High-Level Synthesis of VLSI Circuits

Ping F. YEUNG

Doctor of Philosophy

University of Edinburgh

August 1992

(Graduation Date August 1992)

Acknowledgements

First to my supervisor, Dr. David Rees, for his support and advice which made this work possible.

To Dr. Beth Benmamane, my industrial supervisor, for her patient and careful review of this work.

To Dr. Alex Deas for arranging financial support from the United Kingdom Atomic Energy Authority.

To the Silicon Architecture Research Initiative led by Prof. Peter Denyer for providing me with the initial inspiration.

To our dedicated computing officers whose hard work provides excellent facilities and keeps the system running smoothly.

Finally, the most important, to my family who bring happiness and comfort during this hard working period.

Declaration

I declare that this dissertation was composed by myself and that all the work it describes within is entirely my own except where indicated.

[redacted]

UNIVERSITY OF EDINBURGH

ABSTRACT OF THESIS

(Regulation 3.5.10)

Name of Candidate: **Ping Fai YEUNG**

Address: [redacted]

Degree: **Doctor of Philosophy**

Title of Thesis: **High-Level Synthesis of VLSI Circuits**

No. of words in the main text of Thesis: 75000

Following the widespread acceptance and application of logic synthesis, we are on the way to establishing synthesis metholdologies which can handle higher levels of abstraction. High-level synthesis is the focal point. It should be able to take a behavioural description of the design, a set of constraints and goals, then construct a structural implementation that performs the circuit function while satisfying the constraints.

In order to ensure a smooth transformation and mapping of high-level description onto hardware, a new strategy for high-level synthesis, flexibility damping, is introduced. It allows a large design space to be explored progressively and systematically. It facilitates the propagation of constraints and helps the introduction of user-specified information. To carry out the strategy, two algorithms, resource restricted scheduling and integrated concurrent mapping are developed. Resource restricted scheduling handles complex control structures and schedules operations across basic blocks in order to utilise all the available resources. After the scheduling has established the flexibility of the abstract elements, concurrent mapping is performed to bind operations, storage, and communications onto functional units, register files and buses concurrently. By considering all the resources at the same time, this mapping process ensures an overall minimum cost of implementation.

Contents

Chapter 1

A New Era

1.1 Introduction

Increasing circuit complexity and time-to-market pressures are gradually forcing designers to take on a whole new approach to projects. Attention has moved towards top-down design methodology at the behavioural level. Instead of concentrating on the structure of a circuit, more effort is made to ensure the correct behaviour of the system as a whole. It is estimated that by 1994, one fifth of the design cycle will be devoted to architectural and behavioural level for 90% of all designs implemented. In the coming decade, the focus of *Electronic Design Automation* (EDA) will shift from the chip design level to system level. This up-shift of the design hierarchy is made possible by the growing acceptance of *Hardware Description Languages* (HDLs) and the rapid development of the state-of-art design synthesis tools.

1.2 Top-Down Design Paradigm

In a bottom-up approach, the designer is more concerned with how the circuit is to be constructed and is involved at the structural level throughout the design. For this approach, verification at the structural level is required to determine the correctness of a design. This means a complete gate-level modelling of the design. In addition, as the focus is on the implementation instead of the behavioural aspects of the design, the complexity increases and manipulation becomes extremely difficult. As a result, costly errors are often left undetected until it is too late.

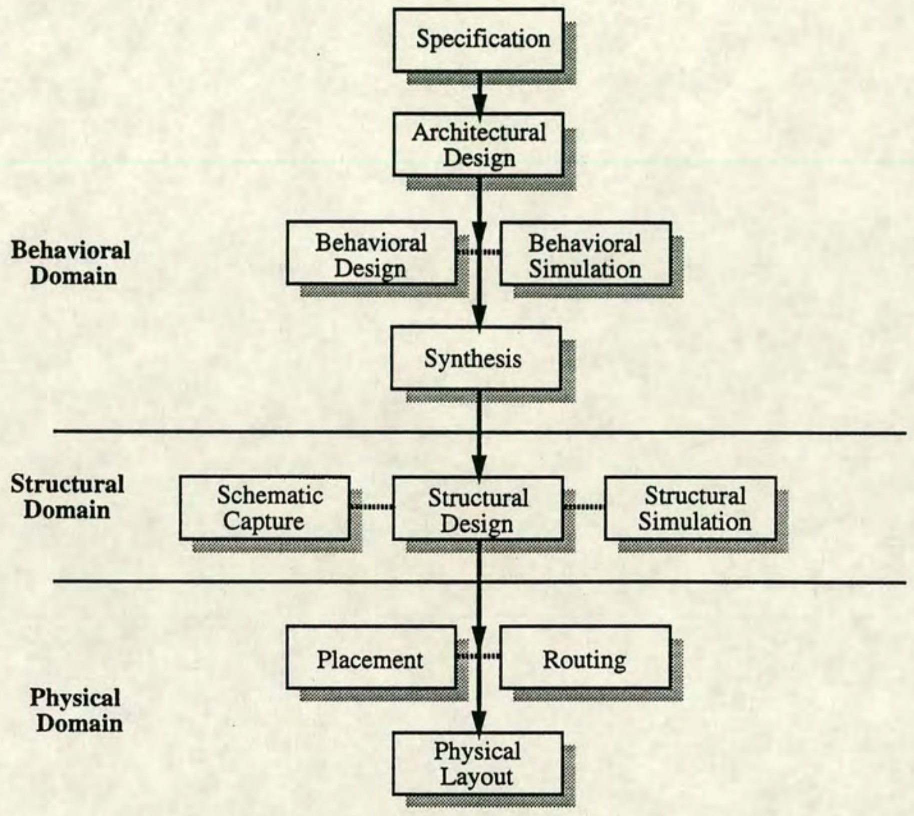


Figure 1.1: Top-Down Design, A New Era

The development of HDLs has smoothed the way to top-down design methodology, and enables designers to describe their system at an abstract, behavioural level rather than dealing with a detailed gate-level description. The aim of top-down design is to carry a design from specification through behaviour to a structural level, which can then be optimized for gates and finally physically realized, figure 1.1. Corresponding to figure 1.2, top-down design means to proceed from the upper left-hand to the lower right-hand corner. *Domains* distinguish between the different descriptions of a digital system, while *levels* define the hierarchical refinement of it. The result of this two dimensional plane is the *hierarchically structured design space* [ZIMM86].

Level	Behavioural Domain	Structural Domain	Physical Domain
System	Performance Specification	CPUs, memories, controllers, buses, etc	Physical partition of chips, modules, cards, boards, subsystems.
Algorithm	Procedural behaviour, manipulation of data structures.	Data structures, procedural partitions, and hardware modules.	Physical connectivity of partitions and clusters
Register Transfer	Concurrent operations, register transfers, state sequencing.	Functional units: ALUs, muxs, registers; and control units.	Macro cells, floorplanning.
Logic	Boolean equations, finite state machines.	gates, flip-flop, and latches.	Cells, module plans.
Circuit	Equations, transfer functions, and timing.	Transistors, capacitors, resistors.	Geometrical layout

Figure 1.2: Levels and domains of design representation in an HDL

1.2.1 Levels

The levels of representation:

- **The System Level**
  This is the specification which includes the behaviour and the performance requirements of the design. These requirements define the range of design flexibility and are usually described in abstract statements. They include cost, clock frequency, power dissipation, and production volume (affected by the market size), sometimes they also include die size (controlled by package size), implementation style, process technology, market (industrial, military), etc.

- **The Algorithm Level**
  This seeks to relate the set of system inputs to the desired outputs. Specification at this level usually makes use of hardware description languages, many of which are well developed and standardized. The behavioural specification should be purely functional. There should be no implication of structural information within the behavioural domain, but in fact, structural bias is present in many popular design languages.

- **The Register Transfer Level**
  This deals with large blocks, such as memories, logic blocks, arithmetic logic

units, together with the interconnections between them. At this level, various aspects of parallelism are explored. Area-time trade-offs are investigated to tailor the implementation towards the design specification.

- **The Logic Level**
  Blocks made up of the architectural units are constructed. Their functionalities are specified in the finest detail with boolean logic. There are many representations of these boolean functions; truth tables, logic equations, gate netlist after mapping, etc.

- **The Circuit Level**
  It is at this level within the overall hierarchy where the implementation is bound to the target technology. Choices of underlying technology like CMOS or ECL, circuit style, static or dynamic, clock methodologies, and the device parameters are all important.

1.2.2 Domains

- **The Behavioural Domain**
  This concerns the purely functional aspects of the design. Attention is concentrated on the behaviour of the design instead of on how it is to be implemented. The description at each level relates the behaviour of the outputs to the inputs. Besides the functionality of the design, there is also a set of performance specifications such as cost, area and timing which have to be satisfied.

- **The Structural Domain**
  This includes schematics, netlists where the primitives are modules and nets. The functional representation is realized by hardware components. As shown in figure 1.2, each level has its own basic elements; from CPUs, memories in the system level to transistors, resistors in the circuit level.

- **The Physical Domain**
  In this domain, the design is to be realized in silicon. It relates closely to the target technology. The main concern in this domain is with the layout geometries; structure-to-geometry mapping of the hardware structures onto

silicon. The process of transforming a design from the structural domain to the final layout will involve partitioning, floorplanning, cell placement, topology layout, and routing.

Figure 1.3 illustrates the generally accepted hierarchical levels, the abstraction they represent, and the supporting *Computer Aided Design* (CAD) tools they require.

Hierarchy Level	Abstraction	Supporting Tools
Algorithm	space-time behaviour as instruction, timing and pin assignment specifications	flowcharts, diagrams, high-level languages
Architecture	global organization of functional entities	HDLs, floorplanning, block diagrams, areas and clock cycles estimator
Register transfer	binding of data flow to functional modules, microinstructions	synthesis, simulation, verification, and test analysis programs; programs for evaluating resources utilizations
Functional modules	primitive operations and control methods	libraries, module generators, schematic entry, test generation programs
Logic	Boolean function of gate circuits	schematic entry, simulation, and verification programs, synthesis programs, PLA tools
Circuit	electrical properties of transistor circuits	RC extraction programs, timing verification. electrical analysis programs
Layout	geometric constraints	Layout editor/compactor, netlist extractor, DRC, floorplanning, placement and routing

Figure 1.3: Hierarchy Levels, Abstractions, and Supporting CAE Tools [LEUN88].

1.3 Design Phases

For the top-down methodology, the design process can be divided into several design phases, figure 1.4. Namely:

1. Architectural (System Level) Design,
2. Structural (RTL) Design,
3. Logic Design, and
4. Layout Design.

With the top-down design approach, the design process starts at the highest level of system definition, and the specifications are prescribed according to the user's needs in terms of functional and performance requirements.

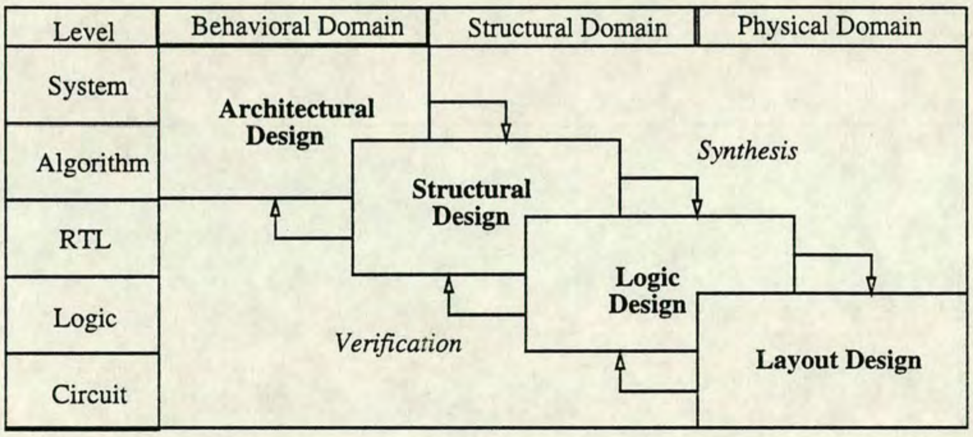


Figure 1.4: The Design Phases in Top-down Design Methodology

1.3.1 Architectural Design

This phase is the most creative and has the greatest impact on the design. Design freedom is restricted only by the behaviour and the performance specifications of the design. The design is a typical planning process. Important parameters such as chip area, execution speed, and production cost can only be estimated. Accurate information will not be available until the design has reached the final stage. Therefore, techniques such as modelling, partitioning or simulation which can help to estimate these parameters, are particularly important.

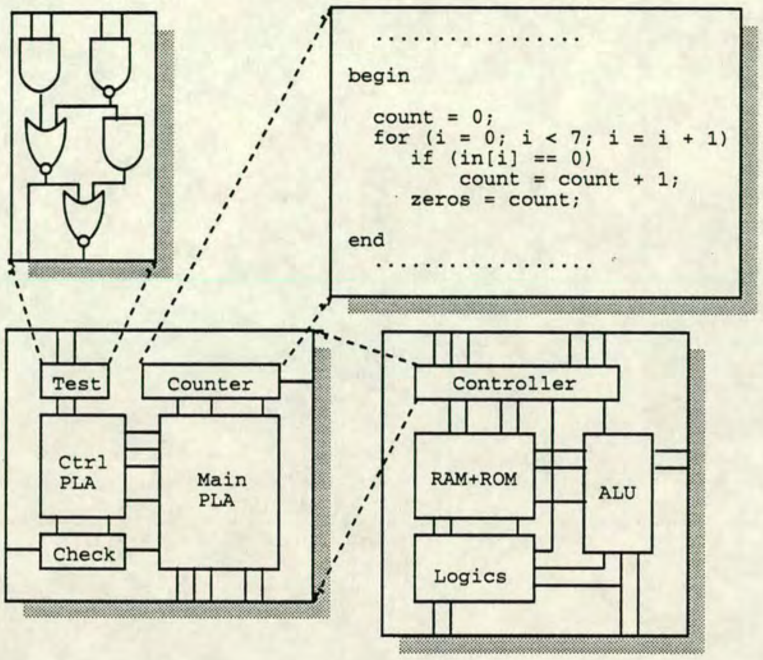
Ideally, at the behavioural level design should be conducted on the basis that the functional and timing behaviour are the most important. We should not worry too much about the constraints imposed by the structural and physical domains. The realization of this paradigm is for designers to:

- create a behavioural model from the specifications representing the system architecture in terms of logical components and relationships. This behavioural model solidifies the specification of the design and helps to iron out functionality problems.
- evaluate the design to ensure that it conforms to the functional and performance specifications.

1.3.2 Structural Design

After the specification of the design is determined, we can move to the structural design phase. It is an iterative decomposition process which resolves the design gradually to a finer model of behaviour and functionality. During the process, physical models are selected or synthesized to implement the logical components. Since design automation tools for this phase are still in the development stage, schematic capture is the most commonly used CAD tool for the job. Although inefficient in some cases, the designer has full control of the design implemention.

As opposed to architectural design, structural design involves assigning structural models to each logical primitive to build up the schematic of the design. As a result, lower level implementation details like the availability of functional units or the performance of the models needs to be considered. After the structural netlist has been laid down, simulation and performance analysis can be performed, e.g. figure 1.5. For simulation, a mixed-level approach is often employed. This is because the representation of the design is not homogeneous. Very often, descriptions at different levels are involved; some implemented modules are in the structural level while others are in the behavioural level. It is clear that there is no straightforward top-down process. After modules are implemented, they are analyzed and the performance information is fed back to the design process. This



```
  ..................
begin

  count = 0;
  for (i = 0; i < 7; i = i + 1)
     if (in[i] == 0)
         count = count + 1;
     zeros = count;

end
  ..................
```

Figure 1.5: Mix-Level Design Simulation

facilitates the continuous transitions of design levels and domains. These transitions provide important information which aid decision making, iterate to correct design mistakes, and tailor the design to the aimed specifications.

1.3.3 Logic Design

Logic design involves transforming the structure modules into gate level implementation. The rapid development of logic synthesis tools has proved to be very helpful at this phase. Logic synthesis is the process of translating a design from a representation, such as FSM representations, boolean equations, and truth tables into an optimal gate-level implementation. The advantages include:

- The ability to synthesize logic so that the speed and area constraints are met.
- The ability to optimize logic for a chosen technology; helping to keep the design independent of technology until the final stage.

- Using a set of constraints, the synthesis process can be directed towards a desired solution.
- Fully automatic and with the speed advantage, it allows a large design space to be examined before the best suited solution is chosen.

In the past, the technology mapping process usually took up a lot of time and effort. In addition to the different selection of gates available in different libraries, the area, timing and power characteristics of each gate varies with the fabrication technology. Determining gates which best meet the timing constraints while not being expensive in terms of area is a very complex task. Logic synthesis overcomes this problem with its speed advantage and its optimization tactics. Through the technology mapping step, all the modules in the library will be taken into consideration and their potential explored. Then, results for different technologies can be compared, trade-offs among cost and performance can be made and the most cost-effective design can be selected for the application.

1.3.4 Layout Design

This is a process of laying out the structural gate netlist of the design on silicon. The basic aim is to optimize the design for area, speed and power consumption at the physical level. After years of research, it is a well developed area. A lot of CAD tools are available to aid the design process. Some of them such as module generators, layout synthesis tools can generate silicon layout directly from structural descriptions. Automatic or interactive floorplanning followed by routing can then be carried out to complete the design. Although most of the layout design steps are mechanised, because accurate performance information can only be obtained after a complete layout, iterations are unavoidable. It is this problem which fuels the recent development of timing driven placement and routing tools. After the design has been implemented on silicon, circuit extraction is performed to look into the physical parameters of the final implementation. Back annotation and simulation are performed and the result is compared with the behavioural simulation to confirm the correctness of the design.

As already mentioned, the design process is a mixture of both top-down and bottom-up approaches with probably a lot of local iterations. The top-down methodology will mainly dominate the behavioural and structural domains. That is at the early stage of the design process before an implementation technology is selected. However, once an implementation technology is fixed, a large number of physical constraints will be generated. The design process will now be an upward propagation of design constraints. Special efforts will be required to optimize the design towards the implementation technology and thereby create the most cost effective design.

Top-down design methodology has several major benefits. Since the design is at the architectural/behavioural level, it is technology independent. Implementation technology can be deferred until the final stage of the design process. The design can be retargeted to a different technology to meet a particular market window or market changes, With this migration path, users can upgrade their design from a low volume programmable implementation to a semi-custom gate array technology or vice versa. Top-down design, especially when combined with mixed-level simulation, allows the entire system to be simulated at any level of the design cycle. This ensures the design implementation matches the specifications. Verification can also be done in parallel with the design enabling problems to be discovered before they are locked into silicon.

1.4 Design Synthesis

Between the gap of functional specification and the implementation, there is design synthesis. Generally, the term *synthesis* implies some intelligent computer-aided design tools which are capable of producing gate-level or mask-level implementation of the VLSI circuits from some input description language. Design synthesis is cost efficient and gives a quicker turn-round time [MCFA90]. Its benefits can be summarized as:

1. Since manual design effort accounts for much of the cost of the chip, automating some parts of the design process can lower the cost and shorten the design cycle.

2. As most of the synthesis methods are based on correct-construction, there will be fewer errors and the debugging time can be reduced significantly.

3. A good synthesis system can usually produce different designs to meet different trade-offs between cost, speed, power etc.

4. An automated system can self-document the design process and keep track of the design decisions and compromises.

At present, there are two approaches to design synthesis.

1.4.1 Structural Approach

For the structural synthesis approach, the input description must specify the structure, the subcircuits and the interconnections of the design. It is not the responsibility of the synthesizer to investigate alternative structures which have the same functionality. High-level trade-off decisions are controlled directly by the user. Implementations are generated in the most straight-forward manner. Since design trade-offs by the tool are minimized, different strategies and results can be tried out easily.

1.4.2 Functional Approach

For the functional synthesis approach, the input description usually contains no predictable structural semantics. The important difference between the functional and structural approaches lies in the area of resource allocation. This is a process of assigning structural components to implement the design. In the structural approach, resource allocation is performed explicitly by the designer, while in the functional approach, decisions are made by the tool after experimenting with various alternatives. This advantage allows the designer to concentrate on the functionality of the design instead of worrying about the implementation of it. The synthesizer shifts a large portion of the work from the designer to the tool and hence increases the productivity of the designer. However, because of the diversity of the possible trade-offs, it is difficult to automate the trade-off selection process. Some guidance on the desired trade-offs must be specified by the designer. Ideally, the synthesizer would be able to identify the critical area of the design; iterate automatically and keep on improving until the supplied performance criteria are met or no more improvement is possible.

Nevertheless, this design synthesis approach has shortened the loop of design iteration dramatically. With accurate performance information, suitable alternatives can be experienced before proceeding to lower stages.

1.5 High-Level Synthesis

The goal of high-level synthesis is to construct a circuit from a high-level description. With increasing popularity, logic or gate-level synthesis tools are constructing and optimizing designs from boolean equations, truth tables or netlists. They will be providing the essential bases for high-level synthesis. High-level synthesis, on the other hand, is the main arm which will lead towards system level design automation. Inside this environment, implementations will be generated automatically from the algorithmic level of the behaviour domain. The entry is an abstract functional description of the design. The synthesis task is to take the behavioural description, a set of constraints and goals, then construct a structural implementation that performs the function while satisfying the constraints. The output of high-level synthesis will be at the structural level referring to the components and interconnections that make up the design.

To summarize, the input to the system consists of three parts:

1. The interface of the system with the external environment.
2. The input-output (functional) behaviour exhibited by the design.
3. The constraints and the desired performance or trade-offs involving cost and time. e.g. area, delays etc.

The synthesis process can be represented using *Gajski's Y-Chart*, Figure 1.6. Basically, it is a process of translating a high-level description in the Behavioural Domain to the Structural Domain. Then, through logic synthesis, placement and routing, it reaches the Physical Domain. For most high level synthesis tools, it goes from the algorithmic and register transfer representation to the structural representation which consists of hardware modules, ALUs, multiplexors, registers, etc. Then, structural synthesizers or silicon compilers can be used to generate the mask-level layout.

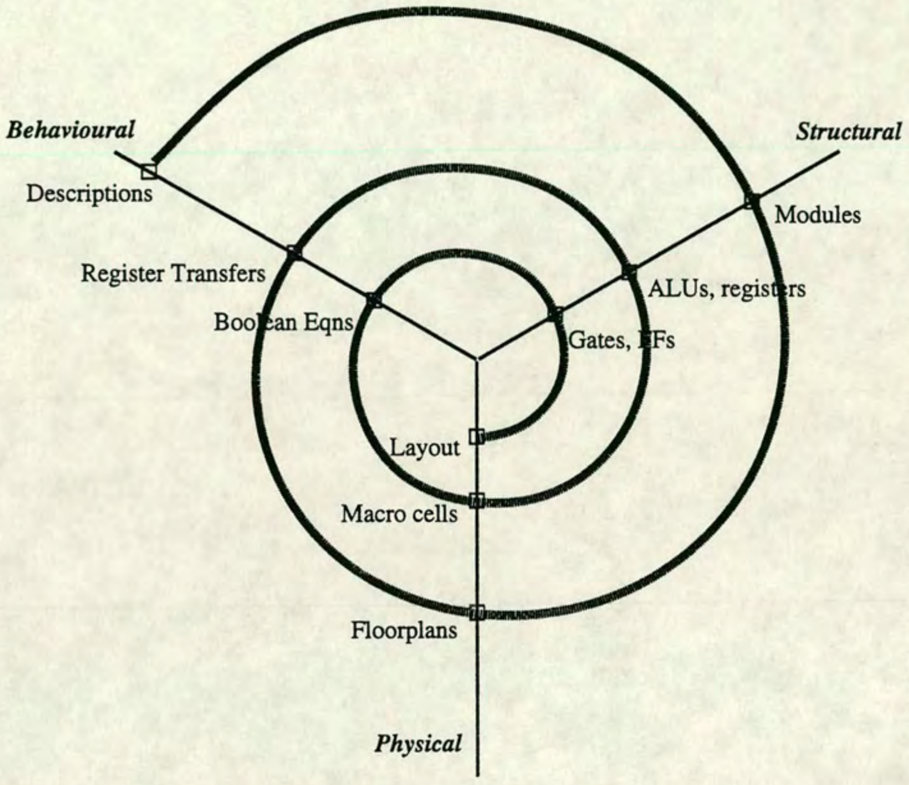


Figure 1.6: Gajski's Y-Chart representation of the synthesis process.

The major tasks in high level synthesis include:

1. **Compilation.**
   The functional description is parsed and compiled into a graph-based representation. For most systems, the internal representation contains both the control and data flow information implied by the specification.

2. **High-level transformations.**
   These include compiler-like optimizations such as dead code elimination, constant propagation, common subexpression elimination, procedure unfolding, loop unfolding and more importantly, hardware oriented transformations. They simplify the representation of the design and facilitate its implementation in hardware.

3. **Data Path Synthesis.**
   The goal is to transform the data path design of the system into its corresponding structural level. Trade-off decisions between the cost of implementation and the speed of the path have to be made. Due to the complexity of this process, it is usually partitioned into the following steps:

   (a) **Operation scheduling.** The data flow graph is partitioned into specific control steps. The aim is to minimize the execution time or the number of control steps needed to implement the design. Very often, there are limitations on the hardware resources.

   (b) **Hardware allocation.** The aim is to minimize the amount of hardware resources required to implement the design. It consists of allocating functional units to execute the operations, storage elements to store the values, and the communication paths to make up the interconnections.

   (c) **Module binding.** This decides how each component of the data path is to be implemented. It includes taking components from a hardware library or synthesizing special customized hardware by logic synthesis.

4. **Control Synthesis.**
   The control hardware is to handle state transitions, control signal generation and interpretation. It can be implemented using a hardwired structure, a single PLA, multiple PLAs, counter-based PLAs, microprogram controller, or even a mixture of all. Which style should be used depends solely on the problem, and a careless selection may result in a very inefficient layout area.

Finally, structural level tools like logic synthesizers and layout generators will be used to complete the design.

1.6 Summary

In this chapter, we have introduced the top-down design paradigm, the design space and the design process for general IC design. The design space is characterized by the five different levels and the behavioural, structural, physical domains, as shown in figure 1.2. They show the transition of a design from functional specification to the final geometrical layout. To drive the design transformation, there is the design process. A clear step-by-step walk-through of the design process is difficult. However, as shown in figure 1.4, it can be divided roughly into several phases.

To transform a design from behaviour domain to physical domain, the most important concept is design synthesis. It facilitates the top-down design process. With high-level synthesis, design can be entered at an abstract level. The tools will be able to perform design trade-offs to satisfy the design constraints. Designers no longer need to worry about the implementation details of a design.

High-level synthesis is the focus of this thesis. After introducing some research systems in Chapter 2, we will detail our synthesis strategy. The high-level synthesis strategy is elaborated module by module. The organization of the chapters is as follows:

- In Chapter 2, a group of high-level synthesis systems are presented. The special features of each system are highlighted. The chapter concludes with an investigation of the scheduling algorithms currently used for some high-level synthesis systems.
- In Chapter 3, the weakness and limitations associated with the previous systems are discussed. A new synthesis strategy, *Flexibility Damping*, is presented.
- In Chapter 4, software compilers and silicon compilers are compared. Then compilation is discussed. The two techniques, *variable renaming* and *expression regrouping* used in the front-end are discussed in detail.
- In Chapter 5, the design representation of our synthesis system is elaborated. Using that internal representation, local and global transformations are presented.
- In Chapter 6, the first scheduling algorithm in the flexibility damping strategy is presented. Using a sophisticated representation, *Resource Restricted Scheduling* can handle designs with complex control structures.

- In Chapter 7, the second and more global scheduling algorithm in the flexibility damping strategy is presented. *Concurrent Scheduling* supports various different architectures. As its name implies, it performs scheduling on operations, storage and communications at the same time. Resource allocation and binding are also considered during the course of scheduling.
- In Chapter 8, elements which are important in the integration of a high-level synthesis system are discussed, and the conclusions of the thesis are presented.

Chapter 2

Current State of Art

In this chapter, we summarize the current state of development of high-level synthesis by looking at some of the interesting design systems. They are:

1.	The Design Automation Assistant	Carnegie-Mellon University
2.	The System Architect's Workbench	Carnegie-Mellon University
3.	The Advanced Design AutoMation	University of Southern California
4.	CATHEDRAL-II	ESPRIT projects 97 and 1058
5.	The Yorktown Silicon Compiler	IBM, Yorktown Heights

Although all of these are high-level synthesis systems, each one has been designed with different objectives in mind. These differences can be identified by:

- the application area, and
- target architecture, which leads to
- the techniques used to explore the design space, and
- the integration of scheduling and allocation

For the wide range of scheduling and allocation algorithms, there is a detailed analysis at the end of the chapter.

2.1 The Design Automation Assistant (DAA)

The Design Automation Assistant (DAA) was developed at Carnegie-Mellon University [KOWA85]. The work could be dated back to the late seventies when the input description, Instruction Set Processor Specification (ISPS), was developed. The aim of the system is to help designers to develop the algorithmic description of the system and interactively add the details required to produce a finished design. The approach is aimed at aiding the designer by producing data paths and

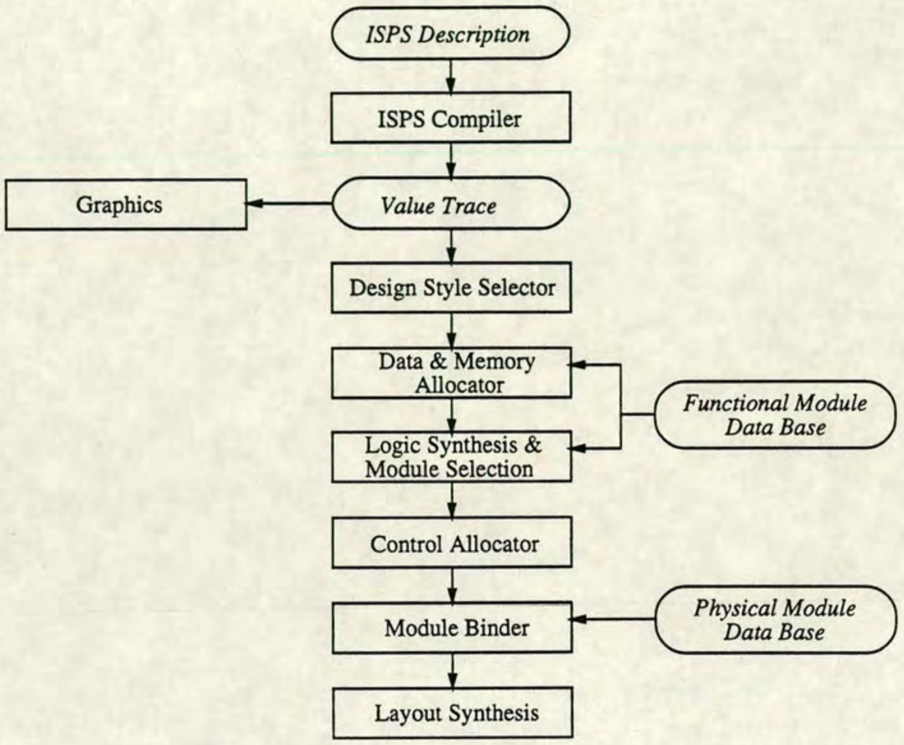


Figure 2.1: The Design Automation Assistant.

control sequences that implement the algorithmic system description within the supplied constraints.

The input to the system includes the ISPS [BARB81] specification of the hardware to be designed, the user's optimization criteria, and a library describing the components available to the design system. First the description is converted into the important internal representation, a directed acyclic graph called the *value trace* (VT) on which various processes can be performed, figure 2.1. They include graphical display of the representation, analysis, transformation to improve the design, partition of the design, control step allocation, data-path allocation, etc. DAA uses a two-pass expert system for data path synthesis. The first pass partitions the design such that data and functions which can efficiently share the same hardware would be within the same partition and the interconnections between the partitions are minimized. Then it assigns components to each element in the data flow graph. The second pass attempts to find a minimum cost implementation of the design by merging components to form the actual structural implementation.

During these two passes, design decisions are made in a top-down manner through a set of rules in the knowledge-based expert systems (KBESs). The KBESs are built up by gathering rules from basic "book knowledge" about synthesis of architectures, interviews with expert designers, improvements and corrections after close examination of many examples. DAA is implemented as a production system via the OPS5 KBES writing system. The system formulates problems by using three major components:

1. The working memory, which consists of a collection of elements describing the current situation.

2. The rule memory consists of a collection of rules and conditional statements, that operate on elements stored in the working memory. These rules enable DAA to synthesize an acceptable design by determining, at each step, whether a certain design extension is within constraints.

3. The rule interpreter which matches the working memory elements against the rule memory, to decide what rules apply in a given situation. The process is repeated until no more rules apply or it is stopped explicitly by a rule.

During the development, DAA has been used to design an IBM System/370, in 47 hours of CPU time on a VAX 11/780 with 6 Mb. of memory. If the backtracking algorithms used in DAA are improved, the CPU time can be dramatically cut down to 4 hours. A comment from the original IBM design team manager shows that the DAA design has exhibited the quality of one of IBM's better designers. The differences between the DAA and the IBM designs are summarized in figure 2.2.

Like many other design systems, DAA is designed to synthesize a microprocessor style architecture with buses. The most distinctive feature of it is the use of an expert system. Therefore, as with human design, the system will continue to grow as more rules are coded. When System/370 was synthesized, DAA used more than 8500 rules. Since the rules encapsulate the knowledge of expert designers, the system exhibited the behaviours and concerns of human design. Because of this concern, floorplanning considerations are embedded in the rules.

	D370	u370
Design	paper design	working chip
Objectives	High performance, technology sensitive independent of power and I/O pins	Strict observance of technology criteria such as number of wired circuits, power, and I/O pins.
ALUs	32-bit, 64-bit, and 68-bit; Binary numbers; hardware for virtual memory, floating point, and multiply	8-bit and 24-bit; Binary and packed-decimal numbers; microcode for virtual memory and multiply
Buses	8-bit, 24-bit and 64-bit; bidirectional out, and bidirectional	Three 8-bit, a 16-bit, two 24-bit; fan-in, fan-
Memories	12-byte buffer; single ported	8-byte buffer; single ported
Registers	Discrete	Memory array

Data from [KOWA85]

Figure 2.2: Differences between the DAA 370(D370) and the IBM System/370(u370).

An encouraging aspect is that connectivity and testability are used to trade-off decisions very early in the design process. However, for some systematic optimizations, the performance is not as good as algorithmic approaches.

2.2 The System Architect's Workbench (SAW)

The System Architect's Workbench (SAW) [THOM90] was also developed at Carnegie Mellon University. Like DAA, the system to be implemented is described in the

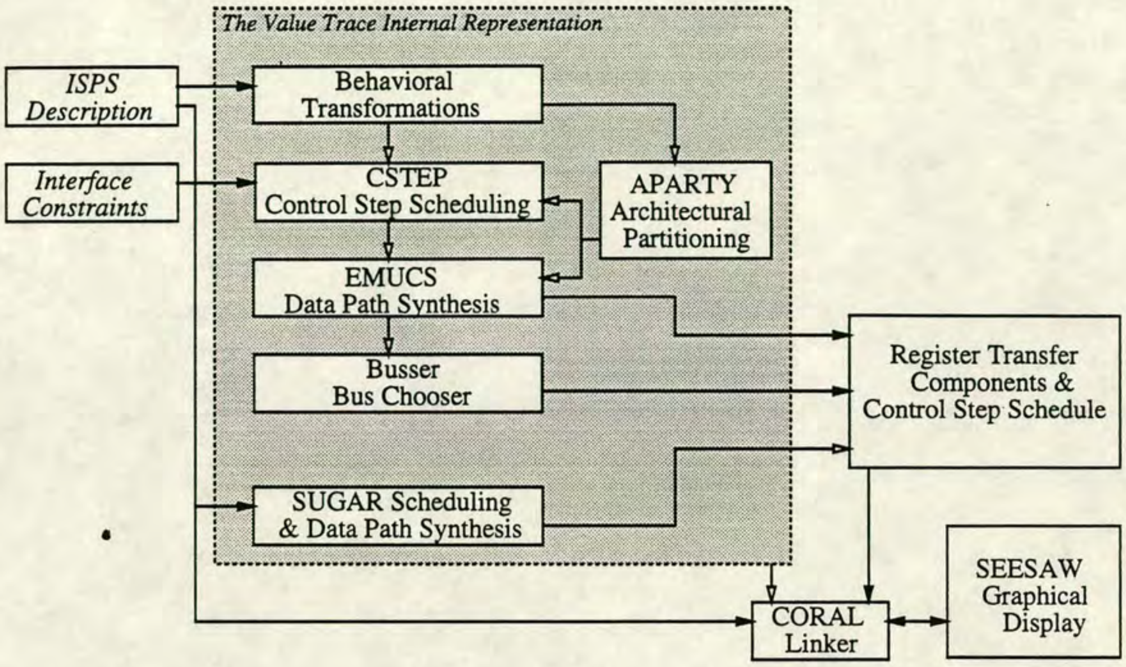

The Value Trace Internal Representation
ISPS Description
Interface Constraints
Behavioral Transformations
CSTEP Control Step Scheduling
APARTY Architectural Partitioning
EMUCS Data Path Synthesis
Busser Bus Chooser
SUGAR Scheduling & Data Path Synthesis
Register Transfer Components & Control Step Schedule
CORAL Linker
SEESAW Graphical Display

Figure 2.3: The System Architect's Workbench (SAW)

ISPS language which is compiled into the value trace (VT). Two synthesis methodologies can then be applied:

1. the target architecture specific approach, SUGAR
   which performs scheduling and data-path synthesis, is shown in the lower portion of figure 2.3. SUGAR is tuned specifically to design microprocessor style architectures and the knowledge involved is integrated into the program. This knowledge includes information about trade-offs and the specific techniques frequently found in commercial microprocessors. For instance, it recognizes subsystems such as the instruction decode unit, condition code logic, branch logic and has knowledge about bus structures commonly used in custom-designed microprocessors. It performs the control schedule based on the data path resources needed, instead of assigning the control steps before data path allocation.

   The synthesis process is divided into several phases. These are:

   – behaviour transformation to remove description inefficiencies,

- control restructuring to allow fast decoding of instructions,
- compiler type flow analysis,
- bus structure selection,
- symbolic register allocation,
- micro-machine code selection,
- assigning control steps to register transfers,
- design improvement by cost/speed tradeoff, and
- replacing symbolic registers by physical registers.

2. the general data-path approach
   which consists of behaviourial transformations, CSTEP – control step scheduling, APARTY – architectural partitioning, EMUCS – data-path synthesis and Busser – bus chooser. This approach is tunable on a per-design basis. A table-driven or knowledge database approach is used where the values in the tables or database, along with the information derived from analysis of previous designs, are used to guide the decision making process. As a result, it is possible for it to synthesize different design styles. As shown in figure 2.3, the implementation flow consists of:

   (a) behavioural and structural transformations on the VT,

   (b) CSTEP to assign operators in the VT to control steps using maximum and minimum timing constraints.

   (c) EMUCS to complete the register-transfer level design,

   (d) Busser to choose the busses to interconnect the data path elements,

   In addition, APARTY is there to provide top-down partitioning information to each of the subtasks. This approach has been used to design interface hardware, small microprocessors and real-time controller hardware.

The CORAL linker maintains the correlations between the initial ISPS description, the VT and the synthesized design, providing a correlated basis for user interrogation, verification, and other applications. The SEESAW graphical display is used to present the various representations to the user and to highlight the relationships between them.

Being specially tailored to deal with processor style design, SUGAR can recognise most processor constructs, like instruction decoding logic, or branch logic. But it is the knowledge about bus structure in processors that gives it the greatest advantage. Both the target specific and the general approach have been used to

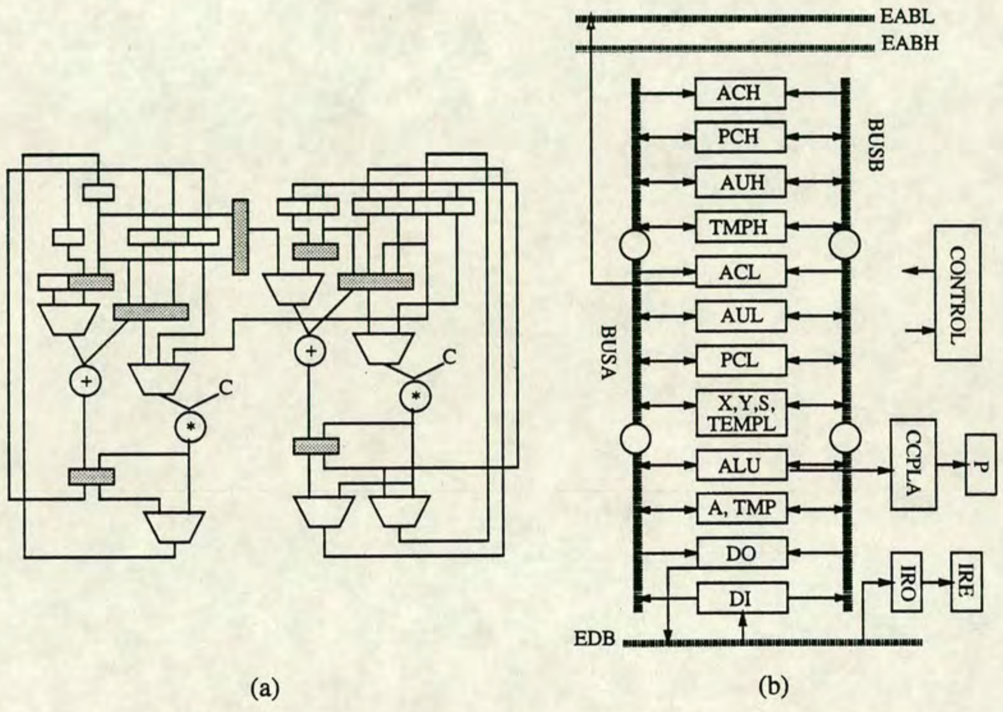


Figure 2.4: Synthesis Result: a. General Approach, b. Specific Approach.

synthesize the M6502 processor. Figure 2.4 demonstrates the differences clearly. And because of this specialisation, SUGAR can produce the design within 2 hours CPU time on a DEC VAXstation II while the general approach takes significantly longer. Compared with DAA, this algorithmic approach proceduces a more optimal solution.

2.3 The Advanced Design AutoMation (ADAM)

This is currently under development in University of Southern California [GRAN85]. ADAM accepts a behavioural description of a digital processor and generates a register-transfer level design. It consists of two major subsystems:

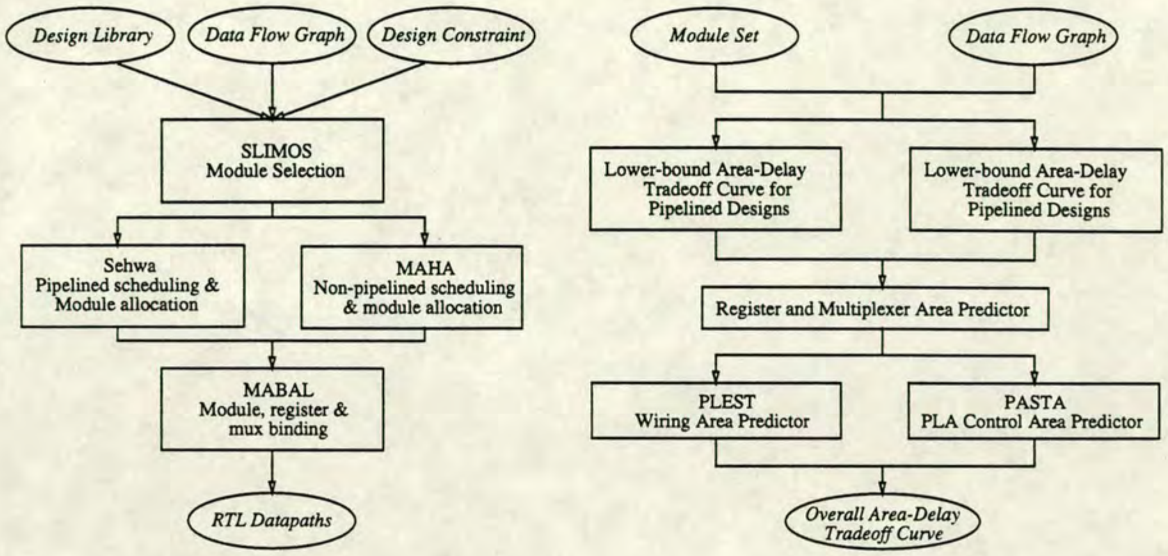

Design Library
Data Flow Graph
Design Constraint
SLIMOS Module Selection
Sehwa Pipelined scheduling & Module allocation
MAHA Non-pipelined scheduling & module allocation
MABAL Module, register & mux binding
RTL Datapaths
Module Set
Data Flow Graph
Lower-bound Area-Delay Tradeoff Curve for Pipelined Designs
Lower-bound Area-Delay Tradeoff Curve for Pipelined Designs
Register and Multiplexer Area Predictor
PLEST Wiring Area Predictor
PASTA PLA Control Area Predictor
Overall Area-Delay Tradeoff Curve

Figure 2.5: The Advanced Design AutoMation (ADAM)

1. the synthesis tools which construct the RTL designs, and
2. the prediction tools which guide the designer in exploring the design space for a good design.

Three inputs are required by the system: a data flow graph representing the behavioural specification, a design library, and a set of design constraints. If an area constraint is given, then the performance is maximized within the area limits. On the other hand, if a performance constraint is given, the area is minimized.

Synthesis Programs

The left hand side of figure 2.5 shows the high level synthesis programs in the ADAM system. Starting from the data flow graph, SLIMOS is used to select the suitable module styles from the library to implement the operations. After that, either MAHA or Sehwa can be called to partition the data flow graph into time steps. MAHA is a non-pipelined scheduling program. It generates schedules with varying quantities of modules and delays, maximizes speed according to area constraints, minimizes area according to time constraint, or provides a set of feasible solutions. Sehwa performs functional pipelining scheduling. It also has the capability to consider conditional branches and resynchronization due to resource conflicts and data dependencies. After the data flow graph has been scheduled, the MABEL program is used to perform functional unit allocation, register allocation, operation/value binding and interconnection allocation. MABEL accepts partial designs, allows intervention and performs module binding with the objective of minimizing the total cost.

Prediction Tools

Starting with a data flow graph and a chosen set of modules, the prediction tools can be used to generate area-delay trade-off curves. They can compute the lower-bound clock cycle and the number of resources for both non-pipelined and pipelined designs. For more accurate estimation, the register and multiplexor predictor, the wiring area predictor, PLEST, and the PLA control area predictor, PASTA, can be invoked to add more information to the lower-bound of the trade-off curve. The area of register and multiplexors are estimated using the input specification, the number of resources and the number of time steps. For wiring area estimation, standard cell placement is assumed. PASTA uses the number of time steps, information about loops and conditions, design style, and the number of registers and multiplexors to predict the area of the controller.
The design data-structure of ADAM is divided into four subspaces:

1. the data flow behavioural subspace,
2. the structural subspace,
3. the physical subspace and
4. the timing and control subspace.

It provides a framework which can integrate a number of heuristic, algorithmic and mathematical techniques for synthesis. A design planner[KNAP86] is also available to aid the designer in interactively selecting synthesis and analysis tasks, in determining which design technique to use and in setting up and monitoring design constraints. Physical information on components is included as this can aid selecting the correct component at the structural level.

Although most synthesis systems have estimation tools, they are usually embedded inside the algorithms. ADAM is the first system which removes this barrier. With these prediction tools, it provides a powerful interactive synthesis environment. The benefits include the ability to reduce the design search space significantly, the ability to locate the most promising design space, and performance estimations on partial designs. On the other hand, in order to provide a more complete system, prediction tools for memory and routabilty are also needed.

2.4 CATHEDRAL-II

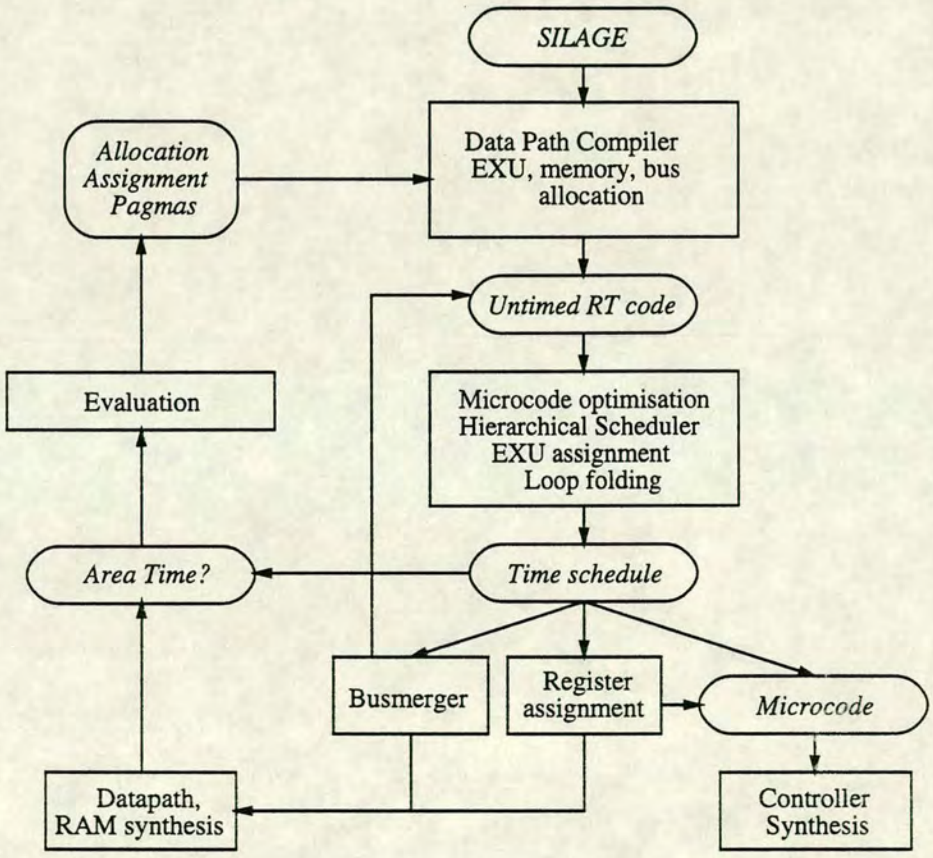

SILAGE
Allocation Assignment Pagmas
Data Path Compiler EXU, memory, bus allocation
Untimed RT code
Evaluation
Microcode optimisation Hierarchical Scheduler EXU assignment Loop folding
Area Time?
Time schedule
Busmerger
Register assignment
Microcode
Datapath, RAM synthesis
Controller Synthesis

Figure 2.6: The CATHEDRAL-II System.

This is a synthesis system specially for multiprocessor digital signal processing systems [DEMA88]. The work is the part of ESPRIT projects 97 and 1058 which involve IMEC, Philips, Siemens, Bell Telephone Manufacturing Company, Silvar-Lisco and Ruhr University of Bochum. The compiler is designed to tie strongly to the target application of a subset of digital signal processing(DSP) algorithms which is highly complex, block-oriented and in the audio- to near video-frequency range. The nature of the algorithms could be architecturally realized by a set of concurrent dedicated bit-parallel processors on a single chip.

The design methodology is called "*meet-in-the-middle design strategy*". The approach is highlighted by the following:

1. The strict separation between system and silicon design levels. The interface is located at the system *functional level* where the silicon primitives, *modules*, are data operators, data storage, controllers and I/O units.
2. The silicon modules are highly reusable and *technology adaptable.* Although they are reusable, they are much more complex than usual standard cells. Special expertise and powerful design environments are needed to realise them.

They believe that an efficient translation to hardware can only be done if it takes into account the properties of the target architecture and that a strong interaction with the designer must be possible. Therefore, the input to CATHEDRAL-II has two separate parts:

- First, the applicative language, SILAGE, is used to describe the behaviour of a signal processing algorithm. The language is designed specially for describing complex DSP algorithms. The main idea is to capture the signal flow graph of the algorithm which is impossible for conventional structured languages like Pascal.
- The second part is *compiler directives* called *pragmas.* This allows designer to make abstract decisions at a very high level by giving *structural hints* to the compiler.

The SILAGE description goes through a *preprocessor.* Besides performing similar tasks common to a software compiler, it also inputs a set of user-defined *allocation* and *assignment* pragmas. Then it comes to the task of generating the data path and the controller for the processors.

1. Data path synthesis: the synthesizer, JACK-THE-MAPPER, translates behavioural primitives into architectural primitives. This includes assigning

primitive SILAGE operations to execution units, defining the bus structure and assigning the variables to register files and memories. In order to be flexible and expandable, it was implemented as a rule-based system. The result is a data path structure and a register-transfer (RT) description of the algorithm.

2. Scheduling/Assignment: the graph-based scheduling tool, ATOMICS, is used to schedule the RT operations to achieve minimum number of cycles. ATOMICS is able to handle repetitive programs and input/output constraints.

3. Register binding/bus merging: After the RT operations are bound to cycles, the dimension of register files and the number of buses can be minimized by life-time analysis and merging.

4. Communication hardware synthesis: to synthesize the interprocessor communication hardware and the central controller by determining the communication protocol, the dimension of the buffer arrays and the timing of the control signals.

After the structure and timing is defined, the Module Generator Environment can be activated to create the final layout of the design.

As distinct from the others, CATHEDRAL-II is designed solely for synthesizing DSP applications. It uses an applicative input description and has a well-defined target architecture. It is designed to implement a multiprocessor system with regular interconnect and synchronous data-passing protocol. Instead of mapping operations onto a large variety of functional units, the target library is relatively small. It has less than ten different execution units (EXUs) but each of them is specially designed and hence very efficient and compact. Providing relatively little choice of functional units, hardware allocation and binding are done before scheduling and cannot be automatically changed afterwards. In addition, instead of using hardwired control logic, two-level microcode is adopted. To increase the architecture freedom, CATHEDRAL-2nd was developed. It supports more flexibility in the composition of EXUs. Basic functional building blocks can be combined into different EXU types to meet the application specific requirements. This has

A complete 801 microprocessor with a streamlined architecture and a 4-stage pipeline has been synthesized in less than 4 hours on an IBM 3090. The performance is the same as a hand design but with 26% more transistors.

2.6 A Study of Scheduling Algorithms

From the systems discussed above, scheduling which assigns computations to control steps is a very influential step in high-level synthesis. As it is also the major topic of this thesis, we are going to have an in-depth study of various scheduling algorithms. In order to help understanding the differences between them, we have categorized them according to the approaches taken:

1. User-defined approach
2. Transformational approaches which include heuristic transformation, state splitting, simulated annealing and self-organising.
3. Constructive/Iterative approaches which include as-soon-as-possible, as-late-as-possible, list and force-directed scheduling.

2.6.1 User-defined approach

The simplest technique is to let the user do the scheduling. This is used in the early Silc system. However, the combinational explosion reduces the usefulness of this technique.

2.6.2 Transformational approaches

A transformational type of algorithm begins with a default schedule usually either maximally serial or maximally parallel, and applies transformations to obtain other schedules. Transformational algorithms differ in how they choose what transformations to apply.

Heuristic Transformation

One approach to scheduling by transformation is to use heuristics to guide the process. Transformations are chosen that promise to move the design closer to the given constraints or to optimize the objective. This approach is used in YSC [BRAY86] and the CAMAD [PENG86] design system.

State Splitting

The Yorktown Silicon Compiler (YSC) [BRAY86] does allocation and scheduling together. It begins with each operation being done on a separate functional unit and all operations being done in the same control step. Additional control steps are added for loop boundaries, and as required to avoid conflicts over register and memory usage. If there is too much hardware or there are too many operations chained together in the same control step, more control steps are added and the datapath structure is again optimized. This is repeated until the hardware and time constraints are met.

Simulated annealing

In [DEVA89], a hill climbing mechanism is incorporated using an algorithm based on simulated annealing. A complex cost function is used and the problem is modelled as a placement problem, an area where simulated annealing has been very successful. It combines scheduling and allocation, with the cost function being a linear combination of estimated area and schedule time length.

Self organising

The methodology is presented in [HEMA90]. It is inherently parallel in nature, has a hill climbing mechanism and has a built in cost weighting mechanism which allows it to do trade-offs in function unit, register and interconnect requirements based on the hardware cost. It treats the schedule space as a continuous space and all operations influence assignment of an operation to a position in the schedule space. It also evaluates several solutions many of them worse than the previous before settling for a near minimal solution.

2.6.3 Constructive/Iterative approaches

This approach builds up a schedule by adding operations one at a time until all the operations have been scheduled. They differ in how the next operation to be scheduled is chosen and how they determine where to schedule each operation.

As-soon-as-possible (ASAP)

The simplest automatic technique is known as *As Soon As Possible* (ASAP) scheduling. It assumes that the number of functional units has already been specified. This approach was used in CATREE, the Facet-Emerald [TSEN86] systems in the early Design Automation Assistant and also in other systems.

ASAP with restrictions

If the amount of hardware is restricted, the operations are conditionally postponed when there is a resource conflict. This approach was used in the MIMOLA [MARW86], and the Flamel [TRIC87] systems. MIMOLA starts from generating a maximum parallel schedule (ASAP) by analyzing the data dependency among the operations. The model global module allocation problem is modelled as an integer programming problem. The system is capable of handling complex multi-function modules and the clock time is determined by the maximum propagation delay path.

The major limitation of these systems is their inability to explore the design space. The only way a new design can be generated is by modifying the input program or the schedule. Further, it is often the case that more critical operations are blocked by less critical ones, resulting in longer than necessary schedules.

In CADDY [CAMP89A], the datapath is built first, assuming maximal parallelism. This is then optimised, locally and globally, guided by both area constraints and timing. The operations are then scheduled, subject to the constraints imposed by the datapath.

As-late-as-possible (ALAP)

An ALAP scheduling algorithm has been reported in [JAIN91]. To perform scheduling, it takes a data-flow graph, a clock cycle and a set of resources as inputs. The heuristic is based on a combination of ASAP scheduling and list scheduling. The algorithm uses the ASAP and ALAP scheduling value of an operation to determine the priority of it.

List scheduling

In list scheduling, the operations that are ready to be scheduled into the current control step are sorted according to a priority function. Then each operation on the list is placed if it is within the resource constraints, otherwise it is deferred into the next step. When no more operations can be scheduled, the algorithm moves to the next step. Again, available operations are found, sorted and the process is repeated. The priority function varies across the systems using this technique.

a. Path length to block end, BUD [MCFA86]

b. Path length to nearest constraints, urgency, Elf [GIRC85]

c. Mobility, Slicer [PANG87B]

On the other hand, functional unit allocation can be done first, followed by scheduling. In the BUD system, operations are first partitioned into clusters, using a metric that takes into account potential functional unit sharing, interconnect, and parallelism.

For stepwise refinement, the approach is to iterate the whole process, first choosing a resource limit, then scheduling, then changing the limit based on the results of the scheduling, rescheduling and so on until a satisfactory design has been found. This is done under user control in the MIMOLA system and under the guidance of an expert system in Chippe [BREW87].

Critical path first, freedom/mobility

In this type of scheduling, the range of possible control step assignments for each operation is calculated. The operations on the critical path are scheduled first. Then, at each step, the unscheduled operation with the least freedom is considered. Therefore, operations that are more critical are taken care of first, before they are blocked.

MAHA [PARK86] accepts a data-flow graph and a module set. It schedules operations, allocates and binds resources simultaneously by first considering the operations in the critical path and then other operations. The system allocates operations to functional units as it schedules, adding functional units only when it cannot share existing ones. By taking into account the area of each functional unit, it minimizes the total functional area of a design.

Force directed scheduling

Force directed scheduling used in the HAL [PAUL87] system has received a lot of attention. It tries to overcome the problem of locally minimum solutions by taking into account more global effects of assigning an operation to a control step. It performs scheduling within a time-constraint so as to balance the number of functional units required in each control step. As a result, it minimizes the number of operators.

Force-directed scheduling (FDS) takes in a data-flow graph, module delays, a clock cycle and performance constraints. It tries to minimize the number of resources of each type subject to the performance constraints. Before performing scheduling, "force" values are calculated for all operations at all feasible control steps. At each iteration, an operation is assigned to a control step which causes the least increase in overall concurrency of operation, storage and interconnect requirement, weighted in proportion to their hardware cost. A force-directed list scheduling (FDLS) [PAUL89] is also proposed. It employs "force" values to determine the priority of the operations. The aim is to maximize the performance while keeping the resources under constraint.

HAL first generates an initial schedule, then it selects, using an expert system, a set of multi-function ALUs, and finally produces the refined schedule. However, once an operation has been assigned to a control step, its assignment is not reconsidered, there is no hill climbing mechanism.

The global analysis approach of FDS was extended in [CLOU90] to combine scheduling, allocation, and mapping of operations in a single algorithm, called SAM. It is able to schedule the operations in the data flow graph to control steps, allocate the necessary hardware, and map the operations onto specific functional instances. During each cycle, all scheduling, allocation, and mapping options are considered. Then a single operation is selected to be scheduled and mapped, and hardware is allocated if needed. The approach is achieved by adding extra terms into the force equation to take care of the compatibility of operations to individual instances. When calculating the forces, all possible mappings of operations to instances are considered.

2.6.4 Summary

According to [MCFA90], scheduling algorithms can be distinguished in two dimensions:

Algorithms	Independent		Interdependent		Combined
Scheduling and Allocation	scheduling first	datapath first	stepwise refinement	simultaneously	
Transformational:					
Heuristic transformation			CAMAD		
State Splitting				YSC	
Simulated annealing					[DEVA89]
Self-organising					[HEMA90]
Iterative/Constructive:					
As-soon-as-possible	Facet		CMUDA		
ASAP with restrictions	MIMOLA				
	Flamel				
		CADDY			
As-late-as-possible	[JAIN91]				
List scheduling		BUD	DAA		
urgency			Elf		
mobility/freedom	Slicer		Chippe		
Critical path first				MAHA	
Force direct scheduling	FDS,FDLS			HAL	SAM

- the type of scheduling algorithm used, and
- the interaction between scheduling and operator and/or datapath allocation.

2.7 Critical Summary

It should be emphasized that high-level synthesis is not a streamlined process. No matter how good the estimation tools are, the final performance of the implementation is difficult to predict. However, in most of these systems, design trade-offs are made early in the design process, where only incomplete information is available to support the decisions. As a result, the success of the design relies heavily on the prediction ability of the system and the allocation process which determines the hardware requirement of the design.

To ease this problem, some of the systems impose restricted design models or a restricted application area. They enforce a simple target architecture and hence this simplifies the algorithms and allows a clear strategy for meeting the design goals. For example, YSC is tied closely to logic synthesis, DAA is tuned towards microprocessor designs, and Cathedral-II is specialized for DSP applications.

Although some of the systems described take into account constraints like cost and speed, they are usually calculated by summing the module area of the functional units and their delay in the critical path. It is true that for most of them, the approach is to minimise the number or the area of functional units. HAL tries to minimse the count of each functional type while MAHA tries to minimze the total functional area. However, the area and delay overheads resulting from routing, steering logic, memory, etc are neglected. The study in [KUCU90] has emphasized that if the area of the functional unit modules are comparable with area of steering logic modules, the capability to trade between multiplexors and tristate drivers and between the interconnection and functional units will be important. Although seldom will it be true for arithmetic operations, this means that efficient designs may not always have optimal functional unit allocation. In addition, except DAA which has rules to consider floorplanning, early consideration of this is absent from most of the others. However, floorplanning is one of the most important criteria for judging the quality of a design. With the feedback from the physical subspace, future developments of ADAM will include this ability.

From another aspect, most of the systems mentioned use global registers to store the intermediate values. Though this is necessary most of the time and results in better utilization of registers, it can easily cause transmission congestion. To perform an operation on a functional unit, transfers of input values from global registers to the unit before the operation starts, and a transfer afterwards will be required. Local storage in the functional units could be a better alternative in some cases. CATHEDRAL-II explores this alternative with input register files designed in the library modules. However, it fails to consider the trade-off between global and local register storage.

Chapter 3

A Strategy for High-Level Synthesis

3.1 Introduction

High-level synthesis (HLS) is a process taking an input specification of a system, a set of constraints and goals, then finding a structure that implements the behaviour while satisfying the goals and constraints [MCFA90] . The input is at the algorithm level while the output will be at the structural level, usually called register-transfer level (RTL). The high-level synthesis process maps the control/data-flow graph which represents the behaviour to the schedule and structural graph which denotes the hardware [CAMP89C]. This implies the mapping of operational computations, intermediate data values, input/output parameters, description control constructs etc. onto hardware functional units, registers, memory units, buses, steering logic, sequential control units, etc. This can be signified as a "mapping" process from algorithm description to register transfer architecture. During the process, constructs in hardware description are attached to abstract hardwares which, in turn, are assigned to hardware structures, figure 3.1.

At the same time, high-level synthesis is a very creative process. Although in some cases, a fully automatic process may be able to deliver an optimal result, there are enough reasons to expect a compromised process. A designer's expert knowledge of the design is too valuable to be ignored. An approach in which the designer can influence and control the process is very beneficial. However, for many systems, if the designer wants to control the outcome of the synthesis process, he is forced to make absolute decisions on the structural implementation of the design. This has a potentially negative effect because at the high-level, there is little information on the performance trade-offs to support these decisions. In addition, they will heavily bias the system to support the constraints introduced. Inferior implementations may result. A fine-tunable decision making process is necessary to support interactive design influence without introducing negative impacts on the design space.

High-level synthesis has been described as a fast but impractical approach for top-down design [CARL91]. Also, there is always a stage in the synthesis process where the available information is no longer sufficient to make sound decisions. To counter this shortfall, the more lower level information is considered, the more realistic will be the result. Recent research has shown interest in an approach with scheduling and floorplanning [WENG91].

The problem is the explosion of lower level information which does not have immediate significance. We do not want to be overwhelmed and its presence also makes the synthesis problem more complex. But more fundamental is that low-level information cannot be derived before the high-level structures are determined. A framework to coordinate the generation and influx of information is necessary. It should enable low-level details to be derived a step ahead of the synthesis process. They can then be annotated back into the framework to assist the decision making process.

Another difficulty lies in the fact that a particular design at the algorithm level can correspond to a large number of potential implementations at the lower levels. As a design proceeds to descend the hierarchy of design representation, from abstract to structural levels, the space of design alternatives grows. These are the design alternatives which high-level synthesis has to explore in order to find one or a number of satisfactory solutions. Hence, it is useful to recognize the *alignments* between different design representation levels so that abstract objects can be projected consistently from the algorithm level to the register transfer level.

The alignments are recognized as in figure 3.1. Through allocation and binding,

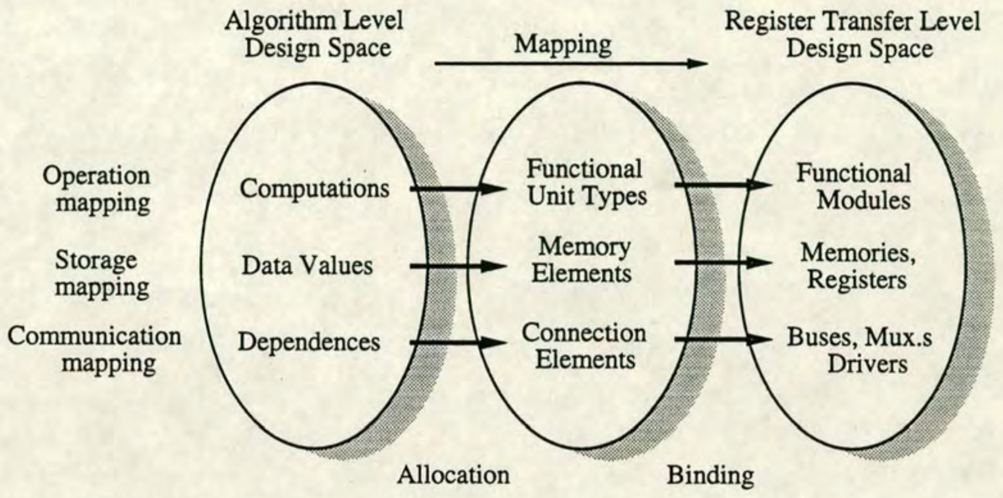


Figure 3.1: Tasks Performed along the Resources Axis.

high-level synthesis has to find an instance in the resource-time design space to implement the high level construct at the register-transfer level. This mapping process can roughly, though not exclusively, be classified into three sub-processes:

- *Operation* subspace mapping,
- *Storage* subspace mapping and
- *Communication* subspace mapping.

In doing so, we place equal importance on the mapping of storage elements and communications as on operation mapping. The three subspaces in the algorithm level can naturally be aligned to the ones in the register transfer level.

$$\begin{array}{rcl} \text{computations} & \Longleftrightarrow & \text{functional modules} \\ \text{data values} & \Longleftrightarrow & \text{memories, registers} \\ \text{dependencies} & \Longleftrightarrow & \text{buses, steering logic} \end{array}$$

This correspondence allows hardware alternatives to be explored efficiently. Basically, the aim of the mapping process is to choose an available module in the pool of resources and an execution time space on the time axis for the implementation of each software construct so that an optimal design can be produced, figure 3.2. The problem which hinders this objective is the inter-relationship between these three subspace mapping processes.

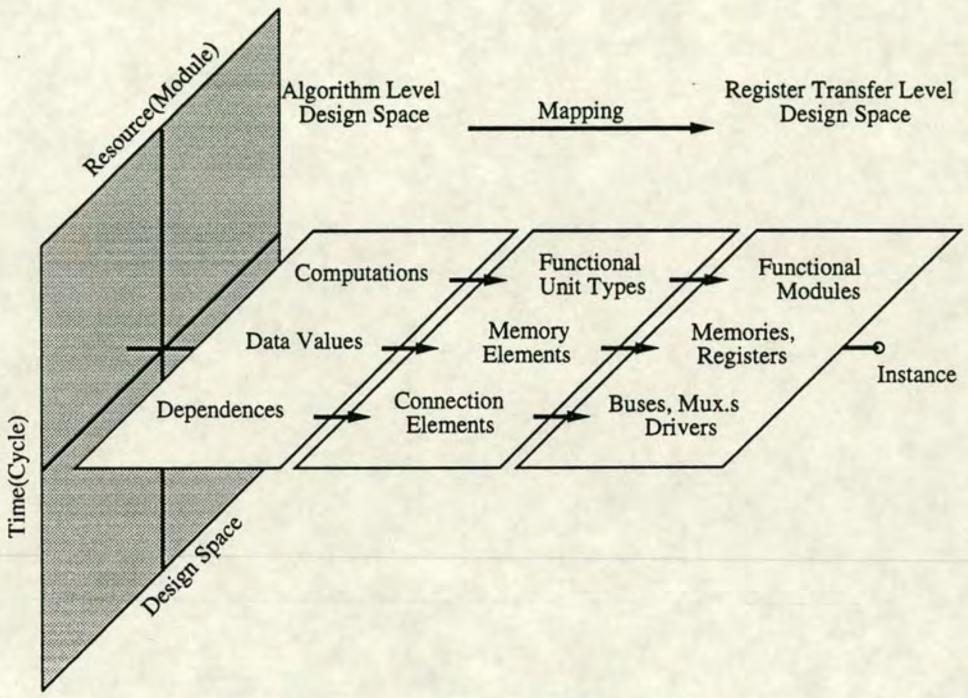


Figure 3.2: The Problem of High-Level Synthesis.

In order to understand these inter-relationship, we need to understand the major tasks and as also their interdependence in high-level synthesis. In the next section, we look at the tasks in high-level synthesis.

3.2 The Tasks in High-Level Synthesis

High-level synthesis consists of a number of major tasks, namely *high-level transformations*, *allocation*, *scheduling* and *binding*. Each has different significance in each mapping process.

1. High-level transformations – These include compiler-like optimizations such as dead code elimination, constant propagation, common subexpression elimination, procedure substitution, loop unfolding and other hardware oriented transformations.

2. Allocation – The aim is to minimize the amount of hardware resources needed. It consists of allocating functional unit types to execute the operations, or memory elements to store the values, or connection elements needed to make up the connections.

3. Scheduling – In this task, computational operations, storage elements and communications are assigned to control steps. The aim is to minimize the execution time or the number of control steps needed to implement the algorithmic description. Very often, there are limitations on the hardware resources.

4. Binding – This decides how each allocated element is to be implemented. Within the three mapping processes, there are binding operations to hardware modules, memory elements to register files, and instantiation of buses, multiplexors and drivers for interconnections. It also includes selection of components from hardware libraries, generating or synthesizing custom hardware.

Due to the interdependent nature of these tasks, it is not necessary that they are executed in any particular order. Actually, in some high-level synthesis systems, this is an iterative process guided by either the human designer or an expert system, while in some others, these tasks are interleaved. We will discuss this in depth in later sections. However, the aims remain the same

- to explore the design space,
- to find a satisfactory *instance* to realize the abstract construct, and
- to optimize the overall design.

An instance is a point in the RTL design space which has well defined time and resource parameters.

3.3 Related Work Overview

Traditionally, mapping of operations, storage elements and communications are performed independently. In Facet [TSEN86], FDS [PAUL89], and some others, allocation, scheduling and binding of operations are performed first, followed by

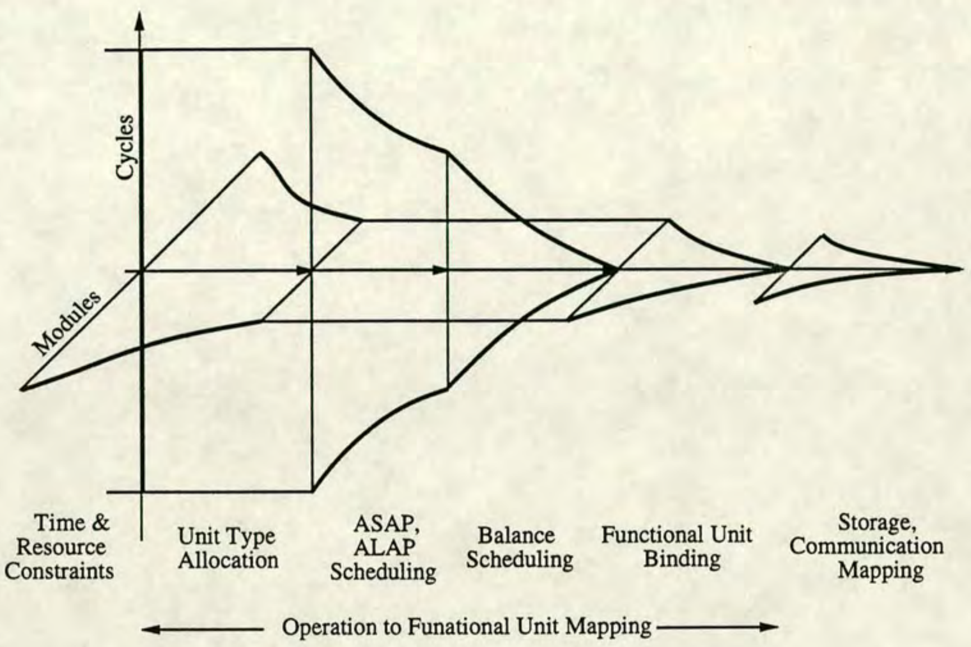


Figure 3.3: Step-by-step Approach.

storage and communication mapping. Figure 3.3 depicts such a step-by-step approach. As shown, each task is executed independently. Each one of them is designed to explore a dedicated portion of the design space. After each execution, based on the results obtained, decisions are made to cut down that portion of the design space. In figure 3.3, allocation explores the area space and by assigning functional unit types to operations, it reduces the resultant choice of modules. ASAP and ALAP scheduling define the *mobility* of the operations. Then, balance scheduling is used to explore the mobility of the operations in order to balance the distribution of the operations. It minimizes the number of functional units with respect to cycle constraints. The outcome is that each operation is assigned to a time period (a cycle or a conservative number of cycles). Although this step-by-step approach is very efficient in cutting down the design space, it has significant drawbacks. The strong interdependence and interactions between the three tasks, allocation, scheduling and binding are ignored. Sub-optimal results are obtained in each step and poor exploration of the design space results.

Because of this disadvantage, algorithms like [DEVA89] and SAM [CLOU90] perform

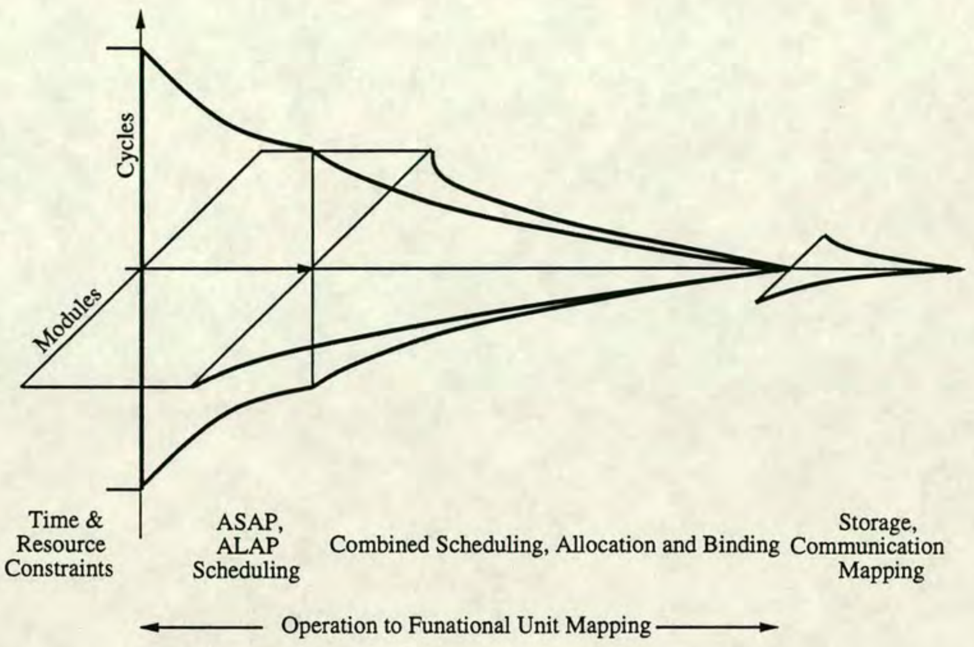

Cycles
Modules
Time & Resource Constraints
ASAP, ALAP Scheduling
Combined Scheduling, Allocation and Binding
Storage, Communication Mapping
Operation to Funational Unit Mapping

Figure 3.4: Combining scheduling, allocation, and binding.

operations to functional unit mapping in one algorithm, as depicted in figure 3.4. They are designed to schedule operations to control steps, allocate the necessary resources, and bind the operations onto the functional units, figure 3.5. During each cycle, all scheduling, allocation, and mapping options are considered. Then, an operation is selected to be scheduled and mapped, and hardware is allocated if needed. Although, this approach is efficient for operation mapping, it takes relatively little consideration of the resultant storage and interconnection efficiency. In SAM, storage mapping is ignored. Communication mapping is considered only through connection compatibility between operations and instances. As depicted in figure 3.4, after operation mapping, there is little flexibility left for storage and communication optimizations. As a result, an optimal operation mapping may require an unacceptable storage and interconnection overhead to be realized.

An alternative method[BALA89] uses an iterative synthesis approach where operation, storage and communication mapping are carried out sequentially in a loop. ASAP, ALAP and double headed scheduling are used. Higher priority is given to the operations closer to the critical path. The criterion is to minimize the cost of

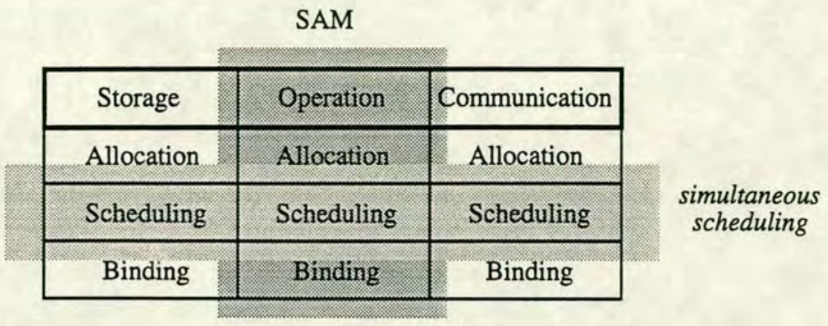


Figure 3.5: Operation, Storage and Communication Mapping.

interconnection. Although it is able to perform the mapping processes iteratively, it still suffers from the drawback we have mentioned before i.e. having sub-optimal results in each step. The iterative process only minimizes the impact of it. Finally, in recent research, [GEBO91] uses simultaneous scheduling to perform all three processes together, figure 3.5. However, it is restricted to DSP style designs with relatively simple functional unit types.

For most of the above approaches, the goal of a task is to explore the design space; then, based on the information obtained, decisions are made. Once the corresponding decisions have been made, the explored design space is cancelled without considering the oncoming tasks. The main problem is that the correctness/efficiency of those decisions will not be known until the synthesis process is approaching the final implementation. Hence, the hidden deficiency: if the decision is found to be unsatisfactory, there will be no easy way out. These decisions not only have to be satisfactory by themselves but they must integrate well with each other. Although cutting down the design space is necessary, the decision which on which it is based is not always sound.

Referring back to the step-by-step approach, the restriction of the design space by ASAP and ALAP scheduling is necessary because scheduling to the prohibited cycles will violate the control or the data dependence. However, it is only sound if the fastest resource type allocation is assumed. Otherwise, it causes *overkill* and no matter how efficient the following tasks are, the access to some of the potential design space is prohibited.

Ideally, we would like to perform the mapping of operations, storage and communications concurrently, as shown in figure 3.5. This formulates a truly global optimization process. However, it would be impossible to cope with the explosion of design space. Some researchers have suggested:

1. partial interactive; leaving the responsibility to the designer,
2. re-running with different performance criteria to affect the decisions,
3. backtracking to points where a poor decision was made,
4. ripping up part of the structure to redesign it [KNAP88],
5. using expert systems to help the decision making process [KOWA85],
6. abstracting and propagating the physical constraints upward to higher abstract level[PANG87A].

Method 1 may be necessary sometimes. For instance, if the target architecture is not fixed at all, it is easier to let the designer specify it than try to find an optimal architecture. Method 2 is a trial-and-error approach which is definitely not a direct way to tackle the problem. Even provided with the understanding of the algorithm, a large number of runs are still needed. Method 3 is promising, but only if you know where a mistake was made. It will require intermediate storage of all the intermediate steps. However, this back-tracking feature is very useful in a semi-interactive system. Method 4 is one of the methods which has proven to be very helpful. By synthesis from partial structure described in [KNAP88], it is possible to repair the design after changes have been made. But again, you need to know where changes should be made and how to start. Method 5 has been implemented in [KOWA85] though a very defined target architecture is needed for it to work efficiently. Method 6 aims to derive high-level constraints from the physical constraints, they are then propagated up the synthesis process to more abstract levels. Thus with these high-level constraints, it enables potential violations to be detected as early as possible.
Summarizing, the problems which we are facing are:

1. The occurrence of overkill. Overkill which happens in early tasks will seriously handicap the efficiency of the later ones.

2. The scale of the problem. Although we would like to consider as much information and perform as many tasks concurrently as possible, it is quite impossible to explore the whole design space at one time. Added to that, the design space is multi-dimensional and discontinuous. An approach which consists of a number of sub-tasks is still essential to explore the design space systematically.

3. User interference. The user should be allowed to interact with the synthesis process. There should be a way to capture his/her degree of confidence and thereby allow slight interference to absolute control.

3.4 Progressive Flexibility Damping

With these concerns in mind, we have developed the approach of *progressive flexibility damping* for high-level synthesis. It allows a large design space to be explored; operation, storage and communication to be synthesized concurrently with even different styles of target architecture. It provides a foundation to support and control the derivation and the influx of low level information. Floorplanning annotations or multiplexor input optimization can be considered progressively in conjunction with high-level constraints. Flexibility damping also allows users to make decisions for controlling the synthesis process with respect to his/her degree of confidence.

Flexibility is the freedom of a not-yet-bound algorithmic level element in the design space. In other words that is the number of instances available to realize it.

$$\begin{aligned} Flexibility(element) &= \{instance\} \\ instance &= resource(sch) \end{aligned}$$

An instance is a well defined continuous number of cycles, *sch*, associated with a resource. As an example, in figure 3.6a,

$$\begin{aligned} Flexibility(oper) &= Flexibility_{FU}(oper) \times Flexibility_{CS}(oper) \\ &= \{Adder1(2,2), ALU1(4,4), ALU2(3,3), ...\} \end{aligned}$$

$$\begin{aligned} Flexibility_{FU}(oper) &= \{Adder1, Adder2, ALU1, ALU2\} \\ Flexibility_{CS}(oper) &= \{(2,2),(3,3),(4,4),(5,5)\} \end{aligned}$$

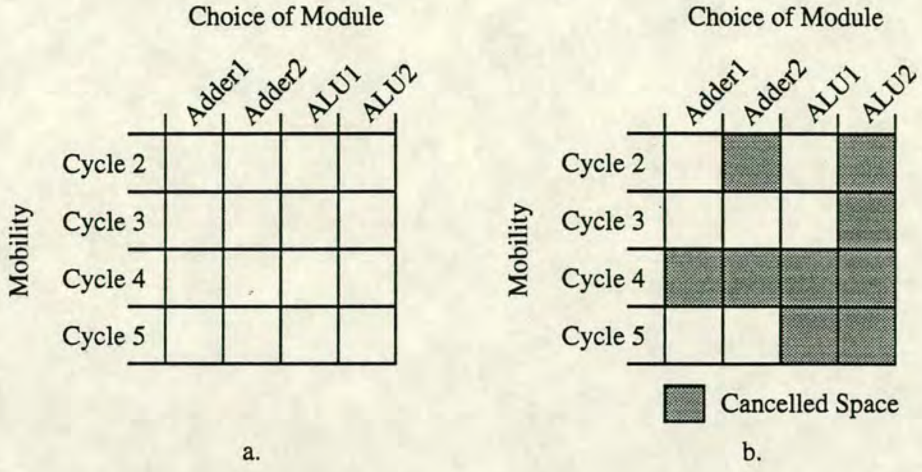


Figure 3.6: Design Space for an Operation, say add(+).

For $Element = \{\, OPER, RMEM, LINK \,\}$ and $Resource = \{\, FU, RF, BUS, ... \,\}$, With operation (*oper*) to functional unit (fu), storage($rmem$) to register file (rf) and interconnection($link$) to buses (bus), we have

$$instance_{oper} = fu(sch) \in \{Flexibility_{CS}(oper) \times Flexibility_{FU}(oper)\}$$
$$instance_{rmem} = rf(sch) \in \{Flexibility_{CS}(rmem) \times Flexibility_{RF}(rmem)\}$$
$$instance_{link} = bus(sch) \in \{Flexibility_{CS}(link) \times Flexibility_{BUS}(link)\}$$

The curves in figure 3.3 and 3.4 represent the availability of instances, hence the flexibility of a construct along the flexibility damping process. In this discontinuous design space, the mobility of an operation does not have much meaning. This is because during the damping process, the freedom is not continuous any more.

On the other hand, damping is the process of constraining the flexibility. It is a gradual process of removing design variations which have proved to be inferior. The philosophy of progressive flexibility damping is to avoid making unnecessary decisions early in the synthesis process. This is to prevent any earlier tasks from commitments which may later prove to be inefficient.

Conceptually, flexibility damping shares some similarities with *delayed binding* [RAJA85], but it has also significant differences. The damping process removes inferior implementation segments which are detected unambiguously to violate some constraints. However, delayed binding defers making decisions until more information is available. As an example in figure 3.6, instead of trying to find the best instance among the 16 choices available, the damping process gradually cancels those which are proven to be unacceptable. In this example, possible reasons may include: mapping to cycle 4 is removed to yield a more balanced schedule; "ALU2" which has delay of two cycles is too slow; "ALU1" which is pipelined cannot be scheduled to the last cycle; "Adder2" has already been committed in cycle 2, etc. As a result, it yields figure 3.6b. It should be noticed that the mobility of the operation is no longer continuous.

With this structure in place for every abstract element, the designer can interfere with the synthesis process by physically removing instances from the feasible design space. If he has absolute confidence in the decision, all but the target of the available instances can be removed.

It is usually necessary to allocate functional unit types to operations before scheduling. This is to provide the necessary delay information on which the scheduling task works. As a result, tremendous pressure is placed on the allocation task to ensure that an optimal set of functional units is chosen. Flexibility damping does not need to know the exact delay of the operations. As demonstrated in figure 3.6, several allocation can be considered at the same time. This avoids the critical decision-making step at the very beginning of the synthesis process, and allows it to be performed in the course of the damping process. The process of doing allocation and scheduling concurrently has been demonstrated in various systems [HEMA90] [CLOU90] [GEBO91].

To enable an efficient iterative cut down of design space, we must establish constraints at higher levels. However, instead of allowing violations to be detected as soon as possible, they are used to limit the number of decisions needing to be made. Each task or process just makes enough decisions to bring the design within the constraints. As a result, it avoids over-ambition and overkill. More flexibility is left for the later processes. The benefit of doing so can be highlighted in the following example.

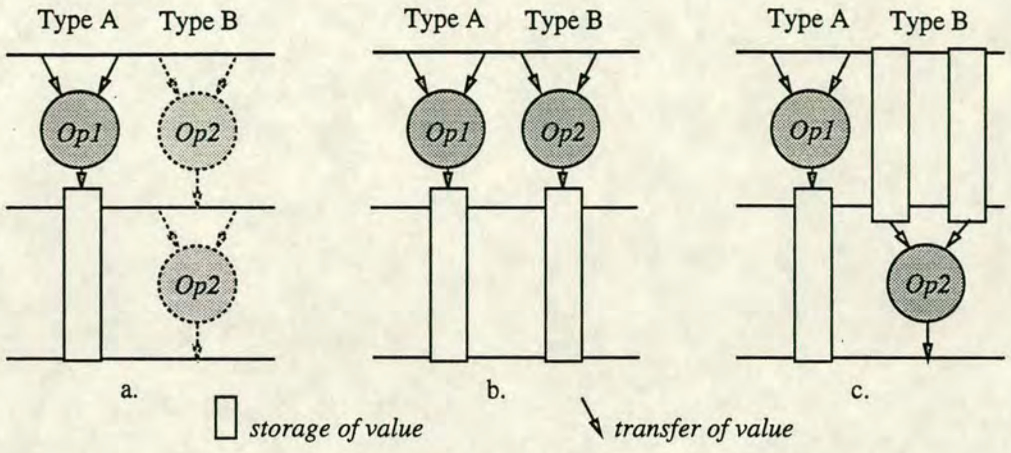


Figure 3.7: One Beneficial Outcome of Flexibility Damping

Traditionally, operation mapping is performed before storage and communication mappings. The reason may be obvious: information for storage and communication mapping is not available until operation mapping is done. Now, let us consider the situation in figure 3.7a. There are two different operations, *Op*1 is Type A and *Op*2 is Type B. Operation *Op*2 can be scheduled either to the first or the second cycles. Since the two operations are of different type, balance scheduling will not have much influence on the outcome. Thus, *Op*2 may be scheduled to the first cycle for various other minor reasons. Only when it comes to the communication mapping phase will it be recognised as an inferior design, figure 3.7b, with 6 buses(4 input and 2 output). It may need to be re-designed. On the other hand, if the scheduling decision is not made, then during the communication mapping process, we have a choice of moving *Op*2 to the second cycle. Figure 3.7c, shows the outcome. Although addition of pre-input storage is required, the number of buses is reduced by half to 3 buses. This is because the buses can be re-used in the second cycle. To yield this result and achieve benefit like this, extra workload coupled with a smart process is essential.

In addition, during the damping process, the user can interfere with the decisions of the system without causing too much disturbance. Operations which are associated with high mobility can be re-assigned to more economic functional units. Due to the nature of the damping process, only the flexibility of the affected operation needs to be updated. A re-run of the scheduling process on the whole graph can be avoided.

3.5 Integrated Concurrent Mapping

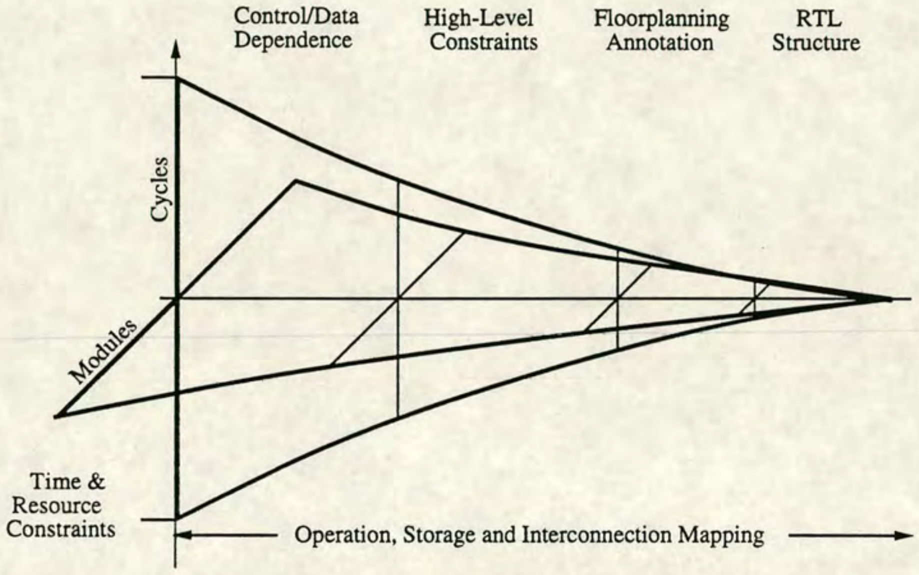


Figure 3.8: Integrated Progressive Flexibility Damping.

Figure 3.8 depicts an ideal flexibility damping process. The design space is cut down smoothly and gradually by waves of procedural design space exploration. The aim of each design exploration is to bring the design within the high-level constraint on each level. It is true that no matter how considerate some decision functions are, there are cases where they are inefficient. The solution is to have constant injection of information to fine tune the accuracy of these decision functions.

1. After the input behavior description has been translated into CDFG, various transformations are performed. At that level, the control/data dependencies are the main concern. The flexibility of each element covers the whole design space.

2. Initially, simple ASAP and ALAP schedulings can be used to define the cycle flexibility, mobility, of the operations. Their types define the module flexibility of resource allocations. Then time constraints, in terms of number of cycles, and area constraints, in terms of number of resource units, can be brought into effect. They are used by resource allocation with resource restricted scheduling. However, instead of selecting the best set of resources, they are used to remove unsuitable resources. Since different resource types will have different execution times, different mobility patterns(as in figure 3.6b), will develop for the flexible design space of an operation.

3. The style of the target architecture is not specified before the synthesis process but is injected as high-level constraints during the damping process. Therefore, various styles can be evaluated before committing to the final architectural style.

4. Floorplans derived from the resource constraints can be annotated to bias the resource binding.

As a whole, the damping process uses a divide-and-conquer approach to handle the complexity of the problem. It allows constraints and optimization concerns to be graded into different levels and to be considered progressively.

In our case, figure 3.9, we have divided the process into several phases. Precautions are taken to make sure that over-ambition or overkill is avoided. Starting from the initial performance and resource constraint, the phases of the flexibility damping process can be divided into:

- High-Level Transformation (HLT),
- Coarse Scheduling and Allocation (CSA),
- Fine Scheduling and Binding (FSB),
- Fine-gain RTL Optimization.

3.5.1 High-Level Transformation

High-level transformation (HLT) is $\overset{\text{used}}{\wedge}$ to modify the control/data flow of the graph so that it can be efficiently implemented in hardware. Common language compiler transformations like constant folding, common subexpression and dead-code elimination, variable renaming etc. can be used to remove redundancy from the description. Hardware oriented optimizations like replacing multiplications with shifts etc. can improve the efficiency of the design.

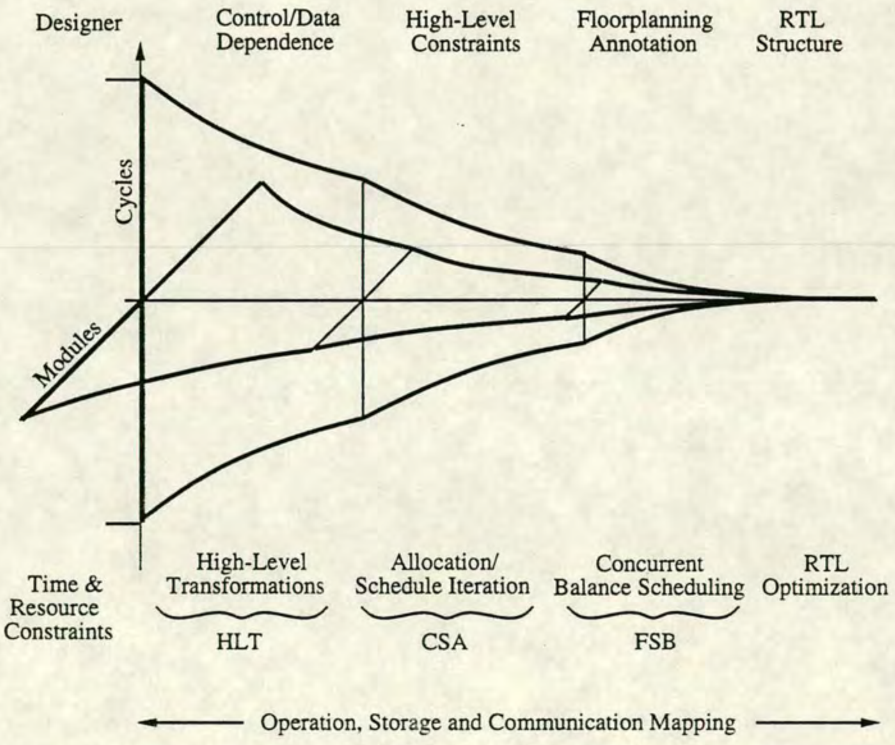

Designer
Control/Data Dependence
High-Level Constraints
Floorplanning Annotation
RTL Structure
Cycles
Modules
Time & Resource Constraints
High-Level Transformations
HLT
Allocation/ Schedule Iteration
CSA
Concurrent Balance Scheduling
FSB
RTL Optimization
Operation, Storage and Communication Mapping

Figure 3.9: Integrated Progressive Flexibility Damping.

Optimizations which simplify the control/data flow graph or tree height reduction such as transformations which re-arrange the graph, will have direct impact on the scheduling process [JAIN89]. They either reduce the amount of computation or increase the amount of parallelism available for scheduling. Control-oriented transformations can improve resource sharing by mutually exclusive conditions and also can explore the advantage of *speculative execution*. Generally speaking, a balanced tree-like graph will require large amounts of resources to realise its full potential, while a serial-like one requires less resources but will have longer delays.

In Chapter 4, we look at two compiler optimization techniques

1. Variable Renaming, and
2. Minimum Execution Time Tree Generation

In Chapter 5, the *Control-Data Flow Graph* (CDFG) is introducted. This is followed by graph transformations, in the order of transformation aggressiveness:

1. Simplification,
2. Local-Data Transformation, and
3. Global-Data Transformation

3.5.2 Coarse Scheduling and Allocation

Coarse scheduling and allocation (CSA) is the coarse design phase where various design styles and strategies are evaluated quickly. The main process is dominated by the allocation-scheduling loop. Various types of functional units from a chosen library are assigned to the operations. Critical parts of the design can be identified and an application specific functional unit can be constructed. The expert knowledge of the designer will be put to good use here. CSA performs:

- functional unit type allocation,
- resource restricted ASAP and ALAP scheduling,

The main objective is to define the design space for the later phases of the damping process. From the timing constraints, possible choices of modules are established. Scheduling is performed with the delay information from the fastest type of functional unit. The fastest ones are chosen in order to maximize the mobility of the operations. If mobility and choice of modules allow, fast functional units can be replaced with more economic ones later in the damping process. As a result, operations of the same type may end up being implemented by different types of functional units depending on their position in the flow graph.

In Chapter 6, we will detail the resource restricted scheduling and other methodologies used in CSA.

3.5.3 Fine Scheduling and Binding

Based on the design space established by CSA, *Fine Scheduling and Binding* (FSB) makes full use of the timing allowance to minimize the cost of the design. FSB schedules and binds

- operations to functional modules
- storage elements to global registers or local buffers
- communications to buses

To facilitate the process, *Distribution Graph* (DG)s of operations, storage and communication are built. The damping process is applied to them simultaneously. This ensures that equal attention is placed on them. Their distributions are balanced accordingly to minimize the total cost of the design.

At the same time, architectural constraints derived from background memory management, functional module configurations, input/output buffers, floorplanning annotations, etc. are introduced to target the design. Various storage and communication schemes are also explored to minimize the cost of global interconnections.

In chapter 7, we will discuss the construction of the distribution graphs, the damping policy, the architectural constraints and how they are used in FSB.

Chapter 4

Compiling

Generally speaking, a compiler takes a source program as input; in the case of programming language compilers, they produce a sequence of machine instructions for the target machine; for silicon compilers however, they produce a netlist describing the hardware architecture and the micro-sequences which execute on it. The term "*Silicon Compiler*" has broad meanings but here we use it to refer to any high-level synthesis system which is capable of generating a hardware netlist from a programming-language-like description.

Extensive research has been done on the compilation process [AHO86]. The process is so complex that it is usually partitioned into a series of subprocesses called phases:

1. Lexical Analysis and Parsing
2. Control and Data Flow Analysis
3. Optimization
4. Resource Allocation
5. Code Generation

Many of these techniques commonly used in software compilers will find immediate application in silicon compilers. For example, the lexical analysis and parsing phase are moreorless identical. Also included are many optimization techniques such as:

– constant folding,
– common sub-expression elimination,
– dead-code removal,

- copy propagation,
- strength reduction,
- loop-invariant motion, etc

However, it is important to understand the differences between software and silicon compilation. In the following sections, we will consider these differences in detail.

4.1 Software Compiler vs Silicon Compiler

4.1.1 Compilation Target

For software compilers, the target machine which will execute the program is well defined. No matter whether it is a general purpose computer or a Very Large Instruction Word (VLIW) machine, different compilers are made available for different machines. The common goal of this compiler is to translate the high-level program description into machine instructions so that the program can be executed efficiently on the machine for which the compiler is designed.

However, for a silicon compiler, the input description says nothing about the piece of silicon or the chip on which the description is to be executed. The job of the compiler is to construct the hardware structure as well as the micro-sequences together to realize the description. The hardware structure usually consists of one or more data-path units. and the micro-sequences are usually implemented by a control unit. Since the design solution space is very large, the acceptance of the solution is usually governed by some pre-defined constraints like speed and area. It is extremely unlikely that the first pass will meet these constraints. A number of design iterations are needed.

4.1.2 Parallelism

Since the target architecture is flexible, in contrast to software compilers, silicon compilers must explore the fine-grain parallelism embedded in the description. If the design goal can be met with any sequential hardware, there would be little point in pursuing a custom design. An off-the-shelf microprocessor would be good enough. This low-level parallelism of the application specific design can usually deliver a performance comparable to any super-computer at a much lower cost and size. Therefore, optimization techniques, such as

loop quantization, loop unrolling, loop partitioning
loop interchanging, loop jamming, and loop splitting

available in parallel software compilers can easily find their way into the front-end of most silicon compilers. The flexibility offered by customised hardware enables most of these algorithms to be implemented with high efficiency. The main reason is that the computational overhead introduced by most of these optimisation algorithms can now be taken up easily by simple hardware elements introduced by the silicon compiler. For instance, consider the following loop unrolling example.

Listing 4.1

```
for i := x to y do                    for i := x to y step 3 do
    A[i] := expr(i);                      A[i] := expr(i);
end;                                      if (i > y) then exit;
                                          A[i+1] := expr(i+1);
                                          if (i+1 > y) then exit;
                                          A[i+2] := expr(i+2);
                                      end;
```

The loop is unrolled three times. The *test-exit* operations may be an expensive overhead in usual sequential platforms; but in silicon they can be implemented by comparison element(s) working in parallel with the array operation. However, before putting in extra hardware, the compiler must make sure that the increase in speed justifies the expansion in area.

4.1.3 Common subexpressions

Generally, common subexpressions can be handled in two ways. Compute the result every time whenever and wherever it is needed or do the computation once and store the result for future usage. Software compilers will favour the second choice. This is because most computation results are stored in memory/cache and access to memory/cache for operands before execution is a common activity in most sequential machines. A similar arrangement in silicon implementation will mean that the result stored has to be distributed to wherever it is used. Besides the memory which is required to store the value, the transfer will imply significant steering logic overhead. This may turn out to be more expensive than re-doing the computation on the spot. This is particularly true when the operation requires simple hardware only. Therefore, trade-offs need to be made.

4.1.4 Procedures

In both cases, the desire for procedures is to create a hierarchy of abstractions so that both design and compilation processes can be more manageable. For software, the cost of having procedures is paid at run-time. That is the stack operations and context switching which involves saving and restoring registers before and after procedure calls. However, in a silicon compiler, there are three kinds of procedure definition. Each has different implications.

1. The first kind are procedures which define library functional units or chips whose implementation is not available to the user. This is similar to the *information hiding* mechanism provided by external modules in most software environments. Only details about how calls can be set up, the interface and the performance are provided. As usual, it can be shared if the operations are mutually exclusive but unlike its software counterpart, the number of units can be increased to improve the performance of the design. As a library unit, it can be designed separately as a unique unit on silicon.

   Listing 4.2

   ```
   component decoder_74xxx(b1, b2 : in bit,
                           d1, d2, d3, d4 : out bit)
   ```

   For instance, listing 4.2, is clearly addressing an external component which can be used as a functional unit in the description.

2. For the second kind, the procedure call represents a *software partition* of the description. This kind of partition shares the same characteristics with its software counterpart and carries no hardware sense. The usual method is to unfold the procedure call. However, if the procedure is used frequently, it could be implemented independently and treated as a functional unit.

   Listing 4.3

   ```
   procedure mean_of_four(d1, d2, d3, d4 : in float;
                          mean : out float)
   begin
     mean := (d1 + d2 + d3 + d4) / 4;
   end;
   ```

For instance, listing 4.3, is a purely software procedure used mainly to improve the modularity of the description. It can either be constructed as a functional unit or can be unfolded to merge with other computations.

3. The third kind are real *hardware partitions* defined by the user. Very often, the procedure is made up of smaller hardware components. If it is separated from the rest of the design and constructed individually, it will behave just like the first kind of procedure definition — a component part. On the other hand, since the structure is not hidden, the composite hardware units can be made visible and shared by other procedures. This resource sharing feature has similar advantages as in-line expansion of procedure calls.

Listing 4.4

```
procedure control_unit(reset : in bit;
                       f1, f2 : in bit;
                       c1, c2 : out bit;
                       d1, d2 : out bit);
        var stage : integer;
begin
  while (reset) do stage := 0; end;
  while (true ) do
      case stage, f1, f2 of
         1, 0, 1 : c1 := 1; c2 := 0; d1 := 1; d2:= 0; stage := 2;
         2, 1, 0 : c1 := X;c2 := 0; d1 := 0; d2:= 1; stage := 3;
         3, 0, X: c1 := X;c2 := X;d1 := 0; d2:= 1; stage := 4;
         4, 1, X: c1 := 1; c2 := 1; d1 := 1; d2:= 0; stage := 1;
      end;
  end;
end;
```

For instance, listing 4.4, is a hardware partition which reflects the architectural structure of the hardware. Since it is describing a piece of independent hardware, it is unwise to unfold it into the main process.

We have to understand that silicon compilers do not have the intelligence to distinguish between these three different kinds of procedure definitions. The programmer has to indicate them explicitly to the compiler. To do so, identifiers, such as *component*, *block*, *procedure* etc, available in most hardware description languages, can be used.

4.1.5 Arrays and Memory Addressing

Traditional computing machines have a main memory block in which all data and instructions are stored. To perform an operation, they read data from the memory, do some calculations and write the result back into the memory. Hence, the reading and writing operations, the memory bandwidth of the machine, play a significant role in the total performance.

In addition, in software, there are different kinds of addressing mode to meet the need of the programming languages. The target architecture of a silicon compiler does not share the same characteristics. Since the number of variables and the computational sequence are known, better arrangement of memory can be made to avoid memory bottlenecks. Variables which are frequently referenced can be stored in registers, individual arrays may have their own register files and variables which are not used at the same time can be arranged into a single register file. For example, for the following expression.

$$t1 = 2 * A[i];$$

The variable t1 will be in a register, the constant 2, will be in ROM and the array element A[i], will be in memory. Moreover, if a register file is assigned to array A, an address calculation like:

$$A[i] = MEM[addr(A) + i - 1]$$

will no longer be needed. Nearly all the memory access will be direct addressing. There will not be any memory bottleneck, the memory bandwidth is tailor-made to meet the computational intensity.

4.2 Parsing

In our system, the input description (INDP) is scanned and parsed in a recursive descent fashion. The compiler translates the source text of the input into an intermediate format (INFT). This intermediate form records the tree structure of the description. Each statement comprises four major fields:

- TYPE indicates the type of the statement,
- INPUT is the set of input variables,

- OUTPUT is the set of output variables,
- BODY is the syntax tree.

The TYPE is one of the following: {Assignment, Procedure Call, *if*, *case*, *while*, *for*}. The statements are created during the initial parsing phase and are threaded to represent the ordering of the statements as they are encountered in the sequential scan of the INDP.

The calculation of the input and the output set of a statement is by *dataflow analysis*. For example listing 4.5:

Listing 4.5

```
BZ: z := 0;
    while (x > 0) loop
       z := 1 + h + x * z;
       x := x - 1;
    end loop;
```

The input and output set of BZ is

$$\begin{aligned} \mathrm{IN}(BZ) &= \{0, 1, \mathrm{x}, \mathrm{h}\} \\ \mathrm{OUT}(BZ) &= \{\mathrm{x}, \mathrm{z}\}. \end{aligned}$$

For the different kinds of statements, the INPUT and OUTPUT sets are as follows:

S1: x := expression;

$$\begin{aligned} \mathrm{IN(S)} &= \{a \text{ in expression referenced on the R.H.S. of the statement}\} \\ \mathrm{OUT(S)} &= \{x \text{ defined on the L.H.S.; usually only one}\} \end{aligned}$$

S2: if C then B end if;

$$\begin{aligned} \mathrm{IN(S)} &= \mathrm{IN}(C) \bigcup \mathrm{IN}(B) \bigcup \mathrm{OUT}(B) \\ \mathrm{OUT(S)} &= \mathrm{OUT}(B) \end{aligned}$$

S3: if C_1 then B_1 elseif C_2 then B_2 ...... else B_n end if;

$$\begin{aligned} \mathrm{IN(S)} &= \bigcup \mathrm{IN}(C_i) \bigcup_{\forall i} \mathrm{IN}(B_i) \bigcup_{\forall i} \mathrm{OUT}(B_i) - \bigcap_{\forall i} \mathrm{OUT}(B_i) \\ \mathrm{OUT(S)} &= \bigcup_{\forall i} \mathrm{OUT}(B_i) \end{aligned}$$

S4: case E_1 is when $E_2 \Rightarrow B_1$ .....when others $\Rightarrow B_n$ end case;

$$\begin{aligned} \mathrm{IN(S)} &= \bigcup \mathrm{IN}(E_i) \bigcup_{\forall i} \mathrm{IN}(B_i) \bigcup_{\forall i} \mathrm{OUT}(B_i) - \bigcap_{\forall i} \mathrm{OUT}(B_i) \\ \mathrm{OUT(S)} &= \bigcup_{\forall i} \mathrm{OUT}(B_i) \end{aligned}$$

S5: while C loop B end loop;

$$\begin{aligned} \text{IN(S)} &= \text{IN}(C) \bigcup \text{IN}(B) \\ \text{OUT(S)} &= \text{OUT}(B) \end{aligned}$$

S6: for $i := E_1$ to E_2 loop B end loop;

$$\begin{aligned} \text{IN(S)} &= i \bigcup \{x \mid x \in E_1\} \bigcup \{x \mid x \in E_2\} \bigcup \text{IN}(B) \\ \text{OUT(S)} &= \text{OUT}(B) \end{aligned}$$

For a conditional statement, which does not cover all the possible branches, special attention is needed. For a loop statement, the intersection of the input and the output set will give the inductive variables. Note that for the last entity, the inductive variable i is not included in the output set of the *for* loop. For IN(BZ) and OUT(BZ), these are constructed by the collection of simple and complex statements inside the body(BZ) of the entity. Let the statements that are in a block(BZ) be S_1, S_2 .... S_n. IN(BZ) and OUT(BZ) can be defined as:

$$\begin{aligned} \text{IN}(BZ) &= \textstyle\bigcup_{\forall i=1}^{n} \left(\text{IN}(S_i) - \bigcup_{\forall j=1}^{i} \text{OUT}(S_j) \right) \\ \text{OUT}(BZ) &= \textstyle\bigcup_{\forall i=1}^{n} \text{OUT}(S_i). \end{aligned}$$

However, as IN(BZ) and OUT(BZ) are usually computed recursively, we can simplify the above equations into:

$$\begin{aligned} \text{IN}(BZ) &= \bigcup \text{OUT}(P) \\ \text{OUT}(BZ) &= \text{GEN}(S) \bigcup (\text{IN}(S) - \text{KILL}(S)) \\ \text{GEN}(S) &= \{x \mid x \text{ defined in } S\} \\ \text{KILL}(S) &= \{a \mid a \text{ killed in } S\} \\ \text{P} &= \text{predecessor of B} \end{aligned}$$

4.3 Variable Renaming

Before doing any variable renaming, we must understand the kinds of *data dependency* which can be present. In total, there are three kinds of dependency:

1. Input dependency

   The result of one operation is an operand of a later operation.

   x := a + b;
   y := x + 1;

2. Output dependency

   Two operations write to the same variable, memory location or member of an array.

   ```
   m := a + b;
   m := 2;
   ```

3. Anti-dependency

   One operation uses a value which will be overwritten by a later operation.

   ```
   k := x + 1;
   x := y + z;
   ```

Potential parallelism in a code segment will usually be improved if renaming is used to eliminate multiple definitions of a variable. For instance, consider the following lines of code:

Listing 4.6

```
s1:  x := a * b;
s2:  y := x * c;
s3:  x := c + d;
s4:  k := x + 1;
```

If *x* in the statement s3 is renamed to be another distinct temporary variable say *t*, then the output dependency, between s1 and s3, and the anti-dependency, between s2 and s3, on *x* can be removed. The assignment to *t*, s3a in listing 4.7, can now be moved forward to execute concurrently with s1. Semantic correctness must be maintained by replacing the renamed variable, i.e. by renaming *x* in s4 to *t*.

Listing 4.7

```
s1:  x := a * b;   s3a:  t := c + d;
s2:  y := x * c;   s4a:  k := t + 1;
```

This technique is widely used in vector compilers to eliminate spurious dependencies and thus increase the potential for parallelism exploitation. Basically, we will rename the variables when they are assigned. So for the above listing 4.6, the result will be:

Listing 4.8

```
s1:   x1 := a * b;
s2:   y1 := x1 * c;
s3a:  x2 := c + d;
s4a:  k1 := x2 + 1;
```

Although renaming is useful to highlight and extract the exact data dependency, precautions must be taken when mixing it with control statements. We will look at this in the following sub-sections.

4.3.1 Iteration statement

Compilers rename multiple definitions of a variable within iterations to maximize the potential parallelism. If these variables are used in later iterations, the variables will be re-assigned at the end of the loop to get ready for the next iteration. For example, consider the following input description:

Listing 4.9

```
while (x > 0) loop
    y := y + x;
    z := z * x;
    x := x - 1;
end loop;
```

In order to maintain the value of *x*, the decrement of *x* cannot be performed until the computation for *y* and *z* have been completed. Renaming will re-arrange it as in listing 4.10

Listing 4.10

```
loop(x > 0)
    y1 := y + x;    z1 := z * x;    x1 := x - 1;
    y := y1;        z := z1;        x := x1;
end loop;
```

where all the computations can be done concurrently. However, as the variables (*x, y, z*) need to be used in the next iteration, they will be re-assigned the new value at the end of the loop. Here, variables *x1, y1, z1* are temporary variables while variables *x, y, z* are value containers.

4.3.2 Conditional statement

Firstly, let us understand what the problem is. For instance,

Listing 4.11

```
if (a = y) then
    a := a + 1;
end if;
x := a + y;
```

Listing 4.12

```
if (a = y) then
    a1 := a + 1;
end if;
x := a(a1) + y;
```

It is clear that if renaming is performed without considering the conditional statement, there will be confusion when the renamed variable is used after the conditional statement, listing 4.12. In this example, the value of the last statement cannot be determined because it does not know which value of *a* to refer to.

The main objective of renaming conditional statements is to equalize all the output variables in different branches; so that variables used later can refer to a unique name. This situation can roughly be divided into four categories

- Unbalanced conditional statement; *if* statement without the *else* body
- Unbalanced re-assignment in the branches
- Different variables in different branches
- Temporary variables which die within the branch

For example, consider the following example,

Listing 4.13

```
if (r = s) then                 if (r = s) then
   t := s + r;                     t1 := s + r;
   a := t + s;                     a1 := t1 + s;
   a := a + r;                     a2 := a1 + r;
   b := t + r;        ⟶            b1 := t1 + r;
else                            else
   a := r + 1;                     a2 := r + 1;
                                   b1 := b;
end if;                         end if;
c := a + b                      c1 := a2 + b1;
```

variable *t* is a temporary variable which dies within one of the branches. Therefore, it does not need to be made visible outside the conditional statement. In the *then* body the variable *a* is assigned twice, but only once in the opposite branch. So, instead of renaming the variable *a* in the *else* body by *a1*, it is renamed to the most up-to-date name *a2* to achieve a unique output. The *dummy assignment*, b1 := b, is also introduced to balance the assignments in the branches. Finally, the conditional statement will have a unique set of output variables.

4.3.3 Procedure call

To support modularity, procedure calls are an essential feature. A procedure takes arguments and produces outputs. Hence, renaming has to be performed on the arguments as well. As there are three kinds of parameter, renaming can be divided into three categories.

1. Input only parameters
   Just like the variables on the R.H.S. of a statement, they will be updated to reflect the data dependency.

2. Output only parameters
   Since the output variables have been given a definition, they will be renamed to reflect this. Although the output variable may be assigned a number of times in a procedure, this does not matter as long as we know that it has been altered. Suppose we have a procedure definition:

   procedure cal(a:in integer; b:in integer; c:out integer)

   renaming the following piece of code will mean renaming *z* as if it was assigned.

   Listing 4.14

   ```
   x := p + q;             x1 := p + q;
   cal(x, y, z);   ⟶      cal(x1, y, z1);
   c := z + 1;             c1 := z1 + 1;
   ```

3. Input and Output parameters
   This is complicated by the fact that the value of the input variable is changed. This situation is similar to self-assignment, for example:

   ```
   x := x + k;   ⟶   x1 := x + k;
   ```

   Simple renaming does not work because it will fail to represent either the input dependency or the output dependency. Hence, we introduce a new identifier to exhibit the change in value. For instance, a call to the following self-increment procedure will be renamed as:

   ```
         procedure increment(x : inout integer)
   increment(a);   ⟶   increment(a \ a1);
   ```

4.3.4 Array References

Array references are quite difficult to handle. The main reason is that two successive array references may or may not aim at the same element of an array. If the array is indexed by variables, it is quite impossible to find out the data dependency at compile time. For the case of a simple variable, renaming a to *a1* will mean two different variables. If the same is applied to arrays, it will mean a copy of all the elements in the old array except the one which has been newly updated. Therefore, in order to simplify matters, we choose *NOT* to rename any array.

4.4 Expression Regrouping

The main goal is to recognize parallel processable tasks in arithmetic or logic expressions and to regroup them for minimum execution time. For example, consider the following statement:

$$X := A + B + C + D + E;$$

An optimal compiler will usually re-arrange it to minimize the number of levels. The additions in the above example will be regrouped as

$$X := ((A + B) + ((C + D) + E)));$$

so that adjacent additions can be evaluated in parallel. The benefit is that the delay is decreased from 4 to 3 adder-delays with 2 adders.

4.4.1 Minimum Execution Time Tree Generation

More generally if the statement consists of different kinds of operations associated with different delays, then the difference in delays should be taken into account. Thus, it becomes important to minimize the total execution time instead of the number of levels.

Consider Figure 4.1, the tree (b), with a maximum level of 4 yields a faster execution time than the tree (a) with 3 levels. This is achieved by arranging expensive operations, such as multiplication, to execute concurrently with several less expensive operations. The resulting delay is calculated by accumulating all the delays on the path of calculation. The algorithm for expression regrouping can be summarized as follows:

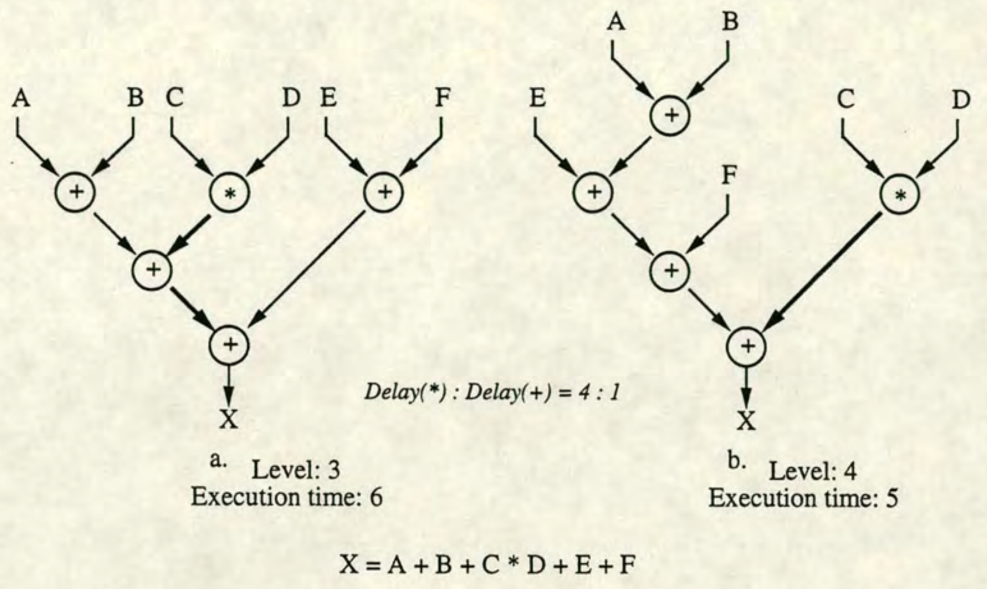


Figure 4.1: Expression level vs Execution time

Step 1:	assign weight to memory references like a[i]; assign weight to inputs(input delay); and get the weight of defined operands.
Step 2:	Sort the operands in increasing order of weight.
Step 3:	Combine the first two least weighted operands and calculate the new weight for the expression using the following: Weight of new expression = MAX(w1, w2) + weight(operation) w1 = weight of the first expression. w2 = weight of the second expression.
Step 4:	Remove the first two operands and put the expressions back with increasing order of weight.
Step 5:	If more than one expression left, go back to Step 3. Otherwise finished with the balanced expression on top.

Figure 4.2 demonstrates the algorithm. It is not essential to know the exact delay of the functional units in order to achieve an accurate minimum execution time tree. However, a rough ratio of the execution time among the operations is sufficient. It does not make much difference whether the delay ratio of an addition to a multiplication is (1 : 2) or (7 : 16).

x := a + b ∗ c + d - e + in

	Weight of Expression	:	Expression
⟹	0	:	a
	0	:	d
	0	:	-e
	2	:	in
	4	:	(b ∗ c)
⟹	0	:	-e
	1	:	(a + d)
	2	:	in
	4	:	(b ∗ c)
⟹	2	:	in
	2	:	((a + d) - e)
	4	:	(b ∗ c)
⟹	3	:	(in + ((a + d) - e))
	4	:	(b ∗ c)
⟹	5	:	((in + ((a + d) - e)) + (b ∗ c))

x<5>:= ((in + ((a + d) - e)) + (b ∗ c))

	the delay time of *reading input* is 2 unit
assuming:	the delay time of ∗ is 4 units
	the delay time of + is 1 unit

Figure 4.2: A run of expression regrouping

4.5 Expression Flattening

Complex expressions are very often used when writing programs or descriptions. The job of expression flattening is to transform a long expression into the following simplified forms:

1. <operand><operator><operand>
2. <operator><operand>
3. <operand>

Temporary variables are introduced to hold the intermediate values. This is done after the expression is regrouped and these simplified expressions are called *isotopic expressions*. The decomposition will proceed by ascending the tree with the smallest sub-tree being flattened first. For instance, the previous balanced expression will be flattened as:

Listing 4.15

```
x := ((in + ((a + d) - e)) + (b * c))

t1 := a + d
t2 := b * c
t3 := t1 - e
t4 := t2 + in
t5 := t4 + t2
```

Besides flattening the R.H.S. of a statement, for an array operation, the indexing expression will be flattened as well. Array referencing will be transformed into a much simpler from:

```
Array Read : <variable>:= array[<variable>];
Array Write : array[<variable>] := <variable>;
```

For instance,

Listing 4.16

```
a[a[x]+y+1] := y + 4

t1 := a[x];
t2 := y + 1;
t3 := t1 + t2;
t4 := y + 4;
a[t3] := t4;
```

For array elements involved in procedure calls, a combination of renaming and flattening will be used:

Listing 4.17

```
increment(a[x]);

t1 := a[x];
increment(t1 \ t2);
a[x] := t2;
```

Chapter 5

Design Representation

After parsing and high-level transformations, the input description is compiled into the internal representation, the *Control-Data Flow Graph*($CDFG$). We assume that variable renaming and other essential transformations have been carried out. Variables are assigned only once or assigned under mutually exclusive conditions. These are well defined single assignments. For data dependence, output and anti-dependence have been eliminated so we only need to consider input dependence. This will be referred to simply as data dependence. In this chapter, we will discuss the $CDFG$ on which various transformations will be carried out.

In the following sections, we shall unfold this internal representation bit-by-bit A small example, $so\Pi o\eta$, will be used throughout this chapter to illustrate the internal representation. The description is shown in figure 5.1 with its flowchart representation.

Before we go on to the internal representation, there are some definitions we would like to clarify.

- Basic Block — is analogous to its definition in software compilation. In figure 5.1, { 1 2 3 4} forms a basic block and { 5 6 7 }, { 8 9 }, { 10 11 }, etc are other blocks.
- Control Block — this is formed by a control statement. Statement 7 and the statements inside it, { 8 9 10 11 }, form a control block.

```
1       x := Xi;
2       y := Yi;
3       a := y > 0;
4       case a is
         when "1" =>
5          p := y + x;
6          b := x > y;
7          case b is
            when "1" =>
8             m := y + p;
9             n := x / 2;
            when "0" =>
10            m := x + p;
11            n := y * 2;
           end case;
         when "0" =>
12         m := x + y;
13         n := 0;
        end case;
14      z := m + n;
15      Zout := z;
```

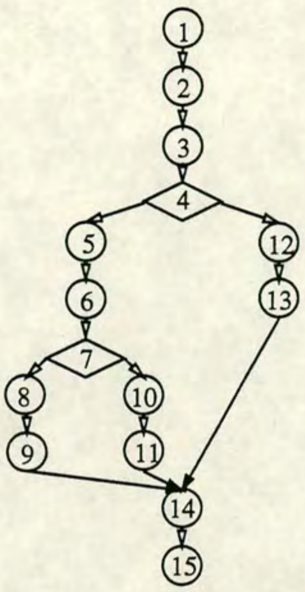


Figure 5.1: Description and the Simple FlowChart Representation of $so\Pi o\eta$.

- Control Level — this is the depth of nesting of the control block within which a statement lies. All nodes { 8 9 10 11 } have a control level of 2.

5.1 Control-Data Flow Graph

The *Control-Data Flow Graph* (*CDFG*) represents the control guards and the operations performed on data flowing from the inputs to the outputs. Operations and control constructs from the compiled description are represented as nodes. The edges describe the dependencies between them. The control-data flow graph, is defined as:

Definition 5.1 *CDFG* is a directed flow graph (⟨NODE⟩, ⟨EDGE⟩); where

⟨NODE⟩ is a set of nodes

= { ⟨N:*fork*⟩ ⋃ ⟨N:*loop*⟩ ⋃ ⟨N:*exit*⟩ ⋃ ⟨N:*call*⟩ ⋃ ⟨N:*stat*⟩ },

where⟨N:*fork*⟩ : set of branch nodes;

⟨N:*loop*⟩ : set of loop boundary nodes;

⟨N:*exit*⟩ : set of loop exit nodes;

⟨N:*call*⟩ : set of procedure call nodes;

$\langle$N:*stat*$\rangle$: set of operation nodes

$\langle$EDGE$\rangle$ is a set of edges

$= \{ \langle \text{E:}\mathit{data} \rangle \bigcup \langle \text{E:}\mathit{ctrl} \rangle \bigcup \langle \text{E:}\mathit{back} \rangle \bigcup \langle \text{E:}\mathit{time} \rangle \}$,

where $\langle$E:*data*$\rangle$: set of data flow edges;

$\langle$E:*ctrl*$\rangle$: set of control flow edges;

$\langle$E:*back*$\rangle$: set of feedback data flow edge of loops;

$\langle$E:*time*$\rangle$: set of timing constraints defined by user

The $\langle$N:*fork*$\rangle$ set represents the *if-then* or *case-when* statements in the input description. The $\langle$N:*loop*$\rangle$ set represents the loop statements, like *loop*, *while* and *for* loops. Since exit statements are usually associated with a condition, it is necessary to separate the $\langle$N:*exit*$\rangle$ nodes from $\langle$N:*loop*$\rangle$ and treat them as a distinct set of nodes. The $\langle$N:*call*$\rangle$ nodes represent calls to internal or external subroutines. The $\langle$N:*stat*$\rangle$ nodes represent various operations.

Definition 5.2 Nodes $\in \langle \text{N:}\mathit{fork} \rangle \bigcup \langle \text{N:}\mathit{loop} \rangle \bigcup \langle \text{N:}\mathit{exit} \rangle$ are called **control** nodes; while nodes $\in \langle \text{N:}\mathit{call} \rangle \bigcup \langle \text{N:}\mathit{stat} \rangle$ are called **operation** nodes; with

$$\begin{aligned} \langle \text{N:}\mathit{ctrl} \rangle &\stackrel{def}{=} \{ \langle \text{N:}\mathit{fork} \rangle \bigcup \langle \text{N:}\mathit{loop} \rangle \bigcup \langle \text{N:}\mathit{exit} \rangle \}, \text{ and} \\ \langle \text{N:}\mathit{oper} \rangle &\stackrel{def}{=} \{ \langle \text{N:}\mathit{call} \rangle \bigcup \langle \text{N:}\mathit{stat} \rangle \} \end{aligned}$$

Definition 5.3 $\langle$N:*stat*$\rangle$ is defined as the union:

$\{ \langle \text{N:}\mathit{arith} \rangle \bigcup \langle \text{N:}\mathit{logic} \rangle \bigcup \langle \text{N:}\mathit{trans} \rangle \bigcup \langle \text{N:}\mathit{port} \rangle \bigcup \langle \text{N:}\mathit{array} \rangle \}$

where $\langle$N:*arith*$\rangle$: set of nodes performing arithmetic operation;

$\langle$N:*logic*$\rangle$: set of nodes performing logic operation;

$\langle$N:*trans*$\rangle$: set of nodes performing simple assignment;

$\langle$N:*port*$\rangle$: set of nodes performing input/output operation;

$\langle$N:*array*$\rangle$: set of nodes performing array read/write operation

Arithmetic and logic operations, such as

p := y + x;   n := a and b;   etc

are the most common nodes and always dominate the graph. Their operations require allocation of functional units. Thus, their quantity will reflect the complexity of the problem. $\langle$N:*trans*$\rangle$ is a set of simple direct assignments, like

n := 0;   p := x;   etc.

Generally, they do not consume any functional units. However, depending on the result of the scheduling, they can be local signals or intermediate values. In the latter case, they will require storage and transmission and hence they occupy resources like registers and buses. ⟨N:*port*⟩ nodes represent the abstract interface with the outside environment. They can have no solid meaning, or like other operations, they can be associated with resources. In that case, input/output ports will be needed but the requirement can be controlled and minimized. Some applications may involve a lot of array operations which will unavoidably require a lot of storage access. An unconstrained implementation could result in very inefficient memory structures. Hence, array read/write operations are introduced and by associating them with their hardware counterparts, the access pattern can be restricted. This also allows memory management to be performed before any scheduling. The internal representation of $so\Pi o\eta$ is shown in figure 5.2.

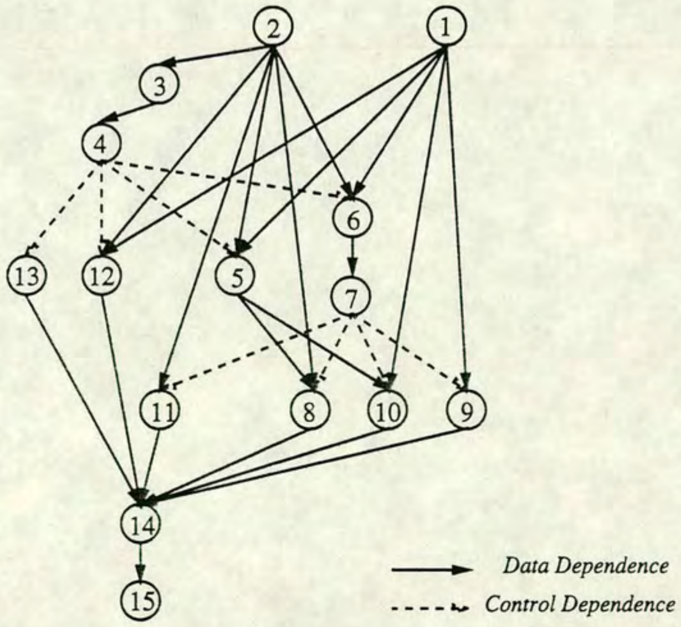


Figure 5.2: The Internal Representation of $so\Pi o\eta$

$\langle$N:*ctrl*$\rangle$	:	{ 4 7 }
$\supseteq$ $\langle$N:*fork*$\rangle$	:	{ 4 7 }
$\langle$N:*oper*$\rangle$	:	{ 1 2 3 5 6 8 9 10 11 12 13 14 15 }
$\supseteq$ $\langle$N:*stat*$\rangle$	:	{ 1 2 3 5 6 8 9 10 11 12 13 14 15 }
$\supseteq$ $\langle$N:*arith*$\rangle$	:	{ 3 5 6 8 9 10 11 12 14 }
$\supseteq$ $\langle$N:*trans*$\rangle$	:	{ 13 }
$\supseteq$ $\langle$N:*port*$\rangle$	:	{ 1 2 15 }

The edges in the $CDFG$ are used to guarantee a correct execution sequence of the operations.

Definition 5.4 An **edge** is a vector, (*bnode* $\rightarrow$ *enode*), where *bnode* $\in$ $\langle$NODE$\rangle$ is the beginning node and *enode* $\in$ $\langle$NODE$\rangle$ is the end(target) node.

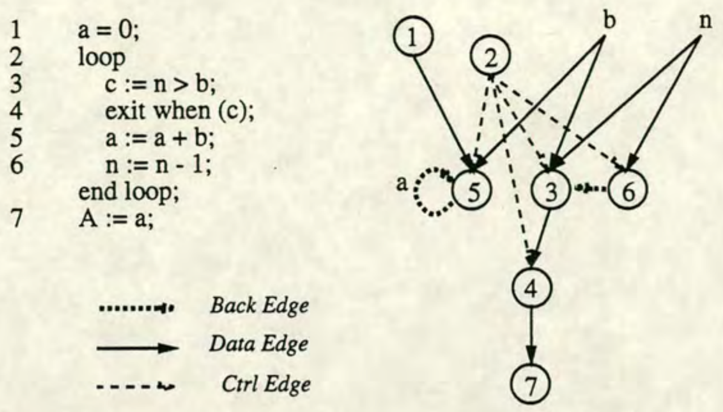
The definitions of various edges described in definition 5.1 are:

assuming *edge* $\bigwedge$ { begin(*edge*) $\in$ Begin_Set $\wedge$ target(*edge*) $\in$ Target_Set }

edge set	*edge*		begin(*edge*)	target(*edge*)
$\langle$E:*data*$\rangle$	*Data_edge*	:	Var_Def	Var_Use
$\langle$E:*ctrl*$\rangle$	*Ctrl_edge*	:	$\langle$N:*ctrl*$\rangle$	$\langle$NODE$\rangle$
$\langle$E:*back*$\rangle$	*Back_edge*	:	Indv_Def	Indv_Use
$\langle$E:*time*$\rangle$	*Time_edge*	:	Time_Beg	Time_End

where	Var_Def	–	{ n $\in$ $\langle$N:*oper*$\rangle$ | where the value is defined }
	Var_Use	–	{ n $\in$ $\langle$NODE$\rangle$ | where the value is used }
	Indv_Def	–	{ n $\in$ $\langle$N:*oper*$\rangle$ | assignment within loop }
	Indv_Use	–	{ n $\in$ $\langle$NODE$\rangle$ | usage within loop }
	Time_Beg	–	{ n $\in$ $\langle$NODE$\rangle$ | time constraint begin node }
	Time_End	–	{ n $\in$ $\langle$NODE$\rangle$ | time constraint end node }

An example of the edges is shown in figure 5.3. Notice that *back_edge* which denotes the data dependence of an inductive variable points in the opposite direction to the program flow, ($node_6 \rightarrow node_3$) in the figure. At the same time, *ctrl_edges* which denote the control dependence emerge from the control node, $node_2$, only. Consider again the example in figure 5.2 which has control and data edges only. The numerous *data_edges* reflect the fact that the variables { x, y } are used widely in the description. *Ctrl_edges* which emerge from $node_4$ go to { $node_5$ $node_6$ $node_7$ $node_{12}$ $node_{13}$ } and those from $node_7$ go to { $node_8$ $node_9$



```
1    a = 0;
2    loop
3       c := n > b;
4       exit when (c);
5       a := a + b;
6       n := n - 1;
     end loop;
7    A := a;
```

Figure 5.3: An Example of the Edges.

$node_{10}$ $node_{11}$ } respectively. For $node_{14}$, since its input variables are { m, n }, it naturally depends on { $node_8$ $node_9$ $node_{10}$ $node_{11}$ $node_{12}$ $node_{13}$ } where { m, n } are defined. Before we go on to the dependent sets, it is clear that besides the *data_edges*, the execution order is determined by the *ctrl_edges* as well.

5.2 Precedent and Successive Sets

To establish the data dependencies, we will require the following definitions associated with the control-data flow graph.

Definition 5.5 The **precedent set** of a node, $node_i$, is defined as:
$\text{PreN}(node_i) \overset{def}{=} \{\ node \mid node \in \langle \text{NODE} \rangle \wedge (node \rightarrow node_i)\ \} \subseteq \langle \text{NODE} \rangle$
$\text{PreE}(node_i) \overset{def}{=} \{\ edge \mid edge \in \langle \text{EDGE} \rangle \wedge \text{target}(edge) = node_i\ \} \subseteq \langle \text{EDGE} \rangle$

Definition 5.6 The **successive set** of a node, $node_i$, is defined as:
$\text{SucN}(node_i) \overset{def}{=} \{\ node \mid node \in \langle \text{NODE} \rangle \wedge (node_i \rightarrow node)\ \} \subseteq \langle \text{NODE} \rangle$
$\text{SucE}(node_i) \overset{def}{=} \{\ edge \mid edge \in \langle \text{EDGE} \rangle \wedge \text{begin}(edge) = node_i\ \} \subseteq \langle \text{EDGE} \rangle$

It is easy to see that

$$edge \in \langle \text{E:}ctrl \rangle\ \forall\ edge \in \text{SucE}(node_i),\ \text{iff}\ node_i \in \langle \text{N:}ctrl \rangle.$$

This is because edges going out of a control node can only be control edges. Similarly, edges going out of an operation node can only be data edges.

$$edge \in \langle \text{E:}data \rangle\ \forall\ edge \in \text{Sucs}(node_i), \text{ iff } node_i \in \langle \text{N:}oper \rangle.$$

However, edges going into control or operation nodes are a collection of control and data edges. Data edge(s) are essential for a control node because to evaluate the conditional expression, data value(s) are needed. Following on from definition 5.5 and 5.6, we have the following definitions:

Definition 5.7 The **data precedent set** of a node, $node_i$
$\text{Data\_PreN}(node_i) \overset{def}{=} \langle \text{N:}oper \rangle \bigcap \text{PreN}(node_i)$
$\text{Data\_PreE}(node_i) \overset{def}{=} \langle \text{E:}data \rangle \bigcap \text{PreE}(node_i)$

Definition 5.8 The **ctrl precedent set** of a node, $node_i$
$\text{Ctrl\_PreN}(node_i) \overset{def}{=} \langle \text{N:}ctrl \rangle \bigcap \text{PreN}(node_i)$
$\text{Ctrl\_PreE}(node_i) \overset{def}{=} \langle \text{E:}ctrl \rangle \bigcap \text{PreE}(node_i)$

Definition 5.9 The **data successive set** of a node, $node_i$
$\text{Data\_SucN}(node_i) \overset{def}{=} \langle \text{N:}oper \rangle \bigcap \text{SucN}(node_i)$
$\text{Data\_SucE}(node_i) \overset{def}{=} \langle \text{E:}data \rangle \bigcap \text{SucE}(node_i)$
where
$\text{Data\_SucN}(node_i) = \text{SucN}(node_i)$ and
$\text{Data\_SucE}(node_i) = \text{SucE}(node_i)$ if $node \in \langle \text{N:}oper \rangle$.
$\text{Data\_SucN}(node_i) = \emptyset \bigwedge$
$\text{Data\_SucE}(node_i) = \emptyset$ if $node \in \langle \text{N:}ctrl \rangle$.

Definition 5.10 The **ctrl successive set** of a node, $node_i$
$\text{Ctrl\_SucN}(node_i) \overset{def}{=} \langle \text{N:}ctrl \rangle \bigcap \text{SucN}(node_i)$
$\text{Ctrl\_SucE}(node_i) \overset{def}{=} \langle \text{E:}ctrl \rangle \bigcap \text{SucE}(node_i)$
where
$\text{Ctrl\_SucN}(node_i) = \text{SucN}(node_i)$ and
$\text{Ctrl\_SucE}(node_i) = \text{SucE}(node_i)$ if $node \in \langle \text{N:}ctrl \rangle$.
$\text{Ctrl\_SucN}(node_i) = \emptyset \bigwedge$
$\text{Ctrl\_SucE}(node_i) = \emptyset$ if $node \in \langle \text{N:}oper \rangle$.

Using definitions 5.1 to 5.3 and 5.7 to 5.10, we can establish a collection of attributes for each node.

Definition 5.11 Each *node* $\in \langle$NODE$\rangle$ has the following fields:

1.	Identity	:	node identifier $\in \{ 1 .. \mid \langle$NODE$\rangle \mid \}$
2.	Type	:	type $\in$ { fork | loop | exit | call | stat }
			if $\in \langle$N:*stat*$\rangle$ then type $\in$ { arith | logic | trans | port | array }
3.	Operation	:	for *node* $\in \langle$N:*stat*$\rangle$, the operation which it performs
4.	Block	:	Basic block identifier
5.	Control	:	Ctrl_PreN(*node*), Ctrl_PreE(*node*)
6.	Pre_Set	:	Data_PreN(*node*), Data_PreE(*node*)
7.	Suc_Set	:	Ctrl_SucN(*node*), Ctrl_SucE(*node*) or
			Data_SucN(*node*), Data_SucE(*node*)

For instance, the attributes of $node_6$ and $node_8$ in figure 5.2 are

1.	Identity	:	7	1.	Identity	:	8
2.	Type	:	fork	2.	Type	:	stat:arith
3.	Operation	:	(*nil*)	3.	Operation	:	+
4.	Block	:	2	4.	Block	:	3
5.	Control	:	{ 4 }	5.	Control	:	{ 7 }
6.	Pre_Set	:	{ 6 }	6.	Pre_Set	:	{ 2 5}
7.	Suc_Set	:	{ 8 9 10 11 }	7.	Suc_Set	:	{ 14 }

Since nodes within the same basic block will have the same *Block* identifier, and nodes within the same control block will have the same *Control* set, they can be used to check the relation between two nodes. Finally, there are dependence relationships with nodes.

Definition 5.12 $Node_i$ is said to be **data dependent** on $node_d$, $(Node_i \overset{data}{\longleftarrow} Node_d)$, iff

$$\exists \text{ a path from } node_d \text{ to } node_i \ s.t. \ \forall \ edge \in \text{path}, \ edge \in \langle \text{E}:data \rangle.$$

Definition 5.13 $Node_i$ is said to be **control dependent** on $node_c$, $(Node_i \overset{ctrl}{\longleftarrow} Node_c)$, iff

$$\exists \ node_j \in \text{Ctrl\_SucN}(node_c) \ s.t. \ \exists \text{ a path from } node_j \text{ to } node_i.$$

For example, $node_{10}$ is data dependent on { $node_1$ $node_2$ $node_5$ } and control dependent on { $node_4$ $node_7$ }. At the same time, $node_{15}$ is data dependent on all the nodes except { $node_3$ $node_6$ $node_4$ $node_7$ } and control dependent on { $node_4$ $node_7$ }.

The original *CDFG* consists of data dependencies and control dependencies. By manipulating these dependencies, various interesting transformations can be derived.

5.3 Simplification

Simplification preserves the control block structure of the original description. The aim is to simplify the dependencies of the nodes. The simplified graph is referred to as the *Program Flow Graph*(*PFG*). *PFG* is a simple representation of the

```
1       x := Xi;
2       y := Yi;
3       a := y > 0;
4       case a is
         when "1" =>
5           p := y + x;
6           b := x > y;
7           case b is
             when "1" =>
8              m := y + p;
9              n := x  /  2;
             when "0" =>
10             m := x + p;
11              n := y * 2;
            end case;
         when "0" =>
12          m := x + y;
13          n := 0;
        end case;
14      z := m + n;
15      Zout := z;
```

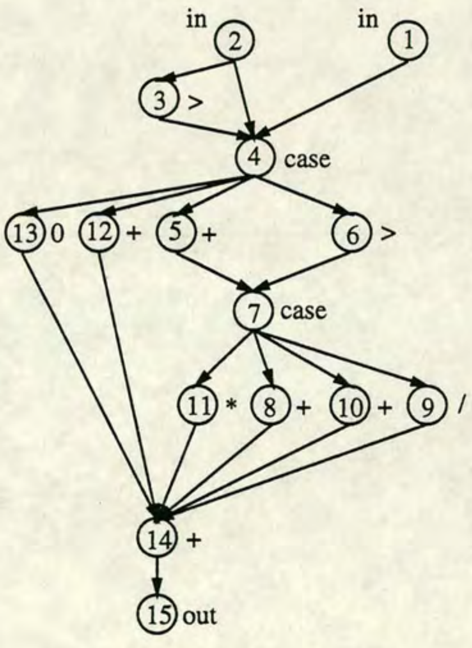


Figure 5.4: The Original Description and the Simplified *CDFG* , *PFG* .

description. It shows the most basic execution sequence. Operations are bound by the control nodes which can be referenced back to the control structures of the original description. However, local parallelism governed by data dependencies within basic blocks is represented explicitly. The *PFG* of *so*Π*o*η is shown in figure 5.4. On comparison with the original description, the graph clearly reflects the structure of the description, the nested condition branches with $node_4$ on top of $node_7$. In addition, $node_4$ and $node_7$ will shield the operations inside their control block from receiving any data. Every control body has only one entrance which will only be activated when all the data required by the operations inside it are valid.

For instance, in figure 5.4, the operations of { $node_8$, $node_9$, $node_{10}$ $node_{11}$ } are bound by the control node $node_7$ which in turn is governed by the input data dependence of the basic blocks. On the other hand, parallelism within the basic block, like { $node_8$, $node_9$ } and some others are shown clearly.

When scheduling is applied on this graph, the schedule sequence will be more-or-less the same as the PFG. The nested control structure will be kept, and hence will be evaluated sequentially; from upper level, $node_4$, to lower level, $node_7$.

To transform the $CDFG$ into PFG, the following sequence

```
• Apply(1)  $\forall\, node \in \langle\textsc{Node}\rangle$,
            $\forall\, n \in \{\, n \mid (n \xleftarrow{data} node)\,\}$
              if $\exists\ (nc \rightarrow n)\ s.t.\ nc \in \langle\text{N:}ctrl\rangle$
                then generate($node \rightarrow nc$).
• Apply(2)  $\forall\ (node_a \rightarrow node_b) \in \langle\textsc{Edge}\rangle$,
            if $\exists\ path$ from $node_a$ to $node_b$ with length($path$) $> 1$
              then eliminate($path$).
```

can be applied recursively until the graph reaches its simplest form. Apply(1) is to direct the dataflow to the single control entry, so that the dataflow is guarded; for example adding edges to $node_4$, ($\{node_1\, node_2\} \rightarrow node_4$) and edges to $node_7$ ($\{node_5\, node_6\} \rightarrow node_7$). Apply(2) is to eliminate any redundant edges, especially those data edges from $node_1$, ($node_1 \rightarrow \{node_5\, node_6\, node_9\, node_{10}\, node_{12}\}$) and those from $node_2$, ($node_2 \rightarrow \{node_5\, node_6\, node_{11}\, node_{12}\}$).

5.4 Temporary Variables

The aim of transformation is to extract as much fine-gain parallelism within the graph as possible. Therefore, it is important to identify the areas with potential parallelism. Then transformations can be applied to extract them. To identify the potential areas, we need to look at the properties exhibited by temporary variables.

Definition 5.14 Temporary variable (tv) is a variable which is defined and used (born and dead) within a basic block. Temporary variable assignment is a node, $node_{tv}$, $\in \langle$N:*oper*$\rangle$ which assigns a value to a temporary variable, tv, such that Define(tv) = $\{ node_{tv} \}$.

Since a temporary variable is defined and used within a basic block, all the nodes in Define(tv) and Use(tv) are inside the same basic block.

$$\text{Block}(node) = \text{Block}(node_{tv}) \; \forall \; node \in \text{Use}(tv).$$

Based on this definition, $node_6$, in figure 5.2, is a temporary variable assignment because

$$\text{Use}(b) = \{ \; node_7 \; \}, \text{Block}(node_7) = \text{Block}(node_6)$$

Transformation 5.15 Code motion with sequential statements.
For an acyclic, loop-free (without *back_edge*) sub-graph, temporary variable assignment can be performed out of sequence, provided that the data dependencies are satisfied and the inputs for the operation are available.

For instance, assume listing 5.16a is a basic block,

Listing 5.16

```
s1:  a_1 := x + a;              s3:  t_1 := x + y;
s2:  b_1 := y + b;              s1:  a_1 := x + a;
s3:  t_1 := x + y;              s2:  b_1 := y + b;
s4:  sum := a_1 + b_1;          s4:  sum := a_1 + b_1;
s5:  reg := t_1 + sum;          s5:  reg := t_1 + sum;
            (a)                             (b)
```

and { a_1 b_1 t_1 } are temporary variables, the temporary variable assignments, { s1 s2 s3 }, can be executed in any order. Actually, provided the input dependencies are maintained, the execution order of the statements inside a basic block can be re-arranged to take advantage of the available resources. In the above description, { s1 s2 s3 } could be executed in parallel if 3 adders were available, or combinations such as { s1 s2 }, { s3 s4 }, etc could be used with less resources.

This potential is captured explicitly in the original $CDFG$ and also in the simplified PFG.

Transformation 5.17 Code motion with condition statement.
For an acyclic sub-graph, control dependence on temporary variable assignment can be ignored. This is because the effect of the assignment will only affect the temporary variable. Wherever it is defined, it will only be used inside its original basic block. However, when a temporary variable assignment is moved out of its original basic block, it is no longer a temporary variable assignment and the assigned variable is no longer a temporary variable.

For example, *tmp* in figure 5.5a is a temporary variable. When the assignment is moved out of the control block, as in (b), its execution is no longer guarded by the condition node and the data dependence of it will cross the control block boundary.

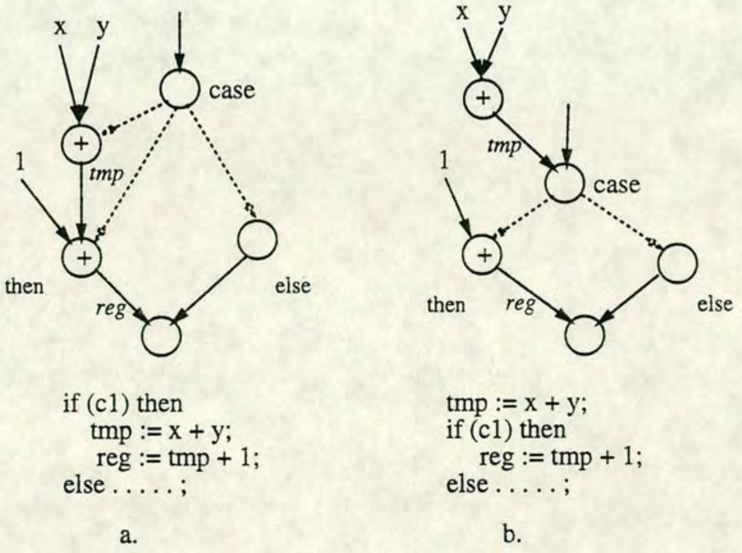


Figure 5.5: Code notion with Temporary Variable Assignment.

5.5 Local-Data Transformation

Instead of waiting until all the data required by the control block is valid, the conditional expression can be evaluated whenever the values it requires are ready. Then, branching can be performed. The execution of the operations inside the control block can wait until their inputs are available.

Transformation 5.18 Parallel branching: nested condition statements Nested condition statements can be merged into one if data dependence is preserved. In this case, the condition expressions can be evaluated and branched together. In particular when a pipelined control unit is used, the branch overhead is significant. Parallel branching can improve the performance. Consider the following,

Listing 5.19

```
if (Cond_1) then
  Block_0;
  if (Cond_2)
     then Block_1;
     else Block_2;
  end if;
else Block_3;
end if;
```

(a)

```
case (Cond_1 // Cond_2)
when "11" ⇒ Block_0; Block_1;
when "10" ⇒ Block_0; Block_2;
when "0X"⇒ Block_3;
end case;
```

(b)

If the condition expression $Cond_2$ does not depend on any value defined in $Block_0$, then it can be evaluated in parallel with the first condition expression $Cond_1$. Sometimes it may not need to be evaluated at all. Adjustment of the test condition can be made to compensate for this effect. The "don't care" condition present in the above example can easily be accomplished during the synthesis of the controller.

Transformation 5.20 Parallel branching: disjoint condition statements Disjoint conditional statements can similarly be grouped into one unit. In this case, if the latter condition expression, $Cond_2$ in listing 5.21a, does not depend on the values defined in the control block of the first one, $Cond_1$, they can be evaluated in parallel. However, this time we need to consider all the possible paths through the two condition statements. Listing 5.21 demonstrates this case, here the basic blocks originally inside the control blocks are combined into the different branches.

Listing 5.21

```
if ($Cond_1$)                   case ($Cond_1$ // $Cond_2$)
   then $Block_0$;              when "11" ⇒ $Block_0$; $Block_2$;
   else  $Block_1$;             when "10" ⇒ $Block_0$; $Block_3$;
end if;                         when "01" ⇒ $Block_1$; $Block_2$;
if ($Cond_2$)                   when "00" ⇒ $Block_1$; $Block_3$;
   then $Block_2$;              end case;
   else  $Block_3$;
end if;
          (a)                              (b)
```

Although this appears very complicated when performed on the description, applied on the *CDFG* , it is much simpler. Figures 5.6 and 5.7 depict the two transformations. For figure 5.6, we assume that there are data dependencies between $Block_0$ and $Block_1$, $Block_2$. Transformation 5.18, parallel branching with nested condition statements, propagates the control edge, $(Cond_1 \to Cond_2)$, from $Cond_1$ to the descendants of $Cond_2$,

- Apply(1) eliminate $(Cond_1 \to Cond_2)$,
- Apply(2) $\forall$ *node* $\in$ Ctrl_SucN($Cond_2$), generate $(Cond_1 \to node)$.

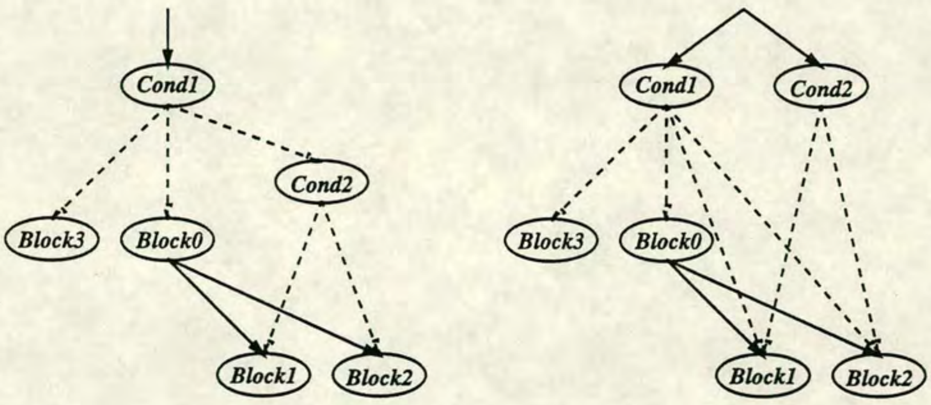


Figure 5.6: Parallel Branching with Nested Condition Statements.

Note that although the two conditions are evaluated in parallel, $Block_0$ is still bound by $Cond_1$ only and not by $Cond_2$. For transformation 5.20, parallel branching with disjoint condition statements, we again assume that there are data depen-

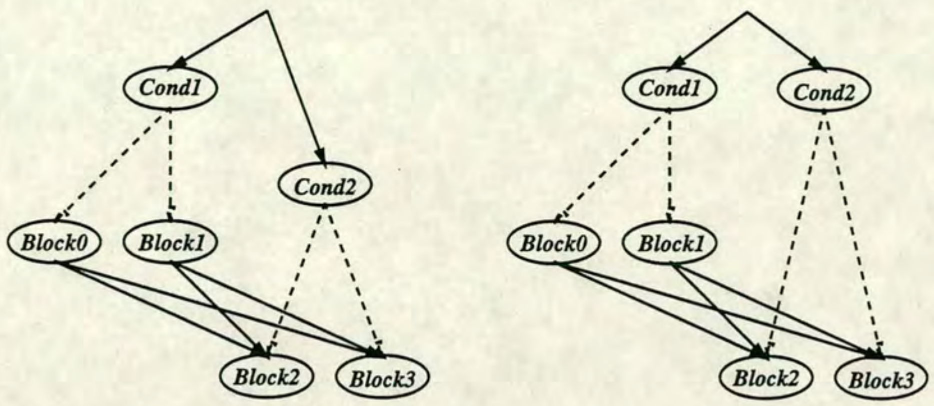
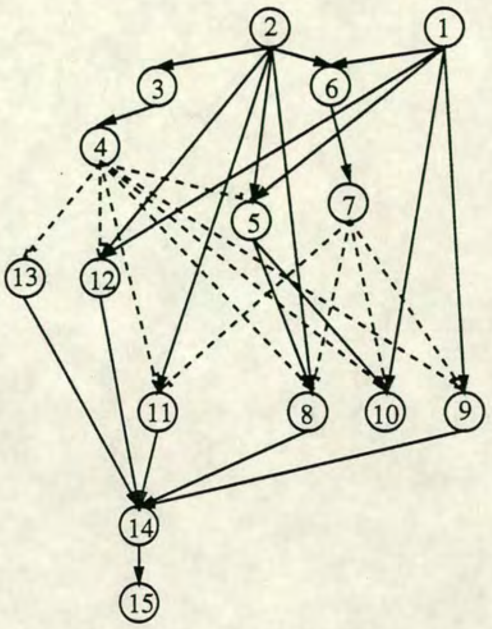


Figure 5.7: Parallel Branching with Disjointed Condition Statements.

dencies between the operations in the control blocks of the two disjointed condition statements. Since $Cond_2$ does not depend on the operations in $Block_0$ and $Block_1$, it can be evaluated as soon as possible and in parallel with $Cond_1$. No extra edge is required as depicted in figure 5.7. The data dependence ensures that $Block_0$ or $Block_1$ will be executed first, the results from which will be passed either to $Block_2$ or $Block_3$.

From the definition of a temporary variable, 5.14, we can identify that b as assigned by $node_6$ in the example *so*Π*o*η is a temporary variable. By applying transformation 5.17, code motion with condition statement, and transformation 5.18, parallel branching with nested condition we get the graph in figure 5.8.
In summary the transformation scheme is:

```
• Apply(1)   identify all the temporary variable assignment, $node_{vt}$.
• Apply(2)   check for transformation 5.17: code motion with condition statement
             $\forall\ node_c \in \text{Ctrl\_PreN}(node_{vt})$
               eliminate $(node_c \rightarrow node_{vt})$.
• Apply(3)   check for parallel branching transformation 5.18 and 5.20
             $\forall\ node_c \in \langle \text{N:}fork\rangle \bigcup \langle \text{N:}exit\rangle$
               if $\exists\ n_c \in \text{Ctrl\_SucN}(node_c)\ s.t.\ n_c \in \langle \text{N:}fork\rangle \bigcup \langle \text{N:}exit\rangle$
                 then eliminate $(node_c \rightarrow n_c)$
                        $\forall\ n \in \text{Ctrl\_SucN}(n_c)$ generate $(node_c \rightarrow n)$.
```



```
1       x := Xi;
2       y := Yi;
3       a := y > 0;
6       b := x > y;
4,7     case (a // b) is
         when "11" =>
5            p := y + x;
8           m := y + p;
9            n := x / 2;
        when "10" =>
5            p := y + x;
10          m := x + p;
11           n := y * 2;
        when "0X" =>
12          m := x + y;
13           n := 0;
         end case;
14      z := m + n;
15      Zout := z;
```

Figure 5.8: Local-Data transformation and the Resultant Description.

These can be applied recursively until no further transformation can be placed.

5.5.1 Global-Data Transformation

This method is the most radical. It tries to extract all the parallelism in the description. First, the following transformation 5.22 is applied to all operation statements.

Transformation 5.22 Statement splitting
The aim is to separate the *computation* part and the *assignment* part of the operation by introducing temporary variables.

Listing 5.23

```
4.   x := a + b;              4a.   t := a + b;
                              4b.   x := t;
         (a)                        (b)
```

When this transformation is applied repeatedly, a lot of temporary variables will be generated. For the example $so\Pi o\eta$, figure 5.9 is generated by the application of statement splitting. Subsequently, code motion is also applied to move all the

```
1      x := xi;
2      y := yi;
3      a := y > 0;
4      case a is
        when "1" =>
5a        it1 := y + x;
5b          p := it1;
6           b := x > y;
7         case b is
           when "1" =>
8a           it2 := y + p;
8b            m := it2;
9a           it3 := x / 2;
9b            n := it3;
           when "0" =>
10a          it4 := x + p;
10b           m := it4;
11a          it5 := y * 2;
11b           n := it5;
          end case;
        when "0" =>
12a       it6 := x + y;
12b       m := it6;
13          n := 0;
        end case;
14     z := m + n;
15     Zout := z;
```

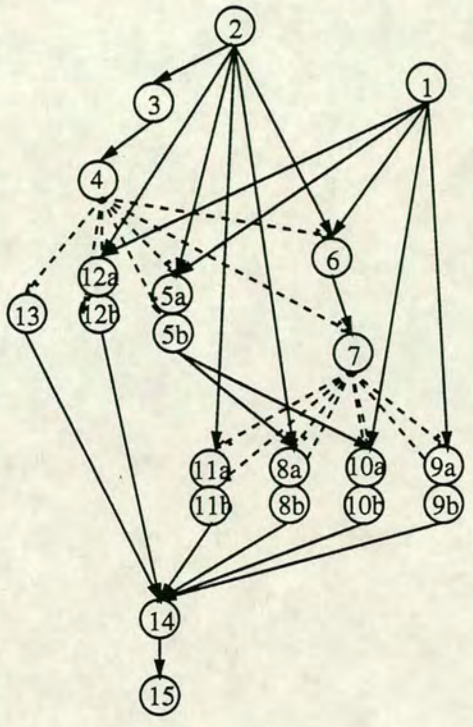


Figure 5.9: Operation Decomposition.

computation parts out of the control blocks.

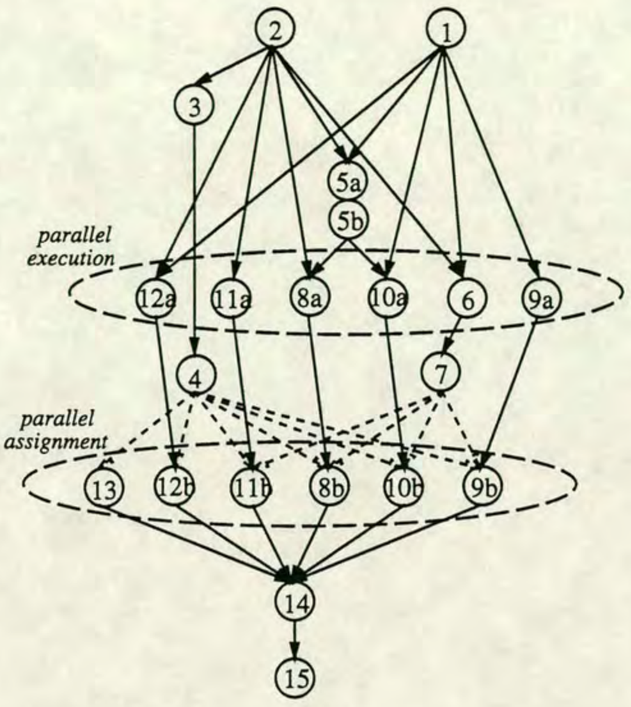
Transformation 5.24 Global speculative evaluation

For a control block to which transformation statement splitting, 5.22, and parallel branching, 5.18 5.20, are applied, the block can be de-structured and divided into two sections:

- computation section: the operation parts of the transformed statements.
- assignment section: the assignments parts guarded by the conditional expressions.

As shown in figure 5.10 the computation section, from statement 1 to 12a, consists only of computation statements. After statement 4,7, there are the selective assignments where temporary variables transfer their values to the corresponding variables when the right conditions are met.

It is important to notice that computations are performed without regard to the outcome of the condition expressions. By storing the results in temporary vari-



```
1       x := xi;
2       y := yi;
3       a := y > 0;
5a      it1 := y + x;
5b      p := it1;
6       b := x > y;
8a      it2 := y + p;
9a      it3 := x / 2;
10a     it4 := x + p;
11a     it5 := y * 2;
12a     it6 := x + y;
4,7     case (a // b) is
        when "11" =>
8b        m := it2;
9b        n := it3;
        when "10" =>
10b       m := it4;
11b       n := it5;
        when "0X" =>
12b       m := it6;
13        n := 0;
        end case;
14      z := m + n;
15      Zout := z;
```

Figure 5.10: Global-Data Transformation and the Resultant Description.

ables, the guarding conditions can be ignored. Then, later, when the conditions are valid, they are returned to their original variables. In this way, the computation section will contain a high degree of parallelism. As in figure 5.10, { 12a 11a 8a 10a 6 9a }, these can all be executed in parallel. It is probably more than sufficient to keep all the resources busy. By using parallel branching, the selected assignment will only involve very short delays and can handle a large number of parallel assignments, { 13 12b 11b 8b 10b 9b }.

It is very good to obtain all these potential parallel-executable computations. However, it seems that we have reached a stage where we have more parallelism than the hardware can digest. The advantage offered by mutually exclusive conditions, such as resource sharing, are also sacificed. As a result, the global-data transformation will require more resources than should be needed and thus cause a waste of resources. On the other hand, if we retreat back to the conventional style in section 5.3, we will lose the flexibility offered by high level synthesis. The key issue is the condition expressions in the condition statements. A method is needed to represent the guarding conditions so that it is possible to release parallelism when there is abundant resources, and to support resources sharing when there is not enough.

5.6 Delay Representation of Functional Unit

In order to handle the delay of different types of functional unit efficiently, we need a method to describe their operational properties. Functional units can be combinational or sequential; ranging from as simple as a NAND gate to as complex as a processing array. The method must be able to handle the wide variety of functional unit easily.

Definition 5.25 The method we used is *operation triplet.* Operation triplet is a union of three timing related properties:

{ delay : latency : cycle-time }

delay	—	The time between the input initiation and the availability of the result.
latency	—	The minimum number of cycles that must beelapse before a new input can be initiated again.
cycle-time	—	The minimum clock period required for correct operation of the functional unit.

Combination circuits will have the delay entity only, the latency and the cycle-time will be ignored. On the other hand, for sequential elements, all three are needed.

Definition 5.26 The delay entity is described as **delay-cycle-delay**. Delay-cycle-delay is a summary of three variables describing the delay position of the functional unit related to the cycle boundary, figure 5.11.

$$([\ x\ ] + [\ y\ ]\ \mathrm{c} + [\ z\ ])$$

x	—	leading-time, the delay at the input before the first latch(ns).
y	—	the number of internal stages($cycle$).
z	—	trailing-time, the delay at the output after the last latch(ns)

For example, if a functional unit has combinational logic in front of its input latches and after its output ones, then, x will be the delay of the input logic, y will be the delay in terms of cycles and z will be the delay of the output logic.

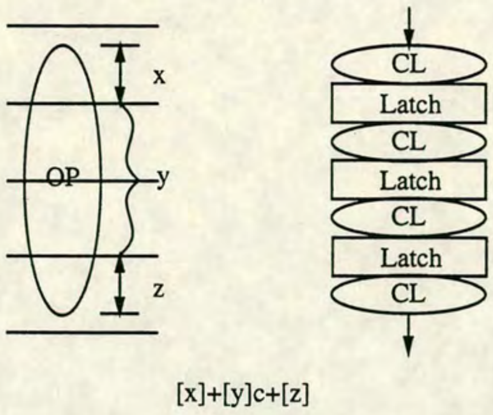


Figure 5.11: The "delay-cycle-delay" notation of a Functional Unit.

Also, the setup and hold time of a register can be represented as $(5 + 0c + 5)$, where y is assigned zero. It means that the delay is crossing a cycle boundary. A similar notation can also be used for the delay of a finite state machine controller. Other examples: the operation triplet of a pipeline functional unit with 5 cycles delay and 2 cycle latency will be:

$$\{\ (10 + 5c + 20) : 2cycles : 30ns\ \}$$

while a simple adder will be:

$$\{\ (40 + 0c + 0) : - : -\ \}.$$

5.7 State Overhead

To avoid delay faults, the length of a cycle must be long enough to accomplish all the transfers, logic and data operations between adjacent storage elements. A cycle can consist of:

$$\text{Regr} \longrightarrow \text{Bus} \longrightarrow \text{Mux} \longrightarrow \text{ALU} \longrightarrow \text{Bus} \longrightarrow \text{Mux} \longrightarrow \text{Regr};$$

The input transfers from registers(Regr) through buses(Bus) and multiplexors(Mux) to the functional unit(ALU); the output transfers from the functional unit through buses and multiplexors back to the registers. For an operation executed in a cycle, as in figure 5.12a, we have:

Definition 5.27 State setup time – time from clock edge to the start of the operation. This accounts for the functional unit input multiplexor delay, and the control signal generation delay.

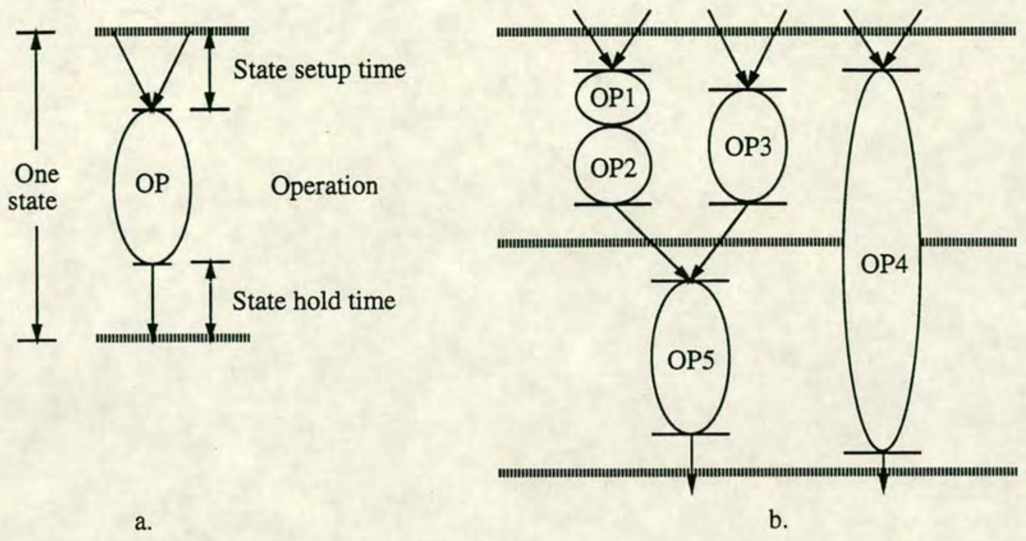


Figure 5.12: State Setup and Hold Time.

Definition 5.28 State hold time – time from the end of the operation to the next clock edge. It consists of the register input multiplexor delay, and the control state transition delay.

Since resource constraints are given, it is possible to estimate the upper bound delay of the multiplexor tree required for each kind of resource. When operations are scheduled with the schemes of multi-cycling and chaining, state setup and hold times are affected according to the boundary conditions. For operations which are scheduled in multi-cycle, such as OP4 in figure 5.12b, the state hold time will apply to the cycle where the execution ends. In the case of operation chaining, such as { OP1 OP2 } in figure 5.12b, the state hold time of the first operation and the state setup time of the following one will be adjusted to account for the transmission and multiplexor delay only.

Chapter 6

Resources Restricted Scheduling

6.1 Introduction

As fabrication technology has continued to grow, we now have the capability to put more functional units on a single chip. In order to maximize the efficiency of a design, resources must be utilized as fully as possible. In the context of high-level synthesis, in order to fit a design within an allowable silicon area, a significant degree of resource re-usage is required. Within the search for an optimal area-time trade-off, area must be used as efficiently as possible. These challenges call for a scheduling methodology which is intelligent enough to extract the greatest timing advantage out of the available resources.

In this chapter, we describe a scheduling methodology for high-level synthesis of designs with a significant amount of control structures. The objective is to utilize all the available resources while scheduling with respect to resource restriction. To do so, a vector/matrix structure at each node is built. It provides a global view of resource usage. It supports the migration of operations across basic blocks to wherever idle resources are available. With it, we formulate a list scheduling algorithm in which the dispatching priority changes dynamically with respect to resource availability.

6.2 Previous Work

Older scheduling algorithms which allocate operations into clock cycles were often applied only to one basic block of the behavioural description at a time. The extraction of potential parallelism from the description as a whole was thereby restricted. The fine-grain parallelism obtained was often not sufficient to keep all the functional units busy. An alternative approach has been adopted in MAHA [PARK86] in which the conditional branching structures are translated into fork-join pairs. The graph is then scheduled as a whole. Unfortunately, MAHA appears not to be able to take advantage of mutually exclusive conditions for resource sharing. Force Directed Scheduling [PAUL87] is able to explore this potential. However, in both cases, when an operation is placed inside a fork-join pair, the scheduler is not allowed to move it even though there are abundant resources elsewhere.

The percolation scheduling algorithm described in [POTA90] performs parallelisation across basic block boundaries. Starting with an optimal schedule, semantics-preserving transformations are applied repeatedly to convert the *program flow graph* into a more parallel one. The name 'percolation scheduling' reflects the style of the transformation to move data-independent operations upwards across basic block boundaries towards the top of the program graph. This technique is closely related to our work. However, it does not take into account the effect of resource restrictions.

6.3 Resource Restricted Scheduling

To overcome these problems, we have developed the *Resource Restricted Scheduling*, R^2Sch scheme. Given the input description and the resource constraints, the operations are scheduled into a minimum number of control steps. Fine-grain parallelism is extracted as much from the description as possible. This is achieved by means of aggressive migration of operations across basic block boundaries. At present, R^2Sch can handle:

- Resource constraints, including restriction on abstract buses and memory I/O,

- Scheduling with respect to resource constraints,
- Scheduling with multicycling and chaining,
- Optimisation beyond the limit of basic blocks,
- Functional units with different delays and stages of pipelining,
- Resource sharing with mutually exclusive conditions,
- State overhead (control and steering logic delay),
- Different controller styles,
- Simple timing constraints.

The main criteria of the scheduling algorithm are:

- Operations should not be restricted by basic block structures or fork-join pairs. They should be allowed to migrate anywhere within the schedule space
- Specification of resource restriction should be allowed for fast area-time trade-off investigation and can be specified either by human designer or by expert system.

The first critique is to ensure that there is enough flexibility associated with each operation so that shortcomings introduced by control boundaries can be overcome. However, when an operation migrates outside a fork-join pair, the condition under which it executes will be changed. Operations originally present in different branches will no longer be mutually exclusive. The scheduler must be able to take care of the condition changes when an operation is scheduled outside its original fork-join pair.

For the second critique, we believe that resource restriction is a more direct and economic scheduling control attribute than the delay constraint mentioned in some literature. It is at a higher level than layout area or power consumption, and hence easier to understand and manipulate. For instance, if the resource unrestricted scheduling achieves the maximum parallelism with 3 multipliers and 10 cycles delay, then by restricting the number of multipliers to 2, a new delay value can be obtained immediately. However, for a delay constraint approach, it is difficult to estimate the cycle difference in order to save a multiplier.

In order to allow operations to migrate freely within the schedule space, we have developed **condition vectors** and **resource vectors**. They represent the conditions and the resources situation of each operation. By manipulating them, we can formulate a global priority function which compares favourably with the list scheduling algorithm. In the following sections, we reveal the algorithm step-by-step. Experimental results, observations and discussion of the method are presented at the end.

The example, $so\Pi o\eta$, in figure 6.1 will be used to explain the various concepts. Figure 6.1 depicts its *Program Flow Graph* (*PFG*) in which local parallelisation is extracted in each basic block. The internal representation of $so\Pi o\eta$ takes the form of a *Control-Data Flow Graph* (*CDFG*) shown in figure 6.2. It represents the detailed control and data dependencies of each node.

```
1         x := Xi;
2         y := Yi;
3         a := y > 0;
4         case a is
           when "1" =>
5             p := y + x;
6             b := x > y;
7             case b is
               when "1" =>
8                m := y + p;
9                 n := x  / 2;
               when "0" =>
10                m := x + p;
11                 n := y * 2;
              end case;
            when "0" =>
12            m := x + y;
13            n := 0;
           end case;
14        z := m + n;
15        Zout := z;
```

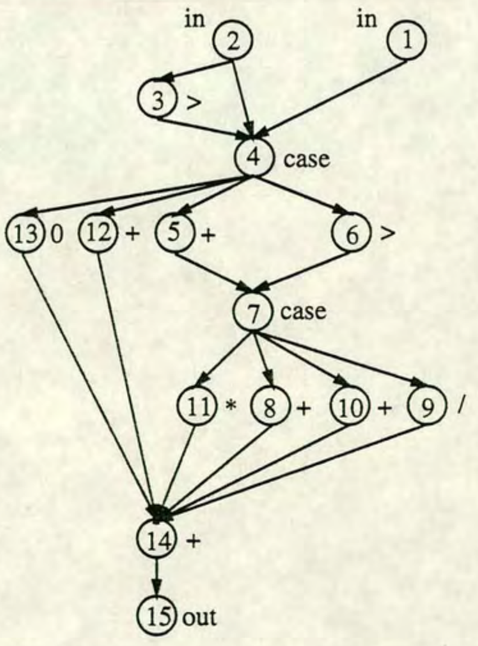


Figure 6.1: The Program Flow Graph of $so\Pi o\eta$.

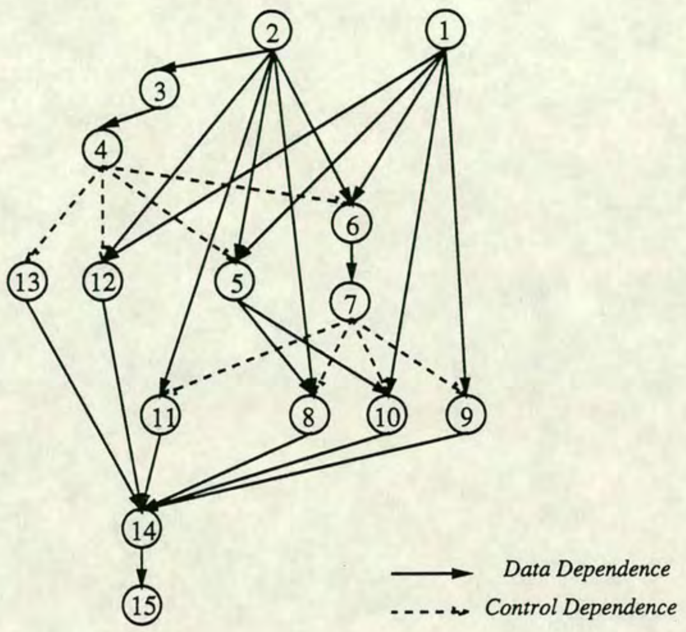


Figure 6.2: The Control, Data Flow Graph of *so*Π*o*η .

6.4 The Resource Vector

When several kinds of resource are used to implement the behaviour, it is necessary to keep track of the usage of each type of resource. To do so, we introduce the *Resource Vector*(*RV*) each element of which represents one kind of functional unit.

$$RV\,(\ FU_1,\ FU_2,\ .\ .\ .\ FU_n,\ iobus,\ iomem\ )$$

FU_i	—	functional units which are needed to implement the design
iobus	—	abstract input-output buses
iomem	—	abstract input-output memory ports

For instance, if adder/subtractor and multiply/divide units are required for implementation of the design, then an *RV* such as: *RV* ([+,−], [∗ ,/], iobus, iomem) can be formulated. Each node in the program flow graph will be assigned an *RV* representing the quantities of each resource that node requires. For the example *so*Π*o*η , in figure 6.3, assuming that adder/subtractor unit (ASU)s are allocated for (+, >)s, multiply/divide unit (MDU)s are allocated for (∗ , /)s, and concern is on functional units and I/O buses only, we have the *RV* :

RV ([+, >], [$*$, /], *iobus*), or
RV (ASU, MDU, *iobus*)

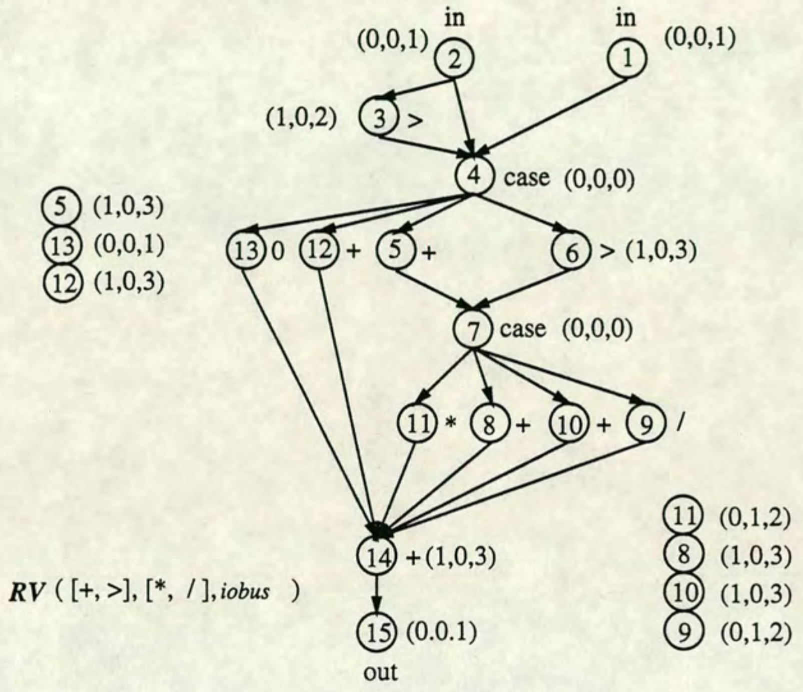


Figure 6.3: Resource Vectors of *so*Π*oη* .

The *RV* s of all the nodes in *so*Π*oη* are depicted in figure 6.3. As the requirement for resources changes, some of the entities are changed dynamically during the scheduling process.

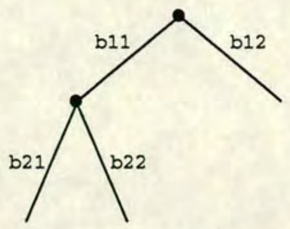
6.5 The Condition Vector

The conditional branching of an acyclic program segment is considered as a tree, *Condition-branch Tree* (*CT*). *Condition branches* are the descendants created by the *conditional statements.* The choice of branch is governed by the evaluated result of the conditional expression. Each conditional statement can have two or more branches:

if-then-else – two branches
case-when – two or more than two branches

```
if c1 then b11
   if c2 then b21
         else b22
else b12
```

(a)



```
if c1 then b11
   if c2 then b21
         else b22
   if c3 then b31
         else b32
else b12
```

(b)

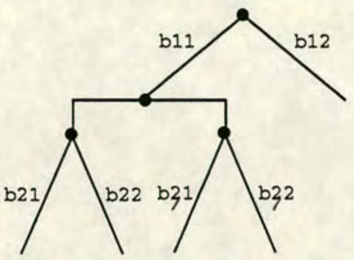


Figure 6.4: Nested and Disjointed Condition Statements

Nested conditional statements, as in figure 6.4a, will generate a descending diversified tree, while *disjointed* conditional statements, as in figure 6.4b will create a horizontally diversified tree. A leaf in the *CT* is a basic block. These branches represent the conditions under which operations will be executed. A *Condition Vector* (*CV*) is an array of bits, each of which represents the conditions under which an operation should execute. This is a very useful tool for detecting mutually exclusive operations.

6.5.1 Branch-Based Condition Vector

The condition vector contains a single entry for each branch in the *CT* and the value of this vector at each branch reflects the position of the branch in the *CT*. Within the vector there is a *sub-vector* for each conditional statement. In figure 6.5, the conditional statement with condition c_1 has a sub-vector consisting of the first two entries. Branches reached by following the first branch will have the first entry 1 and the second 0. Unrelated sub-vectors are left blank.
The complete condition vector is a concatenation of the sub-vectors from all the conditional statements:

$$[\ sbv(CS_1)\ @\ sbv(CS_2)\ @\ sbv(CS_3), \ldots\ldots sbv(CS_n)\ ].$$

For instance, in figure 6.5, $\langle Stat_{b22}\rangle$ has CV [10 01 __]. This means that it is

```
              if c1 then
[01----]         <Stat b11>
                 if c2 then
[1010--]            <Stat b21>
                 else
[1001--]            <Stat b22>
                 end if
                 if c3 then
[10--10]            <Stat b31>
                 else
[10--01]            <Stat b32>
                 end if
              else
[01----]         <Stat b12>
              end if
```

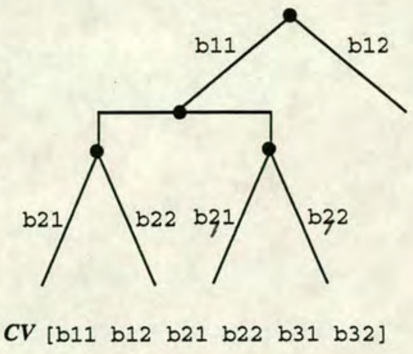


Figure 6.5: The Branch-Based Condition Vector

reachable from { $b11$ $b22$ }, unreachable from { $b12$ $b21$ } and unrelated to { $b31$ $b32$ }.

This *branch-based CV* is different in concept from a *path-based CV*. A path-based *CV* would represent the path in which the node belongs while branch-based *CV* records the conditions under which the node is executed. A branch-based *CV* is better in handling conditional structures. Irrespective of whether they are nested or disjoint, the number of entries will only increase linearly with the number of branches. In the following sections, all *CV* are referred to as branch-based *CV* only. There are two essential operations with condition vectors:

exclusive detection and vector merging.

Exclusivity Detection

For exclusivity detection, two statements are *mutually exclusive* if they belong to different branches of a fork. This fact is reflected in the *CV* s. Two vectors are mutually exclusive if they have entries that are different within their common sub-vector i.e. their common ancestry. Referring to figure 6.5, $\langle Stat_{b21}\rangle$ with *CV* [*10* 10 --] and $\langle Stat_{b12}\rangle$ with *CV* [*01* -- --] are mutually exclusive. The same also applies to $\{\langle Stat_{b21}\rangle$ [10 10 --], $\langle Stat_{b22}\rangle$ [10 01 --]$\}$ but not, for instance, to $\{\langle Stat_{b21}\rangle$ [10 10 --], $\langle Stat_{b31}\rangle$ [10 -- 10]$\}$.

Vector Merging

During scheduling, when two vectors are detected to be mutually exclusive, they can share the same resource and have their vectors merged. To merge two CV s, the common exclusive sub-vectors are OR-ed, and the remaining sub-vectors copied over. The operation is both commutative and associative. Consider $\langle Stat_{b21} \rangle$, $\langle Stat_{b22} \rangle$ and $\langle Stat_{b12} \rangle$.

$$\begin{aligned}
\langle Stat_{b21} \rangle + (\, \langle Stat_{b22} \rangle + \langle Stat_{b12} \rangle \,) &= \text{[10 10 - -] + ([10 01 - -] + [01 - - - -])} \\
&= \text{[10 10 - -] + [11 01 - -]} \\
&= \text{[11 11 - -]} \\
(\, \langle Stat_{b22} \rangle + \langle Stat_{b21} \rangle \,) + \langle Stat_{b12} \rangle &= \text{([10 01 - -] + [10 10 - -]) + [01 - - - -]} \\
&= \text{[10 11 - -] + [01 - - - -]} \\
&= \text{[11 11 - -]}
\end{aligned}$$

6.5.2 Propagated Condition Vector

In order to formulate a global scheduling algorithm, there is a need to extend the branch-based CV to reflect the global data dependencies across basic blocks, i.e. the conditions under which data is defined and used. We call the extended condition vector the *propagated condition vector* (PCV). In this case, the CV of each data define/use node is propagated forward and backward through the control and data dependence arcs. The propagated condition vectors are then *bit-wise* OR-ed with the original CV to form the PCV .

Listing 6.1

```
1    [ - - ]   [ 10 ]    b := x + y;
2    [ - - ]   [ 11 ]    case c is
                         when "1" ⇒
3    [ 10 ]    [ 10 ]      p := x + b;
                         when "0" ⇒
4    [ 01 ]    [ 01 ]      p := y + y;
                         end if;
      CV        PCV
```

Consider the code segment in listing 6.1, the PCV of $node_1$ is able to show that the definition of variable b is used only in $node_3$ but not any other branch. This property can help to enhance mutually exclusive resource sharing. Figure 6.6 depicts the PCV of $so\Pi o\eta$ after the propagation. The PCV of $node_1$ shows that its definition of x is used in all branches while the PCV of $node_5$ indicates that its definition of p is not used in the second branch (the second entry of its PCV).

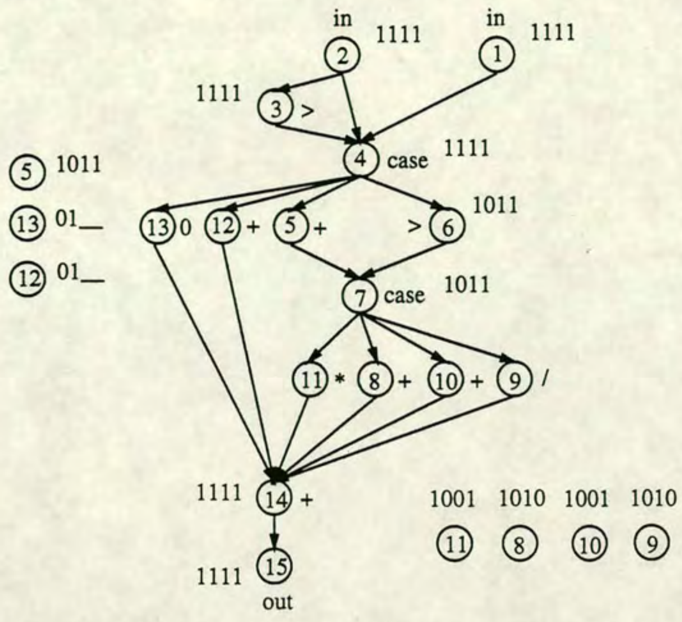


Figure 6.6: Propagated Condition Vectors(PCV) of $so\Pi o\eta$.

6.6 Resource Condition Matrix

The propagated condition vector(PCV) and the resource vector(RV) provide a powerful way to look at the utilisation of resources. By *crossing* the two vectors, we get a 2-dimensional profile, *Resource Condition Matrix* (RCM). This matrix associates the PCV of a node with every resource in its RV. It shows the usage of each resource under every condition. Using these matrices, extensive global resource sharing can be carried out. For instance, with the PCV and RV for $so\Pi o\eta$, the 2-dimensional RCM view of it is constructed in figure 6.7.

$$\begin{array}{rlrlrlrl} RCM & 1011 & RCM & 0000 & RCM & 0100 & RCM & 0100 \\ node_5 & 0000 & a. & 1111 & b. & 0000 & c. & 1111 \\ & 3033 & & 0000 & & 0300 & & 0300 \end{array}$$

The RCM of $node_5$ shows that $branch_2$ is uninhabited and no multiplier is required at all(empty second row). Therefore, the execution of $node_5$ can overlap with any node which requires a multiplier, an ALU in $branch_2$ or buses in $branch_2$, i.e. like (RCM *a*), (RCM *b*) or (RCM *c*).

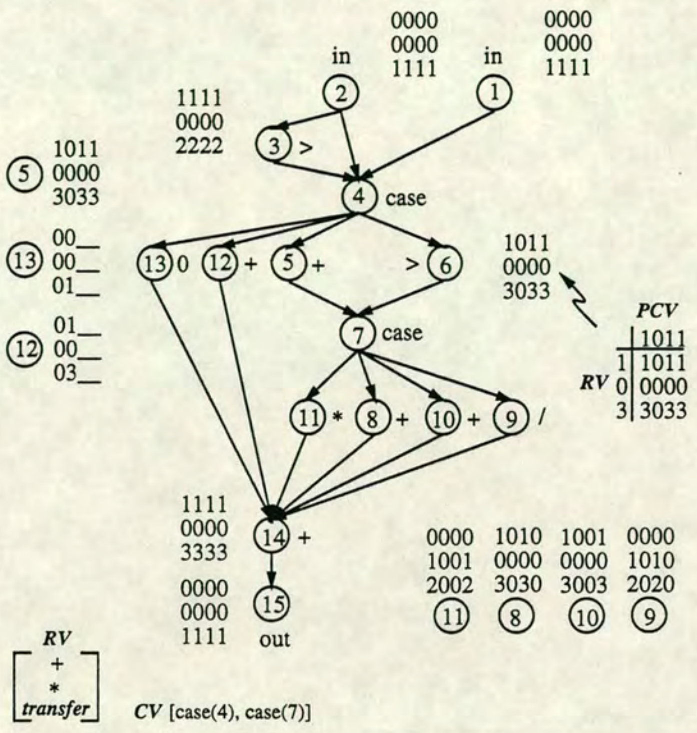


Figure 6.7: Resource Condition Matrix(RCM) of $so\Pi o\eta$.

6.7 List Scheduling

The scheduling algorithm, R^2Sch , can be divided into two phases.

1. The first phase scans the flow graph to gather information. Various static parameters like depths, path delays, CV s, PCV s and RCM s of the priority function are computed.

2. In the second phase, a list of candidates ready for scheduling is established. The most urgent one is chosen according to the priority functions (to be described in the next section). After a node is assigned, the candidate list and the resources-used vector will be updated. During the process, operations subject to data dependence constraints can migrate to wherever resources are available.

6.8 Priority Functions

Three functions are used for determining the priority for scheduling.

- Resource Priority Function
- Path Delay Function
- Depth Function

The resource priority functions of the scheduling candidates are considered first. If they are equal, then the path delay functions are applied. The depth functions will only be used when both the above functions turn out to be insufficient.

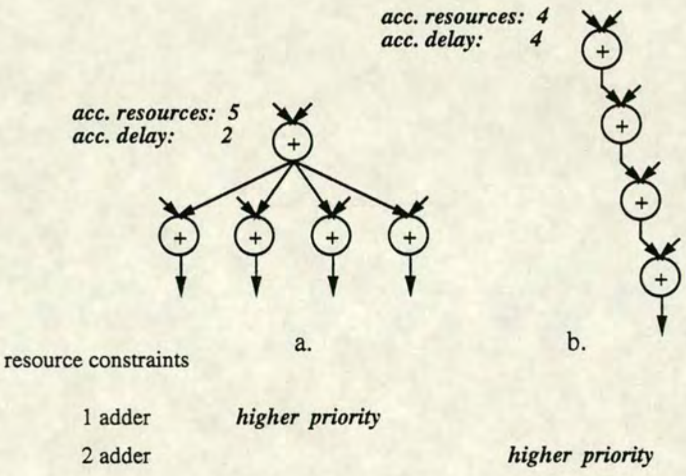


Figure 6.8: Effect of different priority functions.

Why is a composition of priority functions needed? In figure 6.8 for instance, the current node has two different paths, { a b } to the output. If only the path delay or depth function is considered, *path b* will have a higher priority. However, it will only be true if there are enough resources to realize *path*(a) in less than four levels. In particular, if only one adder is available, the above priority is clearly defective. A consideration of the accumulated resources against the resource constraints together with the accumulated delay can reflect a better priority for the whole situation.

The user-defined execution probability of the condition branches will also be considered when there are several computations competing for limited resources. The formulation of the priority function can be summarised in figure 6.9. The axes represent the *conditions*, the *resources*, and the *nodes*. The resource-condition plane

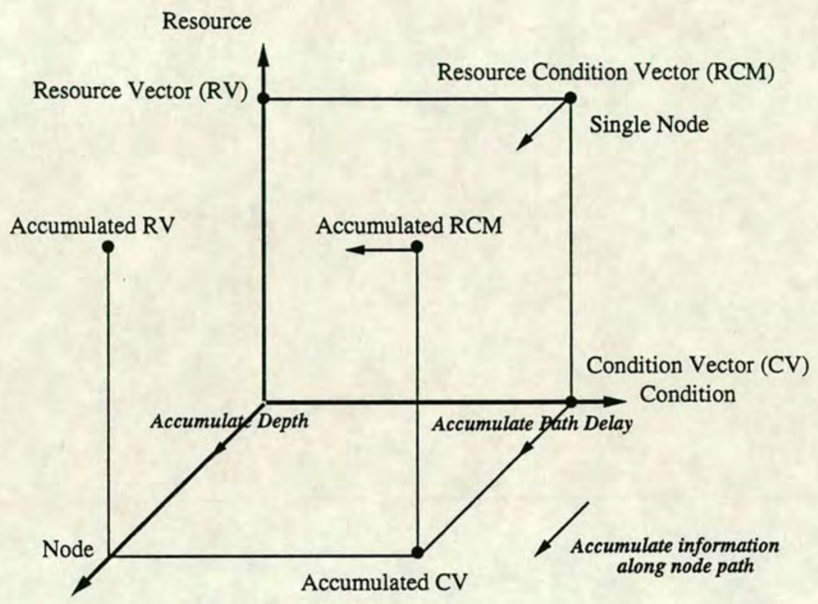


Figure 6.9: The Formulation of the Priority Functions.

represents the information associated with each node. Sliding the plane along the *node* axis, we descend the graph and accumulate information on each node. These accumulated vectors indicate the magnitude of the dependent sub-graph of each node.

6.8.1 Depth Function

The simplest of the functions, which is widely used in microcode scheduling applications. The priority is defined as the distance(node levels) away from the output. This function does not take into account the delay of each operation. No matter how complex they are, they are always assumed to take one unit of time. Therefore, it is only used when the other two priority functions are proven to be insufficient. The depth value of each node in *so*Π*o*η is shown in figure 6.10.

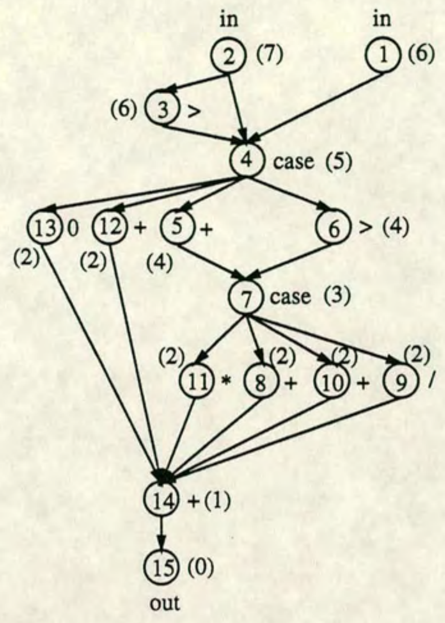


Figure 6.10: Depth Function of $so\Pi o\eta$.

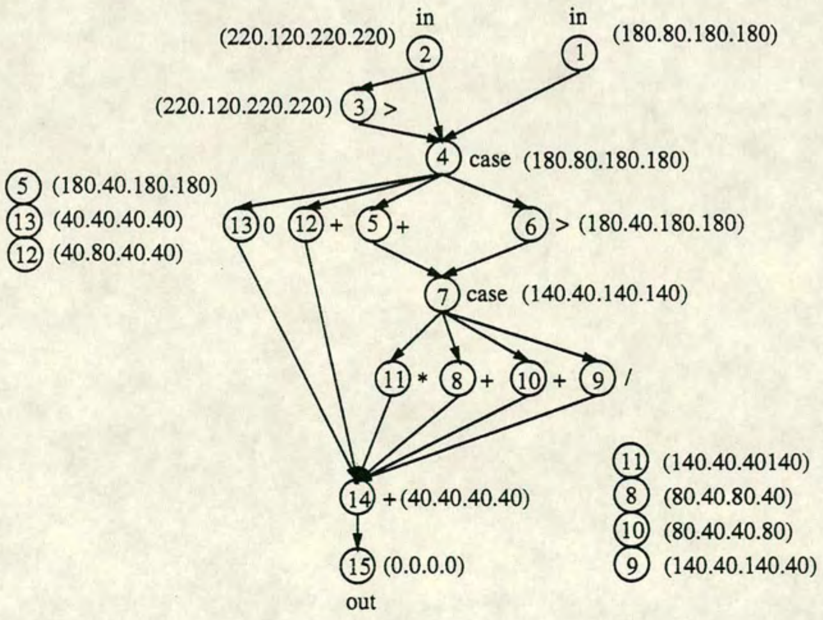


Figure 6.11: Path Delay Function of $so\Pi o\eta$.

6.8.2 Path Delay Function

This is used as priority function in Sehwa[PARK86] for pipeline scheduling. The path delay of a node, $node_k$ is defined as the longest path delay from the current node to the output. It does not however consider the effect of conditional branches. It is therefore multiplied by the condition vector to take into account the delay of different branches. It is defined as

$$\text{Path\_Delay}(node_k) = \textstyle\sum_{n_i \in P(n_i)} \text{Delay}(n_i) * CV(n_i)$$

where Delay(n_i) — the operation delay of n_i
P(n_i) — the longest delay path form (n_i) to output

If we assume that the delay of the ALU is $40ns$ and the delay of the multiplier is $100ns$, then using the *CV* of *so*$\Pi o\eta$ in figure 6.6, we have the view in figure 6.11. The path delay of $node_8$ is [80,40,80,40], i.e. the sum of $node_{14}$:[40,40,40,40] and $node_8$:[40,0,40,0]. This priority function does provide a more realistic view when different types of execution units are used for different operations. It shows how urgent each node is with respect to the timing constraints and also provides information on the delay of each branch.

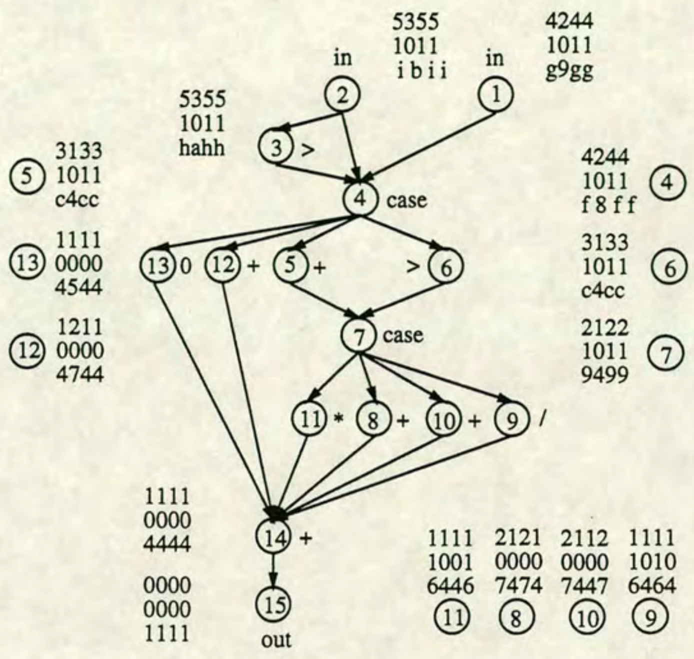
6.8.3 Resource Priority Function

In order to understand the resource priority function, we need to define the *accumulated resource condition matrix* (*ARCM*) first. The *ARCM* of a node is the accumulation of its own *RCM* and the *RCM* s from all the nodes in its descendant sub-graph. This sub-graph contains the nodes which form the paths from the current node to the output.

$$ARCM(node_k) = \textstyle\sum_{n_i \in SG(node_k)} RCM(n_i)$$

where SG($node_k$) — the sub-graph of ($node_k$).

The *ARCM* s of *so*$\Pi o\eta$ is shown in figure 6.12. The *ARCM* provides a view of the foreseeable resource requirement at each node with reference to the requirements on different kinds of functional unit and conditions. By projecting the *ARCM* onto the resource-node plane, figure 6.9, the *accumulated resource vector* (*ARV*) is formed. For instance, the *ARV* of $node_4$ is [4,1,15].



key: a-10 b-11 c-12 d-13 e-14 f-15 g-16 h-17 i-18

Figure 6.12: The Accumulated Resource Condition Matrix of $so\Pi o\eta$.

ARV shows the resource usage in the sub-graph of the current node. When under a resource constraint, it is the very information needed to determine the priority of a node.

$$\begin{aligned} Available\,Vector &= Constraint\,Vector - Used\,Vector(cycle_c) \\ Differential\,Vector &= ARV(node_k) - Available\,Vector \end{aligned}$$

The **constraint vector** is the initial resource constraint input into the scheduling algorithm. The **used vector**($cycle_c$) represents the amount of resource used so far in the current scheduling cycle, $cycle_c$. The difference, the **available vector**, gives the amount of resource still available in the current cycle. The **differential vector**, represents the *resource shortage* situation with respect to each type of functional unit. This difference from the differential vector can be weighted with the resource costs to determine the priority of a node.

The resource priority function gives a global view of the graph, the distribution of different kinds of operations and hence the requirement on different kinds of functional units.

6.9 Cycle Condition/Resource Vector

Since not all conditions in the conditional statements can be evaluated at the very beginning, to record the *evaluated* conditions during the scheduling process, for each cycle in the schedule space, there is a *Cycle Condition Vector* (*cycle-CV*). When the select(case) operation of a conditional statement, CS_i, is assigned to a cycle, the conditions for its branches will be valid. Assuming the current schedule cycle is sc, this is recorded by

$\forall$ *cycle-*CV_c with $c > sc + delay(controller)$,
record $(CV(CS_i)$, *cycle-*$CV_c)$

Analogous to *cycle-CV*, there is also *cycle resource vector* (*cycle-RV*). The *cycle-*RV_{sc} is to record the resources committed so far in the current schedule cycle, sc. It is used in formulating the resource priority function with

$$\text{Used}(sc) = cycle\text{-}RV_{sc}.$$

As *cycle-RV*s will be updated continuously as scheduling proceeds, the resource priority function will also be updated and hence, be able to reflect the priority dynamically.

6.10 Execution Condition/Resource

During the scheduling process, operations subjected to data dependence constraints can migrate to wherever resources are available. After a node ready to be scheduled is put into the candidate list, the order of migration is determined by the priority function. When an operation migrates outside its basic block, the control condition will no longer be the same. This leads to changes in the conditions under which it executes and thereby its resources sharing opportunity. In order to keep track of these changes, the **execution condition vector** (*ECV*) and the **execution resource vector** (*ERV*) are introduced.

While *PCV* assumes that all the conditions are available before any execution, *ECV* represents the actual conditions under which a node is executed. *ECV* is derived from the *PCV* of the operation and the *cycle-CV* of the current schedule cycle, *sc*.

$$\begin{aligned} ECV\,(node_i) &= PCV\,(node_i)\ masked\_by\ cycle\text{-}CV_{sc} \\ [\,10\ \_\_\ \_\_\,] &= [\,10\ \_\_\ 01\,]\ masked\_by\ [\,11\ 11\ \_\_\,] \end{aligned}$$

Due to resource sharing in mutually exclusive conditions, the resource requirement of a node in the current schedule cycle may be less than what it requires in a stand-alone situation. Thus, the *ERV* reflects the actual resource requirement by taking away resources in the *RV* which could be shared exclusively with the nodes already assigned in the cycle.

6.11 Scheduling

The second phase of R^2Sch is to perform the scheduling. Nodes are scheduled according to their data flow dependence. For the nodes in $node\_list_{sc}$ ready to be scheduled in cycle, *sc*,

```
$\forall\ n \in node\_list_{sc}$
    $compute\_ECV(n, cycle\text{-}CV_{sc})$
    $compute\_ERV(n, ECV_n)$
    if $no\_resource(ERV_n, Constraints, cycle\text{-}RV_{sc})$ then
        $defer(n)$
    else $compute\_priority(n)$

$node = get\_priority\_node(node\_list_{sc})$
if $operation(node)$ then
    $settle(node),\ add(ERV_{node}, cycle\text{-}RV_{sc})$
    $distribute(data\_flow\_succ(node))$
else if $fork(node)$ then
    $record\_cycle\_CV(CV(node))$
```

The *ECV* and *ERV* of each node are computed first. The *ERV* is then compared against the constraints, *Constraints* and the committed resources are recorded in $cycle\text{-}RV_{sc}$. If not enough resources are available, the node will be deferred to later cycles. Otherwise, the priority is computed using the static information from the resource priority and the path delay functions. A node with a *null ERV* implies perfect conditional resource sharing and will be given a very high priority. The highest priority node is selected and scheduled. For multi-cycling, the node will be settled into several consecutive cycles. In the case of operation chaining, *distribute* adds the data dependent successors of *node* to the *node_lists* of the current cycle. Otherwise, they are added to some later cycles.

Listing 6.2

```
RV    PCV
[ 1 ] [ 10 ] 1. b := x + y
[ 0 ] [ 11 ] 2. case c
[ 1 ] [ 10 ] 3. < 1 > ⇒ p := x + b
[ 1 ] [ 01 ] 4. < 0 > ⇒ p := y + y
```

Listing 6.3

```
              RV,ERV,PCV,ECV                          RV,ERV,PCV,ECV
              [ 1 ][ 1 ][ 10 ][ - - ]  1. b := x + y
2. case c                          < 1 >                               < 0 >
              [ 1 ][ 1 ][ 10 ][ 10 ]  3. p := x + b  [ 1 ][ 0 ][ 01 ][ 01 ]  4. p := y + y
```

For the description in listing 6.2, under a normal scheduling situation, the conditional expression, *c* in $node_2$ will not be evaluated before $node_1$. Therefore, the condition needed to determine the branching will not be valid until the cycle after $node_2$ is scheduled.

$$cycle\text{-}CV_{c1} = cycle\text{-}CV_{c2} = [\ -\ -\ ],$$
$$cycle\text{-}CV_{c3} = [\ 11\ ].$$

As a result, even though the *PCV* of $node_1$ is [10], its *ECV* will be empty, [- -]. The *ECV* s of $node_3$ and $node_4$ are equal to their *PCV* s. And assuming that $node_3$ is scheduled before $node_4$, since the operation in $node_4$ can share the same resource as $node_3$, the *ERV* of $node_4$ is [0]. This *null ERV* will give $node_4$ a very high priority to be scheduled together with $node_3$. On the other hand, if the conditional expression is evaluated at the very beginning, listing 6.4,

Listing 6.4

```
              RV,ERV,PCV,ECV                          RV,ERV,PCV,ECV
2. case c                          < 1 >                               < 0 >
              [ 1 ][ 1 ][ 10 ][ 10 ]  1. b := x + y  [ 1 ][ 0 ][ 01 ][ 01 ]  4. p := y + y
              [ 1 ][ 1 ][ 10 ][ 10 ]  3. p := x + b
```

all the conditions will be available afterwards with *cycle-CV* s equal to [11]. This mutually exclusive condition can be used for resource sharing. In this case, they are the additions of $node_1$ with ECV [10] and $node_4$ with ECV [01]. The result is reflected in their ERV s.

As already mentioned, by introducing temporary variables, statements can be decomposed into computation and assignment parts. With speculative execution, the computation parts can be migrated freely outside their basic block with changes in their ECV s. In listing 6.4, $node_3$ and $node_4$ can be split into computation and assignment parts, { 3a, 3b }, { 4a, 4b }. If two adders are available, statements 3a and 4a can migrate across $node_2$ and yield listing 6.5.

Listing 6.5

```
            RV,ERV,PCV,ECV                              RV,ERV,PCV,ECV
            [ 1 ][ 1 ][ 10 ][ - - ]   1. b := x + y
            [ 1 ][ 1 ][ 10 ][ - - ]   3a. s := x + b   [ 1 ][ 1 ][ 01 ][ - - ]   4a. t := y + y
  2. case c                                < 1 >                                      < 0 >
            [ 0 ][ 0 ][ 10 ][ 10 ]    3b.   p := s     [ 0 ][ 0 ][ 01 ][ 01 ]    4b.   p := t
```

This should only be carried out if data dependencies are preserved and abundant resources are available. By using this speculative scheduling technique, the computation can be speeded up globally. Significant improvement in resource utility can be achieved and conditional operations within the basic block could be reduced and balanced.

As ECV s and ERV s are updated as operations are moved across the basic block boundaries, they record the execution condition and the resources requirement respectively. With them, the condition and the resource requirement of an operation can be represented dynamically. Scheduling of operations thus becomes more flexible and the use of migration allows operations to be distributed more evenly.

6.12 An Example

To illustrate the benefit of code migration, we compare R^2Sch with a conventional scheduling style i.e. where the mobility of an operation is confined to its basic block. Consider the example named *Gœȷ* in listing 6.6. It consists of two disjoint conditional statements.

Listing 6.6

```
a := A; b := B; d := D; p := P; q := Q;

if (p > q) then  c := p + q;
           else  c := p * q;
end if;
case c is
when 1 => y := ((a + b) * (c + d));
when 2 => y := ((a * b) + (c * d));
when 3 => y := ((a + c) * (b + d));
when 4 => y := ((a * c) + (b * d));
end case;
Y := y;
```

The compiled code is shown in listing 6.7. The statements have been broken down into computation parts and assignment parts. The first vector is the *CV* and the second one is the propagated condition vector, *PCV* .

Definition 6.7

```
 1.  [______] [__1111] [______] [______]   a :=  pgt A;
 2.  [______] [__1111] [______] [______]   b :=  pgt B;
 3.  [______] [__1111] [______] [______]   d :=  pgt D;
 4.  [______] [111111] [______] [______]   p :=  pgt P;
 5.  [______] [111111] [______] [______]   q :=  pgt Q;
 6.  [______] [111111] [______] [______]   fg1 := p gt q;
 7.  [______] [111111] [______] [______]   fork( fg1 )
 8.  [10____] [101111] [10____] [______]     it1 := p + q;
 9.  [10____] [101111] [10____] [10____]     c :=  it1;
10.  [01____] [011111] [01____] [______]     it2 := p * q;
11.  [01____] [011111] [01____] [01____]     c :=  it2;
12.  [______] [111111] [11____] [11____]   fork( c )
13.  [__1000] [__1000] [__1000] [______]     it3 := a + b;
14.  [__1000] [111000] [111000] [11____]     it4 := c + d;
15.  [__1000] [111000] [111000] [11____]     it5 := it3 * it4;
16.  [__1000] [111000] [__0100] [111000]     y :=  it5;
17.  [__0100] [__0100] [__0100] [______]     it6 := a * b;
18.  [__0100] [110100] [110100] [11____]     it7 := c * d;
19.  [__0100] [110100] [110100] [110100]     it8 := it6 + it7;
```

```
20.  [__0100] [110100] [110100] [110100]     y :=  it8;
21.  [__0010] [110010] [110010] [11____]     it9 := a + c;
22.  [__0010] [__0010] [__0010] [______]     it10 := b + d;
23.  [__0010] [110010] [110010] [11____]     it11 := it9 * it10;
24.  [__0010] [110010] [110010] [110010]     y :=  it11;
25.  [__0001] [110001] [110001] [11____]     it12 := a * c;
26.  [__0001] [__0001] [__0001] [______]     it13 := b * d;
27.  [__0001] [110001] [110001] [110001]     it14 := it12 + it13;
28.  [__0001] [110001] [110001] [110001]     y :=  it14;
29.  [______] [111111] [111111] [111111]   Y :=  ppt y;
        CV      PCV     C-ECV    G-ECV
```

Each line corresponds to a node(operation) in the description's *PFG* and *CDFG*. The *PFG* of Gœȷ is shown in figure 6.13. The horizontal axis corresponds to the nodes in the decomposed description while the vertical axis represents the levels. The flow of execution is from top-left to bottom-right. The branch structures and the potential parallelism within basic blocks are shown clearly in the *PFG*. The *CDFG* of Gœȷ is in figure 6.14. The computation nodes are allowed to migrate freely outside their basic blocks to their as-soon-as-possible(ASAP) positions. As a result, the graph gives a brief outlook on the potential global parallelism. Points to note with this graph:

- $\{ node_8, node_{10}, node_{13}, node_{17}, node_{22}, node_{26} \}$ which are the computation nodes, are all outside their basic blocks.
- $\{ node_9, node_{11} \}$ which are assignment nodes, are guarded by the fork node, $node_7$. The same also applies to $\{ node_{16}, node_{20}, node_{24}, node_{28} \}$ which are guarded by $node_{12}$.
- Compared with the *PFG* which has 10 levels, the *CDFG* has 8 levels only.

Resource un-restricted scheduling of both styles is performed. The actual schedules are in figure 6.15 and 6.16; where we assume the delay of an adder is, $\{ (40 + 0c + 0) : - : - \}$ sightly less than one cycle and the delay of a multiplier is $\{ (50 + 0c + 50) : - : - \}$ a bit less than two cycles. There is delay associated with the controller as well, $\{ (5 + 1c + 5) : 1 \ cycle : 30ns \}$ with one cycle latency. This signifies that branching occurs one cycle after the branch condition is valid. From these figures, the effect of computational migration, state overheads and stack times can be observed very clearly.

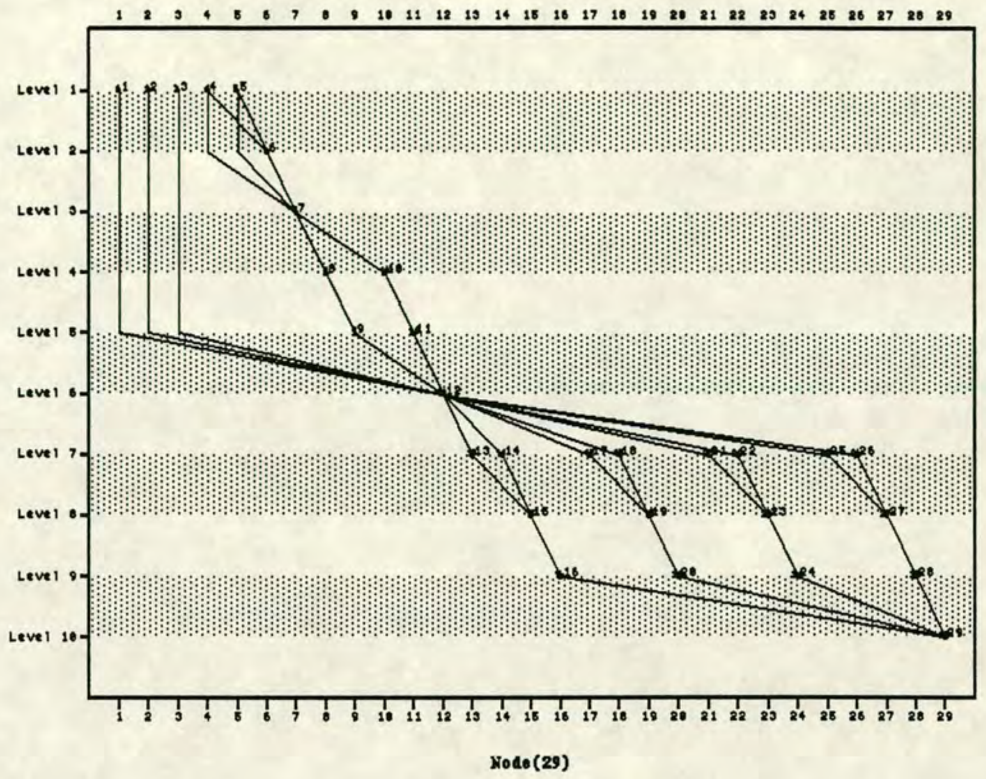


Figure 6.13: The Program Flow Graph of $G\text{œ}\jmath$.

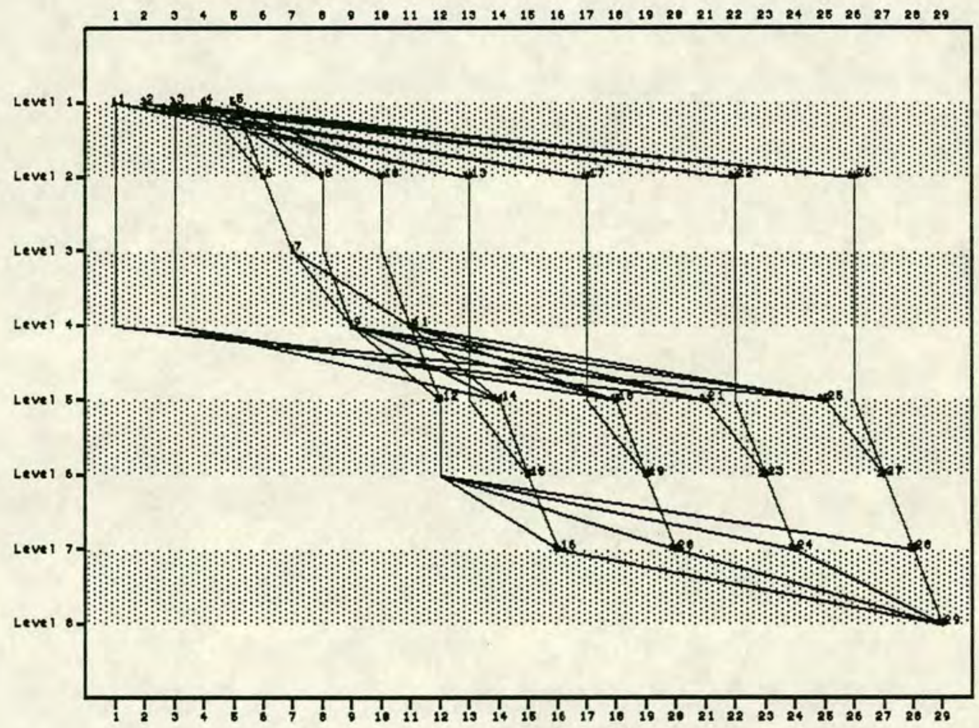


Figure 6.14: The Control-Data Flow Graph of $G\text{œ}\jmath$.

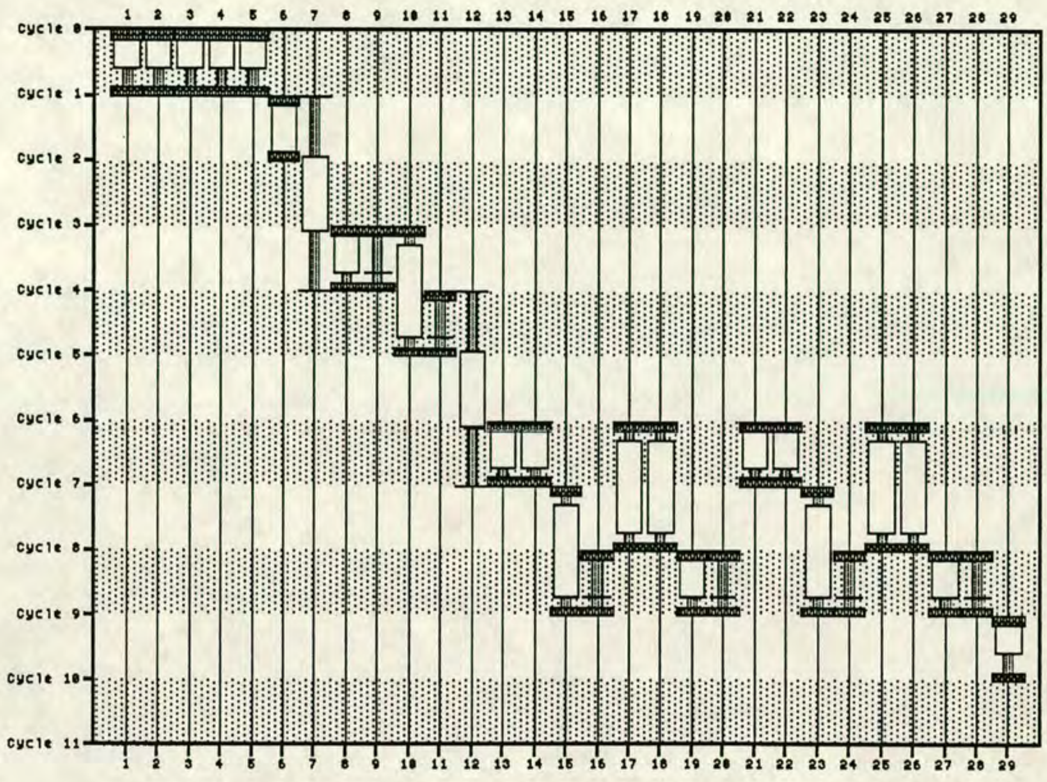


Figure 6.15: Schedule Graph Generated with Conventional Scheduling Style.

The C-ECVs in listing 6.7 are the execution condition vectors, ECVs, of the conventional scheduling style. They are more or less the same as the original CVs. At the beginning of the schedule, since no condition is available, the C-ECVs are empty. And as all the nodes are staying within their basic blocks, no change in C-ECVs is needed. The G-ECVs in the last column are the ECVs for R^2Sch. Compared with C-ECVs, many of the entries in G-ECVs are empty. This reflects the fact that due to unrestricted resources, a considerable number of computation nodes migrate outside their basic blocks. The result is 10 cycles for the conventional style and 7 cycles for R^2Sch. Since resources are not restricted and operations are allowed to move outside their basic blocks, R^2Sch requires significantly more resources.

The conventional approach requires less resources because, as all operations are confined within their conditional branches, resource sharing by mutually exclusive conditions can be utilized. Starting from the resource requirement of the maximum

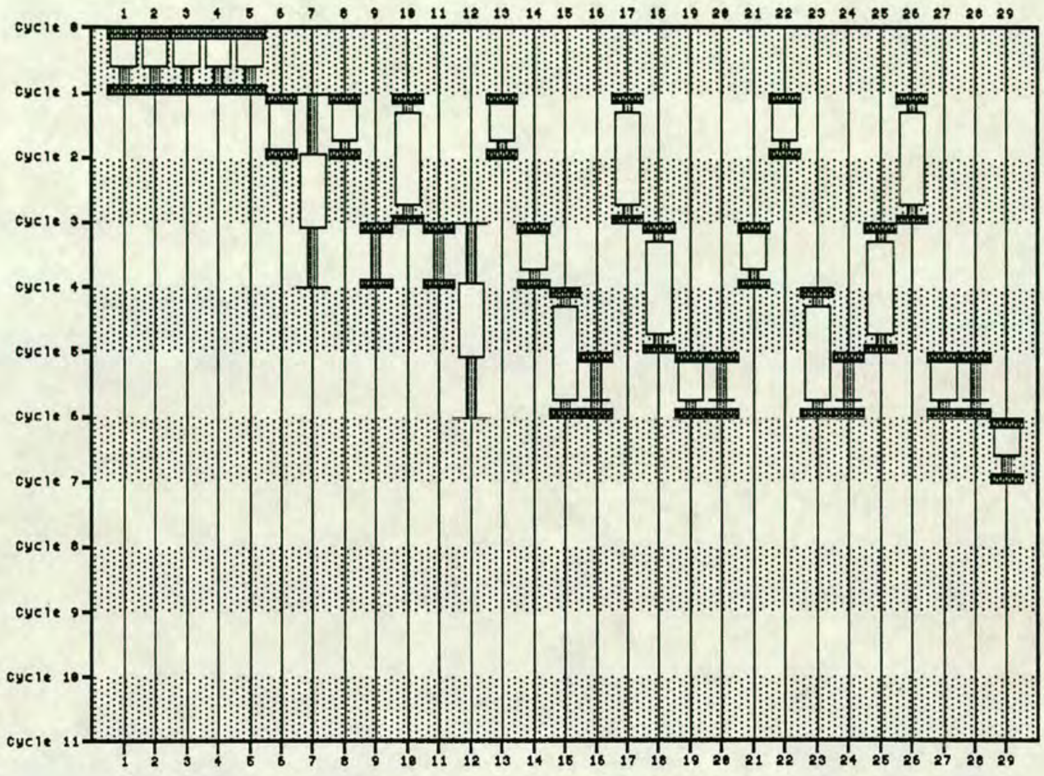


Figure 6.16: Schedule Graph Generated with R^2Sch .

Scheduling Style	Input Constraints	Result	
		Delay(cycles)	Resource Requirement
Conventional	(none)	10	2 Add, 2 Mul, 1 CMP
	1 Add	11	1 Add, 2 Mul, 1 CMP
	1 Mul	12	2 Add, 1 Mul, 1 CMP
	1 Add, 1 Mul	12	1 Add, 1 Mul, 1 CMP
R^2Sch	(none)	7	3 Add, 4 Mul, 1 CMP
	3 Mul	8	3 Add, 3 Mul, 1 CMP
	2 Add, 3 Mul	8	2 Add, 3 Mul, 1 CMP
	2 Add, 2 Mul	9	2 Add, 2 Mul, 1 CMP
	1 Add, 2 Mul	9	1 Add, 2 Mul, 1 CMP
	2 Add, 1 Mul	11	2 Add, 1 Mul, 1 CMP
	1 Add, 1 Mul	11	1 Add, 1 Mul, 1 CMP

Table 6.1: Scheduling with Various Resource Restrictions.

parallel schedule, the number of resources are gradually restricted. The results are summarized in table 6.1. For various resource restrictions, the global scheduling, R^2Sch , still out-performs the conventional one. The performance advantage of R^2Sch is mainly due to the concurrent evaluation of condition expressions and operations.

6.13 Results

To study the efficiency of the algorithm, several widely used examples have been tested.They include an example from [PARK86], the heavily used fifth-order digital elliptic filter, and part of the MC6502 description. The descriptions used are all listed in the appendix.

a. The Example from [PARK86]

	Delay(cycles)		
Resource Requirement	Forward	Backward	Minimum
2 Add, 1 Sub	4	4	4
1 Add, 1 Sub	5	5	5

Table 6.2: Scheduling Result of the Example from [PARK86].

The example consists of 16 addition or subtraction operations and 20 edges. It is chosen first because it contains a substantial number of nested and disjoint condition branches. This helps to establish the effectiveness of R^2Sch in sharing resources under mutually exclusive conditions. The scheduling results are in table 6.2. As in other published schemes, the cycle length is assumed to be long enough to accommodate two operations. R^2Sch is able to achieve the fastest schedule of 4 cycles with { 2 Add, 1 Sub } which is known to be the optimum resource-delay combination i.e. the resources cannot be reduced further without an increase in number of cycles, and vice versa. Because of the small number of resources and cycles involved, it is not surprising that the cycle delay obtained from conventional scheduling and R^2Sch are the same.

b. Fifth-Order Elliptic Filter

Resource Requirement	Delay(cycles) Forward	Backward	Minimum	FDS	FDLS	PBS
3 Add, 3 Mult	17	18	17	17	17	17
3 Add, 2 Mult	18	18	18	18	–	18
2 Add, 2 Mult	19	18	18	19	18	18
2 Add, 1 Mult	21	22	21	21	21	21
1 Add, 1 Mult	28	29	28	–	–	–
3 Add, 2 Pipe	17	18	17	17	17	17
3 Add, 1 Pipe	18	19	18	18	18	18
2 Add, 1 Pipe	19	19	19	19	19	19
1 Add, 1 Pipe	28	28	28	–	–	–

Table 6.3: Scheduling Result of the Fifth-Order Elliptic Filter.

This example is taken from the 1988 High-Level Synthesis Workshop Benchmarks. The filter has 43 operations and 60 edges. 8 of the operations are multiplications and the rest are additions. Scheduling with various resource restrictions was performed. The results are summarized in table 6.3. The results of Force Directed Scheduling(FDS) [PAUL87], Force Directed List Scheduling(FDLS) [PAUL89] and Percolation Based Scheduling(PBS) [POTA90] are summarized for comparison. As in other papers, the functional unit types used are adders with 1 cycle delay, and multipliers with 2 cycles delay. Each schedule takes less than 1 second to compute on a SparcStation 1. For the schedule with 17 cycles, the original design from [KUNG85] requires 4 adders and 4 multipliers. R^2Sch also achieves the schedule with 18 cycles using 2 multipliers and 2 adders. This schedule is not obtained by many other algorithms, including force-directed scheduling, but the later force-directed list scheduling [PAUL89] which takes better consideration of resource restrictions amends the situation.

The second part of the result makes use of a two stage pipelined multiplier. Since data can be input every cycle, better results are obtained. As a whole, R^2Sch is able to equal the best results published hitherto. These are believed to be the best possible results for this heavily studied example. Although R^2Sch performs very well on this example, it is not a good yardstick. The description is a straight line segment of code consisting solely of arithmetic computations with no conditional statements. Nevertheless, it confirms that the underlying algorithm of R^2Sch is sound enough for basic block scheduling.

c. MC6502 Group 1

Conventional	
Resource Requirement	Delay(cycles)
2 ALU	11
1 ALU	12

R^2Sch	
Resource Requirement	Delay(cycles)
2 ALU	9
1 ALU	10

Table 6.4: Scheduling Result of MC6502 Group 1

From the original ISPS description, we have synthesized the group 1 instructions.

group 1 instruction decode
group 1 address generation, and
group 1 instruction execution.

Most subroutines in the group are expanded. The preserved ones are *read*, *write*, *addr*, *setnz*, *adjust*. They were treated as implemented functional units and handled as external procedure calls. During scheduling, it was assumed that the delay information of these procedures was known. The final input description consists of two parts:

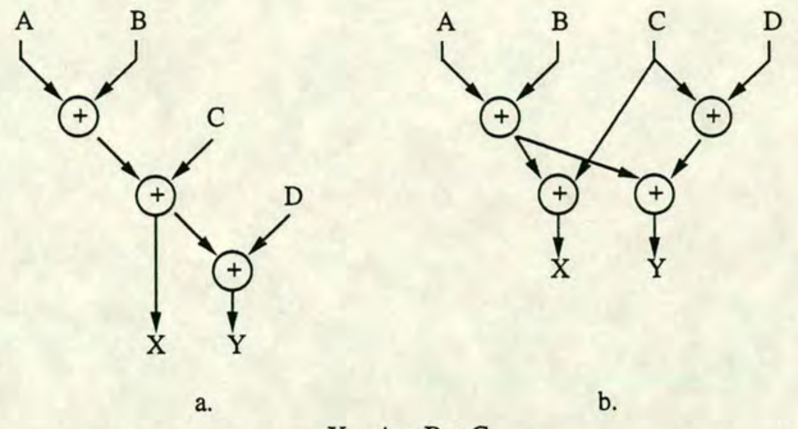
address generation and instruction execution

in two select statements. Each of these consists of 8 branches. As a microprocessor design, it is reasonable to assume that all operations are performed on ALUs. Both conventional style and R^2Sch scheduling were carried out. Although, a large number of operations are involved, due to the mutually exclusive conditions, very few functional units are needed. From the results presented in table 6.4, R^2Sch has out-performed the conventional style by 16% to 18%. After studying the schedule graph, it is clear that the performance gain comes from the migration of computations . This example also demonstrated the benefit of using branch-based CVs instead of path-based ones which would have generated 64 paths.

6.14 Observations

6.14.1 Common Sub-Expression Elimination

Common subexpression elimination is one of the most widely used techniques in software compilers. It has a net gain in reducing the amount of computations. Its role is vital when there is only one execution unit. However, specially when there



X = A + B + C
Y = A + B + C + D

Figure 6.17: Different Degree of Common Sub-Expression Elimination.

are enough or redundant execution units, this technique should not be applied directly. Consider the description in figure 6.17. A typical language compiler will jump to the conclusion that it should be grouped as (a). However, when two execution units are available, one of them will remain idle. A partial sub-expression elimination arrangement like (b) will be able to utilize both execution units and achieve more speed advantage. A word of caution, the sequence in figure 6.17a can be easily justified if the idle execution unit can be utilized to speed up execution in other paths.

The above observation has been applied to the fifth-order elliptic filter example. Two and three level flattening and regrouping are tried. The results are summarized in table 6.5. In the column labeled "original" are the minimum schedule results from table 6.3. Examining the results, they show that when there are extra resources (> 2 adders), it is possible to have a speed-up of one cycle. When only minimum resources are available, 1 adder and 1 multipler, there is no speed up in the case of 2 level regrouping. In the case of 3 level regrouping, a negative effect is apparent due to the extra computations introduced by the flattening and regrouping process.

The observation above shows that in high-level synthesis, besides the interdependence between scheduling and allocation, their interdependence with high-level

	Delay(cycles)		
Resource Requirement	Original	a. 2-Level	b. 3-Level
3 Add, 3 Mult	17	16	16
2 Add, 2 Mult	18	17	17
2 Add, 1 Mult	21	20	20
1 Add, 1 Mult	28	28	29
3 Add, 2 Pipe	17	16	16
3 Add, 1 Pipe	18	17	17
2 Add, 1 Pipe	19	18	18
1 Add, 1 Pipe	28	28	29

a. 2-level Flattening and Regrouping
b. 3-level Flattening and Regrouping

Table 6.5: Different Degrees of Common Sub-Expression Elmination.

transformations should not be underestimated. To tackle this problem, it seems there is a need for a dynamic, resource constraint oriented expression decomposition and regrouping technique.

6.15 Potential

6.15.1 Dynamic Loop Unrolling

Consider the following loop in listing 6.8, since it is dynamic, full-unroll is impossible. The loop boundary also imposes a constraint as the inductive variables cannot be moved outside the loop.

Listing 6.8

```
while (n > b) loop
      a := a + b + n;
      n := n - 1;
end loop

        (a)
```

```
while (n > b) loop
      a := a + b + n;
      n := n - 1;
   if (n > b) then
      a := a + b + n;
      n := n - 1;
   end if;
   if (n > b) then
      a := a + b + n;
      n := n - 1;
   end if;
end loop;

        (b)
```

Assuming all operations are performed on ALUs, a schedule of 3 cycles will need 2 ALUs. The first cycle is to evaluate the conditional expression, followed by computations. The result is a very poor schedule with one of the ALU under utilised. To improve the situation, the loop can be partially unrolled by inserting "if-then" statements to do the intermediate condition check. Listing 6.8b shows one which has been unrolled three times. Then, by variable renaming and aggressive scheduling, R^2Sch can put the partially unrolled loop into 6 cycles with the same resources. There is a speed up of one-third. Usually, in the presence of loops, pipelining can also be applied to improve resource utilization. However, in this case, pipelining can only save one cycle for every two consecutive stages. This means that for an execution sequence like (b), 7 cycles will be required in a pipeline. Moreover, the condition check will also increase the difficulties encountered by pipelining.

6.16 Conclusion

In this chapter, we have presented a new priority function and scheduling methodology to handle dynamic code motion with respect to resource constraints. From the investigations conducted, it is observed that when operations are allowed to migrate to wherever resources are available, better scheduling results can be achieved. Also, from the observations, it is clear that when more than minimum absolute resources are available, high-level transformations will become interdependent with scheduling and allocation. This situation adds further complexities to the problem of high-level synthesis.

R^2Sch is nevertheless still far from meeting the ideals of high-level synthesis. The main reason is that it considers scheduling as an individual task, while tasks in high-level synthesis are highly interdependent. However, because of its good performance, it is being used for front-end coarse scheduling and back-end cycle length tuning and verification. At the front-end, it helps to evaluate the effect of allocation with different combinations of functional units. The flexible allocations and the scheduling results are passed onto the next stage where cycle and module binding are performed. At the back-end, it takes a finished design and refines the cycle length with detail control and interconnection delays.

Chapter 7

Integrated Concurrent Mapping

7.1 Introduction

As described in [KUCU90], the cost of interconnection could have a first-order effect on the area cost of the implememtation. Therefore, it is extremely risky if it is not considered seriously in the early phases of the design process. The problem is that optimal scheduling may require a very expensive interconnection scheme to realize. Recent research shows that more attention is now being paid to the cost of interconnection [CLOU90] [PAPA90] [HUAN90]. However, for some of the approaches, the cost of interconnection, instead of being considered directly, is estimated using cost functions. These cost functions are usually concerned with the interface compatibility among operations and functional units only.

7.2 Target Architectures: Related Research

For high-level synthesis, most of the target architectures reported in the literature can be classified into two types:

- Un-constrained Architecture,
- Constrained Architecture.

In the school of *general* synthesis approach, systems such as Facet [TSEN86], HAL [PAUL89] or Chippe [BREW90], have their target architectures composed of unconstrained and distributed data paths. Separated functional units and memories are connected in a random fashion. The interconnection scheme can be one of the following:

1. point-to-point connection with multiplexors,
2. point-to-point connection with buses, or
3. a mix of the two.

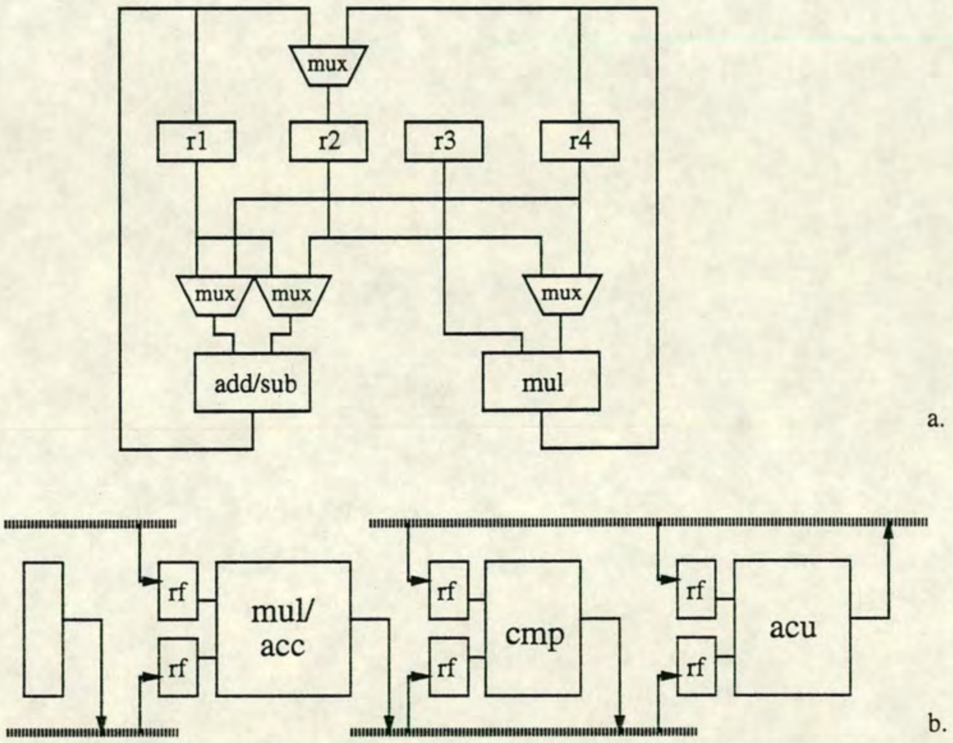


Figure 7.1: Un-constrained(a) and Constrained(b) Architecture.

In this approach, multiplexors are used heavily between elements. For instance, the connectivity binding algorithm, Splicer [PANG88], embedded in Chippe uses two level multiplexing to improve interconnection and the number of multiplexor inputs. As minimization of the interconnection is taken as a post-scheduling task, they assume no constraints on the availability of interconnections. If buses are used, they are usually derived from the original point-to-point connection structures. Heuristic algorithms are applied to group the random interconnections into buses. Since interconnection is not getting enough attention at the beginnning, unavoidably the complexity of it explodes at a later stage of the synthesis process.

At that stage, it is already too late and improvement is not at all simple. Consequently, the resultant architecture involves large numbers of interconnections and buses. This is very difficult to lay out and forms a two dimensional topology with relatively low density.

For the school of *specific* synthesis approach such as Sugar [THOM88], SPAID [HARO88] and Cathedral-II [DEMA90], there is an early commitment of the architecture to a multi-bus style. The resultant architecture is usually a constrained processor style. This avoids the problems caused by random interconnections. Since the knowledge of the final architecture is embedded in the algorithm, a draft floorplan can be constructed. The resultant layout is relatively compact, buses can be identified easily. It forms a one-dimensional topology.

Cathedral-II restricts the number of target functional units to some well-defined structures. Recently, to increase the architectural freedom, Cathedral-2rd was developed to support more flexibility in the composition of execution units [LANN90]. Basic functional building blocks are combined to form different execution units to meet the application specific requirements.

For memory management, in Cathedral-II, a register file is placed on every input of a functional unit. In SPAID [HARO89], a register file is associated with each bus. Comparing these two schemes, Cathedral-II has the potential of using more registers because identical data which are assigned to different functional unit inputs could not be merged, they reside in different register files. Therefore, in SAGE [GRAN90], each memory requirement of an operation is given a private memory element. After all the memory requirements have been attended to, they are grouped into memory blocks with respect to the type of functional unit from which they are generated.

7.3 Adjustable Target Architecture

To benefit from concurrent scheduling, an openly adjustable architecture is necessary and should cover both the constrained and unconstrained architecture men-

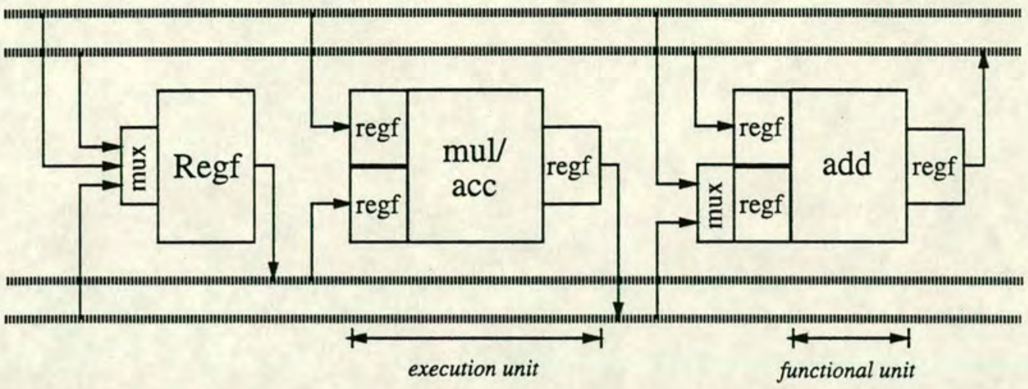


Figure 7.2: Openly Adjustable Target Architecture.

tioned before. The architecture we targeted is similar to other data path oriented structures. It is made up of execution units and global register files connected together by buses. Each *execution unit* consists of a *functional unit* and local input/output register files. Local feedback is possible. The size of the register files are not pre-determined. They are allocated and minimized during the scheduling process. Compared with the architecture in Cathedral-II, besides input register files, we have also output and global register files. These register files help to avoid the duplication of data and provide more flexibility for both storage and communication. We will demostrate that this architecture can reduce the amount of communication and hence the cost of interconnections.

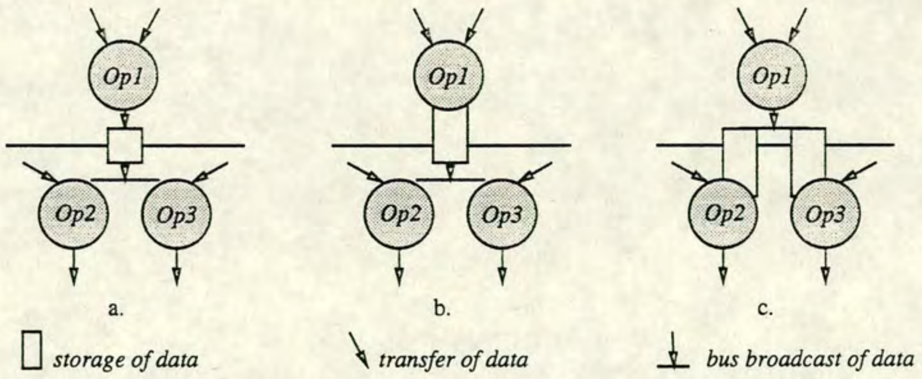


Figure 7.3: Different Storage of Data Value.

Consider the data flow graph schedule in figure 7.3, the data value across the cycle boundary, from *op*1 to *op*2 and *op*3, needs to be stored.

figure 7.3a: If the value is assigned to a global register file (GRF), three data transfers with two buses will be required; one for the output transfer of *op*1 and the other for the input transfers of *op*2 and *op*3.

figure 7.3b: If an output register file (ORF) is used, the output value of *op*1 can be stored locally and broadcast in the next cycle with only one bus.

figure 7.3c: If only input register files (IRF) are allowed, the output value of *op*1 has to be broadcast to the input register files of both *op*2 and *op*3. This results in a duplication of data.

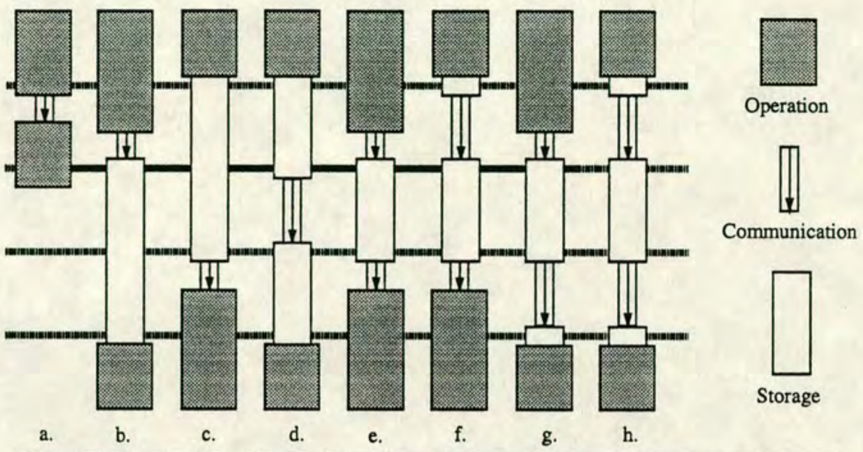
As a whole, output register files help to defer the data transfers to less crowded control steps, while input register files allow data to be received well before it is needed.

7.4 Storage and Communication Schemes

By inter-mixing local register files with global ones, the communication scheme can be summarized as in figure 7.4. There are 8 different possibilities.

Scheme (a : | ⟶ |)	direct connection,
Scheme (b : | ⟶ ▯)	with input register files only, as in figure 7.3c,
Scheme (c : ▯ ⟶ |)	with output register files only, as in figure 7.3b,
Scheme (d : ▯ ⟶ ▯)	with both input and output register files,
Scheme (e : | ⟶ ▯ ⟶ |)	with global register files only, as in figure 7.3a,
Scheme (f : ▯ ⟶ ▯ ⟶ |)	with global and output register files,
Scheme (g : | ⟶ ▯ ⟶ ▯)	with global and input register files,
Scheme (h : ▯ ⟶ ▯ ⟶ ▯)	with global and both input and output register files,

Scheme (a : | ⟶ |) happens only when operations are scheduled to be chained. The output from one operation is transferred directly to the input of the other without going through any storage. Since the two operations are linked back-to-front, the output of the first one must be kept stable long enough for the next one to finish. Usually, this only happens to fast operations scheduled with a long

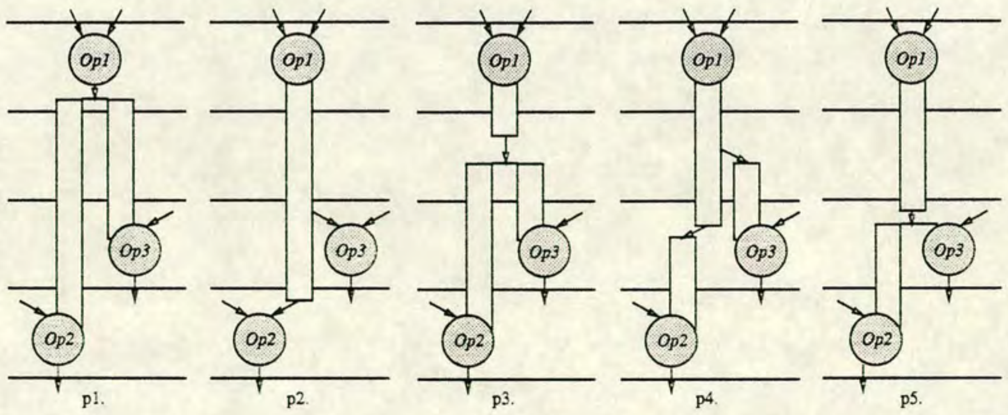


No.	Symbol	No. of Transfers	output buffer	global registers	input buffer
a.	|⟶|	1	-	-	-
b.	|⟶▯	1	-	-	√
c.	▯⟶|	1	√	-	-
d.	▯⟶▯	1	√	-	√
e.	|⟶▯⟶|	2	-	√	-
f.	▯⟶▯⟶|	2	√	√	-
g.	|⟶▯⟶▯	2	-	√	√
h.	▯⟶▯⟶▯	2	√	√	√

Figure 7.4: Storage and Communication Schemes.

cycle length. Scheme (b : |⟶▯) is similar to the target architecture used by Cathedral-II. Register files are stalled at the inputs of the functional units. It has been demonstrated that when the output value of an operation is required by several others, duplication of data is unavoidable. This is necessary as a compromise for reducing data communications. Scheme (e : |⟶▯⟶|) , with global register files only, is the unconstrained architecture used commonly by today's high-level synthesis systems. Since there is no local storage, every operation has to draw its input values from the global register files and has to write the result back immediately afterwards. As communications have to be accomplished immediately before and after the operations, the schedule of the operations will more or less define the communication pattern along the time axis. Therefore, when this scheme is used, it is important to consider the communication pattern during the scheduling process. For a functional unit which has an output register, it will belong to Scheme (f : ▯ ⟶ ▯ ⟶ |) . With local feedback, the single output register is usually used as an accumulator. For some applications, this will significantly reduce the amount of global bus traffic. In addition, for testability, the output register of all the functional units can be linked together to form a scan path/chain. The functionality of each functional unit can then be tested easily. Scheme (h : ▯ ⟶ ▯ ⟶ ▯) is the most powerful. With the support of both global and input/output register files, bus transfers can be scheduled anywhere in the time frame bound by the define and use operations.

Scheme (b), (c) and (d) which consist of only one transfer are the more efficient ones. Global register files are not involved. Input and output register files are utilized to cover the full life time of the intermediate values. For scheme (e), (f), (g) and (h), because of the limited number or even non-existence of local register files, global ones have to be introduced. Unavoidably, they lead to an increase of data transfer.



Part	Scheme	No. of Transfers	No. of Register Cycles
p1.	b	1	5
p2.	c	2	3
p3.	d	1	4
p4.	d	2	4
p5.	c, d	1	3

Figure 7.5: The Usage of Input and Output Register Files.

To demonstrate how input and output register files are used, let us look at a typical case depicted in figure 7.5. There are two data transfers to two operations, *Op*2 and *Op*3, scheduled in two different cycles. In the table, the cost of communication is assessed by the number of transfers and the cost of storage is assessed by the number of register cycles, i.e. the number of cycles that an intermediate value needs to be stored. **P1** and **p2** show the storage and communication schedule with only input and output register files respectively. **P3** and **p4** are two intermediate schedules of Scheme (d : ▯ ⟶ ▯) . As demonstrated in the cost table, they are not optimal. *P*5 which is a mix of both Scheme (c : ▯ ⟶ |) and Scheme (d : ▯ ⟶ ▯) shows the optimal solution. In **p5**, the output register file from *Op*1 is used to store the value for the two later operations. Output transfers to both *Op*2 and *Op*3 occur in one bus broadcast. However, the value is only consumed immediately by *Op*3. The value for *Op*2 is stored in its input buffer. This buffer ensures that there is no duplication of data and an optimal number of buses is used. It is important to stress that **p5** is only a local optimal solution. When there is a graph of operations, a global consideration is essential to determine an optimal schedule for storage and communication.

7.5 Architectural Constraints

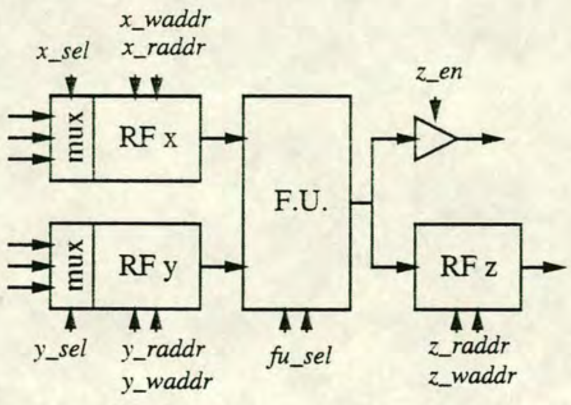


Figure 7.6: The Architecture of an Execution Unit.

The user can influence the target architecture through the input of architectural constraints. These constraints cover both the constrained and the unconstrained architectures. Using these constraints, the designer can tailor the design to a large variety of architectures. Different from the performance constraints such as delay and area, these are classified as *high-level constraints.* These high-level constraints enable inferior or unsuitable designs to be filtered out as soon as possible. The architectural constraints can include restrictions on

- the number of each type of functional unit.
- the number of registers in the input and output register files.
- the number of global register files and the number of registers in them..

If the number of registers at the input and the output register files are set to zero and the size of global register files are restricted to one, then the target architecture is exactly the unconstrained style. In summary:

$N_{reg}(IRF) = 0$ ORFs and GRFs only SAGE style
$N_{reg}(ORF) = 0$ IRFs and GRFs only Cathedral style
$N_{reg}(ORF) = N_{reg}(IRF) = 0$ GRFs only execution unit = functional unit
$N_{reg}(ORF) = N_{reg}(IRF) = 0$ & $N_{reg}(GRF) = 1$ unconstrained architecture

7.6 The Lower Bounds

In the previous section, we have shown that input and output register files can help to reduce the interconnection and storage overhead. In this section, we are trying to establish a lower bound requirement for them. To allow this to be expressed more clearly, the following notation will be adopted.

Object	Whole Set	Sub Set	Element
Control Steps	CS	Cs_i	cs
Functional Units	FU	Fu_i	fu
Operations	OP	Op_i	op
Variables	V	V_i	v
Registers	R	R_i	r
Register Files	RF	Rf_i	rf
Transfers	TF	Tf_i	tf
Buses	BS	Bs_i	bs

The assumptions we make are:

- Every operation has two inputs and one output only.
- All data transfers are performed by buses.
- Data must be stored across cycle boundaries.
- Bus transfer cannot occur across cycle boundaries.
- Input data is stable for one cycle only.
- Multiplexors can be introduced wherever they are needed.

7.6.1 Effect on communications

For each target architecture where only global, output and input register files are allowed respectively, we try to find the minimum requirement on the number of buses.

1. Architecture with global register files, GRFs , only

   For a target architecture which uses global register files only, Scheme $(e : \mid \longrightarrow \square \longrightarrow \mid)$, every operation will require three transfers. That is two input transfers and one output transfer between the execution unit and the global register files. Assuming that all the operations can be distributed evenly along the control steps, CS, the average number of operations per cycle is

   $$N_{AVE}(OP/CS) = \left\lceil \frac{N(OP)}{N(CS)} \right\rceil$$

   and hence the lower bound requirement on buses will be

   $$N_{MIN}(BS_{GRF}) = 3 * N_{AVE}(OP/CS) \tag{7.1}$$

   We cannot assume that the data transfers are distributed evenly. This would be wrong because data has to be provided and stored immediately before and after the operations.

2. Architecture with output register files, ORFs , only

   When output register files are used, Scheme $(c : \square \longrightarrow \mid)$, data generated by operations can be stored locally until they are needed figure 7.5(p2). Since data transfer occurs only at the inputs, there will be two transfers per operation. Again, assuming that all the operations are distributed evenly along the control steps, CS, the lower bound requirement on buses will be

   $$N_{MIN}(BS_{ORF}) = 2 * N_{AVE}(OP/CS) \tag{7.2}$$

3. Architecture with input register files, IRFs , only

   When input register files are used, Scheme (b : $| \longrightarrow []$) , data transfer will occur immediately after the result is generated. If the result is needed by more than one operation, a bus broadcast will occur, figure 7.5(p1). As there is only one transfer per operation, by the same argument as before, the minimum requirement on buses will be

$$N_{MIN}(BS_{IRF}) = N_{AVE}(OP/CS) \tag{7.3}$$

7.6.2 Effect on Storage

For a given operation schedule, no matter which storage and communication scheme is used, the minimum number of cycles where intermediate values need to be stored is fixed. This is because the lifetimes of the intermediate values are governed by the define-use relationships of the operations. The register cycle of an intermediate value, v, is defined as

$$rc(v) = MAX(cycle\_use(v)) - cycle\_def(v)$$

and the total minimum number of register cycles, $RC_{min}(V)$, is

$$RC(V) = \sum_{v \in V} rc(v) \tag{7.4}$$

Since output register files, ORFs act like global register files, GRFs , which store data until they are needed, the number of register cycles is

$$RC_{ORF}(V) = RC_{GRF}(V) = RC(V) \tag{7.5}$$

For input register files, IRFs , storage is at the usage end. The register cycle of an intermediate value, v, is defined as

$$rc_{IRF}(v) = \sum_{cs \in cycle\_use(v)} cs - cycle\_def(v)$$

This is because when v is used by several operations, its value will be stored in several IRFs . This data duplication causes the total number of register cycles, $RC_{IRF}(V)$

$$RC_{IRF}(V) = \sum_{v \in V} rc_{IRF}(v) \tag{7.6}$$

to be greater than $RC(V)$ under most circumstances. However, if all the intermediate values are used only once, i.e. $N(cycle\_use(v)) = 1$, then they will all be the same.

$$RC_{IRF}(V) = RC_{ORF}(V) = RC_{GRF}(V) = RC_{min}(V)$$

On the other hand, when register files with only one read port are used, the minimum number of register files required will be

$$\begin{aligned} N_{MIN}(RF_{GRF}) &= 2 * N_{AVE}(OP/CS) \\ N_{MIN}(RF_{ORF}) &= 2 * N_{AVE}(OP/CS) \\ N_{MIN}(RF_{IRF}) &= 2 * N_{AVE}(OP/CS) \end{aligned} \tag{7.7}$$

They are all the same. This is because they are equal to the average data request in one control step. Notice that there is contradiction in the cases of architecture containing ORFs only. Since the minimum number of functional units for a design is $N_{AVE}(OP/CS)$, the minimum number of ORFs will be equal to the minimum number of functional units. That is $N_{AVE}(OP/CS)$ not $2 * N_{AVE}(OP/CS)$ as formulated above. This implies that some ORFs will need more than one read port unless more than one ORFs per functional unit are used.

7.7 Operations, Storage, and Communications

In order to concurrently map operations, storage and communications, a powerful representation of the design is essential. Traditionally, a directed graph similar to a control/data flow graph is used. It shows only the control/data dependency of the operations and carries little association with the storage and communication. In order to enable storage and communications to be considered early in the synthesis process, a direct representation of them is necessary. Figure 7.7 depicts the interrelations between operations, storage and communications. An operation will involve elements from all three domains. Any constraint or process applied in one domain will affect the other two directly. It is this interrelation which draws us to introduce the *scheduling and binding unit* (SBU). It is a triplet, { *oper rmem link* }, containing information for operation, storage and communication. The SBU appears as an elementary subgraph of a node. It is in tree form with only one level. Figure 7.8 shows an SBU with three data flow arcs.
The data structure of the SBU can be summarized as:

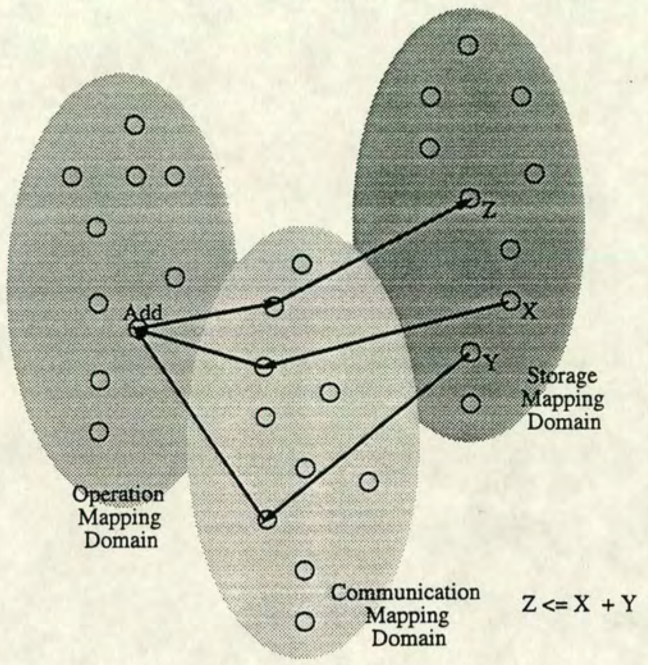


Figure 7.7: The Inter-relation of Operation, Storage and Communication Domain.

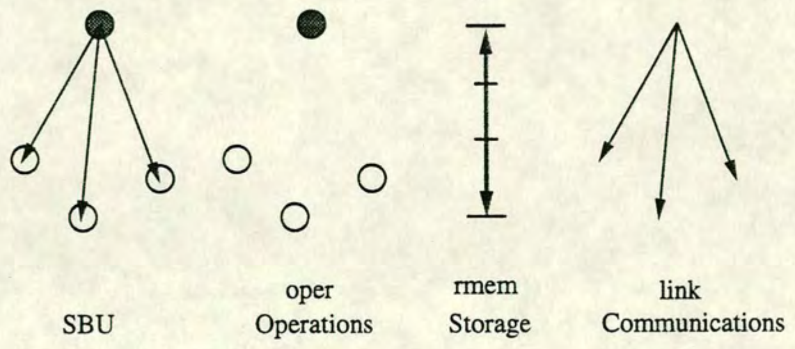


Figure 7.8: The Schedule and Binding Unit (SBU) and its elements.

Control :	*prec*	set of control/data dependency arcs
	succ	set of control/data dependency arcs
Operation :	*oper*	
	initiate	data generation operation
	asap	(cs, cs) : (*basap*, *easap*)
	alap	(cs, cs) : (*balap*, *ealap*)
	target	{ SBU } : set of data reception operation
	flex	{ *instance* } : flexibility, set of instances
	unit	*instance* : allocation information
Storage :	*rmem*	
	brmem	*cs* : output from operation
	srmem	*cs* : output from functional unit
	grmen	*cs* : output from global register file
	ermen	*cs* : end of data life
	flex	{ *instance* } : flexibility, set of instances
	unit	*instance* : allocation information
Communication :	*link*	
	blink	*port* : functional unit output port
	elink	{ *port* } : set of target input ports
	flex	{ *instance* } : flexibility, set of instances
	unit	*instance* : allocation information

Control *prec* and *succ* are sets of control and data dependency arcs which are pointing to the precedent and the successive SBUs. Operation (*oper* : *initiate*) is like the root of the SBU and is a major structure by itself. It contains all the necessary characteristics of an operation. (*Oper* : *target*) is the set of operations which are data dependent on (*oper* : *initiate*). For *rmem*, there are four control step variables representing the storage and communications scheme described in figure 7.4. When in use

- *brmem* to *srmem* defines the register cycles of the output register file.
- *srmem* to *grmem* defines the register cycles of the global register file.
- *grmem* to *ermem* defines the *maximum* register cycles of the input register files.

In cases where only global, output and input register files are used, we have the following properties:

global register files only	:	$brmem = srmem < grmem = ermem$
output register files only	:	$brmem < srmem = grmem = ermem$
input register files only	:	$brmem = srmem = grmem < ermem$

(*Link* : *blink*) is the output port of the execution units. If an output register file exists, it will be the output terminal of the register file. Hence, the control step of (*link* : *blink*) will always be equal to (*rmem* : *srmem*). (*Link* : *elink*) is a set of input ports of the operations in (*oper* : *target*). The number of control steps from (*rmem* : *grmem*) to the control step of individual (*link* : *elink*) elements will define the register cycles of the input register files.

7.8 Concurrent Scheduling

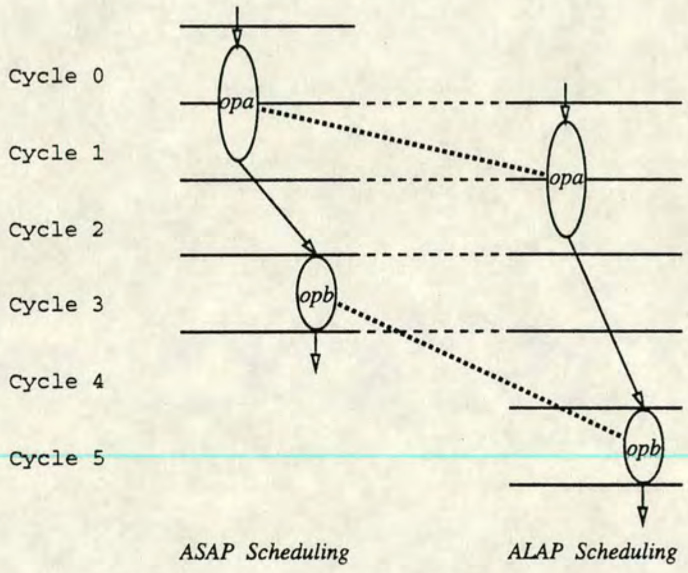


Figure 7.9: ASAP and ALAP Scheduling.

Concurrent scheduling assigns control steps to <u>d</u>ata operations, assigns control step to <u>d</u>ata transfers and arranges storage for <u>d</u>ata values, D^3Sch . With the rich storage and communication schemes detailed before, data transfer from one execution unit to another is very flexible. Added to that there are also three choices of storage: GRFs , ORFs and IRFs . The goal of concurrent scheduling, D^3Sch , is to minimize the density of operation, storage, and communication along the time axis and thereby reduce the overall cost of the design. Unlike most other systems whose aim in scheduling is to minimize the number of functional units, D^3Sch can be driven to minimize the number of transfers, the amount of storage or the total cost of resource with respect to individual constraints.

D^3Sch builds a distribution graph (DG)s of operations, storage and communications, then it tries to balance the distribution of them concurrently. In the next few sections, we will explain how this is performed.

7.8.1 Distribution Graph

In figure 7.9, SBU_a and SBU_b have the relation:

$$\begin{array}{lcl} SBU_b & \in & (\ Control : succ(oper_a)\ ) \\ SBU_a & \in & (\ Control : prec(oper_b)\ ) \end{array}$$

After ASAP and ALAP scheduling, for SBU_a, we have

$$\begin{array}{lcl} asap(oper_a) & = & (\ basap(oper_a),\ easap(oper_a)\ ) \\ alap(oper_a) & = & (\ balap(oper_a),\ ealap(oper_a)\ ) \end{array}$$

$$\begin{array}{lcl} easap(oper_a) & = & basap(oper_a) + delay(oper_a) - 1 \\ ealap(oper_a) & = & balap(oper_a) + delay(oper_a) - 1 \end{array}$$

and

$$mobility(oper_a) = balap(oper_a) - basap(oper_a) + 1$$

With this information, we can define the possible beginning and terminating control steps for the elements in SBU_a.

- for operations, *oper*

$$\begin{array}{lcl} Cs_{beg}(oper_a) & = & basap(oper_a)\ \ldots\ balap(oper_a) \\ Cs_{end}(oper_a) & = & easap(oper_a)\ \ldots\ ealap(oper_a) \end{array}$$

- for storage, $rmen$

$$\begin{array}{lcl} Cs_{beg}(rmem_a) & = & easap(oper_a)+1\ \ldots\ ealap(oper_a)+1 \\ Cs_{end}(rmem_a) & = & basap(oper_b)\ \ldots\ balap(oper_b) \end{array}$$

- for communication, $link$

$$\begin{array}{lcl} Cs_{beg}(link_a) & = & easap(oper_a)\ \ldots\ ealap(oper_a) \\ Cs_{end}(link_a) & = & basap(oper_b)\ \ldots\ balap(oper_b) \end{array}$$

For operations, the *DG*s can be computed by accumulating the distribution probability along the range defined by Cs_{beg} and Cs_{end}.

$$\begin{aligned} &\forall\, cs_b \in\ Cs_{beg}(oper_a)\ \bigwedge\ \forall\, cs_e \in\ Cs_{end}(oper_a) \\ &\quad \forall\, cs_i \in \{\, cs_b \ldots cs_e \,\} \\ &\qquad Density_{Fu_a}(cs_i) \mathrel{<+} \frac{1}{mobility(oper_a)} \end{aligned} \tag{7.8}$$

The *DG*s of storage and communications can be computed by three nested loops. The outer one goes through the possible beginning control steps and the inner one the possible terminating control steps. The innermost loop is just to accumulate probability into the density of the control steps within the range.
For storage

$$\begin{aligned} &\forall\, cs_b \in\ Cs_{beg}(rmem_a) \\ &\quad \forall\, cs_e \in\ Cs_{end}(rmem_a) \\ &\qquad \forall\, cs_i \in \{\, cs_b \ldots cs_e \,\} \\ &\qquad\quad Density_{Rmem}(cs_i) \mathrel{<+} \frac{1}{mobility(oper_a)\ mobility(oper_b)} \end{aligned} \tag{7.9}$$

For communications

$$\begin{aligned} &\forall\, cs_b \in\ Cs_{beg}(link_a) \\ &\quad \forall\, cs_e \in\ Cs_{end}(link_a) \\ &\qquad \forall\, cs_i \in \{\, cs_b \ldots cs_e \,\} \\ &\qquad\quad Density_{Link}(cs_i) \mathrel{<+} \frac{1}{mobility(oper_a)\ mobility(oper_b)}\ \frac{1}{(cs_i - cs_m)} \end{aligned} \tag{7.10}$$

For instance, the probability distribution of figure 7.9 can be computed as in figure 7.10. For operations, figures 7.10a and 7.10b, these are derived as the reciprical of the possible number of schedules, *mobility*. For storage, *rmem*, the schedules of the two operations need to be considered together. With two schedules from $Oper_a$ and three schedules from $Oper_b$, there are in total six possible combinations of data-life. For all of them, the storage requirement is summarized in figure 7.10d. The distribution graph for storage will be the probability sum of the requirements in each cycle. For communications, *link*, as storage, all possible schedules of the two operations are considered, assuming that with input and output register files, Scheme (b : $| \longrightarrow \Box$) , Scheme (c : $\Box \longrightarrow |$) and Scheme (d : $\Box \longrightarrow \Box$) , only one data transfer is needed. The transfer can take place at any of the control steps between the two operations. For the six possible schedules, the probability distribution of *link* is in figure 7.10e

CS	Fu_a		
0	1		1
1	1	1	2
2		1	1
3			
4			
5			
a.	/2	/2	/2

CS	Fu_b			
0				
1				
2				
3	1			1
4		1		1
5			1	1
b.	/3	/3	/3	/3

CS	Fu_{ab}			
0	1/2			3
1	1/2	1/2		6
2		1/2		3
3	1/3			2
4		1/3		2
5			1/3	2
c.				/6

CS	*Rmems*						
0							
1							
2	1	1	1				3
3	1	1	1	1	1	1	6
4		1	1		1	1	4
5			1			1	3
d.	/6	/6	/6	/6	/6	/6	/6

CS	*Links*						
0							
1	1/3	1/4	1/5				47
2	1/3	1/4	1/5	1/2	1/3	1/4	112
3	1/3	1/4	1/5	1/2	1/3	1/4	112
4		1/4	1/5		1/3	1/4	62
5			1/5			1/4	27
e.	/6	/6	/6	/6	/6	/6	/360

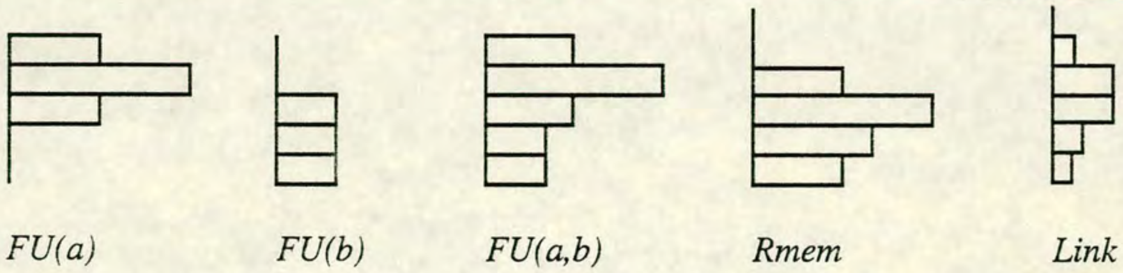


Figure 7.10: Probability Distribution of the Schedule.

Now we have a set of *DG*s for functional units, a *DG* for storage and one for communications. Distribution graphs provide vital information about the current state of a design. The *mean* of a distribution graph gives the theoretic minimum requirements of the resources. The control step with maximum density gives the minimum requirement of that particular type of resource. If some resources are restricted, a vertical line can be drawn on the distribution graph to indicate the *threshold* value. Control steps where their density overshoots the threshold are prime candidates for minimization. Priorities with respect to the cost of each resource can also be derived. Otherwise, a simple method is to locate the maximum density control step and re-schedule the elements. The variance of the distribution graph can then be improved to minimize the resource requirement. To try out the effect of assigning different kinds of resources, various distribution graphs can be built to study the possible density distribution. The effectiveness of this has been demonstrated in [PAPA90]. In that paper, distribution graphs corresponding to different allocations are kept so that scheduling and allocation can be performed together. In addition, in [PAUL87], it has been shown that by overlapping the two distribution graphs with displacement equal to the target pipeline latency, the distribution graph can be optimized for a pipeline design.

7.8.2 Flexibility Damping and Shock Wave

To balance all the distribution graphs at the same time, we use the strategy of *flexibility damping* introduced in Chapter 3.4.

$$\begin{aligned}
DG_{rs} &= \text{DG of resource, } rs \\
rs(sch) &= \text{$inst$, instance : a time range, sch, of resource rs} \\
cs_{max} &= \text{the control step with maximum density} \\
&\quad\; density(cs_{max}) = MAX(density(cs)) \bigwedge cs \in DG_{rs} \\
Inst(cs) &= \text{the instance list of a control step} \\
&\quad\; \{\ rs(sch) : cs \in sch\ \} \\
DL_{max} &= \text{damping list : } Inst(cs_{max}) \\
inst_{min} &= \text{the instance with minimum probability} \\
&\quad\; prob(inst_{min}) = MIN(prob(inst)) \bigwedge inst \in DL_{max} \\
elem_{min} &= \text{the element with $inst_{min}$ in its flexibility list} \\
&\quad\; inst_{min} \in Flexibility(elem_{min})
\end{aligned}$$

```
damping ( $inst_{min}$, $elem_{min}$, $DG_{rs}$ )
    $rs_{min}(sch_{min}) = inst_{min}$
    $\forall\ cs \in sch_{min}$,   remove $inst_{min} \in Inst(cs)$
    remove $inst_{min} \in Flexibility(elem_{min})$
    update $prob(inst)s \in Flexibility(elem_{min})$
    update $prob(cs)s \in DG_{rs}$
```

To balance all the distribution graphs at the same time, the DG of a high density resource, DG_{rs}, is chosen. The control step, cs_{max}, with the highest density is identified. The list of instances which are entitled to be scheduled in cs_{max}, *damping list* or DL_{max}, is examined. The instance, $inst_{min}$, with the *least* probability is selected from DL_{max}. This is because it is *least* likely to be assigned to the control step, cs_{max}. This also ensures the *least* turbulence. The instance, $inst_{min}$, is discarded from the DL_{max} and the flexibility list, $Flexibility(elem_{min})$, of its corresponding element, $elem_{min}$. This process has direct impact on the DG . It will reduce the highest density by the amount defined by $prob(inst_{min})$. At the same time, it also disturbs the probability distribution of the instances. The probability of each of the remaining flexible instances in $Flexibility(elem_{min})$ will be increased. They will be re-calculated and the affected control steps of DG_{rs} will be updated. Finally, other parameters in the SBU will be updated as well.

These changes caused by the damping process will also be propagated up and down to the neighbouring SBUs through the sets of dependency arcs, $(Control : prec)$ and $(Control : succ)$. This creates a *shock wave* scenario. A SBU which is hit by the wave-front will update its own elements. Decisions are then made to determine whether the changes need to be propagated onwards. The propagation will continue until the wave reaches an SBU which has enough *stack* to absorb the shock.

```
shock propagation ( $elem_{min}$, $DG_{rs}$ )
  $\forall\ elem_i \in (Control : prec) \bigwedge (Control : succ)$
    $\forall\ inst_i \in$ flexibility_ overlap ($elem_i$, $elem_{min}$)
      damping ( $inst_i$, $elem_i$, $DG_{rs}$ )
```

The propagation decision is based on the *flexibility overlap* of the two interdependent SBUs.

$$Flexibility_{CS}(SUB_a) \bigcap Flexibility_{CS}(SUB_b) \neq \emptyset$$

It is this dependency relationship between the two operations, $\{\ R\ :\ oper_a \longrightarrow oper_b\ \}$ which must be maintained.

$$\begin{aligned} easap(oper_a) &\leq basap(oper_b) \\ ealap(oper_a) &\leq balap(oper_b) \end{aligned} \tag{7.11}$$

If $\quad easap(oper_a) > basap(oper_b) \quad or \quad ealap(oper_a) > balap(oper_b)$,

they are flexibility overlapped and the changes must be propagated. Otherwise, a flexibility stack exists.

To illustrate this process, let us consider the ASAP and ALAP scheduling in figure 7.11. Initially, their flexibilities are:

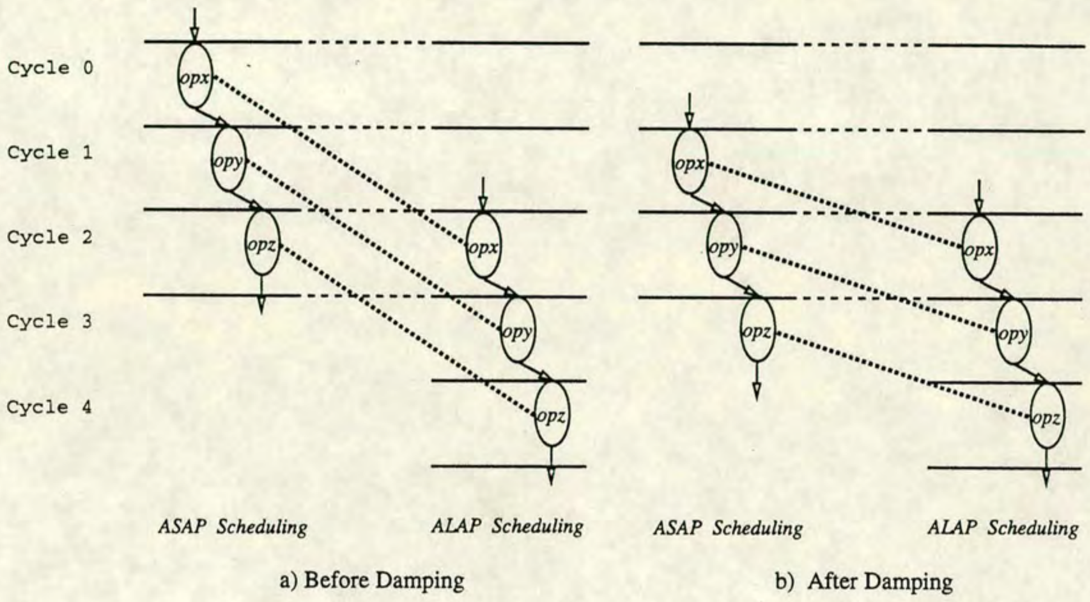


Figure 7.11: The Propagation of Shock Wave.

$$\begin{array}{lcl}
Flexibility_{CS}(oper_x) & = & \{\,(0,0),\ (1,1),\ (2,2)\,\} \\
Flexibility_{CS}(rmem_x) & = & \{\,(1,1),\ (1,2),\ (1,3),\ (2,2),\ (2,3),\ (3,3)\,\} \\
Flexibility_{CS}(link_x) & = & \{\,0,\ 1,\ 2,\ 3,\ \} \\
Flexibility_{CS}(oper_y) & = & \{\,(1,1),\ (2,2)\ (3,3)\,\} \\
Flexibility_{CS}(rmem_y) & = & \{\,(2,2),\ (2,3),\ (2,4),\ (3,3),\ (3,4),\ (4,4)\,\} \\
Flexibility_{CS}(link_y) & = & \{\,1,\ 2,\ 3,\ 4,\ \} \\
Flexibility_{CS}(oper_z) & = & \{\,(2,2),\ (3,3),\ (4,4)\,\} \\
Flexibility_{CS}(rmem_z) & = & \{\,(3,3),\ (3,4),\ (3,5),\ (4,4),\ (4,5),\ (5,5)\,\} \\
Flexibility_{CS}(link_z) & = & \{\,2,\ 3,\ 4,\ 5,\ \}
\end{array}$$

If the instance (0, 0) of $oper_x$ is to be discarded, the elements, $rmem_x$ and $link_x$ will be updated. A shock wave will propagate from SBU_x to SBU_y and then to SBU_z. It will cause the removal of instance (1, 1) in $Flexibility_{CS}(oper_y)$, which in turn causes the removal of instance (2, 2) in $Flexibility_{CS}(oper_z)$. After all the related SBUs are updated, their flexibilities will be:

$$\begin{array}{lcl} Flexibility_{CS}(oper_x) & = & \{ (1,1),\ (2,2) \} \\ Flexibility_{CS}(rmem_x) & = & \{ (2,2),\ (2,3),\ (3,3) \} \\ Flexibility_{CS}(link_x) & = & \{ 1,\ 2,\ 3,\ \} \\ Flexibility_{CS}(oper_y) & = & \{ (2,2)\ (3,3) \} \\ Flexibility_{CS}(rmem_y) & = & \{ (3,3),\ (3,4),\ (4,4) \} \\ Flexibility_{CS}(link_y) & = & \{ 2,\ 3,\ 4,\ \} \\ \\ Flexibility_{CS}(oper_z) & = & \{ (3,3),\ (4,4) \} \\ Flexibility_{CS}(rmem_z) & = & \{ (4,4),\ (4,5),\ (5,5) \} \\ Flexibility_{CS}(link_z) & = & \{ 3,\ 4,\ 5,\ \} \end{array}$$

An important feature about flexibility, unlike mobility, is that the flexibility of an element does not need to be continuous. This allows damping to be applied precisely at the control step where the instance density is high. Using the last example, we can discard the instance (1, 1) of $oper_x$ without any problem. The flexibility list of the SBU_x will be updated to

$$\begin{array}{lcl} Flexibility_{CS}(oper_x) & = & \{ (0,0),\ (2,2) \} \\ Flexibility_{CS}(rmem_x) & = & \{ (1,1),\ (1,2),\ (1,3),\ (3,3) \} \\ Flexibility_{CS}(link_x) & = & \{ 0,\ 2,\ 3,\ \} \end{array}$$

After the changes in SBU_x have been made, the flexibility instance of SBU_y are still valid. Since these changes do not affect any neighboring SBUs, no shock wave is generated. Finally, after the shock wave has died down and all the elements updated, the affected density population of the distribution graphs are recalculated.

7.8.3 Anchoring

At some stage in the damping process, the flexibility of some of the elements will be reduced down to unity. That is

$$\mid Flexibility_{FU}(elem_a) \times Flexibility_{SCH}(elem_a) \mid = 1$$

At this point, $rs_a(sch_a)$ of $elem_a$ will be removed from the instance lists and bound. We call this process *anchoring*.

anchoring ($inst_a$, $elem_a$, DG_{rs})

```
if $| Flexibility(elem_a) | = 1$
    $rs_a(sch_a) = inst_a$
    $\forall\ cs \in sch_a, \quad remove\ inst_a \in Inst(cs)$
    $remove\ \ inst_a \in Flexibility(elem_a)$
    $binding\ (\ inst_a,\ elem_a\ )$
```

Similar to shock propagation, neighbouring SBUs in $(Control : prec)$ and $(Control : succ)$ will be checked to see whether they are affected.

7.9 Results

The flexibility damping strategy has been implemented with sufficient modules in place to demonstrate the method. Several benchmark examples have also been processed.

7.9.1 Differential Equation Example

This example was first presented in [PAUL86]. The computation consists of 6 multiplications, 2 subtractions, 2 additions and 1 comparison. The computation is scheduled to 8 cycles with a 70ns period. For D^3Sch , the following conditions are assumed:

Cycle length: 70 ns
Number of cycles: 8
ALU : { 60+0c+0 : - : - }
Pipeline Multiplier : { 40+0c+40 : 1 : 40ns }
Architectural Constraint : 1 ALU and 1 Pipeline Multiplier

Architecture	Unconstrained	Adjustable Architecture			
	Architecture	A(GRF)	A(IRF)	A(ORF)	A(IRF, ORF)
No. of Buses	3	3	2	2	2
No. of Reg.s	6	6	6	6	6
No. of RFs	-	4	4, 4	3, 1	4, 3
Sum of RFs	-	6	11	7	11

Table 7.1: D^3Sch results of the Differential Equation Example.

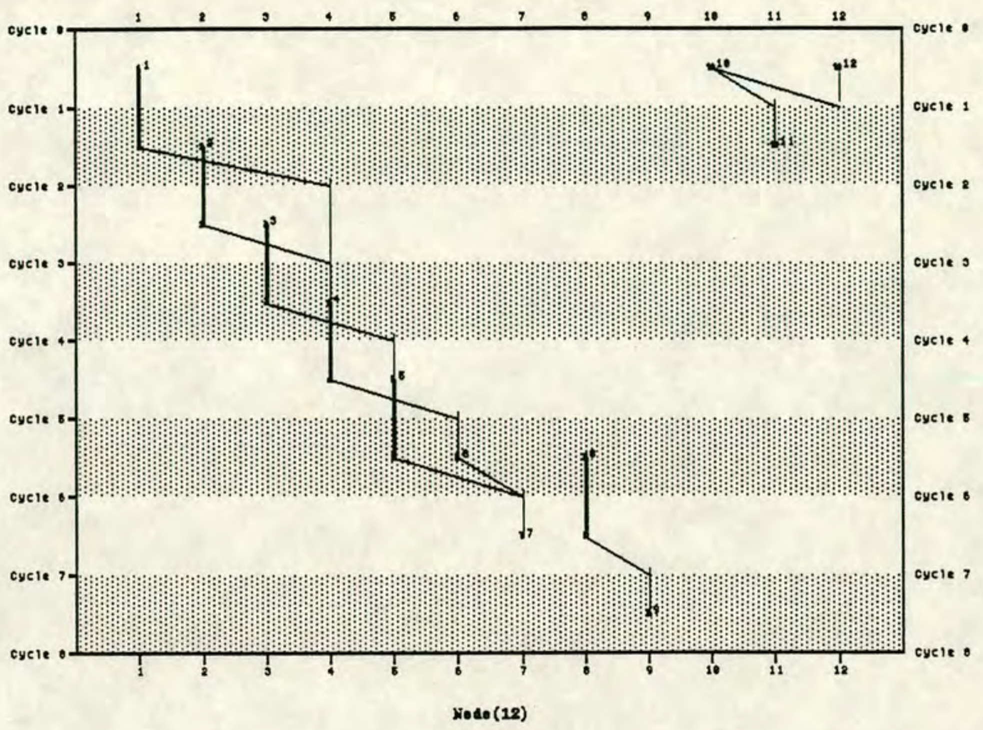


Figure 7.12: D^3Sch of the Differential Equation Example with "A(IRF)".

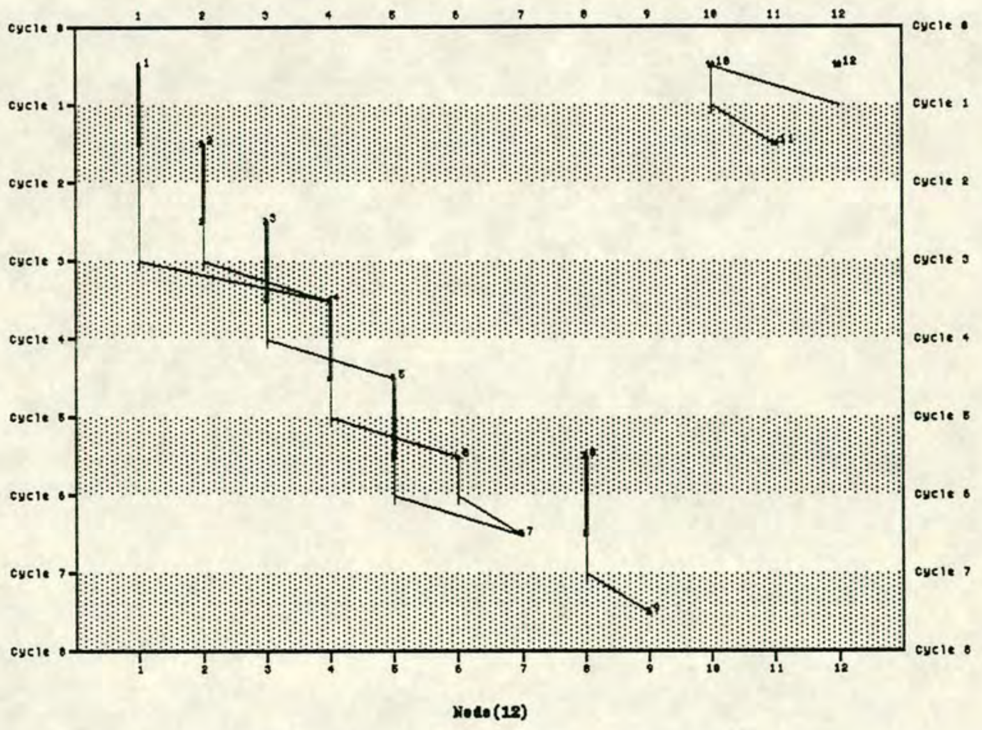


Figure 7.13: D^3Sch of the Differential Equation Example with "A(ORF)".

Subtractions, additions and comparisons are performed with the combinational ALU with 40 ns delay. A pipeline multiplier with two cycles delay and one cycle latency is used to perform all the multiplications. Scheduling with different architectural schemes is performed. These are summarized and presented in Table 7.1.

The first column corresponds to the unconstrained distributed architecture with point-to-point connections. The rest use a bus-based constrained architecture with various storage and communication schemes. The schemes are injected to the flexibility damping process through high-level architectural constraints.

In Scheme A(GRF), data values are assigned to global register files only. Scheme A(IRF) uses input and global register files while Scheme A(ORF) uses output and global register files only. The schemes A(IRF) and A(ORF) can be seen very clearly in figures 7.12 and 7.13. They display the operation schedule, the data transfer schedule and the data storage schedule. Thick lines and numbered nodes are the operations. Thin vertical lines are data storage. They are connected by data transfers in thin slant lines. By counting the number of data transfers in each cycle, it is easy to see that no more than 2 buses are needed in both cases.

Global register files are necessary in all cases. They are used to store the output results in Scheme A(IRF) and input constants in Scheme A(ORF). For the damping process in D^3Sch , user defined resource constraints on the number of functional units and buses are considered first. After the design is brought within these constraints, attention is turned to minimize the number of interconnections. In a traditional approach, this would not be possible because after scheduling, the time frames for the elements would have been fixed. However, for flexibility damping, as long as there is still flexibility left, we can still move the elements around. The small number of buses in Scheme A(IRF) and Scheme A(ORF) reflects the benefit of this approach. In Scheme A(GRF), because there is no input/output register files, nearly all the interconnections have been anchored after the initial constraints have been met.

While adjustable architecture does better in minimizing the number of buses, unconstrained architecture achieves a smaller number of registers. This is because as registers are not attached to the input or the output of functional units, massive sharing and better utilization are possible. As mentioned before, data duplication is $\overset{\text{un}}{\wedge}$avoidable in architectures with input register files. For Scheme A(IRF), the total number of registers in all the register files, $SumofRFs$, is 11, far from the optimal value of 6.

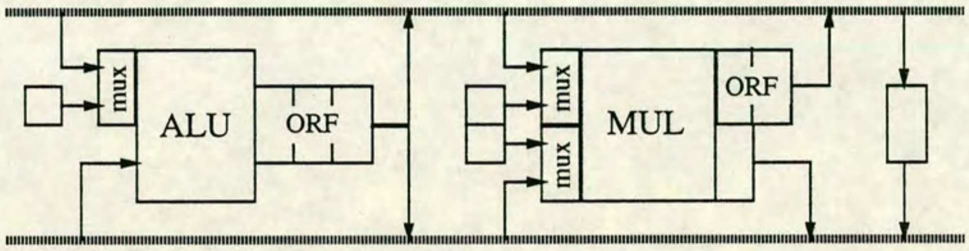


Figure 7.14: Target Datapaths derived from "A(ORF)" result of D^3Sch .

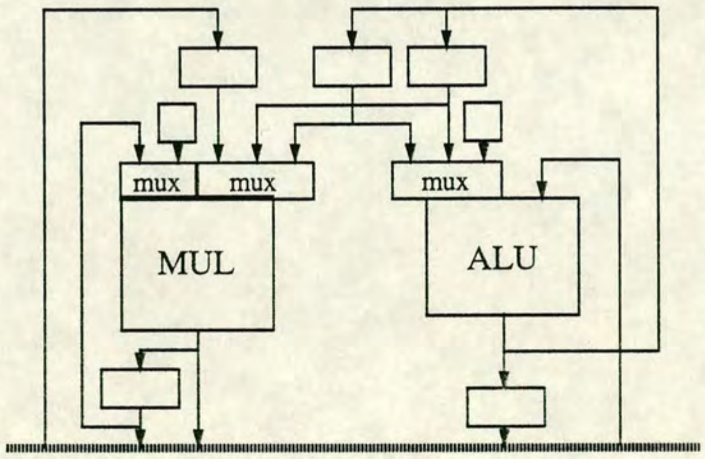


Figure 7.15: Target Datapaths from HAL.

Although the result is fairly good, resource binding has not been performed yet. There is still flexibility available for the binding process. At this stage, some floor-planning information can be considered. Using the initial resource constraints, initial placement can be carried out. From the draft floorplan, the connectivity matrix can be derived which can then be used to drive the anchoring and the binding process. This demostrates one of the strengths of flexibility damping. External information can be injected easily to facilitate the synthesis process.

Figure 7.14 shows the final architecture of the Scheme A(ORF). The bus-based structure is very noticeable. It forms a linear topology. The dual register files at the output port of the multiplier(MUL) reflect the argument discussed in section 7.6.2, i.e. for the case of ORF based architecture, sometimes the number of register files required is greater than the number of output ports in the functional units. Therefore, some of the output ports will have more than one register files. The target architecture synthesized by HAL [PAUL88] is shown in figure 7.15 for comparison. The unconstrained architecture and random interconnections is very likely to cause problems for placement and routing.

7.9.2 5th-Order Elliptic Filter

This example is taken from the 1988 High-Level Synthesis Workshop Benchmarks and has been used in the last chapter for resource restricted scheduling. The filter has 43 operations and 60 edges. 8 of the operations are multiplications and the rest are additions. For D^3Sch with a pipeline multiplier, the following conditions are assumed:

Cycle length: 70 ns
Number of cycles: 19
ALU : { 60+0c+0 : - : - }
Pipeline Multiplier : { 40+0c+40 : 1 : 40ns }
Architectural Constraint : 2 ALU and 1 Pipeline Multiplier

Architecture	Unconstrained	Adjustable Architecture			
	Architecture	A(GRF)	A(IRF)	A(ORF)	A(IRF, ORF)
No. of Buses	6	6	2	4	3
No. of Reg.s	11	11	14	11	15
No. of RFs	-	8	6	5	6, 3
Sum of RFs	-	14	17	13	25

Table 7.2: D^3Sch result of the 5th-Order Elliptic Filter.

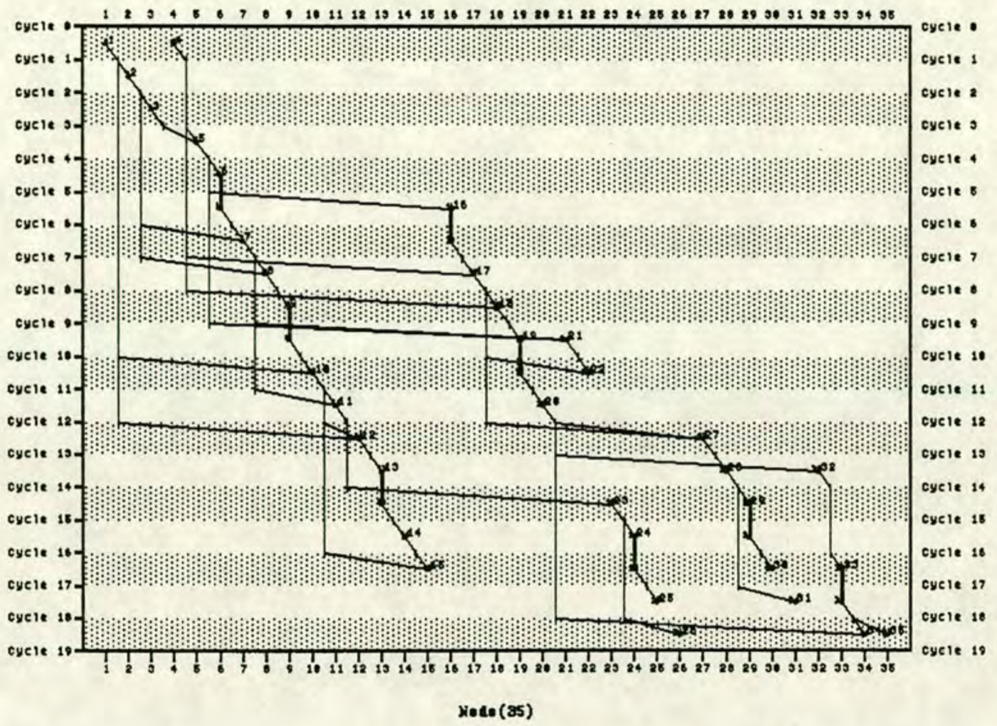


Figure 7.16: D^3Sch of the 5th-Order Elliptic Filter with A(GRF).

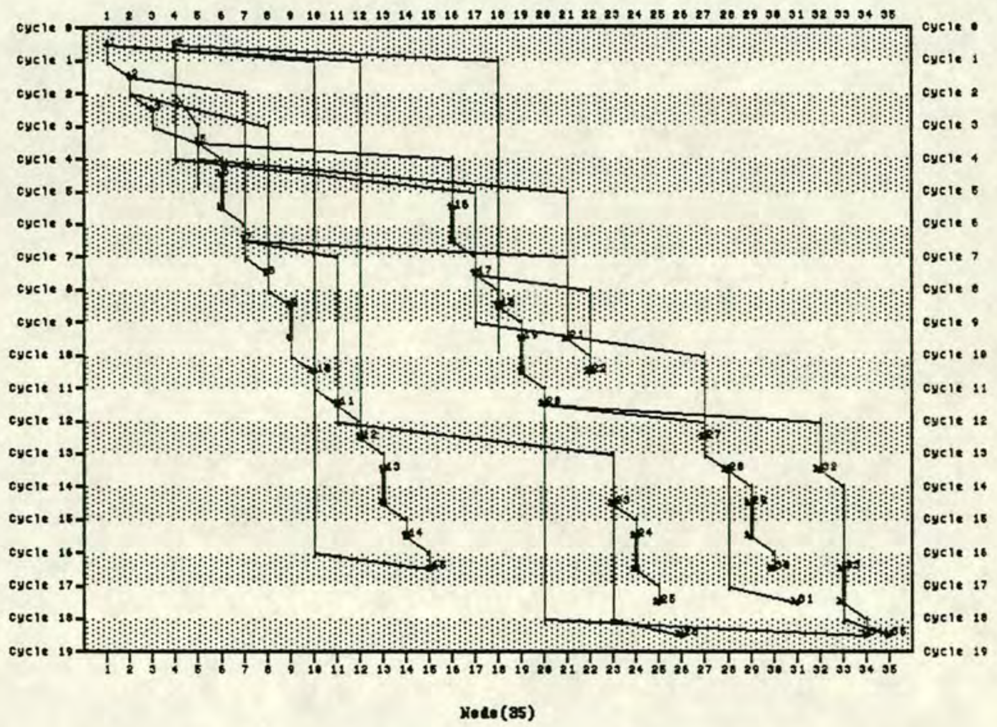


Figure 7.17: D^3Sch of the 5th-Order Elliptic Filter with A(IRF, ORF)

For D^3Sch with a normal multiplier, the following conditions are assumed:

Cycle length: 90 ns
Number of cycles: 21
ALU : { 60+0c+0 : - : - }
Multiplier : { 50+0c+50 : - : - }
Architectural Constraint : 2 ALU and 1 Multiplier

Architecture	Unconstrained	Adjustable Architecture			
	Architecture	A(GRF)	A(IRF)	A(ORF)	A(IRF, ORF)
No. of Buses	6	6	3	3	2
No. of Reg.s	11	11	12	10	12
No. of *RF*s	-	8	6	5	6, 3
Sum of *RF*s	-	15	16	13	27

Table 7.3: D^3Sch result of the 5th-Order Elliptic Filter.

It is first synthesized with a pipeline multiplier with two cycles delay and a cycle latency, and is then synthesized with a normal multiplier with two cycles delay. Additions are performed by two ALUs. The resource restricted ASAP and ALAP schedulings from the last chapter are used to define the initial flexibility along the control steps. Figure 7.16 shows the schedule using a pipeline multiplier with global register files. Because of the dual data transfer, from functional unit to GRF and from GRF to functional unit, nearly twice the amount of buses are required. Figure 7.17 shows the schedule with IRFs, ORFs, and GRFs. Because of the register files, significantly fewer buses are required. However, the early transmission of values at the output port causes massive data duplication at the input register files. There is much room for improvement especially in the scheduling of output transfers.

For comparison, the results are summarized in table 7.4; A(GRF), A(IRF) and A(ORF).
These are compared with results from:

- simultaneous scheduling and allocation, OASIC [GEBO91],
- delay reduction scheduling within CADDY system [ROSE91],
- zone scheduling with integer linear programming, ALPS [HWAN90],

Algorithms	Cycles	Pipe.s	Mult.s	Add.s	Buses	RF.s	Reg.s
A(GRF)	19	1		2	6	8	14
A(IRF)	19	1		2	2	6	17
A(ORF)	19	1		2	4	5	13
OASIC	19	1		2	7		9
CADDY	19	1		2	5		10
ALPS	19	1		2	6		
SPAID	19	1		2	5	5	19
FDLS	19	1		2	8		12
SAM	19	1		2			12
SAW	19		2	2	7		12
CADDY	19		2	2	5		10
A(GRF)	21		1	2	6	8	15
A(IRF)	21		1	2	3	6	16
A(ORF)	21		1	2	3	5	13
ALPS	21		1	2	4		
SPAID	21		1	2	6	6	19
Schalloc	21		1	2	9		13
FDLS	21		1	2			12

Table 7.4: Results of 5th-Order Elliptic Filter.

- DSP silicon compiler, SPAID [HARO89],
- force-directed list scheduling, FDLS [PAUL89],
- simultaneous scheduling and connectivity binding, Schalloc [BERR90],
- combined scheduling, allocation and binding, SAM [CLOU90].

Since slightly different target architectures are used in each case, we must be careful not to compare the results directly. CADDY uses global register files with result storage available for each functional unit. In ALPS, functional units have both input and output latches. SPAID attaches a register file to each bus and input latches are available. FDLS uses global registers only. Even so, provided the same amount of cycles and resources, architecture with IRFs or ORFs, A(IRF) and A(ORF), requires significantly less global communication buses than the others.

It may be thought that the damping process is tedious and slow. However, the run time for the filter example is only a few seconds on a SPARCstation 1.

7.9.3 Fast Discrete Cosine Transform

This is the biggest example processed. The following conditions are assumed:

Cycle length: 70 ns
Number of cycles: 14
ALU : { 60+0c+0 : - : - }
Pipeline Multiplier : { 40+0c+40 : 1 : 40ns }
Architectural Constraint : 3 ALU and 4 Pipeline Multiplier

Architecture	Unconstrained	Adjustable Architecture			
	Architecture	A(GRF)	A(IRF)	A(ORF)	A(IRF, ORF)
No. of Buses	12	12	8	7	6
No. of Reg.s	15	15	-	-	-
No. of RFs	-	8	13	9	12, 7
Sum of RFs	-	19	33	24	44

Table 7.5: D^3Sch result of the Fast Discrete Cosine Transform.

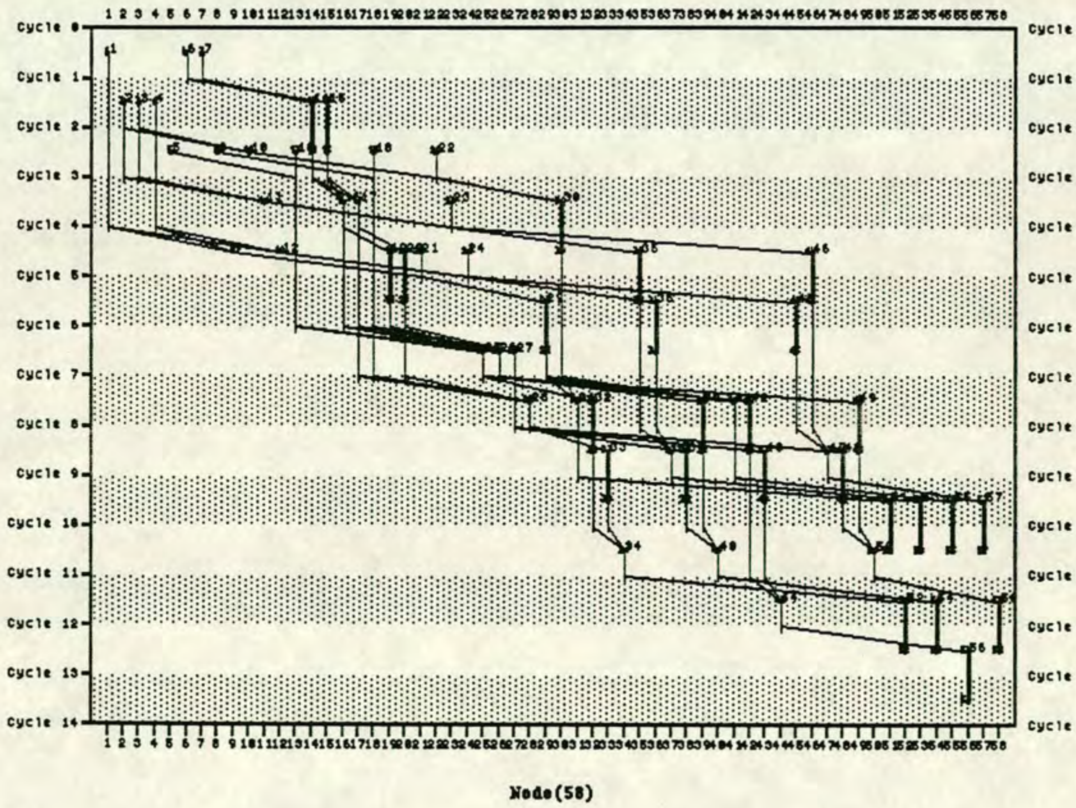


Figure 7.18: D^3Sch of the Fast Discrete Cosine Transform with A(ORF).

It consists of 26 multiplications/divisions and 26 additions/subtractions. The multiplications/divisions are performed on 4 pipeline multipliers with two cycles delay and one cycle latency. The additions/subtractions are performed on 3 combinational ALUs. Figure 7.18 shows the schedule with the architectural constraint: no input register file. Global register files are only used when necessary. The cycle which uses 7 buses is `Cycle 8`.

7.10 Conclusion

In this chapter, we have detailed the adjustable target architecture and the concurrent mapping process. With the advantage of various storage structures, the schedules of storage and communications can be distributed evenly along the control steps. Data transfers are not forced to happen immediately before and after the operations. The concept of concurrent mapping with concurrent scheduling, D^3Sch , using the strategy of flexibility damping has been investigated. The result shows that by considering operations, storage and communication simultanously, a more global cost effective implementation can be achieved. The strategy of flexibility damping avoids the risk associated with premature early decisions. Constraints and different optimizations can be considered gradually and systematically without the loss of generality. In summary, the advantages of flexibility damping are:

- it allows systematic design space exploration;
- it prevents overkill, thus preserving possible good solutions;
- it eliminates possible inefficient use of decision functions which are not fine enough to pick out the best option;
- it allows user interference without direct impact or over-constraining.

On the other hand, the disadvantages are:

- it is slow in obtaining the first point solution;
- for a large design space, efficient algorithms are needed to define the initial flexibility design space to start with. In these cases, predictors as described in [JAIN89] can certainly help.

Chapter 8

Conclusion

In the previous chapters, we have described some of the important steps of *progressive flexibility damping*. Flexibility damping provides a foundation to facilitate the exploraton of a large design space, and allows a stream of tasks to be performed on the design space systemically. By "flexibility damping" the solution space, tasks are prevented from over-committing to some unfulfillable solutions. The occurrence of overkill is avoided and as more information is introduced and made available through the process, better decisions can be made.

Just to recapitulate, the flexibility damping process comprises:

- High-Level Transformation (HLT),
- Coarse Scheduling and Allocation (CSA),
- Fine Scheduling and Binding (FSB), and
- Fine-gain RTL Optimization.

High-level transformation consists of language level transformations (Chapter 4: variable renaming and minimum execution time tree generation), and internal graph transformations (Chapter 5: simplification, local-data and global-data transformations). The main objective of these transformations is to extract parallelism from the original input description. With speculative execution, a massive amount of concurrent operations can be identified. However, significantly more hardware resources are required. Therefore, tradeoffs have to be made between the amount of hardware resources and execution parallelization.

Coarse scheduling and allocation is performed by the resource restricted scheduling algorithm in Chapter 6. By utilizing the parallelism extracted, operations are scheduled into minimum number of clock cycles. At the same time, in order to utilize all the constrained resources as fully as possible, operations are allowed to migrate to wherever resoures are available. Using this partial speculative execution method, with the same amount of resource, better scheduling results are achieved.

Fine scheduling and binding has been implemented in the integrated concurrent mapping process in Chapter 7. By applying the philosophy of flexibility damping, data operations, storage and communications are mapped onto various instances of the target architecture concurrently. This policy ensures that all the cost factors have been taken into account so that the final layout are is minimized.

From the results presented after concurrent mapping at the end of chapter 7. It is shown that the strategy of flexibility damping is highly successful. Compared with the published figures on the benchmark designs from other systems, significantly better results are achieved. Also, with the well defined adjustable target structure, a wider range of architectures are covered.

8.1 Continuous Development of HLS

Next, we would like to take a broader view of high-level synthesis in the real design enviroment. High-level synthesis has been under development for nearly a decade, from the original CMU-DA project started in Carnegie Mellon University around late seventies [PARK79] to the public release of SAW around 1989 [THOM88]. The initial response from industry has been good [SARM90] [FUHR91], but the adoption of high-level synthesis as part of the design process has been slow. High-level synthesis is mostly used in research synthesis systems with very few industrial applications.
The adoption of high-level synthesis has been hindered by a number of factors. In the following sections, we detail the areas in which shortcomings need to be

addressed and the directions for future developments.

8.2 Synthesis Input Language

After the standardization of VHDL as IEEE STD 1076 [IEEE87] in 1987, it has been widely accepted as the simulation and documentation language. Following its success in the simulation sector, enormous pressure has built up forcing synthesis systems to accept VHDL as the input language.

However, VHDL is not well suited for synthesis purposes. The flexibility of allowing user-defined data types and operations introduces extra difficulties into the already complicated problem. They may turn out to be unnecessarily complex and expensive to realize.

Also, VHDL does not support any unified approach for common hardware description constructs, like truth-table, finite state machine, which are commonly found in other HDLs. Possible description styles for FSM are numerous [YEUN91A]. Although they would give the same simulation results, interpreting them for synthesis would not be obvious. It would only be possible if the synthesis system is designed with the description styles in mind and hence can "understand" the input description. In that case, it will be highly stylised, and designers will be forced to follow the guidelines. As these guidelines would be synthesis system specific, it raises the doubt about the suitability of VHDL as the standard synthesis language.

VHDL is a rich language and is based on procedural semantics. Transformation of these models into logic level specification is complex. Some guidelines for standard practice will be very useful, especially for beginners. Writing descriptions which are both simulation and synthesis efficient is very important. On the other hand, to help the specification of constraints, the synthesis subcommittee of the IEEE VASG Modeling Group is defining a "Standard Synthesis Package" for use with synthesis.

8.3 Synthesis Intermediate Format

ASIC users are demanding a universal design methodology within which they can explore the ASIC technologies and perform design trade-offs. To make that possible, the design representation must be technology independent so that the design can be kept *technology transparent*. After it has been functionally verified, it can be targeted according to the target market cost/performance ratio.

A technology independent synthesis intermediate format, as shown in figure 8.1, captures the functionality and the requirements of a design at the front-end. It allows designers to utilize the most appropriate design entry method to enter the design. After the synthesis intermediate format is in place, it serves as a central repository on which synthesis operations can be performed. The result of these operations are back-annotated back into the format through which constraint propagation and design tradeoff can be made. As synthesis goes on, the synthesis intermediate format is gradually transformed from the behavioural level down to the gate level.

Technology independent synthesis tools can be applied in any order, interleaved to ensure efficient transformations and optimizations. These operations also help retargeting and re-optimizing the design to various technologies. After the technology independent optimization steps, different technology fitters can be applied to map the design onto a chosen technology, such as PLD, PGA, gate-array or standard cells. Different fitters are required because different technologies could have very different architecture and require radically different synthesis, optimization and physical design algorithms. Technology migration is a technology switching process. It allows designers to up-grade a design from FPGAs through gate arrays to standard cells for mass production; or vise versa for prototyping.

For highly active research areas such as high-level synthesis, algorithms and methodologies are evolved rapidly. Synthesis intermediate format can facilitate an incremental integration of synthesis systems. Software modules can be arranged like building blocks. Prototype systems can be built quickly. Advanced algorithms can be integrated progressively as they become mature.

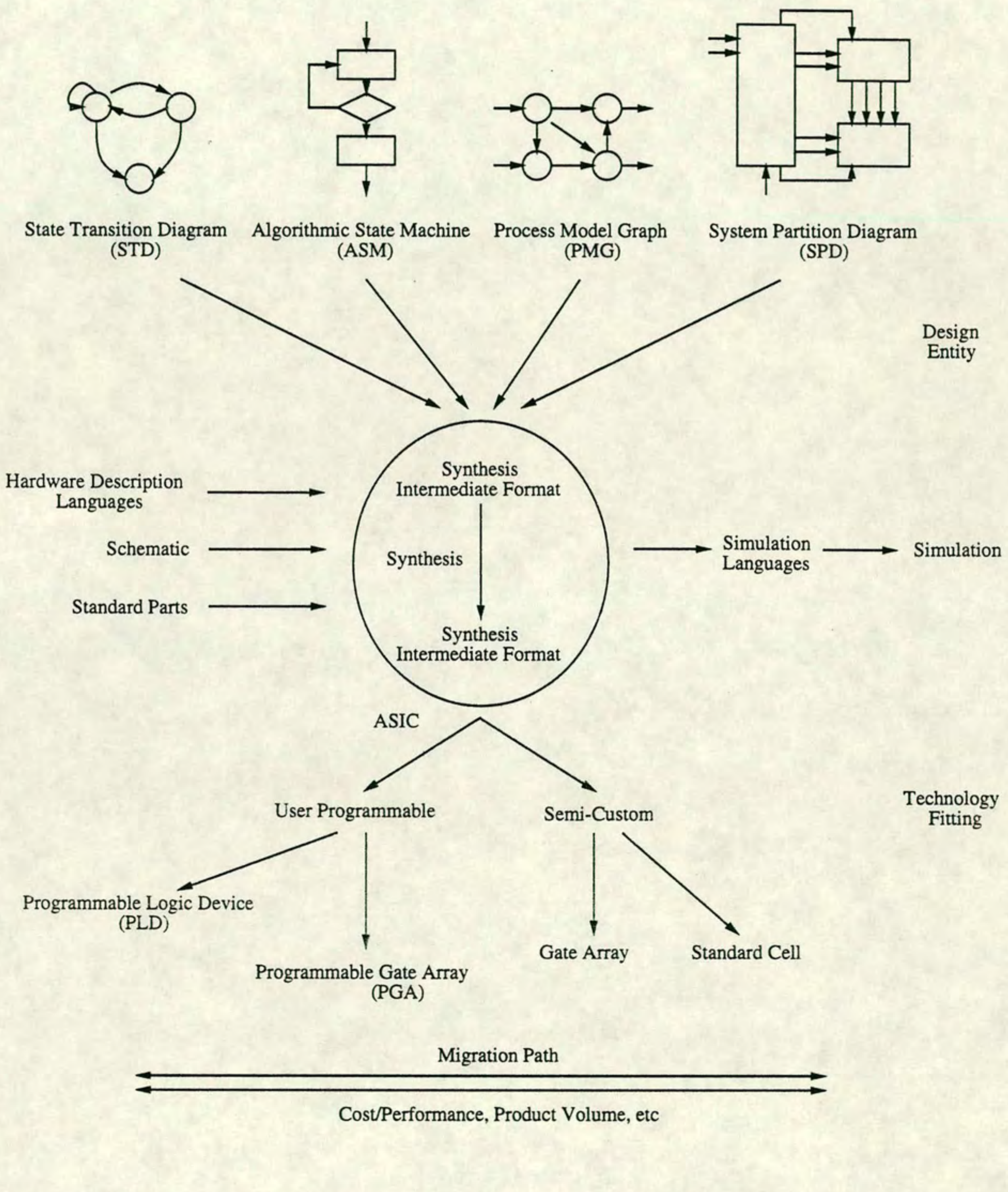

State Transition Diagram (STD)
Algorithmic State Machine (ASM)
Process Model Graph (PMG)
System Partition Diagram (SPD)
Design Entity
Hardware Description Languages
Schematic
Standard Parts
Synthesis Intermediate Format
Synthesis
Synthesis Intermediate Format
Simulation Languages
Simulation
ASIC
User Programmable
Semi-Custom
Technology Fitting
Programmable Logic Device (PLD)
Programmable Gate Array (PGA)
Gate Array
Standard Cell
Migration Path
Cost/Performance, Product Volume, etc

Figure 8.1: Technology Transparent design with Synthesis Intermediate Format.

8.4 Integration with Low-Level Synthesis

It has been observed that optimization for high-level synthesis is particularly difficult since a design is synthesized in a top-down fashion and its performance can only be evaluated after the design has been mapped onto a given technology. As a result, tight design constraints are particularly difficult to meet with this top-down design approach. There are significant interactions between high-level synthesis and low-level synthesis, such as logic and layout synthesis. We cannot merely optimize a design at one level, but instead, must consider the interactions between the levels.

Although high-level synthesis has developed rapidly, the interactions between high-level synthesis and its back-end, low-level synthesis, have not been well understood [CAMP89B]. Low-Level synthesis with sophisticated optimization techniques and the use of mega-functions, mega-cells and module generators all have different implications on high-level decisions. If a high-level synthesis system does not understand the performance and the requirements of the low-level synthesis tools which are used as its backend, severe constraints will be imposed on the lower level with little justification. Low-Level synthesis will also have no channel to feed their results back to high-level synthesis. Consequently, high-level synthesis cannot obtain the information it needs for making important decisions and low-level synthesis does not receive the RTL structure on which low-level optimization can perform well. Detailed experiments on this scenario have been documented in [YEUN91B].

To tackle this problem, it would be ideal if low-level synthesis could be integrated into high-level synthesis. However, the complexity of both problems has made that impossible. To improve the situation, top-down design synthesis approach must be coupled with information passing and constraint propagation in both directions. "Close-loop" design iterations can help low-level synthesis to pass performance information upwards back to high-level synthesis. Using this information, high-level synthesis can do a better job in partitioning the specification constraints for lower levels.

8.5 System Level Synthesis

Following the maturity of high-level synthesis, synthesis research is now pointing towards system level synthesis. The aim of system level synthesis is to partition and implement a system level specification into one or a number of chips. Standard components should also be included wherever appropriate if they can help in satisfying the performance specification. Here, to give an overall view of the subject, we briefly introduce three research systems currently under development.

8.5.1 SpecSyn: System Level Synthesis

SpecSyn is a system level synthesis tool under development in University of California, Irvine. Taking a system level specification and a set of constraints, it synthesizes an interconnected set of chips which satisfies the constraints. Designs are captured by the graphic user interface, SpecCharts [VAHI91]. It allows the designer to enter the functionality and the requirements of a system. SpecCharts describe a system in terms of a hierarchy of state diagrams. The functionality of the primary state is described in sequential statements. Given the specification, *estimators* predict the performance parameters, such as area, execution delay, pin count, power dissipation, etc. This information is fed into the partitioner, *partitor*. If the design is bigger than the maximum size constraint of a chip, it will be partitioned. The *arbiter* constrains and defines the amount of concurrent data access among the chips. After that, communication protocols will be defined. These protocols are handled by *interface synthesis* which puts in ports, interconnections and the necessary signal assignments. Finally, each partitioned chip can be constructed individually by some high-level synthesis tools.

8.5.2 MICON: System for Computer Design

MICON from Carnegie Mellon University [BIRM89] is an assistant tool for computer hardware designers. It targets at single board computers. Implementations are assembled using a set of *off-the-shelf* components to meet the design specification. Its knowledge base system is able to capture previous design information, formalize good design practices, and disseminate the accumulated design knowledge to assist its users. The task performed by MICON is referred to as *configuration* based on *design reuse*. It is ideal for rapid prototyping of computer system. It shortens the learning time, the trial and error time and the design time.

The system consists of a set of tools. **M1**, the rule-based system, transforms the input specification into netlist. **CGEN** (Code Generator) is the knowledge acquisition tool for M1. **FAILURE** analyzes the cause of failures and allows user interaction. **ASSURE** (Automated Synthesis for Reliability) analyzes the implementation produced by M1 and suggests modifications for improvement. **ADEFT** (Automated Design for Testability) modifies the design to meet the testability required. **Database** stores all the parts and components used by the tools.

8.5.3 USC: Unified System Construction

Unified System Construction (USC) [PARK91] from University of Southern California is different from the previous two systems as it targets at multiprocessor systems. It consists of:

- **BAD**: Behavioral Area-Delay Predictor
- **SOS**: Synthesis of Application-Specific Multiprocessor Architectures [PRAK91]
  SOS performs the automatic design of a heterogeneous multiprocessor system. Then, it maps the subtasks onto the architecture and provides a schedule for the execution of the task.
- **CHOP**: A Constraint-Driven System-Level Partitioner [KUCU91]
  It assists the designer to partition the behaviour specification onto multiple chips in order to satisfy the hard constraints which can include individual chip areas, pin counts, system performance and system delay.
- 3D Scheduling [WENG91]
  To consider interconnection delay/cost concurrently with the delay/cost tradeoffs of operations, 3D Scheduling is designed to perform scheduling and floorplanning simultaneously.

8.6 Graphical User Interface

8.6.1 Design Entry

Honestly, designers are not really interested in writing design descriptions in VHDL if there are other easier methods to input the design and simulate them. Designers should be given the choice of describing the design in any well-known high level capture methodologies including, block diagrams, boolean equations, bubble diagram, flow chart, and behavioural specification. Different entry methodologies suit different design target domains and levels of sophistication. In figure 8.1, design can be entered by means of one of the following graphical methodologies:

State Transition Diagram(STD) — bubble diagram,
Algorithmic State Machine(ASM) — flow chart,
Process Model Graph(PMG) — interface handshaking model,
System Partition Diagram(SPD) — block diagram

A graphical hierarchical representation of a system can be built up easily with one graphical representation encapsulated within another. As each of these input methodologies have its own hardware-oriented meanings, it makes sense to generate the system intermediate format directly from the design entry tools. It saves the expensive steps of design compilation, semantic analysis and understanding of the hardware description for synthesis. This is even more difficult especially when there is no standard on synthesis language. For simulation, textual descriptions can be generated automatically in one of the many hardware description languages.

8.6.2 Design Correlation

Unlike logic synthesis, high-level synthesis is not sophisticated enough to handle all the design steps automatically. Even if it is capable of doing so, designers are not comfortable enough with the "hands off" the design process yet. Designers want to interfere with the design and want some assistance from the system so that they can understand what has been generated by the tools. From these understandings, they can then tune the design towards their requirements.

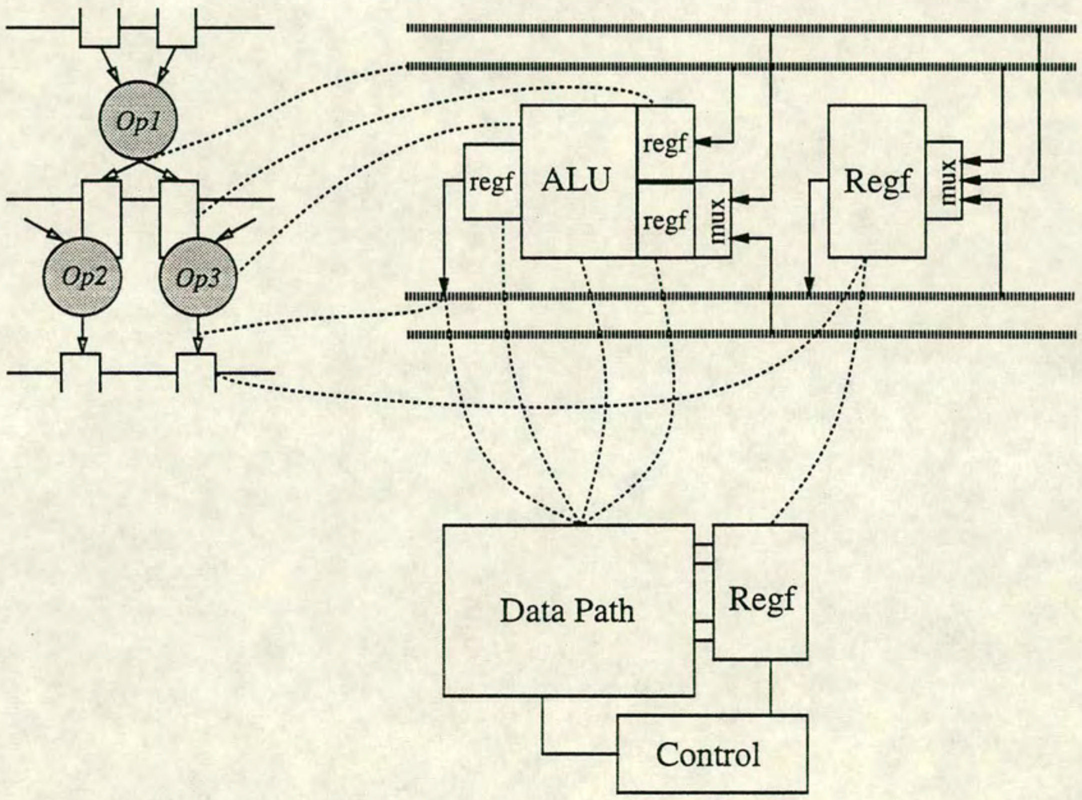


Figure 8.2: Correlation between Design Levels.

The CORAL [BLAC88] linker and the SEESAW graphical tool are the first to introduce the idea of *correlation* between design representations at different levels. The tool displays the input behaviour description, the synthesized control sequence and the resultant RTL schematic. By selecting a construct in the language display, the system will highlight the corresponding modules in the schematic and the corresponding steps in the control sequence. It is very helpful for both designers and beginners to understand how the design specification is implemented by the synthesized structure.

8.7 Conclusion

The distinctive feature of an ASIC product is the design process. At the moment, top-down design with synthesis holds the key to the success of the design process. We have seen how logic synthesis has transformed the industrial design practice. The continuous advancement of synthesis technology, from logic synthesis to high-level synthsis and to system-level synthesis, will continue to be one of the most important factors for the development of ASIC technology.

Chapter 9

Appendix: Input Descriptions

9.1 Differential Equation Example

```
procedure diffeq(
    A : in integer;
    Dx : in integer;
    U : inout integer;
    X : inout integer;
    Y : inout integer)
is
   variable x       : integer;
   variable y       : integer;
   variable u       : integer;
   variable a       : integer;
   variable dx      : integer;
   variable du      : integer;
   variable u1      : integer;
   variable x1      : integer;
   variable y1      : integer;
begin

   x := X;
   y := Y;
   a := A;
   u := U;
   dx := Dx;
   while (x < a) loop
      du := u * dx;
      x1 := x + dx;
      u1 := u - 5 * x * du + 3 * y * dx;
      y1 := y + du;
      x := x1;
      u := u1;
      y := y1;
   end loop;
   X := x;
   Y := y;

end diffeq;
```

9.2 An Example from [PARK86]

```
procedure parker1986(
    in1 : in integer;
    in2 : in integer;
    in3 : in integer;
    in4 : in integer;
    in5 : in integer;
    in6 : in integer;
    out : out integer)
is
   variable t1 : integer;
   variable t2 : integer;
   variable t3 : integer;
   variable t4 : integer;
   variable t5 : integer;
   variable t6 : integer;
   variable t7 : integer;
   variable ot : integer;
begin

      if (in5 ≠ 0) then
         t2 := in2 + in3;
            if (t2 ≠ 0) then
                  t3 := in1 − 4;
                  if (t3 ≠ 0) then t4 := in2 + 4;
                            else t4 := in3 − in5;
         end if;
            else
                  t3 := in4 − 5;
                  t5 := t3 + 5;
                  if (t5 ≠ 0) then t6 := in1 + in2;
                  else
                        t7 := in1 − in2;
                        t6 := t7 + in1;
                  end if;
                  t4 := t6 − in4;
            end if;
            t6 := t4 + in4;
      else
         t1 := in5 − in6;
            if (t1 ≠ 0) then t6 := in2 + 5;
                      else t6 := 8 − in4;
      end if;
      end if;
      if (t6 ≠ 0) then ot := in1 − 5;
                else ot := 8 + in5;
   end if;
   out := ot;

end parker1986;
```

9.3 Fifth-Order Elliptic Filter

```
procedure Elliptic_Filter_Loop(
   In : in integer; i2  : in integer; o2  : out integer;
            i13 : in integer; o13 : out integer;
            i18 : in integer; o18 : out integer;
            i26 : in integer; o26 : out integer;
            i33 : in integer; o33 : out integer;
            i38 : in integer; o38 : out integer;
            i39 : in integer; o39 : out integer; Out : out integer)
is
  const ccc: integer;
  var x12: integer;
  var x14: integer;
  var x15: integer;
  var x17: integer;
  var x19: integer;
  var x25: integer;
  var x27: integer;
  var x3 : integer;
  var x31: integer;
  var x32: integer;
  var x35: integer;
  var x37: integer;
  var x40: integer;
  var x6 : integer;
  var x8 : integer;
begin

    x3 := i2 + In;
    x12 := x3 + i13;
    x32 := i33 + i39;
    x25 := x12 + i26 + x32;
    x19 := ccc * x25 + x12;
    x8 := ccc * (x19 + x12) + x3;
    o2 := ccc * (x8 + x3) + In + x8;
    x27 := ccc * x25 + x32;
    x31 := i39 + ccc * (x27 + x32);
    o26 := x25 + x19 + x27;
    x15 := x8 + x19 + i18;
    x17 := ccc * x15 + i18;
    o18 := x17;
    o13 := x17 + x15;
    x35 := x31 + x27 + i38;
    x37 := ccc * x35 + i38;
    o38 := x37;
    o33 := x35 + x37;
    x40 := ccc * (x31 + i39);
    o39 := x40 + x31;
    Out := x40;

end Elliptic_Filter_Loop;
```

9.4 MC6502 Group1 Instruction

```
extern procedure read(adr : in  word;
                      dbb : out byte)

extern procedure write(adr : in  word;
                       dbb : out byte)

extern procedure addr(adh : in  word;
                      adl : in  word;
                      adr : out word)

extern procedure setnz(ta : in byte)

extern procedure adjuct(tac : in array[9] of bit;
                        opd : in byte)

procedure group1()
is
  variable adr : word;
  variable cpc : word;
  variable opd : word;
  static Pc : word;
  static Ir : byte;
  static X  : byte;
  static Y  : byte;
  static A  : byte;
  static c  : bit;
begin

    read(Pc, cpc);

    case Ir[4:2] is
        when 0 => addr(cpc+X+1, cpc+X, adr);
        when 1 => adr := cpc;
        when 2 => adr := Pc;
        when 3 => addr(Pc+1, Pc, adr); Pc := Pc + 1;
        when 4 => addr(cpc+1, cpc, adr); adr := adr + Y;
        when 5 => adr := cpc + X;
        when 6 => addr(Pc+1, Pc, adr); adr := adr + Y; Pc := Pc + 1;
        when 7 => addr(Pc+1, Pc, adr); adr := adr + X; Pc := Pc + 1;
    end case;

    Pc := Pc + 1;
    read(adr, opd);

    case Ir[7:5] is
        when 0 => A := A | opd; setnz(A);
        when 1 => A := A & opd; setnz(A);
        when 2 => A := A xor opd; setnz(A);
        when 3 => adjuct(A + c + opd, opd);
        when 4 => if (Ir ≠ H"89") then write(adr, A); end if;
        when 5 => setnz(A);
        when 6 => setnz(A − opd); c := A ≥ opd;
        when 7 => adjuct(A + c − opd, opd);
    end case;

end group1;
```

9.5 Fast Discrete Cosine Transform

```
procedure fdct_1d(
   i_plane : in  array[7] of integer;
   f_plane : out array[7] of integer)
is
  variable b : array[7] of integer;
  variable c : array[7] of integer;
  variable d : array[7] of integer;
  variable e : array[7] of integer;
begin

   # first pass
   b[0] := i_plane[7]+i_plane[0];
   b[1] := i_plane[6]+i_plane[1];
   b[2] := i_plane[5]+i_plane[2];
   b[3] := i_plane[4]+i_plane[3];
   b[4] := i_plane[3]+i_plane[4];
   b[5] := i_plane[2]+i_plane[5];
   b[6] := i_plane[1]+i_plane[6];
   b[7] := i_plane[0]+i_plane[7];

   # second pass
   c[0] := b[3]+b[0];
   c[1] := b[2]+b[1];
   c[2] := b[1]−b[2];
   c[3] := b[0]−b[3];
   c[4] := b[4];
   c[5] := 91*b[6]−91*b[5];    # 128*[cos[pi/4]*b6 − cos[pi/4]*b5]
   c[6] := 91*b[6]+91*b[5];    # 128*[cos[pi/4]*b6 + cos[pi/4]*b5]
   c[7] := b[7];

   # scale c5 & c6
   c[5] := c[5]/128;
   c[6] := c[6]/128;

   # third pass
   d[0] := c[0];
   d[1] := c[1];
   d[2] := c[2];
   d[3] := c[3];
   d[4] := c[4]+c[5];
   d[5] := c[4]−c[5];
   d[6] := c[7]−c[6];
   d[7] := c[7]+c[6];

   # fourth pass
   e[0] := 91*d[0]+91*d[1];     # 128*[cos[pi/4]*b0 + cos[pi/4]*b1]
   e[4] := 91*d[0]−91*d[1];     # 128*[cos[pi/4]*b0 − cos[pi/4]*b1]
   e[2] := 49*d[2]+118*d[3];    # 128*[sin[pi/8]*b2 + cos[pi/8]*b3]
   e[6] := 49*d[3]−118*d[2];    # 128*[cos[pi/8]*b3 − sin[pi/8]*b2]
   e[1] := 25*d[4]+126*d[7];    # 128*[sin[pi/16]*b4 + cos[pi/]*b7]
   e[5] := 106*d[5]+71*d[6];    # 128*[sin[5*pi/16]*b5 + cos[5*pi/16]*b6]
```

```
        e[3] := 106*d[6]−71*d[5];    # 128*[cos[3*pi/16]*b6 − sin[3*pi/16]*b5]
        e[7] := 25*d[7]−126*d[4];    # 128*[cos[7*pi/16]*b7 − sin[7*pi/16]*b4]

        f_plane[0] := e[0]/128;
        f_plane[1] := e[1]/128;
        f_plane[2] := e[2]/128;
        f_plane[3] := e[3]/128;
        f_plane[4] := e[4]/128;
        f_plane[5] := e[5]/128;
        f_plane[6] := e[6]/128;
        f_plane[7] := e[7]/128;

    end fdct_1d;
```

Chapter 10

Bibliography

A good survey and summary of references on High-Level Synthesis can also be found in:

"A Survey of High-Level Synthesis Systems" R.A. Walker and R. Camposano, Kluwer Academic Publishers, 1991

[AHO86] A. Aho, R. Sethi, and J. Ullman "Compilers: Principles, Techniques, and Tools", Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[BALA89] M. Balakrishnan, P. Marwedel "Integrated Scheduling and Binding: A Synthesis Approach for Design Space Exploration", Proc. 26th Design Automation Conference, pp.68-74, 1989.

[BARB81] M.R. Barbacci "Instruction Set Processor Specifications(ISPS)", IEEE Transaction on Computers, Jan. 1981.

[BERR90] N.V. Berrey, B.M. Pangrle "SCHALLOC: An Algorithm for Simultanous Scheduling & Connectivity Binding in a Datapath Synthesis System", Proc. European Design Automation Conference, pp.78-82, 1990.

[BIRM89] The MICON Sytem for Computer Design "W.P. Birmingham, A.P. Gupta, D.P. Siewiorek", Proc. 26th Design Automation Conference, pp.135-140, 1989.

[BLAC88] R.L. Blackburn, D.E. Thomas, and P.M. Koenig "CORAL II: Linking Behavior and Structure in an IC Design System", Proc. 25th Design Automation Conference, pp.529-535, 1988.

[BRAY86] R.K. Brayton, R. Camposano, G. DeMicheli, R.H.J.M. Otten, and J.T.J. van Eijndhoven "The Yorktown Silicon Compiler", IBM Research Report, RC 12500(#56205), Yorktown Heights, Dec. 1986.

[BREW87] F.D. Brewer and D.D. Gajski "Knowledge-Based Control in Micro-Architecture Design", Proc. 24th Design Automation Conference, pp.203-209 1987.

[BREW90] F.D. Brewer and D.D. Gajski "Chippe: A System for Constraint Driven Behavioral Synthesis", IEEE Transactions on Computer-Aided Design, pp.681-694, Huly, 1990.

[CAMP87] R. Camposano and J.T.J. van Eijndhoven "Combined Synthesis of Control Logic and Data Path", Proc. International Conference on Computer-Aided Design, pp.327-329, 1987.

[CAMP89A] R. Camposano and W. Rosenstiel "Synthesizing Circuits From Behavioral Descriptions", IEEE Transactions on Computer-Aided Design, Feb, 1989.

[CAMP89B] R. Camposano and L.H. Trevillyan "The Integration of Logic Synthesis and High-Level Synthesis", Proc. IEEE ISCAS Conference, pp.744-747, 1989.

[CAMP89C] R. Camposano, R.M. Tabet "Design Representation for the Synthesis of Behavioral VHDL Models", Proc. 9th Computer Hardware Description Languages and their Applications, 1989.

[CARL91] S. Carlson "Modeling Style Issues for Synthesis", Applications of VHDL to Circuit Design, Kluwer Academic, pp.123-161, 1991.

[CLOU90] R.J. Cloutier and D.E. Thomas "The Combination of Scheduling, Allocation, and Mapping in a Single Algorithm", Proc. 27th Design Automation Conference, pp.71-76, 1990.

[DEMA88] H. De Man, J. Rabaey, J. Vanhoof, G. Goossens, P. Six and L.Claesen "CATHEDRAL-II – A Computer-Aided Synthesis System for Digital Signal Processing VLSI Systems", Computer-Aided Engineering Journal, pp.55-66, April 1988.

[DEMA90] H. De Man, F. Catthoor, G. Goossens, J. Vanhoof, L. Van Meerbergen, S Note, J.A. Huisken "Architecture-Driven Synthesis Techniques for VLSI Implementation of DSP Algorithms", Proceedings of the IEEE, pp.319-335, Feb. 1990.

[DEVA89] S. Devadas, A.R. Newton "Algorithms for Hardware Allocation in Data Path Synthesis", IEEE Transactions on Computer-Aided Design, pp.768-781, July, 1989.

[FUHR91] T.E. Fuhrman "Industrial Extensions to Univeristy High Level Synthesis Tools: Making It Work in the Real World", Proc. 28th Design Automation Conference, pp.520-525, 1991.

[GEBO91] C.H. Gebotys, M.I. Elmasry "Simultaneous Scheduling and Allocation for Cost Constrained Optimal Architectural Synthesis", Proc. 28th Design Automation Conference, pp.2-7, 1991.

[GIRC85] E.F. Griczyc, R.J.A.Buhr, and J.P. Knight "Applicability of Subset of Ada as an Algorithmic Hardware Description Language for Graph-Based Hardware Compilation", IEEE Transactions on Computer-Aided Design, pp.134-142, April 1985.

[GRAN85] J. Granacki, D. Knapp, A. Parker "The ADAM Advanced Design Automation System", Proc. 22nd Design Automation Conference, June 1985.

[GRAN90] D.M. Grant, P.B. Denyer "Memory, Control and Communications Synthesis for Scheduled Algorithm", Proc. 27th Design Automation Conference, pp.162-167, 1990.

[HARO88] B.S. Haroun, M.I. Elmasry "SPAID: An Architectural Synthesis Tool For DSP Custom Applications", Proc. IEEE 1988 Custom Integrated Circuits Conference, 14.4, 1988.

[HARO89] B.S. Haroun, M.I. Elmasry "Architectural Synthesis for DSP Silicon Compilers", IEEE Transactions on Computer-Aided Design, pp.431-447, April. 1989.

[HEMA90] A. Hemani, A. Postula "A Neural Net Based Self Organising Scheduling Algorithm", Proc. European Design Automation Conference, pp.136-140, 1990.

[HUAN90] C.Y. Huang, Y.S. Chen, Y.L. Lin, Y.C. Hsu "Data Path Allocation Based on Bipartite Weighted matching", Proc. 27th Design Automation Conference, pp.499-504, 1990.

[HWAN90] C.T. Hwang, Y.C. Hsu, Y.L. Lin "Optimum and Heuristic Data Path Scheduling Under Resource Constraints", Proc. 27th Design Automation Conference, pp.65-70, 1990.

[IEEE87] IEEE Std 1076-1987 "IEEE Standard VHDL Language Reference Manual", IEEE 1987.

[JAIN89] R. Jain, K. Kucukcakar, M.J. Mlinar and A.C. Parker "Experience with the ADAM Synthesis System", Proc. 26th Design Automation Conference, pp.56-61, 1989.

[JAIN91] R. Jain, A. Mujumdar, A. Sharma, H. Wang "Empirical Evaluation of Some High-Level Synthesis Scheduling Heuristics", Proc. 26th Design Automation Conference, pp.686-689, 1991.

[KNAP86] D.W. Knapp and A.C. Parker "A Design Utility Manager: the ADAM Planning Engine", Proc. 23nd Design Automation Conference, pp.48-54, 1986.

[KNAP88] D.W. Knapp and A.C. Parker "Synthesis from Partial Structure", Proc. IFIP 10th Conference on Design Methodologies for VLSI and Computer Architecture, 1988.

[KOWA85] T. J. Kowalski "An Artifical Intelligence Approach to VLSI Design", Kluwer Academic Publishers, Boston, 1985.

[KUCU91] K. Kucukcakar and A.C. Parker "CHOP: A Constraint-Driven System-Level Partitioner", Proc. 28th Design Automation Conference, pp.514-519, 1991.

[KUNG85] S.Y. Kung, H.J. Whitehouse and T. Kailath "VLSI and Modern Signal Processing", Prentice-Hall Information and System Sciences Series, 1985..

[LANN90] D. Lanneer, F. Catthoor, G. Goossens, M. Pauwels, J.Van Meerbergen, H.De Man "Open-ended System for High-Level Synthesis of Flexible Signal Processors", Proc. European Design Automation Conference, pp.272-276, 1990.

[LEUN88] S.S. Leung, P.D. Fisher, M.A. Shanblatt "A Conceptual Framework for ASIC Design", Proceedings of the IEEE, pp.741-755, July 1988.

[MARW86] P. Marwedel "The MIMOLA Design System: Tools for the design of Digital Processors", Proc. 23nd Design Automation Conference, pp.587-593, 1986.

[MCFA86] M.C. McFarland "BUD: Bottom-Up Design of Digital Systems", Proc. 23nd Design Automation Conference, pp.474-479, 1986.

[MCFA90] M.C. McFarland, A.C. Parker, R. Camposano "The High-Level Synthesis of Digital Systems", Proceedings of the IEEE, pp.301-319, Feb. 1990.

[PANG87A] B.M. Pangrle and D.D. Gajski "Design Tools for Intelligent Silicon Compilation", IEEE Transaction on Computer-Aided Design, pp.1098-1112 Nov. 1987.

[PANG87B] B.M. Pangrle and D.D. Gajski "Slicer: A State Synthesizer for Intelligent Silicon Compilation", Proc. International Conference on Computer Design, Oct. 1987.

[PANG88] B.M. Pangrle "Splicer: A Heuristic Approach to Connectivity Binding", Proc. 25th Design Automation Conference, pp.536-541, 1988.

[PAPA90] C.A. Papachristou, H. Konuk "A Linear Program Driven Scheduling and Allocation Method Followed by an Interconnect Optimization Algorithm", Proc. 27th Design Automation Conference, pp.77-83, 1990.

[PARK79] A.C. Parker, D. Thomas, D. Siewiorek, M. Barbacci, L. Hafer, G. Leive and J. Kim "The CMU Design Automation System: An Example of Automation Data Path Design", Proc. 16th Design Automation Conference, pp.73-80, 1979.

[PARK86] A.C. Parker, J.T. Pizarro and M. Mlinar "MAHA: A Program for Datapath Synthesis", Proc. 23rd Design Automation Conference, pp.461-466, 1986.

[PARK91] A.C. Parker, K. Kucukcakar, S. Prakash, J.P. Weng "USC: Unified Systen Construction", High-Level VLSI Synthesis, Kluwer Academic, pp.331-354.

[PAUL86] P.G. Paulin, J.P. Knight and E.F. Girczyc "HAL: A Multi-Paradigm Approach to Automatic Data Synthesis", Proc. 23rd Design Automation Conference, pp.263-270, 1986.

[PAUL87] P.G. Paulin and J.P. Knight "Force-Directed Scheduling in Automatic Data Path Synthesis", Proc. 24th Design Automation Conference, pp.195-202, 1987.

[PAUL88] P.G. Paulin and J.P. Knight "High-Level Synthesis Benchmark Results Using a Global Scheduling Algorithm", Proc. International Workshop on Logic and Architecture Synthesis for Silicon Compilers, pp.211-228, 1988.

[PAUL89] P.G. Paulin and J.P. Knight "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's", IEEE Transaction on Computer-Aided Design, 1989.

[PENG86] Z. Peng "Synthesis of VLSI Systems with the CAMAD Design Aid", Proc. 24th Design Automation Conference, pp.203-209, 1987.

[POTA90] R. Potasman, J. Lis, A. Nicolau, D.D. Gajski "Percolation Based Synthesis", Proc. 27th Design Automation Conference, pp.444-449, 1990.

[PRAK91] S. Prakash, A.C. Parker "Synthesis of Application-Specific Multiprocessor Architectures", Proc. 28th Design Automation Conference, pp.8-13, 1991.

[RAJA85] J.V. Rajan and D.E. Thomas "Synthesis By Delayed Binding Of Decisions", Proc. 22rd Design Automation Conference, pp.367-373, 1985.

[ROSE91] W. Rosenstiel and H. Kramer "Scheduling and Assignment in HIgh Level Synthesis", High-Level VLSI Synthesis, Kluwer Academic, pp.355-381, 1991.

[SARM90] R.C. Sarma, M.D. Dooley, N.C. Newman and G. Hetherington "High-Level Synthesis: Technology Transfer to Industry", Proc. 27th Design Automation Conference, pp.549-554, 1990.

[THOM88] D.E. Thomas, E.M. Dirkes, R.A. Walker, J.V. Rajan, J.A. Nestor, R.L. Blackburn "The System Architect's Workbench", Proc. 25th Design Automation Conference, pp.337-343, 1988.

[THOM90] D.E. Thomas, E.D. Lagnese, R.A. Walker, J.A. Nestor, J.V. Rajan, R.L. Blackburn "Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench", Kluwer Academic Publishers, 1990.

[TRIC87] H. Trickey "Flamel: A High-Level Hardware Compiler", IEEE Transaction on Computer-Aided Design, pp.259-270, March 1987.

[TSEN86] C.J. Tseng and D.P. Siewiorik "Automated Synthesis of Dath Paths in Digital Systems", IEEE Transactions on Computer-Aided Design, pp.379-386, July 1986.

[VAHI91] F. Vahid, S. Narayan, D.D. Gajski "SpecCharts: A Language for System Level Synthesis", Proc. Computer Hardware Description Languages and their Applications, 1991.

[WENG91] J.P. Weng and A.C. Parker "3D Scheduling: High-Level Synthesis with Floorplanning", Proc. 28th Design Automation Conference, pp.668-673, 1991.

[YEUN91A] P. Yeung, D. Rees "A Comparison of Hardware Description Languages", CSR-13-91, Dept. of Computer Science, University of Edinburgh.

[YEUN91B] P. Yeung, D. Rees "The Role of Logic Synthesis in High-Level Synthesis", CSR-15-91, Dept. of Computer Science, University of Edinburgh.

[YEUN91C] P. Yeung, D. Rees "Resources Restricted Global Scheduling", Proc. IFIP International Conference, VLSI 91, 7.2.1 1991.

[YEUN92] P. Yeung, D. Rees "Resources Restricted Aggressive Scheduling", Proc. European Design Automation Conference, 1992.

[ZIMM86] G. Zimmermann "Top-down Design of Digital System", Logic Design and Simulation, Advances in CAD for VLSI, North-Holland.