High-Level Synthesis Using Structural Input

Iain William Finlay

Thesis submitted for the degree of
Doctor of Philosophy
University of Edinburgh
March 1992.

Abstract

The task of a high-level behavioural synthesis system is to create a structure to implement a given abstract specification of behaviour. The behaviour is specified at the algorithmic level, typically in the form of a high-level programming language. The synthesized structure is described at the register-transfer level. In such systems the synthesis task is guided only by the behaviour and some physical design constraints such as speed and area. This approach frequently leads to difficulties in synthesizing a suitable architecture.

The synthesis system reported in this thesis tackles this problem by enabling the designer to specify structural input alongside the behaviour. The structural input is described at the register-transfer level and need not define a complete structure. The synthesis tool makes use of this input structure by incorporating it into the design where appropriate or as instructed.

This structurally directed approach is shown to give the designer greater control over structural aspects of the design in addition to enabling greater exploration of possible structural solutions.

Declaration of Originality

This thesis, with the exception of Section 5.2.1, was researched and composed by the author in the Department of Electrical Engineering at the University of Edinburgh between October 1988 and February 1992.

Acknowledgements

I would like to acknowledge the help of my supervisor, Prof. Peter Denyer, and the financial support provided by the Science and Engineering Research Council and British Aerospace. Over the past few years I have received help and advice from my contemporaries: Douglas 'Gooby' Grant, Paul Neil, Jonathan Puddicombe and David Mallon.

Iain William Finlay.

Contents

Chapter 5
Scheduling and Binding. 73

Chapter 6
Data Path Allocation. 97

Chapter 1
Introduction.

1.1 Design Automation.

The advent of VLSI technology has made it possible to construct integrated circuits consisting of hundreds of thousands of transistors. In achieving this scale of integration advances have been required in design methodology as well as in fabrication process technology. The complexity of circuits has demanded a structured, hierarchical approach to design to enable the efficient use of a team of designers. As complexity has increased there has been a trend towards automating tasks at higher levels in the design hierarchy [1].

A hierarchical approach is used because it makes it possible to partition the design process into smaller, more manageable subtasks. This permits a more efficient use of designers as each can be apportioned a clearly defined subtask. Moreover, it provides the necessary framework for representing the design at different levels of abstraction. As an example, consider a circuit which adds two numbers. At the most abstract level this could be represented by a module with two inputs and one output, which is defined to perform an addition operation. At a lower level, this circuit could be represented as an interconnection of logic gates; lower still it could be represented by an interconnection of transistors. In moving to levels of higher abstraction, the representation becomes less detailed and so more manageable.

The VLSI design process has three domains of design representation: behavioural, structural and physical. The behavioural domain is used to house the description of the function of the circuit. The structural domain is a transitional representation between the behaviour and the physical circuit; it encompasses descriptions of circuit schematics. The physical domain is encountered in the description of the layout of the circuit in silicon. A behaviour can have many correct structural implementations, just as a structure can have many possible physical implementations [2].

Within each domain a circuit can be represented at various levels of abstraction [3]. A number of commonly defined levels used in VLSI design are presented in Figure 1.1 .

Level	Behavioural	Structural	Physical
Register-transfer	Algorithmic	Interconnection of processors, ports and memory	IC Floorplan, Macro cells
Logic	Logic Equations	Gates, Flip-flops	Standard Cell Layout
Circuit	Network Equations	Transistors, Connections	Transistor Layout

Figure 1.1 Levels of domain abstraction in VLSI design.

The ideal design of a circuit starts with a behavioural description at a high level of abstraction. From this point synthesis proceeds through the structural domain, descending to lower levels of abstraction until it arrives at the layout description of the circuit's transistors in the physical domain. The constraints of the synthesis process are usually expressed as circuit requirements in the physical domain such as the circuit's minimum speed of operation and maximum silicon area.

The first design tasks to be automated centered around the transition from the structural to the physical domain at low levels of abstraction. These were tasks involving circuit layout such as floor-planning and routing. Automation offers a number of advantages. Most notably both the design time and the risk of errors can be reduced. Accordingly, the greatest benefits from automation are gained when it is applied to large scale tasks. This is why layout received the initial attention. As designs increase still further in terms of complexity and size, there is a greater demand for automating synthesis at higher levels of abstraction. Consequently, automation is beginning to spread to the behavioural domain.

At present there are a number of commercial systems which automate synthesis from a register-transfer level description of structure. These systems are commonly termed 'silicon compilers' [2]. In addition, a small number of systems perform logic synthesis. These take a behavioural description at the logic-level and create a corresponding structural implementation. Very few systems exist which accept behavioural descriptions at a higher level. This is where much of the current research effort is concentrated: the high-level behavioural synthesis task.

1.2 High-Level Behavioural Synthesis.

The task of a high-level synthesis system is to provide a means of generating a low level circuit description from a high level specification. Although often considered synonymous, behavioural synthesis denotes a process which is distinct from high-level synthesis. Behavioural synthesis starts from a description of behaviour defined at any level of abstraction with the aim of producing a structural description. Therefore, high-level synthesis need not start from the behavioural domain and that behavioural synthesis need not start at a high-level of abstraction [3] [4] [5].

The synthesis task tackled in this thesis can be termed high-level behavioural synthesis. This requires that the input description be a high level specification of behaviour. This specification is usually algorithmic in nature and is typically expressed using a high-level programming language. A high level description of structure is synthesized from the behavioural specification. In a high-level synthesis system, the structural description would then be 'compiled' to a physical layout representation of interconnected transistors, the task of silicon compilation.

The synthesis task is performed at the register-transfer level. The behavioural description is composed of a set of primitive operations available in hardware at that level. An example in Figure 1.2 defines the behaviour of a wave digital filter at the register-transfer level [6]. The primitive operations are addition and multiplication.

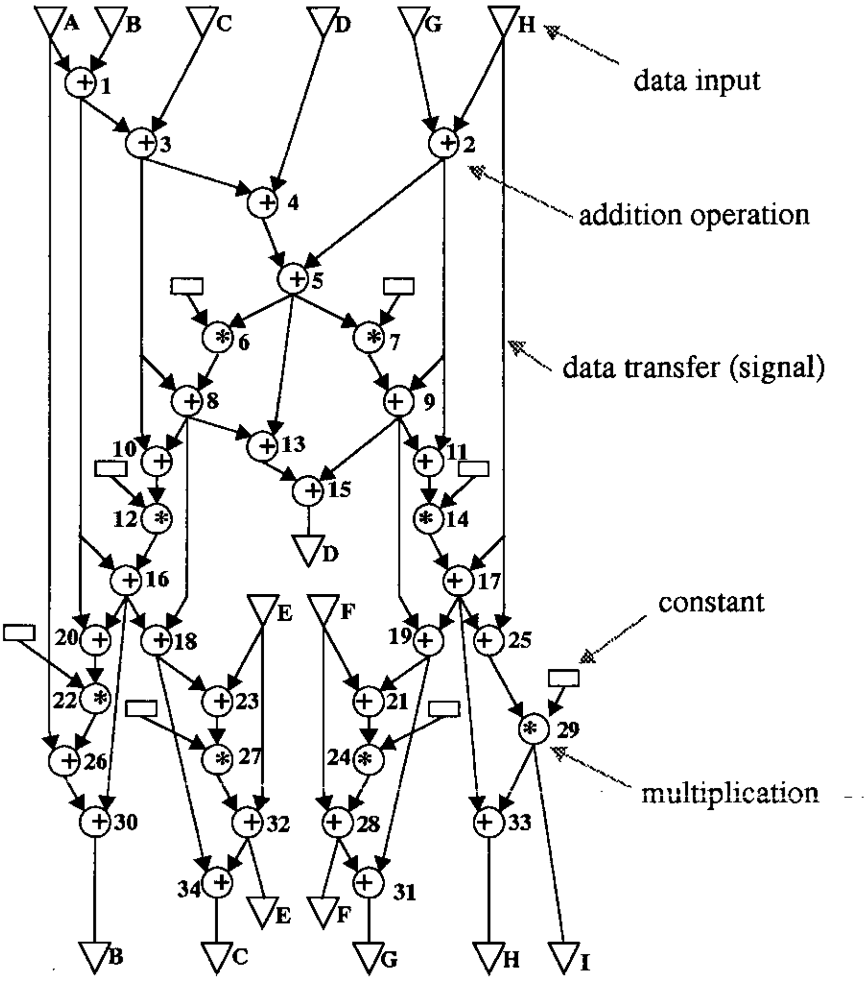


Figure 1.2 Wave digital filter behaviour in a data flow format.

A structure synthesized to implement this behaviour is shown in Figure 1.3.

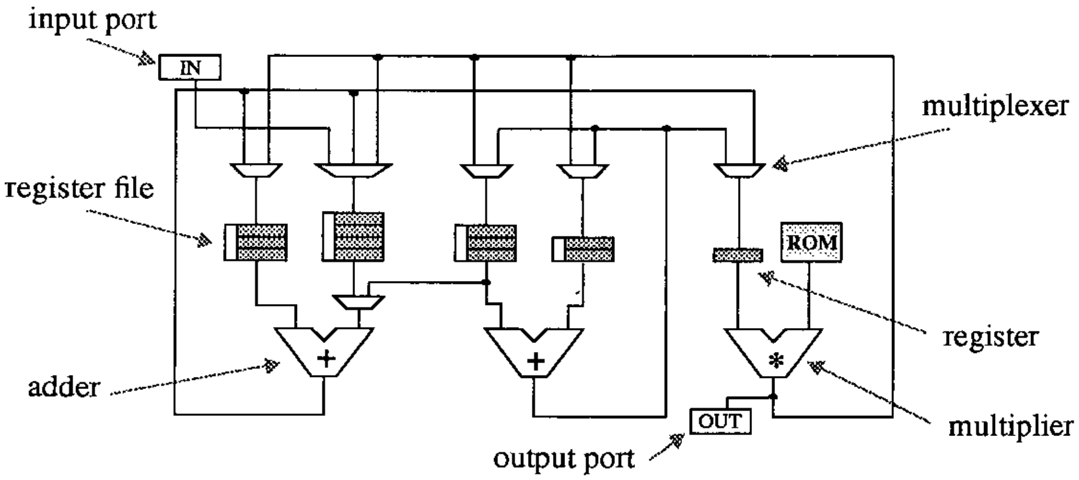


Figure 1.3 Register-transfer schematic of a wave digital filter data path.

This is a schematic view of the data path with control structure omitted. In addition to implementing the behaviour, the circuit is invariably required to meet physical constraints. The most common constraints involve a limit on the silicon area available in conjunction with a minimum speed of operation. As these constraints are made in the physical domain, it is important that a synthesis system supports a suitable back propagation of these requirements to influence higher level design decisions in the structural and behavioural domains. This is a problem widely recognised in high-level synthesis research.

1.3 Structural Direction.

The high-level behavioural synthesis task involves the creation of a register-transfer structure to implement a specified high-level behaviour. Constraints in terms of speed and size will typically be placed on the physical implementation. These constraints will consequently be the sole factors influencing the design architecture. A circuit's architecture is commonly accepted as being the major factor in determining performance. Architecture is a high-level structural attribute defining the style of component interconnection.

The difficulty encountered in most synthesis systems is in producing an architecture to suit a particular problem when given only a specification of behaviour and some physical requirements. When the designer knows of a suitable architecture, there should be some mechanism for suggesting it to the synthesis tool. In most systems, control over the synthesized register-transfer structure is severely limited. At best only the number of allocated components can be influenced. In only one other system is there a facility for controlling the pattern of interconnection [7].

The work presented in this thesis, which takes the form of a prototype high-level behavioural synthesis tool, offers a structurally directed approach to the aforementioned problem. The designer is encouraged to specify partial or complete structures alongside the behavioural input in an attempt to direct the synthesis process towards a suitable architecture. The synthesis tool makes use of these structures by incorporating them into the design where appropriate or as instructed. The structural input is specified at the register-transfer level, comprising of an interconnection of processing, memory and communications components. As illustrated in Figure 1.4, a partial structure is one where interconnection is left incomplete. It is, therefore, acceptable to leave some or all component ports unconnected.

The use of structural input brings a number of benefits. These stem from the ability to control the synthesized structure. If the designer is dissatisfied with the product of fully automated synthesis, he or she is in the position to alter the architectural style or to modify unacceptable parts of the design by repeating synthesis with structural input. Modifications can be made to the synthesized design and need not be complete or correct before repeating synthesis with the altered design. In this way an effective iterative approach to synthesis is possible. In addition, structures may be reused in future designs. This is particularly useful in the case of 'general purpose' structures which require only modification of control to implement a different behaviour. The synthesis system would then be acting as a 'compiler' for such general purpose data paths.

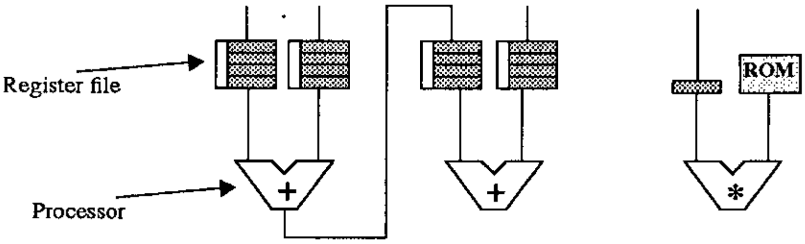


Figure 1.4 Data path schematic of a partial structure used as input.

1.4 Thesis Overview.

In summary, this thesis describes a high-level behavioural synthesis system which accepts a specification of structural input alongside a specification of behaviour. The structural input acts as architectural direction in the synthesis process. This is achieved through the use of specified structures within the synthesized data path. The intention of this approach is primarily to give the designer greater control over the exploration of structural implementations of the behaviour.

To make use of structural input, the synthesis system must be capable of working out how a behaviour can be implemented by a structure in addition to appreciating the performance of different structures in relation to a particular behaviour. This task is termed *mapping*. In this thesis, synthesis has been broken down into three tasks, each performed by a separate tool. These tasks are *mapping*, *scheduling* and *data path allocation*. They are discussed separately in chapters 4 to 6. Chapter 3 outlines the basic framework designed to represent behaviour and structure and to support the synthesis process.

Chapter 2
Behavioural Synthesis Review.

2.1 Historical Overview.

Research into high-level, or behavioural, synthesis started as long ago as the 1960s. The ALERT system developed at IBM's T.J. Watson Research Center produced a logic level implementation from a register-transfer level, behavioural description [10]. Nevertheless, until the 1980s, work in design automation tended to focus on the lower levels of the design hierarchy such as layout. High-level synthesis research was restricted mainly to universities.

At Carnegie-Mellon University, the EXPL system was developed in the early 1970s. This system was the first to enable exploration of the design space by performing serial-parallel trade-offs in hardware. The structural implementation of designs was made using a set of pre-defined register-transfer modules in order to simplify the synthesis task. The EXPL system was superseded by the CMU-DA (Carnegie-Mellon University Design Automation) System in the late 1970s [11]. This new system addressed a range of behavioural synthesis tasks: behavioural specification and internal representation, scheduling and allocation. In addition, it experimented with a wide range of algorithmic techniques for implementing these tasks, ranging from heuristic methods to expert systems.

In Europe, research started in 1976 at the University of Kiel which led to the MIMOLA system [12]. MIMOLA was capable of synthesising a microprocessor from an input specification of behaviour. Synthesis was performed iteratively under the guidance of a designer until constraints were met. At the University of Karlsruhe development of the CADDY/DSL system [13] started in 1979.

The 1980s have seen a dramatic increase in research into high-level synthesis. Advances in VLSI technology demanded greater design automation and rapid developments in both computer hardware and software provided greater opportunity. Automation of the lower levels in the design hierarchy, such as layout, routing and logic synthesis, were being accepted in industry. Many research groups sprang up, expanding research to encompass a wider range of architectural styles and applications. At the University of Edinburgh, the FIRST system was developed for bit-serial application [14]. This was followed in 1987 by the SAGE system [15] [16]. This was developed at the Silicon Architectures Research Initiative, a collaborative venture between the University and six companies. It addressed a wider range of bit-parallel applications using the VHDL language to represent both the behavioural specification and structural design. Similarly at IMEC, synthesis work started with a bit-serial system, Cathedral I. Cathedral II moved into the bit-parallel field but was directed at synthesising designs for a small range of DSP applications for which it had a target architecture [17]. Work at Eindhoven University has led to the development of the EASY system [18].

In North America there has been widespread interest in high-level synthesis. In Canada the Universities of Carelton and Waterloo have developed the HAL [20] and SPAID [25] systems respectively with the backing of Bell-Northern Research (BNR). In the United States some of the more prominent research projects are also in universities: the University of Southern California (ADAM) [27], Carnegie-Mellon University (SAM) [34], University of California at Irvine (VSS) [35], University of California at Berkely (HYPER) [36] and the University of Illinois (IBA) [7].

High-level synthesis has been gaining a foothold in industry. The Pyramid and Phideo systems are being developed at Philips Research Labs [37] and the CALLAS system [38] at Siemens. In the United States the YSC [39] and V-compiler [40] are under development at IBM's T.J. Watson Research Center, Bridge [41] and SAM [42] at AT&T Bell Labs and the Parsifal system at General Electric [43]. Work is also being carried out in Japan at NEC [44] and NTT [45].

2.2 Synthesis Methodologies.

Within every approach to high-level synthesis can be identified four basic tasks which must be addressed. These tasks are as follows:

- *Representation.*

  A typical synthesis system will accept a specification of behaviour and produce a specification of structure. These specifications are usually made using the constructs of high-level languages and hardware description languages respectively. Some languages such as VHDL incorporate both behavioural and structural constructs [46]. Within the system, behaviour and structure are usually represented in a graphical format to highlight the relationships or connectivity between elements.

- *Structural Requirements.*

  In addition to implementing the desired behaviour, the structure is usually required to meet physical and performance constraints. Typical performance constraints are speed of operation and data throughput. Physical constraints are most likely to be in terms of the silicon area which can be occupied by the design or its power consumption.

The remaining three tasks form what is commonly referred to as data path synthesis.

- *Scheduling.*

  Scheduling dictates the relative timing of behavioural operations. Relative timing is expressed in terms of control steps which represent synchronous states based on clock cycles.

- *Allocation.*

  Allocation governs the use and interconnection of structural components. Different types of component, defined at the register-transfer level, are allocated during different stages in the synthesis process. Processors are usually allocated first and interconnecting components, such as multiplexers, wires and buses, are allocated last.

- *Binding.*

  Binding provides the link between behaviour and structure. It is effectively the mapping of behavioural elements to structural components. It is used to determine which processor will implement a particular operation and which memory will store a particular data value.

The synthesis aim is to create a structure which will implement a desired behaviour while satisfying structural requirements. Creating a structure to realise a given behaviour does not present the main problem: it is guiding the synthesis process so that it produces a structure which satisfies realistic requirements which is difficult.

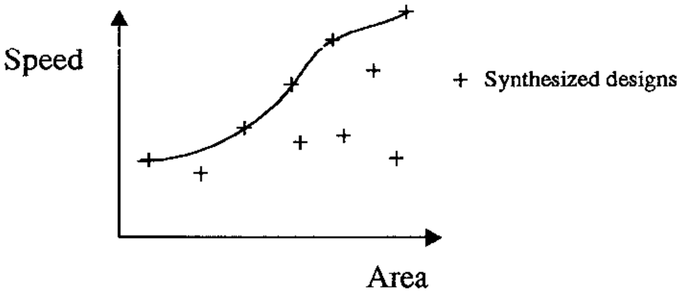


Figure 2.5 Design space bounded by speed and area.

Consider a design where only *speed* and *area* requirements are specified. As shown in Figure 2.5, a simple two-dimensional design space can be constructed using speed and area axes. For many examples, as in Figure 2.5, there is a noticeable trade-off between speed and area. A high-speed design will exploit parallelism in the behaviour, demanding more processing units and, hence, use more silicon area. On the other hand, a smaller area, low-cost design would opt for a more serial approach to operation execution which required fewer processing units but operated at lower speed. This is a highly idealised view of design space exploration. Only for a design whose behaviour contains sufficient parallelism will there exist an obvious trade-off between speed and area which is dominated by the processor allocation. Other factors such as memory, communications and control can have a dominating effect in certain applications. In image processing, memory is frequently the dominating factor, whereas in microprocessor design control is of greater importance. McFarland [47] has demonstrated the effect of communications, involving the use of multiplexers and buses, on both the speed and area of a number of designs. In the majority of cases, efficient design space exploration requires consideration of the combination of processing, memory and communication as all these factors are required to formulate an architecture suited to a particular application.

Synthesis systems can be categorised according to the way in which they attempt to satisfy structural requirements. To do this effectively the system must impart a suitable architectural style to the design. Figure 2.6 outlines three categories of synthesis system.

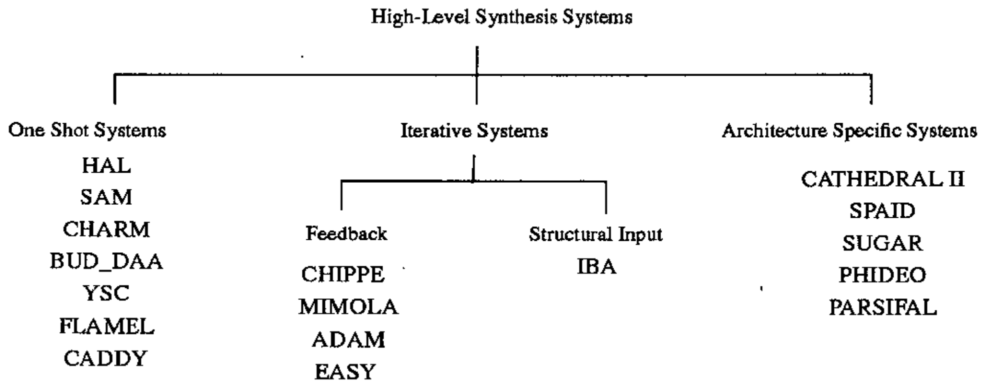


Figure 2.6 Different synthesis methodologies.

2.2.1 'One Shot' Systems.

With 'one shot' systems, designs are discarded if they fail to meet structural requirements. There is no facility to analyse a failed design with a view to modifying it or to directing subsequent synthesis attempts. In other words, the designer has only 'one shot' at synthesising a design to meet a given set of requirements. In some systems it may be possible for the designer to influence certain component allocations or to manipulate the structural requirements in an attempt to synthesize an adequate design. However, the correlation between these parameters and the final solution is not direct enough to enable an effective search of the design space to be made in the majority of cases.

'One shot' systems tend to employ more complex, global algorithms than those using other approaches. There are several reasons for this. Firstly, the algorithms must locate a satisfactory solution first time without relying on designer guidance. Secondly, there is no time constraint imposed on run time because the designer is not expected to interact with the synthesis process or to use it in an iterative fashion. Finally, the algorithms are expected to cope with a whole range of applications and so cannot be simplified by tailoring them to a specific application using heuristics and designer knowledge.

HAL can be classed as a 'one shot' system [20]. The HAL system is dominated by a highly effective scheduling algorithm, called*force-directed* scheduling [21], which has been extended to take into account storage and interconnection factors. Memory and interconnection are synthesized after scheduling using a *weight-directed* clique partitioning method which concentrates on reducing memory costs while taking into account the cost of interconnection. These costs are estimated in terms of the numbers of registers and multiplexer inputs respectively. Both *force-directed* scheduling and *weight-directed* clique partitioning are computationally complex algorithms. Although an attempt has been made to give greater weight to memory and interconnection factors, the allocation of processors and scheduling of operations still dominate the architectural style. This is borne out by the random interconnection topology of synthesized data paths.

The original *force-directed* scheduling algorithm performed processor allocation. The designer could only influence the final design through the specification of speed and area constraints. To enable better design space exploration, *force-directed* list scheduling (FDLS) was developed. FDLS produces a schedule under resource constraints thus giving the designer control over the number of processors or buses used [23]. The effectiveness of the *force-directed* scheduling approach is highlighted by the number of researchers who have made use of this method [18] [48].

2.2.2 Iterative Systems.

Systems which fall into the iterative category make use of synthesised designs which fail to meet structural requirements. The failed design is analysed in an attempt to discover why it failed. Information from the analysis is then used in one of two ways: either to direct subsequent synthesis attempts or to modify the existing design. The iteration loop need not be completed entirely in software but if analysis is left to the designer then he or she must be allowed sufficient control over the synthesis process so as to be able to direct it.

2.2.2.1 Directing Subsequent Synthesis Attempts.

CHIPPE [49] describes itself as a system for constraint driven behavioural synthesis. It performs iterative synthesis entirely in software. An expert system is used to control the resource allocation and strategies used in the synthesis tasks of scheduling and data path allocation. The expert system is guided in these tasks by an evaluator which assesses the quality of the present, complete or partial, design with respect to the design goals.

CHIPPE effects an iterative or closed-loop design methodology by repeating the synthesis process with constrained resources and refined trade-off knowledge gleaned from previous iterations. As the iterations proceed, estimations become more detailed and greater optimization is permitted with the design tools. This means that the first few iterations can be performed quickly, wasting little time on optimizing designs which will be discarded.

The implementation of CHIPPE is focused on CMOS gate array design which simplifies many of the technological and architectural trade-offs, especially in the analysis of bus delays.

The ADAM system [26] adopts a similar approach but addresses the problem of full-custom design. The iterative design loop must, however, be completed by the designer. The ADAM system is effectively a one shot system with area-prediction tools aimed at assisting the designer in making high-level design decisions. Within an iteration, the synthesis process is modelled as a search tree whose nodes represent design decisions. At any node in this tree, if the area-prediction tools show that no acceptable design can be reached then backtracking to other nodes in the tree is supported. Thus, there is the opportunity for considerable design space exploration within each iteration.

2.2.2.2 Modification of Existing Designs.

When a structure is synthesized to implement a specified behaviour, bindings will be created between behavioural elements, structural components and a timing model. This binding information dictates how the structure implements the behaviour and is essentially a specification of the control task. If the structure fails to meet the necessary requirements, but is to be modified in another attempt to satisfy them, then there are at least two possible iterative methods which could be employed. The first method is based on transformations. Structural transformations which maintain the validity of the bindings can be applied to the structure under the guidance of a structural analysis tool or the designer. The second method permits the structure to be modified without considering the bindings. Synthesis tools would then reconstruct the bindings making additional modifications, if necessary, to implement the behaviour. The second method is more advanced as the tools must have knowledge of how a structure implements behaviour rather than just a set of transform rules.

The IBA system [7] developed at the University of Illinois addresses both approaches. The system is composed of three tools, FASOLT, RLEXT and LE, which interact within the framework of the IMBSL language. It is aimed at designing microprocessor-like instruction-set processors. In the majority of such applications there will not be more than one operator of a particular type and so, as a simplification, scheduling is left entirely to the designer.

The transformational approach to iterative synthesis centres around FASOLT and LE. The layout estimator (LE) drives the choice of optimizing transformations which are housed in FASOLT. There are nine possible transformations, which operate at the level of scheduling and allocation. Typical transformations involve component replacement and interconnection component merging and bypass.

The approach which permits structural modification by the designer involves RLEXT (Register Level Exploration Tool). In IMBSL it is possible to specify a partial structure as a guide to synthesis. RLEXT constructs bindings between behaviour and structure and corrects inconsistencies in the specified structure. In IBA's application area, where the structure may contain only a single processing unit, the problem of constructing a binding between operations and processors is obviously trivial. There is no mention in the current literature of a method for constructing a binding to a structure containing multiple processors.

The system presented in this thesis follows the second approach. It accepts partial or complete structural specifications as input and computes feasible bindings between behaviour and structure which the designer is permitted to modify.

2.2.3 Architecture Specific Systems.

Cathedral II is aimed at synthesising synchronous, multiprocessor ICs for DSP applications [17]. There is a target architecture which is tailored to the behaviour. At the highest level the architecture is composed of concurrently operating processors. Each processor is tuned to a subtask of the behaviour and consists of a dedicated data path and controller. The data path is assembled from execution units which are interconnected by a restricted number of buses. Each execution unit has memory in the form of register files which are placed on its input side.

Cathedral II operates a 'meet-in-the-middle' methodology between silicon and system design parts. The design of execution units, which includes layout, is performed by the silicon design part and is saved so that it can be used in other designs.

SPAID [25] is also aimed at DSP applications. Its target architecture is aimed at creating a linear communications topology. Registers are stored in files, each of which is connected to a separate bus. Consequently data will only pass through a single register file and bus between processors. This reduces interconnection delays and simplifies the layout task. The synthesis process allows interconnection to be constrained in the allocation phase prior to scheduling. A two-phase bus cycle is imposed on data transfers, separating reads from writes. The control and external communication architectures are also pre-defined.

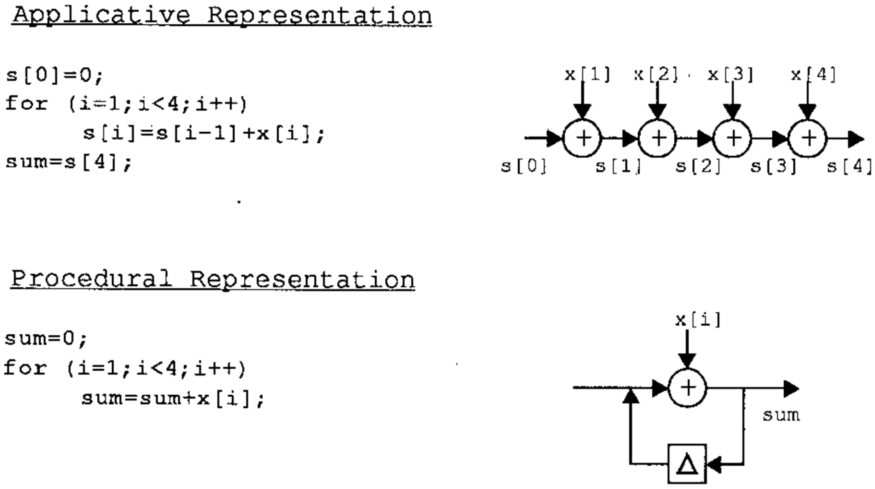
The SUGAR system is dedicated to the synthesis of microprocessors [52]. The synthesis tools contain embedded knowledge about microprocessor subsystems such as the instruction decode unit and condition code branch logic. They also have knowledge of busing structures found in commercial microprocessors. It should be noted that control issues such as the decoding of instructions and assignment of machine code as well as the choice of busing structure are given precedence over scheduling and register assignment.

2.3 Representations.

In the majority of systems, behaviour is the input and structure is the output. It is only in the IBA system [7] and in the system presented in this thesis that a structural specification will be accepted as input.

Behaviour is typically specified in an algorithmic form as a textual description in a high-level language. Most systems have developed their own languages, often based on a subset of an existing procedural language such as ADA, PASCAL or C. More recent approaches make use of hardware description languages, such as VHDL, which can be used to describe both behaviour and structûre.

DeMan advocates the use of applicative languages, such as SILAGE and ELLA, over standard procedural ones [53]. A major difference between procedural and applicative is that procedural languages will permit the multiple assignment of variables as shown in Figure 2.7 whereas applicative languages will only allow single assignments. By allowing a variable to appear on both sides of an assignment a serial implementation is suggested. An applicative language is a more direct representation of true data flow. However, multiple assignment can be easily detected and removed during compilation removing the need for a new language. Furthermore, it can be left to the designer to decide whether he or she uses multiple assignment of variables within a procedural language.



Applicative Representation

```
s[0]=0;
for (i=1;i<4;i++)
      s[i]=s[i-1]+x[i];
sum=s[4];
```

Procedural Representation

```
sum=0;
for (i=1;i<4;i++)
      sum=sum+x[i];
```

Figure 2.7 Comparison of applicative and procedural representations.

SILAGE offers a number of constructs tailored to data flow representations for DSP applications. These include a built in data stream concept; a delay concept on streams of bit-true type and decimation and interpolation constructs.

Algorithmic descriptions of behaviour are compiled into an intermediate format. Compilation can be used to perform common software complier tasks such as:

- *dead code and common sub-expression elimination;*
- *constant propagation;*
- *in-line expansion of procedures;*
- *loop unrolling.*

A number of systems take this opportunity to carry out some hardware-specific transformations.

Internal formats tend to be graphical, consisting of interconnected vertices (nodes) and arcs (edges). Stok has identified three criteria for classifying data and control models of behaviour [18]:

- *using a tree or directed graph representation of data flow;*
- *using a semi-data flow graph or true data flow graph representation;*
- *representing control flow separate from data flow.*

A true data flow graph G(V,E) has vertices (V) which represent only operations and edges (E) which model the data flow between these operations. A semi-data flow graph G(V,W,E), which is usually represented as a bipartite graph, contains two distinct sets of vertices for operations (V) and for variables (W). The edges (E) are used to show the mapping of variables to and from operations.

2.3.1 Tree Representations.

The Cathedral [17] and Mimola [12] systems both represent behaviour using parse trees, a representation taken from software compiler technology. An example of a parse tree representation is shown in Figure 2.8.

```
get(x);
Y:=0.22+0.89*X;
I:=0;
while I<4 do
      Y:=0.5*(Y+X/Y);
      I:=I+1;
endwhile;
put(Y);
```

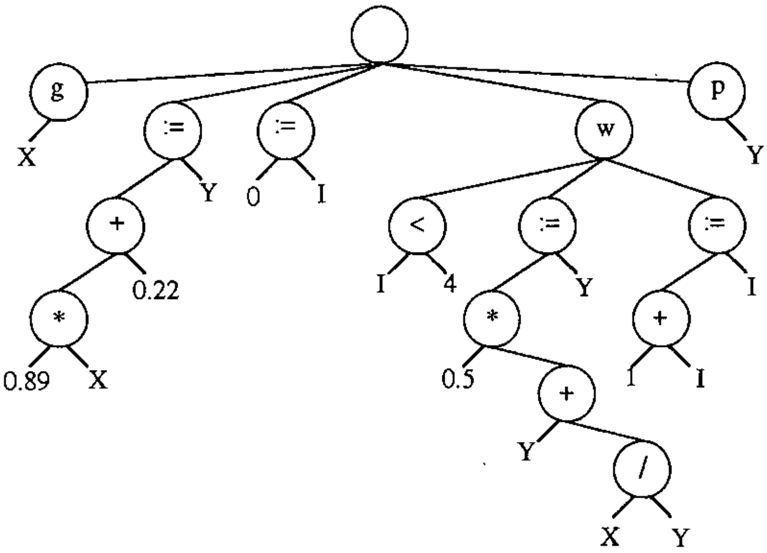


Figure 2.8 Parse tree representation.

2.3.2 Semi-data Flow Graphs.

DDS is the representation model for the ADAM system [54]. It contains four separate models: data flow, timing and control flow, logical structure and physical structure. The semi-data flow graph is bipartite and acyclic. The behaviour is represented hierarchically. At the top level a single node represents the entire behaviour. This is then recursively subdivided until all nodes represent primitive functions. Control is represented using a directed acyclic graph, in which vertices represent events and arcs denote causal relationships between them.

The system at the University of Karlsruhe (CADDY) uses three graphs which share the same vertex representation of operations. The first graph expresses precedence relations between operations. The second indicates the mapping between operations and variables and the third describes timing constraints between operations. From these graphs a true data flow graph is constructed, introducing temporary variables to cope with multiple assignments as shown in Figure 2.9.

```
get(x);
Y:=0.22+0.89*X;
I:=0;
while I<4 do
      Y:=0.5*(Y+X/Y);
      I:=I+1;
endwhile;
put(Y);
```

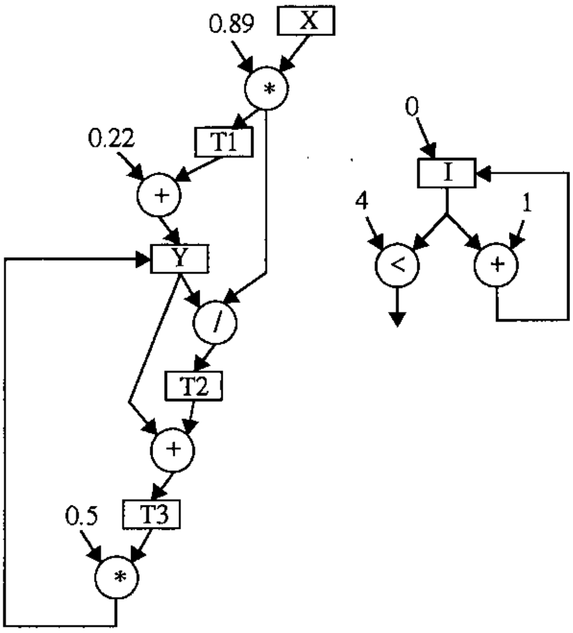


Figure 2.9 Semi-data flow representation.

2.3.3 Separate True Data and Control Flow Representations.

The VHDL Synthesis System (VSS), being developed at the University of California at Irvine, uses separate graphs to represent data and control [35]. Maximal sequences of operations which contain no control flow are defined as basic blocks. These blocks, which represent pure data flow, are then represented by vertices in the control flow graph as shown in Figure 2.10.

```
get(x);
Y:=0.22+0.89*X;
I:=0;
while I<4 do
      Y:=0.5*(Y+X/Y);
      I:=I+1;
endwhile;
put(Y);
```

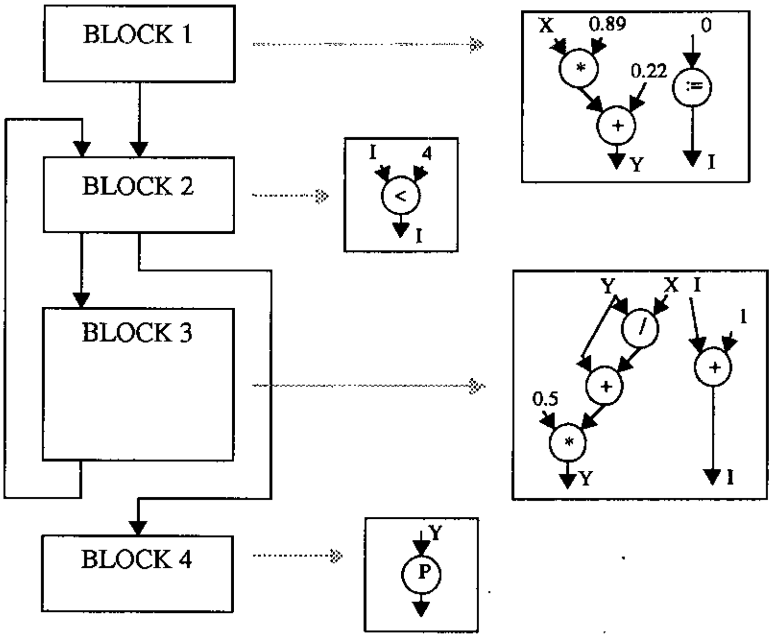


Figure 2.10 True data flow representation.

The ASM system of the University of Illinois uses a similar approach [18]. Camposano [39] uses a slightly different representation for VHDL behavioural models. Operation vertices are present in both the data and control flow graphs.

2.3.4 Combined Representations.

SAW [52] and older CMU-DA systems use a combined data and control flow representation called the Value-Trace (VT). VT is a directed acyclic graph. As in other representations, operations are denoted by vertices; values, or signals, are denoted by edges. Conditional branching is introduced by separate 'select vertices'. Subroutines are mapped as labelled blocks onto subgraphs which are identified by special 'call vertices' which pass parameters. The EASY system [18] uses a data flow graph representation based on the 'demand' graph shown in Figure 2.11. Conditional statements and looping constructs are modelled using the same set of branch and merge vertices.

```
get(x);
Y:=0.22+0.89*X;
I:=0;
while I<4 do
      Y:=0.5*(Y+X/Y);
      I:=I+1;
endwhile;
put(Y);
```

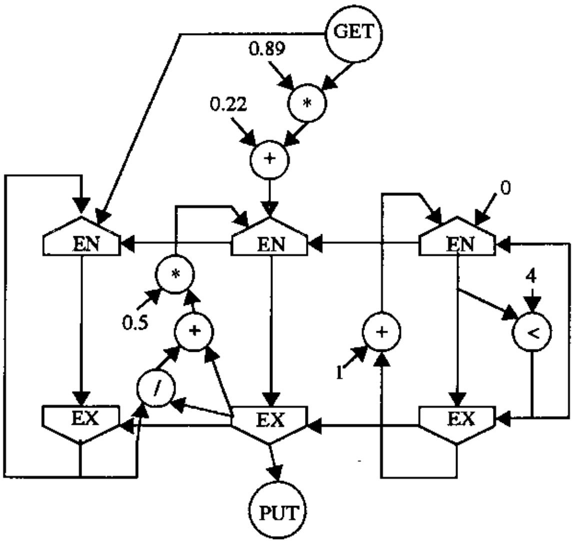


Figure 2.11 EASY demand graph representation.

2.4 Scheduling Techniques

2.4.1 Interaction with Allocation.

Scheduling and allocation are interdependent synthesis tasks. Before an operation can be scheduled, it must be known if a suitable processor is available. Therefore, some allocation must have been made. However, to decide upon a suitable allocation, scheduling information is required so that a trade-off between resource cost and schedule length can be made within specified constraints. Although this is commonly stated as a vicious circle, it is not an intractable problem. The permutations of processor allocation are small for real examples. It is, therefore, even possible to try all realistic possibilities. Consider the wave digital filter example[1] [6]. The data flow graph is large, containing 34 operations, yet the fastest schedule requires an allocation of two multipliers and three adders while the slowest needs one multiplier and one adder. Consequently the total number of realistic allocations is only six. The upper bound on allocation for the fastest schedule can be obtained by analysing the maximum processor requirements for an ASAP[2] schedule.

2.4.1.1 Independent Scheduling and Allocation.

The CMU-DA [11], FLAMEL [70] and DAA [55] systems perform processor allocation before scheduling. The allocation can be set by either the program or the designer. The CHIPPE [49] and MIMOLA [12] systems adopt a similar strategy with the added ability to iteratively re-allocate and re-schedule until a satisfactory result is obtained. MIMOLA uses the designer to control the iterations whereas CHIPPE employs an expert system which governs the data path allocator.

2.4.1.2 Interdependent Scheduling and Allocation.

An alternative approach is to combine processor allocation with scheduling and perform them simultaneously. The *force-directed* scheduling technique of the HAL

[1] Further details of this example can be found in Appendix B.3 on page 162.

[2] As Soon As Possible schedule - definition can be found in section 3.2.2 on page 41.

system tries to balance the load of operations on processors in an attempt to meet timing constraints with the minimum allocation [21]. HAL also employs a *force-directed* list scheduling technique which takes a fixed resource allocation. This is a more effective scheduler as it is not burdened by the allocation problem. It is used in conjunction with the *force-directed* scheduler to enable a more effective search of the design space.

The MAHA (ADAM) scheduler [26] uses a list scheduling approach driven by the notion of operation *freedom* rather than force. The system first invokes the Clocking Scheme Synthesis Package (CSSP) [33] which identifies the critical path and divides it into steps, one per clock cycle. MAHA then allocates functional units for the critical path on a first come first served basis. The notion of *freedom* is used to direct the scheduling of nodes outside the critical path. The operation with least *freedom* is chosen for scheduling and allocated a functional unit if necessary.

2.4.1.3 Step-wise Refinement.

The BUD-DAA system [55] uses a step-wise refinement approach to scheduling and allocation. Operations from the behaviour are grouped into clusters according to their potential for sharing processors or interconnect, or for exploiting parallelism. Processors are then assigned to each cluster before scheduling is performed. A number of different clusters are investigated and the selection producing the result most appropriate to the resource-time criteria is selected. BUD uses list scheduling with a similar *a priori* function to that used in SLICER (CHIPPE) [50] and MAHA [26].

In the YSC system [39] scheduling is separated into two stages. Initially each operation is bound to a separate processor and the same control step. Control steps are added to accommodate loop boundaries and memory usage conflicts. In the second stage the hardware is optimized by attempting to share resources as much as possible. Extra control steps are added, if necessary, to reduce the amount of hardware used and the optimization stage is repeated.

2.4.2 Scheduling Algorithms.

Scheduling algorithms can classed in one of three basic categories: transformational, global or constructive.

2.4.2.1 Transformational.

A transformational scheduling algorithm starts with a default schedule. This is usually either maximally parallel or maximally serial. Transforms are then applied in an attempt to obtain a schedule which meets specified requirements. The basic transforms move operations or blocks of operations to either exploit parallelism and reduce the schedule length or exploit serialism and reduce the resource overhead.

The EXPL system uses a exhaustive search approach starting with a maximally serial configuration. YSC uses heuristics to guide the transformations, starting instead with a maximally parallel configuration. Transformations used in the YSC system can be shown to produce the fastest possible schedule, in terms of control steps, for a given specification [3].

2.4.2.2 Global Scheduling.

So-called *global scheduling* algorithms apply an existing optimization technique to the scheduling task. Three such techniques are integer linear programming (ILP), simulated annealing and neural networks.

(a) Integer Linear Programming.

There have been a number of different approaches to scheduling which use ILP. Papachristou's formulation [56] of the scheduling problem attempts to minimise the cost of processing units. A binary decision variable x_{ij} is defined to represent an operation i being assigned to control step j. Another set of variables Mt_i show the number of permitted processors (modules) of type t_i. Constraints are added to ensure that:

- *no control step contains more than Mt_i operations of type, t_i;*
- *operations are scheduled within their execution windows;*
- *all predecessors of an operation are scheduled before the operation itself.*

This formulation performs scheduling under timing constraints. An alternative formulation put forward by Huang [57] allows scheduling to be done under hardware constraints.

The main draw-back of the ILP approach is the time taken to compute results, even for medium-sized data flow graphs. Most methods are aimed at reducing the problem complexity. Huang's method [57] considers only a few control steps at a time in order to reduce the problem size. This, of course, means that it is no longer being applied as a global optimization technique.

Gebotys [58] has put forward an alternative formulation which, it is claimed, can be solved efficiently by drawing on similarities between the node-packing problem and the way in which the scheduling problem is expressed.

(b) Neural Networks.

Hemani [59] has put forward a self-organising algorithm based on a neural network model which can schedule a data flow graph under a timing constraint. Operations compete for control steps within their execution intervals. The neural model takes into account the effect of scheduling an operation on its neighbours. The size of the neighbourhood is gradually reduced until only the movement of an individual operation is considered. The effect is a uniform distribution of operations across the schedule space. Again, this sort of approach is computationally intensive and, hence, time consuming. However, it is suited to running on massively parallel machines.

(c) Simulated Annealing.

Two approaches using simulated annealing have been identified [60] [61]. In both, the scheduling problem is viewed as one of two-dimensional placement of operations, where the dimensions are those of resource and time. The aim of the algorithm is to minimise the resource-time area according to a costing function which advises on a suitable rectangular shape as a boundary to the operation placement. The simulated annealing algorithm selects operations at random and perturbs them in resource-time space according to a current value of temperature set by the algorithm. An operation is entitled to move to a position of lower global cost. However, if the temperature is high enough, it is also possible that a move to a position of higher global cost will be accepted. The temperature is controlled by the algorithm to be initially high then cooling off in an exponential manner. This algorithm is often hailed as optimal. It does offer a mechanism for bypassing local cost minima by enabling operations to move to higher cost states initially but there is no guarantee that this will result is an optimal solution. The performance of the algorithm is dependent upon the number of peturbations or iterations and the temperature cooling function. Finding a good solution can, therefore, be time consuming for large examples.

2.4.2.1 Constructive Scheduling.

The majority of scheduling algorithms fall into the constructive category. These algorithms are characterised by the fact that operations are scheduled individually, in turn. Within this category, two different schemes can be identified: list scheduling and distribution based scheduling. List schedulers operate on a control step by control step basis starting with the first step. Distribution based schedulers, on the other hand, select operations for scheduling based on criteria relating to the current state of the schedule.

(a) List scheduling.

Within the algorithm, a list of operations is constructed for each control step. This is a list of all operations whose data constraints have been satisfied; in other words those operations whose inputs are available at the time of the current control step.

In an ASAP scheduler all operations in this list are scheduled in the current step.

This is effectively scheduling without hardware constraints as sufficient hardware is allocated to accommodate all operations in each step. The CMU-DA system adopted this simple approach to scheduling which produces the fastest possible schedule although often at the expense of excessive hardware cost [11].

In other approaches to list scheduling the allocation of resources is constrained before scheduling begins. Consequently, the situation may arise where there is insufficient hardware available to accommodate all operations in the list. In this case the scheduling of some of the operations must be postponed, or deferred, until a later control step. Deferred operations will automatically then appear in the list constructed for the next control step. List scheduling approaches differ in the way in which operations are selected for deferral.

The ELF system [62] uses an *urgency* weight to determine deferral. The *urgency* of an operation is defined as the length of the shortest path from the operation to a timing constraint. The operations with the greatest *urgency* are scheduled first leaving operations of lower *urgency* to be deferred. The CSTEP (SAW) scheduler uses a similar method [52].

SLICER (CHIPPE) calculates a *mobility* criterion for deferral from initial ASAP and ALAP schedules [50]. The difference between ALAP and ASAP schedule times for an operation is defined as its *mobility*. Operations with a greater *mobility* are more likely to be deferred. Operations with the same *mobility* are sorted according to the number of successors; the operation with the most successors will be scheduled first. The MAHA (ADAM) system uses a *freedom* calculation which is identical to that of *mobility* [26]. Operations on the critical path will automatically have no *mobility* or *freedom*. Therefore, both the MAHA and SLICER approaches can be called critical path scheduling as operations on the critical path will always be given priority. In the ATOMICS scheduler of CATHEDRAL II operations on the critical path are also scheduled first [17].

The *force-directed* list scheduling approach of the HAL system uses a force calculation to determine deferral [21]. Force measures an operation's attraction to the current control step based on the predicted distribution of operations on resources over time. It is the operation with the lowest force which is deferred. The deferral process is a repeated calculation of forces for operations, followed by the deferral of the operation with the lowest force until all remaining operations can be scheduled in the current step.

All of the list-schedulers outlined so far attempt to calculate the minimum schedule under a resource constraint. A scheduler which minimises resources under a time constraint is described in [57]. In this approach operations are deferred in an attempt to keep the resource overhead as low as possible provided that the time constraint will not be violated.

(b) Distribution-based Scheduling.

The main exponent of the distribution-based approach is Paulin's *force-directed* scheduler which is used in the HAL system [21]. The ASAP and ALAP schedules are calculated, as in SLICER, to determine the 'time frames' for each operation. It is assumed that an operation will have an equal probability of being scheduled anywhere in its time frame. This enables distribution graphs to be constructed for each operation type. These graphs show the probable number of operations of a given type which will occur in each control step. A force is then calculated which attempts to balance the load of operations across the control steps. This calculation takes into account the effect of scheduling on an operation's predecessors and successors. The operation selected for scheduling is the one which yields the lowest force. This operation is then scheduled and the process repeated.

The CASCH scheduler in the CADDY system [13] is also distribution-based in that it attempts to average the number of operations of each type across the schedule.

2.5 Data Path Allocation Techniques.

The data path allocation task typically encompasses the allocation of memory and communications components and the binding of behavioural elements to structural components. In a number of systems the binding of operations to processing components is done during the scheduling phase. Data path allocation is performed after scheduling and there is seldom interaction between the two phases.

Data path allocation is usually broken down into three tasks which are mutually dependent:

- *binding of operations to processors;*
- *register allocation: allocation of memory and binding of signals to memory;*
- *allocation of interconnect to enable transfer of data between memory and processors.*

In most systems the data path allocation centres around register allocation. A number of optimal algorithms exist which will solve this problem. The cost of the interconnect allocation is, however, dependent on the register allocation. As McFarland has illustrated, interconnect is an important cost factor in designs [47]. Consequently, there has been considerable effort made to integrate the memory and interconnect allocation tasks.

The greatest problem encountered in data path allocation is assessing the quality of designs. Interconnection cannot be measured directly as a number of components or units. Its impact does not become clear until after layout and routing, by which stage data path allocation decisions will have been obscured by those taken in layout and routing. The majority of systems make do with three simple metrics in assessing data path quality: the number of register, the number of buses and the number of multiplexer inputs. The number of multiplexer inputs is taken to include the number of bus inputs. It is equivalent to the number of two-input multiplexers or tri-state drivers required.

2.5.1 Optimal Register Allocation.

Initial work in the area of data path allocation focused around the register allocation problem and ignored interconnection issues. As signals can share the same register if they do not exist at the same time, the register allocation problem is viewed as one of grouping signals with disjoint life-times with the aim minimising the number of groups.

The most effective method is the Left-Edge Algorithm used in REAL [30], part of the ADAM system. The algorithm, taken from channel routing, models signal lifetimes as wires which have to be assigned to a minimum number of tracks, representing registers, without overlap. This is achieved simply by placing signal lifetimes in the left most register which avoids lifetime overlap.

Clique covering is an optimal, although NP-complete, technique used in a large number of data path allocation schemes [63]. An undirected graph G(V,E) is partitioned into cliques. A clique is a subgraph where all vertices connect to all other vertices in the subgraph via a single arc. An example is given in Figure 2.12. Clique partitioning algorithms tend to be constructed to find only maximal cliques, those which are not subsets of any others. In clique covering, the aim is usually to identify the smallest set of cliques which include the most vertices of the graph.

When applied to register allocation, the vertices (V) in the undirected graph, G(V,E), denote signals and the edges, (E), indicate disjoint lifetimes between adjoining signals. G(V,E) is termed a compatibility graph. The aim of the algorithm is therefore to find the minimum clique covering of the graph as each clique represents a register allocation. This approach was used initially by Tseng [64].

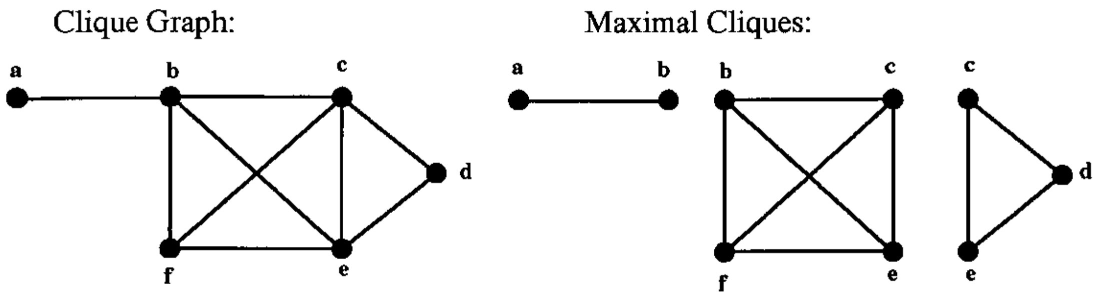


Figure 2.12 Clique partitioning.

Edge colouring is another graph theoretical technique used in register allocation. Again, the problem is posed as an undirected graph where the vertices denote signals. The edges, however, represent lifetime clashes. Stok uses an edge colouring formulation to tackle the problem of cyclic register allocation [19]. This looks at an optimal assignment of signals to registers in the presence of loops in the data flow graph. The problem is transformed into a multicommodity network flow problem for which well-known solutions exist.

2.5.2 Graph Theoretical Techniques taking into account Interconnect.

Clique covering and colouring techniques have been extended to take into account the effects of register allocation on the interconnect allocation.

The HAL data path allocation scheme uses a weight-directed clique partitioning method [24]. A standard compatibility graph is constructed. Each arc is then labelled with a weight corresponding to the interconnection cost of combining the two signals into a single register. Four different interconnection patterns are recognised, however, the weight associated with them is an estimate of the interconnection area that would be saved and so it is design dependent rather than simply heuristic. Clique partitioning is performed on the part of the graph which exceeds a given threshold. The vertices of the located cliques are merged and the weights recalculated. The threshold is progressively lowered and partitioning repeated until no more cliques can be found. As clique partitioning is a NP-complete problem, keeping the graph size small by using a threshold enables exhaustive clique searching, even for large problems. Multiplexer merging is performed using a similar technique. Here, the threshold weight is the number of common inputs between multiplexers.

EASY performs data path allocation in three stages [18]. The binding of operations to processors is performed first using a maximal weight clique cover. The cliques with the highest sum of edge weights are selected. Again, a compatibility graph is constructed. The edge weightings are termed *advantages*. *Advantage* is defined as the net cost of binding operations to the same processor taking into account the multiplexing cost. The edge weights are updated as the cliques are merged. Register allocation, the second stage, uses a modified left-edge algorithm. The *advantage* weighting is used again to try to force signals with similar sources and destinations into the same register. This reduces the amount of interconnect required while still yielding the minimum number of registers. The final stage is called register grouping. This is an attempt to reduce the number of buses by grouping registers into register files. The maximal weight clique cover algorithm is again used in this optimization step. The compatibility graph represents registers as vertices and has arcs between registers which are never accessed simultaneously. The edge weighting reflects the number of common sources and destinations of adjoining registers.

The CADDY system uses a weighted graph colouring technique [13]. Two graphs are constructed: a restriction graph and a preference graph. The former is the complement of the compatibility graph. The latter depicts preferred combinations of signals because of shared sources and destinations. The restriction graph is coloured using a general vertex colouring heuristic to establish a minimum number of colours and, hence, the minimum number of registers. Colouring is then repeated, taking into account preferences. If the number of colours required in the second process is close to the minimum then the colouring is accepted and the registers allocated. If it is not accepted then edges with the smallest weight are removed from the preference graph and the colouring process is repeated. Binding of operations and the construction of buses is performed using a similar colouring approach with both restriction and preference graphs.

2.5.3 Branch and Bound Technique.

SPLICER [50] performs the tasks of register allocation and operation binding while allocating interconnect. Control step by control step, starting with registers containing initial values, the algorithm provides interconnection to processors and then from processors to registers. At each step an attempt is made to use existing hardware. The algorithm is based on a branch and bound search which allows backtracking.

2.5.4 Iterative Techniques.

CHARM [42] claims to have an algorithm which performs register allocation and binding by dynamically exploiting the trade-off between the binding of operations to processors, the binding of signals to registers, the number of processors and the number of registers. The algorithm is based on the iterative merging of compatible operations and their respective data path allocations. Operations are compatible if they can share the same processor and are performed at different times. On each iteration all possible merges of operations and groups of operations are considered. The merge which yields the largest cost gain is selected. Iteration stops when no merge produces a cost saving. The costing function is a combination of four factors: the area costs of processors, registers and multiplexers and the estimated area cost of connecting wires. It is during costing that registers and interconnect are allocated.

In MABAL (ADAM) [28] the main algorithm tries to minimise the total cost by trading off between component cost and interconnect cost for each binding decision. The algorithm allows a limited amount of backtracking. It deals with operations and signals in the order in which they were scheduled. Incremental allocation decisions are taken on the basis of costs calculated for possible allocations with respect to the current partial design. In this way an operation can be allocated a new processor if the interconnection cost of re-using an existing one is too high.

2.5.5 Rule-based Schemes.

DAA, which is a rule-based expert system, makes allocation decisions on partitions of a design provided by BUD [55]. BUD clusters data flow operations and signals into partitions according to factors including common functionality, degree of interconnection and potential parallelism. Each partition is allocated its own processors, memory and interconnect. Local optimization procedures are then applied to remove or combine components. Finally, optimizations are applied globally to remove unneeded components and allocate bus structures.

In CATHEDRAL II [17], the rule-based 'Jack the Mapper' performs the data path allocation tasks. CATHEDRAL II generates data paths based on a fixed architectural template and this is reflected in the sort of rules used in 'Jack the Mapper'. These rules contain knowledge such as how to construct addressing schemes for array memory, bus structures and loop counters for iterations.

Chapter 3
Models for Behavioural Synthesis.

3.1 Introduction.

The synthesis system reported here operates on input specifications of behaviour and structure to produce design specifications of structures with associated control information. The control information is necessary to enable the synthesized structure to implement the specified behaviour. The synthesis of structure from behaviour can be performed at different structural levels such as register-transfer, logic gate and device. High-level behavioural synthesis systems typically relate the behaviour to the structure at the register-transfer level as shown in Figure 3.1. In the system presented in this thesis all structures are defined and constructed at the register-transfer level.

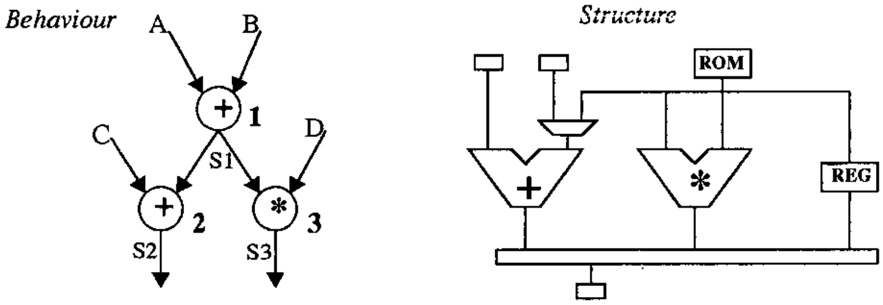


Figure 3.1 A representation of behaviour and structure at the register-transfer level.

The behaviour is related to the structure by a mapping from its behavioural elements to structural components. In order to avoid unnecessary composition or decomposition of behavioural elements, they are defined at a level corresponding to that of the structural components. In other words, a structure consisting of adders, multipliers and registers would relate to a behaviour consisting of additions, multiplications and binary data words.

Behaviour, structure and control form three separate domains of the synthesis system which communicate externally. Each is represented within a separately defined model. The following three sections discuss each model in turn, defining the subset of the domain which is represented and how it is specified. The final section outlines the use of these models within the synthesis system and how the designer can interact with them to direct the synthesis process.

3.2 Behavioural Model.

Behaviour is represented within the system by a single data flow graph of the form shown in Figure 3.2. The data flow consists of operations and signals. Within the graph operations are denoted by vertices and signals by directed arcs. Input signals and constants are supplied by signal-source vertices and output signals are removed by signal-sink vertices. The directed arcs within the graph indicate the direction of data flow, the transfer of data with time. Consequently, data flow graphs are acyclic: it is impossible for a loop to exist as this would imply that data could be consumed before it was created.

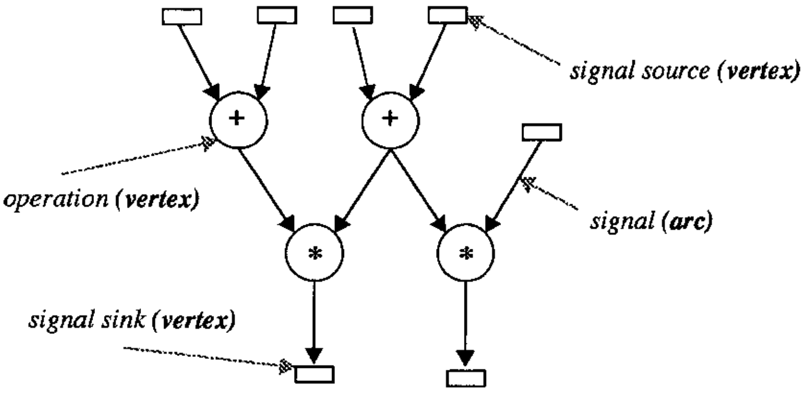
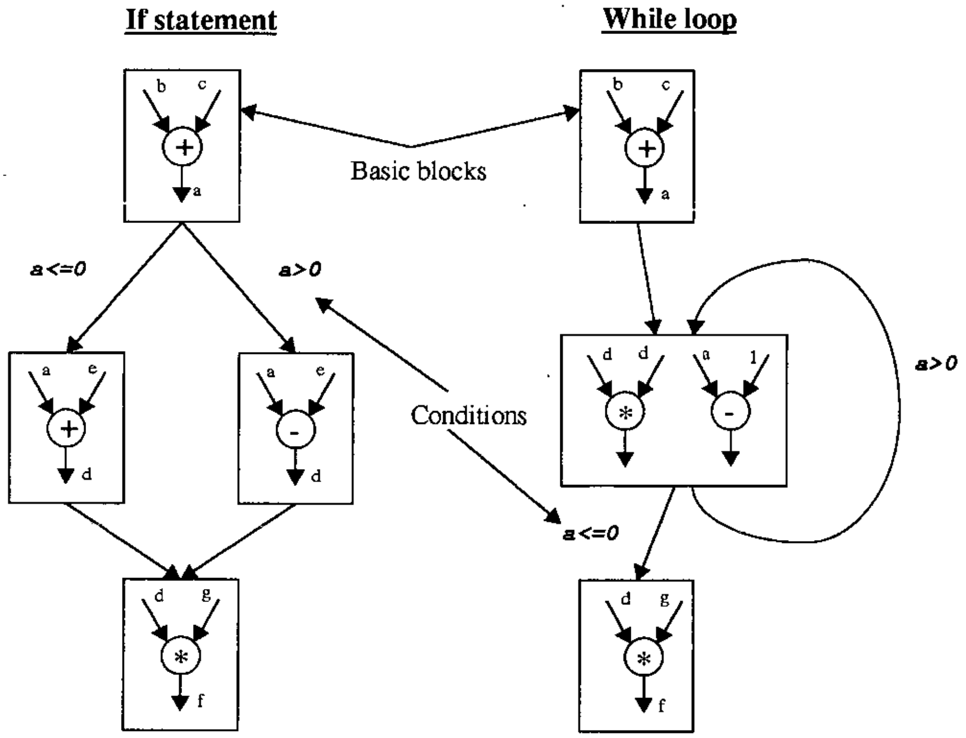


Figure 3.2 Representing behaviour in a data flow graph.

This simple behavioural representation cannot express conditional branching within the data flow. This is when the direction of data flow at some point in the graph depends upon an actual data value. The ability to represent conditional branching is necessary if the behaviour is to express high level programming constructs such as 'while loops' and 'if statements'. The synthesis system presented in this thesis does not tackle conditional branching within the data flow graph.

However, it is envisaged that the approach to synthesis reported here can be harnessed within a larger system to tackle behaviours containing conditional branching. Using a representation from compiler technology called 'basic blocks', a behaviour containing such constructs can be modelled [65]. Each 'basic block' is a simple data flow graph representation of behaviour. These blocks form vertices in a control flow graph. The directed arcs in the graph represent conditions based on data flow values and point to the next data flow graph blocks which will be realised if the condition is satisfied. In this way, housing data flow graphs within a control flow graph, data dependent data flow can be represented. A simple example is illustrated in Figure 3.3.



```
a:= b+c;
if a> 0 then
      d:=a-e;
else
      d:=a+e;
end if;
f:=d*g;
```

```
a:=b+c;
while a>0 loop
      d:=d*d;
      a:=a-1;
end loop;
f:= d*g;
```

Figure 3.3 Representing conditional constructs with basic blocks.

The synthesis system reported in this thesis can tackle individual basic blocks from the control graph. Assuming no pipelining, the execution of all basic blocks is mutually exclusive: only one block can be active at a given time. Even if pipelining is used basic blocks originating from the same conditional branch will remain mutually exclusive. A reduction in the structural overhead can therefore be made if structures can be synthesized to implement more than one basic block. A synthesis system using structural input can be used to address this problem. A structure synthesized to implement one block can be specified as the structural input in the synthesis of an other basic block in order to make additional use of the structures's components.

3.2.1 Behavioural Elements.

The data·flow graph behaviour is constructed entirely from operations and signals. Because the structure is defined at the register-transfer level, which consists of a transfer of data between processors and memory, the operations are specified at a level corresponding to the functionality of the processors and the signals at a level corresponding to the data storage format of the memory.

3.2.1.1 Operations.

All operations are dyadic. They are restricted to two inputs and one output. Operations with more inputs must be represented by a combination of two-input operations. The order of inputs is important. For non-commutative operations such as subtraction and division, the order of inputs must be maintained to preserve functional correctness. Each operation is defined as belonging to a type. There is no limitation placed on operation types but if the operation is to be performed then its type must be supported by a processor in the structure. Typical operation types are addition, subtraction and multiplication.

3.2.1.2 Signals.

Each directed arc in the data flow graph is a unique signal. There are four classes of signal: input, output, local and constant. The classification denotes a signal's usage within the data flow graph as depicted in Figure 3.4. Signals are assigned names to identify the data which they convey. Signals originating from the same operation convey the same data and so possess the same data name. Similarly, constant signals carrying the same data value will have the same data name.

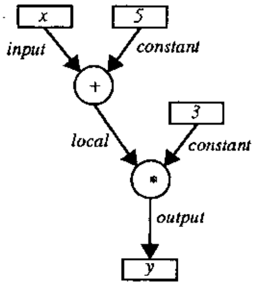


Figure 3.4 Signal classifications within a data flow graph.

3.2.1.3 Textual Representation.

The behaviour is specified in a textual format from which the data flow graph is constructed. An example is shown in Figure 3.5. The signals and operations are specified separately within the behaviour's text file. The syntax for signals is of the form:

```
signal     <data name>       <signal class>    end
```

For operations, which are specified after the signals, the syntax is as follows:

```
operation <operation name> <operation type>
          <left data name> <right data name> <output data name>   end
```

Textual Representation

```
network DIFF_EQ

signal  U    input      end
signal  X    input      end
signal  U1   output     end
signal  dx   constant   end
signal  c3   constant   end
signal  c5   constant   end
signal  s1   local      end
signal  s2   local      end
signal  s3   local      end
signal  s4   local      end
signal  s5   local      end
signal  s6   local      end

operation  mul_1 mul   U    dx   s1   end
operation  mul_2 mul   c5   X    s2   end
operation  mul_3 mul   s1   s2   s3   end
operation  mul_4 mul   c3   Y    s4   end
operation  sub_5 sub   U    s3   s5   end
operation  mul_6 mul   dx   s4   s6   end
operation  sub_7 sub   s5   s6   U1   end

end DIFF_EQ
```

Data Flow Graph

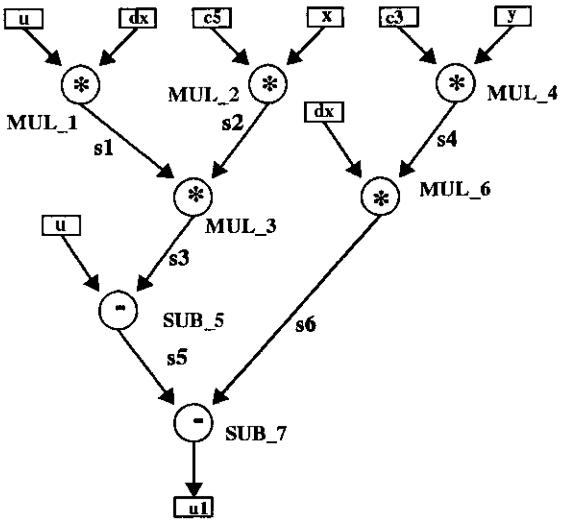


Figure 3.5 Construction of a data flow graph from the textual representation.

3.2.2 Data Flow Terminology.

A number of terms are used to describe relations between operations in a data flow graph. The directed arcs in the graph represent data dependencies between operations. An operation which is the source of an arc must be performed before the destination operation because data must be created before it can be consumed. The arcs can, therefore, be seen to impose an ordering on the execution of operations. The order imposed by data arcs is more clearly expressed when time is represented by discrete steps; operations must occur entirely within individual steps and signals must traverse at least one step boundary indicating that they are flowing forward in time. A simple schedule based solely on data dependencies results from this representation.

Two extreme forms of such a schedule, both requiring the same number of steps, can be constructed for any data flow graph. These are called the As- Soon-As-

Possible (*ASAP*) schedule and the As-Late-As-Possible (*ALAP*) schedule. In the *ASAP* schedule operations are placed in the earliest possible step subject to data flow constraints and in the *ALAP* schedule operations are placed in the latest possible step without exceeding the total number of steps required in the *ASAP* schedule. The *ASAP* and *ALAP* schedules of a small data flow graph are shown in Figure 3.6.

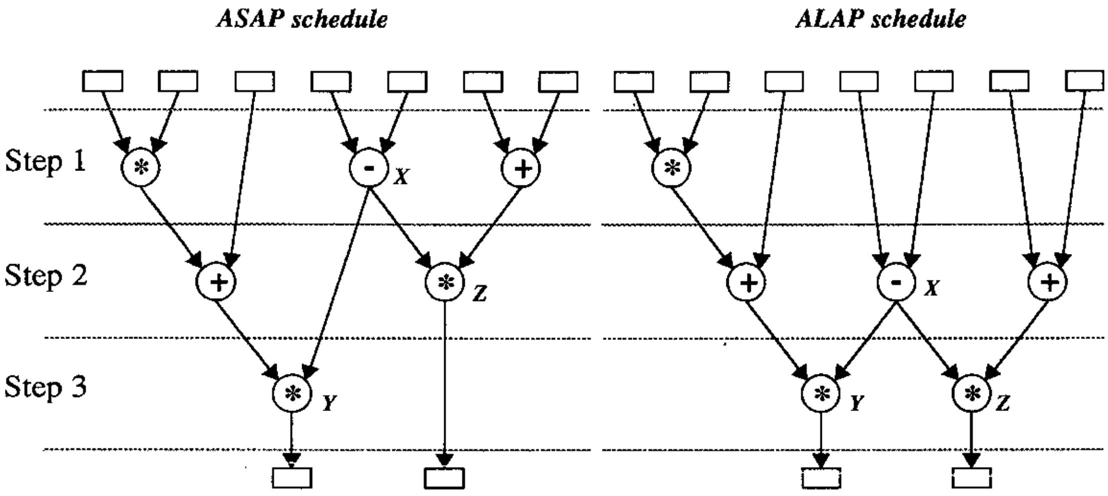


Figure 3.6 ASAP and ALAP schedules.

Consider the ASAP schedule in Figure 3.6. The operations in each step can be thought of as separate ***generations***. Operations in the first generation give rise to those in the second generation and so on. More specifically, operation *X* is termed a ***predecessor*** of operations *Y* and Z. Conversely, *Y* and *Z* are ***successors*** of *X*.

3.3 Structural Model.

Structure is specified at the register-transfer level. It is defined in terms of the interconnection of four basic components types. These are processors, memory, communications and input-output ports. The interconnection of these components is constrained to the syntactic form shown below in Figure 3.7.

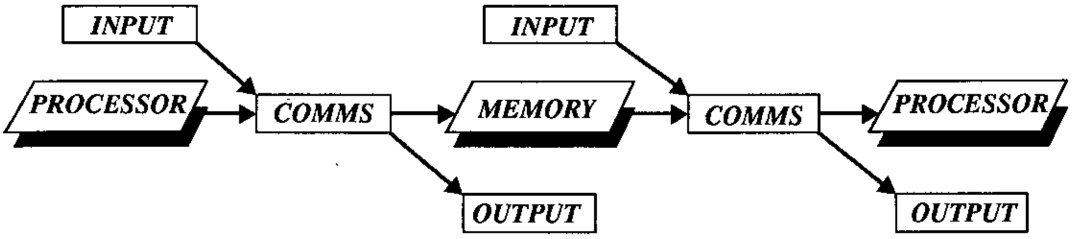


Figure 3.7 Interconnection constraints of structural components.

Conditional constructs cannot be represented in the behavioural model. Therefore, control components are omitted from the structural model as there will be no interaction between the data path and controller. The control sequence can have no dependence on actual data values and, consequently, the structure is purely a data path representation.

The structure is represented in a graphical format. All component types are denoted by vertices and directed arcs represent connections between components identifying the permitted direction of data transfer. Figure 3.8 illustrates how a register-transfer schematic is represented as a structure graph.

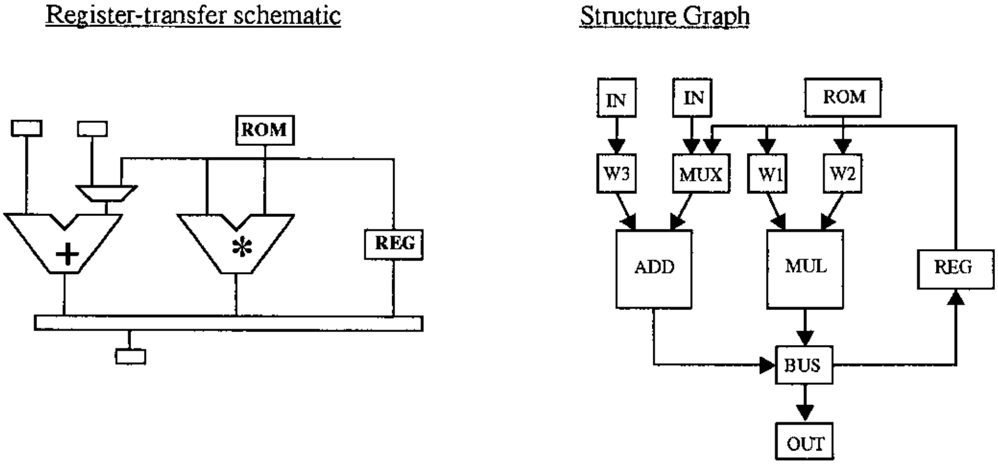


Figure 3.8 Structure graph representation of a register-transfer structure.

3.3.1 Structural Components.

In this system, the components are generic and comply with the timing model discussed in Section 3.4. The specifications have been kept as simple as possible in an effort to simplify the synthesis algorithms.

3.3.1.1 Processing Components.

Processors are combinatorial devices capable of performing operations of specified types. They have two input ports, labelled left and right, and a single output port. It is important that operands are passed to the correct ports for non-commutative operations.

- *Textual Representation.*

The syntax for specifying a processor is as follows:

```
processor     <name>
type          <processor type>
functions     <operation type list>
adapt         <true/false>
ports         left      <port name>     from     <net name list>
              right     <port name>     from     <net name list>
              out       <port name>     to       <net name list>
allocation    <operation name list>
```

The *processor type* identifier is used to access performance characteristics such as execution and re-use times. The *functions* are types of operations which can be implemented by the processor. Within a list, entries are separated by commas. The list is terminated by a semi-colon. If *adapt* is set false then no changes will be made to the processor's port connections. Each processor port is assigned a unique identifier to distinguish connections to the component. These ports can only be connected to communications components as indicated in Figure 3.7. The *allocation* list of operations enables the designer to specify a binding prior to scheduling and data path allocation. Any allocation specified before scheduling does not automatically preclude the binding of other operations to the processor and it will not violate the validity of the schedule.

3.3.1.2 Memory Components.

Four different types of memory component are supported. These are registers, register files, RAM and ROM. Each is assumed to function in the same way. Memory components have separate and single input and output ports. In the case of ROMs the input port is ignored. Data is written to and read from memory in separate phases of a single control step. It is possible for an individual location to be read from and then written to within the same control step.

- *Textual Representation.*

```
memory        <name>
type          <memory type>
adapt         <true/false>
capacity      <integer>
ports
              in       <port name>    from    <net name list>
              out      <port name>    to      <net name list>
allocation    <data name list>
```

The number of storage locations in the component is entered as its capacity. Data to be stored in the component can be entered as an allocation.

3.3.1.3 Communication Components.

These components are collectively termed *nets*. Wires, multiplexers and buses are recognised communications components. Wires have single sources and destinations; multiplexers have multiple sources and single destinations; and buses have multiple sources and multiple destinations.

- *Textual Representation.*

```
net           <name>
type          <net type>
adapt         <true/false>
from          <port name list>
to            <port name list>
```

3.3.1.4 Input and Output Ports.

Input and output ports are the points of external connection and, hence, external communication for structures. They connect to either processors or memory via *nets*.

- *Textual Representation.*

```
io_port         <name>
type            <io_port type>
adapt           <true/false>
from            <net name list>
to              <net name list>
allocation      <data name list>
```

3.3.2 Specifying Structures.

The structure graph is constructed from the textual specification of individual components. An example is shown in Figure 3.9. Components of the schematic are labelled with their names from the specification.

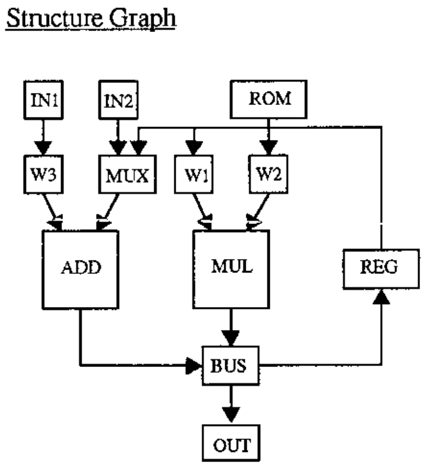
```
processor  ADD
type       ADD
adapt      FALSE
functions  ADD;
ports      right  a0     from MUX;
           left   a1     from WIRE_3;
           out    a2     to   BUS;
allocation;

processor  MUL
type       MUL
adapt      FALSE
functions  MUL;
ports      right  m0     from WIRE_1;
           left   m1     from WIRE_2;
           out    m2     to   BUS;
allocation;

memory     REG
type       REG
adapt FALSE
capacity   1
ports      in     r0     from BUS;
           out    r1     to   WIRE_1,MUX;
allocation;

memory     ROM
type       REG
adapt FALSE
capacity   1
ports      in     c0     from ;
           out    c1     to   WIRE_2;
allocation;

net        BUS
type       BUS
adapt FALSE
from       a2, m2;
to         r0, OUT;
```



```
net        WIRE_1
type       WIRE
adapt      FALSE
from       r1;
to         m1;

net        WIRE_2
type       WIRE
adapt      FALSE
from       c1;
to         m0;

net        WIRE_3
type       WIRE
adapt      FALSE
from       IN_1;
to         a1;
```

```
io_port    IN_1
type       INPUT
adapt      FALSE
from;
to         WIRE_3;
allocation;

io_port    IN_2
type       INPUT
adapt      FALSE
from;
to         MUX;
allocation;

io_port    OUT
type       OUTPUT
adapt      FALSE
from       BUS;
to;
allocation;
```

Figure 3.9 Construction of a structure graph from a textual representation.

3.4 Control Model.

The control model relates behaviour, structure and time. It contains all the information needed to construct an instruction sequence for a controller. The control model is constructed around a basic timing model which defines the operation of the structure.

3.4.1 Timing Model.

The fundamental action of the structure is the execution of an operation. This action can be performed in two parts: the read phase and the write phase. During the read and write phases signals are retrieved from memory or input ports and transferred to the inputs of processors, operated on and then transferred from the outputs of processors and deposited at output ports or in memory. The purpose of the timing model is to enable these events to be sequenced correctly by the controller. The problem can be simplified by viewing the structure as a sequential machine consisting of a combinatorial network and memory connected as shown in Figure 3.10.

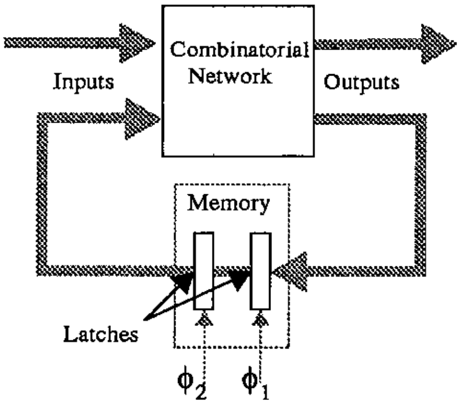


Figure 3.10 Sequential machine representation.

As the combinatorial network represents the structure's processors, the basic operation of the machine is essentially the execution of a behavioural operation as described earlier. Each machine state, when the combinatorial network has stable inputs and outputs, represents the execution of an operation. The timing of the feedback of data through memory must ensure that machine states do not overlap. Timing problems will occur if an output can be fedback.as an input within the same state with the possible effect of corrupting the output data. To ensure that this cannot occur the memory operates in a master-slave fashion, constructed from two independently controlled latches as shown in Figure 3.10. This divides the operation of the machine into two separate time slots or phases.

- *Write phase: outputs from the network are stored in the master latch. The slave latch maintains stable data at the input of the combinatorial network during this time.*

- *Read phase: data is transferred from the master latch to the slave latch, changing the input data to the combinatorial network.*

The clock used to sequence the machine must have two non-overlapping phases, ϕ_1 and ϕ_2. An overlap between ϕ_1(high) and ϕ_2(high) will cause simultaneous transparent mode in both master and slave latches causing unwanted asynchronous feedback where outputs are returned as inputs within the same state. As shown in Figure 3.11 the read phase occurs between ϕ_2(high) and ϕ_1(high) and the write phase between ϕ_1(high) and ϕ_2(high). The maximum speed of circuit operation is set as the minimum combined duration of read and write times. Both these times must allow for maximum propagation times through structural components.

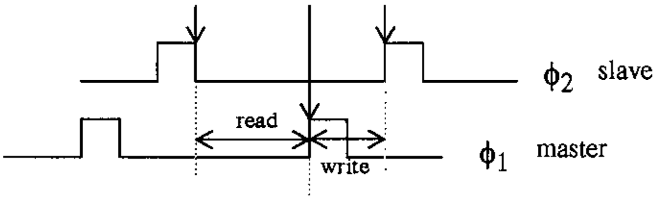


Figure 3.11 Timing of read and write phases.

The propagation delay through different processors may vary considerably. A multiplier is likely to take much longer to execute an operation than an adder. However, this does not mean that the write cycle time must be set to accommodate the slowest processor. It is acceptable to delay the sampling of a processor's output until a later write cycle. For this reason the execution time of processor's is specified as a whole number of control steps. Pipelined processors can also be modelled. In Figure 3.12 the processor has been divided into stages by registers comprising of master and slave latches. Consequently each stage is independent and operates in a separate state. The latches within the processor are controlled by the same clock as the external memory and so the sequential machine shown in Figure 3.12 is simply an expanded form of the one in Figure 3.10.

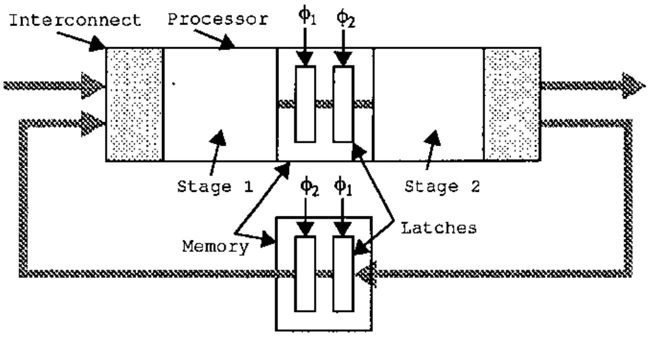


Figure 3.12 Representing a pipelined processor.

Pipelined processors are characterised by two numbers: execution time and re-use time. These are illustrated in Figure 3.13. The re-use time is the minimum number of control steps which must separate successive inputs.

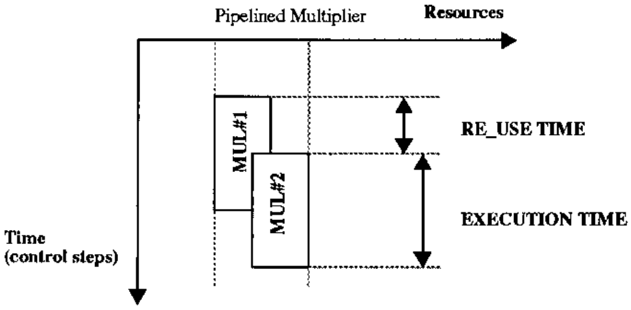


Figure 3.13 Execution and re-use times for a pipelined processor.

3.4.2 Binding Model.

The binding model expresses the mapping between behavioural elements and structural components. Figure 3.14 illustrates how behaviour is mapped to structure. Behaviour is composed of operations and signals.

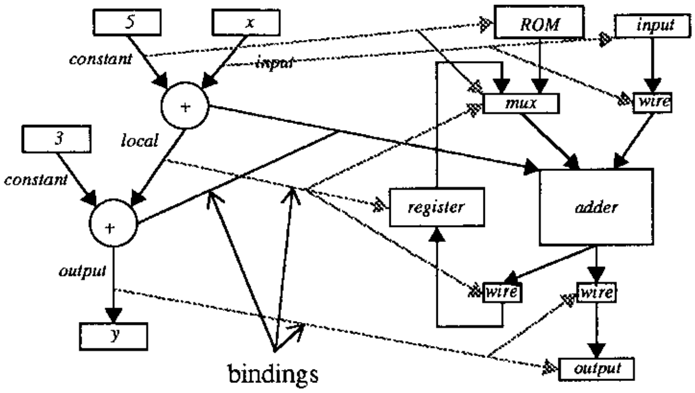


Figure 3.14 Binding behavioural elements to structural components.

By definition, operations can only be performed by processors. If a processor is to perform a particular operation then that operation is said to be bound to the processor and so a mapping is created from the operation to the processor. Signals transfer data between operations and so, as operations are bound to processors, signals must be bound to paths between processors. These data paths are made up of memory, communication and input-output components. Signals are, therefore, bound to these components types.

3.4.3 Representing Control.

Control is represented by a mapping of behavioural elements to both control steps and structural components, also termed resources. The mapping is expressed as a two-dimensional array indexed by resources and control steps, called a resource-time graph.

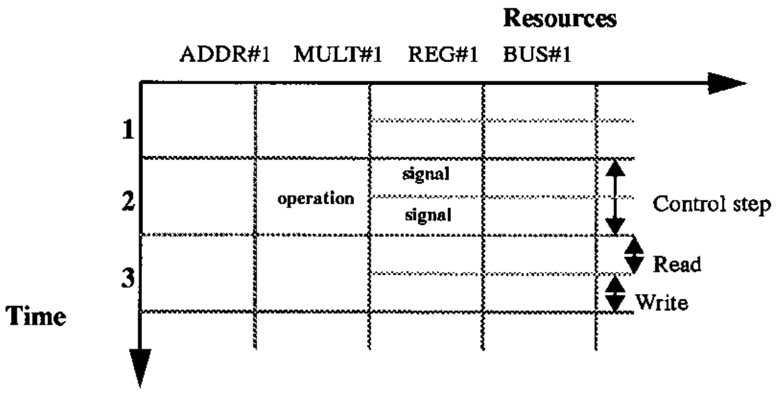


Figure 3.15 Resource-time graph representation.

Behavioural elements are entered at appropriate locations within the array to signify a mapping to a particular control step and resource. In the resource-time graph, as shown in Figure 3.15, the control steps are divided into read and write phases for data path resources. Signals are then mapped into the appropriate phase.

3.5 System Overview.

The adopted approach to synthesis using structural input can be divided into three tasks, mapping, scheduling and data path allocation. The structure of the CAD synthesis system reflects this partitioning: the tasks are performed by separate software modules named mapper, scheduler and data path allocator as shown in Figure 3.16.

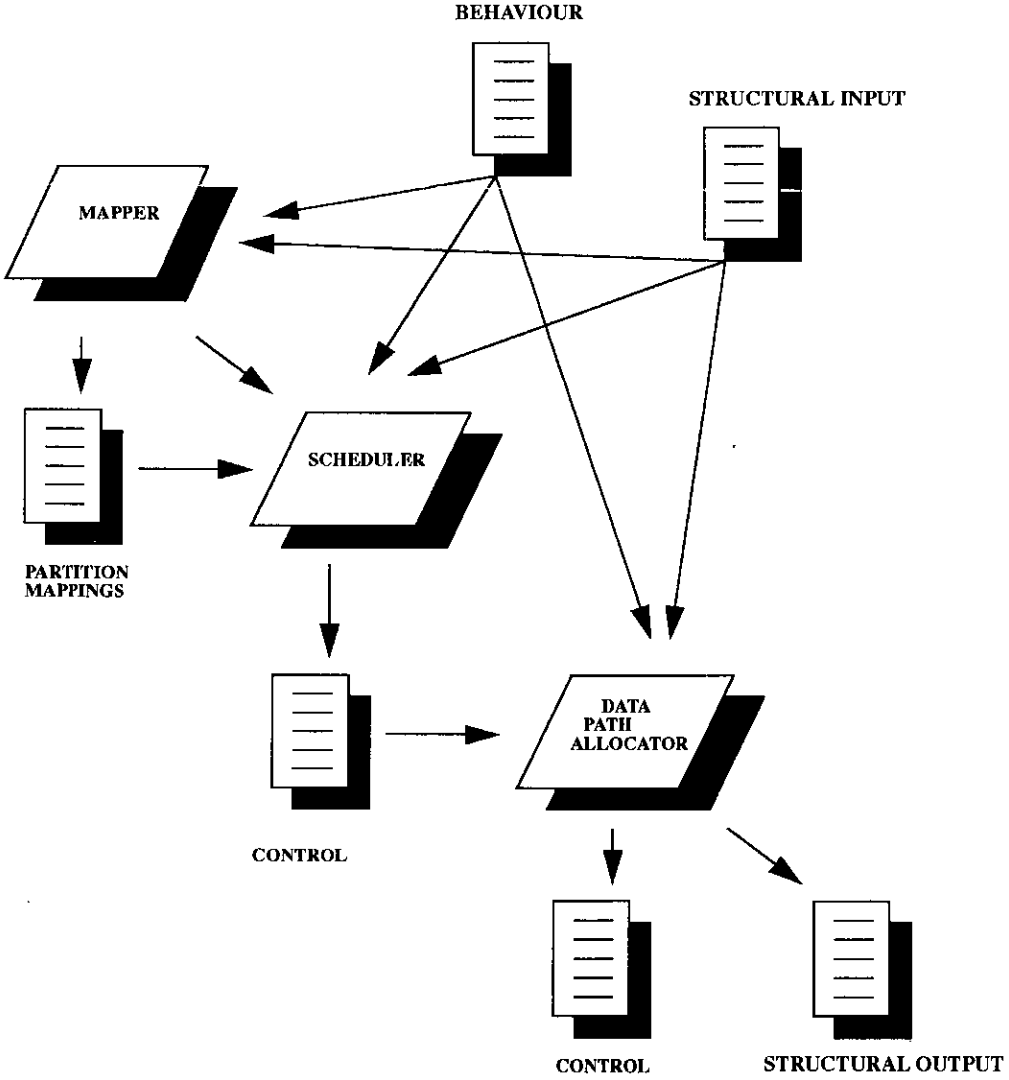


Figure 3.16 The Synthesis System.

The following three chapters are concerned with the algorithms used to perform these synthesis tasks whereas this section concerns itself with the interfaces between these modules and, consequently, the ways in which the designer can use them to direct the synthesis process.

3.5.1 The Mapper.

The mapper accesses both the behavioural and structural models. It identifies potential mappings between partitions of behaviour and the structure and associates operations from within partitions with processors from the structure. The mapper then selects a compatible set of partitions which will then dictate the mapping to the initial structure. The output from the mapper can be passed to the scheduler via the data model or a text file. All identified partitions are included in the text file allowing the designer to alter the partition selection. Figure 3.17 is an example of such a text file. It consists of lists of operations for each partition. Next to each operation are the processors with which they are associated in the mapping.

```
mapping        <design name>

partition      <number>
selected       <true/false>
operation      <operation name>   processor  <processor name list>
  :
operation      <operation name>   processor  <processor name list>
signals        input       <signal list>     end
               output      <signal list>     end
               local       <signal list>     end
               constant    <signal list>     end

partition      <number>
selected       <true/false>
operation      <operation name>   processor  <processor name list>
  :
operation      <operation name>   processor  <processor name list>
signals        input       <signal list>     end
               :
finish
```

Figure 3.17 Textual representation of partitions.

3.5.2 The Scheduler.

The scheduler performs the tasks of scheduling and binding. It constructs the control model for all operations and for signals connected with the partitions. The information from the selected partitions enables the scheduler to schedule behavioural elements onto interconnected processors. A resource-time graph representation of the control model is output from this module to a text file. The schedule and binding can, therefore, be altered by the designer before data path allocation.

3.5.3 The Data Path Allocator.

This synthesis module operates with an older version of the data model and, consequently, communication with the other modules is done using text files. The data path allocator uses the same behavioural specification as the other two but the structure is specified in a slightly different way. The data path allocator creates additional data path components to accommodate signals outwith the behavioural partitions and updates the control model accordingly.

Chapter 4
Mapping Behaviour to Structure.

4.1 Introduction.

Other systems synthesise a complete structure entirely from a specification of behaviour. A set of goals related to the required performance or size of the structure are specified with the aim of directing the synthesis process towards the creation of a suitable architecture. The system reported in this thesis also synthesises structure to implement a specified behaviour, but does so from an initial specification of structure. That is, if the behavioural synthesis task is represented as a function on behaviour to yield structure: $F(B) \rightarrow S$ then the function of the system presented here can be expressed as: $F'(B, S') \rightarrow S; (S' \subset S)$ .The initial structural specification need not be complete but it must, nevertheless, contain all the processing elements to be permitted in the final design. Separate groups of interconnected processors within the structural specification are referred to as structures. These structures then direct synthesis towards architectures favoured by the designer, augmented only by an optional timing constraint.

The key to implementing this approach is relating the behaviour to the specified structures. The fundamental aim of behavioural synthesis is to constrain the behaviour in time and structural resource: to dictate where and when behavioural operations occur and how data are transferred between them. This involves operations being mapped to processors and scheduled to occur within a given control step. The data are then mapped to interconnect and memory to enable their transfer between processing elements at the scheduled time. In the context of synthesis from existing structure, the important issue is the mapping of operations to processors: if the processors are interconnected then the mapping must ensure that the existing interconnection can accommodate the necessary transfer of data between mapped operations. In section 4.2 a method is described for mapping a behaviour onto a structure. However, the problem cannot be restricted to mapping a single behaviour to a single structure. As the structural specification may contain a number of distinct structures, desirable solutions may be found by apportioning partitions of the behaviour to each structure. This problem is dealt with in section 4.3.

Thus the initial problem can be restated as being that of partitioning the behaviour such that the resulting partitions can be implemented on structures within the structural specification. It should be noted that if no interconnection of processors is present in the structural specification then this synthesis phase is reduced to the simple matchmaking process between operations and processors which occurs in conventional systems.

4.2 Mapping a Data Flow Graph to a Structure.

The term *matchmaking* is used to describe the process of determining which processors are capable of implementing the individual operations of the behaviour. This process is straight-forward when the processors are not interconnected. Each processor is defined as being capable of performing a set of operation types and so any operation of a type belonging to that set can be matched to it. Matchmaking produces a set of *candidate* processors for each operation: it will be one of these *candidates* which is subsequently bound to the operation [66].

At this point attention is focused on mapping a single behaviour to a single structure. The algorithmic approach put forward here does not consider the possibility of parts of the behaviour being realised by the structure. If the structure is incapable of supporting the entire behaviour then the process halts; if it succeeds then candidate processors are mapped to each operation in the behaviour. The method examined in the following section incorporates the approach put forward here in dealing with the problem of implementing partitions of the behaviour on different structures.

Mapping is achieved using two algorithms: the first is an extension of the matchmaking process employed in existing synthesis systems and is called the *matchmaking algorithm*; the second deals with the fact that the set of candidate processors for a given operation is dependent upon both the data dependencies of the operation and the structural connections of the processors and is referred to as the *dependency algorithm.*

4.2.1 The Matchmaking Algorithm.

In addition to matching the type of processor to the type of operation, as in 'adder' to 'addition', the *matchmaking algorithm* deals with some structural issues in order to simplify the *dependency algorithm.* The signal types associated with each operation are checked against the structural connections of prospective processors to weed out obvious mismatches.

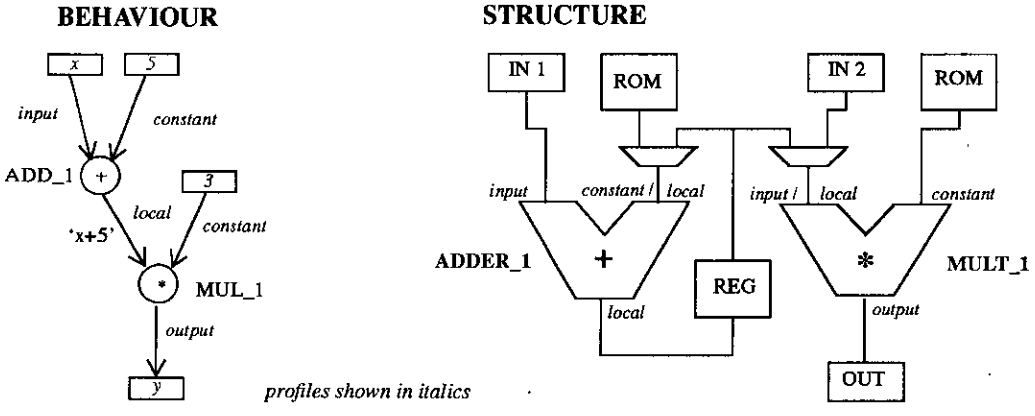


Figure 4.1 Profiles of behavioural signals and structural processor ports.

Referring to Figure 4.1 as an example, only signal *x*, which is of class *input*, can be presented at the left input port of the *ADDER_1*. This is termed signal profile matching. In order that a processor can be matched with an operation, the following conditions must be satisfied: the operation type must be supported by the processor and both input and output signals of the operation must have profiles which are compatible with those of the corresponding ports of the processor. In the figure, *ADDER_1* can be matched to *ADD_1* and *MULT_1* matched to *MUL_1*.

There are four categories of profile: input, output, constant and local. The port profiles of a processor are dependent upon the structures to which the port is connected. A processor input port connected to memory and an external input port, such as the left port of *MULT_1*, would possess both input and local profiles. The profile of a signal is determined by its class within the behaviour. The signal class refers to the transfer of the signal relative to the behaviour: if the signal is an input to the behaviour, such as *x*, then it has the signal class *input* and so gains an input profile.

Matchmaking is applied to the operations in order of the data dependencies. When a set of candidate processors has been drawn up for an operation, it is necessary to 'store' the signal produced by the operation in all the memory units which can be reached from the output ports of the candidate processors. Future operations dependent upon this and other signals must be assigned a set of candidate processors which are all capable of retrieving signals from one of the memory units in which they were stored. In Figure 4.1, matchmaking would be performed for *ADD_1* first. The match with *ADDER_1* will be successful and so the output signal $x+5$ is stored in *REG*. Therefore, when matchmaking is performed for *MUL_1* and *MULT_1*, the local profile match of the left inputs will force *REG* to be checked for signal $x+5$. A signal and a processor port are matched if they have a common profile and if that profile happens to be local then the signal must be present in one of the memory units connected to the processor port.

After matchmaking has been completed each operation will have an associated set of candidate processors. Any operation which has an empty candidate set is removed from the behaviour. The input and output signals of such an operation then acquire output and input profiles respectively. Matchmaking is then repeated for operations connected with these signals in order to update their candidate sets. The process of removing and updating is repeated until no operations in the behaviour have empty candidate sets or there are no operations remaining in the behaviour. In the case of the latter, the structure is completely unsuited to the behaviour and not even partitions of the behaviour can be mapped to it. This removal of operations from the behaviour reduces the computation required in the partitioning algorithm discussed in section 4.3.

4.2.2 The Dependency Algorithm.

The matchmaking process described above yields a behaviour with a reduced set of operations, each associated with a preliminary set of candidate processors. The purpose of the algorithm described here is to ensure that for any mapping of operation to candidate processor there exists a valid mapping of the entire behaviour to the structure. Once an operation has been bound to a single processor, the sets of candidate processors for dependent operations may well be reduced. This, however, will be taken care of within the scheduling phase.

In order to simplify the dependency algorithm, relevant structural information is presented in a functionality graph. This is a representation of the interconnection within the structure. Processors are represented by vertices and each directed arc denotes a data path through memory from the output of a processor to the input of a processor. The information conveyed by an arc indicates which memory unit and processor input port are used. The structure in Figure 4.1 has the functionality graph shown in Figure 4.2.

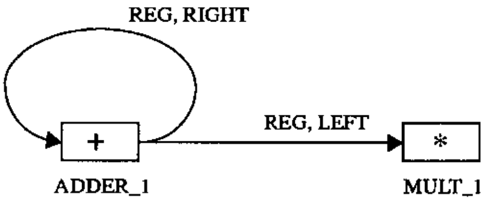


Figure 4.2 The Functionality Graph

The matching of local profiles indicates suitable transfer paths in the structure to support behavioural data transfers. As illustrated in Figure 4.3, it does not, however, ensure that such paths are compatible. The sequence of operations A, B and C can only be implemented by the structure if transfers from $A \longrightarrow B$ and $B \longrightarrow C$ can be found where B is mapped to the same processor. The only data path which will support this sequence is $P \longrightarrow Q \longrightarrow Q$. Therefore, S and R should be removed from the candidate sets of A and B.

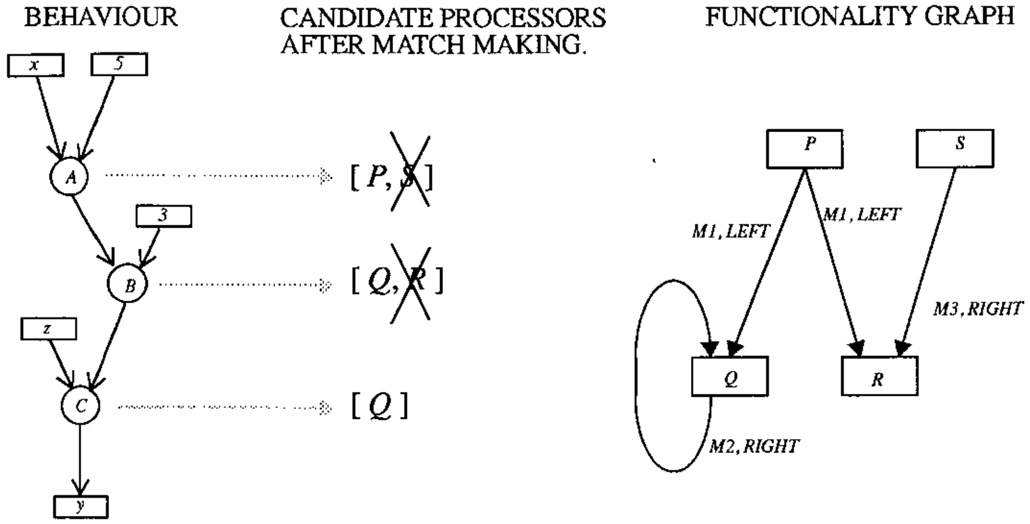


Figure 4.3 The 'Dependency Problem'.

Two functions, *pred* and *succ*, short for 'predecessors' and 'successors', are useful in describing the algorithm. These functions operate on directed graphs such as the data flow and functionality graphs. In Figure 4.3, A is the predecessor of the left input signal of B and the successors of P are Q and R. Hence:

pred(left_signal(B)) = A **succ(P) = (Q,R)**

Extending the example in Figure 4.3 to the general case, it can be stated that the candidate processors of an operation must be accessible within the structure from the candidate processors of operations immediately preceding and succeeding. To ensure this accessibility the candidate processors of operations are updated on two passes. The first pass deals with preceding operations and the second with succeeding operations. On the first pass, the operations, ordered from the second generation of the data flow graph to the last, are processed using equation 4.1. 'Cand_proc(op)' denotes the candidate processors of the operation 'op'.

cand_proc(op):= cand_proc(op) ∩ [succ(cand_proc(pred(left_signal(op)))) ∩ succ(cand_proc(pred(right_signal(op))))] **Equation 5.1**

On the second pass, the operations, ordered from the penultimate to the first, are processed using equation 4.2.

$$\text{cand\_proc(op)} := \text{cand\_proc(op)} \cap [\text{pred(cand\_proc(succ(left\_signal(op))))} \cap \text{pred(cand\_proc(succ(right\_signal(op))))}] \quad \textbf{Equation 5.2}$$

Operations in the first generation need not be processed on the first pass as they have no preceding operations. Similarly, operations in the last generation have no succeeding operations and so are excluded from the second pass.

4.3 Mapping Partitions of a Data Flow Graph to Structures.

Section 4.2 focused on identifying a mapping between a data flow graph representation of behaviour and a single structure. This mapping is conveyed by an association of candidate processors to operations. The limitation in this approach is that the structure must implement the entire behaviour.

The aim of this section is to present a method to partition the behaviour so that its constituent parts can be implemented by different structures. The behaviour is represented by a data flow graph and so partitions will take the form of connected subgraphs. This means that there is a connection via data arcs and operations between every pair of operations within the partition. The mapping algorithms presented in section 4.2 will function with partitions provided that the classification of signals is made relative to the partition. Thus the input and output signals of the partition, some of which may be local to the behaviour as a whole, must be considered as inputs and outputs.

The approach adopted here is to seek out partitions which satisfy the requirements of the matchmaking algorithm and then to apply the dependency algorithm to them. A selection from the surviving partitions is then made in an attempt to find the partitions of behaviour best suited to exploiting the available structures.

4.3.1 Identifying Partitions.

A partition is a collection of operations connected by data dependencies. These dependencies can be used to order the operations into generations[1]. From operations in the first generation, termed seeds because they give rise to the partition's data flow, it is possible to reach every operation in the partition via signals and other operations within it. This arises from the fact that it is a connected graph. This fact is central to the identification process. The remaining problems lie, firstly, in uncovering these seed operations and, secondly, identifying the boundary of the partition. Both are addressed by the matchmaking algorithm which assesses whether the structural requirements of the operation's signals can be met. Operations which lie on the boundary of the partition must map to processors on the boundary of the structure. An output signal from the partition must have access to an external port and, hence, the operation which produces it must have candidate processors which have their outputs connected to such external ports. Recall that a partition possesses its own classification of signals; these will influence the profiles in matchmaking which, in turn, defines the boundary of the partition and, hence, the classification of signals. Nevertheless, the only change in signal classification which occurs is from local to input or, local to output or output local. Therefore, in the search for partitions, additional input and output profiles are assigned to local signals. Thus, matchmaking which involves common input or output profiles indicates a possible partition boundary across the signal in question. Seed operations are identified on a boundary by the fact that neither input signal matches with only a local profile from the structure.

4.3.1.1 Selecting Seeds.

The first step in the partition identification process is identifying seed operations. Matchmaking is performed on the operations of the behaviour to remove operations incompatible with the structure, to list the seeds and to list the data arcs which can be supported as inputs to or outputs from the partition. The seed operations will form the basis of all feasible partitions within the behaviour. The first generation

[1] Refer to section 3.2.2 on page 41.

of the data flow in a partition must be composed of some combination of these seeds. With a total of *n* seeds, this combination could number between 1 and *n* different seeds. The number of possible choices of *r* objects from *n* is given by: $\frac{n!}{n!\,(n-r)\,!}$ Hence, the total number of combinations of seeds is a summation of the form given below:

$$\sum_{r=1}^{n} \frac{n!}{r!\,(n-r)\,!} = 2^{n} - 1$$ **Equation 5.3**

A method for computing all combinations of *n* objects can be found by representing each object as the digit of an n digit binary number. In letting the digit '1' correspond to the inclusion of an object and '0' the exclusion, it is possible to derive all 2^{n}-1 combinations by incrementing the *n* digit number from 1 to 2^{n}-1 with evaluation of a combination at each step.

4.3.1.2 Searching for Partitions.

A partition is identified by tracing the data flow from a combination of seeds until data arcs are encountered which can be supported structurally as inputs to or outputs from the partition. These data arcs are identified during the matchmaking process as a result of the matching of operations to processors connected to external ports.

The search for a partition starts as a subgraph containing the selected combination of seed operations. The external arcs of this subgraph, with the exception of the input arcs to the seeds, are processed in turn. Arcs directed out of the partition are dealt with first. If they cannot be supported as outputs by the structure then the operations which they feed are added to the subgraph. In the case of arcs directed into the subgraph, if they cannot be supported as inputs the operations which produce them are incorporated into the subgraph. In this way the subgraph expands until one of two possibilities occurs: the subgraph attempts to exceed the boundary of the behaviour, indicating that no partition exists for that particular combination of seeds, or that all external arcs of the subgraph can be supported by the structure. The latter indicates a possible partition. However, the subgraph must be found to be *connected* before it is passed to the dependency algorithm. Furthermore, it may be possible to expand the subgraph to reveal other partitions based on the same set of seeds.

A number of the external arcs of the subgraph may not be restricted to serving as inputs and outputs. It may be possible that they can be accommodated within the subgraph by including additional operations. Such external arcs can be identified by the fact that in the matchmaking process they formed local profile matches as well as input or output profile matches. In order to find all possible extensions of the subgraph, all combinations of these arcs must be investigated separately. This is the same process that was applied to the selection of seed combinations and so there will be 2^n-*1* combinations of *n* arcs.

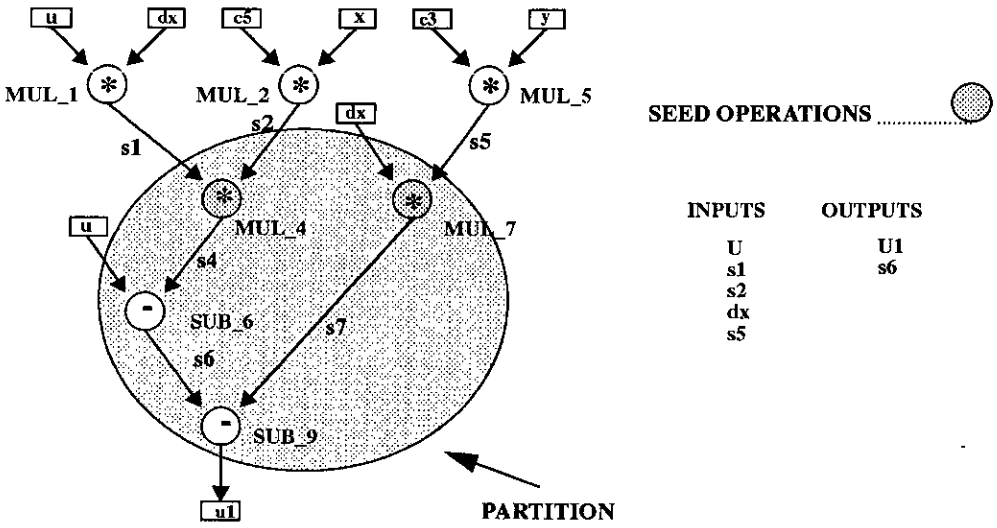


Figure 4.4 Identifying partitions.

A small example of a partition search is presented in Figure 4.4. The seed operations are *MUL_4* and *MUL_7*. Possible partition input and output signals, produced by the matchmaking algorithm, are listed in the figure. The search starts by investigating *s4* and *s7*. Both data arcs must be accommodated within the subgraph as neither appear in the output list. The inclusion of operation *SUB_9* is prompted by *s7*. Although *U1* can be supported as an output, *s6* becomes an unresolved input arc. The addition of *SUB_6* is forced by *s4* and this takes care of *s6* as it is now supported internally. As *U* is a valid input arc the search is complete. The subgraph highlighted is connected and so it can be passed to the dependency algorithm. Note that without the inclusion of both *SUB_6* and *SUB_9* the subgraph would not be connected.

4.3.2 Partition Selection.

The selection of partitions is constrained by the fact that no two selected partitions may contain the same operation. The partitions which have been identified could be associated with any of the structures present in the initial specification. These structural associations will influence the execution time but, without actually scheduling, it is only possible to predict the minimum execution time using various structures. The selection task attempts to select partitions which, while encompassing as many of the behaviour's operations as possible, map to structures with sufficient processing power to meet the timing constraint.

4.3.2.1 Timing Estimates.

The user imposed timing constraint, $\mathbf{T}_{\mathrm{usr}}$, is the maximum number of control steps to be used in scheduling the data flow graph. If the number of operations in the data flow graph, $\mathbf{N}_{\mathrm{ops}}$, is divided by $\mathbf{T}_{\mathrm{usr}}$ then the result is the average concurrency of operations in each control step. Accordingly, if the timing constraint is to be met, the average operation concurrency of each partition must be greater than or equal to this number.

$$\sum_{partitions} P_{con} \geq \frac{N_{ops}}{T_{usr}}$$

Equation 5.4

The operation concurrency figure of a partition, $\mathbf{P}_{\mathrm{con}}$, can be calculated by dividing the number of operations in the partition by the minimum number of control steps required by the processors to perform all the operations, assuming the processors operate in parallel and neglecting the data dependencies between operations.

The problem of selecting partitions to maximise the coverage of operations within the behaviour can be tackled using an existing graph theoretical algorithm: clique partitioning[1]. Clique partitioning identifies all cliques within a graph. A clique is defined as a group of vertices where each vertex is connected by an arc to every other vertex in the group. To make use of clique partitioning for partition selection, a graph is constructed in which vertices represent the partitions and arcs indicate mutually exclusive sets of operations for the partitions represented by the adjoining vertices. Hence each clique will form a possible selection as it will contain a maximal set of partitions which have no operations in common. After computing the concurrency factors for each clique a selection weighting can be calculated as shown below:

$$weight = \left\{1 - \left(\frac{N_{clique}}{N_{ops}}\right)\right\} + \left\{\left(\frac{N_{ops}}{T_{usr}}\right) - \left(\sum_{partitions} P_{con}\right)_{clique}\right\}$$

where N_{clique} is number of operations in clique **Equation 5.5**

The clique with the lowest weighting is chosen as the partition selection.

4.4 The Integrated Approach.

A basic outline of the method of mapping behaviour to structure is traced in figure 4.5. All separate structures within the structural specification are identified first. The mapping of partitions of behaviour to each of these structures is then attempted in turn. The behaviour takes the form of a data flow graph and the structure is represented by a functionality graph. Matchmaking is used initially to remove operations which cannot be supported by the types of processor in the structure. In addition, it lists the seed operations for partitions and the signals which can form the inputs to and outputs from partitions.

A search for subgraphs containing a combination of seeds is then performed. If a subgraph is identified then a search is made for larger subgraphs based on the same

[1] A clique partitioning algorithm, coded in Ada, is provided in Appendix A.1 on page 147.

combination of seed operations. This search is based on combinations of input and output signals from the original subgraph which can be supported within a partition. Any identified subgraphs must be found to be connected before being processed further. Matchmaking is applied to each connected subgraph, where signals are classed relative to the subgraph. The dependency algorithm then checks that the interconnection of the structure can support the dependencies of operations and updates the candidate processors of each operation accordingly. If a valid partition is identified then it is stored together with the mapping to candidate processors, awaiting partition selection.

Partition selection occurs after mapping has been attempted with every structure. A set of partitions is selected in an attempt to maximise the number of operations included while favouring partition-structure mappings likely to meet the timing constraint.

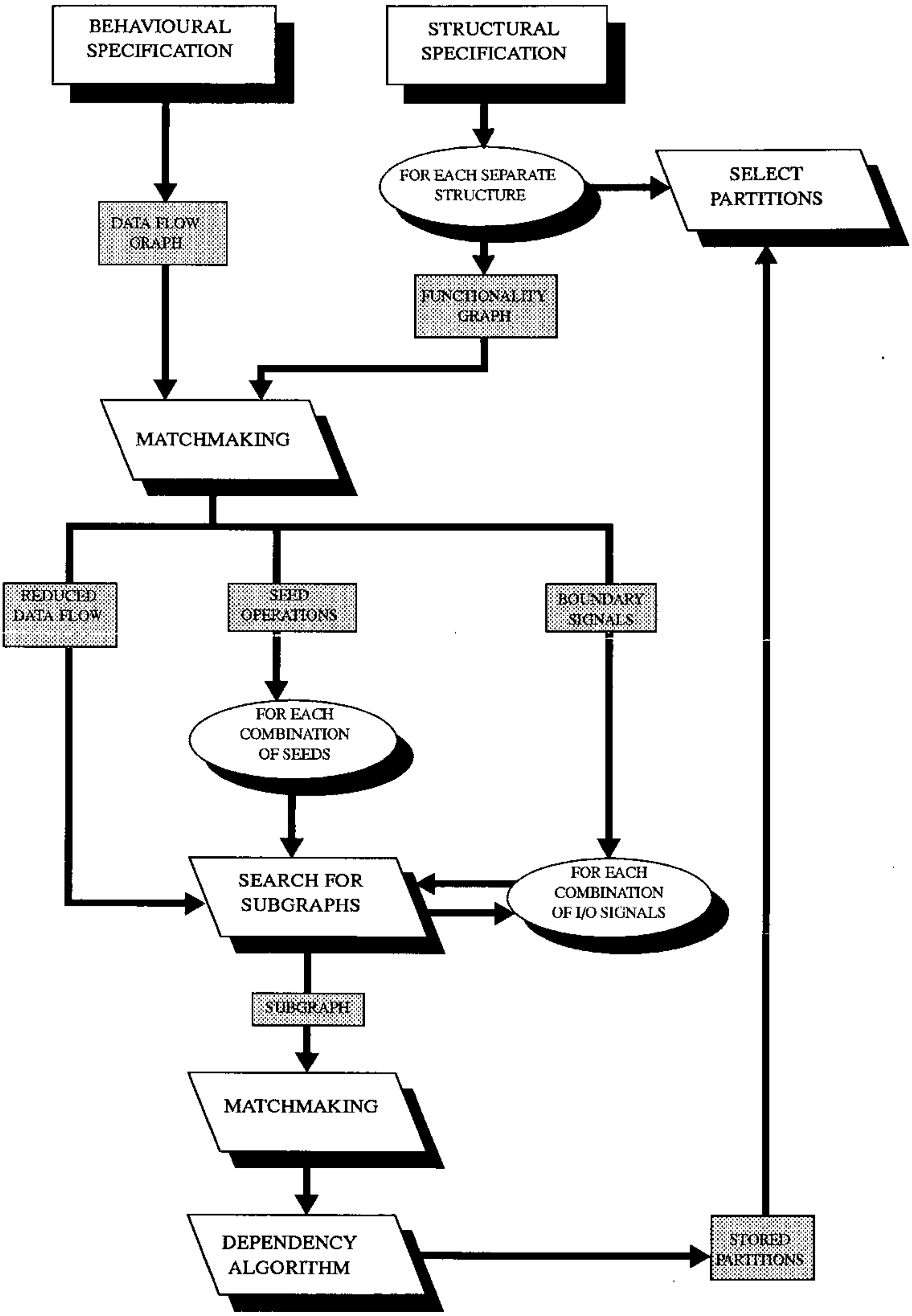


Figure 4.5 The outline of mapping behaviour to structure.

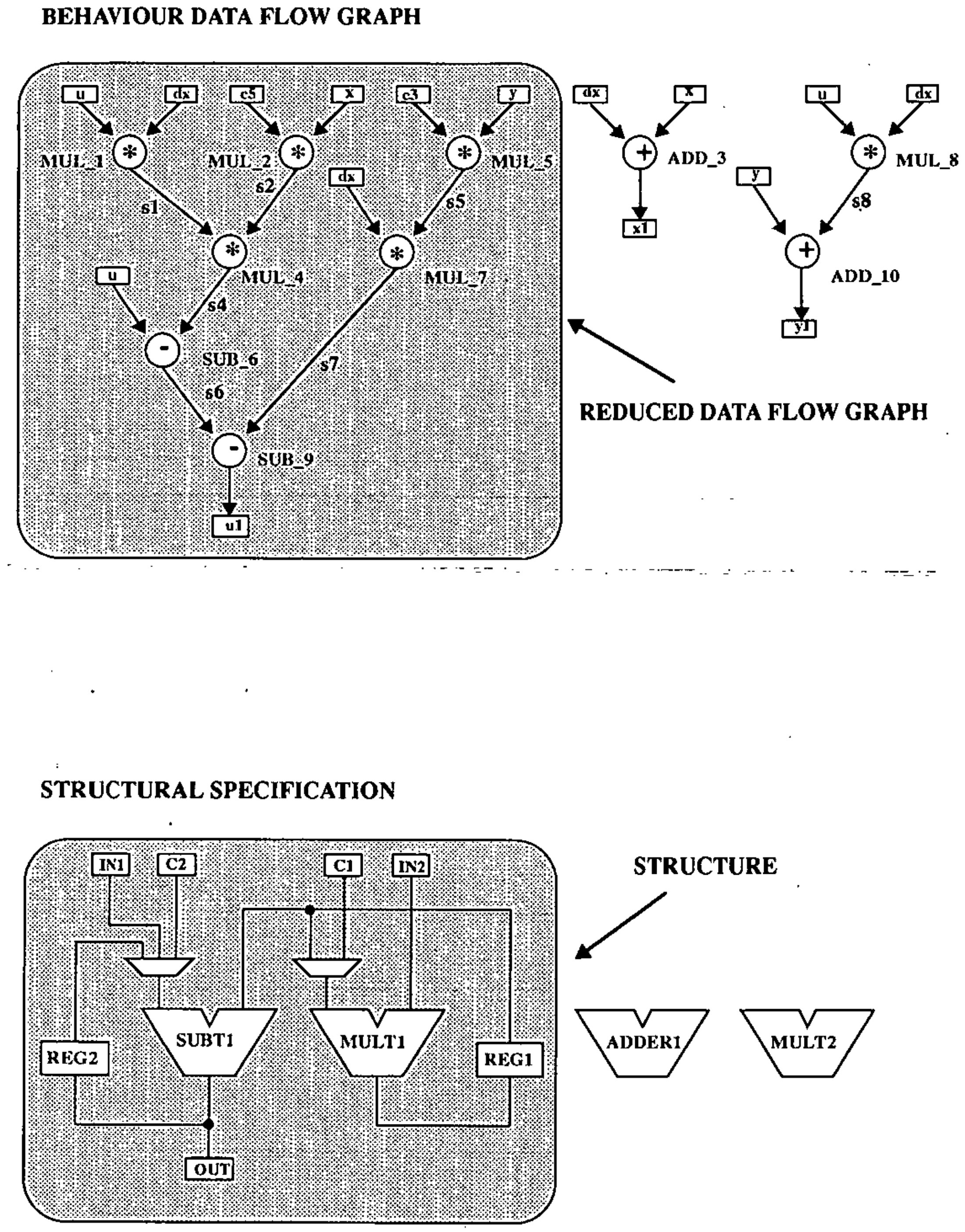
4.4.1 Worked Example.

The following example is used to illustrate the approach outlined in Figure 4.5. The behaviour and structure are presented schematically in Figure 4.6. The behaviour represents part of the solution to a differential equation and is a common benchmark in the synthesis field. It is described fully in Appendix B.2 on page 157. As the subtractions are non-commutative, the order of left and right inputs must be preserved in the mapping to structure in order to maintain correctness. The inputs to the multiplier or adder may, however, be reversed as such operations are commutative.

Only one structure exists within the specification. It contains two processors: a subtractor and a multiplier. The initial matchmaking step removes operations *ADD_3*, *ADD_10* and *MUL_8*. The additions are removed because the structure is unable to support that type of operation. The output signal of the multiply is used in an addition. For this to be possible the signal would have to be output from the structure. However, *MULT_1* does not have access to an external output port and so *MUL_8* must also be removed.

The seeds which are identified are all multiplications: *MUL_1*, *MUL_2*, *MUL_5* and *MUL_7*. As the right input of the subtractor is connected only to local memory, subtractions cannot act as seed operations because the right signal must be generated within the structure and so come from within the partition. For the multiplier to support a seed the operation must have a constant as an input signal. The order of the operation's input signals does not matter as multiplication is commutative. With a total of four seeds, there are fifteen combinations of seed operations which can give rise to partitions [Equation 5.3].

Only five partitions are identified and these are shown in Figure 4.7. *MUL_1* and *MUL_2* cannot both be performed by the structure because *S1* and *S2* would be fed to the same multiplier input preventing *MUL_4*. The selection, comprising of a single partition, is highlighted. The remaining operations will be performed by the unconnected adder and multiplier.



BEHAVIOUR DATA FLOW GRAPH

REDUCED DATA FLOW GRAPH

STRUCTURAL SPECIFICATION

STRUCTURE

Figure 4.6 Behaviour and structure for the worked example.

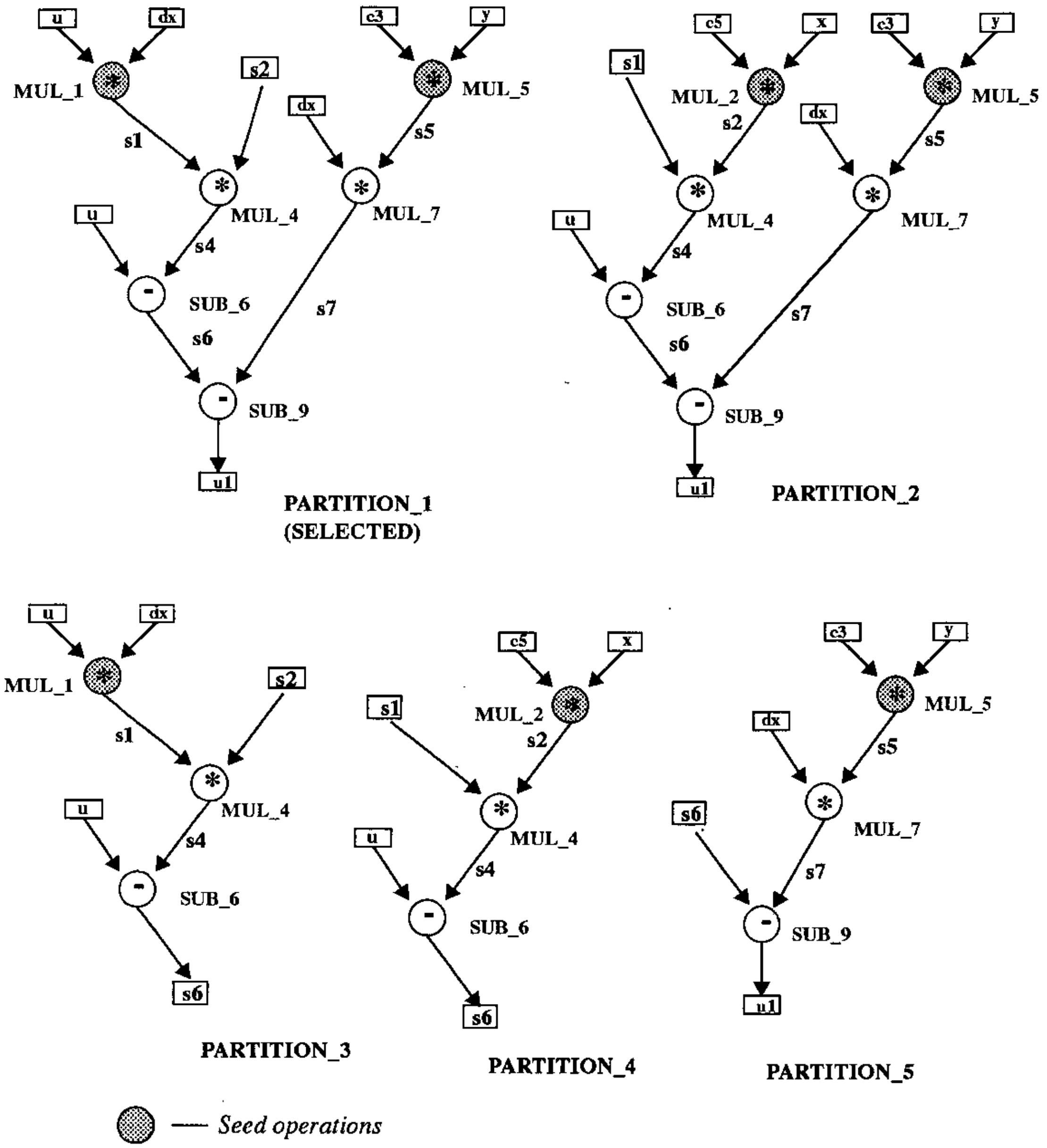


Figure 4.7 Identified partitions.

4.5 Concluding Remarks.

Ways of mapping behaviour to existing structures have not been addressed directly in contemporary research. The algorithms described in this chapter demonstrate an effective way of mapping behaviour to structure so that scheduling and data path allocation can proceed while constrained by existing structures. When the structural input contains no interconnected processors, mapping is unnecessary and so synthesis proceeds in a similar fashion to existing systems.

4.5.1 Computational Complexity.

The overriding problem encountered in mapping behaviour to structure is computational complexity. The complexity is a function of the size of behaviour and degree of non-specialisation of the structure. It is this combination which yields excessive numbers of partitions. Specialisation is composed of several factors: the type of processors in the structure, the inter-processor communications and the I/O connections. Processor types can be excluded as a specialisation factor here because matchmaking eliminates all operations in the behaviour which cannot be supported by processors within a structure. A non-specialisation index can be computed for a structure from the remaining factors.

- *Inter-processor communications.*

In a structure which is completely non-specialised, there will exist a communication path between every processor output and every processor input. The contribution made to the index is the percentage that the actual connections of processor outputs to inputs form of the total required for complete non-specialisation.

- *I/O connections.*

The contribution is the percentage of I/O connections which exist to processors relative to the number required for there to be a connection from every processor input to an input port and output to output port.

The index is calculated as an average of the two percentages. Mapping any behaviour to a structure which has an non-specialisation index of 100% requires only the matchmaking algorithm as structural interconnection will not constrain the binding of operations to processors. Structures with high non-specialisation indices can have a restriction placed on the combinations of seeds investigated in order to reduce computation. Non-specialised structures are typically used in situations where there are no critical performance constraints and so mappings of large partitions will be favoured. Therefore, by investigating only the larger combinations of seeds the probability of identifying the larger partitions will remain high while the computation is reduced.

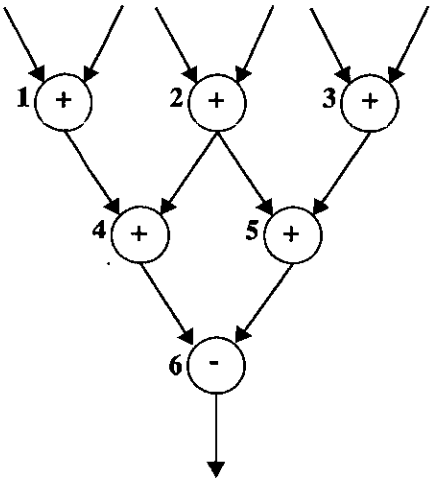
4.5.2 Memory Considerations.

A mapping between behavioural and structural specifications has been realised where the relationship is between operations and processors. Memory has not been used as a constraining factor in this mapping process. The reason for this lies with the fact that no timing information exists before scheduling and so it is not possible to determine whether memory size has been exceeded or there are clashes in access. Clashes in memory access are identified in scheduling and during data path allocation it is determined whether memory size has been exceeded. In both cases memory can act as a constraining factor although it acts later in the synthesis process and so has less influence on the architecture.

Chapter 5
Scheduling and Binding.

5.1 Introduction.

This synthesis task is concerned with scheduling data flow operations into control steps and binding them onto processors; it constrains the behaviour in time and in place. The outcome of the whole process can be represented on a resource-time graph. As its name suggests, the resource-time graph is a two-dimensional array indexed by control steps (time) on one axis and structural resources on the other. Operations occupy slots corresponding to their scheduled control step and processor binding. An example is shown in Figure 5.1.



Resource →

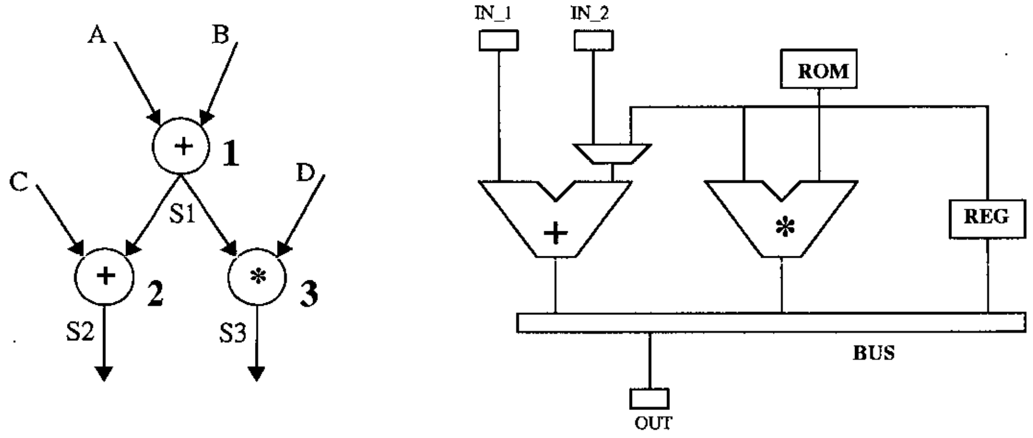
Time ↓	ADD#1	ADD#2	SUB#1
1	1	2	
2	4	3	
3		5	
4			6

Figure 5.1 Resource-time graph representation for operations and processors.

Scheduling is a process which is constrained by both behavioural and structural factors: these are data dependencies within the data flow graph and the number and type of available resources. In Figure 5.1 data dependencies dictate a minimum schedule of three steps but the resource set forces a schedule of four steps. An extra adder would be required to complete the schedule in three steps.

The aim of scheduling is to minimise the number of control steps needed to carry out the behaviour under the given allocation of resources. The full processor allocation is defined in the structural input.

The approach to synthesis using structural input introduced in Chapter 4 presents some additional problems to the scheduling task. Scheduling must accommodate the tentative mapping $B' \rightarrow S'$ which is expressed as a binding constraint between certain operations in the behaviour, $operations \in B'$, and processors in the structural input, $processors \in S'$. To preserve the correctness of the mapping $B' \rightarrow S'$, other structural components of S' must be considered to ensure that no usage clashes occur. Refer to the example presented in Figure 5.2.



Resource →

Time ↓

Time	ADD#1	MUL#1	REG#1	BUS#1	OUT#1	IN#1	IN#2	ROM#1	
1	1								READ
			S1	S1		A	B		WRITE
2	2		S1			C			
3		3	S1	S2	S2			D	
				S3	S3				

Figure 5.2 Scheduling with interconnection.

As a result of the interconnection of the processors, the bus and register cause a bottleneck forcing the schedule to take an extra step. Such problems are tackled by introducing memory components, IO ports and buses as resources in the resource time graph. These data path components are assigned signals during the binding phase while processors are assigned operations. As the resource-time graph will only permit a single operation or signal in any slot usage clashes are prevented.

5.2 Scheduling.

The scheduling approach adopted here is based on the *force-directed list* scheduling (FDLS) method devised by Paulin et al. [21]. It is based on the well established list scheduling method which is the simplest and most obvious way to schedule operations from a data flow graph onto a fixed processor resource set. List scheduling is a constructive method; it deals with control steps in chronological order. For each control step a list of candidate operations is constructed. These are unscheduled operations whose input signals have been made available in previous control steps. If there are more candidate operations of any type than available processors which can perform them then deferral must occur. This is when one or more of the operations is removed from the list without being scheduled. The candidate list drawn up for the next control step will therefore contain these deferred operations. Deferral causes the scheduling of operations to be delayed until suitable resources become available.

Choosing which operations to defer is particularly important: it can affect both the overall schedule length and the balance of operation concurrency.

5.2.1 The Force-Directed Deferral Mechanism.

The *force-directed* deferral mechanism is invoked whenever the number of candidate operations of a given type exceeds the number of suitable processors available. Forces are calculated for these operations with the aim of deferring the operation with the lowest force until all the remaining operations can be accommodated by the available processors. Force can be thought of as a measure of the attraction of an operation to the current control step or as a measure of how much it will resist deferral.

5.2.1.1 Time Frames.

ASAP and ALAP schedules are constructed. As described in Chapter 3, these schedules take no account of resource constraints but they do observe the effects of scheduling and deferral. If an operation is scheduled into a control step then the ASAP and ALAP schedules will also place the operation in that step. Similarly, if an operation is deferred from a step then the ASAP schedule will also exclude it from that step. From these schedules the time frame of each operation is calculated. The time frame is the stretch of control steps bounded by the ASAP and ALAP schedule times. Operations on the critical path will have a time frame of a single step indicating that there is no freedom for scheduling unless the overall timing constraint is relaxed. If a critical path operation is deferred then the ASAP and ALAP schedules are extended.

5.2.1.2 Distribution Graphs.

Distribution graphs are used to indicate the concurrency of operation types in control steps. A separate distribution graph is constructed for each operation type. For each control step of a graph, the probability of each operation of the type occurring is summed and entered as the distribution value. The probability of an operation occurring in a particular control step is defined in Equation 5.1.

$$P(op, i) = \begin{cases} \dfrac{1}{alap - asap + 1} & asap \leq i \leq alap \\ 0 & \end{cases}$$

Equation 5.1

It is assumed that the operation has an equal probability of occurring between its ASAP and ALAP schedule steps. The distribution value of a particular step in a graph is expressed by Equation 5.2.

$$DG(i) = \sum_{op \in type} P(op, i)$$

Equation 5.2

When multi-function processors, such as ALUs, are present in the structural allocation, the distribution graphs of the operation types supported by the multi-function processors are merged forming a multi-class distribution graph.

5.2.1.3 Force Calculation.

The force on an operation associated with the reduction of its initial time frame, bounded by steps *t* and *b*, to a new time frame bounded by steps *nt* and *nb* is expressed in Equation 5.3.

$$Force = \sum_{i=nt}^{nb}\left[\frac{DG(i)}{(nb-nt+1)}\right]-\sum_{i=t}^{b}\left[\frac{DG(i)}{(b-t+1)}\right] \qquad \textbf{Equation 5.3}$$

Each summation term is an average of the distribution graph over the time frame. The force is, therefore, the difference in the average distribution resulting from the change in the operation's time frame.

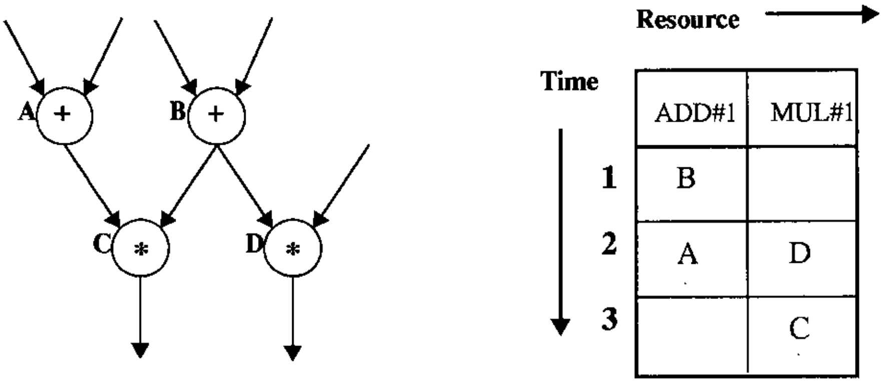
Paulin improved the effectiveness of the *force-directed* algorithm by incorporating a look-ahead measure in the force calculation. The distribution values in the new time frame term of Equation 5.3 are modified to take on a value between the current one and the value which would be obtained after the change in the operation's time frame. The force calculation with look-ahead modification is given in Equation 5.4.

$$Force = \sum_{i=nt}^{nb}\left[\frac{DGi}{nb-nt+1}+\frac{x(i)}{3}\right]-\sum_{i=t}^{b}\left[\frac{DG(i)}{(b-t+1)}\right]$$

$$x(i) = \left[1-\left(\frac{nb-nt+1}{alap-asap+1}\right)\right] \qquad \textbf{Equation 5.4}$$

Scheduling an operation reduces its time frame to a single step. This may, in turn, have an effect on the time frames of successor operations. If time frames of other operations are affected then the 'successor forces' generated from these operations are added to the initial force presenting a more global reaction to the potential scheduling. The effect of successor forces is illustrated in Figure 5.3. One of the additions must be deferred because there is only one adder. Both operations have the same direct force. However, it is obviously better to defer A as neither multiplication can be attempted before B is performed. This is taken into account by successor forces: B receives a contribution from both multiplications whereas A receives only one force contribution.

Consequently, B has the higher force and so A is deferred.



Time	ADD#1	MUL#1
1	B	
2	A	D
3		C

Figure 5.3 Successor forces on operation deferral.

5.3 Binding.

Binding is the process by which behavioural elements are mapped to structural components: operations are bound to processors and signals are bound to data path components such as memory, buses and I/O ports. It is closely linked to allocation, which is the provision of structural components for binding. Synthesis using structural input removes much of the need for allocation within the synthesis system. Only a proportion of data path components will ever be allocated in a design. The task of processor allocation does not interact with scheduling; all permitted processors are incorporated into the structural specification. Much of the task of allocation is therefore thrust upon the designer giving him greater control over the search of the solution space.

Within scheduling it is necessary to determine if there exists a feasible binding between a set of operations to be scheduled concurrently and a set of available processors. This task is dealt with entirely within the scheduling algorithm. It is achieved by type matching: if the type of operation is supported by an available processor then a feasible binding exists. The aspect of binding which will receive attention in this section is how to bind operations to processors when a choice remains after type matching. Consider the example in Figure 5.4.

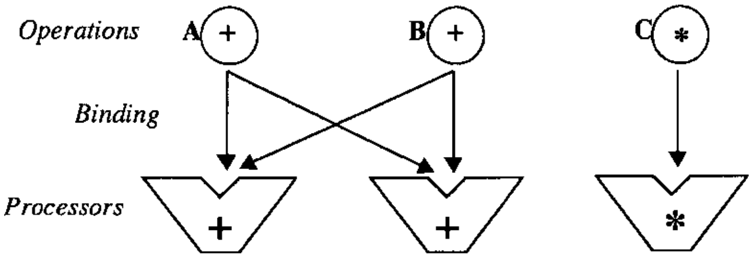


Figure 5.4 Binding operations to processors.

Type matching forces the binding of the multiplication to the multiplier but there still remains a choice for the additions. Defining the set of operations of the same type which are scheduled to occur in the same control step as O_T and the set of available processors capable of implementing that type as P_T, the aspect of binding which is being addressed can be expressed as the binding of O_T, where $|P_T| \geq 2$

Why is such a seemingly insignificant task deserving of attention? This aspect of binding has no influence over the schedule. Consequently, its importance must be assessed by observing the quality of data path designs resulting from different bindings. Figure 5.5 shows the spread of solutions obtained from random bindings for the Wave Digital Filter Example[1] using an allocation of two adders and one multiplier.

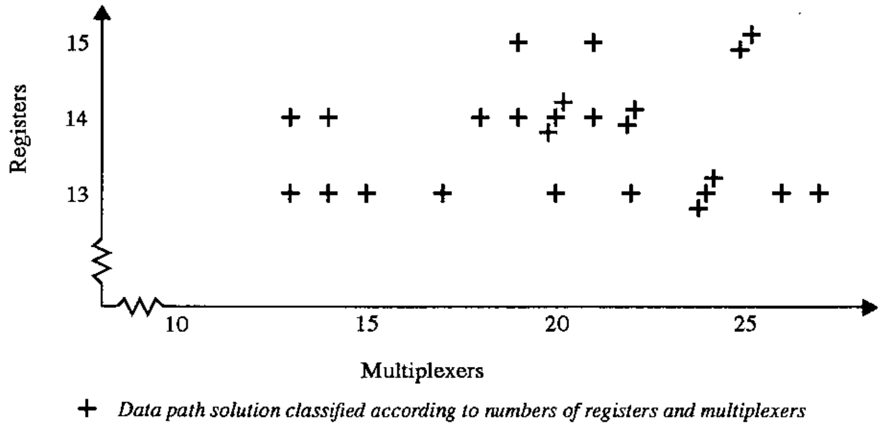


Figure 5.5 Data path solutions for random bindings of the Wave Digital Filter example.

[1] Details of this example are given in Appendix B.3 on page 162.

The aim of the binding algorithm is to select bindings which will simplify the data path allocation task. Three selection criteria have been identified to calculate the binding *affinity* of operation to a processor:

- *Common signals.*

  In binding an operation to a processor, an obligation is placed on the data path allocation algorithm to ensure the transfer of the input signals of that operation to the processor. This criterion attempts to bind operations with common input signals onto the same processor to reduce the number of different structural destinations for signals. A weighting factor is calculated for each processor to which an operation can be bound. It is the percentage of input signals from operations bound to the processor which are common to the unbound operation.

- *Common connections.*

  The second criterion is an attempt to minimise the number of connections between processors. If operations bound to a processor, *P*, require connections to a set of processors, P_{succ}, to accommodate their data flow successors, then binding should attempt to minimise the size of P_{succ} by binding operations to *P* whose data flow successors can be bound to processors of P_{succ}. Unfortunately, the successors of some of the operations, including the one to be bound, will not have been bound and so the attempt must be aimed at minimising operation types rather than the actual processors. A weighting factor is calculated as the percentage of bound operations which share a common successor operation type with the successors of the operation to be bound.

- *Mirroring behavioural patterns.*

  A number of data flow graphs can be divided into easily recognisable partitions. These partitions are groups of operations which have few external signals. It is advantageous to mirror this partitioning in the structure. By binding operations of partitions to separate groups of processing elements, global interconnection can be kept to a minimum. To achieve this result a crude but simple addition is made to the common signals weight. The input signals of the operation to be bound are also compared against the output signals of bound operations. This introduces an attraction of closely related operations to the same processors. An example is provided in section 5.6 .

The percentages calculated for these criteria are averaged to compute the affinity of an operation for a processor.

5.4 Scheduling and Binding with Interconnected Structure.

In synthesis using structural input, a mapping, $B' \rightarrow S'$, is established between a subset of the behaviour, B', and the initial structural input, S'. The algorithms presented in Chapter 4 create a tentative mapping between the operations of B' and the processors of S'. This tentative mapping is the association of operations with sets of candidate processors. By constraining the possibilities for binding in this way, the scheduling process is affected. Furthermore, once an operation has been bound to a single processor, the candidate processor sets of all the operation's data flow successors will be affected[1].

In forming a proper mapping between B' and S', not only must the operations of B' be bound to the processors of S' but the signals of B' must be mapped to the data path components of S'. Hence, a flow of data between operations is mapped to a data path between processors. Data paths are usually synthesised in the data path allocation phase (Chapter 6) after scheduling and binding. However, the mapping must make use of existing paths and as these will affect both scheduling and binding decisions, signals must be bound to data path components at the same time as operations are bound to processors. This is illustrated in Figure 5.6.

[1] Refer to section 4.2.2 on page 58.

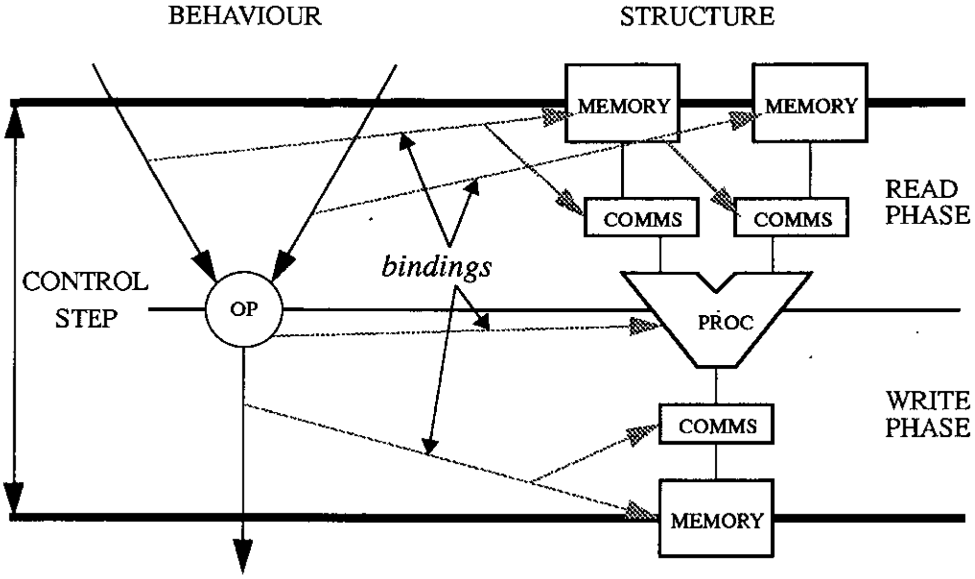


Figure 5.6 Binding operations and signals to structural components.

In this application, scheduling and binding are interdependent and the tasks are performed simultaneously. All the following conditions must be satisfied before an operation can be scheduled:-

- *All the input signals of the operation must be available from memories or input ports.*
- *A processor from the operation's candidate set must be available.*
- *There must be data paths available from the input signals to the processor.*
- *There must be an available path to a memory component which has access to at least one of the candidate processors of each of the operation's immediate successors.*
- *The memory component must have a free location in which to store the operation's output signal.*

The availability of any structural resource is determined from the resource-time graph. For data path components, such as registers and buses, each control step within the graph is divided into read and write phases. If a signal is being read from memory then it is assigned to the read part of the step. Conversely, if it is being written to memory it occupies the write part.

The structural constraint of scheduling now extends from the processor allocation to encompass the data path. Memory and communications can cause data flow bottlenecks and so the usage of these components must be monitored in the resource-time graph alongside the processors'. The scheduling and binding phase completes the mapping $B' \rightarrow S'$ provided that memories are sufficiently large and that the data path network of S' complies with the specification presented in Chapter 3. Structures which require data transfer between memories cannot be supported.

5.4.1 Combining with List Scheduling.

Because scheduling and binding must be performed simultaneously, it is necessary to incorporate binding into the algorithm. The core of the list scheduling approach is in the assignment of candidate operations to the current control step and the deferral of others if there is an insufficiency of suitable, available processors. Little modification is required to adapt this method to cope with interconnected structure. The matching of candidate operations to available processors is extended to include the matching of their signals to available data path components. Operations and signals which are not deferred are then bound before the scheduling algorithm progresses to the next step.

5.5 Integrated Scheduling and Binding.

Scheduling is constrained by data dependencies within the data flow graph and by the type and number of allocated resources. In synthesis using structural input, partitions of the behaviour may have been mapped to structures before the scheduling phase. Operations within these partitions, termed *partitioned operations*, can only be bound to one of a restricted set of candidate processors identified in the matching phase. The remaining *non-partitioned operations* must be bound to processors in the specification which are unconnected. These relationships are outlined in Figure 5.7.

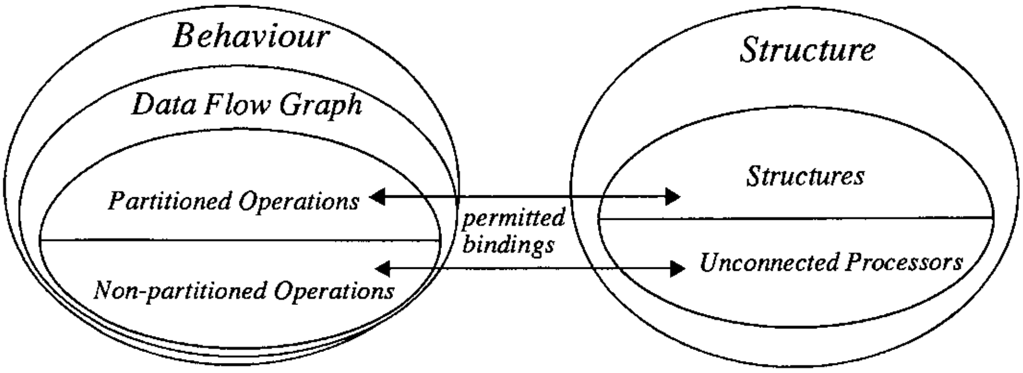


Figure 5.7 Restrictions on binding operations to processors.

All operations belong to one data flow graph which means that data dependencies will exist between partitioned and non-partitioned operations. Consequently, the scheduling of these two categories of operation is mutually dependent. A list scheduling approach is used to encompass all data dependencies within the data flow graph. An outline of the algorithm is listed in Figure 5.8.

```
time_constraint := ASAP_time;
step := 1;
while step <= time_constraint loop;
     s := available_signals(dfg,step);
     O:= candidate_operations_of(dfg,s);
     po:= partitioned_operations_of(O);
     no:= non_partitioned_operations_of(O);
     schedule_and_bind_partitioned(po,step);
     schedule_and_bind_non_partitioned(no,step);
     step := step +1;
end loop;
```

Figure 5.8 List scheduling for partitioned and non-partitioned operations.

Operations are scheduled into each control step in turn, starting with the first step. The outline in Figure 5.8 shows the main core of the scheduling algorithm. The loop allows each step to be dealt with in turn and within each loop the current candidate list of operations satisfying data constraints is put together. *Available signals*, as the name suggests, are those which are available for use by an operation in the current control step. Signals classed as *input* or *constant* will always be present in this list. Other signals are made available by operations which satisfy Equation 5.5.

$$step_{schedule}(op) + steps_{execution}(op) \leq step_{current} \qquad \textbf{Equation 5.5}$$

Candidate operations are those for which both input signals are available. This part of the algorithm, therefore, takes care of the data dependency constraints within the data flow graph. What remains to be addressed are the structural constraints which may force operations to be deferred and further constrain binding.

Partitioned and non-partitioned operations will be bound to mutually exclusive sets of processors. As a result there will be a separate deferral process for each category of operation because only operations from the same category can compete for the same processor. In the case of partitioned operations and for reasons discussed in Section 5.3, binding of an operation and its associated signals is performed at the time of its scheduling. For simplicity, the binding of non-partitioned operations to processors is also carried out during scheduling. Consequently, partitioned operations in the current candidate list undergo deferral and binding separate from non-partitioned operations.

5.5.1 Deferral and Binding for Partitioned Operations.

Figure 5.9 outlines the way in which partitioned operations undergo deferral and binding. An operation which is not deferred will be scheduled and bound in the current control step along with its input and output signals. The operations are listed according to the force attracting them to the current step. The operation yielding the greatest force of attraction to the current step is placed first in the list. Operations and their associated signals are bound to resources on a first-come-first-served basis and so an operation with a stronger force has greater choice of resources.

An operation is deferred when there is no suitable combination of available resources to accommodate it and its input and output signals. Resources include data path components such as registers and buses as well as processors. When an operation is deferred to a step later than its ALAP schedule time, the time constraint is relaxed by a step to enable a valid force calculation to be made in the following step. The binding of an operation and its associated signals is performed in the following manner.

```
ASAP(DFG);
ALAP(DFG,time_constraint);
UPDATE_DISTRIBUTION_GRAPH;
CALCULATE_FORCES(po);
ORDER_BY_FORCE(po);
while SIZE(po) > 0 loop         --for each partitioned candidate operation
     o:=FIRST_OF(po);
     ORDER_BY_BINDING_WEIGHT(P,o);
     deferral:=true;
     P:= AVAILABLE_PROCESSORS(o,RT(STEP));
     while SIZE(P)>0 loop       --for each available candidate processor
          p:=FIRST_OF(P);
          ROUTE(o,p,fail);      --bind operation and signals if possible
          if not fail then
               deferral:=false;
               exit;
          end if;
          DELETE_FIRST(P);
     end loop;
     if deferral and ALAP_OF(o) <= STEP then   --critical path deferral
          time_constraint:=time_constraint+1;
     end if;
     DELETE_FIRST(po);
end loop;
```

Figure 5.9 Deferral and binding for partitioned operations.

The candidate processors of the operation which are available in the current step are identified. They are listed in order of the binding weight which looks for common signals with previously bound operations. Taking each of these processors in turn, starting with the most heavily weighted, an attempt is made to bind the operation's signals to data path components surrounding the processor. This is termed the routing procedure. The procedure is applied until a successful binding is found or all the processors have been tried. When the latter occurs the operation is deferred.

An outline of the routing algorithm is given in Figure 5.10. It considers both ways that a commutative operation can be bound to a processor and if a solution is found then *bind_paths* performs the bindings by updating the resource-time graph. The procedure *locate_path_in* starts with the memory components which store the input signals and looks for a communication component in the structure which connects from the memory to the specified processor input port. The *locate_path_out* procedure looks for paths for the output signal to memory components which have access to processors which are in the candidate sets of operations which consume the output signal.

```
if standard_binding (op,proc) then
      locate_path_in(right_port,right_signal,right_path);
      locate_path_in(left_port,left_signal,left_path);
      locate_path_out(out_port,out_signal,out_paths);
elsif not successful and reverse_binding(op,proc) then   --commutative case
      locate_path_in(right_port,left_signal,right_path);
      locate_path_in(left_port,right_signal,left_path);
      locate_path_out(out_port,out_signal,out_paths);
end if;
if successful then
      bind_paths(op,proc,right_path,left_path,out_path);
end if;
```

Figure 5.10 Routing: binding signals to data path components.

5.5.2 Deferral and Binding for Non-partitioned Operations.

The deferral and binding processes applied to non-partitioned operations, those not already mapped to structure, are outlined in Figure 5.11. It is based on the Force-Directed List Scheduling (FDLS) approach of Paulin et al.[21]. The only significant difference is the inclusion of binding.

```
no := non-partitioned_operations;
procs := available_unconnected_processors
while deferral_necessary(no,procs) loop
      if critical_path_deferral then
            time_constraint := time_constraint+1;
      end if;
      ASAP(DFG);
      ALAP(DFG,time_constraint);
      UPDATE_DISTRIBUTION_GRAPH;
      CALCULATE_FORCES(no);
      DEFER_LOWEST_FROM(no);     --discard operation with lowest force
end loop;
BIND(no,procs);
```

Figure 5.11 Deferral and binding for non-partitioned operations.

5.5.2.1 Deferral.

Deferral is necessary when the number of operations of a particular type exceeds the number of available processors capable of performing that type of operation. Deferral can be divided into three separate cases for consideration. To aid the identification of these cases the following definitions are introduced:

- *O_T: the set of operations of type T.*

- P_T: *the set of available processors capable of implementing type T.*
- O_{CPT}: *the set of operations of type T which are on a critical path of the data flow graph.*

Using this representation, deferral is necessary when Equation 5.6 holds for any operation type T.

$$|O_T| > |P_T| \qquad \text{Equation 5.6}$$

The critical path is a set of operations which form a continuous chain, via directed arcs, and for which the sum of execution times of individual operations is equal to the total schedule time. There may be more than one critical path in a data flow graph. Referring to Figure 5.12, path *A* contains more operations than path *B*. It would therefore appear that path A is the critical path.

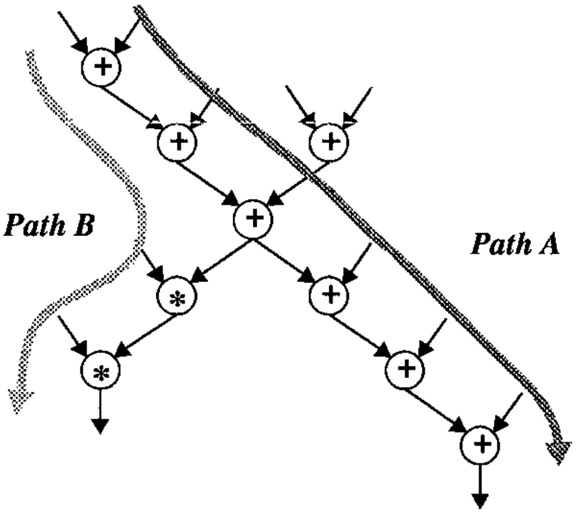


Figure 5.12 The critical path of a data flow graph.

However, if multiplications take twice as many steps to compute as additions then path *B* becomes the critical path. Operations on the critical path are easily identified by the fact that their ASAP schedule step is the same as their ALAP schedule step. Deferral of such an operation would therefore force an increase in the overall schedule time.

The first of the three deferral cases to consider is the simplest to deal with. Equation 5.7 identifies this case. It arises when there are no available processors to perform operations of a given type. The action taken is to defer all operations belonging to O_T.

$$|O_T| > |P_T| \qquad |P_T| = 0 \qquad \text{Equation 5.7}$$

The remaining cases, where $|P_T| > 0$, are tackled within the *force-directed* deferral mechanism.

The second case is defined by Equation 5.8.

$$|O_{CPT}| > |P_T| \qquad O_{CPT} \subseteq O_T \qquad \text{Equation 5.8}$$

This is when the number of critical path operations of a particular type exceeds the number of available processors. The reason for identifying this situation is that the solution will inevitably result in the deferral of a critical path operation forcing an increase in the overall schedule time. In order to accommodate such a deferral the time constraint for the schedule is relaxed by one step and the ALAP schedule and distribution graphs are updated before the force calculations are made for the candidate operations. As a result, a more accurate estimate of the consequences of deferring an operation is obtained from the force calculation. The operation yielding the lowest force is deferred.

The third and final case encompasses the remaining possibilities and so is not identified explicitly. Forces are calculated for all candidate operations and, again, the one yielding the lowest force is deferred.

One of the advantages of the *force-directed* algorithm is that there is no need to identify explicitly the set or sets of operations O_T responsible for deferral. If deferral is necessary then the operation with the lowest force will be a member of a set O_T.

5.5.2.2 Binding.

After any deferrals have been made, the remaining candidate operations are bound to the available processors and assigned to the current control step. This information is stored in the resource-time graph. Binding decisions are based on the *affinity* calculations described in Section 5.3. The operation which has the greatest *affinity* for an available processor is bound first. This process is repeated until all operations have been bound. This greedy method is adequate when the number of candidate operations of any given type is small and when there are no multi-function processors such as *ALU*s. When multi-function processors are involved it is conceivable that the binding process would fail to find a solution involving all the operations. Consider the example presented in Figure 5.13. .

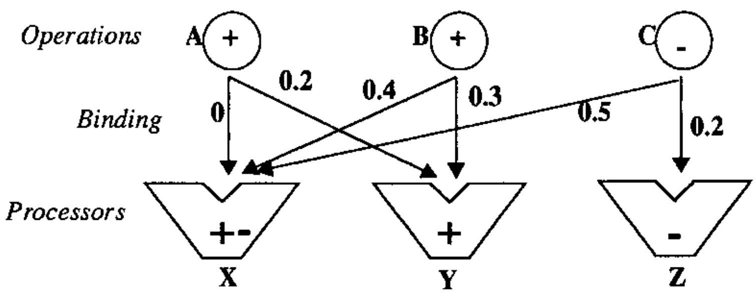


Figure 5.13 Binding affinities of operations to processors.

The greedy approach described above would bind the subtraction to the ALU depriving an addition of a processor. This problem can be avoided by employing the minimum cost flow algorithm. Candidate operations and available processors form the two sets of vertices of a bipartite graph as illustrated in Figure 5.14.

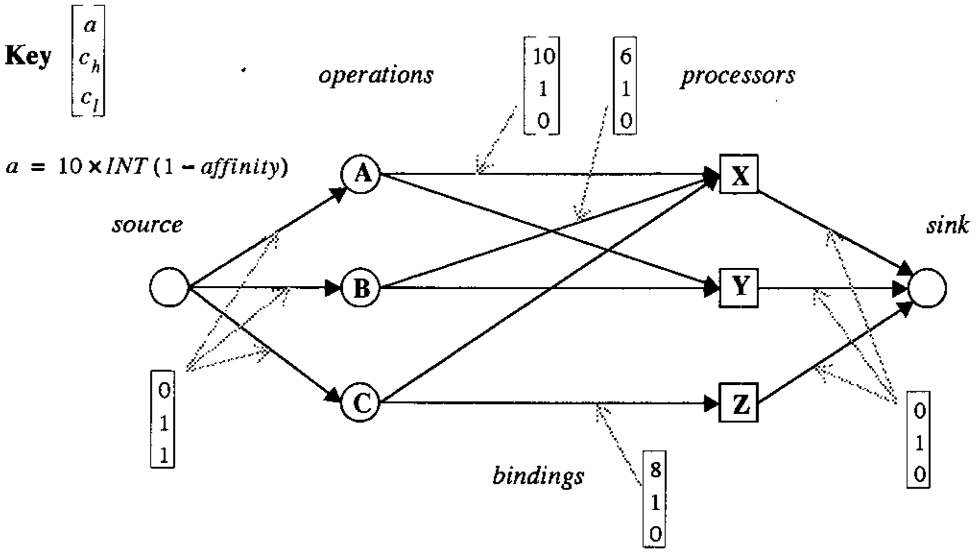


Figure 5.14 Representation of binding using a bipartite graph.

Directed arcs from operations to processors represent possible bindings and each of these arcs is associated with a cost related to the binding affinity, denoted by a.

The network flow algorithm circulates units of flow around the network satisfying the following conditions:

- *flow into vertex = flow out of vertex; flow is denoted by f*
- $c_l \leq f \leq c_h$; *c_l and c_h are the upper and lower capacities of an arc.*

To ensure that all operations are bound, the arcs from the flow source vertex are forced to carry a single unit of flow. The arcs returning flow from the processors to the sink are constrained to carry at most one unit thus preventing the binding of more than one operation to a processor. The flow algorithm is optimal and is based on the out-of-kilter method devised by Ford and Fulkerson [67]. It is coded in Ada and is included in Appendix A.2 on page 151.

Once the algorithm has arrived at a minimum cost flow solution the bindings can be determined from the arcs between operations and processors which carry a unit of flow. Such arcs represent a binding between source and destination.

All that now remains is to enter the information in the resource time graph. The processor will be occupied for the following range of control steps:

$$step_{current} \rightarrow step_{current} + [steps_{re-use} - 1]$$

The operation will be complete by $step_{current} + steps_{execution}$ at which time the output signal of the operation will be available for use.

5.6 Results.

The first example schedules and binds the Differential Equation data flow graph, Figure 5.15, with the structural specification outlined in Figure 5.16.[1] It illustrates the combined scheduling of partitioned and non-partitioned operations and the binding of

[1] Refer to Appendix B.2 on page 157 for further details.

signals to data path components. The structural specification contains four processors, two of which are interconnected to form a structure. Mapping identified a single partition, highlighted in Figure 5.15, to be mapped to the structure.[1] The results are presented in the form of resource-time graphs in Figure 5.17.

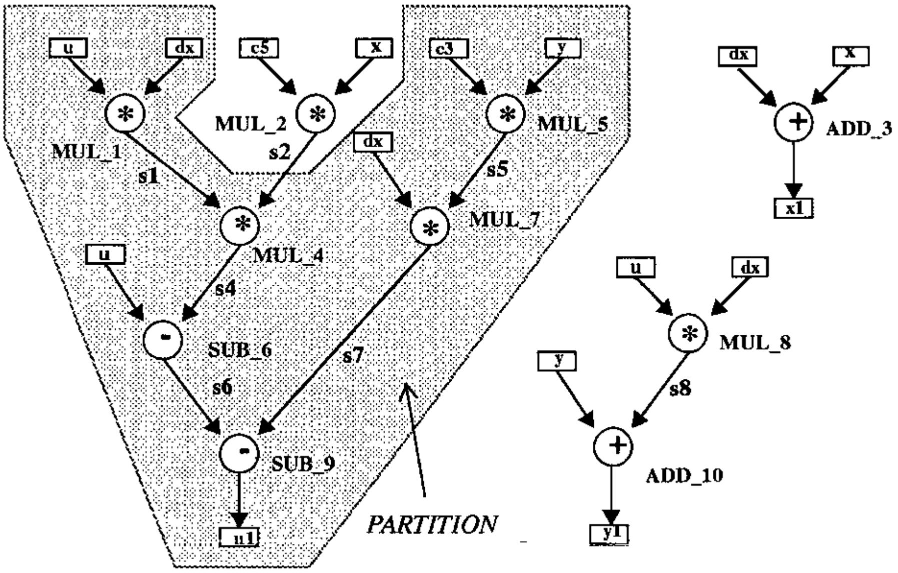


Figure 5.15 Data flow graph behaviour for a differential equation.

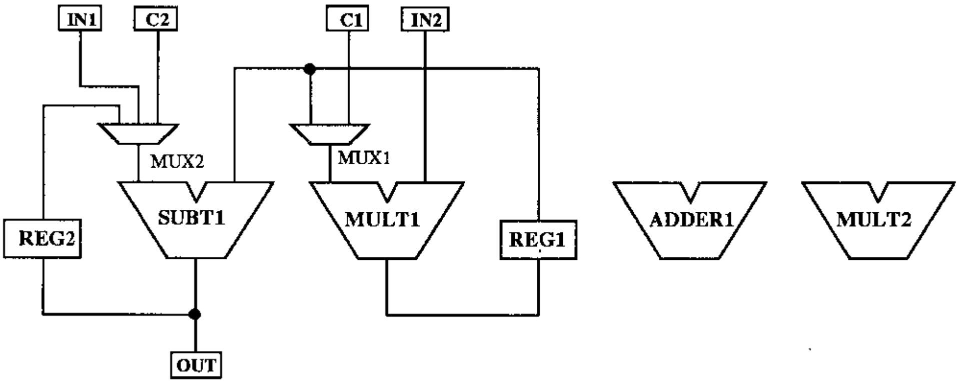


Figure 5.16 Structural specification used in the Differential Equation example.

There are two points worthy of note in these results. Firstly, signal S2 links a non-partitioned operation to a partitioned one. As all signals must enter a structure through input ports, S2 is transferred from MULT_2 to MULT_1 via input port IN_2. Secondly, in order to accommodate their input signals in the data path, the binding of

[1] Refer to section 4.4.1 on page 68 for futher details of mapping for this example.

operations MUL_1 and MUL_7 has been reversed. This means that the left input signal of the operation will enter the right port of the processor and that the right signal will enter the left port. This is permissible because the operations are commutative.

Resource ⟶

ne	*SUBT1*	*MULT1*	*ADDR1*	*MULT2*	*REG1*	*REG2*	*MUX1*	*MUX2*	*C1*	*C2*	*IN1*	*IN2*	*OUT*	
1		*mul5*	*add3*	*mul8*			*c3*					*y*	▨	**REAL**
					s5		▨	▨	▨	▨	▨	▨		**WRITI**
2		*mul1*[1]	*add10*	*mul2*			*dx*		*dx*			*u*	▨	
					s1		▨	▨	▨	▨	▨	▨		
3		*mul4*			*s1*		*s1*					*s2*	▨	
					s4		▨	▨	▨	▨	▨	▨		
4		*mul7*[1]			*s5*		*s5*					*dx*	▨	
					s7		▨	▨	▨	▨	▨	▨		
5	*sub6*				*s4*			*u*			*u*		▨	
						s6	▨	▨	▨	▨	▨	▨		
6	*sub9*				*s7*	*s6*		*s6*					▨	
							▨	▨	▨	▨	▨	▨	*u1*	

1. Input signals reversed.

Figure 5.17 Resource-time graphs for Differential Equation example.

The Wave Digital Filter[1] is perhaps the most scheduled example in behavioural synthesis. The data flow graph is shown in Figure 5.18. The scheduling and binding obtained from a processor allocation of two adders and a pipelined multiplier is presented in the form of a resource-time graph in Figure 5.19. Results obtained from a number of different allocations are provided in Appendix B.3 on page 162.

The behaviour in Figure 5.18 is divided into two partitions. Each partition can be seen to have a similar data flow. With an allocation of two adders, if the structure is to mirror this behavioural partition then the additions from each partition should be bound to separate adders. In the resource-time graph (Figure 5.19) this binding arrangement has been achieved as a result of the influence of binding affinities. The data

[1] Refer to Appendix B.3 on page 162 for further details.

path solution obtained for this result contains 13 registers and 13 multiplexers and can be found in Figure 6.17 on page 122. Solutions obtained for other binding can be seen in Figure 5.5 on page 79.

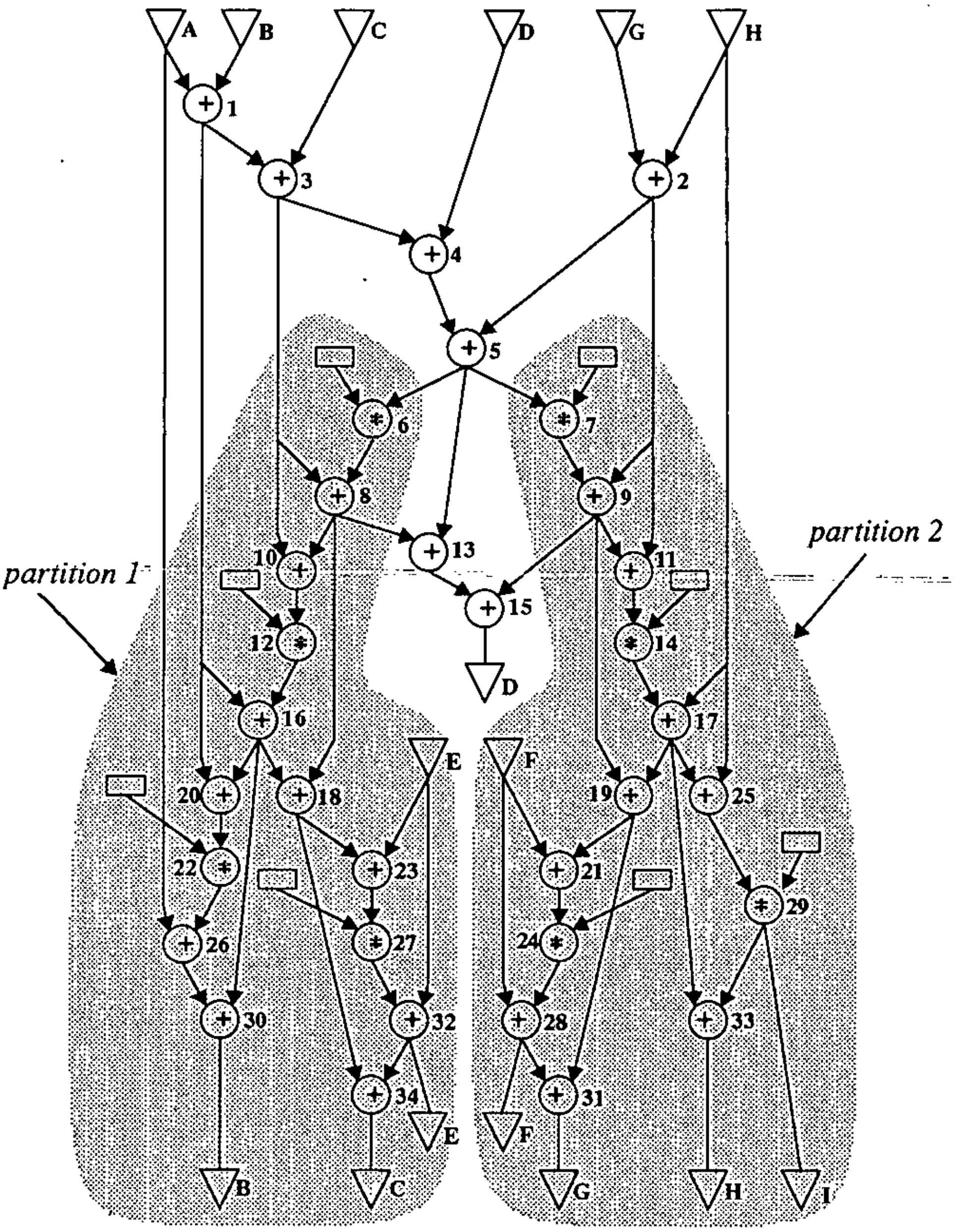


Figure 5.18 Data flow graph for the 5th Order Elliptic Wave Digital Filter

Resource →

Time ↓

Step	*Adder #1*	*Adder #2*	*Multiplier*
1	***1***	***2***	
2	***3***		
3	***4***		
4		***5***	
5			***6***
6			***7***
7	***8***		
8	***10***	***9***	
9	***13***	***11***	***12***
10	***15***		***14***
11	***16***		
12	***20***	***17***	
13	***18***	***25***	***22***
14	***23***	***19***	***29***
15	***26***	***21***	***27***
16	***30***	***33***	***24***
17	***32***		
18	***34***	***28***	
19		***31***	

partition 1 *partition 2*

Figure 5.19 Resource-time graph for Wave Digital Filter example.

5.7 Concluding Remarks.

The scheduling and binding algorithm presented in this chapter achieves the two main objectives posed by the adopted approach to synthesis using structural input. The first of these is concerned with the mapping of partitions of behaviour to specified structures. The mapping algorithms of Chapter 4 identify the partitions and construct tentative mappings to the structures by specifying possible bindings between operations and processors which will ensure a correct functional implementation of the behaviour. In scheduling such operations, these imposed binding constraints are observed and the algorithm ensures, in addition, that free data paths exist to enable the transfer of signals between processors at the scheduled times. The second task addressed by the algorithm is to ensure that operations not involved in mappings to structures are scheduled and interface correctly with other partitions of the behaviour.

The *force-directed* deferral mechanism is used because it fits conveniently with the approach needed to tackle the two afore-mentioned tasks and it produces good results for allocations of unconnected processors. The schedules computed using the *force-directed* method have not been bettered by other systems in terms of the number of control steps required. It is not surprising that many others have also chosen to make use of it.

A limitation of the synthesis system presented in this thesis is its inability to support inter-memory data transfer. In order to accommodate such transfers modifications would have to be made to the timing model and the approach to binding partitioned operations.

Chapter 6
Data Path Allocation.

6.1 Introduction.

The purpose of data path allocation is to provide the structures necessary to support the transfer of data between processors at scheduled times. This involves the synthesis of both memory and communications. The timing model discussed in Chapter 3 dictates that data which exists within more than one control step must be stored in memory. A data transfer which traverses a control step boundary must follow a data path in the structure of the form shown in Figure 6.1.

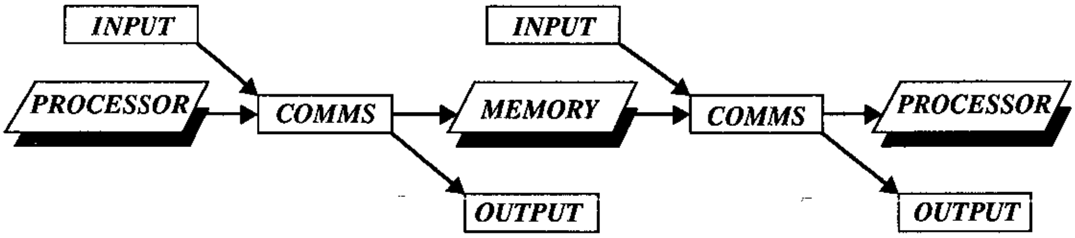


Figure 6.1 Permissible data paths.

As illustrated in Figure 6.2, scheduling dictates the timing of the data transfer: when it is created and consumed and, hence, how long it must be stored in memory. Binding dictates the structural source and destination of the data.

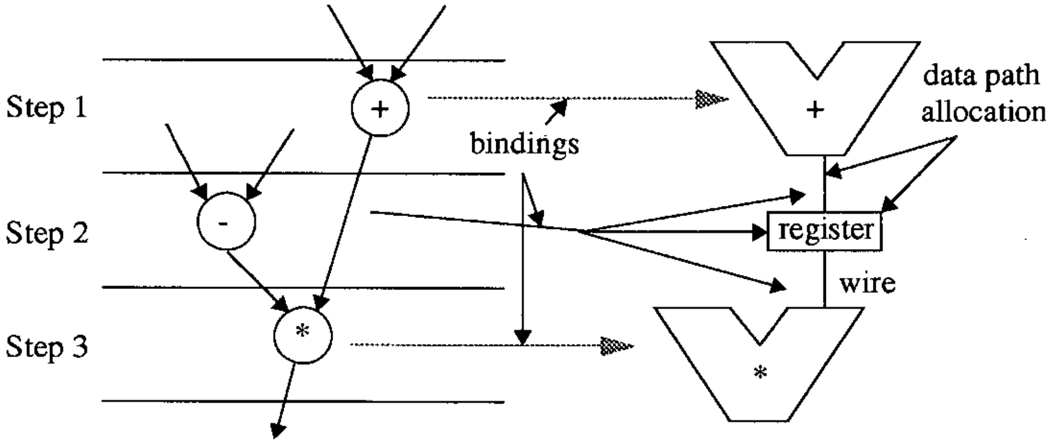


Figure 6.2 The task of data path allocation.

The aim of data path allocation algorithms is to minimise the area occupied by memory and communications subject to the constraints imposed by the schedule and binding of operations. It is straightforward to judge the effect of data path design decisions on the number of memory and communication components as these are actually created in the data path allocation phase. However, components constitute only part of the area cost; the remainder is occupied by wiring. Wiring is not laid down until the routing phase of silicon compilation which is much later in the synthesis process and so it is difficult to assess the effects of data path allocation design decisions.

An alternative approach is to reduce the complexity of the problem presented to the placement and routing algorithms so that they can operate more effectively. This is achieved by influencing the architectural style. An attempt is made to group registers into files or RAM in order to reduce the total number of components and, hence, interconnections. The interconnection topology is also kept simple by restricting the transfer of data to the structural paths outlined in Figure 6.1. This results in a data path which has at most two levels of multiplexing: one is used in the transfer of data to processors and the other in the transfer of data to memory. An additional advantage of this restiction is a reduced time delay for data transfer which means a smaller control step time.

In synthesis using structural input, data path allocation is only required for signals that have not been mapped to structural components. Considering the behavioural synthesis task as a transformation $F'(B, S') \rightarrow S; (S' \subseteq S)$, the initial step is the formation of the mapping $B' \rightarrow S'; (B' \subseteq B)$ which is described in chapter 4. The remainder of the behaviour, $\neg B' \cap B$, consists of operations bound during the scheduling phase and signals which remain unbound. It is for these signals that memory and communications must be synthesised. The algorithms employed must be capable of interacting with the existing data path. For this reason the algorithms are constructive in approach. That is, they operate on a signal by signal basis at some point allowing the algorithm to build upon the prior binding of other signals to existing data path components.

6.2 Memory.

All memory units are deemed to conform to the separate read and write phases of operation presented in Chapter 3. There are two issues concerning the storage of data in memory: firstly, determining whether two signals can share the same location and, secondly, whether they can be stored in different locations of the same memory.

6.2.1 Single Location Storage.

Two signals may share the same location if their lifetimes do not overlap; a signal must be read for the last time before another signal can be written to its location. In addition, referring to the data flow graph representation, two signals represented by separate arcs which come from the same operation will carry the same data but may have different destinations and read times. These signals can, of course, share the same location and so the data need only be written once. There are several well established algorithms which can be used to group signals into a minimum number of memory locations. Clique partitioning and graph colouring techniques are common but the simplest is the 'Left-Edge Algorithm' [30]. This latter technique is used here. The operation of the algorithm is best described using a two-dimensional array which forms the basis of the data structure as illustrated in Figure 6.3.

This array represents memory: the columns are separate locations and the rows form a time axis measured in control steps. A signal occupies a single column from its write time to its last read time: this is its duration or lifetime. Each signal is represented in this way on the array. Minimisation of locations is performed by forcing each signal as close to the left edge of the array as possible without overlapping lifetimes or allowing lifetimes to overlap columns. The number of occupied columns indicates the minimum number of locations required and the signal binding to each location is given by the contents of its column.

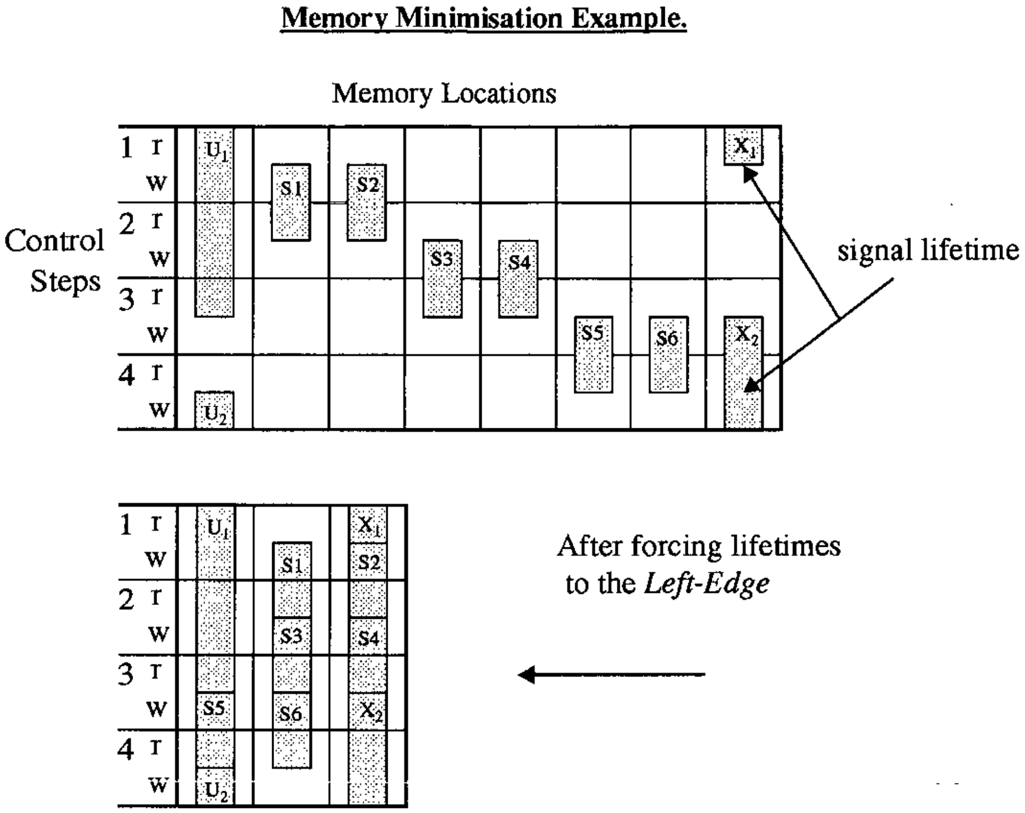


Figure 6.3 The Left-Edge Algorithm.

When loops occur in the data flow, signal lifetimes may overlap the loop boundary. In Figure 6.3, signal U is produced in the first iteration and consumed in the second. In the circular or folded representation shown in the figure the lifetime of U is separated into two parts denoted by U_1 and U_2. To ensure continuity, the portion labelled U_2 must always appear in the same column as U_1 otherwise an attempt will occur to read data from a location to which it has not been written.

6.2.2 Multiple Location Storage.

Structures, such as *register files* and *RAM*, contain a number of storage locations of which only one may be addressed at a given time. The allocation of signals to locations within the memory is made using the Left-edge Algorithm [30]. However, the fact that only one signal can be accessed at a given time leads to an additional constraint: no two signals within the memory can share the same read or write time.

Two methods of assigning signals to multiple location memories are described in section 6.4. The benefits of multiple location memory structures arise from reduced interconnection and control overheads. By grouping *n* registers into a single register file the number of control lines can be reduced by a factor of $\frac{\lceil \log_2 n \rceil}{n}$. In addition, the total number of interconnected components will be reduced thus easing the task of the placement and routing algorithms.

6.3 Interconnection.

Three types of interconnecting structure are synthesised: wires, multiplexers and buses. The definitions of each are detailed below:

- a *wire* connects a single source to one or more destinations;
- a *multiplexer* enables multiple sources to connect to a single destination with the restriction that only one source may be active at any time;
- a *bus* enables multiple sources to connect to multiple destinations but only one source may be active at any time.

Multiplexers and buses are costed in terms of their *equivalent multiplexer inputs*. This term refers to the number of inputs of the device. Hence a bus with two inputs is deemed equivalent in area cost to a multiplexer with two inputs irrespective of the number of bus outputs. The total number of multiplexer inputs (mux. inputs) in a data path design has become a factor for comparison alongside the number of registers in evaluating the merits of a data path design.

6.3.1 Communications Synthesis.

Wires are created initially to convey data between processors, memory and I/O ports in order to satisfy the communications requirements stipulated by the binding of operations to processors and signals to memory. Multiplexers and buses are created as a consequence of interconnection minimisation: the merging of wires. Wires which have common connections can be merged to reduce interconnect area provided that none of the signals carried by the wires have clashing read or write times. Whether a bus or multiplexer is created is dependent upon the number of outputs resulting from the merge. Possible merges are illustrated in Figure 6.4.

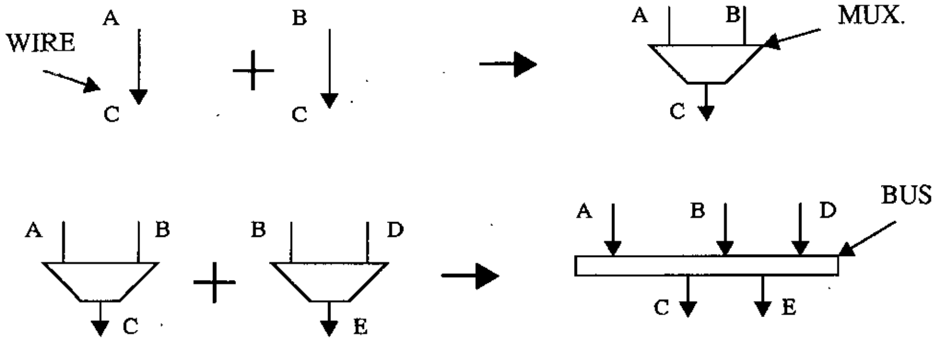


Figure 6.4 Merging Interconnections.

6.3.2 Influence of Scheduling and Binding.

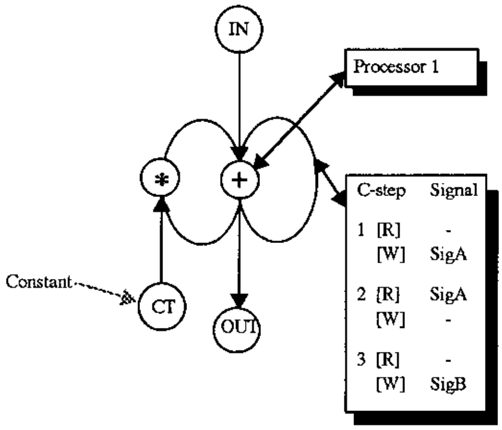
From the scheduled and bound data flow graph it is possible to determine the minimum number of distinct processor-to-memory or memory-to-processor paths required in the design. The communications network conveying data from processors to memory is termed the write network. Similarly, the read network conveys data from memory to processors. Both these communication networks operate in the read phase of the timing model and so no component can be used in both the read and write networks. The minimum number of separate interconnection components in the read network is equivalent to the maximum number of data reads which occur in any control step of the schedule. This naturally also applies to the write network. It can, therefore, be concluded that an attempt to balance the number of operations occurring in each control step may benefit the task of interconnection minimisation by spreading out the data reads and writes. A natural consequence of the fact that the same rules apply to merging interconnect as to combining memory locations is that minimisation techniques will be mutually beneficial. The minimum number of memories will be the same as the minimum number of separate interconnection components in the read network.

6.4 Synthesis Algorithms [68].

The tasks of allocating signals to multiple location memories and merging interconnect are governed by the same rule: there must be no read or write clashes between any signals using the components. The remaining task of minimising the number of memory locations used is achieved by grouping signals whose lifetimes do not clash. Signals with disjoint lifetimes form a subset of those with disjoint read and write times. Hence, an algorithm which groups signals with disjoint read and write times does not exclude the grouping of signals with disjoint lifetimes. Consequently, a synthesis process whose primary objective is the grouping of signals into memories with the aim of reducing the numbers of both interconnection and memory components will help reduce the overall memory size. This is the philosophy behind both algorithms for data path allocation presented in this section.

6.4.1 Clique Partitioning Method.

A connectivity model is the means by which existing structural information is introduced to this synthesis algorithm. The model is initialized to express the connectivity of the structural input. Using data transfer information gleaned from the operation scheduling and binding phases, the model is updated to show the necessary data transfer paths between processors and ports. The connectivity is modelled as a directed graph with processors and ports as vertices, and arcs, termed wires, representing permissible signal flow, or data transfer, between them. Each directed arc in the connectivity graph has associated with it a transfer slot for the read and write times of each control step. A slot can only be filled by a signal which is to be transferred between the source and destination of the arc during the control step. This is illustrated in Figure 6.5. Each signal transfer in the data flow graph is associated with a suitable wire in the connectivity graph. The wire must connect from the processor or port producing the signal to one consuming it. In addition, there must be free slots at the times the signal is transferred and the lifetime of the signal must not overlap with any others in the wire. It is, therefore, possible for a signal to be present in more than one wire. If a suitable wire does not exist then one is created. Wires can be considered as connecting processors and ports through single registers.



C-step		Signal
1	[R]	-
	[W]	SigA
2	[R]	SigA
	[W]	-
3	[R]	-
	[W]	SigB

Figure 6.5 A simple connectivity graph.

6.4.1.1 The Algorithm.

The aim of the algorithm is to minimise the number of memory locations, multiplexers and connections by merging wires in the connectivity graph. Any two wires can be merged provided that no signals carried by them have to be transferred at the same time. The algorithm finds all maximal merges of wires in the connectivity graph using clique partitioning. A maximal merge is one which is not a subset of any other merge. From this list of merges produced, the best is selected based on estimates from memory and multiplexer cost functions. When a merge has been selected its component wires are removed from the remaining merges which are then sorted to remove those which have become non-maximal. The selection process is then repeated until no wires remain. This is illustrated in Figure 6.6.

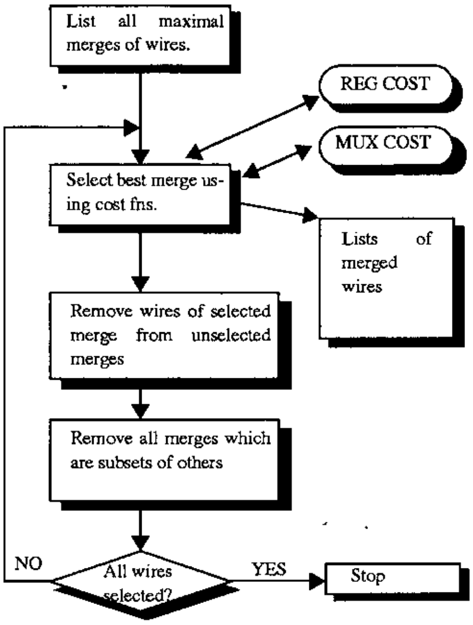

List all maximal merges of wires.
REG COST
MUX COST
Select best merge using cost fns.
Lists of merged wires
Remove wires of selected merge from unselected merges
Remove all merges which are subsets of others
NO
All wires selected?
YES
Stop

Figure 6.6 Flow diagram of synthesis algorithm.

6.4.1.2 Clique Partitioning.[1]

The clique partitioning algorithm is used to find the largest, distinct, groups of wires which can be merged together. In order to do this a compatibility graph is constructed in which vertices represent wires and arcs represent possible merges between two wires. As exhaustive clique partitioning is an NP-complete problem it is important to ensure that the complexity of the compatibility graph does not exceed that which can be partitioned in realistic computing time. The complexity of the graph increases with the number of vertices and cliques it contains. The algorithm used, which is a variation of the Bron-Kerbosch Algorithm, has been tested on Moon-Moser graphs, which contain the most cliques per vertex and so are the worst case for clique partitioning. The performance for these graphs compared with that for graphs encountered in real examples is shown in Figure 6.7.

[1] A clique partitioning algorithm, coded in Ada, is provided in Appendix A.1 on page 147.

Graph	Vertices	Density[1]	Cliques	Time (s)
Moon-Moser	18	0.88	729	0.6
	21	0.90	2187	1.9
	24	0.91	6561	5.9
	27	0.92	19683	17.8
	33	0.94	177147	168.6
	39	0.95	1594323	1591.4
Elliptic Filter				
21 c-steps	18	0.56	37	0.6
19 c-steps	21	0.66	39	0.9
18 c-steps	16	0.58	26	0.3

[1] Density = 2*arcs/(vertices*(vertices-1))

Figure 6.7 Clique partitioning performance.

6.4.1.3 Memory Cost Function

The cost function computes the number of memory locations required for a given merge of wires and provides a measure of the cost gain made. The Left Edge Algorithm [4] is used to calculate the number of memory locations. The cost gain is based on the reduction in required memory locations resulting from the merge and the utilisation of the resultant locations. Memory utilisation is defined as the percentage of the total time that locations are storing data. This is shown in Equation 6.1 for a memory with n locations storing data over time t.

$$usage = (n \times t) - \sum_{signals} lifetimes \qquad \textbf{Equation 6.1}$$

The memory cost is then calculated as follows:-

$$cost = -\Delta locations - (1 - usage) \qquad \textbf{Equation 6.2}$$

6.4.1.4 Multiplexer Synthesis and Cost Function

In the connectivity model described earlier there are two situations which require the synthesis of multiplexers: firstly, more than one output port connected to a wire and, secondly, more than one wire connected to a single input port. The former is easily resolved: given a merge consisting of x wires coming from y different sources then a y input mux is required and the number of mux inputs is increased by y. The latter is more complicated. In the case of processors executing non-commutative operations, such as subtraction, the input signals are constrained to arrive at pre-defined ports to preserve functionality. In the case of commutative operations, such as addition, however, it is possible for an input signal to arrive at any port provided that two signals used in the same operation do not arrive at the same port. If a number of wires are to be connected to the input ports of a processor, this problem of multiplexer synthesis can be expressed as how port assignment should be made so as to minimise the number of wires which must be connected to both ports. An algorithm which produces the best port assignment of wires for each processor is presented in Figure 6.8.

```
-- RESOLVE PORT ASSIGNMENTS
-- COMMUTATIVE CASE
I:= set of input wires;
C[1..n]:= `n' combinations of input wires;

lowest:= 2*SIZE(I);
for left in 1 .. (n - 1) loop;
   for right in left .. n loop
      spread := C[left] ∪ C[right];
      remainder := I ∩ spread;
      if SIZE(remainder) < lowest then
         lowest := SIZE(remainder);
         left_port := wires_of(C[left]);
         right_port := wires_of(C[right]);
      end if;
   end loop;
end loop;
```

Figure 6.8 An algorithm to resolve port assignments.

The multiplexer cost function is an estimate of the reduction in multiplexer inputs which will be made by merging a given set of wires. A simple strategy is to base the cost function on the number of common sources and destinations of the wires in the proposed merge. This is augmented by the change in the number of multiplexer inputs resulting from the merge as shown Equation 6.3.

$$cost = \Delta mux - [2 \times wires - (sources + destinations)] \qquad \textbf{Equation 6.3}$$

The change in the number of multiplexer inputs is an important selection criterion towards the end of synthesis when the cost of multiplexing processor inputs emerges.

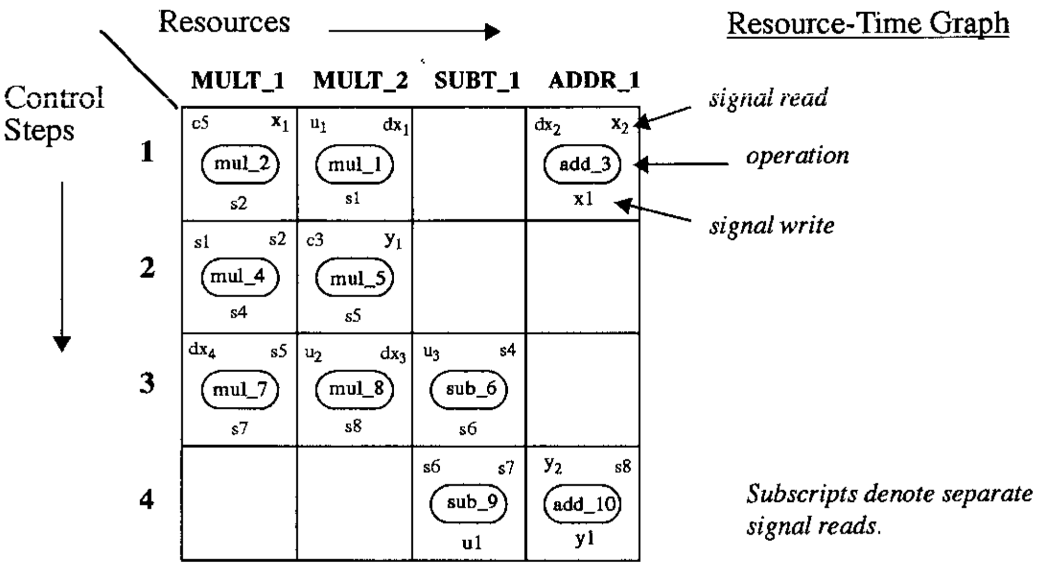
6.4.2 Assignment Method.

This algorithm groups signals into sets: a set contains only signals with disjoint read and write times. Each set represents the assignment of signals to a memory component. The minimum number of sets will, therefore, be at least the maximum number of data transfers in any control step. In any scheduled data flow graph reads will outnumber writes. This arises from the fact that processors have more inputs than outputs.

The first step in the algorithm is to tabulate the signal reads according to the control step in which they occur. Each row in the table contains signals from a single control step. The signals occupy separate columns. Figure 6.9 illustrates this.[1]

Differential Equation Example.

Resource-Time Graph



Control Steps	MULT_1	MULT_2	SUBT_1	ADDR_1
1	c5 x_1 mul_2 s2	u_1 dx_1 mul_1 s1		dx_2 x_2 add_3 x1
2	s1 s2 mul_4 s4	c3 y_1 mul_5 s5		
3	dx_4 s5 mul_7 s7	u_2 dx_3 mul_8 s8	u_3 s4 sub_6 s6	
4			s6 s7 sub_9 u1	y_2 s8 add_10 y1

Signal Read Table

Control Steps	Signals →					
1	c3	dx_1	dx_2	x_1	x_2	u_1
2	s2	s1	c5	y_1		
3	s5	s4	dx_3	dx_4	u_2	u_3
4	s8	s7	s6	y_2		

Figure 6.9 Tabulated signal reads for the Differential Equation Example.

[1] Details of the Differential Equation Example are given in Appendix B.2 on page 157.

After construction, the rows of the table are sorted according to the number of different data values present within them. The row with the most data values is placed first. Rows with the same number are sorted according to the number of signals which exist during their respective control steps. This sorting is effectively an ordering of control steps according to the level of communication activity within them. As signals which belong to the same row share the same read time, they cannot share the same memory set unless they convey the same data. Consequently, the number of different data values in the first row, the most active control step, indicates the minimum permissible number of sets necessary to accommodate the signals. This number of sets is then created to house the first row of signals. The algorithm now proceeds on a row by row basis starting with the next most active row. The largest rows, in terms of data transfers, are the most awkward to assign to memory sets because of the greater potential for clashing. For this reason such rows are dealt with first, thus maximising the freedom of assignment for the signals within these rows. Each signal in the row is assigned to a memory set in an attempt to minimise a cost composed of memory and communication factors. Three rules govern the assignment to ensure that no read or write clashes can occur.

- All signals from the row must be assigned to memory sets.
- No two signals from the same row can be assigned to the same set unless they convey the same data.
- The signal being assigned cannot have the same write time as any other signal within the set unless they convey the same data.

6.4.2.1 Commutative Operations.

The binding of operations to unconnected processors does not stipulate how the operation's input signals are to be mapped to the processor's input ports. This is implicit in the case of non-commutative operations such as subtraction where the order of operands must be observed to preserve the correct functionality. That is, the left signal must be mapped to the left input port and the right signal to the right port. However, for commutative operations, such as multiplication or addition, the ordering does not affect the functionality and so there is a degree of freedom available in deciding the port of destination for the signal. This freedom is beneficial in communications synthesis.

6.4.2.2 Signal Constants.

Constants are assigned to memory sets but they are not included in registers, register files or RAM. For this reason they do not affect the memory cost function. They are associated with memory sets solely for interconnection reasons. The constants are fed into the read network using multiplexers. This multiplexing cost is introduced through their assignment to a memory set.

6.4.2.3 Cost Estimation.

The calculated costs reflect estimated increases in the number of datapath components resulting from the addition of a signal to a memory set. They are intended as a means for differentiating between possible assignments of signals within the same row and are not intended as a global prediction.

- *Interconnection cost.*

This is calculated as the effective increase in the number of mux. inputs, for both the read and write networks, resulting from the addition of a signal to a memory set. In the write network, the number of mux. inputs can be calculated as the number of different sources of signals of the memory set if that number is greater than one. Therefore, the mux. write cost incurred by adding a signal can be calculated as follows:

$$\cos t = \begin{cases} 0 & \text{no change in number of sources} \\ 1 & \text{extra source to memory set: number of sources} > 1 \\ 2 & \text{extra source to memory set: number of sources} = 1 \end{cases}$$

Calculations for the read network operate following the same principle. However, because the memory sets are inputs to the read network, the destinations of

$$cost = \begin{cases} 0 & \text{no change in destinations} \\ 1 & \text{extra destination in network: number} > 1 \\ 2 & \text{extra destination in network: number} = 1 \end{cases}$$

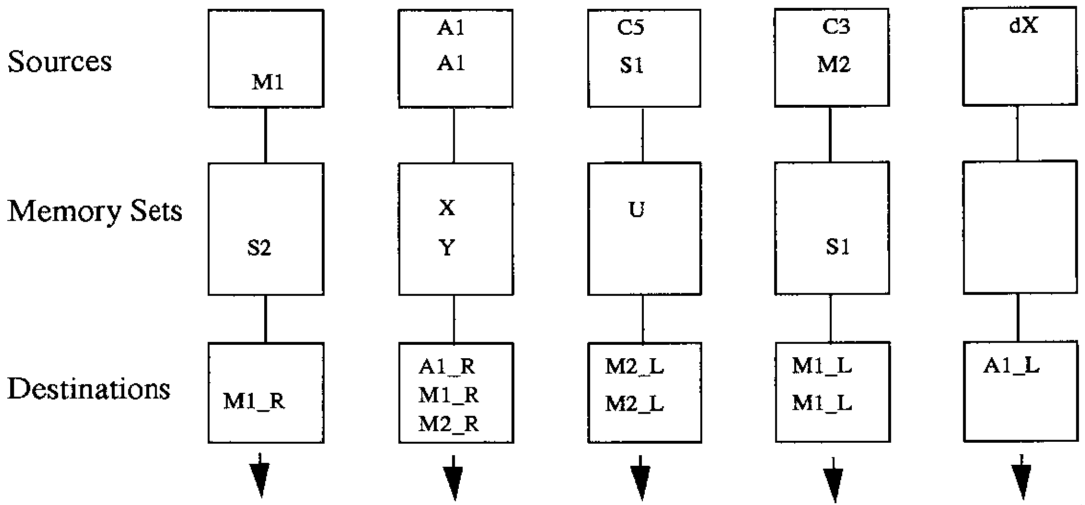
all the memory sets must be considered together. The number of mux. inputs can be calculated as the number of times each destination is used from different sets provided that it is used more than once. The mux. read cost is calculated as follows:

- *Memory Cost.*

The memory cost is the increase in the number of locations, if any, as a result of the addition of a signal. This calculation is performed using the principle of the Left-Edge Algorithm [30]. It requires little computation as the data structure, the array of signal lifetimes, is maintained and updated throughout the assignment process. In estimating the cost, the lifetime of the signal is compared against the array of lifetimes of the memory set to determine whether an increase in locations is required. An additional factor is included in the memory cost: if the signal to be assigned conveys the same data as a signal within the memory set then the cost is decremented. The reason for this is that the total lifetime of stored signals has been reduced by an overlapping of lifetimes. Conversely, if the signal to be assigned will prevent the assignment of a signal conveying the same data as a signal within the memory set then the cost is incremented.

Cost calculation is illustrated using the example presented in Figure 6.9. The first two rows of signals have been assigned forming the memory sets shown in Figure 6.10. The costs calculated for S5 the first signal of the third row, are outlined therein. The signal S5 is to be transferred from MULT_2 to MULT_1. As the operation MUL_7 is commutative, S5 can be sent to either M1_L or M1_R, the labels for the left and right input ports of MULT_1 respectively.

Cost calculations for the assignment of s5 to memory from the output port of MULT_2 (M2).



PORTS	M1_L	M1_R	M1_L	M1_R	M1_L	M1_R	M1_L	M1_R	M1_L	M1_R
REG	0	0	0	0	0	0	0	0	2	2
MUX_IN	1	1	1	1	1	1	0	0	1	1
MUX_OUT	1	0	1	0	1	1	0	1	1	1
TOTAL	2	1	2	1	2	2	0	1	4	4

Figure 6.10 Cost calculation for signal assignment to memory sets.

6.4.2.4 Assignment Mechanism.

Within each row of signals to be assigned there will exist pairs of signals which are used in the same operation. If the operation is commutative then the signals have a choice of processor port. This influences the interconnection costing and so each signal feeding a commutative operation has two costs associated with each possible assignment to a memory set: one cost for each port. The costs can be set out in an array format as shown in Figure 6.11, a continuation of the example introduced in Figure 6.9. An extra memory set is included in the costing array to allow the possibility of using an extra memory instead of excessively increasing the interconnection cost of an existing one. The cost of a new memory set is specified as 3 for the examples shown in this section.

Signals and Ports

		S8		S7		S6		Y	
		LEFT	RIGHT	LEFT	RIGHT	LEFT	RIGHT	LEFT	RIGHT
Memory Sets	1	2	2		▨ 0	2		3	3
	2	3	2		2	2		0	▨ -1
	3	2	2		2	▨ 0			
	4	▨ 1	1		2	2		3	3
	5	2	3		3	3		2	3

▨ Selected assignment of signal, port and memory set.

Figure 6.11 Cost array generated for assignment of third row of the example.

The assignment mechanism establishes a mapping between signals and memory sets and, in the case of commutative operations, processors input ports. The fist step involves identifying the lowest cost solution for each signal; that is the position of the lowest cost in each row of the costing array. The signal assignment from each row proffering the lowest cost is put forward for selection. If two or more possible assignments in the row share the lowest cost then the difference between this lowest cost and the lowest alternative solutions for each memory set involved is calculated. This is a column cost difference. The assignment put forward is then the one which has the largest memory set cost difference. Once a possible assignment has been selected for each row the difference between the cost of this assignment and the next lowest in the row is calculated. This value is a row cost difference. The assignment which threatens the highest cost difference is selected. If, again, a number of assignments lay claim to the highest cost difference then the assignment with the lowest cost is chosen.

After an assignment has been selected, the signal's row is removed from the cost table. If the signal is used in a commutative operation then the costs associated with the port which it uses are also removed. Other possible assignments to the memory set involved are removed unless they involve the same signal data. The process of selecting an assignment and updating the cost array is repeated until all the signals have been assigned.

Once each row of signal reads in the transfer table has been dealt with, the assignment of signals to memory will be complete. The memory set binding can be translated directly into a data path. The number of storage locations required by each memory set is calculated using the Left-Edge Algorithm [30] as mentioned in *Memory Cost*. If only one location is required then a single register is used. If more than one is needed then a register file or RAM is used. The components are interconnected according to the sources and destinations of the signals within the memory. Multiplexers are created where wires converge on a common destination.

6.4.2.5 Using existing data paths.

The assignment method accommodates existing interconnection. Memory components from the initial specification of structure are represented as memory sets. The signal binding to each memory resulting from scheduling is placed in each memory set. These sets form part of the initial group of sets created to accommodate signals from the most active control step, the first to be assigned. The following aspects of the data path from the structural specification can be controlled:

- *the size of memory components;*
- *the connections to memory components;*
- *the binding of additional signals;*
- *the number of additional memory components.*

6.4.3 Synthesising Buses.

In the read or write network, if there are more interconnection components than the maximum number of simultaneous data transfers then it may be possible to reduce the interconnection further by creating buses. Each network is dealt with separately. Clique partitioning is used to identify possible merges of interconnection components. A compatibility graph is constructed with vertices representing the components and arcs indicating possible merges. The merging of two components is possible if the components are not used at the same time. The resultant cliques are the largest groups of components which can be merged. The clique which offers the greatest reduction in mux. inputs is selected and its constituent components are merged to form a bus. These components are removed from the remaining cliques. This process is repeated until no cliques remain.

6.5 Results.

In only a few scientific papers on high-level synthesis are data path designs shown in a schematic form and in none is the actual schedule and binding for such designs stated. This prevents an accurate comparison of the effectiveness of the various data path allocation techniques as the algorithms will not necessarily be operating on the same scheduling and binding input data although the number of control steps and the processor allocation may be the same. The results presented for the clique-based (CLIQUE) and assignment-based (ASSIGN) techniques described in section 6.4 are derived from the same input data.

When comparing data paths which perform the same function there are other points to compare besides the number of registers and mux. inputs. Wiring forms a significant proportion of data path area and so it is worth considering the number of wires, and the number of point to point connections which they make. Figure 6.12

contains a single wire which makes four point to point connections.

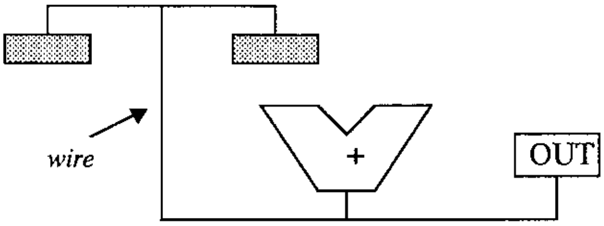


Figure 6.12 Wire with four point to point connections.

Furthermore, wiring is also required in the control part of the data path. The number of control wires required by data path components can be calculated as follows:-

- *Multiplexers.*

The number of control lines required is a function of the number of inputs (n) as presented in Equation 6.4.

$$wires = \lceil \log_2 n \rceil \quad +$$

Equation 6.4

- *Registers.*

Registers are assumed to require a single control line to indicate whether a read or write is to take place.

- *Register files and RAM.*

A single read/write wire is assumed to be required in conjunction with a number of address lines. The number of address lines is a function of the number of locations (n) and can be calculated using Equation 6.4.

- *Buses.*

These require a separate control line for each tri-state driver. This is equivalent to the number of bus inputs.

Three examples are used: an example first introduced by FACET [69], the Differential Equation Example and the Wave Digital Filter Example. Details of all these examples are to be found in Appendix B.

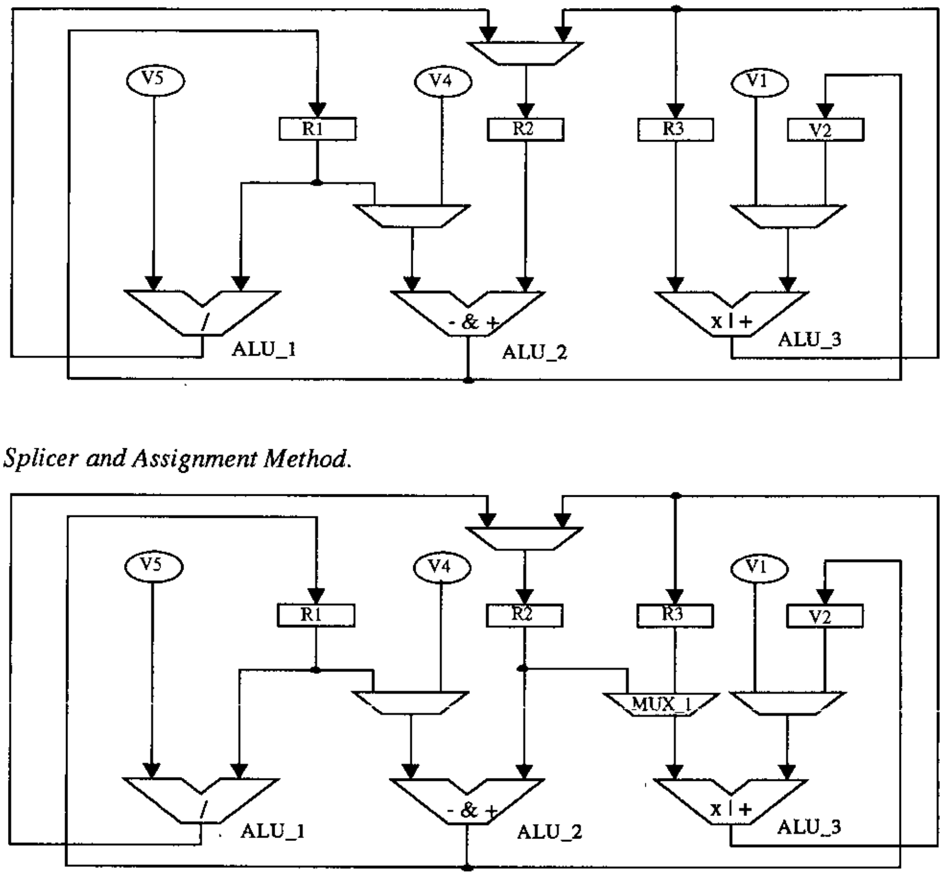
6.5.1 The FACET Example.

A resource time graph displaying the schedule and allocation used is provided in Appendix B.1.2 on page 156. The results in comparison with other systems are given in the table in Figure 6.13.

SYSTEM	STEPS	PROCESSORS	REGISTERS	MUX INPUTS
FACET [69]	4	[/] [-&+] [*|+]	8	11
SPLICER[50]	4	[/] [-&+] [*|+]	7	8
ADPS [56]	4	[/] [-&+] [*|+]	7	10
CLIQUE	4	[/] [-&+] [*|+]	7	6
ASSIGN	4	[/] [-&+] [*|+]	7	8

Figure 6.13 Table of results for the FACET example.

Data path schematics are shown in Figure 6.14. The data paths produced by SPLICER and ASSIGN are identical. SPLICER uses an exhaustive branch and bound technique and claims that an optimal design in terms of mux. inputs results. However, CLIQUE produces a data path with only three multiplexers instead of four, two mux. inputs fewer. Examination of the contents of register, *R2*, reveals that only signal $s1_1$ needs to be transferred to *ALU_3* via *MUX_1*. This signal can be stored in register *R3* thus removing the need for *MUX_1*. This is the solution arrived at by CLIQUE.



(a) Using the Clique Method.

(b) Splicer and Assignment Method.

Figure 6.14 Data paths for the FACET example.

6.5.2 The Differential Equation Example.

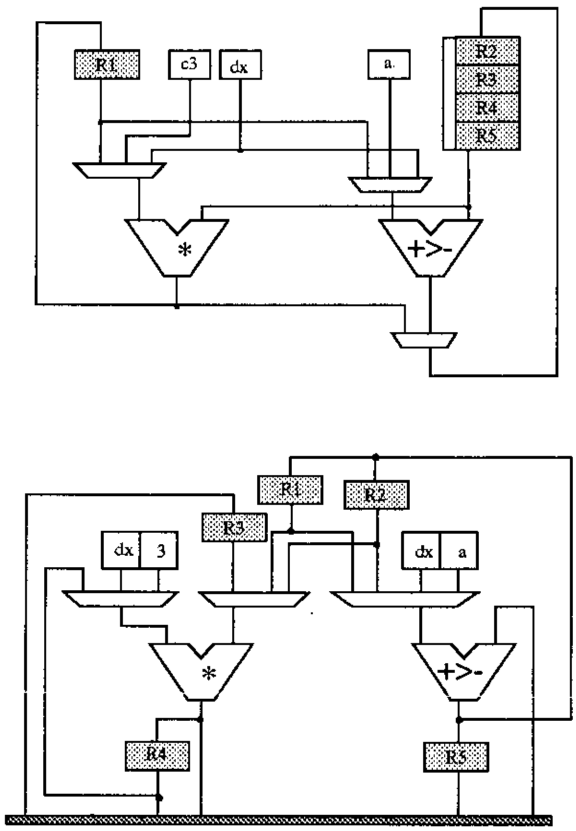
The resource time graphs used by the CLIQUE and ASSIGN algorithms are shown in Appendix B.2.3 on page 161. Data path schematics are shown in Figure 6.16. A summary of the results is given in Figure 6.15.

HAL and ASSIGN produce data paths requiring a similar amount of multiplexing and the same amount of memory. One third of the multiplexing in the ASSIGN data path is used in decoding the register file. However, as a result of grouping four registers into a file significant savings are made in the number of wires (47%), control wires (31%) and point to point connections (41%). These reductions are made at the expense of a single mux. input.

SYSTEM	STEPS	PROCESSORS	REGISTERS	MUX INPUTS
HAL[21]	4	2*,+,-,>	5	10
SPLICER[50]	4	2*,+,-,>	6	11
CLIQUE	4	2*,+,-,>	5	11
ASSIGN	4	2*,+,-,>	5	11
HAL[21]	8	*, [+->]	5	11
ASSIGN	8	*, [+->]	5	8 (+4)[1]

[1] Register file decoding shown in brackets.

Figure 6.15 Table of results for the Differential Equation Example.



(a) ASSIGN

Data Path :-
Registers = 5
Mux. inputs = 8+4
Control wires = 9
Wires = 9
Point-to-point connections =13

(b) HAL

Data Path :-
Registers = 5
Mux. inputs = 11
Control wires = 13
Wires = 17
Point-to-point connections =22

Figure 6.16 Data paths for Differential Equation Example.

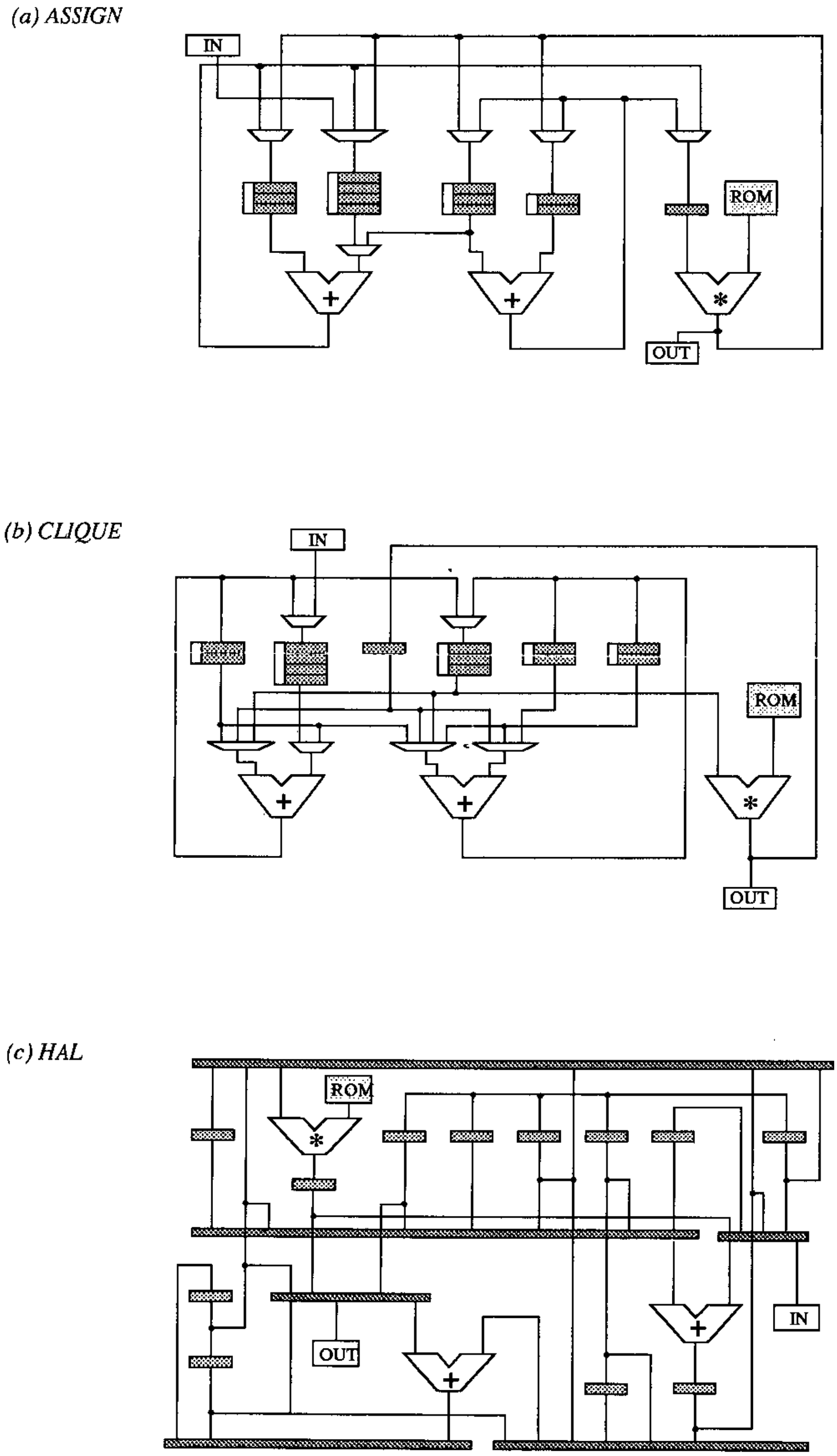
6.5.3 The Wave Digital Filter Example.

This is the largest and most popular example used in behavioural synthesis. It contains only commutative operations which maximises the freedom for processor port assignment. Only one data path schematic using two adders and a pipelined multiplier has been uncovered in existing literature: a design synthesised by HAL which operates in 19 control steps. Data path schematics are shown for comparison in Figure 6.17. The schedule and binding from which these data paths were synthesized is shown in Figure 5.19.

Comparing the solutions synthesised by (a) ASSIGN and (b) CLIQUE in Figure 6.17 it can be seen that the ASSIGN data path has fewer memory components and fewer multiplexer inputs. In fact it uses the minimum number of memory components (6); this is the maximum number of simultaneous data transfers which occur in the schedule. Nevertheless, the numbers of wires, control wires and point to point connections required by the data paths are similar as illustrated in Figure 6.18. The multiplexing concentration is highest in the write network for ASSIGN and the read network for CLIQUE. This can be attributed to differences is mux. input costing.

The architectural style of ASSIGN and CLIQUE differs significantly from the bus based approach of HAL. A bus supports communication between a number of sources and destinations, but it can only transfer a single item of data at any time. This means that it is not an effective device for communicating between highly utilised components. Figure 6.19 shows the percentage utilisation of the memory components of ASSIGN and CLIQUE. In neither of these data paths is it cost effective in terms of equivalent mux. inputs to create a bus. By splitting the register files into single registers the number of components is increased and the utilisation lowered.

This is the situation in the HAL data path. Such conditions are more suited to the synthesis of a bus based solution. However, in Figure 6.18, it can be seen that reducing the number of memory components can lead to fewer wires (45% reduction), fewer control wires (50% reduction) and fewer point to point connections (36% reduction).Tables in Figure 6.20 compare the results to those of other tools for different processor sets. Indicated within these tables in brackets is the additional mux. cost incurred by using register files.



(a) ASSIGN

(b) CLIQUE

(c) HAL

Figure 6.17 Data path solutions for Wave Digital Filter Example.

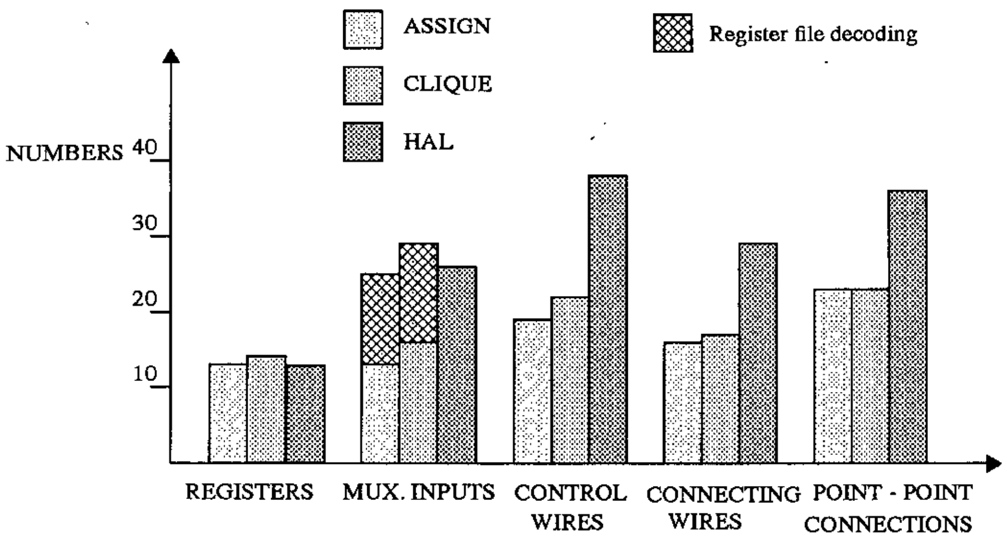


Figure 6.18 Points comparison between ASSIGN, CLIQUE and HAL data paths.

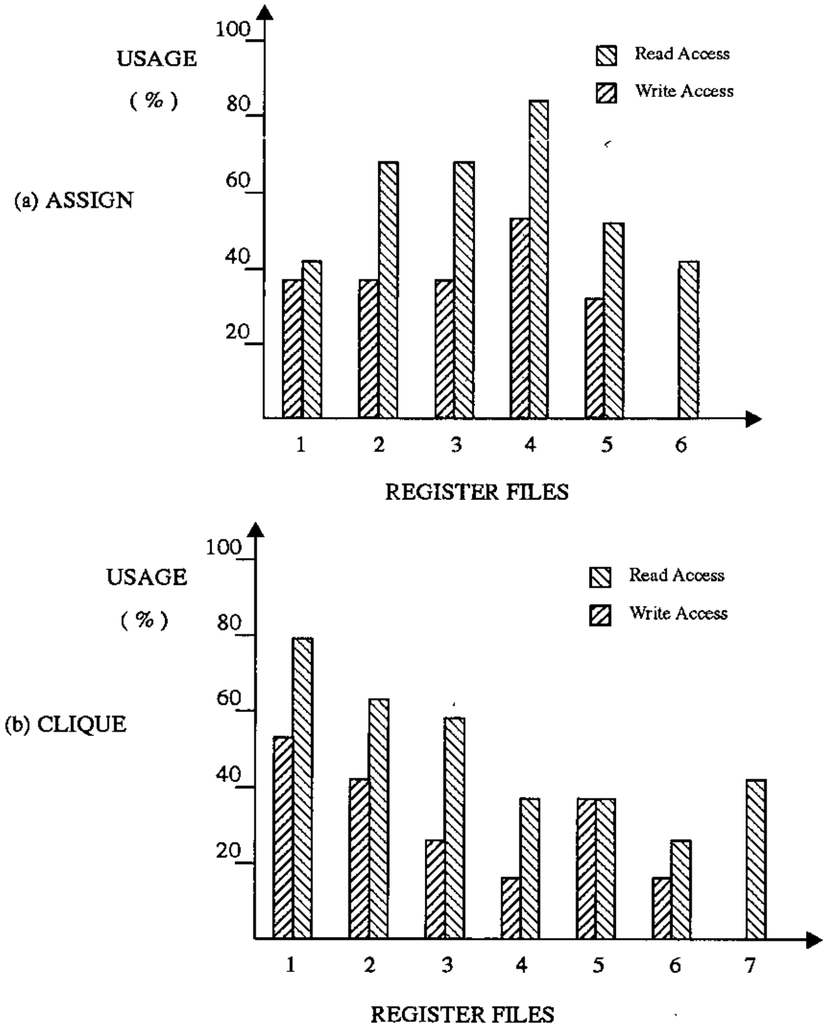


Figure 6.19 Utilisation of memory components in ASSIGN and CLIQUE data paths.

SYSTEM	STEPS	PROCESSORS	REGISTERS	MUX INPUTS[4]	TIME (S)[3]
MABAL [28]	19	2+,1*P	10	32	
MC^2 [71]	19	2+,1*P	16	16 (+ 14)	
HAL [21]	19	2+,1*P	12	26	
ASYL [72]	19	2+,1*P	12	26	
ESC [19]	19	2+,1*P	15	25[1]	
CLIQUE	19	2+,1*P	14	16 (+13)	
ASSIGN	19	2+,1*P	13	13 (+12)	10.6
SAM [34]	18	2+,2*P	12	27[2]	
SAW [52]	18	2+,2*P	12	34	
CLIQUE	18	2+,2*P	14	17 (+ 13)	
ASSIGN	18	2+,2*P	15	17 (+13)	11.8[3]
SPLICER[50]	21	2+, 1*	N/A	35	
HAL [21]	21	2+, 1*	12	30	
ESC [19]	21	2+, 1*	16	23[1]	
ASSIGN	21	2+, 1*	13	13 (+12)	10.7[3]

1. Register file decoding not included.
2. Uses manual register assignment.
3. Computation performed on SUN 3/60.
4. Register file decoding indicated in brackets.

Figure 6.20 Table of results for the Wave Digital Filter Example.

6.6 Concluding Remarks.

The ASSIGN method is favoured over the CLIQUE method because it tends to produce a more balanced data path with fewer memory components and global communications. By balanced it is meant that the memory components tend to have similar numbers of allocated signals and connections. This is a natural consequence of the stepwise assignment technique and is illustrated by the utilisation figures in Figure 6.19. In addition, the computation time required by the ASSIGN algorithm is lower.

A number of systems, including HAL, ESC and SAM, encourage the transfer of signals between memories to reduce the number of locations required. This has not been attempted in either the ASSIGN or CLIQUE methods but manual investigation has shown that it could be used to reduce the memory overhead to an equivalent of 12 registers without increasing the multiplexing.

The results produced using both methods compare favourably with those of other systems. They underline the reductions in interconnection which can be made by using multiple location memories to reduce the number of components and by allowing interconnection factors as much influence over the design process as memory size.

Chapter 7
Worked Examples.

7.1 Introduction.

The aim of including the following worked examples is to highlight the benefits which can be gained by using structural input.

A number of the following worked examples are based around the single data flow graph behaviour of the Differential Equation Example[1] used extensively in previous chapters. It has been used for a number of reasons: it contains no control constructs; it consists of a number of different operation types including subtraction, which is non-commutative; and it makes repeated use of constants. In addition, as it is popular with contemporary systems, such as HAL and SPLICER (CHIPPE), a number of synthesized data paths exist for comparison. The final worked example makes use of a schedule for the Wave Digital Filter[2] which was generated as an exampie in Chapter 5. The Wave Digital Filter behaviour contains a large number of operations and so a large number of data transfers. The schedule used contains only three processors. The resulting high ratio of data transfers to processors gives the opportunity to synthesize a compact, highly used data path. This situation is desirable for exhibiting the potential of a data path allocation scheme. Moreover, the fact that this example is a common benchmark means that data paths synthesized by other systems exist for comparison.

All the run times stated in this chapter were measured on a Sun 3/60 Workstation.

7.2 Reuse of Structure.

A complete data path specification can be input to the system alongside a specification of behaviour. The synthesis system can then be instructed to determine if

[1] Refer to Appendix B.2.
[2] Refer to Appendix B.3.

and how the structure can implement the behaviour.

To demonstrate this, a data path synthesized by HAL [22] for the Differential Equation Example is used as structural input alongside the Differential Equation behaviour. The task of the synthesis system is, therefore, to produce the necessary control information, which will be expressed as a resource-time graph. It must be noted that the HAL data path could not be used if it supported data transfers between memory components.

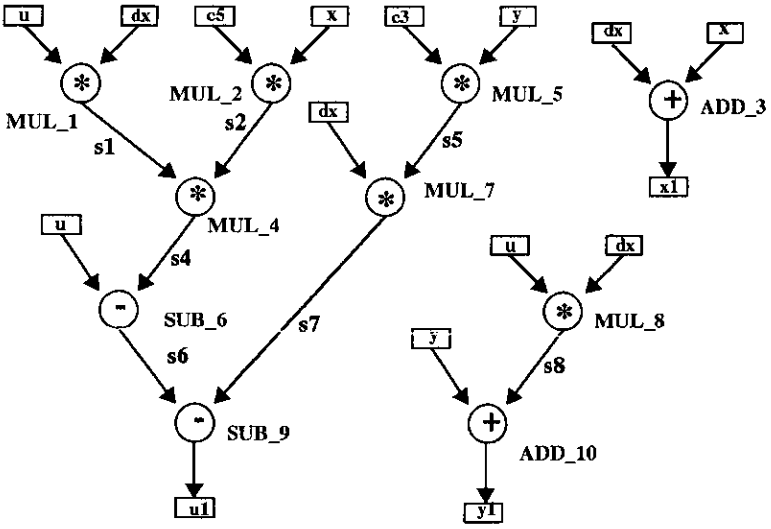


Figure 7.1 The Differential Equation data flow graph behaviour.

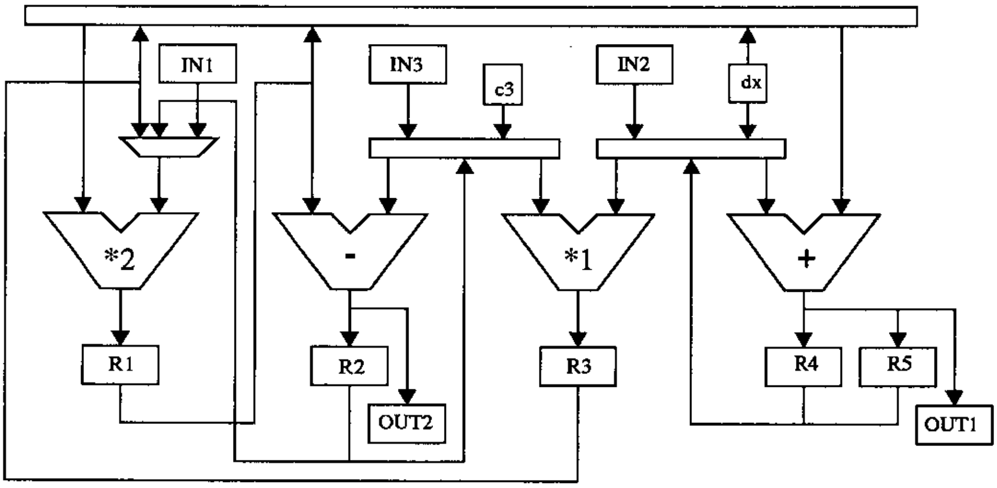


Figure 7.2 HAL data path for Differential Equation.

The behavioural and structural inputs are shown schematically in Figure 7.1 and Figure 7.2 respectively. The textual specifications are provided in Appendix B.2. The first synthesis tool to be invoked is the Mapper. This produces the tentative operation to processor bindings outlined in Figure 7.3. The Scheduler is the next to be applied. Figure 7.4 illustrates the routing task performed in binding operations and signals in the first control step. It is a diagnostic output from the software. All the behaviour's operations are contained within partitions and so the scheduler performs all signal binding in addition to operation scheduling and binding. This means that there is no need for data path allocation. The resultant resource-time graph is shown in Figure 7.5. The original HAL solution uses the same number of control steps. The Mapper and Scheduler, in a combined run, took 8.6s to generate the resource-time graph for this example.

```
mapping HAL

partition
operation mul_8 processors mul_1 true false end
operation add_10 processors add_1 true false end
signals in u y end
 out y1 end
 local s8 y1 end
 constant dx end

partition
operation mul_1 processors mul_2 false true end
operation mul_2 processors mul_1 true false end
operation mul_5 processors mul_1 true false end
operation mul_7 processors mul_2 true false end
operation mul_4 processors mul_2 true false end
operation sub_6 processors sub_1 true false end
operation sub_9 processors sub_1 true false end
signals in u x y end
 out u1 end
 local s1 s2 s5 s7 s4 s6 u1 end
 constant dx c3 end

partition
operation add_3 processors add_1 false true end
signals in x end
 out x1 end
 local x1 end
 constant dx end

finish
```

Figure 7.3 Identified partitions.

```
ROUTING [MUL_2 - 2]
SEARCHING FOR PATH FOR [X - 2] INPUT TO [MUL_1 - 2] RIGHT PORT
 [BUS_3 - 12] to access [IN_2 - 2] OK VALID
SEARCHING FOR PATH FOR [C3 - 8] CONST TO [MUL_1 - 2] LEFT PORT
 [BUS_2 - 11] to access [C3 - 7] OK VALID
SEARCHING FOR PATH FOR [S2 - 10] FROM [MUL_1 - 2] OUT_OF PORT
 [WIRE_4 - 4] [REG_3 - 3] VALID

ROUTING [MUL_1 - 1]
SEARCHING FOR PATH FOR [U - 1] INPUT TO [MUL_2 - 3] RIGHT PORT
 [MUX_1 - 9] to access [IN_1 - 1] OK VALID
SEARCHING FOR PATH FOR [DX - 7] CONST TO [MUL_2 - 3] LEFT PORT
 [BUS_1 - 10] to access [DX - 6] OK VALID
SEARCHING FOR PATH FOR [S1 - 9] FROM [MUL_2 - 3] OUT_OF PORT
 [WIRE_2 - 2] [REG_1 - 1] VALID

ROUTING [MUL_8 - 8]
SEARCHING FOR PATH FOR [DX - 7] CONST TO [MUL_1 - 2] RIGHT PORT
[MUL_8 - 8] DEFFERED

ROUTING [ADD_3 - 3]
SEARCHING FOR PATH FOR [DX - 7] CONST TO [ADD_1 - 1] RIGHT PORT
 [BUS_1 - 10] (occupied by [DX - 7]) to access [DX - 6] matched OK VALID
SEARCHING FOR PATH FOR [X - 2] INPUT TO [ADD_1 - 1] LEFT PORT
 [BUS_3 - 12] (occupied by [X - 2]) to access [IN_2 - 2] matched OK VALID
SEARCHING FOR PATH FOR [X1 - 5] FROM [ADD_1 - 1] OUT_OF PORT
 [WIRE_7 - 7] [OUT_1 - 4] VALID
 [WIRE_6 - 6] [REG_5 - 5] VALID
 [WIRE_5 - 5] [REG_4 - 4] VALID

ROUTING [MUL_5 - 5]
SEARCHING FOR PATH FOR [Y - 3] INPUT TO [MUL_1 - 2] RIGHT PORT
DEFFERED
```

Figure 7.4 Routing signals for operations in the first control step.

STEP	PROCESSORS			
	ADD1	MUL1	MUL2	SUB1
1	ADD_3[1]	MUL_2	MUL_1[1]	
2		MUL_5	MUL_4	
3		MUL_8	MUL_7	SUB_6
4	ADD_10			SUB_9

[1] Reversed inputs.

STEP	MEMORY (WRITE AND READ)									
	REG1		REG2		REG3		REG4		REG5	
	W	R	W	R	W	R	W	R	W	R
1	S1				S2		X1			
2	S4	S1			S5	S2				
3	S7	S4	S6		S8	S5				
4		S7	U1	S6		S8			Y1	

STEP	COMMS			
	MUX1	BUS1	BUS2	BUS3
1	U	DX	C3	X
2	S2	S1	C3	Y
3	S5	DX	U	DX
4		S8	S6	Y

STEP	INPUT AND OUTPUT PORTS				
	IN1	IN2	IN3	OUT1	OUT2
1	U	X		X1	
2		Y			
3			U		
4		Y		Y1	U1

Figure 7.5 Resource-time graph for the behaviour with the HAL data path.

7.3 Iterative Synthesis.

The purpose of this example is to demonstrate the control which the designer can exercise over the synthesis of a structure. Again the Differential Equation behaviour is used.

Starting with an allocation of two multipliers, an adder and a subtractor specified as the input structure, the scheduling and data path allocation tools synthesized the data path shown in Figure 7.6 in 8.3 seconds.

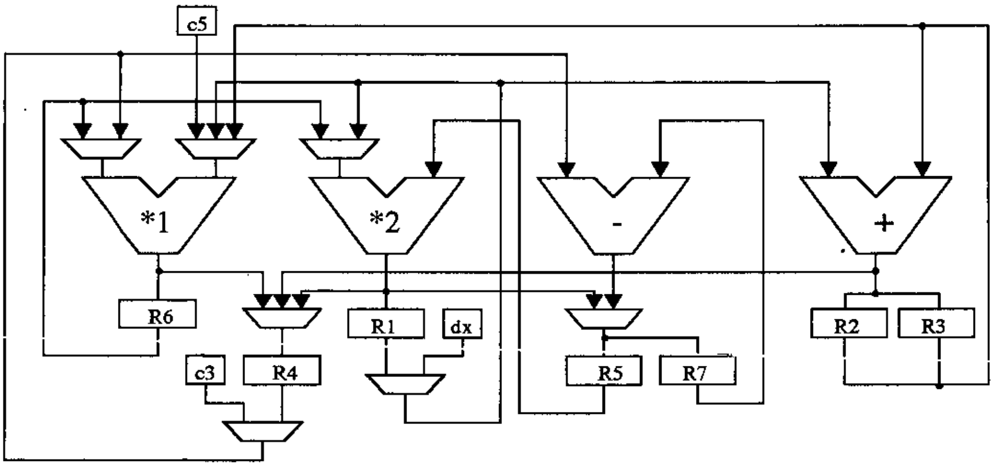


Figure 7.6 First data path generated from Differential Equation behaviour.

STEP	PROCESSORS ADD1	MUL1	MUL2	SUB1
1		MUL_2	MUL_1	
2	ADD_3	MUL_5	MUL_4	
3		MUL_7	MUL_8	SUB_6
4	ADD_10			SUB_9

Figure 7.7 Resource time graph for first data path.

The resource-time graph for the above data path, in Figure 7.7, shows that the first addition is delayed to the second control step. It would appear beneficial to move this operation to the first control step as it has inputs in common with the multiplications performed there. Repeating synthesis with this new schedule yields the improved data path shown in Figure 7.8. Synthesis of this new data path took 8.1 seconds.

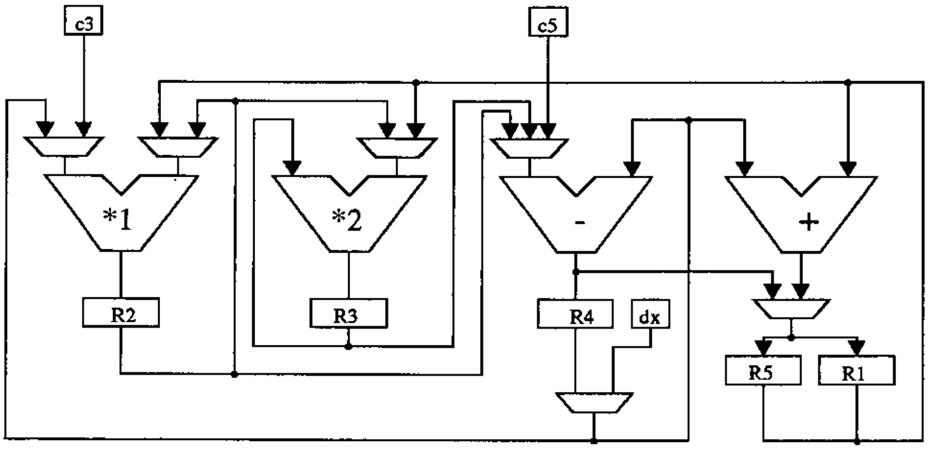


Figure 7.8 Second data path generated after altering the schedule.

By repeating data path allocation with an alteration to the multiplier bindings as shown in Figure 7.9 it is possible to produce the data path in Figure 7.10. This last stage took 7.4 seconds to run.

	PROCESSORS			
STEP	ADD1	MUL1	MUL2	SUB1
1	ADD_3	MUL_2	MUL_1	
2		MUL_5	MUL_4	
3		MUL_8	MUL_7	SUB_6
4	ADD_10			SUB_9

Figure 7.9 Binding alteration on resource-time graph.

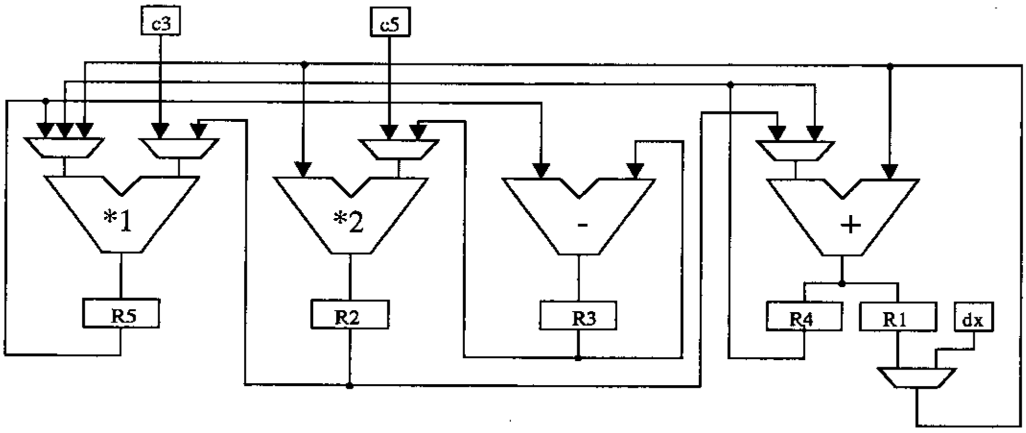


Figure 7.10 Final data path synthesized after alteration of operation bindings.

7.4 Adapting Structure.

It is possible to adapt an existing structure to perform additional behaviours. This is particularly useful when synthesizing a design to implement a behaviour comprising of a number of basic blocks. In this situation it would be desirable to implement mutually exclusive blocks containing common operations on the same hardware.

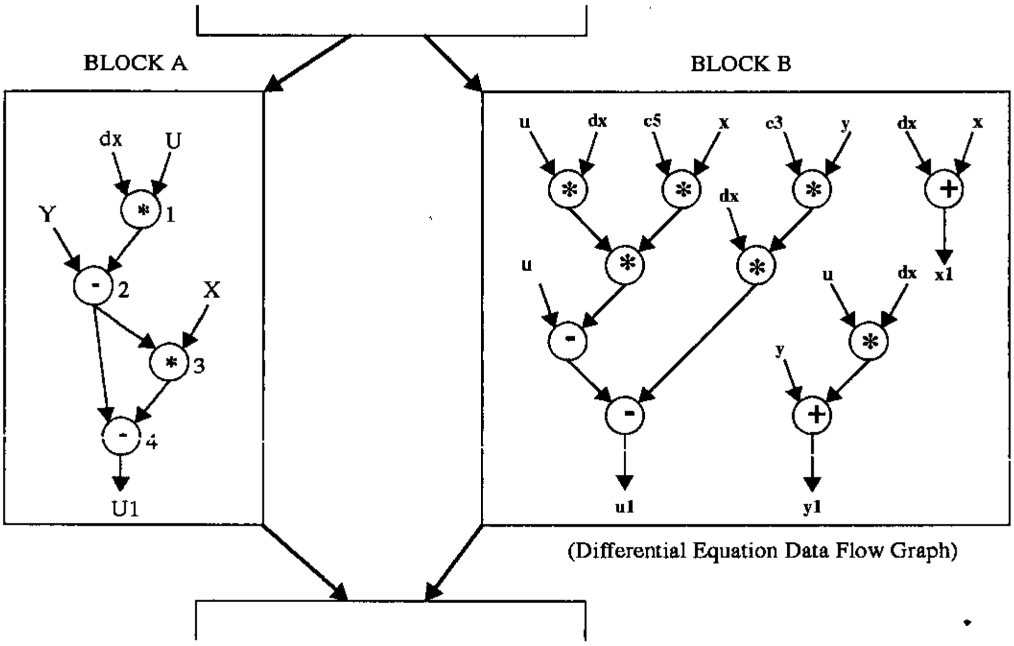


Figure 7.11 Behaviour comprising of two basic blocks.

The behaviour described in Figure 7.11 consists of two such mutually exclusive basic blocks. The structure shown in Figure 7.12 was synthesized for block A.

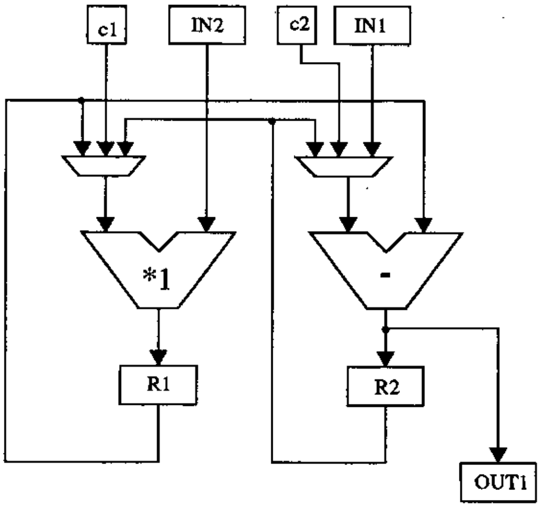


Figure 7.12 Structure A implementing the behaviour of block A.

Structure A was then combined with an additional, unconnected processor allocation comprising of a multiplier and an adder. The resulting structural specification was then used alongside block B to create the structure shown in Figure 7.13.

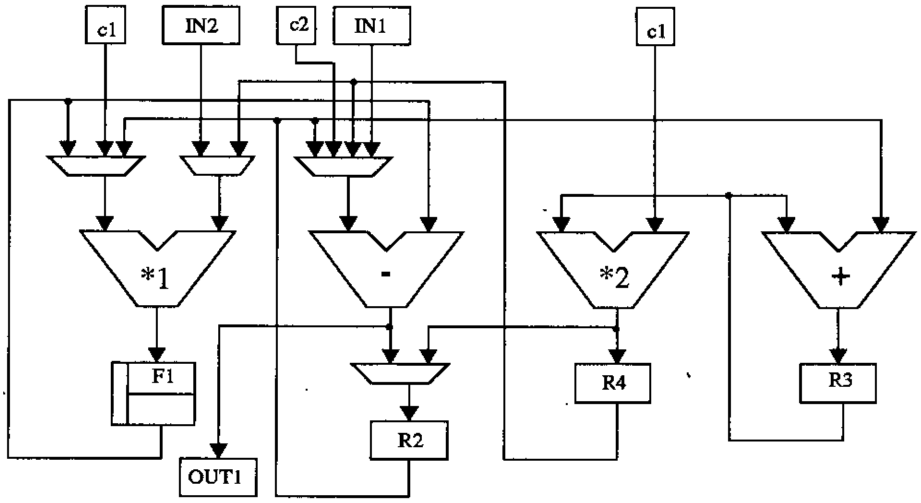


Figure 7.13 Structure B synthesized to implement blocks A and B.

The Mapper identified a partition of block B which could be implemented by structure A. This partition is shown in Figure 7.14. Scheduling produced the resource-time graph shown in Figure 7.15. Mapping and scheduling required a total computation time of 8.6s. The resource-time information was passed with structure A to the Data Path Allocator which added in the extra communications to accommodate signal transfer between structure A and the allocated processors. Data path allocation took 6.5s.

```
SELECTED PARTITIONS :

OPERATIONS : [MUL_1 - 1] -> [MUL_1 - 4] REV
 [MUL_5 - 5] -> [MUL_1 - 4] STD
 [MUL_7 - 7] -> [MUL_1 - 4] STD REV
 [MUL_4 - 4] -> [MUL_1 - 4] STD REV
 [SUB_6 - 6] -> [SUB_1 - 3] STD
 [SUB_9 - 9] -> [SUB_1 - 3] STD

INPUTS : [DX - 7] [U - 1] [Y - 3] [S2 - 10] [S6 - 13]
OUTPUTS : [U1 - 4] [S6 - 13]
CONSTANTS : [DX - 7] [C3 - 8]
LOCAL : [S1 - 9] [S5 - 12] [S7 - 14] [S4 - 11] [S6 - 13] [U1 - 4]
```

Figure 7.14 Partition of block B which can be implemented by structure A.

STEP	PROCESSORS ADD1	 MUL1	 MUL2	 SUB1
1	ADD_3	MUL_5	MUL_8	
2	ADD_10	MUL_1[1]	MUL_2	
3		MUL_4		
4		MUL_7[1]		
5				SUB_6
6				SUB_9

[1] Reversed inputs.

STEP	MEMORY (WRITE AND READ) FILE1 W	 R	 REG2 W	 R	 CONST1 W	 R	 CONST2 W	 R
1	S5				▨	C3	▨	
2	S1				▨	DX	▨	
3	S4	S1			▨		▨	
4	S7	S5			▨		▨	
5		S4	S6		▨		▨	
6		S7	U1	S6	▨		▨	

STEP	COMMS MUX1	 MUX2
1	C3	
2	DX	
3	S1	
4	S5	
5		U
6		S6

STEP	INPUT AND OUTPUT PORTS IN1	 IN2	 OUT1
1		Y	
2		U	
3		S2	
4		DX	
5	U		S6
6			U1

Figure 7.15 Resource-time graph for block B with structure A.

7.5 Influencing the Architecture.

Architecture is a term used to describe the style of structure. It is influenced by the number and type of components and the way in which they are interconnected. As the Data Path Allocator produces the interconnection of memory and communications components surrounding the processors, it is this tool which has the most noticeable impact on the architecture. The schedule and allocation are naturally important in influencing the architecture; previous examples have shown how the designer can exercise control over these tasks.

This section demonstrates how different data path architectures can be synthesized by specifying constraints to the Data Path Allocator. Three data paths are constructed to implement the same schedule of the Wave Digital Filter example. Details of this schedule can be found in Appendix B.3.2.

The first data path, shown in Figure 7.16, was generated without using constraints. It uses a mixture of register files and multiplexers in the solution.

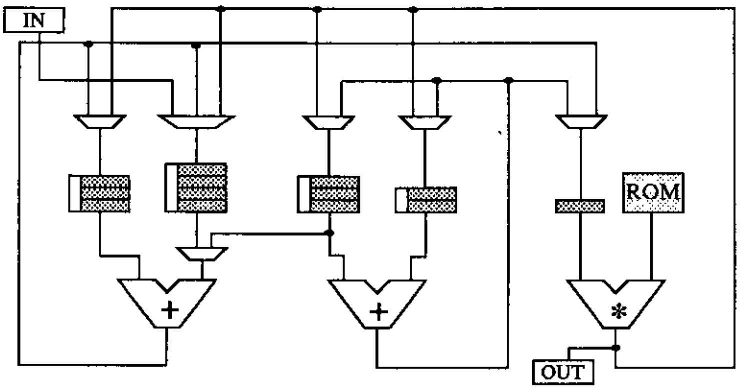


Figure 7.16 Wave Digital Filter data path without constraints.

The second example restricts the position of the register files to the inputs of the processors. Each register file serves one input of a processor. This is similar to the architectural constraint imposed in the Cathedral II system [17]. By specifying the structural input shown in Figure 7.17 the data path shown in Figure 7.18 can be synthesized.

```
structure WDF 6

     memory FILE_1 R_FILE                 memory FILE_4 R_FILE
     capacity 5                           capacity 5
     signals ;                            signals ;
     connection to ADD1_LEFT;             connection to ADD2_RIGHT;
           from ;                               from ;
     ports inputs R0;                     ports inputs R6;
           outputs R1;                          outputs R7;

     memory FILE_2 R_FILE                 memory FILE_5 R_FILE
     capacity 5                           capacity 5
     signals ;                            signals ;
     connection to ADD1_RIGHT;            connection to MUL1_LEFT;
           from ;                               from ;
     ports inputs R2;                     ports inputs R8;
           outputs R3;                          outputs R9;

     memory FILE_3 R_FILE                 memory FILE_6 ROM
     capacity 5                           capacity 5
     signals ;                            signals ;
     connection to ADD2_LEFT;             connection to MUL1_RIGHT;
           from ;                               from ;
     ports inputs R4;                     ports inputs R10;
           outputs R5;                          outputs R11;

                              finish
```

Figure 7.17 Constraining the position of register files.

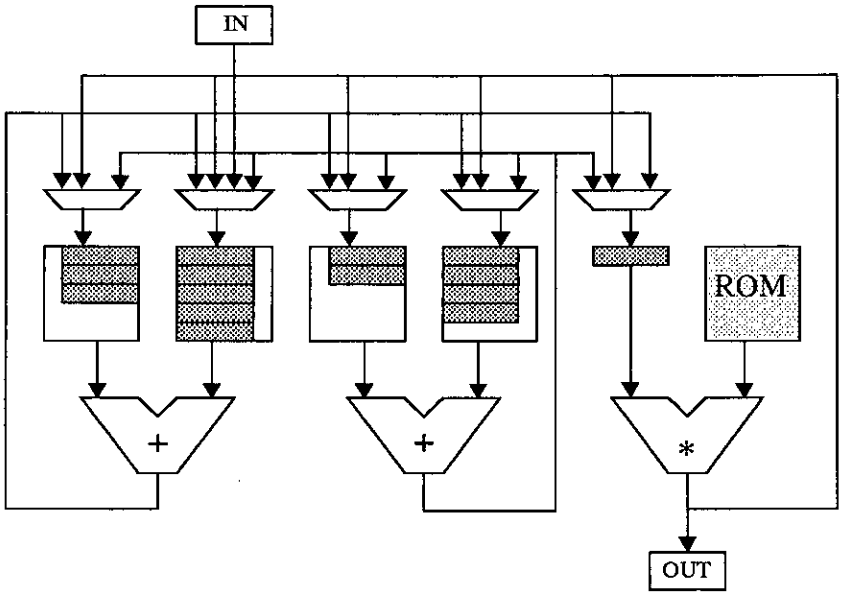


Figure 7.18 Wave Digital Filter data path with register files at processor inputs.

The final example attempts a style similar to that produced by HAL (Figure 6.17). This means using registers instead of register files. To achieve this the input shown in Figure 7.19 is used. The use of single registers provides greater potential for merging multiplexers into buses. this can be seen in the synthesized data path in Figure 7.20.

```
structure WDF 14

      memory FILE_1 ROM
      capacity 1
      signals CT;
      connection to MUL1_RIGHT;
            from CT;
      ports inputs R0;
            outputs R1;

      memory FILE_2 REGISTER
      capacity 1 adapt
      signals ;
      connection to ;
            from ;
      ports inputs R2;
            outputs R3;

      memory FILE_3 REGISTER
      capacity 1 adapt
      signals ;
      connection to ;
            from ;
      ports inputs R4;
            outputs R5;
```

```
      .
      .
      .
memory FILE_12 REGISTER
capacity 1 adapt
signals ;
connection to ;
      from ;

memory FILE_13 REGISTER
capacity 1 adapt
signals ;
connection to ;
      from ;
ports inputs R10;
      outputs R11;

memory FILE_14 REGISTER
capacity 1 adapt
signals ;
connection to ;
      from ;
ports inputs R10;
      outputs R11;
finish
```

Figure 7.19 Preventing the use of register files in the structural specification.

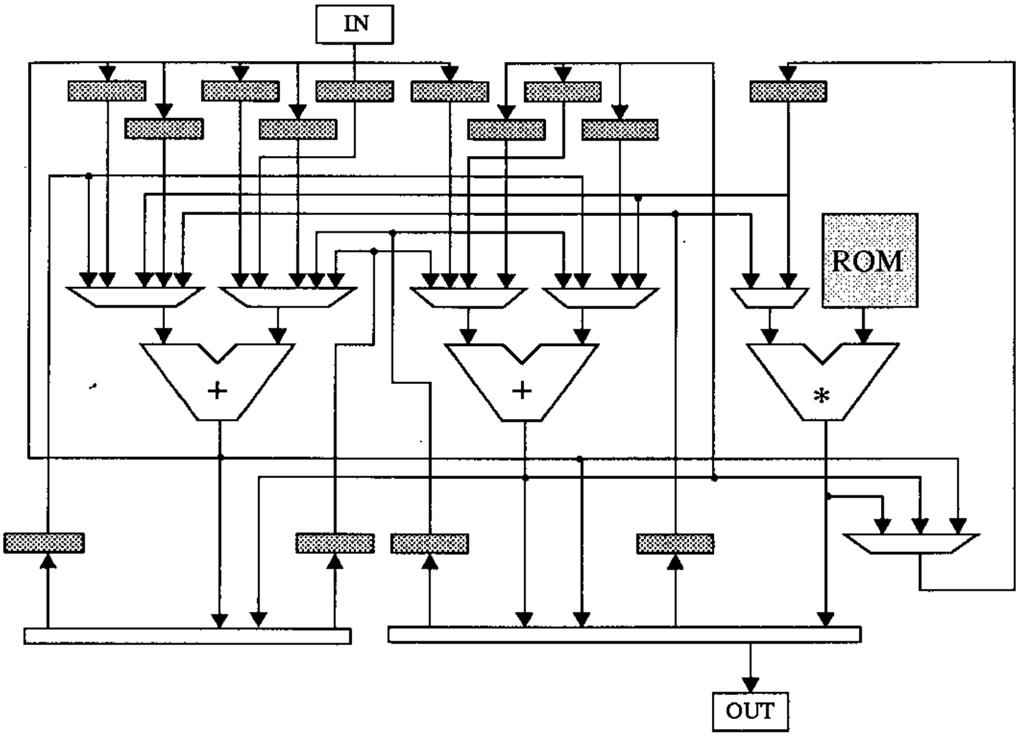


Figure 7.20 Wave Digital Filter data path without register files.

Chapter 8
Conclusion and Future Directions.

8.1 Conclusion.

A high-level behavioural synthesis system which makes use of structural input has been developed. It has been shown to give the designer greater control over structural aspects of the design enabling greater exploration of possible structural solutions. The system presented in this thesis accepts both complete and partial specifications of structure.

The use of complete structure, in the form of a completely connected data path, has been illustrated in a number of examples. The system has been shown to make use of a data path synthesized by the HAL system, in addition to reusing its own designs. It is this capability which makes iterative synthesis possible. Designs failing to meet constraints can be fed back as input allowing modifications to be made. The structural specification need not be capable of implementing the entire behaviour: the system can use the structure to perform parts of the behaviour and synthesize additional structure to implement the remainder.

The ability to specify partial structure, an incomplete interconnection of processing and memory components, enables the designer to direct synthesis towards a suitable architecture. Examples have been shown where the number, type and interconnection of components has been constrained. This is particularly useful for influencing the style of data path interconnection. Simple specifications of partial structure can be used to generate diverse styles. In one example, memory and communications was performed entirely by registers and buses. In another, a single register file was tied to each processor input.

In addition to providing structural input, the designer is given the opportunity to modify the schedule and binding of behavioural elements. Consequently, he or she can dictate the control step in which an operation is scheduled; the processor to which it is bound; and the memory location in which it is stored. This gives the designer greater freedom to explore different structural solutions. The system will indicate errors resulting from the designer's modifications.

The synthesis system does not depend on structural input to produce good solutions. In a solution to the Differential Equation example, based on an allocation of one multiplier and one ALU and compared to other published results, the synthesized data path required 27% fewer multiplexer inputs, 47% fewer wires and 41% fewer point to point connections. For the Wave Digital Filter example, with an allocation of two adders and a pipelined multiplier, the HAL system has synthesized a data path using fewer registers and multiplexers than existing systems. The data path generated by the system reported here uses an additional register but needs half the number of multiplexer inputs. Although there is an additional overhead of four register files, savings are made in wiring: 45% fewer connecting wires, 50% fewer control wires and 36% fewer point to point connections.

8.2 Future Developments with Structural Input.

In the synthesis system using structural input reported in this thesis restrictions have been imposed on the input specifications of behaviour and structure in order to reduce the complexity of the synthesis task. Future developments would be aimed at removing certain restrictions so that a wider range of applications can be tackled.

In common with the majority of synthesis systems, this system cannot handle behaviours which contain control or hierarchy. It is envisaged that a 'basic block' format would be used internally to represent these constructs. Each block is effectively a single data flow graph and so individual blocks can be handled by the present system. It is intended that a synthesis system which handled control and hierarchical constructs would make use of the present system to deal with these individual blocks. The use of a system which accepts structural input is advantageous in this respect as it would be possible to synthesise hardware to perform more than one task.

The absence of 'control' from the behaviour removed the need to handle control structures in the structural input. The ability to handle control structures would not be considered a high priority unless microprocessor-type design was to become a major application. At present the synthesis system reported here cannot recognise the potential for inter-memory data transfer within an input structure: data transfers must always be made between memory and processing components. However, only minor changes to the mapping and scheduling tools would be required to remove this restriction.

8.3 Future Directions in High-Level Synthesis.

As the size and complexity of VLSI circuits increases, there will be a greater need for high-level synthesis to be included in commercial design automation. The primary task of high-level synthesis is to reduce the complexity of the design task presented to engineers enabling them to produce designs more rapidly and with a reduced risk of error. There is also the additional objective of minimising the silicon area of a circuit within the allotted design time. Present synthesis systems have failed to match the quality of designs produced by human designers and so current research is looking into ways of both improving synthesis algorithms and using help from a designer.

The development of more sophisticated synthesis algorithms is receiving more attention. A synthesized structure at the register-transfer level is composed of four classes of component: processing, memory, communications and control. Most synthesis systems dealt with the classes of components in the order stated: an allocation of processors is decided first; a schedule is constructed based on that allocation; memory and communications are then constructed around the processors to support the schedule; and then control is added to ensure the correct sequence of events. However, this synthesis methodology is not well suited to the wide range of applications for VLSI circuits as it focuses heavily on the processing aspect of the design architecture. There are many applications where more suitable architecture can be synthesized if focus is shifted onto one of the other components classes. In image processing applications the memory architecture is often of greater importance whereas in microprocessor design the control and communications architectures usually take priority. To tackle a wide range of applications effectively the system must have a flexible synthesis methodology. A methodology is required which will support a trade-off between processing, memory, communications and control based on the requirements presented by the input behaviour and associated constraints.

Increasingly, researchers are investigating ways of enabling the designer to interact with the synthesis process. A number of systems permit the designer to alter low-level decisions such as the control-step in which an operations is scheduled or the register to which a signal is assigned. However, it is generally accepted that the use of designers is most advantageous in making high-level architectural decisions. There are two key problems with designer interaction which must be addressed: the first is how the system should present the design task and the second is how the designer should communicate his or her advice. The ability to support designer interaction at different levels in the design hierarchy would be a useful development. This would give the designer the opportunity to outline the basic architecture at a high level and then focus on particular aspects. Interaction looks set to be performed in a graphical manner. The designer will most probably be required to specify graphically a partial structure with some tentative relationship to behaviour as a means of guiding or advising the synthesis process.

References

[1] M.H. Lang and P.E. McCormick, *Hierarchical Design Methodologies: A VLSI Necessity*, pp. 123-149 in Advances in CAD for VLSI, Vol. 6, "Design Methodologies", edited by S. Goto, Elsevier Science Publishers (North-Holland) ISBN 0-444-87896-3.

[2] *Silicon Compilation*, edited by D.D. Gajski, Addison-Wesley, 1988, ISBN 0-207-09915-2.

[3] M.C. McFarland, A.C. Parker and R. Camposano, *The High-Level Synthesis of Digital Systems*, in Proc. IEEE, vol. 78 no. 2, pp. 301-318, February 1990.

[4] M.C. McFarland, A.C. Parker and R. Camposano, *Tutorial on High-Level Synthesis*, in Proc. 25th Design Automation Conf., pp. 330-336, June 1987.

[5] G. Borriello and E. Detjens, *High-Level Synthesis: Current Status and Future Directions*, in Proc. 25th Design Automation Conf., pp. 447-482, June 1987.

[6] S.Y. Kung, H.J. Whitehouse and T. Kailith, *VLSI and Modern Signal Processing*, Prentice-Hall, 1985.

[7] David W. Knapp, *Synthesis from Partial Structure*, in Design Methodologies: VLSI and Computer Architecture, pp. 35-51, Elsevier Science Publishers B.V. (North-Holland) IFIP 1989.

[8] David W. Knapp, *Datapath Optimization Using Feedback*, in European Design Automation Conf., pp. 129-134, Amsterdam, February 1991.

[9] David W. Knapp, *Manual Rescheduling and Incremental Repair of Register-Level Datapaths*, in Proc. ICCAD-89, pp. 58-61, Santa Clara, November 1989.

[10] T.D. Friedman and S.C. Yang, *Methods used in an Automatic Logic Generator: ALERT*, IEEE trans. on Computers, vol. C-18, pp.593-614, 1969.

[11] S.W. Director, A.C. Parker, D.P. Siewiorek and D.E. Thomas, *A Design Methodology and Computer Aids for Digital VLSI Systems*, in IEEE trans. on Circuits and Systems, vol. CAS-28, no. 7, pp.634-645, July, 1981.

[12] P. Marwedel, *The MIMOLA Design System: A Design System which spans several Levels*, in Methodologies of Computer System Design, edited by B.D. Shriver, pp.223-237, North-Holland,1985.

[13] H.Kramer and W. Rosenstiel, *System Synthesis using Behavioural Descriptions*, in Proc. European Design Automation Conf., pp. 277-282, Glasgow, March 1990.

[14] P. B. Denyer, D.A. Renshaw, and N.W. Bergmann, *A Silicon Compiler for VLSI Signal Processors*, in Proc. ESSIRC 1982, pp. 215-218.

[15] P.M. Grant, *The DTI-Industry Sponsored Silicon Architectures Research Initiative*, IEE Electronics & Communications Engineering Journal, vol. 2, no. 3 June 1990.

[16] D.M. Grant, *Address Generator Synthesis*, Ph.D. Thesis, University of Edinburgh, October 1991.

[17] H. DeMan, J. Rabaey, P. Six and L. Claesen, *Cathedral II: A Silicon Compiler for Digital Signal Processing*, in IEEE Design and Test, pp.13-25, December 1986.

[18] L. Stok, *Architectural Synthesis and optimization of Digital Systems*, Ph.D. Thesis, ISBN 90-9003966-X.

[19] L. Stok, *Interconnect Optimisation During Data Path Allocation*, in Proc. European Design Automation Conf., pp. 141-145, Glasgow, March 1990.

[20] P.G. Paulin, J.P. Knight and E.F. Girczyc, *HAL: A Multiparadigm Approach to Automatic Data Path Synthesis*, in Proc. 23rd Design Automation Conf., pp. 263-270, June 1986.

[21] P.G. Paulin and J.P. Knight, *Force-Directed Scheduling for the Behavioural Synthesis of ASIC's*, in IEEE trans. on CAD, vol. 8 no. 6, pp. 661-679, June 1989.

[22] P.G. Paulin and J.P. Knight, *Algorithms for High-level Synthesis*, in IEEE Design and Test of Computers, pp.18-31, December 1989.

[23] P.G. Paulin and J.P. Knight, *Scheduling and Binding Algorithms for High-level Synthesis*, in Proc. 26th Design Automation Conf., Las Vegas, pp. 1-6, June 1989.

[24] P.G. Paulin, *High-Level Synthesis of Digital Circuits Using Global Scheduling and Binding Algorithms*, Ph.D. Thesis, Carleton University, January 1988.

[25] B.S. Haroun and M.I. Elmasry, *Architectural Synthesis for DSP Silicon Compilers*, in IEEE trans. on CAD, vol. 8 no. 4, pp. 431-447, April 1989.

[26] A.C. Parker et al., *MAHA: A Program for Data Path Synthesis*, in Proc. 23rd Design Automation Conf., pp. 416-466, June 1986.

[27] R. Jain, K. Kucukcakar, M. Mlinar and A.C. Parker, *Experience with the ADAM Synthesis System*, in Proc. 26th Design Automation Conf., pp. 56-61, June 1989.

[28] K. Kucukcakar and A.C. Parker, *Data Path Tradeoffs using MABAL*, in Proc. 27th Design Automation Conf., pp. 505-510, June 1990.

[29] N. Park and A.C. Parker, *SEHWA: A Program for Synthesis of Pipelines*, in Proc. 23rd Design Automation Conf., pp. 454-460, June 1986.

[30] F.J. Kurdhai and A.C. Parker, *REAL: A Program for Register Allocation*, in Proc. 24th Design Automation Conf., pp. 210-215, June 1987.

[31] R. Jain, A.C. Parker and N. Park, *Module Selection for Pipelined Synthesis*, in Proc. 25th Design Automation Conf., pp.542-547, June 1988.

[32] N. Park and F.J. Kurdhai, *Module Assignment and Interconnect Sharing in Register-Transfer Synthesis of Pipelined Data Paths*, in Proc. ICCAD-89, pp. 16-19, Santa Clara, November 1989.

[33] A. Nagle, R. Cloutier and A. Parker, *Synthesis of Optimal Clocking Schemes*, in Proc. 22th Design Automation Conf., June 1985.

[34] R.J. Cloutier and D.E. Thomas, *The Combination of Scheduling, Allocation, and Mapping in a Single Algorithm*, in Proc. 27th Design Automation Conf., pp. 71-76, June 1990.

[35] J.S. Lis and D.D. Gajski, *Synthesis from VHDL*, in Proc. IEEE Int. Conf. on Computer Design, pp.378-381, 1988.

[36] M. Potkonjak and J. Rabaey, *A Scheduling and Resource Allocation Algorithm for Hierarchical Signal Flow Graphs*, in Proc. 26th Design Automation Conf., pp. 7-12, June 1989.

[37] P.E.R. Lippens et al., *A Silicon Compiler for High Speed Algorithms*, in Proc. European Design Automation Conf., pp. 436-441, Amsterdam, February 1991.

[38] J. Scheichenzuber and W. Grass, *Global Hardware Synthesis from Behavioural Dataflow Descriptions*, in Proc. 27th Design Automation Conf., pp.456-461, June 1990.

[39] R. Camposano, *Structural Synthesis in the Yorktown Silicon Compiler*, in VLSI '87, VLSI Design of Digital Systems, pp.61-72, North-Holland, Amsterdam 1988.

[40] V. Bertis, *The V Compiler: Automating Hardware Design*, in IEEE Design and Test of Computers, pp.8-17, April 1989.

[41] C.J. Tseng et al., *Bridge: A Versatile Behavioral Synthesis System*, in Proc. 25th Design Automation Conf., pp. 415-420, June 1988.

[42] Nam-Sung Woo, *A Global, Dynamic Register Allocation and Binding for A Data Path Synthesis System*, in Proc. 27th Design Automation Conf., pp. 505-510, June 1990.

[43] A.E. Casavant et al., *A Synthesis Environment for Designing DSP Systems*, in IEEE Design and Test of Computers, pp.35-43, April 1989.

[44] K. Wakabayashi and T. Yoshimura, *A Resource Sharing and Control Synthesis Method for Conditional Branches*, in ICCAD-89, pp. 62-65, Santa Clara, November 1989.

[45] T. Tanaka, T. Kobayashi and O. Karatsu, *HARP: Fortran to Silicon*, in IEEE trans. on CAD, vol. 8 no. 6, pp. 649-660, June 1989.

[46] IEEE Standard VHDL Language Reference Manual, IEE std. 1076, 1987.

[47] M.C. McFarland, *Reevaluating the Design Space for Register-Transfer Hardware Synthesis*, in Proc. ICCAD-87, pp. 262-265, Santa Clara, November 1987.

[48] W.F.J. Verhaegh et al., *Improved Force-Directed Scheduling*, in Proc. European Design Automation Conf., pp. 430-435, February 1991.

[49] F. Brewer and D. Gajski, *CHIPPE: A System for Constraint Driven Behavioural Synthesis*, in IEEE trans. on CAD vol. 9 no. 7, pp. 681-695, July 1990.

[50] B.M. Pangrle, *Splicer: A Heuristic Approach to Connectivity Binding*, in Proc. 25th Design Automation Conf., pp. 536-541, June 1988.

[51] B.M. Pangrle and D. Gajski, *Slicer: A State Synthesizer for Intelligent Silicon Compilation*, in Proc. ICCD., pp. 42-45, October1988.

[52] D.E. Thomas, E.M. Dirkes et al., *The System Architect's Workbench*, in Proc. 25th Design Automation Conf., pp.337-343, June 1988.

[53] H. DeMan, *Silicon Compilation for Real Time Signal Processing Systems*, Tutorial on High-Level Synthesis, Part 2, EDAC 1990.

[54] D.W. Knapp and A.C. Parker, *A Unified Representation for Design Information*, in Proc. 7th Int. Symp. on Computer Hardware Description Languages and their Applications, Tokyo, August 1985.

[55] M.C. McFarland and T.J. Kowalski, *Incorporating Bottom-Up Design into Hardware Synthesis*, in IEEE trans. on CAD, vol. 9 no. 9, pp. 938-950, September 1990.

[56] C.A. Papachristou and H. Konuk, *A Linear Program Driven Scheduling and Allocation Method Followed by an Interconnect Optimization Algorithm*, in Proc. 27th Design Automation Conf., pp. 77-83, June 1990.

[57] C. Huang et al., *Data Path Allocation Based on Bipartite Weighted Matching*, in Proc. 27th Design Automation Conf., pp. 499-504, June 1990.

[58] C.H. Gebotys and M.I Elmasry, *A Global Optimization Approach for Architectural Synthesis*, in ICCAD-90, pp.258-261, November 1990.

[59] A. Hemani and A. Postula, *A Neural Net based Self-Organising Scheduling Algorithm*, in Proc. European Design Automation Conf., pp. 136-139, Glasgow, March 1990.

[60] S. Devadas and R. Newton, *Algorithms for Hardware Allocation in Data Path Synthesis,* in IEEE trans. on CAD, vol. 9 no. 7, pp. 768-781, July 1989.

[61] P.J. Neil and P.B. Denyer, *Exploring Design Space Using SAVAGE: A Simulated Annealing based VLSI Architecture GEnerator*, Proc. 33rd Mid-west Symp. on Circuits and Systems, Calgary, August 1990.

[62] E.F. Girczyc, R.J. Buhr and J.P. Knight, *Applicability of a Subset of ADA as an Algorithmic Hardware Description Language for Graph Based Hardware Compilation*, in IEEE trans. on CAD, vol. CAD-4, no. 2, April 1985.

[63] A. Gibbons, *Algorithmic Graph Theory*, Cambridge University Press, 1988, ISBN 0521-28881-9.

[64] C.J. Tseng and D.P. Siewiorek, *Automated Synthesis of Data Paths in Digital Systems*, in IEEE trans. on CAD, vol. CAD-5, no. 3, July 1986.

[65] Aho, Sethi and Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley, ISBN 0-201-10194-7, 1986.

[66] P. Marwedel, *Matching System and Component Behaviour in MIMOLA Synthesis Tools*, in Proc. European Design Automation Conf., pp. 146-155, Glasgow, March 1990.

[67] L.R. Ford and D.R. Fulkerson, *Flows in Networks*, Princeton University Press, 1962.

[68] I.W. Finlay, J.P. Neil and P.B. Denyer, *Filter Synthesis Using Behavioural Design Tools*, in Proc. 16th European Solid State Circuits Conference, pp. 205-208, Grenoble, September 1990.

[69] C.J. Tseng and D.P. Siewiorek, *FACET: A Procedure for Automated Synthesis of Digital Systems*, in Proc. 20th Design Automation Conf., pp. 490-496, 1983.

[70] H. Trickey, *FLAMEL: A High-Level Hardware Compiler*, in IEEE trans. on CAD, vol. CAD-6, no. 2, pp.259-269, March 1987.

[71] D.M. Grant and P.B. Denyer, *Memory, Control and Communications Synthesis for Scheduled Algorithms*, in Proc. 27th Design Automation Conf., pp. 499-504, June 1990.

[72] A. Mignotte and G. Saucier, *Matching Method for Concurrent Operator, Register and multiplexer Allocation*, Synthesis and Simulation Meeting and International Interchange, pp.215-222.

Appendix A
Graph Theoretical Algorithms.

A.1 Clique Partitioning.

This algorithm uses the Bron-Kerbosch method and is coded in Ada.

```
with LIST;


generic
 MAX_VERTICES : in POSITIVE;

package CLIQUE_DEF is


 DEBUG : BOOLEAN := FALSE;



 --------------------------------------------------
 -- DATA TYPES AND STRUCTURES FOR GRAPHS AND SETS --
 --------------------------------------------------

 -- A SET is represented by a Boolean array. In this application we
 -- know that sets will always be of vertices. If we number all vertices
 -- from 1 to N then a vertex m, can be considered a member of a set S
 -- if S(m) is TRUE.
 -- We can represent a graph using an adjacency matrix. This is easily
 -- implemented as an array of sets; each set representing a row of
 -- the adjacency matrix.



 subtype VERTEX_RANGE is INTEGER range 1..MAX_VERTICES;

 type SET_TYPE is array (VERTEX_RANGE) of BOOLEAN;
 type GRAPH_TYPE is array (VERTEX_RANGE) of SET_TYPE;


 type SET_PTR is access SET_TYPE;
 package CLIQUE is new LIST(ITEM => SET_PTR); use CLIQUE;


 ------------------------------------------
 -- SET AND GRAPH FUNCTIONS AND CONSTANTS --
 ------------------------------------------


 EMPTY_SET : constant SET_TYPE := (SET_TYPE’RANGE => FALSE);

 -- For set union (i.e. U ) use logical OR
 -- For set intersection (i.e. n ) use logical AND
 -- For set inversion use logical NOT
 -- For vertex neighbours use index, i.e. neighbours of vertex v in
 -- graph G are given by G(v)
```

```
  function SIZE_OF (SET : in SET_TYPE) return INTEGER;
  -- Returns an integer in the range 0..max_vertices indicating the number
  -- of elements in the specified set.

  function ANY_MEMBER (SET : in SET_TYPE) return INTEGER;
  -- Returns an element a of the specified set. Returns zero (0) if
  -- the set is empty; otherwise returns the index of a TRUE element of the set.

  function UNIT_SET (ELEMENT : in VERTEX_RANGE) return SET_TYPE;
  -- Returns a set where all but the specified elements are set to FALSE; i.e.
  -- a set countaining only the specified member.


  procedure CLEAR (GRAPH : in out GRAPH_TYPE);
  -- Clears the contetnts of the specified graph, i.e. all => FALSE.


  -----------------------------------
  -- CLIQUE PARTITIONING FUNCTIONS --
  -----------------------------------

  generic
  with procedure USE_CLIQUE (THE_CLIQUE : in SET_TYPE);
  -- This routine is used within “find_and_use_all_cliques”. It does whatever
  -- is needed to be done to each clique which is found.

  procedure FIND_AND_USE_ALL_CLIQUES (GRAPH : in GRAPH_TYPE);
  -- The routine which takes a graph and returns all the maximal cliques
  -- in the graph. The cliques will not be in lexicographic order.
  -- The routine will not find 1-cliques.
  -- This routine calls “use_clique” to perform some processing of each
  -- clique - whether it be listing it, manipulating it, or adding it to some
  -- data-structure.



end CLIQUE_DEF;


with TEXT_IO; use TEXT_IO;


package body CLIQUE_DEF is


 ------------------------------------
 -- VARIABLES LOCAL TO THE PACKAGE --
 ------------------------------------

 THE_FOUND_CLIQUES : CLIQUE.LIST;
 CLIQUES_FOUND : NATURAL := 0;


 ------------------------------------------
 -- SET AND GRAPH FUNCTIONS AND CONSTANTS --
 ------------------------------------------


 function SIZE_OF (SET : in SET_TYPE) return INTEGER is
```

```
-- Returns an integer in the range 0..max_vertices indicating the number
-- of elements in the specified set.
COUNT : INTEGER := 0;
begin
     for INDEX in SET_TYPE'RANGE loop
          if SET(INDEX) then COUNT := COUNT + 1; end if;
     end loop;
     return COUNT;
end SIZE_OF;


function ANY_MEMBER (SET : in SET_TYPE) return INTEGER is
-- Returns an element a of the specified set. Returns zero (0) if
-- the set is empty; otherwise returns the index of a TRUE element of the set.
MEMBER : INTEGER := 0;
begin
     for INDEX in SET_TYPE'RANGE loop
          if SET(INDEX)
          then
               MEMBER := INDEX;
               exit;
          end if;
     end loop;
-- IF member = 0 THEN RAISE misuse;
     return MEMBER;
end ANY_MEMBER;


function UNIT_SET (ELEMENT : in VERTEX_RANGE) return SET_TYPE is
-- Returns a set where all but the specified elements are set to FALSE; i.e.
-- a set countaining only the specified member.
RESULT : SET_TYPE := (SET_TYPE'RANGE => FALSE);
begin
     RESULT(ELEMENT) := TRUE;
     return RESULT;
end UNIT_SET;


procedure CLEAR (GRAPH : in out GRAPH_TYPE) is
begin
     GRAPH := (GRAPH_TYPE'RANGE => (SET_TYPE'RANGE => FALSE) );
end CLEAR;


-----------------------------------
-- CLIQUE PARTITIONING FUNCTIONS --
-----------------------------------


procedure FIND_AND_USE_ALL_CLIQUES (GRAPH : in GRAPH_TYPE) is
-- The routine which takes a graph and returns all the maximal cliques
-- in the graph. The cliques will not be in lexicographic order.
-- The routine will not find 1-cliques.
-- This routine calls "use_clique" to perform some processing of each
-- clique - whether it be listing it, manipulating it, or adding it to some
-- data-structure.

VERTICES : SET_TYPE;

procedure EXPAND (THE_COMBINATION,
THE_INTERSECTION,
THE_POSSIBLE : in SET_TYPE;
GRAPH : in GRAPH_TYPE) is
-- This is the routine which actually does all the hard work.
-- It recursively performs a depth-first search of the search-space.
-- The actual algorithm is a modified form of the Bron-Kerbosh method.
S : SET_TYPE;
LOCAL_INTERSECTION : SET_TYPE;
I : INTEGER;
SIZE_OF_S : INTEGER;
COMBINATION : SET_TYPE := THE_COMBINATION;
INTERSECTION : SET_TYPE := THE_INTERSECTION;
POSSIBLE : SET_TYPE := THE_POSSIBLE;
begin
     if DEBUG
     then PUT_LINE(STANDARD_OUTPUT, “Entering EXPAND”);
     end if;

     loop
-- calculate s, using the Simplified B-K method..
     S := POSSIBLE and not ( GRAPH(ANY_MEMBER(INTERSECTION)) );

     SIZE_OF_S := SIZE_OF(S);
     if (SIZE_OF_S = 0)
     then
          if DEBUG then PUT_LINE(STANDARD_OUTPUT, “exiting loop”); end if;
               exit;
          else
          while (SIZE_OF_S > 1) loop
               I := ANY_MEMBER(S);
               S := S and (not UNIT_SET(I));
               SIZE_OF_S := SIZE_OF_S - 1;
               POSSIBLE := POSSIBLE and (not UNIT_SET(I));
               LOCAL_INTERSECTION := INTERSECTION and GRAPH(I);
               if (LOCAL_INTERSECTION = EMPTY_SET) then
                               if GRAPH(I) /= EMPTY_SET
                               then USE_CLIQUE(COMBINATION or UNIT_SET(I));
                               end if;
               else EXPAND(COMBINATION or UNIT_SET(I),
                               LOCAL_INTERSECTION,
                               POSSIBLE and GRAPH(I),GRAPH);
               end if;
          end loop;
          I := ANY_MEMBER(S);
          COMBINATION := COMBINATION or UNIT_SET(I);
          INTERSECTION := INTERSECTION and GRAPH(I);
          if (INTERSECTION = EMPTY_SET)
          then
-- extra check added to try to prevent generation of spurious
-- 1-cliques when using a sparse matrix.
-- use_clique(combination);
--
               if GRAPH(I) /= EMPTY_SET
               then USE_CLIQUE(COMBINATION);
               end if;
               if DEBUG then PUT_LINE(STANDARD_OUTPUT, “exiting loop”);
               end if;
               exit;
          else
               POSSIBLE := POSSIBLE and GRAPH(I);
          end if;
     end if;
 end loop;
 if DEBUG then PUT_LINE(STANDARD_OUTPUT, "exiting EXPAND"); end if;
 end EXPAND;

 begin
     VERTICES := not EMPTY_SET;
     EXPAND(EMPTY_SET, VERTICES, VERTICES, GRAPH);
 end FIND_AND_USE_ALL_CLIQUES;
end CLIQUE_DEF;
```

A.2 Minimum Cost Flow.

This algorithm, also coded in Ada, is an implementation of the Out-of-kilter method described in [67].

```
generic
 N : in POSITIVE; -- number of vertices;

package MINIMAL_COST_FLOW is


 subtype VERTEX_RANGE is INTEGER range 1..N;

 type LABEL_REC is
      record
           L : BOOLEAN;
           S : BOOLEAN;
           V : INTEGER;
           E : INTEGER;
 end record;

 type LABEL_T is array (VERTEX_RANGE) of LABEL_REC;

 type VERTEX_ATTRIBUTE is array (VERTEX_RANGE) of INTEGER;
 type ARC_ATTRIBUTE is array (VERTEX_RANGE,VERTEX_RANGE) of INTEGER;

 procedure MINIMISE_COST (c,l,a : in ARC_ATTRIBUTE;
 FEASIBLE : in out BOOLEAN;
                              f : in out ARC_ATTRIBUTE);

end MINIMAL_COST_FLOW;

with TEXT_IO; use TEXT_IO;
with MISC;
use MISC;
package body MINIMAL_COST_FLOW is

procedure MINIMISE_COST(c,l,a : in ARC_ATTRIBUTE;
 FEASIBLE : in out BOOLEAN;
                              f : in out ARC_ATTRIBUTE) is

package INT_IO is new INTEGER_IO(INTEGER);
use INT_IO;
```

```
-- global variables
k,a_bar : ARC_ATTRIBUTE;
label : LABEL_T;
pi : VERTEX_ATTRIBUTE;
s,t : INTEGER;
OUT_OF_KILTER : BOOLEAN;

procedure UPDATE_KILTER_NUMBERS is


begin
 for x in 1..N loop
      for y in 1..N loop
           if c(x,y) > 0 then -- arc exists
               a_bar(x,y) := a(x,y) + pi(x) - pi(y);
               if a_bar(x,y) > 0 then
                if f(x,y) = l(x,y) then
                               k(x,y) := 0;
 elsif f(x,y) < l(x,y) then
                               k(x,y) := l(x,y) - f(x,y);
 else
                               k(x,y) := a_bar(x,y)*(f(x,y) - l(x,y));
 end if;
 elsif a_bar(x,y) = 0 then
                if f(x,y) < l(x,y) then
                               k(x,y) := l(x,y) - f(x,y);
 elsif f(x,y) > c(x,y) then
                               k(x,y) := f(x,y) - c(x,y);
 else
                               k(x,y) := 0;
 end if;
 else
                if f(x,y) = c(x,y) then
                               k(x,y) := 0;
 elsif f(x,y) < c(x,y) then
                               k(x,y) := a_bar(x,y)*(f(x,y) - c(x,y));
 else
                               k(x,y) := f(x,y) - c(x,y);
 end if;
 end if;
 end if;
 end loop;
 end loop;

end UPDATE_KILTER_NUMBERS;

procedure LABELLING (s : in INTEGER; t : in INTEGER) is

sink,x,d,e,u,v : INTEGER;
SINK_FOUND : BOOLEAN;
NON_BREAKTHROUGH : BOOLEAN;

begin
      if a_bar(s,t) > 0 and f(s,t) < l(s,t) then
      label(t) := (TRUE,TRUE,s,(l(s,t) - f(s,t)) );
      sink := s; x := t;
 elsif a_bar(s,t) = 0 and f(s,t) < l(s,t) then
      label(t) := (TRUE,TRUE,s,(c(s,t) - f(s,t)) );
      sink := s; x := t;
 elsif a_bar(s,t) < 0 and f(s,t) < c(s,t) then
```

```
      label(t) := (TRUE,TRUE,s,(c(s,t) - f(s,t)) );
      sink := s; x := t;
      elsif a_bar(s,t) < 0 and f(s,t) > c(s,t) then
      label(s) := (TRUE,TRUE,-t,(f(s,t) - c(s,t)) );
      sink := t; x := s;
elsif a_bar(s,t) > 0 and f(s,t) > l(s,t) then
      label(s) := (TRUE,TRUE,-t,(f(s,t) - l(s,t)) );
      sink := t; x := s;
elsif a_bar(s,t) = 0 and f(s,t) > c(s,t) then
      label(s) := (TRUE,TRUE,-t,(f(s,t) - l(s,t)) );
      sink := t; x := s;
      end if;

-- start of loop
SINK_FOUND := FALSE;
SEARCH: loop
-- search from x
      label(x).S := TRUE;
      for y in 1 .. N loop
      if c(x,y) > 0 then
           if not label(y).L then
               if a_bar(x,y) > 0 and f(x,y) < l(x,y) then
                e := MIN (label(x).E, l(x,y) - f(x,y));
                label(y) := (TRUE,FALSE,x,e);
                if y = sink then
                                SINK_FOUND := TRUE;
                exit;
end if;
elsif a_bar(x,y) <= 0 and f(x,y) < c(x,y) then
                e := MIN (label(x).E, c(x,y) - f(x,y));
                label(y) := (TRUE,FALSE,x,e);
                if y = sink then
                                SINK_FOUND := TRUE;
                exit;
end if;
end if;
end if;
end if;
      if c(y,x) > 0 then
           if not label(y).L then
               if a_bar(y,x) >= 0 and f(y,x) > l(y,x) then
                e := MIN (label(x).E, f(y,x) - l(y,x));
                label(y) := (TRUE,FALSE,-x,e);
                if y = sink then
                                SINK_FOUND := TRUE;
                exit;
end if;
elsif a_bar(y,x) < 0 and f(y,x) > c(y,x) then
                e := MIN (label(x).E, f(y,x) - c(y,x));
                label(y) := (TRUE,FALSE,-x,e);
                if y = sink then
                                SINK_FOUND := TRUE;
                exit;
end if;
end if;
end if;
end if;
      end loop;

      exit SEARCH when SINK_FOUND;
```

```
     -- find labelled and unscanned node
     NON_BREAKTHROUGH := TRUE;
     for i in 1 .. N loop
          if label(i).L and not(label(i).S) then
               x := i;
               NON_BREAKTHROUGH := FALSE;
               exit;
end if;
end loop;
     exit SEARCH when NON_BREAKTHROUGH;

     end loop SEARCH ;

     -- breakthrough
     if SINK_FOUND then
e := label(sink).E;
          v := sink;
          loop
               u := label(v).V;
               if u < 0 then -- reverse arc
                u := -u;
                f(v,u) := f(v,u) - e;
else
                f(u,v) := f(u,v) + e;
end if;
v := u;
               exit when v = sink;
end loop;

elsif NON_BREAKTHROUGH then
     -- find delta
     d := INTEGER'LAST;
     for x in 1 .. N loop
          for y in 1 .. N loop
               if c(x,y) > 0 then
                if label(x).L and not label(y).L then
                               if a_bar(x,y) > 0 and f(x,y) <= c(x,y) then
                               d := MIN (d,a_bar(x,y));
end if;
                elsif not label(x).L and label(y).L then
                               if a_bar(x,y) < 0 and f(x,y) >= l(x,y) then
                               d := MIN (d,-a_bar(x,y));
end if;
end if;
end if;
end loop;
end loop;
     if d = INTEGER'LAST then
          FEASIBLE := FALSE;
else
     -- update vertex numbers
     for x in 1 .. N loop
          if not label(x).L then
               pi(x) := pi(x) + d;
end if;
end loop;
end if;
end if;
end LABELLING;


begin
 FEASIBLE := TRUE;
 for i in 1 .. N loop
      for j in 1..N loop
           k(i,j) := 0;
 end loop;
      pi(i) := 0;
 end loop;
 loop

 UPDATE_KILTER_NUMBERS;

 OUT_OF_KILTER := FALSE;
 -- find out of kilter arc
 OUTER: for x in 1 .. N loop
      for y in 1 .. N loop
            if c(x,y) > 0 and k(x,y) > 0 then
                 OUT_OF_KILTER := TRUE;
                 s := x; t := y;
                 exit OUTER;
 end if;
 end loop;
 end loop OUTER;
 if OUT_OF_KILTER then

      -- reset labels
      for i in 1 .. N loop
            LABEL(i) := (FALSE,FALSE,0,INTEGER'LAST);
 end loop;

 LABELLING(s,t);
 else
      exit;
 end if;
 if not FEASIBLE then
      exit;
 end if;
 end loop;
end MINIMISE_COST;

end MINIMAL_COST_FLOW;
```

Appendix B
Data for Examples.

B.1 Facet Example.

B.1.1 Behaviour.

This is a description of the data flow graph in [50] and [64]. The types of operations have been changed but the flow graph and schedule bindings are equivalent.

```
network FACET
signal    V1    constant end
signal    V2    local    end
signal    V3    local    end
signal    V4    constantend
signal    V5    constantend

signal    S1    local    end
signal    S2    local    end
signal    S3    local    end
signal    S4    local    end
signal    S5    local    end
signal    S6    local    end


operation MUL_1MULF V2  V3  S1  end
operation MUL_2MULF V1  S1  S2  end
operation MUL_3MULF V2  S2  S3  end
operation MUL_4MULF V2  S3  V3  end
operation ADD_5ADDF S1  V4  S4  end
operation ADD_6ADDF S1  S4  S5  end
operation ADD_7ADDF S5  S6  V2  end
operation DIV_8DIVF S4  V5  S6  end
end FACET
```

B.1.2 Resource-time Graph.

Although the types of processor and operation are different, the schedule and binding are equivalent to [50].

```
RESOURCE - [ADDER_1 - 1]
STEP 1 ----                  RESOURCE - [MULT_1 - 2]      RESOURCE - [DIV_1 - 3]
STEP 2 [ADD_5 - 5]           STEP 1 [MUL_1 - 1]           STEP 1 ----
STEP 3 [ADD_6 - 6]           STEP 2 [MUL_2 - 2]           STEP 2 ----
STEP 4 [ADD_7 - 7]           STEP 3 [MUL_3 - 3]           STEP 3 [DIV_8 - 8]
                             STEP 4 [MUL_4 - 4]           STEP 4 ----
```

B.2 Differential Equation.

B.2.1 Behaviour.

There are two widely used versions of the Differential Equation behaviour. The version used in Section 7.1 is the same as the data flow graph used by HAL [24]. All other examples are based on the version used by SPLICER [50].

```
network SPLICER
signal     U     input    end
signal     X     input    end
signal     Y     input    end
signal     U1    output   end
signal     X1    output   end
signal     Y1    output   end

signal     DX    constant end
signal     C3    constant end
signal     C5    constant end

signal     S1    local     end
signal     S2    local     end
signal     S4    local     end
signal     S5    local     end
signal     S6    local     end
signal     S7    local     end
signal     S8    local     end

operation MUL_1 MUL  U    DX   S1   end
operation MUL_2 MUL  C5   X    S2   end
operation ADD_3 ADD  DX   X    X1   end
operation MUL_4 MUL  S1   S2   S4   end
operation MUL_5 MUL  C3   Y    S5   end
operation SUB_6 SUB  U    S4   S6   end
operation MUL_7 MUL  DX   S5   S7   end
operation MUL_8 MUL  U    DX   S8   end
operation SUB_9 SUB  S6   S7   U1   end
operation ADD_10ADD  Y    S8   Y1   end
end SPLICER
```

```
network HAL
signal     U     input    end
signal     X     input    end
signal     Y     input    end
signal     U1    output   end
signal     X1    output   end
signal     Y1    output   end

signal     DX    constant end
signal     C3    constant end

signal     S1    local     end
signal     S2    local     end
signal     S4    local     end
signal     S5    local     end
signal     S6    local     end
signal     S7    local     end
signal     S8    local     end

operation MUL_1 MUL  U    DX   S1   end
operation MUL_2 MUL  C3   X    S2   end
operation ADD_3 ADD  DX   X    X1   end
operation MUL_4 MUL  S1   S2   S4   end
operation MUL_5 MUL  C3   Y    S5   end
operation SUB_6 SUB  U    S4   S6   end
operation MUL_7 MUL  DX   S5   S7   end
operation MUL_8 MUL  U    DX   S8   end
operation SUB_9 SUB  S6   S7   U1   end
operation ADD_10ADD  Y    S8   Y1   end
end HAL
```

B.2.2 Structures.

Three different input specifications of structure are used in Chapter 7 with Differential Equation behaviours. Structure HAL is a specification of a data path from [22] and is used in Section 7.1. Structures STRUCT_A and STRUCT_B are used in Section 7.3.

```
structure HAL

processor ADD_1
 type ADD
 adapt FALSE
 functions ADD;
 ports
 right a0 from BUS_1;
 left a1 from BUS_3;
 out a2 to
WIRE_5,WIRE_6,WIRE_7;
 allocation ;

processor MUL_1
 type MUL
 adapt FALSE
 functions MUL;
 ports
 right m0 from BUS_3;
 left m1 from BUS_2;
 out m2 to WIRE_4;
 allocation ;

processor MUL_2
 type MUL
 adapt FALSE
 functions MUL;
 ports
 right m3 from MUX_1;
 left m4 from BUS_1;
 out m5 to WIRE_2;
 allocation ;

processor SUB_1
 type SUB
 adapt FALSE
 functions SUB;
 ports
 right s0 from WIRE_1;
 left s1 from BUS_2;
 out s2 to WIRE_3,
WIRE_8;
allocation ;

memory REG_1
 type REG
 adapt FALSE
 capacity 1
 ports
 in r0 from WIRE_2;
 out r1 to WIRE_1,BUS_1;
 allocation ;

memory REG_2
 type REG
 adapt FALSE
 capacity 1
 ports
 in r2 from WIRE_3;
 out r3 to MUX_1,BUS_2;
 allocation ;

memory REG_3
 type REG
 adapt FALSE
 capacity 1
 ports
 in r4 from WIRE_4;
 out r5 to MUX_1,BUS_1;
 allocation ;

memory REG_4
 type REG
 adapt FALSE
 capacity 1
 ports
 in r6 from WIRE_5;
 out r7 to BUS_3;
 allocation ;

memory REG_5
 type REG
 adapt FALSE
 capacity 1
 ports
 in r8 from WIRE_6;
 out r9 to BUS_3;
 allocation ;

memory DX
 type CONST
 adapt FALSE
 capacity 1
 ports
 in c0 from ;
 out c1 to BUS_1,BUS_3;
 allocation ;

memory C3
 type CONST
 adapt FALSE
 capacity 1
 ports
 in c2 from ;
 out c3 to BUS_2;
 allocation ;

net WIRE_1
 type WIRE
 adapt FALSE
 from r1 ;
 to s0;

net WIRE_2
 type WIRE
 adapt FALSE
 from m5 ;
 to r0;

net WIRE_3
 type WIRE
 adapt FALSE
 from s2 ;
 to r2;

net WIRE_4
 type WIRE
 adapt FALSE
 from m2 ;
 to r4;

net WIRE_5
 type WIRE
 adapt FALSE
 from a2 ;
 to r6;

net WIRE_6
 type WIRE
 adapt FALSE
 from a2 ;
 to r8;

net WIRE_7
 type WIRE
 adapt FALSE
 from a2 ;
 to OUT_1;

net WIRE_8
 type WIRE
 adapt FALSE
 from s2 ;
 to OUT_2;

net MUX_1
 type MUX
 adapt FALSE
 from r5, r3,IN_1 ;
 to m3;

net BUS_1
 type BUS
 adapt FALSE
 from r5, r1, c1;
 to m4, a0;

net BUS_2
 type BUS
 adapt FALSE
 from c3, r3, IN_3;
 to s1, m1;

net BUS_3
 type BUS
 adapt FALSE
 from c1, r7, r8, IN_2 ;
 to a1, m0 ;

io_port IN_1
 type INPUT
 adapt FALSE
 from ;
 to MUX_1;
 allocation ;

io_port IN_2
 type INPUT
 adapt FALSE
 from ;
 to BUS_3;
 allocation ;

io_port IN_3
 type INPUT
 adapt FALSE
 from ;
 to BUS_2;
 allocation ;

io_port OUT_1
 type OUTPUT
 adapt FALSE
 from WIRE_7;
 to ;
 allocation ;

io_port OUT_2
 type OUTPUT
 adapt FALSE
 from WIRE_8;
 to ;
 allocation ;
```

```
structure STRUCT_B

processor ADD_1
 type ADD
 adapt FALSE
 functions ADD;
 ports
 right n14 from ;
 left n15 from ;
 out n16 to ;
 allocation ;

processor MUL_2
 type MUL
 adapt FALSE
 functions MUL;
 ports
 right n17 from ;
 left n18 from ;
 out n19 to ;
 allocation ;

processor SUB_1
 type SUB
 adapt FALSE
 functions SUB;
 ports
 right n3 from WIRE_3;
 left n4 from MUX_2;
 out n5 to WIRE_4,WIRE_5;
 allocation ;

processor MUL_1
 type MUL
 adapt FALSE
 functions MUL;
 ports
 right n0 from WIRE_1;
 left n1 from MUX_1;
 out n2 to WIRE_2;
 allocation ;

memory REG_1
 type REG
 adapt FALSE
 capacity 1
 ports
 in n6 from WIRE_2;
 out n7 to WIRE_3,MUX_1;
 allocation ;

memory REG_2
 type REG
 adapt FALSE
 capacity 1
 ports
 in n10 from WIRE_4;
 out n11 to MUX_2,MUX_1;
 allocation ;

memory CONST_1
 type CONST
 adapt FALSE
 capacity 1
 ports
 in n8 from ;
 out n9 to MUX_1;
 allocation ;

memory CONST_2
 type CONST
 adapt FALSE
 capacity 1
 ports
 in n12 from ;
 out n13 to MUX_2;
 allocation ;

net WIRE_1
 type WIRE
 adapt FALSE
 from IN_2 ;
 to n1;

net WIRE_2
 type WIRE
 adapt FALSE
 from n2 ;
 to n6;

net WIRE_3
 type WIRE
 adapt FALSE
 from n7 ;
 to n3;

net WIRE_4
 type WIRE
 adapt FALSE
 from n5 ;
 to n10;

net WIRE_5
 type WIRE
 adapt FALSE
 from n5 ;
 to OUT_1;

net MUX_1
 type MUX
 adapt FALSE
 from n7, n9, n11 ;
 to n0;

net MUX_2
 type MUX
 adapt FALSE
 from n11, n13, IN_1 ;
 to n4;

io_port IN_1
 type INPUT
 adapt FALSE
 from ;
 to MUX_2;
 allocation ;

io_port IN_2
 type INPUT
 adapt FALSE
 from ;
 to WIRE_1;
 allocation ;

io_port OUT_1
 type OUTPUT
 adapt FALSE
 from WIRE_5;
 to ;
 allocation ;

finish
```

```
structure DIFF2

processor SUB_1
 type SUB
 adapt FALSE
 functions SUB;
 ports
 right n6 from MUX_3;
 left n7 from MUX_5;
 out n8 to MUX_1, WIRE_5;
 allocation ;

processor MUL_1
 type MUL
 adapt FALSE
 functions MUL;
 ports
 right n0 from WIRE_1;
 left n1 from WIRE_2;
 out n2 to MUX_1, MUX_2;
 allocation ;

processor MUL_2
 type MUL
 adapt FALSE
 functions MUL;
 ports
 right n3 from MUX_4;
 left n4 from WIRE_3;
 out n5 to MUX_2;
 allocation ;

memory REG_1
 type REG
 adapt FALSE
 capacity 1
 ports
 in n9 from MUX_1;
 out n10 to MUX_3,MUX_4,-
MUX_5;
 allocation ;

memory REG_2
 type REG
 adapt FALSE
 capacity 1
 ports
 in n11 from MUX_2;
 out n12 to WIRE_3, MUX_5,
MUX_3;
 allocation ;

memory CONST_1
 type CONST
 adapt FALSE
 capacity 1
 ports
 in n13 from ;
 out n14 to WIRE_1;
 allocation ;

memory CONST_2
 type CONST
 adapt FALSE
 capacity 1
 ports
 in n15 from ;
 out n16 to MUX_4;
 allocation ;

net WIRE_1
 type WIRE
 adapt FALSE
 from n14 ;
 to n0;

net WIRE_2
 type WIRE
 adapt FALSE
 from IN_1 ;
 to n1;

net WIRE_3
 type WIRE
 adapt FALSE
 from n12 ;
 to n4;

net WIRE_5
 type WIRE
 adapt FALSE
 from n8 ;
 to OUT_1;

net MUX_1
 type MUX
 adapt FALSE
 from n2, n8 ;
 to n9;

net MUX_2
 type MUX
 adapt FALSE
 from n2, n5 ;
 to n11;

net MUX_3
 type MUX
 adapt FALSE
 from n10, n12, IN_2 ;
 to n6;

net MUX_4
 type MUX
 adapt FALSE
 from n16, n10 ;
 to n3;

net MUX_5
 type MUX
 adapt FALSE
 from n12, n10 ;
 to n7;

io_port IN_1
 type INPUT
 adapt FALSE
 from ;
 to WIRE_2;
 allocation ;

io_port IN_2
 type INPUT
 adapt FALSE
 from ;
 to MUX_3;
 allocation ;

io_port OUT_1
 type OUTPUT
 adapt FALSE
 from WIRE_5;
 to ;
 allocation ;

finish
```

B.2.3 Resource-time graphs.

Resource-time graph used in Section 6.5.2 and Section 7.3

```
RESOURCE - [ADD_1 - 1]              RESOURCE - [MUL_1 - 3]
STEP 1 [ADD_3 - 3]                  STEP 1 [MUL_1 - 1]
STEP 2 ----                         STEP 2 [MUL_4 - 4]
STEP 3 ----                         STEP 3 [MUL_7 - 7]
STEP 4 [ADD_10 - 10]                STEP 4 ----

RESOURCE - [SUB_1 - 2]              RESOURCE - [MUL_2 - 4]
STEP 1 ----                         STEP 1 [MUL_2 - 2]
STEP 2 ----                         STEP 2 [MUL_5 - 5]
STEP 3 [SUB_6 - 6]                  STEP 3 [MUL_8 - 8]
STEP 4 [SUB_9 - 9]                  STEP 4 ----
```

Resource-time graph used in Section 6.5.2 for Figure 6.16.

```
RESOURCE - [ALU_1 - 1]              RESOURCE - [MUL_1 - 2]
STEP 1 [ADD_3 - 3]                  STEP 1 [MUL_2 - 2]
STEP 2 ----                         STEP 2 [MUL_1 - 1]
STEP 3 ----                         STEP 3 [MUL_5 - 5]
STEP 4 ----                         STEP 4 [MUL_4 - 4]
STEP 5 [SUB_6 - 6]                  STEP 5 [MUL_7 - 7]
STEP 6 [SUB_9 - 9]                  STEP 6 [MUL_8 - 8]
STEP 7 [ADD_10 - 10]                STEP 7 ----
```

B.3 Wave Digital Filter.

B.3.1 Behaviour.

This behaviour is taken from [6]. It is a description of the same data flow graph used by [18][24][52][50].

```
network WDF
signal A input  end

signal B local  end
signal C local  end
signal D local  end
signal E local  end
signal F local  end
signal G local  end
signal H local  end

signal CT local constant end

signal S1 local  end
signal S2 local  end
signal S3 local  end
signal S4 local  end
signal S5 local  end
signal S6 local  end
signal S7 local  end
signal S8 local  end
signal S9 local  end
signal S10 local  end
signal S11 local  end
signal S12 local  end
signal S13 local  end
signal S14 local  end
signal S16 local  end
signal S17 local  end
signal S18 local  end
signal S19 local  end
signal S20 local  end
signal S21 local  end
signal S22 local  end
signal S23 local  end
signal S24 local  end
signal S25 local  end
signal S26 local  end
signal S27 local  end
signal S29 local  end

operation ADDF_1 ADDF A B S1 end
```

```
 operation ADDF_2 ADDF G H S2 end
 operation ADDF_3 ADDF C S1 S3 end
 operation ADDF_4 ADDF D S3 S4 end
 operation ADDF_5 ADDF S4 S2 S5 end
 operation MULF_6 MULF S5 CT S6 end
 operation MULF_7 MULF S5 CT S7 end
 operation ADDF_8 ADDF S3 S6 S8 end
 operation ADDF_9 ADDF S2 S7 S9 end
 operation ADDF_10 ADDF S3 S8 S10 end
 operation ADDF_11 ADDF S9 S2 S11 end
 operation MULF_12 MULF S10 CT S12 end
 operation ADDF_13 ADDF S5 S8 S13 end
 operation MULF_14 MULF S11 CT S14 end
 operation ADDF_15 ADDF S13 S9 D end
 operation ADDF_16 ADDF S1 S12 S16 end
 operation ADDF_17 ADDF S14 H S17 end
 operation ADDF_18 ADDF S16 S8 S18 end
 operation ADDF_19 ADDF S9 S17 S19 end
 operation ADDF_20 ADDF S1 S16 S20 end
 operation ADDF_21 ADDF F S19 S21 end
 operation MULF_22 MULF S20 CT S22 end
 operation ADDF_23 ADDF E S18 S23 end
 operation MULF_24 MULF S21 CT S24 end
 operation ADDF_25 ADDF H S17 S25 end
 operation ADDF_26 ADDF A S22 S26 end
 operation MULF_27 MULF S23 CT S27 end
 operation ADDF_28 ADDF F S24 F end
 operation MULF_29 MULF S25 CT S29 end
 operation ADDF_30 ADDF S16 S26 B end
 operation ADDF_31 ADDF S19 F G end
 operation ADDF_32 ADDF E S27 E end
 operation ADDF_33 ADDF S17 S29 H end
 operation ADDF_34 ADDF S18 E C end
end
```

B.3.2 Resource-time graphs.

18 control-step schedule.

```
RESOURCE - [ADD_1 - 1]          RESOURCE - [ADD_2 - 2]
STEP 1 [ADDF_2 - 2]             STEP 1 [ADDF_1 - 1]
STEP 2 ----                     STEP 2 [ADDF_3 - 3]
STEP 3 ----                     STEP 3 [ADDF_4 - 4]
STEP 4 [ADDF_5 - 5]             STEP 4 ----
STEP 5 ----                     STEP 5 ----
STEP 6 ----                     STEP 6 ----
STEP 7 [ADDF_9 - 9]             STEP 7 [ADDF_8 - 8]
STEP 8 [ADDF_11 - 11]           STEP 8 [ADDF_10 - 10]
STEP 9 ----                     STEP 9 [ADDF_13 - 13]
STEP 10 ----                    STEP 10 [ADDF_15 - 15]
STEP 11 [ADDF_17 - 17]          STEP 11 [ADDF_16 - 16]
STEP 12 [ADDF_25 - 25]          STEP 12 [ADDF_20 - 20]
STEP 13 [ADDF_19 - 19]          STEP 13 [ADDF_18 - 18]
STEP 14 [ADDF_21 - 21]          STEP 14 [ADDF_23 - 23]
STEP 15 [ADDF_33 - 33]          STEP 15 [ADDF_26 - 26]
STEP 16 ----                    STEP 16 [ADDF_30 - 30]
STEP 17 [ADDF_28 - 28]          STEP 17 [ADDF_32 - 32]
STEP 18 [ADDF_31 - 31]          STEP 18 [ADDF_34 - 34]
```

```
SRESOURCE - [MUL_1 - 3]         RESOURCE - [MUL_2 - 4]
STEP 1 ----                     STEP 1 ----
STEP 2 ----                     STEP 2 ----
STEP 3 ----                     STEP 3 ----
STEP 4 ----                     STEP 4 ----
STEP 5 [MULF_7 - 7]             STEP 5 [MULF_6 - 6]
STEP 6 ----                     STEP 6 ----
STEP 7 ----                     STEP 7 ----
STEP 8 ----                     STEP 8 ----
STEP 9 [MULF_14 - 14]           STEP 9 [MULF_12 - 12]
STEP 10 ----                    STEP 10 ----
STEP 11 ----                    STEP 11 ----
STEP 12 ----                    STEP 12 ----
STEP 13 [MULF_29 - 29]          STEP 13 [MULF_22 - 22]
STEP 14 ----                    STEP 14 ----
STEP 15 [MULF_24 - 24]          STEP 15 [MULF_27 - 27]
STEP 16 ----                    STEP 16 ----
STEP 17 ----                    STEP 17 ----
STEP 18 ----                    STEP 18 ----
```

19 control-step schedule.

```
RESOURCE - [ADD_1 - 1]      RESOURCE - [ADD_2 - 2]      RESOURCE - [MUL_1 - 3]
STEP 1 [ADDF_2 - 2]         STEP 1 [ADDF_1 - 1]         STEP 1 ----
STEP 2 ----                 STEP 2 [ADDF_3 - 3]         STEP 2 ----
STEP 3 ----                 STEP 3 [ADDF_4 - 4]         STEP 3 ----
STEP 4 [ADDF_5 - 5]         STEP 4 ----                 STEP 4 ----
STEP 5 ----                 STEP 5 ----                 STEP 5 [MULF_6 - 6]
STEP 6 ----                 STEP 6 ----                 STEP 6 [MULF_7 - 7]
STEP 7 ----                 STEP 7 [ADDF_8 - 8]         STEP 7 ----
STEP 8 [ADDF_9 - 9]         STEP 8 [ADDF_10 - 10]       STEP 8 ----
STEP 9 [ADDF_11 - 11]       STEP 9 [ADDF_13 - 13]       STEP 9 [MULF_12 - 12]
STEP 10 [ADDF_15 - 15]      STEP 10 ----                STEP 10 [MULF_14 - 14]
STEP 11 ----                STEP 11 [ADDF_16 - 16]      STEP 11 ----
STEP 12 [ADDF_17 - 17]      STEP 12 [ADDF_20 - 20]      STEP 12 ----
STEP 13 [ADDF_25 - 25]      STEP 13 [ADDF_18 - 18]      STEP 13 ----
STEP 14 [ADDF_19 - 19]      STEP 14 [ADDF_23 - 23]      STEP 14 [MULF_22 - 22]
STEP 15 [ADDF_21 - 21]      STEP 15 [ADDF_26 - 26]      STEP 15 [MULF_27 - 27]
STEP 16 [ADDF_33 - 33]      STEP 16 [ADDF_30 - 30]      STEP 16 [MULF_24 - 24]
STEP 17 ----                STEP 17 [ADDF_32 - 32]      STEP 17 [MULF_29 - 29]
STEP 18 [ADDF_28 - 28]      STEP 18 [ADDF_34 - 34]      STEP 18 ----
STEP 19 [ADDF_31 - 31]      STEP 19 ----                STEP 19 ----
```

21 Control Steps.

```
RESOURCE - [ADD_1 - 1]      RESOURCE - [ADD_2 - 2]      RESOURCE - [MUL_1 - 3]
STEP 1 [ADDF_2 - 2]         STEP 1 [ADDF_1 - 1]         STEP 1 ----
STEP 2 ----                 STEP 2 [ADDF_3 - 3]         STEP 2 ----
STEP 3 ----                 STEP 3 [ADDF_4 - 4]         STEP 3 ----
STEP 4 [ADDF_5 - 5]         STEP 4 ----                 STEP 4 ----
STEP 5 ----                 STEP 5 ----                 STEP 5 [MULF_6 - 6]
STEP 6 ----                 STEP 6 ----                 STEP 6 [MULF_6 - 6]
STEP 7 ----                 STEP 7 [ADDF_8 - 8]         STEP 7 [MULF_7 - 7]
STEP 8 [ADDF_13 - 13]       STEP 8 [ADDF_10 - 10]       STEP 8 [MULF_7 - 7]
STEP 9 [ADDF_9 - 9]         STEP 9 ----                 STEP 9 [MULF_12 - 12]
STEP 10 [ADDF_15 - 15]      STEP 10 [ADDF_11 - 11]      STEP 10 [MULF_12 - 12]
STEP 11 ----                STEP 11 [ADDF_16 - 16]      STEP 11 [MULF_14 - 14]
STEP 12 [ADDF_18 - 18]      STEP 12 [ADDF_20 - 20]      STEP 12 [MULF_14 - 14]
STEP 13 [ADDF_23 - 23]      STEP 13 [ADDF_17 - 17]      STEP 13 [MULF_22 - 22]
STEP 14 [ADDF_19 - 19]      STEP 14 [ADDF_25 - 25]      STEP 14 [MULF_22 - 22]
STEP 15 [ADDF_21 - 21]      STEP 15 [ADDF_26 - 26]      STEP 15 [MULF_27 - 27]
STEP 16 ----                STEP 16 [ADDF_30 - 30]      STEP 16 [MULF_27 - 27]
STEP 17 [ADDF_32 - 32]      STEP 17 ----                STEP 17 [MULF_24 - 24]
STEP 18 [ADDF_34 - 34]      STEP 18 ----                STEP 18 [MULF_24 - 24]
STEP 19 [ADDF_28 - 28]      STEP 19 ----                STEP 19 [MULF_29 - 29]
STEP 20 [ADDF_31 - 31]      STEP 20 ----                STEP 20 [MULF_29 - 29]
STEP 21 ----                STEP 21 [ADDF_33 - 33]      STEP 21 ----
```