.

It was shown that for acyclic DDGs, a minimum cost architecture, with the cost predicted from the solution of its TREs, can be obtained. An iterative heuristic algorithm for a DDG with a single outer loop was proposed. Finally, a divide-and-conquer heuristic was suggested to reduce the scheduling complexity for DDGs with independent or nested loops. It was shown that for DDGs with nested loops the divide-and-conquer heuristic allows scheduling of loops individually without paying attention to constraints due to other loops. However, the heuristic can be applied only for DDGs which satisfy the criterion of uniform criticality.

A methodology based on the foregoing theoretical framework was developed and demonstrated. To this end, the front-end program, DDGTool, was used to draw DDGs. A fast heuristic scheduler, based on a variation of the list scheduling heuristic, was developed to synthesize architectures with minimum number of computation and communication components for acyclic multi-rate DDGs given in input latency. The scheduler was extended to allow an external specification of schedule cycle length and numbers of components in the architecture.

Two new heuristics were introduced into the implemented scheduler. One was to predefine execution times of operations by components and to schedule the DDG accordingly. This heuristic, although simple to implement and fast to execute, was found to have certain drawbacks, as it had the propensity to produce large graph latency. But since in a pipelined architecture, the graph latency constraints may not be particularly important, the heuristic is deemed quite suitable. The main advantage of this heuristic is to disconnect the assignment and scheduling processes. The second heuristic is applied during the assignment step, during which it helps to reduce the connectivity in the architecture.

In conclusion, the present research has created a unified framework in which to do synthesis of architectures for an extended domain of algorithms using an extended class of components. The framework may be further extended to allow hierarchical design and to facilitate methodological handling of data-dependent computations and multiple related DDGs.

BIBLIOGRAPHY

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1974.

[2] J. R. Allen, Ken Kennedy, Carrie Forterfield, and Joe Warren. Conversion of control dependence to data dependence. In *POPL-83*, pages 177–189, 1983.

[3] John B. Anderson, Tor Aulin, and Carl-Erik Sundberg. *Digital Phase Modulation.* Plenum Press, 1986.

[4] M. Annaratone et al. The Warp computer: Architecture, implementation and performance. *IEEE Transactions in Computers*, pages 1523–1538, December 1987.

[5] Arvind and David E. Culler. Dataflow architectures. In *Annual Reviews in Computer Sciences*, pages 1:225–53, 1986.

[6] M. R. Barbacci. Instruction set processor specification. *IEEE Transactions in Computers*, pages 24–40, January 1981.

[7] R. K. Bryton et al. The Yorktown Silicon Compiler. In Daniel D. Gajski, editor, *Silicon Compilation*, pages 204–310. Addison-Wesley Publishing Company, 1988.

[8] K. M. Chandy. The analysis and solutions for general queueing networks. In *Proceedings of the Sixth Annual Princeton Conference on Information Sciences and Systems*, 1972.

[9] Marina Chen. A synthesis method for systolic designs. Research Report YALEU-DCS-RR-334, Department of Computer Science, Yale University, 1985.

[10] Edward G. Coffman and Peter J. Denning. *Operating Systems Theory.* Prentice-Hall, Inc., 1973.

[11] Editor Daniel D. Gajski. *Silicon Compilation.* Addison-Wesley Publishing Company, 1988.

[12] Alan L. Davis and Robert M. Keller. Data flow program graphs. *Computer*, pages 26–41, February 1982.

[13] H. DeMan, J. Rabaey, P. Six, and L. Clasaen. Cathedral-II: A silicon compiler for digital signal processing. *IEEE Design & Test of Computers*, pages 13–25, December 1986.

[14] Jack B. Dennis. Data flow supercomputers. *Computer*, pages 48–56, November 1980.

[15] Sanjay R. Deshpande, Cheng-Liang Lin, and Mukund Belliappa. DDG-Tool: A graphics device for drawing Data Dependency Graphs. Technical report, Department of Computer Science, University of Texas at Austin, 1990.

[16] G. Estrin and R. Turn. Automatic assignment of computations in a variable structure computer system. *IEEE Transactions in Computers*, pages 755–773, December 1963.

[17] Alfred Fettweis. Realizability of digital filter networks. *Arch. Elek. Ubertragung*, pages 90–96, February 1976.

[18] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H Freeman and Company, 1979.

[19] S. M. German and K. J. Liberherr. Zeus: A language for expressing algorithms in hardware. *Computer*, February 1985.

[20] E. F. Girczyc. Loop winding — a data flow approach to functional pipelining. In *Proceedings of the International Symposium on Circuits and Systems*, pages 382–385, 1987.

[21] Baher S. Haroun and Mohamed I. Elmasry. Architectural synthesis for DSP silicon compilers. *IEEE Transactions on Computer-Aided Design*, pages 431–447, April 1989.

[22] C. Y. Hitchcock and D. E. Thomas. A method of automatic data path synthesis. In *IEEE Design Automation Conference*, pages 484–489, 1983.

[23] E. Horowitz and S. Sahni. *Algorithms: Design and Analysis.* Computer Science Press, 1978.

[24] Chua-Huang Huang and Christian Lengauer. An incrementally mechanical development of systolic solutions to the algebraic path problem. Technical Report TR-86-28, Department of Computer Science, University of Texas, 1986.

[25] R. Jain, F. Catthoor, J. Vanhoof, B. DeLoore, G. Goossens, N. Goncalvez, L. Claesen, J. Van Ginderdeuren, and H. DeMan. Custom design of a VLSI PCM-FDM transmultiplxer from system specification. *IEEE Journal of Solid State Circuits*, pages 73–85, February 1986.

[26] C. S. Jhon, G. E. Sobelman, and D. E. Krekelberg. Silicon compilation based on a data-flow paradigm. *IEEE Circuits and Devices Magazine*, pages 21–28, May 1985.

[27] D. Johannsen. Bristle blocks: A silicon compiler. In *IEEE Design Automation Conference*, pages 310–313, 1979.

[28] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal of Applied Mathematics*, pages 1390–1411, November 1966.

[29] Marc T. Kaufman. An almost-optimal algorithm for the assembly line scheduling problem. *IEEE Transactions in Computers*, pages 1169–1174, November 1974.

[30] R. M. Keller, G. Lindstrom, and S. S. Patil. Data-flow concepts for hardware design. In *COMPCON 80*, pages 105–111, 1980.

[31] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal, Vol 49, no. 2*, pages 291–307, February 1970.

[32] H. T. Kung. Why systolic architectures? *Computer Magazine*, pages 37–46, January 1982.

[33] S. Y. Kung, H. Whitehouse, and T. Kailath. *VLSI & Modern Signal Processing.* Prentice-Hall, 1985.

[34] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions in Computers*, pages 24–35, January 1987.

[35] C. E. Leiserson and J. B. Saxe. Optimizing synchronous systems. In *22nd IEEE Symposium on Foundations of Computer Science*, pages 23–36, 1981.

[36] Charles Eric Leiserson. *Area-Efficient VLSI Computation.* PhD thesis, Department of Computer Science, Carnegie Mellon University, 1981.

[37] Horng-Dar Lin and David G. Messerschmitt. Transforming the data dependencies of data flow graphs. In *International Symposium on Circuits and Systems*, pages 408–413, 1989.

[38] M. C. McFarland. Bud: Bottom-up design of digital systems. In *IEEE Design Automation Conference*, pages 474–479, 1986.

[39] Giovanni De Micheli and David C. Ku. Hercules - a system for high-level synthesis. In *IEEE Design Automation Conference*, pages 483–488, 1988.

[40] Muroga. *VLSI System Design.* Wiley, 1982.

[41] S. Note, J. Van Meerbergen, F. Catthoor, and H. DeMan. Automatic synthesis of a high speed cordic algorithm with the Cathedral-III compilation system. In *IEEE International Symposium on Circuits and Systems*, pages 581–584, 1988.

[42] B. M. Pangrle and D. D. Gajski. Slicer: A state synthesizer for intelligent silicon compilation. In *Proceedings of the IEEE International Conference on Computer Design*, 1987.

[43] C. M. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall Inc., 1982.

[44] Nohbyung Park and Alice Parker. Sehwa: A program for synthesis of pipelines. In *IEEE Design Automation Conference*, pages 454–460, 1986.

[45] A. C. Parker, J. Pizzaro, and M. Milnar. MAHA: A program for data path synthesis. In *IEEE Design Automation Conference*, pages 461–466, 1986.

[46] P. G. Paulin, J. P. Knight, and E. F. Girczyc. HAL: A multi-paradigm approach to automatic data path synthesis. In *IEEE Design Automation Conference*, pages 263–270, 1986.

[47] Pierre P. Paulin and John P. Knight. Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Transactions on Computer Aided Design*, pages 661–679, June 1989.

[48] Roger G. Pridham and Ronald A. Mucci. A novel approach to digital beamforming. *Journal of the Acoustical Society of America*, pages 425–434, February 1978.

[49] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrence equations. In *Proc. 11th Ann. Int. Symp. on Computer Architecture*, pages 208–214, 1984.

[50] I. V. Ramakrishnan. *A Theory Of Mapping Program Graphs Onto Linear Arrays.* PhD thesis, University of Texas at Austin, 1983.

[51] Markku Renfors and Yrjo Neuvo. The maximum sampling rate of digital filters under hardware speed constraint. *IEEE Transactions on Circuits and Systems*, pages 196–202, March 1981.

[52] David A. Schwartz. *Synchronous Multiprocessor Realizations of Shift-Invariant Flow Graphs.* PhD thesis, School of Electrical Engineering, Georgia Institute of Technology, 1985.

[53] M. Shahdad. An overview of VHDL language and technology. In *IEEE Design Automation Conference*, pages 320–326, 1986.

[54] E. A. Snow. *Automation of Module Set Independent Register Transfer Level Design.* PhD thesis, Carnegie-Melon University, 1978.

[55] Gilbert Strang. *Linear Algebra and Its Applications.* Academic Press, 1980.

[56] Texas Instruments. *TTL Databook*, 1988.

[57] D. E. Thomas et al. The System Architect's Workbench. In *IEEE Design Automation Conference*, pages 337–343, 1988.

[58] H. Trickey. Flamel: A high-level hardware compiler. *IEEE Transactions on Computer-Aided Design*, pages 259–269, March 1987.

[59] C. Tseng and D. P. Siewiorek. Automated synthesis of data paths in digital systems. *IEEE Transactions on Computer-Aided Design*, pages 379–395, July 1986.

[60] Robert A. Walker and Donald E. Thomas. Behavioral transformation for algorithmic level IC design. *IEEE Transactions on Computer-Aided Design*, pages 1115–1128, October 1989.

VITA

Sanjay Raghunath Deshpande, son of Nisha and Raghunath Deshpande, was born on 13th September, 1956, in Bombay, India. He completed his education leading to the Secondary School Certificate Examination in Indian Education Society's English Medium School, Bombay, in 1972. In 1979 he received a Bachelor of Technology degree in Electrical Engineering from Indian Institute of Technology, Bombay. In 1982, he received a Master's Degree in Computer Science from the University of Texas at Austin.

Permanent address: 3543 Greystone, #1043
Austin, Texas 78731

This dissertation was typeset[1] with LaTeX by the author.

[1] LaTeX document preparation system was developed by Leslie Lamport as a special version of Donald Knuth's TeX program for computer typesetting. TeX is a trademark of the American Mathematical Society. The LaTeX macro package for The University of Texas at Austin dissertation format was written by Khe-Sing The.