Simulated Annealing Based Datapath Synthesis

John Paul Neil

A thesis submitted for the degree of
Doctor of Philosophy
to the Faculty of Science of the University of Edinburgh.

1994

Abstract

The behavioural synthesis procedure aims to produce optimised register-transfer level datapath descriptions from an algorithmic problem definition, normally expressed in a high-level programming language. The procedure can be partitioned into a number of subtasks linked by a serial synthesis flow. Graph theoretic algorithms can be used to provide solutions to these subtasks. Many of these techniques, however, belong to a class of algorithm for which there is no exact solution computable in polynomial time. To overcome this problem, heuristics are used to constrain the solution space. The introduction of heuristics can cause the algorithm to terminate in a local cost minimum.

This thesis supports a global formulation of the behavioural synthesis problem. An algorithm which can avoid local minima, simulated annealing, forms the basis of the synthesis system reported.

A modular software system is presented in support of this approach. A novel data structure enables multiple degrees of optimisation freedom within the datapath solution space. Synthesis primitives, tightly coupled to a solution costing mechanism directed towards the prevalent datapath implementation technologies, form the core of the system. The software is exercised over small and large-scale synthesis benchmarks. The synthesis paradigm is extended by the provision of optimisation routines capable of supporting the generation of functional pipelines.

Declaration of Originality

Except where noted in the text, the research recorded in this thesis is the original and sole work of the author.

John Paul Neil
May 1994

Acknowledgements

I wish to thank Professor Peter Denyer for providing a stimulating environment in which to work, and for his advice and direction during the course of the research recorded in this thesis. The financial support for this work was provided by the Science and Engineering Research Council and The University of Edinburgh; I am indebted to both institutions.

I acknowledge the support of the engineers seconded to the Silicon Architectures Research Initiative, notably Martin Ryder, David Mallon, Colin Carruthers, Ross Kennedy and Douglas Chisholm. My fellow research students, Douglas Grant and Iain Finlay have also been a source of great encouragement.

I would like to thank Dr. Edward McDonnell for proofreading this thesis.

Finally, my heartfelt thanks go to Janet for her love and support during the preparation of this thesis.

Contents

To my mother and father, who have scaled the mountains with me.

1 Introduction

The last decade has seen enormous leaps in the complexity of Integrated Circuits (ICs). Correspondingly, the design process associated with these circuits has grown lengthy and expensive.

Whereas the first ICs had typical complexities ranging from a few tens to a few hundreds of discrete logic structures, current devices have complexities ranging from many thousands to, in the case of dynamic Random Access Memory (RAM) designs, millions of logic structures. This increased complexity has arisen through advances in process technology and shrinking mask geometries. Moore's law [Moore79] states that the number of discrete components that can be placed on a single substrate will double every eighteen months. This affords engineers greater opportunities to increase the levels of integration within their designs, towards the level of total system integration on a single substrate.

A good example of this effort is the evolution of a single-chip fingerprint recognition system [Anderson91] developed at the University of Edinburgh. In its original conception, the system was composed of two Application Specific Integrated Circuits (ASICs) executing the recognition algorithm, an imaging subsystem, and two printed circuit boards containing interface and support logic. Subsequently, the recognition algorithms together with the imaging array were successfully integrated onto a single substrate [Anderson93].

While advances in process technology will ultimately be limited by fundamental physical constraints, the potential increase in functionality offered by shrinking mask geometries will act as a significant incentive to systems engineers to realise their designs in silicon for some time to come.

1.1 ASIC Design Domains

An ASIC may be specified in four separate domains:

(i) **Behavioural.** Here the specification captures the *functionality* of the final circuit at a high level, but contains no details of the circuit *implementation.* The ideal behavioural specification is a number of sentences describing function of the circuit, but most behavioural specifications are captured using a high level programming language, such as VHDL, Verilog, C, or ADA.

(ii) **Macroarchitecture.** This specification describes the circuit in terms of the functional blocks required to implement the circuit behaviour (communally known as a *datapath*), their interconnection, and the sequencing of the target algorithm on the datapath. This specification level is known as the *Register-Transfer Level* (RTL).

(iii) **Microarchitecture.** The circuit microarchitecture models the diversity of implementation of the macroarchitecture at a gate level. For example, an adder block can be implemented in a number of different forms, e.g. ripple, carry lookahead, carry propagate, Manchester chain, and so on.

(iv) **Physical.** The physical specification of the circuit takes place at the transistor level. The gate level description of the design is translated to a transistor level netlist containing sizing and connectivity information.

These domains suggest an ordering of the ASIC design process, with the functionality of a device captured first, followed by datapath macro- and microarchitectural development, prior to any physical implementation. Historically, however, successive generations of Computer Aided Design (CAD) tools have operated first in the physical domain, followed by the datapath microarchitectural level, and finally at the datapath macroarchitectural level. Current research efforts are focused on the development of CAD tools operating in the behavioural domain. Automatic translation between the design domains is termed *synthesis*.

1.2 Synthesis Tool Evolution

The first synthesis tools were targeted at the physical domain. During the design cycle of early ICs, each mask layer was specified separately, with engineers undertaking all the layout effort.

The advent of regular layout technologies such as standard cell logic and gate array structures, together with the development of physical design tools for block placement and routing, have enabled circuit engineers to concentrate on the development of a small number of primitive cells. These primitives can then be instanced many times to form the logic structures required. In the case of cell-based designs, logic gates are

defined in terms of fundamental building blocks (or *standard cells*). For gate array-based designs, logic structures are defined as the metalisation layers required to connect pre-instantiated gates.

Translation between the microarchitectural level and the physical domain exploits these regular layout technologies and the software tools supporting them. Logic gates can be specified in terms of Boolean equations [Brayton84] or entered in a schematic form and optimised prior to compilation into layout.

Translation between the macroarchitectural level and the physical level yielded the first software tools to be called *Silicon Compilers* [Johan78]. Typically, a functional circuit description was mapped into a template datapath architecture. The FIRST silicon compiler [Denyer82] was targeted towards bit-serial architectures. Similarly, research efforts at IMEC and Leuven University yielded the CATHEDRAL series of silicon compilers. The range of architectural templates included bit-serial (CATHEDRAL-I [Clae86]), communicating multiprocessor (CATHEDRAL-II [DeMan88]) and bit-sliced datapath (CATHEDRAL-III [Note88]). The quality of solution achieved in these systems is dependent on the mapping between the input algorithm and the architectural template.

1.3 Behavioural Synthesis Tools

Current CAD research is directed towards achieving a successful translation between the behavioural and macroarchitectural domains. Interest in high-level synthesis is motivated by the advantages of such an approach:

(i) Specification of an ASIC at a high level allows rapid functional verification. This should be contrasted with the gate-level verification required when a design is captured at the ASIC microarchitecture level.

(ii) A high-level approach allows engineers to rapidly evaluate design alternatives at an early stage in the ASIC project cycle. Indeed Broderson [Broder89] argues that the true gains of realising a system in silicon arise through the selection of appropriate silicon architectures.

(iii) High-level tools are *correct by construction.* The use of behavioural and logic synthesis to realise a design removes the possibility of errors introduced during manual translation between the design domains outlined above.

(iv) High-level synthesis tools represent an *enabling technology.* By abstracting the design process away from the physical domain, the need for detailed circuit implementation knowledge is removed. Thus, ASIC technology becomes more accessible to system-level engineers.

The advantages offered by the use of behavioural synthesis tools address the problems imposed by market pressures described previously.

McFarland [McFarl88] defines the behavioural synthesis procedure:

> *"The synthesis task is to take a specification of the behaviour required of a system and a set of constraints and goals to be satisfied, and to find a (datapath) structure that implements the behaviour while satisfying the goals and constraints."*

A typical behavioural specification is given in figure 1.1. A datapath optimised to offer a minimum area solution is shown in figure 1.2.

```
procedure DIFF_EQ (X, U, Y : in out INTEGER; A : in INTEGER) is

  X1,Y1,U1 : INTEGER;
  DX,C3 : CONSTANT;

begin

  while (X < A) loop
     X1 := X + DX;
     U1 := U - (3*X*U*DX) - (3*Y*DX);
     Y1 := Y + (U*DX);
     X := X1; U := U1; Y := Y1;
  end loop;

end DIFF_EQ;
```

Figure 1.1 Behavioural specification.

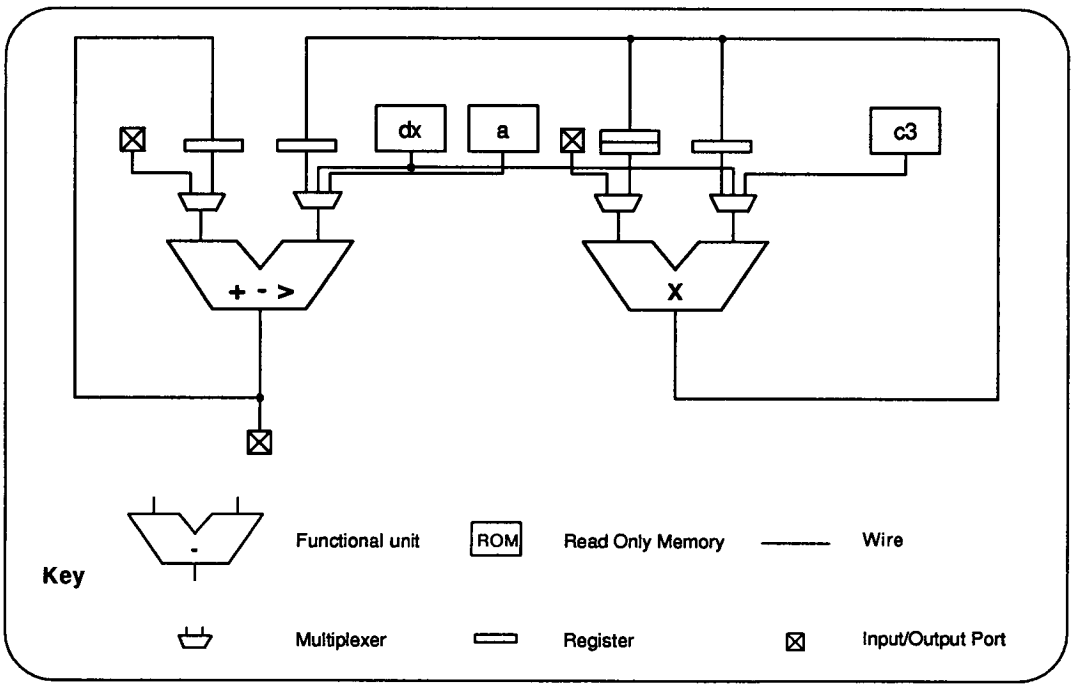


Figure 1.2 Datapath synthesised from specification shown in figure 1.1.

The behavioural synthesis procedure may be partitioned into the following subtasks:

(i) Translation of the behavioural description into some suitable

intermediate format. This data representation must retain all the characteristics of the original description, whilst presenting the data in a suitable and recognisable form. It is at this stage that compiler-like optimisations take place, such as loop unrolling (partial or complete) and dead code elimination [Aho86].

(ii) **Operator scheduling.** This corresponds to the assignment of a control step value to each operation. A *control step* (c-step) corresponds to a single state of a finite state machine.

(iii) **Processor allocation.** This step assigns individual operations to execute on particular hardware resources. These resources may be specialised (e.g. adders, multipliers and subtractors) or generic ALU-type structures.

(iv) **Memory allocation.** An appropriate set of memory components must be synthesised to store intermediate results and input/output values.

(v) **Interconnect optimisation.** A communications infrastructure is synthesised which connects all modules allocated in steps (iii) and (iv), and completes the datapath topology.

(vi) **Controller synthesis.** This final phase generates a controller capable of sequencing all the operations and data transfers as defined in stages (ii) - (v).

The research presented in this thesis is concerned with datapath synthesis techniques (i.e. tasks (ii) - (iv)).

1.4 Summary of Research

The partitioning of the behavioural synthesis procedure presented above, and the serial synthesis flow it suggests, has allowed optimisation techniques drawn from algorithmic graph theory [Gibbons87] to be used to solve the subtasks. In many cases, however, the solution algorithm belongs to a class of algorithms for which there is no

exact solution computable in polynomial time [Garey79]. The introduction of heuristics overcomes this problem by constraining the solution search space. The use of heuristics can, however, lead to the algorithm terminating in a local cost minimum, thus degrading the quality of the solution.

A global formulation of the behavioural synthesis procedure which avoids local minima is proposed in this thesis. A solution algorithm, *simulated annealing*, drawn from the field of statistical physics is used to control the search through the datapath solution space.

A suite of software tools is presented in support of this approach. Innovative features include:

- A multiple-plane data model which provides support for global optimisation of tasks (ii) - (v) above.
- A solution quality assessment procedure targeted towards the production of datapaths amenable to implementation in standard cell and gate array technologies.
- A cost multiplier mechanism which allows the engineer to influence the overall datapath architecture without *direct* synthesis intervention.
- The application of external system constraints is enabled through the use of synthesis directives or *pragmas*[1].

Further, a general purpose algorithm for the generation of functional pipelines, which is suited to a simulated annealing-based implementation is presented.

1. This term was first introduced in [Clae86].

1.5 Thesis Structure

Chapter 2 develops two related models, pre-requisite to any discussion of the behavioural synthesis procedure or its subtasks. Following a comparison of two differing models of behaviour, the semantics and syntax of a suitable representation are described. Similarly, a structural notation for datapaths is also presented. A review of controller and timing issues concludes the chapter.

The solution techniques for the synthesis subtasks described in section 1.3 above are considered in chapter 3. The scheduling strategies reviewed are split into three categories: iterative, state transformational and integer linear programming (ILP). The merits and demerits of each are discussed. Allocation techniques for the solution of subtasks (iii) - (v) in the synthesis flow above are then reviewed. Graph theoretic algorithms form the core material presented in this section, which concludes with a brief discussion of expert system-based approaches to allocation.

Chapter 4 proposes a global approach to the behavioural synthesis problem and suggests a formulation of the synthesis task as a combinatorial optimisation problem. Local and global search techniques are presented, and the termination of local search algorithms in a non-optimum state is demonstrated. The simulated annealing algorithm is introduced as a global technique capable of escaping from local minima. The cooling schedule techniques associated with the simulated annealing algorithm are then reviewed. A brief survey of NP-complete algorithms used as solution techniques for the synthesis subtasks concludes this section. This is followed by a review of a simulated annealing-based high-level synthesis system developed at the University of California at Berkeley.

Chapter 5 introduces a set of software tools (the SAVAGE system) capable of

transforming a behavioural description into an optimised RTL datapath structure. Central to the method presented is a multi-dimensional data structure capable of supporting simulated annealing-based optimisation in each plane. A review of the core synthesis routines precedes the presentation of the optimisation primitives used to generate the datapath solution states. A solution quality assessment procedure is developed, and a novel system allowing the engineer to assign cost multipliers to the individual cost function components is presented. The synthesis tools are then exercised on a small-scale benchmark.

Two large-scale comparative studies are presented in chapter 6. The first, a Fast Discrete Cosine Transform kernel, permits a comparison between the synthesis method reported in this thesis and an interactive synthesis tool developed at the University of Edinburgh. The second benchmark, a Wave Digital Filter, allows comparisons to be drawn between the simulated annealing-based approach and a greater number of behavioural synthesis tools.

A general-purpose algorithm for the generation of functional pipelines, and its integration into the synthesis system reported herein, forms the core of chapter 7. Structural and functional pipelining techniques are reviewed and metrics for the assessment of pipeline performance defined prior to the introduction of the pipelining algorithm. Extensions to the SAVAGE system are described, and datapaths from the large-scale benchmarks of chapter 7 are re-synthesised to demonstrate the approach.

Finally, chapter 8 summarises the work presented in this thesis, and suggests extensions to the research.

2 Behavioural and Structural Models for Synthesis

This Chapter describes two related models germane to the behavioural synthesis task. In the first instance, a suitable behavioural representation must be developed. This representation must accurately capture the functionality of the source text, while its internal form should remain amenable to manipulation by the various synthesis procedures.

A notation that describes datapath structure must also be developed. To remain technology independent, the notation should not contain any physical attributes, but should at the same time contain enough meaningful information to allow the synthesis tools to optimise the datapath topology. This implies a level of abstraction between the RTL notation and the compiled gate-level description of the datapath. For example, rather than measure delay through an adder in terms of nanoseconds, the delay can be abstracted into multiples of the system clock (or *control steps*). Similarly, area

measurement does not take place in square microns, rather in notional gate equivalents. By building this level of abstraction into the structural model, it can be ensured that the output RTL descriptions are technology independent.

Following a brief discussion on two alternative representations of behaviour, the data flow semantics and syntax used throughout the work described in this thesis are presented. Closely linked to that is the development of a structural notation which supports the abstraction described above.

As a brief interlude, a naive mapping (or *binding*) between behavioural and structural domains based on a rudimentary set of axioms is considered. This binding contains no sequencing or control information. The synthesis of a suitable controller lies outside the scope of the work described in this thesis, but a simple control and timing model is developed and presented.

2.1 Two Alternative Behavioural Representations

Consider the addition of two integer variables, a and b. This addition can be represented by the following equation:

$$x = a + b$$

where x is the integer variable containing the result of the addition. The lexical convention observed here is known as *infix* notation. Here, the operands (a and b) are separated by the operator (+), thus:

$$x <op> (a <op> b)$$

The equality symbol in the equation represents the assignment operator. (The brackets noted above indicate the operator precedence).

Similarly, the addition of *a* and *b* could be represented in *prefix* notation. Here, the operator precedes both operands (in the case of non-commutative operations, the ordering of operands becomes important). Thus:

$$= x + a\ b$$

Both notations are valid and functionally equivalent. The difference between infix and prefix notation hints at the alternative representations of behaviour.

Consider the following, more complex, expression[1] in infix notation.

$$ul = u - 3xudx - 3ydx$$

A graphical representation can be created by parsing through the expression left to right, observing operator precedence and replacing operators with vertex tokens whose input arcs correspond to the operands associated with the operator. The graphical representation for the infix expression is shown in figure 2.1 below.

Consider the same expression in prefix notation:

$$= ul - u - 3xudx\ 3ydx$$

Here the graphical representation is derived by parsing left to right through the

1. This expression forms the basis of Paulin's seminal differential equation example [Paulin89b], and is used as an illustration throughout this thesis.

expression creating a tree-like structure whose vertices are the operators, with the operands represented by leaf nodes. This is shown in figure 2.2.

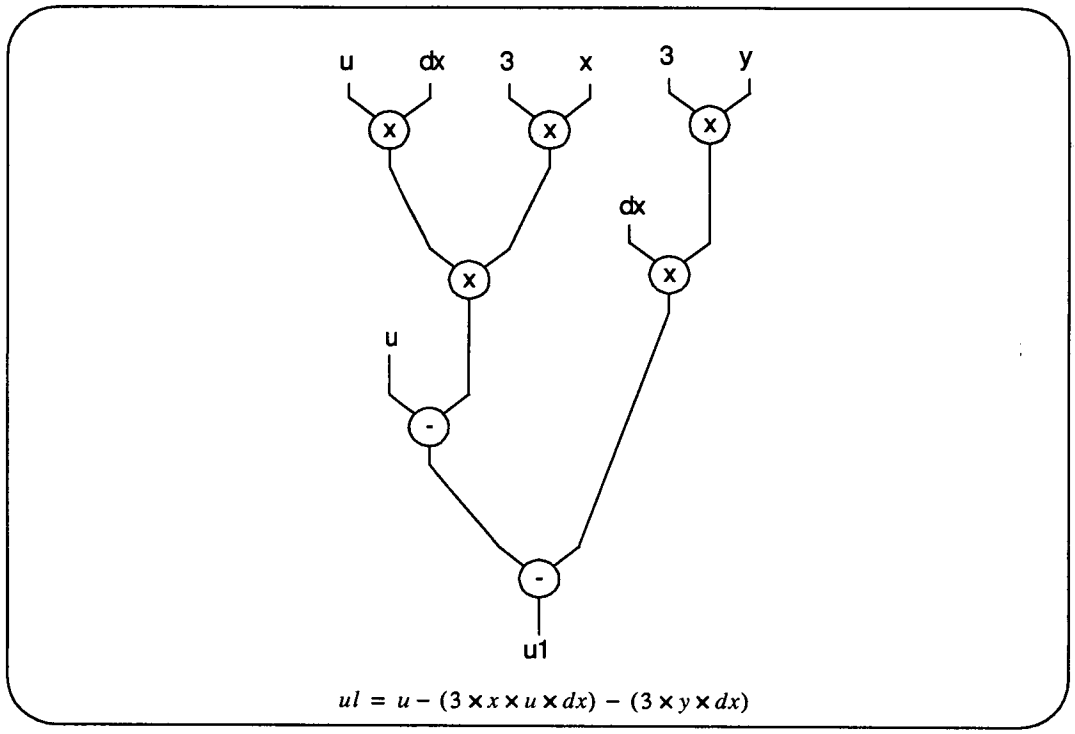


Figure 2.1 Infix-based graphical representation.

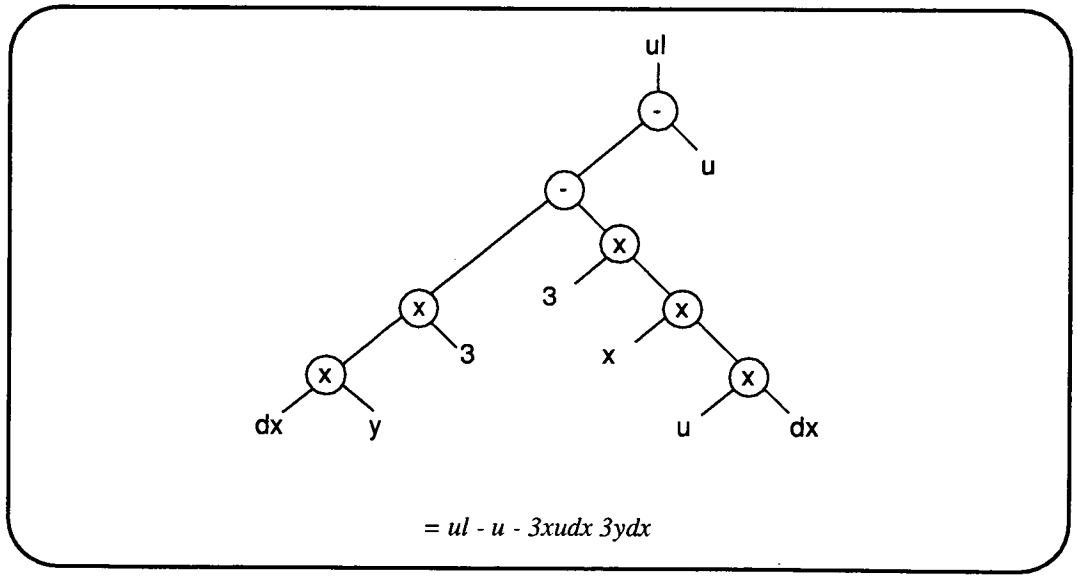


Figure 2.2 Prefix-based graphical representation.

In both representations, the assignment operator has been removed, as assignment is implicit.

These two representations are functionally equivalent. It is the axiom used to construct each that differentiates between the two. The first axiom is the more intuitive (as a result of infix notation predominating in mathematics teaching), while the second forms the basis of expression evaluation within many programming language compilers.

The first graphical representation is a *data flow graph*, with the second known as a *parse tree*.

2.1.1 Discussion

At an intuitive level, a flow graph provides the most straightforward engineering representation. Indeed, Broderson [Broder89] argues that most electronic engineers begin the design process with a basic data flow representation. For an in-depth discussion of the flow graph, readers are directed to [Orail86]. From an automated standpoint, however, it can be argued that parse trees generated directly from the input expressions are also a suitable representation of behaviour.

For many synthesis tool designers, the choice of behavioural representation is decided by the availability of compilers for the input language. Bearing this in mind, language issues are now discussed.

Language Issues

In selecting a high level language for behavioural input, a tool designer has to choose between using a standard procedural language or a specialised language which uses a more explicit data flow representation. Probably the single most important difference

between the two types is the use of multiple assignment of a single variable. Consider the descriptions of the filter shown in figures 2.3 and 2.4.

```
function SINGLE_ASSIGNMENT_FILTER (COEFF : in INTEGER[1:8]) is
  ACCUM[0:8] : integer;
  COUNT : integer;

  begin
     ACCUM[0] := 0;
     for(COUNT=1;COUNT<=8;COUNT++)
        ACCUM[COUNT] := ACCUM[COUNT-1] + COEFF[COUNT];
     return ACCUM[8];
end SINGLE_ASSIGNMENT_FILTER;
```

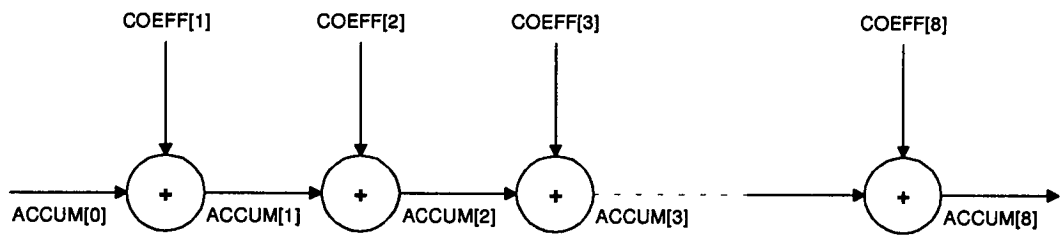


Figure 2.3 Single variable assignment.

The description shown in figure 2.3 permits single assignment of variables only. That is, each intermediate result is specified explicitly in the loop body (hence the requirement for an array of result variables). This description is an accurate representation of true data flow. Probably the best known of all data flow languages, SILAGE [Hilfin84], has been tailored specifically for Digital Signal Processing (DSP) applications. Bit delays, decimation and interpolation constructs are present in the language syntax.

```
function MULTIPLE_ASSIGNMENT_FILTER (COEFF : in INTEGER[1:8]) is
  ACCUM : integer;
  COUNT : integer;

  begin
     ACCUM := 0;
     for(COUNT=1;COUNT<=8;COUNT++)
        ACCUM := ACCUM + COEFF[COUNT];
     return ACCUM;
end MULTIPLE_ASSIGNMENT_FILTER;
```

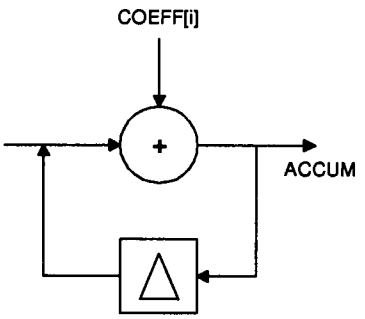


Figure 2.4 Multiple variable assignment.

Against the 'naturalness' of the signal flow description, procedural languages such as PASCAL, C and ADA enjoy a well-established user base. Advances in compiler technology, specifically loop unrolling, have enabled multiple assignment, as shown in figure 2.4, to be detected and replaced with single assignment constructs, preserving the natural data flow within the intermediate representation. The use of multiple *vs.* single assignment then becomes a matter of designer preference when using a procedural language. As a caveat, however, it should be noted that this detection and replacement strategy is only valid for loop structures with finite limits.

The intuitive nature of the data flow description, together with the availability of a data flow based compiler (SLANG [Sey89]) during the course of the work described in this thesis ultimately prompted the adoption of a data flow model. The semantics and syntax of the model are now formalised.

2.2 Data Flow Semantics

A data flow graph is defined as a tuple, $D=(V,E)$, where $V=\{V_1,V_2,..,V_n\}$ is a finite set of vertices and $E = \{E_{ij};\ i,j = 1,2,..,n\}$ is a set of edges connecting elements of V. E is a set of directed arcs; data cannot be consumed before it is produced. Thus, E contains both precedence constraints and connectivity information. The vertices within V can be partitioned into two main types:

Transformational: This vertex type performs a transformation on the input data. It is most readily associated with the arithmetic and logical operators found within the instruction set of a typical microprocessor. Further, associated with each vertex is a tuple, $P=(I,O)$ where $I = \{I_1,..,I_n\}$ is a set of input ports and $O=\{O_1,..,O_n\}$ is a set of output ports. These port sets provide a mechanism whereby the commutative law can be exercised during optimisation.

Boundary: This class of vertex is a synchronisation mechanism allowing external data to be input to and output from the data flow graph.

This partial definition allows the specification of straight line code segments, as shown in figure 2.5.

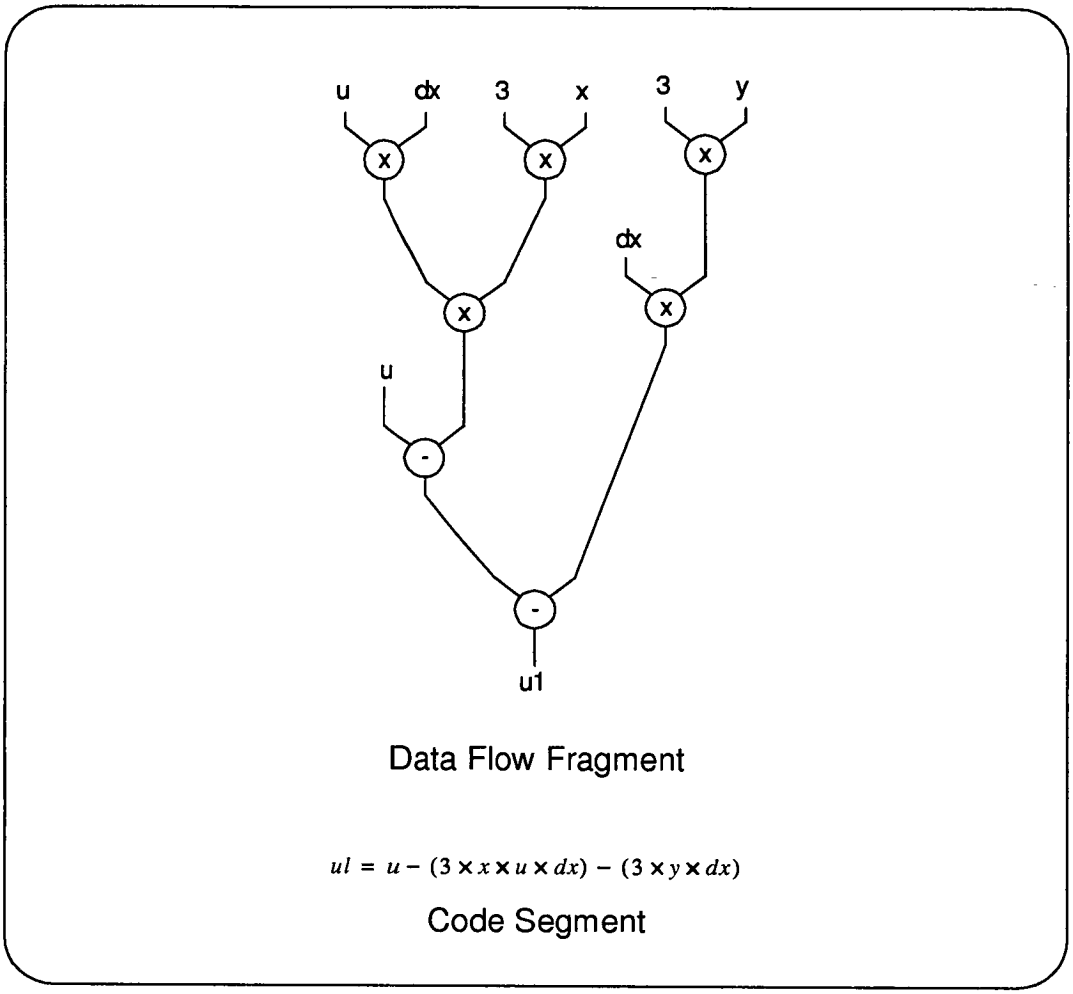


Figure 2.5 Straight line code representation.

It can be seen, however, that if the high level behavioural constructs, including conditions (**If** <cond> **then** <tbody> **else** <ebody>) and looping (**while** <cond> **loop** <body>) are to be accommodated, then further refinement of this definition is required.

Two further vertex types, $fork_n$ and $join_n$ are specified. They perform one-to-many and many-to-one edge mappings respectively, according to the value of a control edge, E_c. Loop and conditional statements may then be represented as shown in figure 2.6.

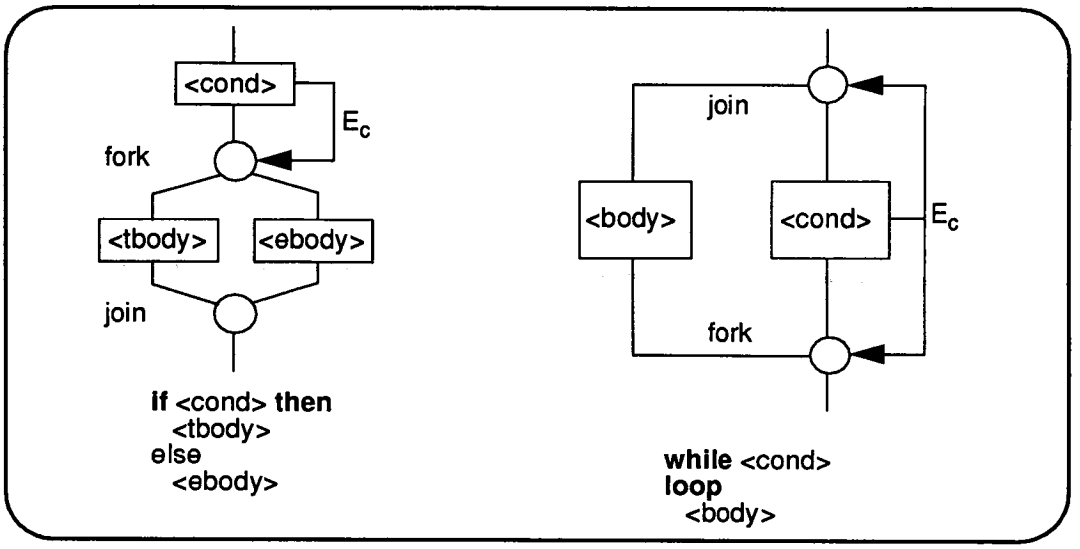


Figure 2.6 Representing conditional and loop structures.

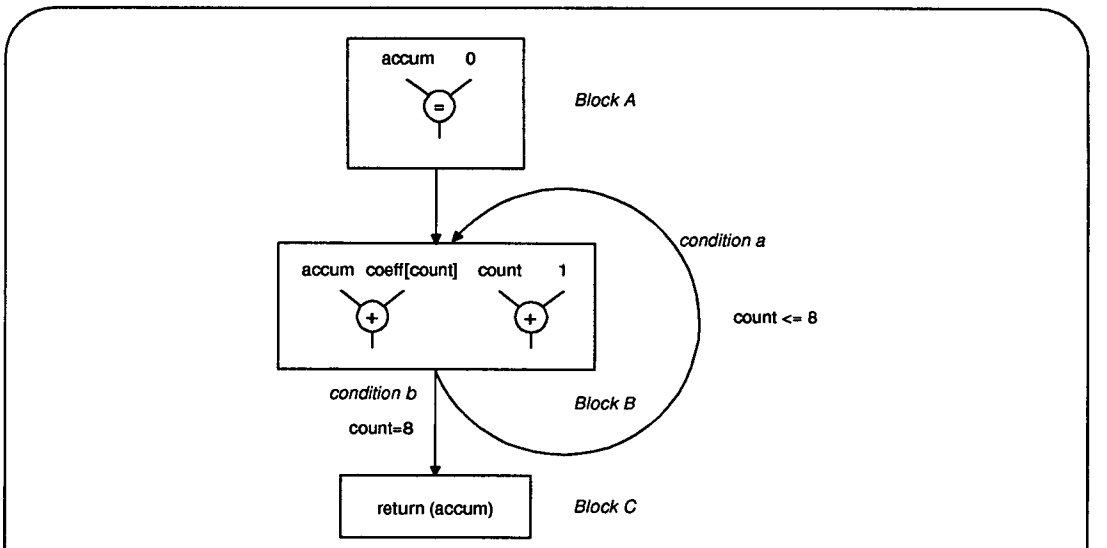
The definition of these types leads to an important classification issue within data flow representation.

If the $fork_n$ and $join_n$ nodes are associated with the vertex set, V, and the control edges, E_c, are incorporated into the data edge set, E, then the resulting data/control flow graph is said to be *combined.* The EASY system [Stok88] models both control and data flow in a single graph. A similar representation is the 'value trace' concept [Thomas87], developed for the CMU-DA system. Value trace groups combine control and data flow nodes to form 'vtbodies'; a direct analogue of the software subroutine.

If, however, the new vertices and edges form a graph, $C=(V,E_c)$, then the resulting graph pair, $G=(C,D)$ represent *separate* control and data flow graphs. Camposano [Campos89] uses such a representation to synthesise datapaths from behavioural VHDL descriptions.

This graph-pair notation can be extended to permit the definition of a procedural

hierarchy within the behavioural specification. If, instead of a single data flow graph, a multiplicity of data flow graphs is defined, one for each segment of straight line code, then each data flow graph is said to represent a *basic block*. These basic blocks are inserted as nodes within the control flow graph, much as the <body> instances are shown in figure 2.6. Thus, a procedure call may be represented as a multiple instance of a basic block corresponding to the procedure body within the control flow graph. This representation is used in [Lis88]. The code fragment from figure 2.4 is shown in a basic block structure illustrated in figure 2.7.




```
function MULTIPLE_ASSIGNMENT_FILTER (COEFF : in INTEGER[1:8]) is
  ACCUM : integer;
  COUNT : integer;

  begin
     ACCUM := 0;
     for(COUNT=1;COUNT<=8;COUNT++)
        ACCUM := ACCUM + COEFF[COUNT];
     return ACCUM;
end MULTIPLE_ASSIGNMENT_FILTER;
```

Figure 2.7 A basic block structure.

2.3 Data Flow Syntax

A textual representation[2] for the data flow semantics described above is now developed. A data flow graph is encapsulated as a *network*. A network contains all data flow information associated with the compiled source code. This is restricted to a single basic block in keeping with the definition above. Correspondingly, a network contains no control information.

The textual representation for a network is given below:

```
network <identifier>
-- vertex and edge definitions
end <identifier>
```

Each vertex in the data flow graph is represented as an *operation*:

```
operation <identifier> <type> <A> <B> <Z> end
```

The **operation.type** field indicates the type of transformation carried out by the data flow vertex. Typically these are: add, subtract and multiply. Further types are permissible provided that they constitute dyadic operations and that they are supported in the library of hardware functions available to the synthesis tools. The **operation.A** and **operation.B** fields correspond to the input data edges. For commutative operations, the ordering of input data on the data edges is unimportant, but for non-commutative operations, ordering is important if the operation is to remain functionally correct. For non-commutative operations, the ordering is A <op> B (e.g. A-B, A/B etc.). The **operation.Z** field is the output edge.

2. The syntax used here is a subset of the BABBLE language [Ryder89] used as input to the SARI Architecture Generator (SAGE) [Denyer89].

Edges within the data flow graph are represented as *signals*:

```
signal <identifier> <type> end
```

The **signal.type** field is defined as one of `input`, `output`, `constant` or `local`. `Input` and `Output` types correspond to signals whose source or sink is a boundary vertex within the flow graph. The `local` type is used to classify signals whose source and sink vertices are internal to the current network definition. Finally, the `constant` type is used to represent signals having a single pre-computed value which may be replicated throughout the network definition.

As an analogy, the signal types may be best thought of as parameters passed to a software procedure, and local variables declared within that procedure.

Figure 2.8 shows a portion of the network description of Paulin's differential equation example.

2.4 A Structural Notation

The most basic notion of structure in the context of the work described in this thesis is that of the *datapath*. A datapath can contain four types of component, namely: processors, memories, communication structures (hereafter referred to as *nets*) and I/O ports.

The syntax specifying a single datapath is given below:

```
datapath <identifier> begin
   -- processor, memory, net and I/O instances.
end datapath <identifier>;
```

```
network DIFF_EQ

signal u    input     end
signal x    input     end
signal u1   output    end
signal dx   constant  end
signal c3   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
```

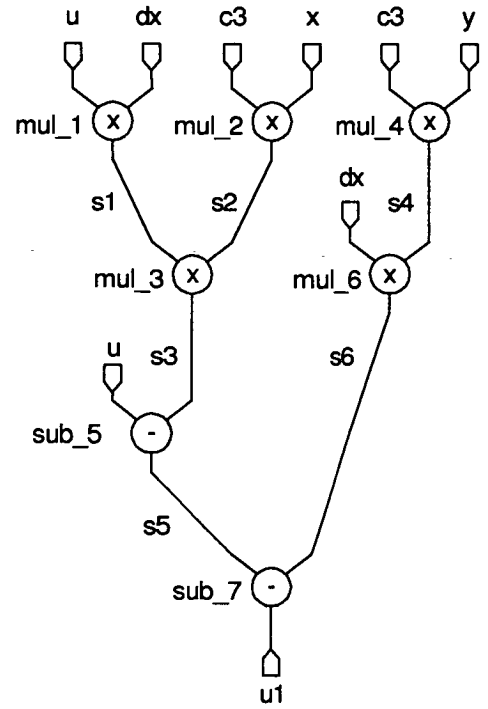


Figure 2.8 Differential equation flow graph and network description.

2.4.1 A Processor Model

The processors defined here are combinational units capable of performing simple arithmetic and logical operations. They are restricted to two input ports and a single output port (the Z port). In order to preserve non-commutivity, the input ports are labelled A and B. The most commonly used non-commutative operation, subtraction, is restricted to A-B.

The syntax for a processor is given below:

```
processor <identifier> begin
   attributes <attribute_list>;
   type <operation_type_list>;
   commutative <boolean>;
   ports    A source <net>;
            B source <net>;
```

```
            Z sink <net>;
end processor <identifier>;
```

The **processor.attribute** field contains processor specific information, such as layout area, processor latency and reuse time. The **processor.type** field contains a list of valid operation types that may be executed on this processor. In most cases, this will be a single type, but this type specification allows for the definition of generic ALU structures. The **processor.commutative** field is a flag for the synthesis tools to determine whether a port swap is a valid optimisation move.

2.4.2 A Memory Model

The structural description supports three types of memory, namely ROM (Read Only Memory), single registers and multiple registers grouped together in a register file. Associated with each register file there is decoding logic for register selection. The extra complexity associated with register files is accounted for in the datapath quality assessment (see Section 5.3).

The syntax for a memory component is given below:

```
memory <identifier> begin
   attributes <attribute_list>;
   type <ROM | register | file>;
   case[3] type of
      ROM | file => locations <integer>;
   end case;
   ports
   case type of
      register | file => A source <net>;
   end case;
      Z sink <net>;
   case type of
```

3. In this notation, the case statement affords the opportunity to conditionally instance component fields. For example, a ROM will have no A port connection, only a Z port connection.

```
      register | file => signals <signal_list>;
   end case;
end memory <identifier>;
```

The **memory.attributes** field holds memory specific information such as layout area. This is used during the synthesis procedure. The **memory.locations** field gives an indication of the cardinality of the instanced component. This corresponds to the number of registers contained in a single file or the number of values held in a ROM. The data stored in the memory component is appended to the **memory.signals** field.

2.4.3 A Communications Model

The structural model fully supports point-to-point (wire), multiplexer and bus based communications strategies. All of these components are classed as nets. Table 2.1 summarises the port connection options associated with each communications component.

Component	Source	Sink
Wire	Single	Single
Mux	Multiple	Single
Bus	Multiple	Multiple

Table 2.1 Net source and sink options.

The syntax for a net is given below:

```
net <identifier> begin
   type <wire | mux | bus>;
   case type of
      mux | bus => cardinality <integer>;
   end case;
   source <port_list>;
   sink <port_list>;
end net <identifier>;
```

2.4.4 Input and Output Ports

The input and output ports provide a means of external communication with the datapath. The syntax for an I/O port is given below:

```
I/O <identifier> begin
   type <in | out | bid>;
   case type of
      in => A sink <port_list>;
      out =>   Z source <port_list>;
      bid =>   A sink <port_list>;
               Z source <port_list>;
   end case;
end I/O <identifier>;
```

The case statement is used here to instance I/O connections relevant to the port type.

2.5 Interlude - A Naive Mapping

Having developed both behavioural and structural models, a naive mapping between the two domains is offered, based on the following axioms:

(i) Operations should be mapped onto processors capable of executing the operation type.

(ii) Where a signal traverses a c-step boundary, a memory element will be instanced to preserve the signal state between control steps.

(iii) Each signal should be mapped to a wire or bus connecting the source and sink modules.

(iv) Instance I/O ports of the appropriate type where the **signal.type** field is either input or output

(v) Instance a ROM where the **signal.type** field is constant.

Thus, the code fragment of figure 2.1 can now be mapped to a simple datapath structure as shown in figure 2.9. (The datapath description is presented in Appendix A).

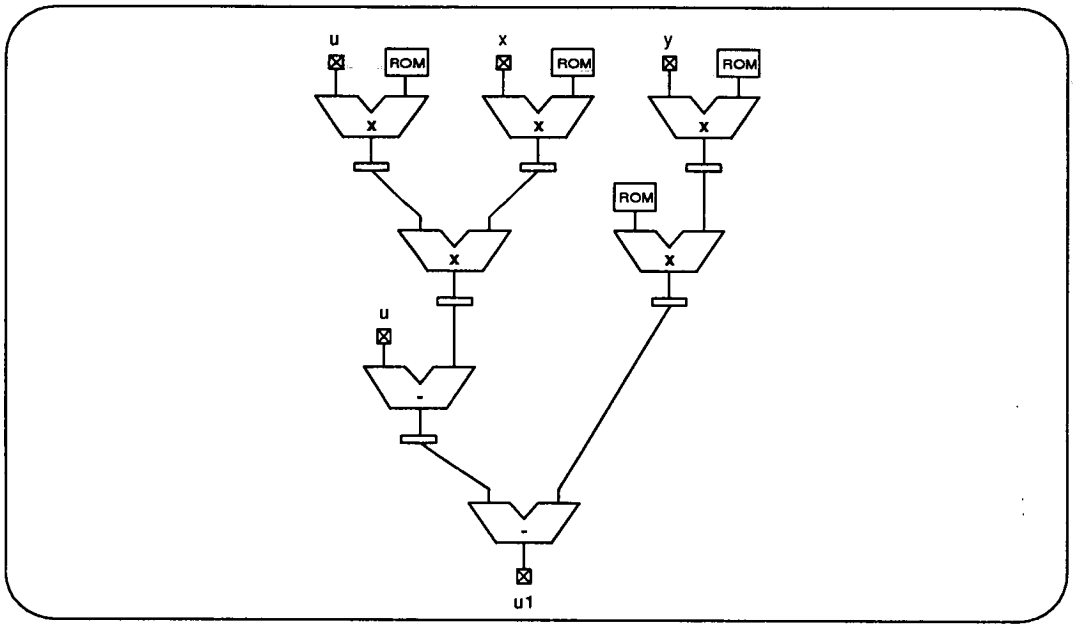


Figure 2.9 Naive differential equation datapath.

This solution offers particularly poor resource utilisation (the RTL description of this datapath produced using the structural notation is presented in Appendix A). The axioms yield a low-quality datapath with replicated hardware components. Possibly the only benefit of such a solution is that it entails zero control overhead, and produces a solution after three cycles of the system clock. The synthesis techniques in the following chapter, together with the tools presented in chapter 5, describe methods of optimising the datapath topology.

2.6 A Control Model

Consider the execution of a typical data flow operation. The data is retrieved from memory or the input ports of the datapath, operated on by the datapath element, then placed back in memory or made available externally via the datapath output ports.

The combination of processor and memory elements is a synchronous sequential machine corresponding to the Mealy model [Mealy54], as shown in figure 2.10.

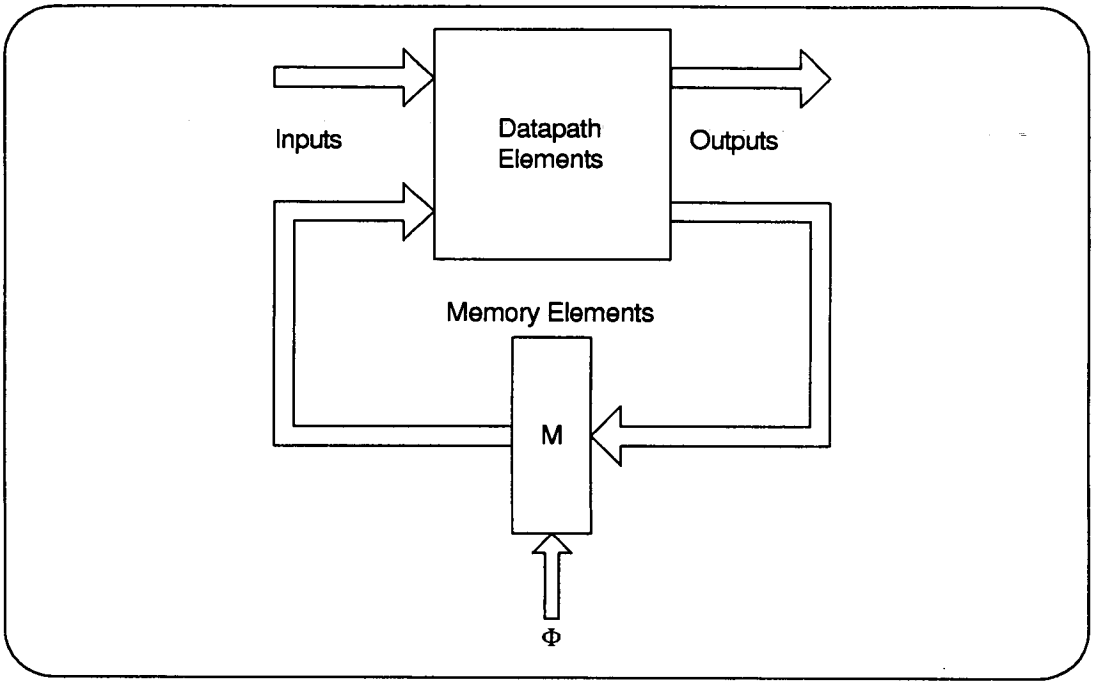


Figure 2.10 Synchronous sequential machine (with potential race)

The composition of the memory elements holding the current state, M, is critical. In order to avoid race hazards [Seitz80] latches and edge triggered flip-flops *must* be avoided. The transparent mode of the latch prevents any state being retained, and the model shown above becomes an unpredictable asynchronous system. Edge-triggered flip-flops are also unsatisfactory because it can not be guaranteed that all elements of M will latch their input data simultaneously. Some memory elements may latch marginally early; changes on the outputs of M could then loop round through the combinatorial datapath to the input and produce an unwanted race condition.

To avoid the race problem, M should be based upon a *master-slave* latch structure, as shown in figure 2.11. When the master clock is high, the outputs of the combinatorial datapath network are stored in the master latches. During this time, the slave latch

maintains consistent data at the inputs to the datapath network. When the slave clock is high, data is transferred from the master to the slave latch, and thus the input data to the datapath network is changed safely. To ensure correct operation, the master clock (Φ_1) and the slave clock (Φ_2) are defined so as to be non-overlapping (an overlap between Φ_1 and Φ_2 would cause both latches to go into transparent mode, again causing the critical race problem).

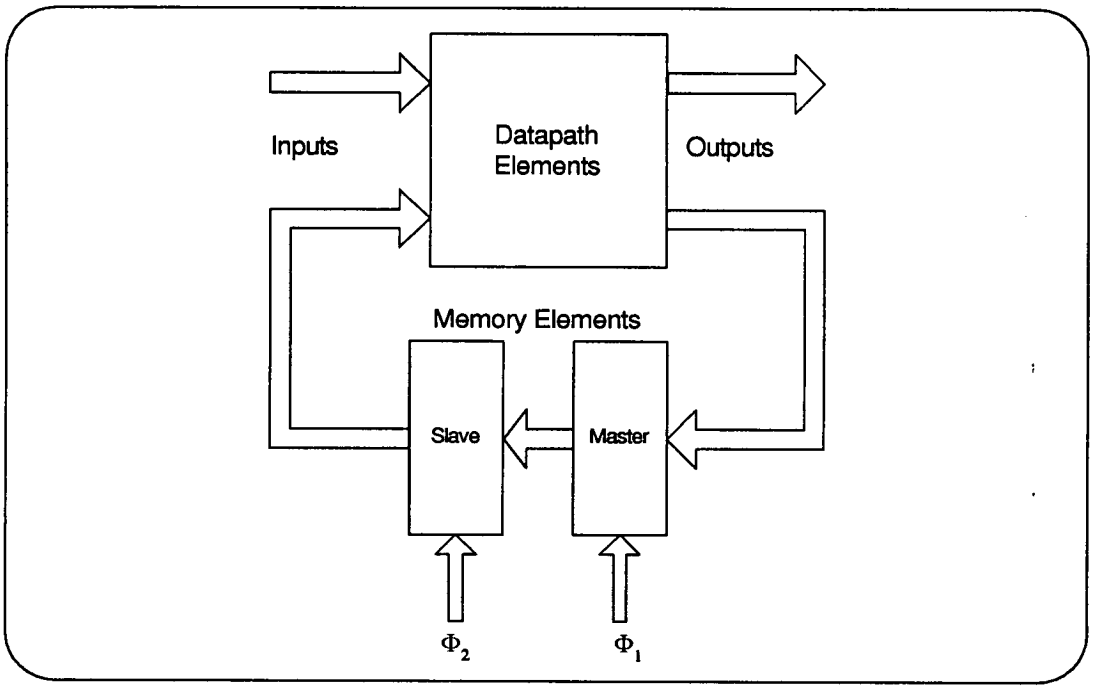


Figure 2.11 Synchronous sequential machine with master-slave memory structure.

Relating this timing model to the execution of the dataflow operation, the *read* phase of a control step (i.e. the fetching of datapath input data from memory or the datapath input ports) can be specified as the time interval between Φ_2 and Φ_1, and the *write* phase of a control step (i.e. the placing of datapath output data in memory or at the datapath output ports) as the time interval between Φ_1 and Φ_2. This is shown in figure 2.12.

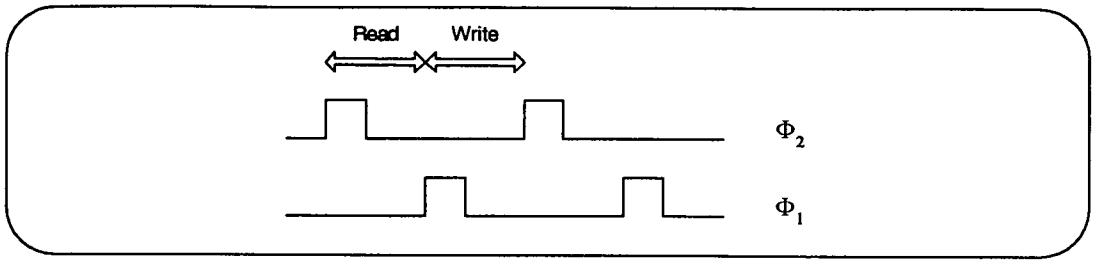


Figure 2.12 Read and Write phase timing.

2.7 Summary

Following a review of behavioural representation, this Chapter has developed behavioural and structural models suitable for high-level synthesis. Subsequently, a naive binding between the behavioural and structural domains based on greedy heuristics was produced. This binding yielded a particularly low quality solution. The next Chapter considers techniques designed to optimise the binding quality.

3 Datapath Synthesis Techniques

The previous chapter introduced behavioural and structural models capable of supporting the synthesis operations described in section 1.3. This chapter reviews the algorithmic techniques currently used to produce an optimised register-transfer description of a datapath from this intermediate representation.

Techniques for solving the scheduling subtask are considered separately from those techniques applicable to the processor, register and interconnect allocation subtasks. The complex inter-relationship between the various synthesis subtasks, and an indication of the drawbacks of a serial synthesis flow is exemplified by considering the essential dichotomy that exists between the scheduling and processor allocation subtasks.

The Scheduling and Allocation Dichotomy

The scheduling subtask aims to assign execution times to all nodes in a data flow graph. Without knowledge of the number of processors available during scheduling,

however, an optimal solution is impossible to derive.

Similarly, the number of processors needed to execute all data flow graph nodes is dependent on the amount of parallelism within the graph at any particular control step. This information is generated by the scheduler. Thus, there is a cyclic relationship.

From this basic observation, it may be deduced that any synthesis flow where the scheduling and allocation operations are disjoint (regardless of the order in which the subtasks are performed) may ultimately compromise the quality of the datapath solution.

3.1 Scheduling Techniques

The three major classes of scheduling strategy are discussed in this section. The first, and largest class of scheduling algorithm is known as iterative scheduling. This class operates on a node-by-node basis, and is characterised by the order in which the nodes are visited. An important subclass is the distribution-based scheduler, which is examined in some detail. The second major class uses serialising and parallelising transformations on unary and fully serial schedules. Finally, a small class of synthesis systems formulate the scheduling as an integer programming problem.

3.1.1 Iterative Scheduling Schemes

The first class of iterative scheduling scheme constitutes the base scheduling strategies, where no hardware bound is placed on the resource set available to the scheduler. In effect, this class of scheduler operates independently of any allocation system. The most common base schedules are: AS-SOON-AS-POSSIBLE (ASAP) and AS-LATE-AS-POSSIBLE (ALAP). More unusual variations, such as AS-FAST-AS-POSSIBLE (AFAP [Campos90]) do exist, but are not in common use.

In ASAP schedulers, nodes are scheduled to occur as soon as their input data are available. Similarly, in ALAP schedules, nodes are scheduled to occur in the control step preceding the earliest consumption of their output data. The effects of ASAP and ALAP schedules are shown in figures 3.1 and 3.2.

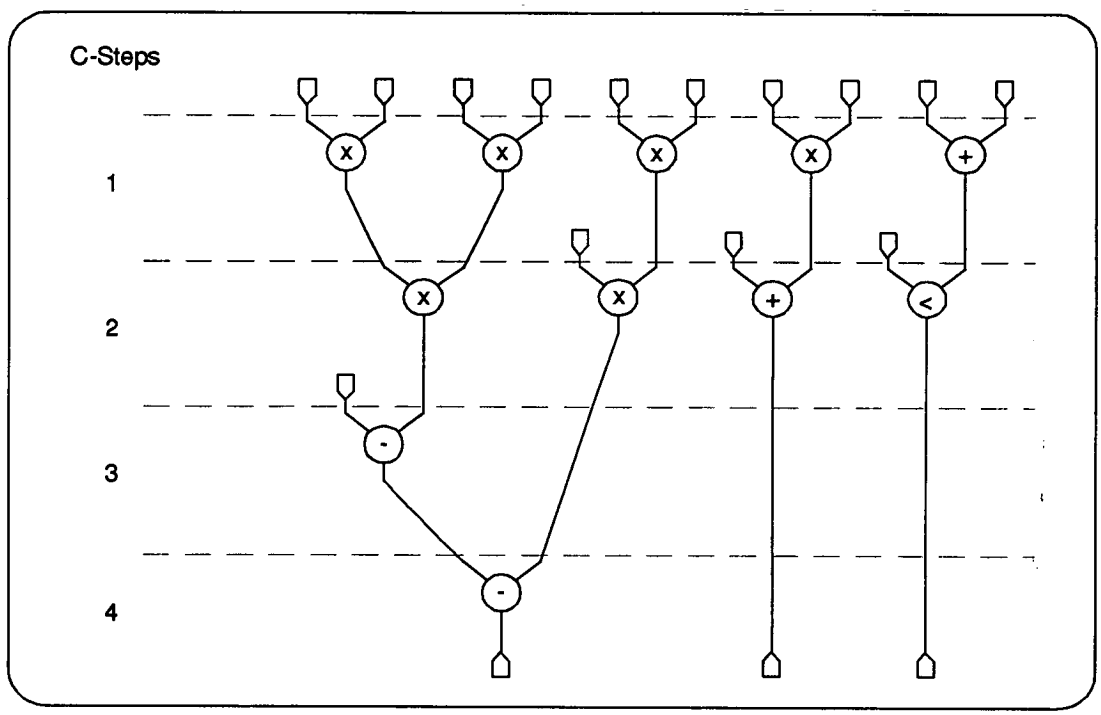


Figure 3.1 As Soon As Possible (ASAP) Scheduling Strategy.

These strategies, while producing high speed solutions can be wasteful in terms of the excess hardware required to realise the data flow graph. As can be seen from figure 3.1, the ASAP schedule requires 4 multipliers executing concurrently, while the ALAP schedule (figure 3.2) requires only 2, at no overall execution time penalty. Consequently, these base schedules are not generally employed on their own within synthesis systems.

While the base schedules simply correspond to an ordering of the input data flow graph subject only to the data flow constraints themselves, constraining the number of available processors *a priori* necessitates the maintenance of a 'ready list'. This list

contains details of all operations capable of being scheduled at a particular control step.

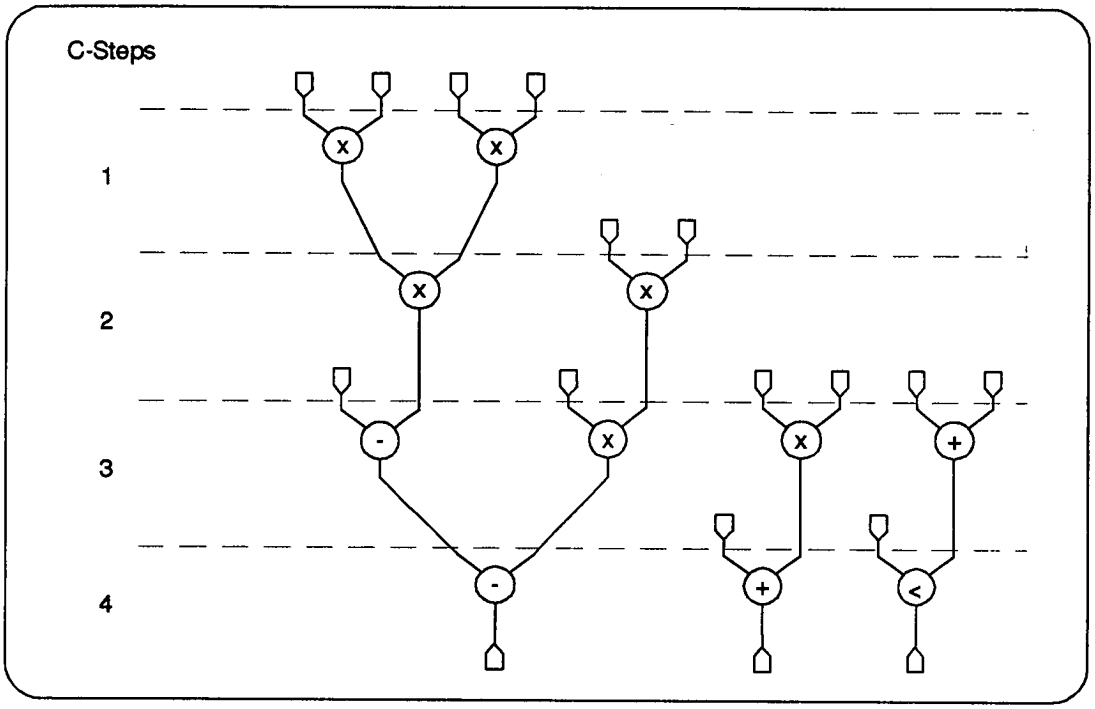


Figure 3.2 As Late As Possible (ALAP) Scheduling Strategy.

The scheduler processes the data flow graph a control step at a time, removing operations from the ready list and scheduling them according to a prioritising function. This prioritising function resolves resource conflicts when the amount of operational parallelism present in the flow graph exceeds the processor parallelism. The most common form of prioritising function assigns a weight to each data flow graph node. This weight is then used as a selection criterion which determines the node or nodes to be scheduled next, or identifies suitable candidate nodes for deferment.

The simplest prioritising scheme schedules nodes on the critical path of the data flow graph to execute first. In the scheme described above, this may be viewed as a binary weighting. The ATOMICS [Goosse87] scheduler in the CATHEDRAL-II system is

based around this strategy. Parker [Parker86] refines this by determining the *freedom* for all remaining operations once the operations present on the critical path have been scheduled. The operation freedom is defined as the difference between the ASAP and ALAP schedules for that operation, less the propagation delay of the processor executing that operation. For operations executing on the critical path, the node freedom is zero. This approach emphasises the interrelation of the scheduling and allocation subtasks. The effects of critical path and operator freedom based scheduling are illustrated in figures 3.3 and 3.4.

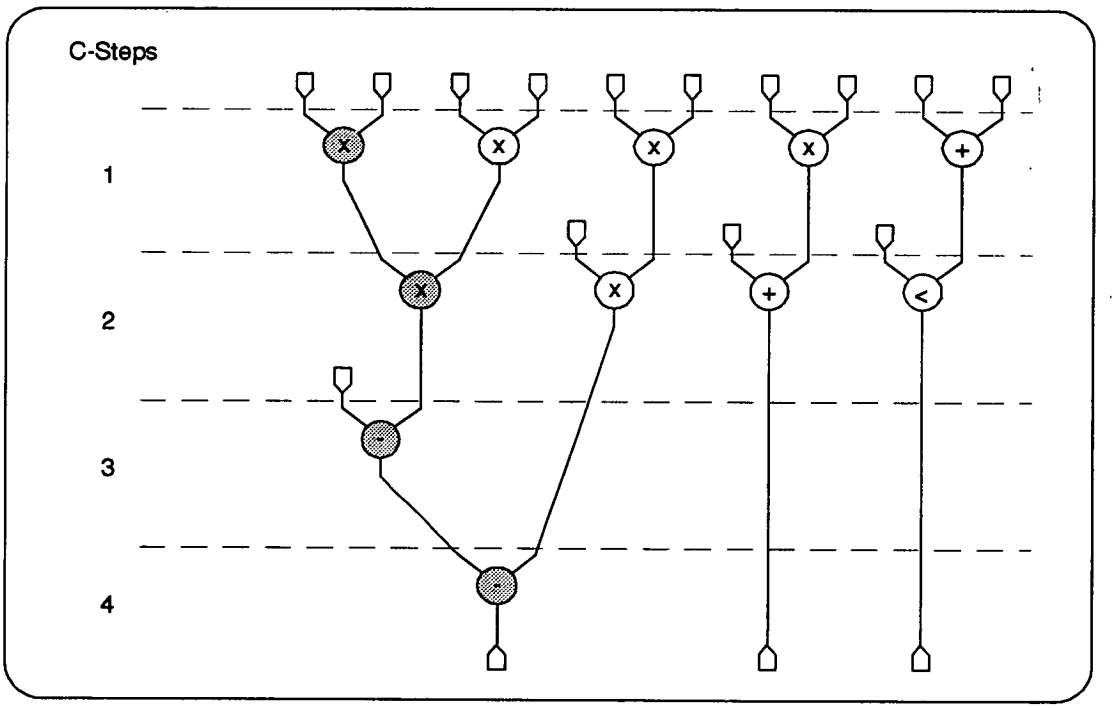


Figure 3.3 Critical Path Analysis (Shown in gray).

Girczyc [Girczyc85] uses node urgency as a selection criterion. Node urgency corresponds to the minimum number of control steps required to execute all operations between the current operation and the nearest timing boundary (e.g. system output, or basic block boundary.) Nodes with the greatest urgency function are given the highest priority during scheduling. This is shown in figure 3.5.

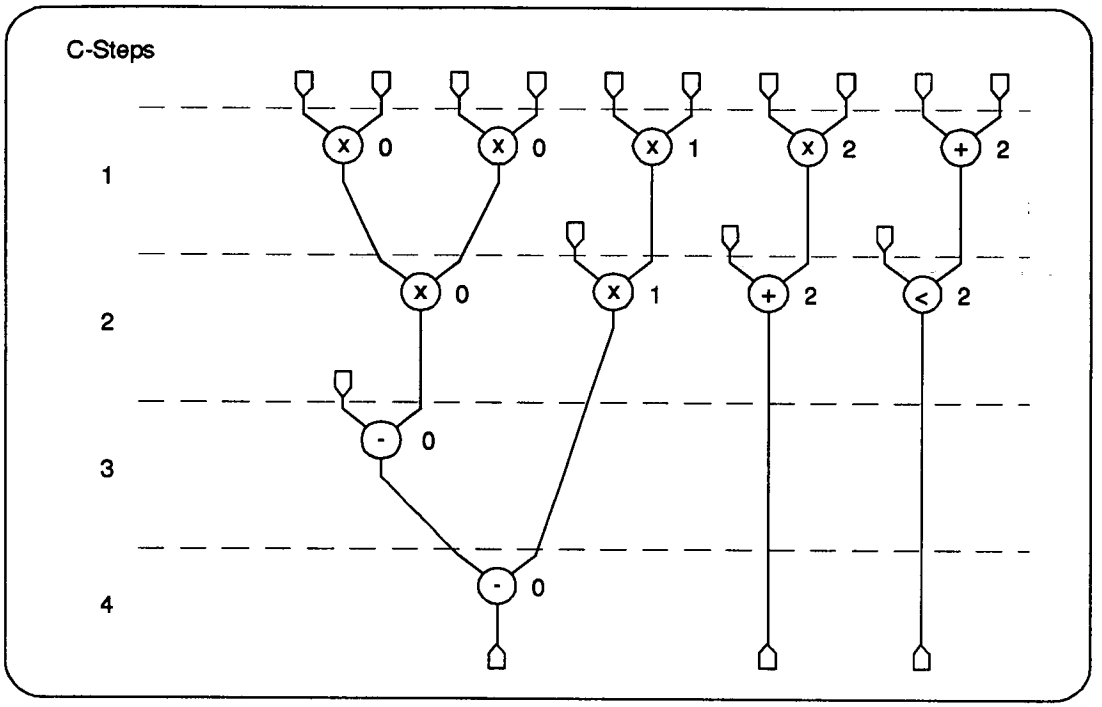


Figure 3.4 Operator Freedom Analysis

Splicer [Pangrle87, Pangrle88] uses node freedom (called node *mobility*) as a primary selection mechanism. Where resource conflicts occur, and node mobility values do not differentiate between schedulable operations, a secondary selection mechanism operates. Here, the node to be scheduled is selected based upon the length of path remaining to the nearest timing boundary. The node with the longest path is selected.

The SEHWA system [Park88] uses two urgency based schedulers. The first, which orders the data flow graph without hardware constraints establishes the *maximal* schedule. A second urgency scheduler is executed on a hardware constrained system to produce the *feasible* schedule. If the maximal schedule is better than the feasible schedule, then a third, exhaustive, scheduler is applied to produce a shorter feasible schedule. If the converse is the case, then the original feasible schedule is used.

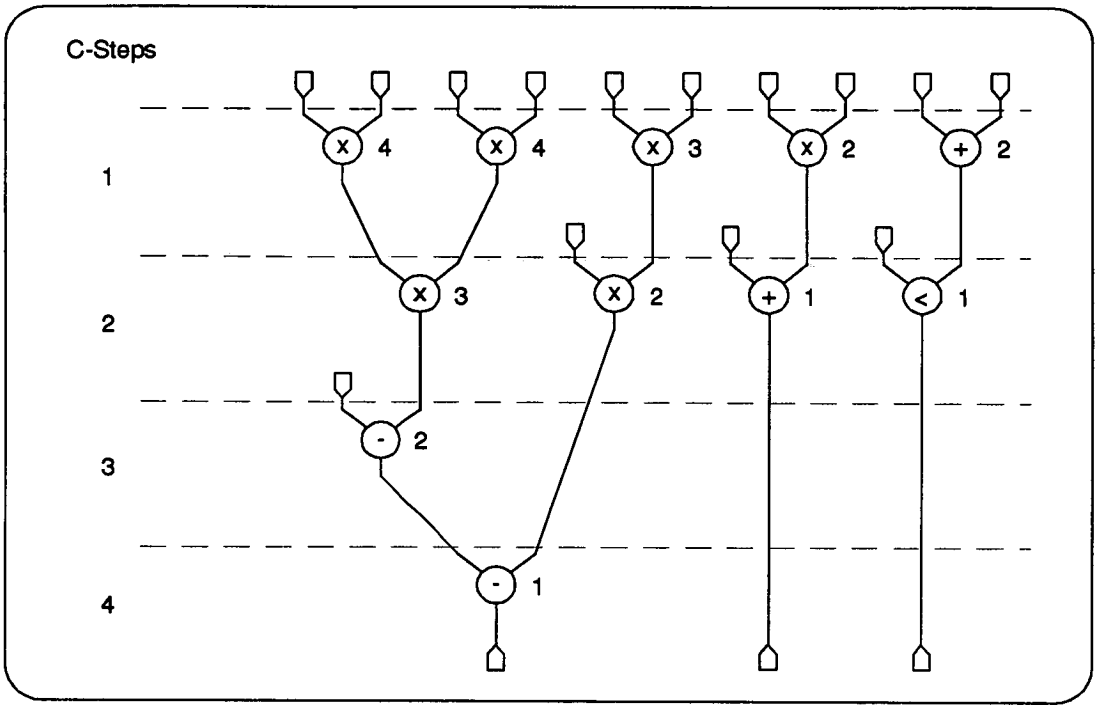


Figure 3.5 Node Urgency Analysis

Balancing operator concurrency : force-directed scheduling

Paulin [Paulin89b] developed an important class of scheduler which aims to balance the operational concurrency within a data flow graph on a control-step to control-step basis. This class of scheduler is referred to as *force-directed* scheduling. The algorithm is partitioned into three main stages:

Determination of Time Frames. During this step, ASAP and ALAP schedules are generated for an individual node. This determines the feasible schedule range, and is similar in form to the notion of operator freedom, as introduced by Parker [Parker86].

Creation of Distribution Graphs. For each control step, a summation of the probabilities that individual operations of a similar class will execute in that

control step is formed. The resulting distribution graph (DG) represents an indication of the concurrency of similar classes of operation for a particular control step, and is defined:

$$DG(i) = \sum_{Op-class} P(Op,i) \qquad [3.1]$$

where i is the current control step under consideration, and $P(Op,i)$ is the probability of the selected operation occurring during that control step.

Calculate Forces. Here, the force associated with assigning an operation to a particular control step is determined, and is defined as the difference between the Distribution Graph value associated with the trial assignment and the average DG values over the time frame of the operations. Thus:

$$F(j) = DG(j) - \sum_{i=t_1}^{t_2} \left[\frac{DG(i)}{(t_2 - t_1 + 1)} \right] \qquad [3.2]$$

where $F(j)$ is the force associated with assigning the selected operation to control step j, and the time frame of the operation runs from time t_1 to time t_2.

Further, *indirect force* is defined as the force associated with the implicit assignment of control step values to nodes which have direct data dependencies on the initial trial node, j. Once the calculation of direct and indirect forces is completed for an operation, the control-step assignment is selected yielding the lowest total force. By definition, this balances the concurrency most effectively for a particular operation.

This force-directed scheduling technique produces datapaths capable of satisfying fixed timing constraints, but does not address the problem of scheduling under resource constraints.

Figures 3.6-3.8 illustrate the use of the force-directed technique for the differential equation example.

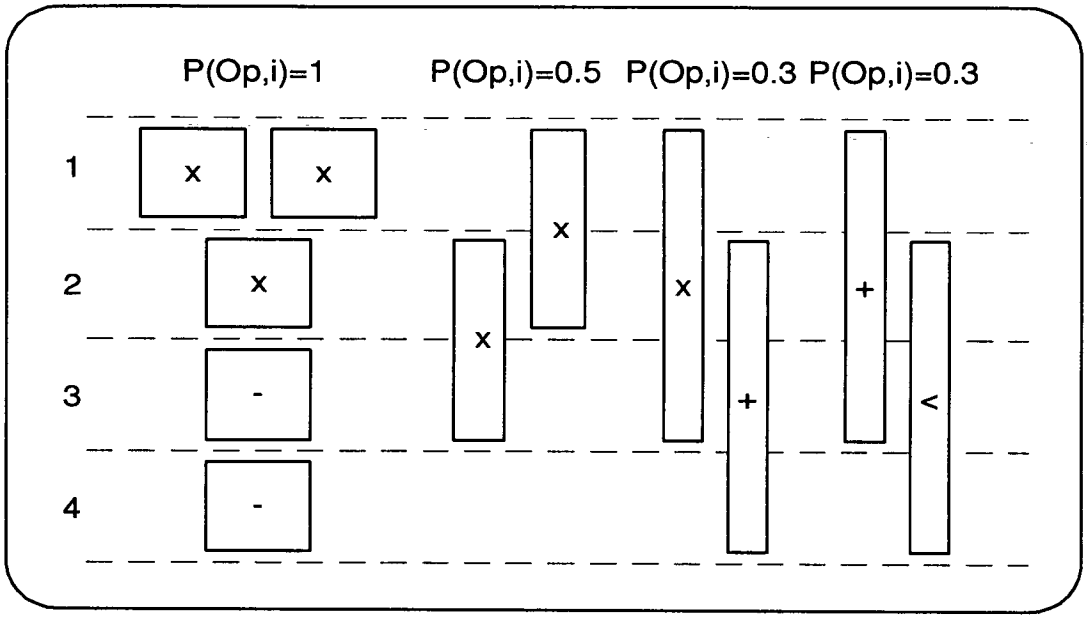


Figure 3.6 Initial time frames for differential equation example (after [Paulin89b]).

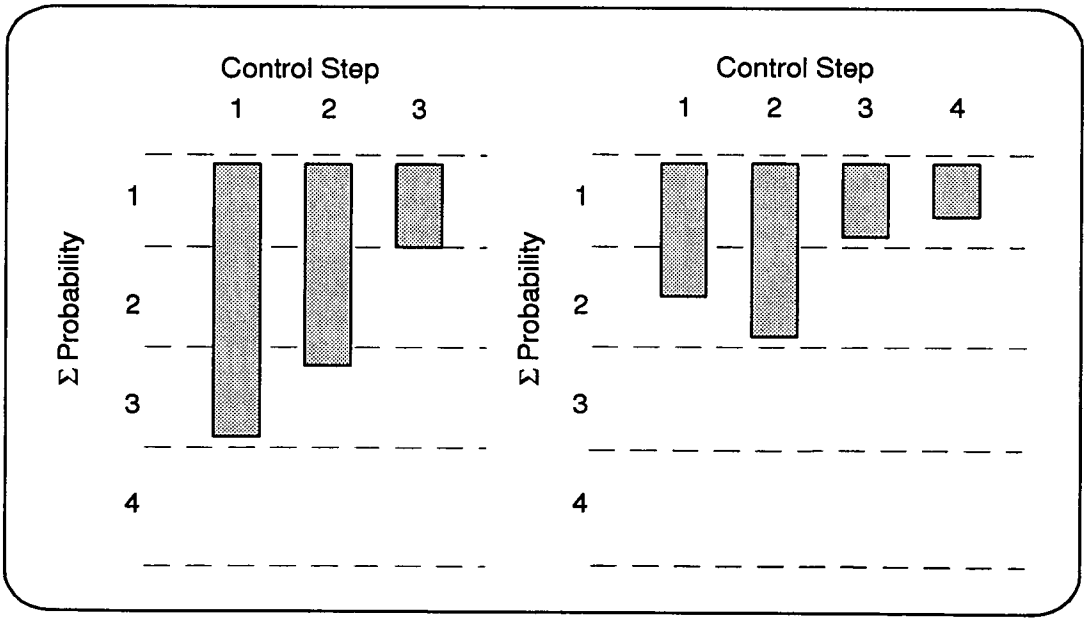


Figure 3.7 Initial distribution graphs for multiply (left) and add, subtract and compare.

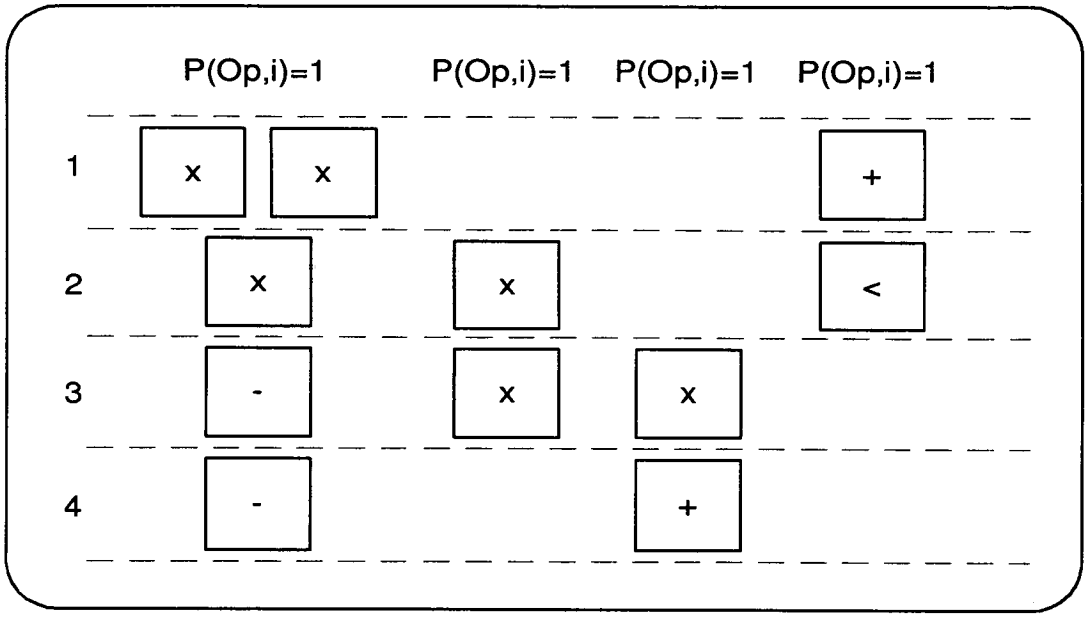


Figure 3.8 Final time frames for differential equation example.

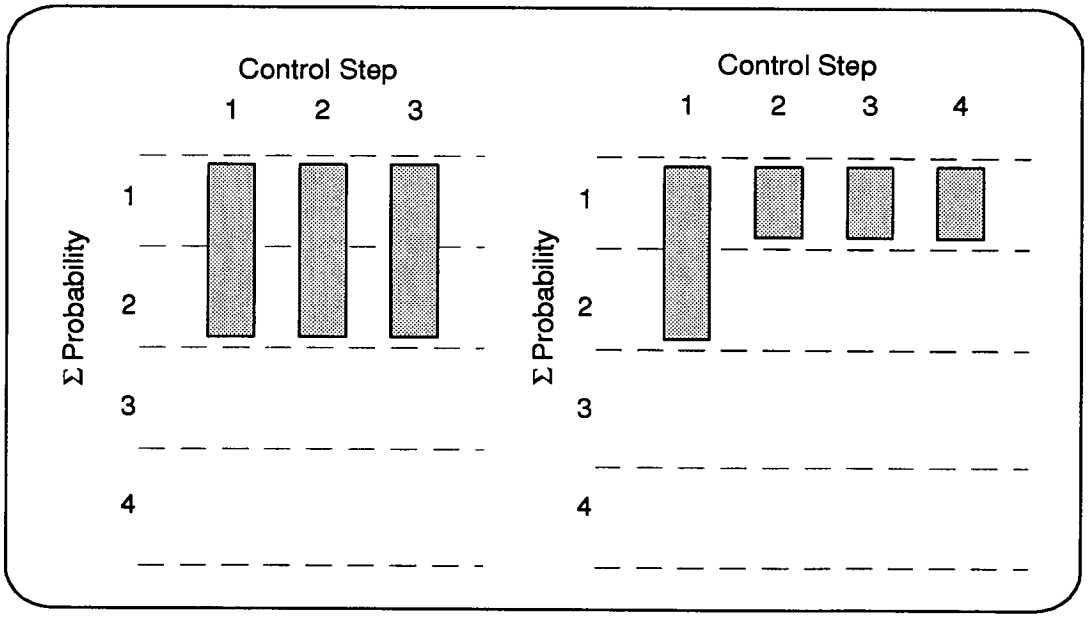


Figure 3.9 Final distribution graphs for multiply (left) and add, subtract and compare.

Force-Directed List Scheduling

In common with other list scheduling approaches, the FDLS algorithm sorts the data-flow graph nodes according to data and control dependency. Those operations put onto

the 'ready list' are capable of being assigned to the first control step. In the case when operational parallelism exceeds the amount of parallel hardware available during a particular control step, then one or more of the 'ready' operations must be deferred to subsequent control steps. As detailed above, operations are selected according to a prioritising function. The FDLS algorithm selects the operation from the ready list which has the lowest total force value associated with it, i.e. the operation to control step assignment producing the lowest global increase in concurrency.

3.1.2 State Transformation Scheduling

This class of scheduler operates under a resource-constrained regime. State transformations on the initial schedule are governed by two factors: data dependency and resource availability.

State Merging Transformations

In this case, a fully serial schedule is transformed by merging those operations in adjacent control steps, subject to data dependency and resource availability. Pseudo code for the algorithm is given in figure 3.10.

State Splitting Transformations

This algorithm begins with a unary schedule (i.e. a schedule where all operations occur in a single control step, thereby violating data dependency constraints). Once again, subject to data dependency and resource availability, extra control steps are added, and operations assigned to them until no further constraints are violated. This process is illustrated in figure 3.11. Both the Yorktown Silicon Compiler (YSC) [Brayton88] and the Linköping CAMAD [Peng86, Peng87] systems use this approach.

```
function STATE_MERGE (g : graph);
begin
  for (all nodes in g)
      assign node to differing csteps;
  repeat
      select (cstep);
      if (merge-possible) then
          merge(adjacent csteps);
  until no merges possible;
end STATE_MERGE;
```

Figure 3.10 State Merging Transformations.

```
function STATE_SPLIT (g : graph);
begin
  for (all nodes in g)
      assign node to single cstep;
  repeat
      select (cstep);
      if (split-possible) then
          split(cstep);
  until no splits needed;
end STATE_SPLIT;
```

Figure 3.11 State Splitting Transformations.

3.1.3 Integer Linear Program Scheduling (ILP)

Lee [Lee89] formulates the scheduling problem as an integer linear programming problem. Consider a data flow graph containing n operations. Each operator, O_i, has pre-computed ASAP and ALAP schedules, S_i and L_i, respectively. If a resource constraint is added stating that there are m types of processor of type t_i, each having a cost C_{ti}, then M_{ti} denotes the number of processors of type t_i required. The decision variables, $x_{i,j}$ are set to 1 if O_i is scheduled in control step j, and zero otherwise.

The scheduling problem is formulated thus :

$$min \sum_{i=1}^{m} C_{ti} M_{ti} \qquad [3.3]$$

Equation 3.3 states that the object is the minimisation of the total processor cost. This is subject to the following constraints :

$$\sum_{i=1}^{n} x_{ij} - M_{tk} \leq 0 \qquad 1 \leq j \leq n \qquad 1 \leq k \leq m \qquad [3.4]$$

This ensures that the resulting schedule has no more than M_{tk} functional units of type t in any one control step.

$$\sum_{j=S_i}^{L_i} x_{ij} = 1 \qquad 1 \leq i \leq n \qquad [3.5]$$

This constraint ensures that O_i is scheduled between the precomputed ASAP and ALAP schedules. Finally, for the data flow dependencies to be satisfied, equation 3.6 must be applied.

$$\sum_{j=S_i}^{L_i} jx_{ij} - \sum_{j=S_k}^{L_k} jx_{kj} \leq -1 \qquad [3.6]$$

for all nodes O_i and O_k constrained by data flow dependencies.

For the differential equation example, with a multiplier cost of 5 ($C_{tmult} = 5$) and an ALU cost of 1 ($C_{tALU} = 1$) (values taken from Lee [Lee89]), the problem formulation and resulting schedule are shown in figures 3.12 and 3.13.

3.1.4 Discussion

Most of the scheduling schemes described above operate on a flattened segment of the data flow graph, consistent with the basic block representation described in Section 2.2. For datapath dominated designs (i.e. designs where the amount of data flow

contained in individual basic blocks is significant compared to the amount of control flow), the deferral-based scheduling schemes are appropriate.

minimise ($5 \times M_{mult} \times M_{alu}$) subject to :

$x_{1,1} + x_{2,1} + x_{6,1} + x_{8,1} - M_{mult} <= 0;$

$x_{3,2} + x_{6,2} + x_{7,2} + x_{8,2} - M_{mult} <= 0;$

$x_{7,3} + x_{8,3} - M_{mult} <= 0;$

$x_{10,1} - M_{alu} <= 0;$

$x_{9,2} + x_{10,2} + x_{11,2} - M_{alu} <= 0;$

$x_{4,3} + x_{9,3} + x_{10,3} + x_{11,3} - M_{alu} <= 0;$

$x_{1,1} = 1;\ x_{2,1} = 1;\ x_{3,2} = 1;$

$x_{4,3} = 1;\ x_{5,4} = 1;$

$x_{6,1} + x_{6,2} = 1;\ x_{7,2} + x_{7,3} = 1;$

$x_{8,1} + x_{8,2} + x_{8,3} = 1;\ x_{9,2} + x_{9,3} + x_{9,4} = 1;$

$x_{10,1} + x_{10,2} + x_{10,3} = 1;$

$x_{11,2} + x_{11,3} + x_{11,4} = 1;$

$x_{6,1} + 2x_{6,2} - 2x_{7,2} - 3x_{7,3} < = -1;$

$x_{8,1} + 2x_{8,2} + 3x_{8,3} - 2x_{9,2} - 3x_{9,3} - 4x_{9,4} <= -1;$

$x_{10,1} + 2x_{10,2} + 3x_{10,3} - 2x_{11,2} - 3x_{11,3} - 4x_{11,4} <= -1;$

Figure 3.12 ILP formulation of differential equation example.

Conversely, for control dominated designs, those scheduling schemes may yield sub-optimal results. Potkonjak [Pot89] describes a hierarchical scheduling scheme which relates more closely to the behavioural synthesis paradigm (i.e. the ability to describe an algorithm in a high level programming language). This hierarchical method does require extensive traversal of the synthesis hierarchy and backtracking.

In terms of individual scheduling techniques, the iterative schemes are capable of producing optimised results when operating in conjunction with an appropriate allocation algorithm. Force-directed list scheduling produces marginally better quality results than the other list based scheduling approaches at a negligible increase in computational complexity.

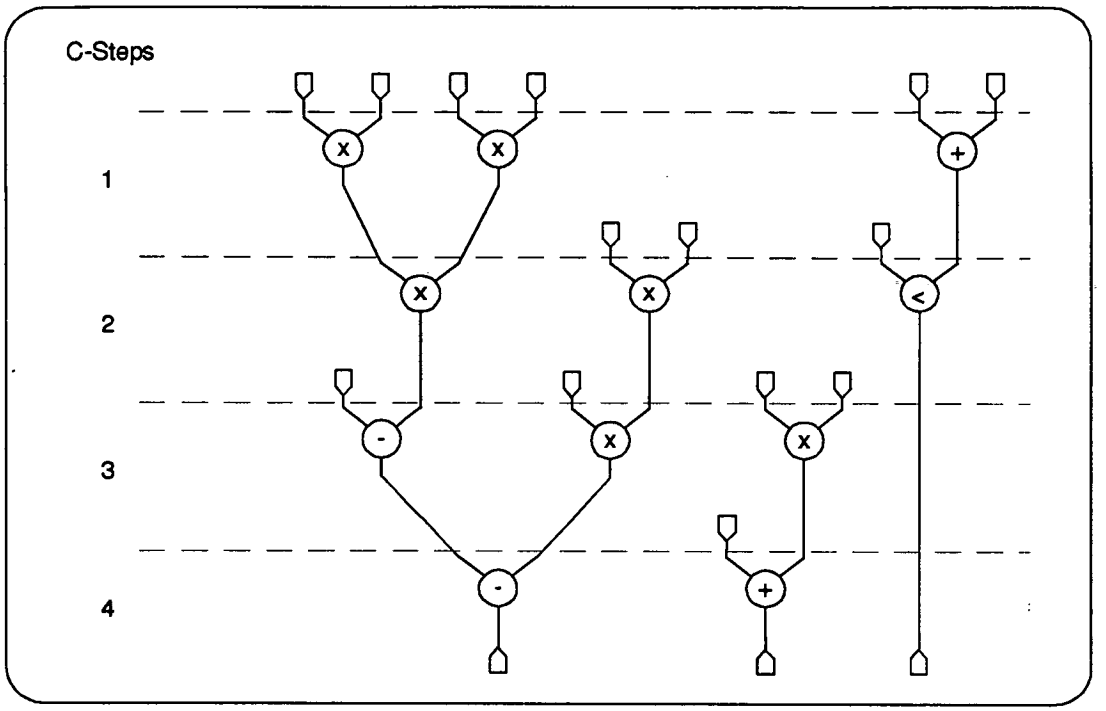


Figure 3.13 ILP Scheduling solution.

The transformational schemes benefit from the fact that both algorithms are easily implemented. In both cases, however, the order in which nodes are merged into control steps, or split into new ones is arbitrary, and in many cases, this scheme produces low quality solutions [Fin92].

ILP techniques have only successfully been applied on problem instances of no practical significance [Lee89]. The solution of the decision matrix requires large amounts of compute time, even for modestly sized problems, and the formulation of the problem may prove unwieldy. Attempts have been made to partition the overall ILP scheduling problem on a control-step by control-step basis. These are reported in Huang [Huang90a]

In most cases, the scheduling techniques described can be modified to include multicycling (i.e. operators requiring more than one control step to execute), operator

chaining (i.e. combinatorial operators occurring within a single control step), and functional pipelining, as illustrated in figure 3.14.

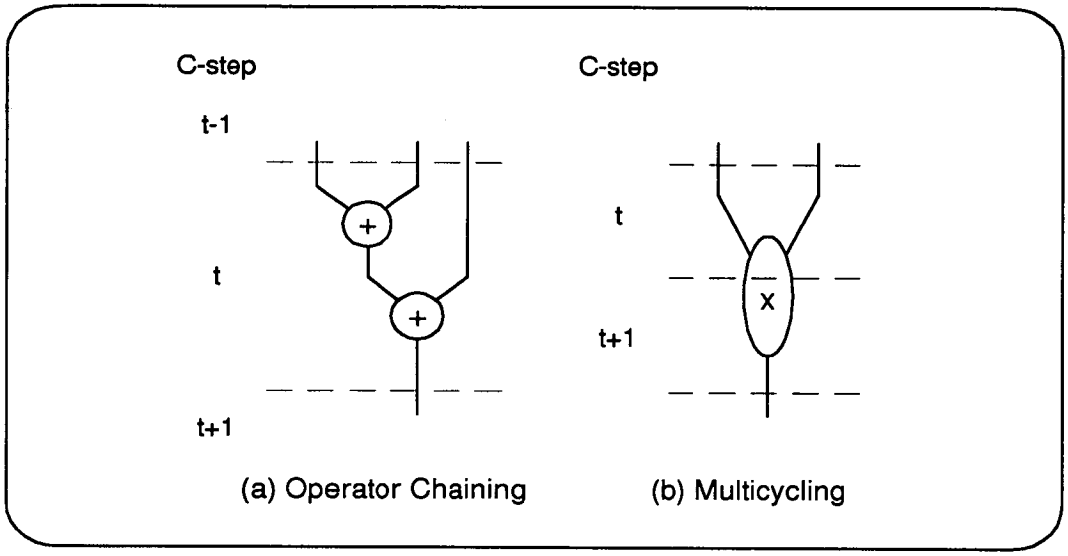


Figure 3.14 Operator chaining and multicycling.

3.2 Allocation Techniques

Data path allocation corresponds to stages (iii) - (v) in the behavioural synthesis design flow outlined in Section 1.3. These subtasks are grouped together in this case to emphasise the inter-relationship between the subtasks, and to establish an informal taxonomy of algorithms suitable for their solution.

The discussion of the scheduling and allocation dichotomy revealed the relationship between the operator schedule and processor allocation. To reiterate, for effective processor allocation, the degree of parallelism within the data flow graph must be known for every control step. Compounding this problem, the need to use generic alu structures may cause allocation clashes in individual control steps.

Further, only once the schedule has been fixed can the register allocation subtask be completed. Completing the operator scheduling specifies a set of tuples, L_v,

comprising of the signal value and its corresponding life time. A signal life time is defined as the length of time between the production of a value, and its latest consumption time. Resource clashes can also occur on registers where data is written to a particular unit before the previous value has been read. Correspondingly, a further component of the register allocation subtask is the grouping of registers.

Again, this grouping affects the final allocation subtask: interconnect allocation. The ultimate aim is to provide a set of data transfer tuples, D_t, comprised of source and destination modules (either registers or processors) and the value transferred. From this tuple set, a suitable communications infrastructure can be synthesised.

Unlike the algorithms developed to solve the scheduling subtask, allocation algorithms are of a more diverse nature. (In the survey of scheduling algorithms, it was shown that most scheduling schemes are constructed from two elements: a base scheduling strategy, and a conditional deferment, or prioritising, function.) Broadly, however, algorithms solving the allocation subtask can be partitioned into two main types. The first is based on algorithms designed to solve graph theoretic problems, and rely on the formulation of the problem as a simple, undirected graph. The second type uses expert system and greedy iterative techniques. The former category benefits from the fact that all three subproblems, as outlined above, can be formulated in a similar way for solution, while the latter allows incremental construction of a solution datapath.

3.2.1 Graph Theoretic Algorithms for Allocation

These algorithms formulate the allocation procedure as an undirected graph. Johnson [Johns76] defines a **clique** as a *"maximal, completely connected subgraph of a simple undirected graph"*. This is illustrated in figure 3.15. Johnson develops a family of programs capable of generating all the cliques of a graph. This family of programs is used throughout the survey of clique based algorithms.

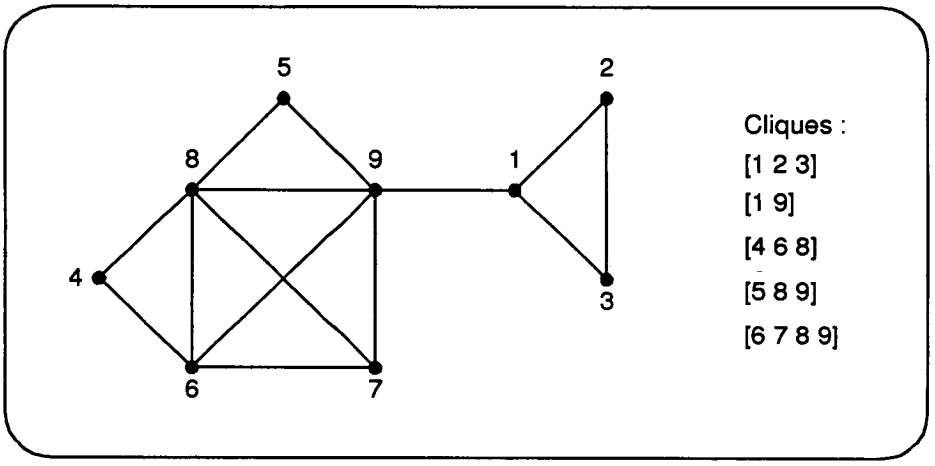


Figure 3.15 Cliques of a graph (after [Johns76]).

Clique Partitioning

Processor Allocation: A graph is defined $G(O_i,M_e)$, where the vertex set, O_i, represents all the operations present in the schedule. The edge set, M_e, contains all those edges that represent mutually exclusive operations of the same type (i.e. operations which do not execute concurrently therefore they can share the same processor). Thus adjacent vertices can execute on the same processor. Introducing a clique coverage maximally groups operations to processors. This operation is repeated for all operation classes, and is shown in figure 3.16.

Register Allocation: A graph is defined $G(V_i,M_e)$ where the vertex set, V_i represents all values that require storage. The edge set, M_e, contains all those edges between vertices that represent mutually exclusive values (i.e. the two values do not overlap in the schedule therefore they can be stored in the same register). Thus, adjacent vertices can be stored in the same register. By introducing a clique coverage, values can be maximally grouped into registers. This process is shown in figure 3.17.

Interconnect Allocation: A graph is defined $G(D_t,M_e)$, where the vertex set, D_t, represents all data transfers present throughout the schedule. The edge set, M_e,

contains all those edges that represent mutually exclusive transfers (i.e. the transfers take place at different times in the schedule therefore the transfers can use the same interconnect).

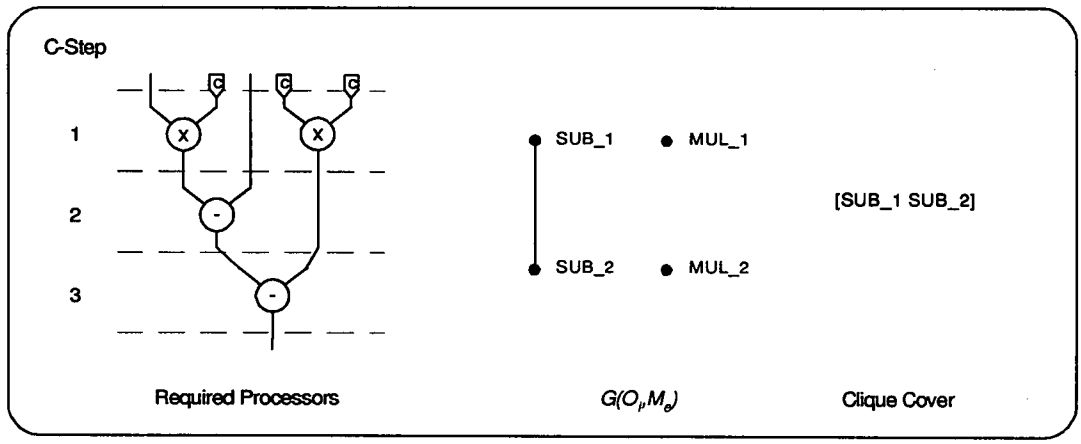


Figure 3.16 Processor allocation by clique coverage.

Thus adjacent vertices can use the same interconnect. Introducing a clique coverage maximally groups data transfers to interconnect. This process is shown in figure 3.18.

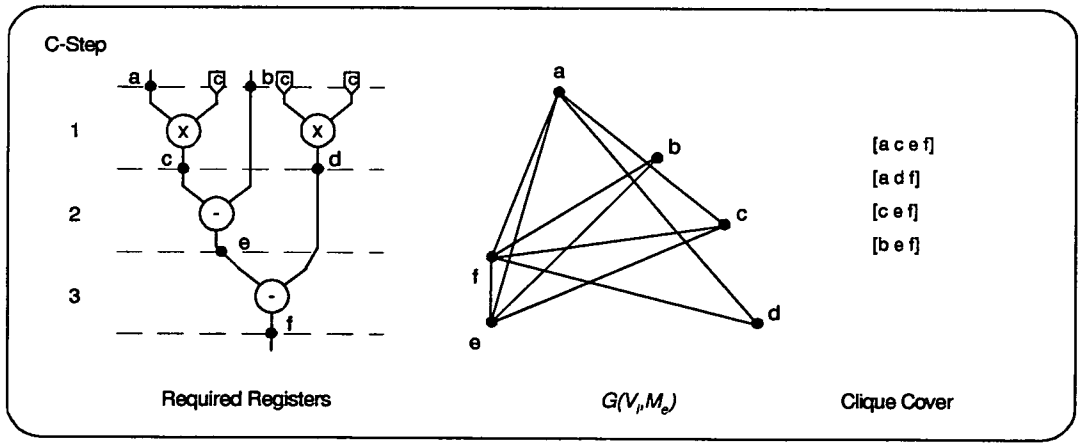


Figure 3.17 Register allocation by clique coverage.

Selecting Appropriate Clique Coverage

In many cases, there is no definitive clique cover. Rather, a group of cliques is produced. Introducing a heuristic selection technique [Tseng86] can eliminate non-

optimal cliques. In the case of the register allocation, Tseng uses a heuristic which selects the maximal clique, thus maximising the register utilisation.

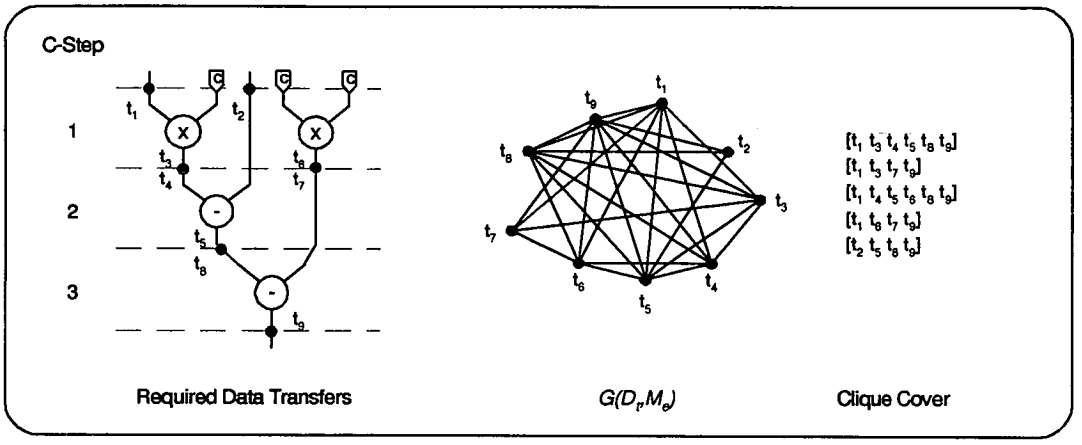


Figure 3.18 Interconnect allocation by clique coverage

A more common approach is to use weighted clique coverage. Tseng advocates this approach during processor allocation an interconnect allocation in the FACET toolset. In both instances, a hierarchical weighting scheme is adopted. For processor allocation, a four level weight is introduced according to the degree of similarity in the source and destination units for each vertex (a high weighting is assigned to inputs and outputs with the same source and destination; correspondingly, a low weighting is given to dissimilar sources and destinations). This scheme is also used in the interconnect allocation for data transfer source and destinations. The HAL system [Paulin89c] uses weighted clique partitioning to perform register allocation. Paulin computes the weighting values as a function of the saved interconnect area after register merging. This is shown in figure 3.19.

A general clique coverage technique has been examined which can be applied to all datapath allocation stages. It is important to note, however, that in Tseng's original treatise, the order in which the various allocation subtasks are completed is specified: register allocation, processor allocation, and finally interconnect allocation. This

ordering allows the heuristic weighting scheme to operate as the weights are derived directly from the register allocation.

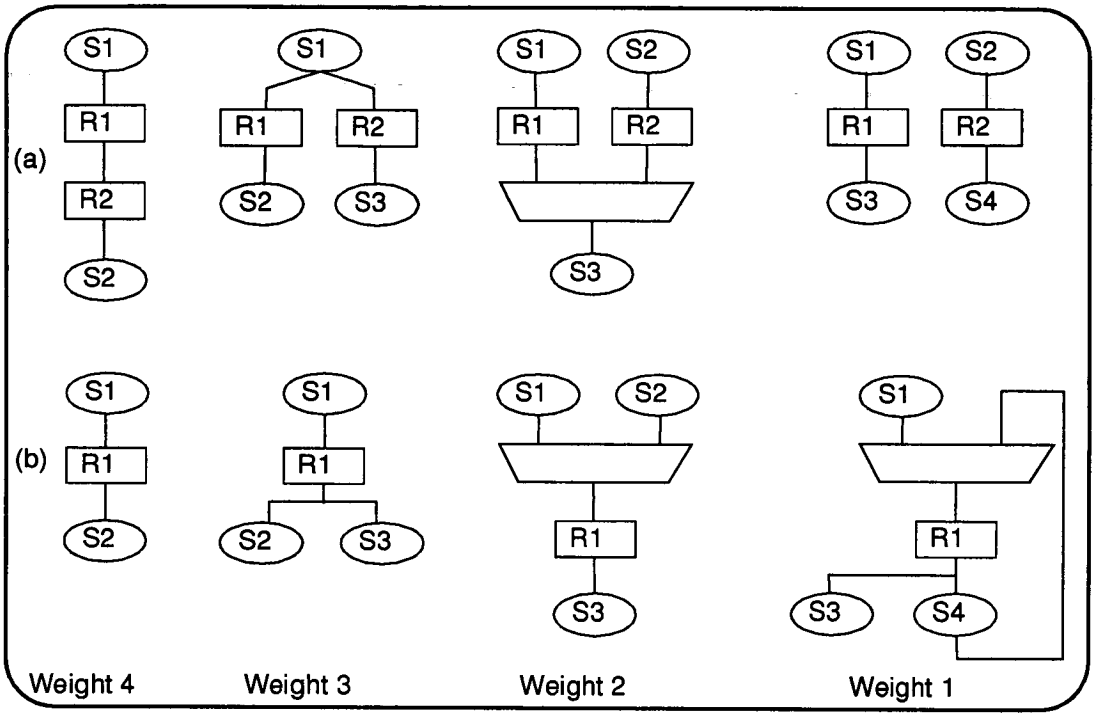


Figure 3.19 Register merging, before (a), and after (b). (after [Paulin89c])

3.2.2 The Left Edge Algorithm.

Kurdahi and Parker [Kurdahi87] have shown that the register allocation problem can be modelled as channel routing. In this representation, the goal is to assign values (wires) to registers (tracks) using the minimum number of registers. This is a well understood problem, and is documented in Hasimoto [Hasimoto71]. The left edge algorithm will always produce the minimum number of registers required, but does *not* guarantee optimal value grouping within those registers. Its operation is shown in figure 3.20.

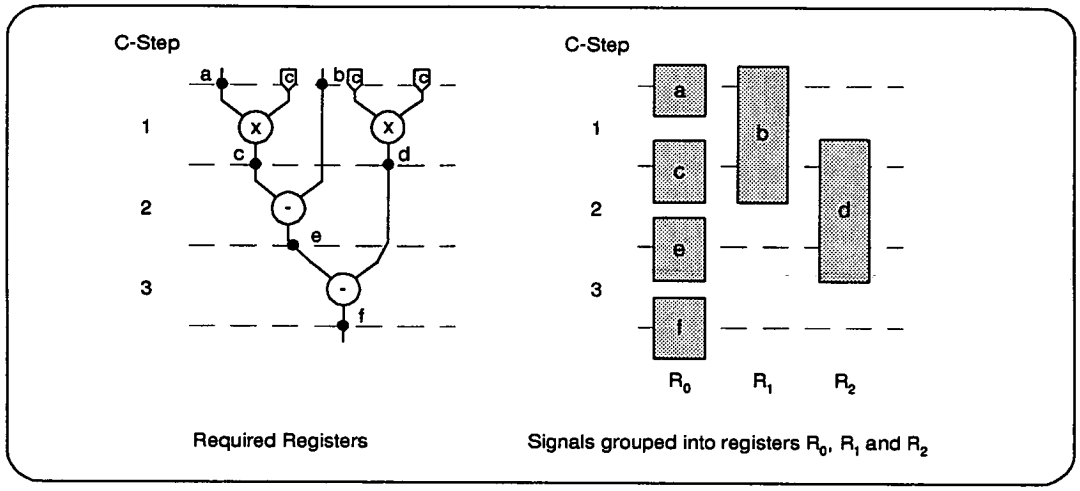


Figure 3.20 Left-Edge register allocation.

Values are allocated to register on a "first available basis" as shown. The lifetime analysis which this graph provides has been incorporated in a further graph theoretic algorithm for register allocation.

3.2.3 Bipartite Matching

A graph is defined $G(V,R,M_e)$ whose vertex sets V and R represent the values to be stored and the available registers, respectively. R is determined via a lifetime analysis derived from the left edge algorithm outlined above. Edges are added between values and register if and only if there are no lifetime conflicts on that register, as shown in figure 3.21. In effect, this matching enumerates all value-register combinations as opposed to the first available matching given by the left edge algorithm.

Once again, however, heuristics must be employed to select the most appropriate allocation. In Huang [Huang90b], the heuristics estimate the number of similar interconnects for source and destination pairs.

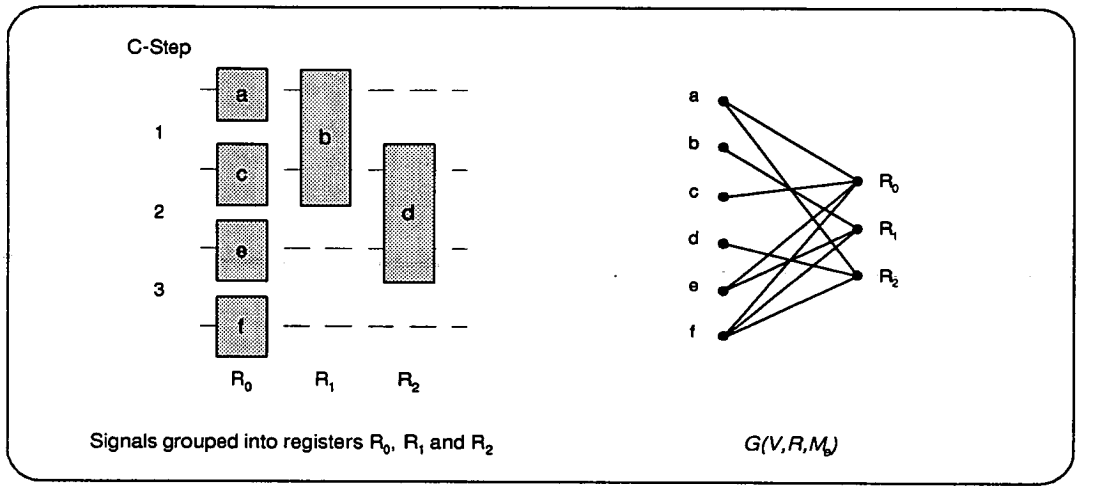


Figure 3.21 Bipartite register allocation.

3.2.4 Edge Colouring

Stok [Stok90a, Stok91] defines a data path allocation scheme which deals explicitly with the register grouping problem.

A graph is defined $S(C_S,WR_t)$, where the vertex set, C_S, represents all control steps present in the schedule. The edge set, WR_t, contains edges which represent the read and write times for storage values. Vizing's theorem [Vizing64] for general graph colouring[1] problems states that for a graph of degree Δ (maximum number of edges incident to a node) and multiplicity M, (maximum number of edges joining any two vertices), then :

$$\Delta \leq \Psi_e(G) \leq \Delta + M \qquad [3.7]$$

where $\Psi_e(G)$ is the number of colours required to colour the graph edges[2]. Therefore, in the register grouping problem, equation 3.7 states that the variables may be grouped into, at most, $\Delta+1$ register files. This is shown in figure 3.22.

1. The graph colouring problem is concerned with finding a partition of the set of vertices into a minimum number of independant sets. Such a partition is called a *colouring*.
2. In this application of Vizing's theorem, there is no meaningful interpretation where M >1.

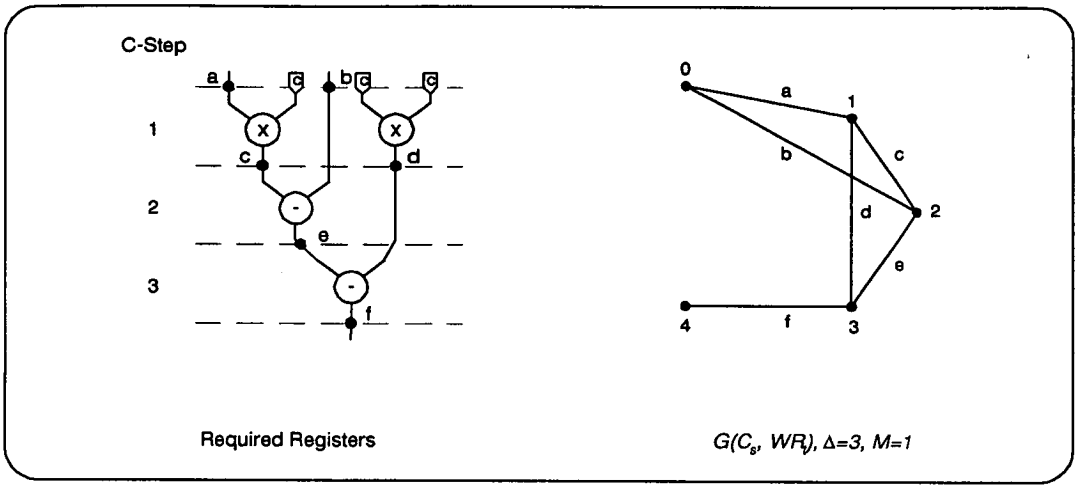


Figure 3.22 The edge-colouring algorithm for register grouping.

Introducing a two phase clocking scheme produces a bipartite graph, Vizing's theorem can be reduced to :

$$\Psi_e(G) = \Delta \qquad [3.8]$$

This is shown in figure 3.23.

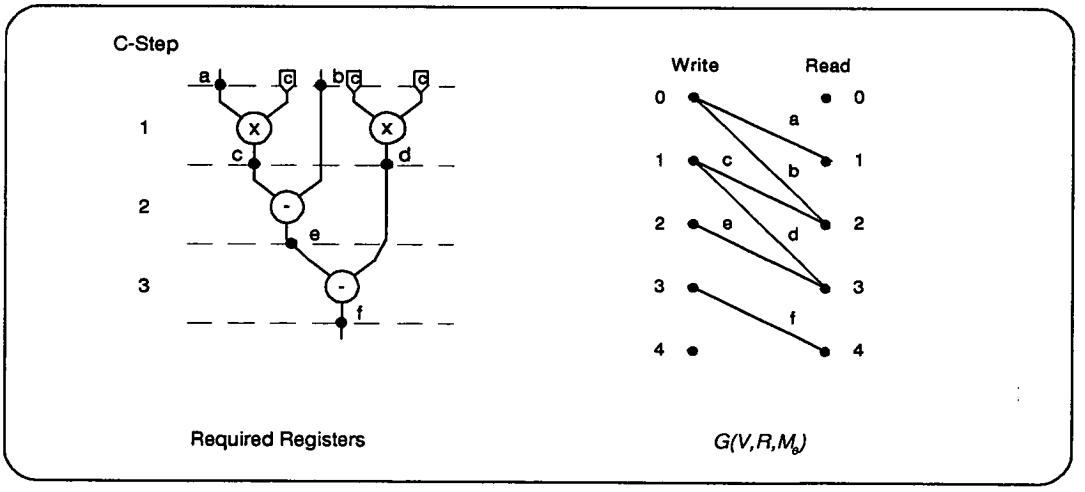


Figure 3.23 Bipartite edge colouring.

3.2.5 Expert System/Greedy Allocation Schemes

The expert system approach to the allocation problem applies rules generated by an

expert designer on an initial allocation of processors, register and interconnect. After applying a decision criterion, the inference engine of the expert system will apply further predicates aimed at minimising the overall objective function.

JACK-THE-MAPPER [Goosse88] is a three level expert system embedded within the CATHEDRAL II system. The outer level is a standard expert system shell interface, with the intermediate level containing predefined predicates capable of performing specific algorithmic tasks, such as bus merging. The rule base itself contains over 100 transformation rules. These transformations allow the modelling of multiplication as shift and add operations, and the generation of counters for loop structures, for example.

As with all expert system approaches, significant user interaction is required, with the design engineer producing the processor allocation manually. Registers and interconnect are allocated initially, one per value and data transfer, respectively. JACK-THE-MAPPER then performs the optimisation transformations. The designer can influence these translation steps by writing architectural pragmas in the input SILAGE description. The allocation is completed during the scheduling phase with the Atomics tool.

Greedy allocation algorithms select operations from the data flow graph. Processors, register and interconnect are all allocated when needed. In most cases, the algorithm aims to locally optimise the cost of introducing operations into the existing datapath.

The node selection order and the local costing criteria characterise greedy allocation schemes. Nodes may be selected at random, via some prioritising function (e.g. critical path analysis), or in the order determined by the schedule. The MABAL system [Kucukc89] selects operations using the latter technique. A more global analysis may

be adopted, with all data flow nodes considered, regardless of scheduled order. This approach is typified by EMUCS [Thomas88], part of the System Architect's Workbench. The EMUCS cost function selects the functional unit with the lowest binding cost (i.e. the cost of adding operations) per operation. Similarly, the MABAL system determines local cost via an analysis of the partial architecture already generated. The MABAL system improves the accuracy of the cost function by formulating the register and functional unit allocations problems together.

3.2.6 Other Techniques

Branch and bound allocation schemes have been developed and implemented [Pangrle88, Marwed86]. They search through datapath components already instanced to provide an allocation solution. The search depth of branch and bound is generally controlled by passing the scheduled data flow graph on a control step by control step basis.

ILP techniques have been applied to the various allocation subtasks [Hafer83], but are restricted to impractical problem instances.

3.2.7 Discussion

The graph theoretic algorithms presented provide the most elegant solution to the allocation subtasks. Based on well established theory, a number of efficient algorithms have been developed for their solution. In the case of clique coverage and bipartite matching, however, the introduction of heuristics will degrade the performance of the algorithms in a restricted subset of applications.

Viewing the allocation task as a series of separated problems produces solution schemes where one component of the allocation task dominates. Indeed, interconnect

and memory based optimisation approaches have been developed [Stok91, Park89, Grant 90a]. The EASY system [Stok88] uses correlated clique coverage to try and account for the effects of the other allocation phases.

The rule based and greedy allocation approaches allow incremental datapath construction. Expert system based allocation is slow, and requires a large rule base. Without efficient backtracking, this approach is more likely to produce solutions based on local minima. The experience of JACK-THE-MAPPER, and Kowalski's DAA [Kowal85] indicate that for a restricted application area, this approach is applicable. Greedy allocation is strongly dependent on the order in which the optimisation takes place, and yields poor quality results.

Finally, the branch and bound and ILP formulations have proved too computationally expensive to implement for practical problem instances.

4 Combinatorial Optimisation and Simulated Annealing

The techniques surveyed in chapter 3 provide methods for optimising the binding between behaviour and structure. The partitioning of the synthesis procedure into a number of well understood tasks enables standard algorithmic techniques to be brought to bear on these subproblems. The introduction of heuristics in the solution of the subproblems, in particular in the selection of appropriate clique coverage tends, however, to introduce *local minima* into the solution. (The definition of a local minimum in the context of this thesis is presented in section 4.1).

This thesis advocates a global approach to the behavioural synthesis problem. This chapter lays the foundations for the formulation of the behavioural synthesis procedure as a *combinatorial optimisation problem*, and reviews local and global solution techniques for this problem class. A candidate technique, known as *simulated annealing*, is introduced and evaluated. A simulated annealing based behavioural

synthesis system, developed at the University of California at Berkeley is then presented.

Some basic nomenclature is introduced as a precursor to the formulation of a combinatorial representation of the behavioural synthesis procedure.

4.1 Nomenclature and Definitions

Two types of optimisation problems exist. The first type, whose solution is selected from a set of real numbers or functions is known as a *continuous optimisation problem.* The second may be thought of as a "one of many selection"; selection of an object from a finite set of discrete candidate objects, and is known as a *combinatorial optimisation problem* [Papadim82].

Definition 4.1 *The solution domain of a combinatorial optimisation problem, P, is defined as a set of tuples (D,c) where D represents a set of discrete solution instances, and c is the cost function, such that*

$$c : D \rightarrow R^n$$

where R^n represents n dimensional real vector space.

From this definition of the solution space, a criterion may be stipulated which leads to the global optimum:

Definition 4.2 *The solution selected from P satisfies :*

$$c(s) \leq c(y) \text{ for all } y \in D,\ s \in D$$

s is called the ***globally optimal*** *solution.*

The solution space may be subdivided into manageable sections by introducing the concept of a *locality.*

Definition 4.3 *A locality, L, within an optimisation problem, P, with tuples (D,c) is defined as a mapping :*

$$L : D \rightarrow 2^D \text{ for each } D.$$

The locality concept is used in discriminating between local and global optima. Global optima satisfy definition 4.2, while local optima are defined :

Definition 4.4 *Within a locality, L, and given a tuple (D,c), a solution is called locally optimal with respect to L if*

$$c(d) \leq c(g) \text{ for all } g \in L(d)$$

Papadimitriou [Papadim82] provides a suitable example of global and local optima in the context of a 1-dimensional Euclidean optimisation problem.

Example 4.1 *Let the problem tuple (D,c) be defined*

$D = [0,1] \subseteq R^1$ *and c be defined as shown in figure 4.1*

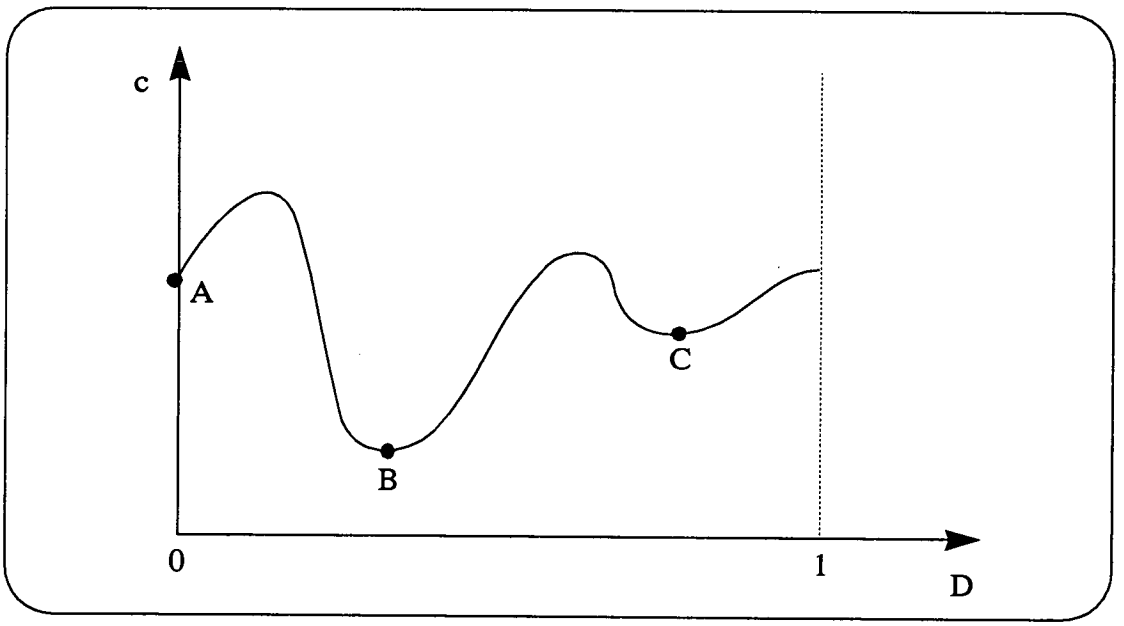


Figure 4.1 Local and Global Optima.

In this instance, let the locality be defined as a distance, ε, from the test points, A, B and C. Thus, if ε is small, then A, B and C are local optima, with B the global optimum.

4.2 Searching The Solution Space

Two potential search strategies of the combinatorial solution space are now described. For the sake of brevity, they are labelled *local* and *global* search. Both algorithms are based upon a basic state generation mechanism which provides a random perturbation to the current state, and is followed by an assessment of the cost, or goodness, of the newly generated state. A decision criterion is then applied to determine whether the new state is accepted. It is at this decision criterion where the two techniques become divergent.

A simple example is presented to illustrate the difference. Figure 4.2 shows the state graph representing all 1-change permutations of the set {a,b,c}, excluding the empty set. The directed arcs point to the lower cost next states. The cost hierarchy in this example is defined to be: $c(1) < c(5) < c(3) < c(6) < c(2) < c(4)$.

If LOCAL_SEARCH is defined as shown in figure 4.3, with a cost assessment criterion which accepts lower cost solutions only, then given an initial state of S(6), the next acceptable state transformation will result in either S(1) or S(5). If S(5) is the state selected, then there will be no acceptable lower cost state transformations available to the algorithm, which will then terminate in a local minimum. Thus, it may be seen that the success of this greedy heuristic search strategy is strongly dependent on the initial solution state. Also, the upper bound on the computation time is unknown for many problems. (The worst case time complexity of LIN_LOCAL_SEARCH [Lin65] remains uncomputed. This procedure a search strategy for the travelling salesman problem (TSP [Dantzig54]).)

For a global search strategy to succeed where LOCAL_SEARCH terminates in a local minimum, some acceptance of lower quality solutions (i.e. solutions with a higher cost

function) must be attained.

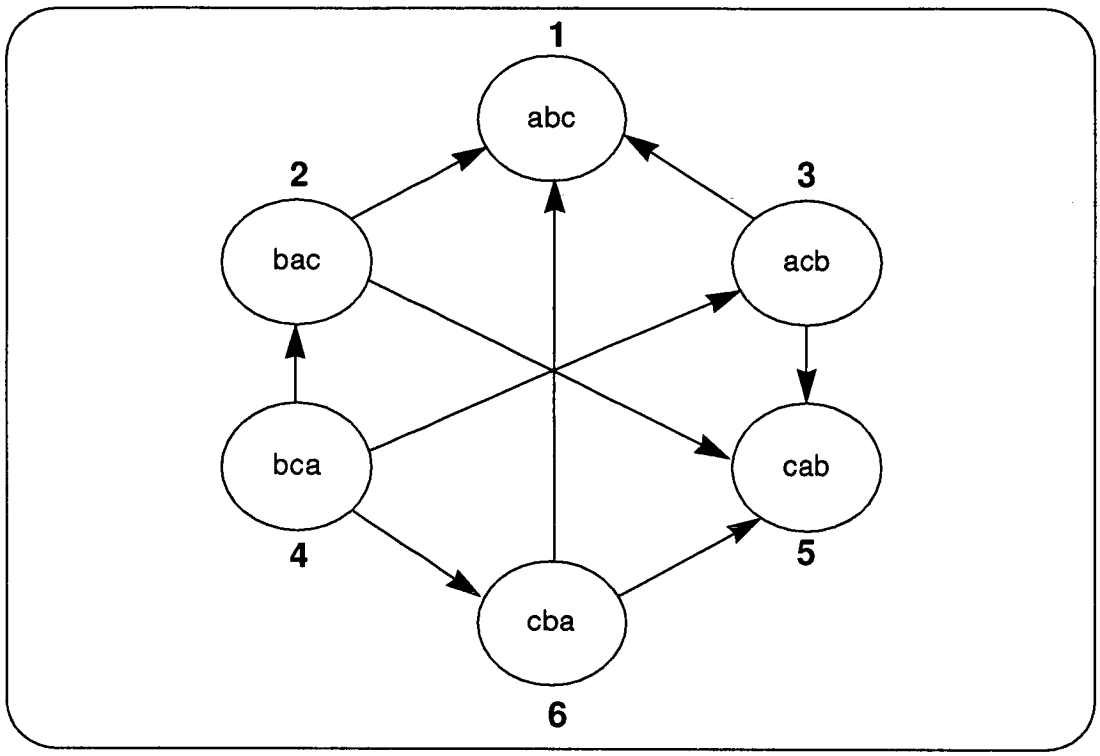


Figure 4.2 1-change state graph for {a,b,c}.

```
function LOCAL_STATE(i0 : in STATE);
  i,j : STATE;

begin

  i := i0;
  repeat
      GENERATE(i,j);
      if cost(j) < cost(i) then i := j;
  until cost(j) >= cost(i) for all possible state GENERATIONS;

end LOCAL_SEARCH;
```

Figure 4.3 LOCAL_SEARCH function.

In the case of the example, if all potential higher cost moves are marked in (figure 4.4.), it may be seen that independent of the initial placement within the solution

space, a global minimum can be attained. It is this *hill climbing* property of global search algorithms which makes their use in combinatorial optimisation problems attractive

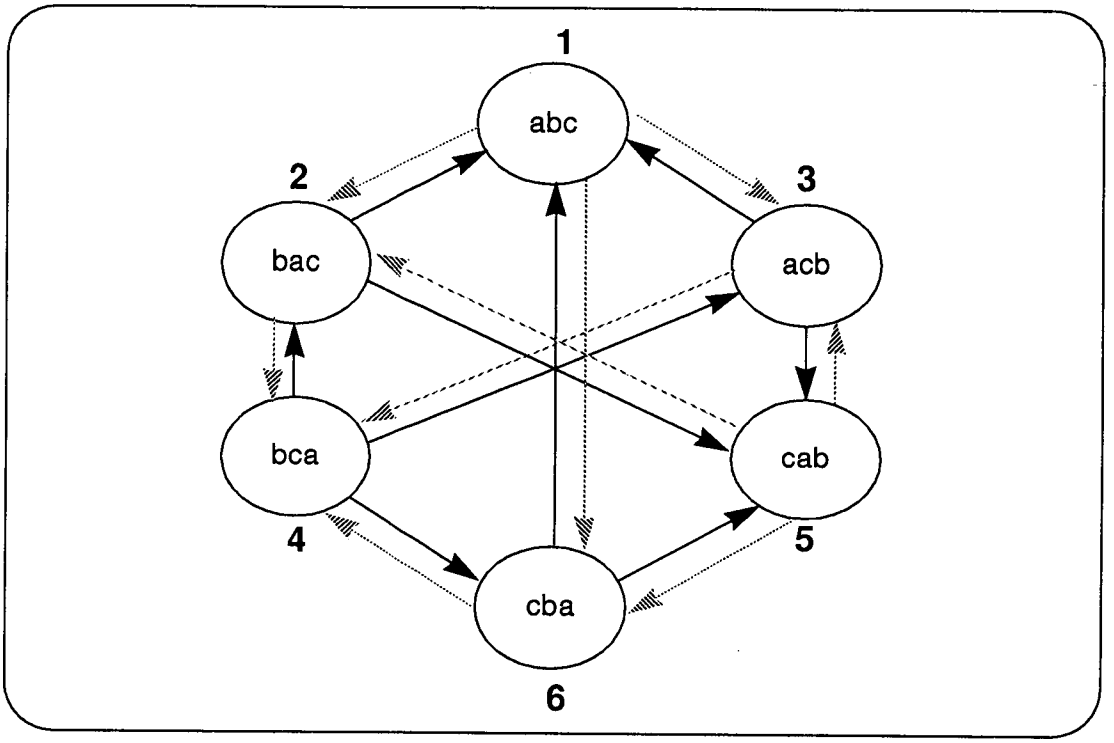


Figure 4.4 1-change state graph for {a,b,c} with hill climbing moves.

4.3 The Simulated Annealing Algorithm

The simulated annealing algorithm belongs to the class of algorithms known as *probabilistic hill climbing* algorithms, which display the desirable qualities described in the previous section. The simulated annealing algorithm is based upon the Metropolis Monte Carlo Method [Met53, Bin78], a technique which first found prominence in the field of statistical mechanics.

The Monte Carlo method is a computational technique used to simulate the attainment of thermal equilibrium in a cooling solid. For a given solid with state i, and internal energy E_i, the Monte Carlo method generates a candidate state, j, by introducing a

random perturbation to the current state. This newly generated state has internal energy E_j. If ΔE, where $\Delta E=E_j-E_i$, is less than or equal to zero, then the newly generated state is accepted unconditionally. If ΔE is greater than zero, then the state transition from i to j is accepted according to the following criterion, known as the *Metropolis Criterion*:

$$state\,(i) \rightarrow state\,(j) \text{ if } e^{\left(\frac{\Delta E}{k_B T}\right)}, \text{ when } \Delta E > 0. \qquad [4.1]$$

where T is the temperature of the current state, and k_B is the Boltzmann constant.

If the temperature is lowered slowly enough, then thermal equilibrium can be attained for each temperature. The Monte Carlo Method models this by generating a sufficiently large number of random state transitions for each value of T.

Kirkpatrick, Gelatt and Vecchi [Kirk83] adapted the Monte Carlo Method for the simulation of the physical annealing process, described above, as a solution technique for general combinatorial optimisation problems. Returning to the nomenclature of Section 4.1, definition 4.1 states that the solution domain of a combinatorial optimisation problem is defined as a set of tuples, (D,c), with D representing a set of discrete solution instances, and c a domain cost function. A control parameter, k, is the analogue of temperature. The underlying equivalence suggested here is that a physical particle based system can be modelled as a combinatorial solution domain, and that the associated cost function is an analogue of internal system energy.

The basic simulated annealing algorithm is given in pseudo-code format in figure 4.5.

The acceptance function, *ACCEPT()*, generates a random number between 0 and 1 using a uniform distribution.

If the number generated is less than the value of $e^{\left(-\frac{\Delta E}{k}\right)}$, then the state transition is accepted; if the random number is greater, then the state transition is rejected. The pseudo-code for this function is given in figure 4.6.

```
procedure SIM_ANNEAL (S0 : in STATE; K0 : in CONTROL)is

  S    : STATE;          -- The state variable
  TEMP : STATE;          -- The temporary generated state
  K    : CONTROL;        -- Temperature analogy

  begin

  S := S0;
  K := K0;

  -- While loop looks for global optimum

  while (not (STOPPING_CRITERIA)) loop
      for COUNT in 1 .. M loop
          -- Generate a new state
          TEMP := GENERATE();
          -- Compute cost difference
          COMPUTE_DELTA_E;
          -- Lower cost state; always accept.
          if (ΔE < 0) then S := TEMP;
          -- Higher cost state; may generate a hill climbing move
          else
             if (accept) then S := TEMP; end if;
          end if;
      end loop;
      -- Update control parameter
      UPDATE(K);
  end loop;
end SIM_ANNEAL;
```

Figure 4.5 The Simulated Annealing Algorithm.

From these definitions, it may be seen that the rate of convergence of the algorithm is dependent upon the selection of the following: M, the inner loop criterion (i.e. the number of states generated to simulated the attainment of thermal equilibrium); the function *UPDATE()*, which decrements the value of k, the control parameter; k_0, the initial value of k; and the selection of an appropriate stopping criterion for the algorithm. Communally these items are known as a *cooling schedule*.

```
function ACCEPT() return BOOLEAN is

  R : REAL;

begin
  R = RANDOM(1);
  if (R < e^(-ΔE/k)) then
     RETURN true;
  else
     RETURN false;
  end if;

end ACCEPT;
```

Figure 4.6 Pseudo-code state acceptance function.

While this thesis is primarily concerned with the pragmatic application of the simulated annealing algorithm to the behavioural synthesis problem, a brief review of current cooling schedule techniques is presented.

4.3.1 Cooling Schedule Techniques

Reaching Thermal Equilibrium

Two major approaches are well-documented for the determination of M, the number of states generated at each value of the control parameter necessary to simulate thermal equilibrium.

The first uses fixed-length Markov chain modelling [Aar85] to determine when equilibrium has been reached. That is, a fixed number of state generation trials are attempted at each control parameter value. Aarts [Aar85] states that a *quasi-equilibrium* is achieved if, for a Markov chain (k), of length L_k, and cost parameter c_k:

$$\left\| a\left(l_k, c_k\right) - q\left(c_k\right) \right\| \qquad [4.2]$$

where $a(L_k,c_k)$ is the probability distribution of the solutions after L_k trials of the k^{th} Markov chain, and $q(c_k)$ is the stationary distribution of the Markov chain at c_k.

The equilibrium condition determined by Huang [Huang86] is based on the following observation: When equilibrium is established, the ratio of the number of new states generated whose cost is within a range, δ, from the average cost ($\overline{C}$) to the total number of newly accepted state transformations will reach a stable value, λ. For high values of k, the cost distribution is observed to be close to a normal distribution. A range of state generations whose cost lies within the range $(\overline{C}-\delta, \overline{C}+\delta)$, known as the *within count,* is established. The ratio of this range, χ, to the number of newly accepted state transformations is given by :

$$\chi = erf\left(\frac{\delta}{\sigma}\right) \qquad [4.3]$$

where ***erf()*** is the error function [Fel70]. In this system, the development of the equilibrium condition is based upon the selection of the within count, and the application of a maximum count value. Equilibrium is attained if the within count is reached before the maximum count is exceeded. If the maximum count is exceeded, then both counters are reset, and the process repeats. The value of δ is set to be such that the final state at any value of k is close to the average cost ($\delta < \sigma$).

Updating the Control Parameter

The simplest technique for the UPDATE() procedure which decrements the control parameter is to implement an exponential function such as :

$$k_n = k\alpha(k) \text{ , with } 0 < \alpha(k) < 1 \qquad [4.4]$$

where k_n is the new value of the control parameter, and k is the current value. This technique is proposed in Kirkpatrick [Kirk83]. Sechen [Sechen88] reports that the most effective cooling schedules are generated when α lies between 0.8 and 0.99. This method is inefficient for high values of k, however, where the current state is generally many transitions away from the final, minimum cost solution. An alternative is to use

an adaptive technique.

Huang [Huang86] uses the relationship between average cost of the current system configuration ($\overline{C}$) against the logarithm of the value of the control parameter for that configuration. Plotting this *annealing curve* allows the control parameter to be set such that $\overline{C}$ decreases in a uniform manner. The slope of that curve is given by :

$$\frac{d\overline{C}}{dln(k)} = k\frac{d\overline{C}}{dk} \qquad [4.5]$$

In his treatise on statistical and thermal physics, Reif [Reif65] states:

$$\frac{d\overline{C}}{dk} = \frac{\sigma^2}{k^2} \qquad [4.6]$$

If a linear approximation to the curve, as shown in figure 4.7, is used, substitution of equation 4.6 into equation 4.5 yields:

$$\frac{\Delta C}{ln(k') - ln(k)} = \frac{\sigma^2}{k} \qquad [4.7]$$

Rearrangement of equation 4.7 leads to :

$$k' = k \cdot exp\left(\frac{k\Delta C}{\sigma^2}\right) \qquad [4.8]$$

A key assumption used in the derivation of equation 4.5 is that equilibrium can be maintained provided that $\Delta C < \sigma$. Substituting $\Delta C = -\lambda\sigma$, where $\lambda \leq 1$, into equation 4.8 yields :

$$k' = k \cdot exp\left(-\frac{\lambda k}{\sigma}\right) \qquad [4.9]$$

This derivation represents one of the most commonly used decrement functions.

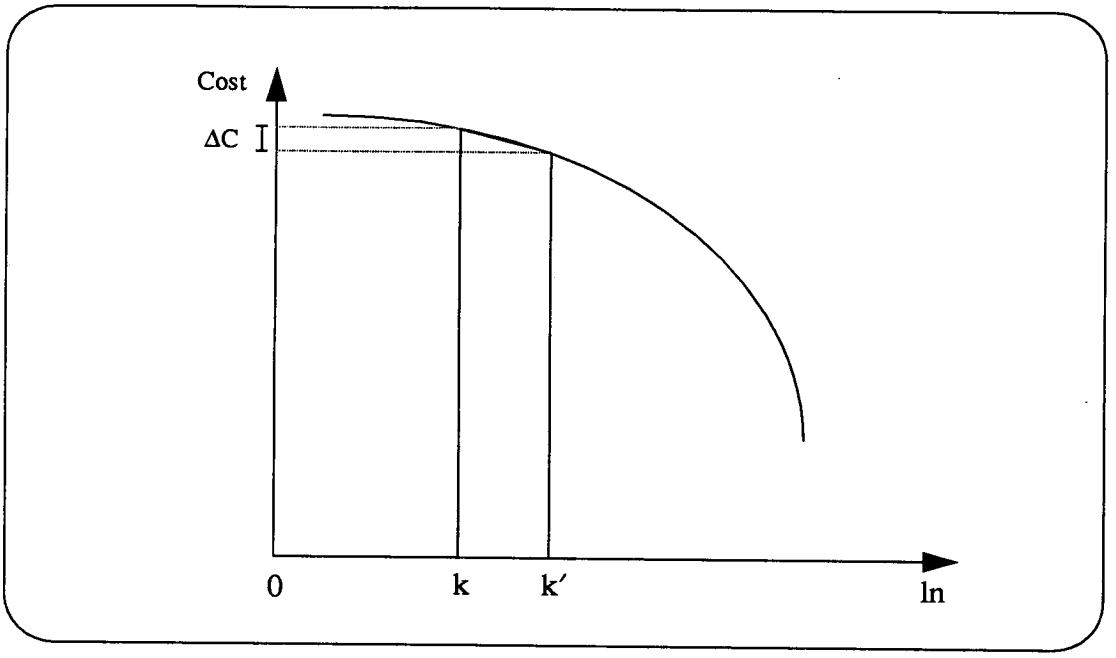


Figure 4.7 Linear approximation technique used in [Huang86].

Initial Control Parameter Selection

The initial value of the control parameter, k_0, should be selected such that all state transitions can be accepted. In the domain of physical annealing, this corresponds to heating the solid up until rearrangement at the atomic level can take place freely. When translated into the simulation domain, the criteria here must ensure that at the initial value almost all possible state transitions should be accepted. Defining an acceptance ratio $\chi(k)$ at the k^{th} transition:

$$\chi(k) = \frac{\text{number of accepted transitions at k}}{\text{number of proposed transitions at k}} \qquad [4.10]$$

hence:

$$\chi_0 \approx 1 \qquad [4.11]$$

The actual value of χ_0, known as the *initial acceptance ratio* may vary from 0.95 - 0.98 [Aar89]. For a sequence of trial state generations at a value of the control parameter, k, let $C1$ represent the number of trails that result in a decreased (or

unchanged) cost function and $C2$ represent the number of trials that result in an increase. Let Δ represent the average difference in cost for all trials in $C2$. The acceptance ratio can be approximated [Aar85]:

$$\chi = \frac{(C1 + C2)\, e^{\left(-\frac{\Delta}{k}\right)}}{(C1 + C2)} \qquad [4.12]$$

Rearrangement of 4.12 leads to:

$$k = \frac{\Delta}{ln\left(\frac{C2}{C2\chi - C1\,(1-\chi)}\right)} \qquad [4.13]$$

The control parameter, k, is set to zero initially. A sequence of trial state generations, M then takes place. After each trial, equation 4.13 is used to calculate a new value of k, with $\chi = \chi_0$. C1 and C2 correspond to the total number of increasing and decreasing cost state generations obtained. When $M = C1 + C2$, then the value of k at that point is taken as the initial value of the control parameter, k_0. Published results [Aar89] indicate that this method provides fast convergence to k_0.

Stopping Criterion

The stopping criterion can be implemented in a simple, but effective form. The cost of equilibrium states (C) for a number (Π) of successive control value parameters is compared. The algorithm is terminated if :

$$C_i = C_j \quad for\ all \quad i, j \in \Pi \qquad [4.14]$$

In practice, the value of Π can be particularly low, typically 3. Markov chain analysis [Aar89] reinforces this observation.

4.4 The Class NP and Behavioural Synthesis

The simulated annealing algorithm has been shown to be effective for a wide range of combinatorial and NP (**N**on-**P**olynomial time) complete problems [Cook71, Karp72]. While an in-depth survey of the theory of NP-completeness lies outside the scope of this thesis, a brief discussion is presented.

NP complete problems may be defined as a class of computational problem for which there is no *exact* solution capable of being produced by a polynomial algorithm. As a corollary to the above statement, it should be noted that if there is a polynomial algorithm for any NP-complete problem, then there is a polynomial solution for *all* NP-complete problems. The theory of NP-completeness therefore provides a convenient watershed between tractable and intractable computational problems. At best, the amount of computational effort required to solve an NP-complete problem will be exponential, and therefore impractical for all but trivial problem instances.

In assessing whether a problem is in the class NP, polynomial transforms are applied to the problem to try to reduce it to a known NP-complete form. The mechanics of this reduction lie outside the scope of this thesis, but it should be noted that in many cases the transformation process can be as time consuming as producing an approximate (or heuristic) solution to the problem. Gary [Gary79] provides a comprehensive and readable introduction to, and survey of, NP-complete problems. A brief summary is presented below of known NP-complete problems which form constituent parts of the behavioural synthesis procedure.

The MAX_CLIQUE problem (section 3.2.1), is known to belong to the class NP [Cook71]. This illustrates the need for heuristics to be applied in the selection of the most appropriate clique coverage. Without the introduction of heuristics, for problems

of practical size, the MAX_CLIQUE problem would remain intractable. Also related to the allocation procedures, the GRAPH_COLOURING problem [Law76, Holyer81] is NP-complete, along with a variant of the matching problem, 3-D_MATCH [Karp75]. Further, the MULTIPROCESSOR_SCHEDULING [Ullman75] problem belongs to NP. Even techniques used to reduce the complexity of the data flow structures encountered in the behavioural synthesis procedure [Fin92], reduced to the PARTITION problem have also been shown to be NP-complete [Karp75].

4.5 Simulated Annealing and Behavioural Synthesis

A simulated annealing based algorithm, originating at the University of California at Berkeley [Dev89], formulates the datapath generation algorithm as a two dimensional placement of microinstructions on a grid representing available hardware resources. This representation is a common thread throughout all simulated annealing based systems, and it represents an extension of the successful application of the simulated annealing algorithm to the field of global placement and routing of blocks in a VLSI layout environment [Sechen88].

The Berkeley core synthesis system takes a description of the input algorithm in terms of explicit statements of sequential, parallel and disjoint blocks, as shown in figure 4.8a.

The specification of *disjoint* statements allows mutually exclusive code segments to execute on the same hardware resource.

The function used to compute the cost of the intermediate datapaths is simply a weighted sum of the areas of constituent datapath parts, namely generic ALUs (the Berkeley system does not use dedicated hardware blocks), registers and buses or links,

along with a weighting which incorporates the overall system execution time (figure 4.8b). From this definition, it may be seen that the Berkeley system represents an optimisation over a restricted hardware architecture. In many cases, especially in signal processing architectures, a bus based system can introduce a significant overhead in terms of silicon area.

The ALU cost function is not simply a sum of all operation costs associated with the particular ALU. Devedas and Newton [Dev89] observe that an ALU capable of performing both addition and subtraction is only marginally bigger than an ALU capable of performing addition alone. With this in mind, a *cost table* for all potential ALU functions along with register and bussing costs was defined. A typical cost table is shown in figure 4.9.

A similar rationale is applied when assigning costs to registers. The table sets a threshold so that designs using large register files are penalised more heavily. This is also used when assigning interconnect costs.

In the cost function defined in figure 4.9, *p3* is a function of the number of registers used in the design, while *p4* is a complex function of both the number of registers and ALUs in the data path. The values in the cost table are derived from an evaluation of the change in layout area given incremental addition of registers, interconnect and register numbers for a given layout style. The resultant function is piecewise linear, and Devedas and Newton use a data set small enough to given an improvement in accuracy of solution over a linear approximation.

```
(serial
   (parallel
      (add x1 y1 z1)
      (add x2 y2 z2)
   )
   (parallel
      (mult z1 y3 z3)
      (minus z2 y4 z4)
   )
   (disjoint
      (divide z3 x3 z5)
      (divide z4 x4 z5)
   )
)
```

(a)

```
Cost = p1 * (#ALU) + p2 * (execution_time) + p3 * (#register)
+ p4 * (bus)
```

(b)

Figure 4.8 (a) Berkley input behavioural description (b) Berkely cost function

```
#cost of different operations in an ALU
     ALU
     add 50
     sub 50
     fadd 100
     mult 250
     add minus 60

     #register costs
     REGISTER
     #starting from register 1, each has a cost of 10 units
     1 10
     #starting from register 5, each has a cost of 15 units
     5 15

     #execution time
     EXECUTION
     1 50
     50 50

     #bus costs
     1 100
     3 150
```

Figure 4.9 Berkeley costing table.

4.6 Discussion

Simulated annealing was introduced as an effective technique for generating high quality optimised solutions for general purpose combinatorial optimisation problems. By considering the potential solution space for a behavioural synthesis problem as an enumeration of all possible state transitions, and hence all possible datapaths, the behavioural synthesis problem can be represented as a combinatorial optimisation problem.

Chapter 3 presented algorithmic techniques for the solution of subtasks within the behavioural synthesis procedure. This partitioning is known to lead to the generation of locally optimal solutions through the introduction of heuristics. The simulated annealing algorithm avoids this by accepting the generation of inferior states, and therefore can effectively 'climb out' of local minima.

The controlling algorithm itself is known to be simple and robust. The mechanics of the annealing process are well understood, and can be analytically modelled. A wide variety of general purpose cooling schedules exist, and have been proven to be particularly effective in the field of VLSI layout.

Pioneering work on the use of the simulated annealing algorithm in the field of behavioural synthesis has taken place, and the result is a working system capable of taking an algorithmic description and producing a datapath optimised for area performance based around a costing function using a weighted sum to compute datapath costs for a restricted hardware architecture.

The remainder of this thesis develops this work into a system capable of producing optimised solutions for both area and speed performance over a full range of datapath components.

5 Simulated Annealing Based Synthesis Techniques

This chapter develops a set of datapath synthesis techniques based upon the simulated annealing algorithm introduced previously. These techniques are drawn together in the SAVAGE (a Simulated Annealing based VLSI Architecture GEnerator) toolset, which is a modular software package capable of synthesising an RTL description of a datapath from a fragment of behavioural code.

The chapter is organised into a number of separate sections. The first introduces a novel data structure, capable of sustaining an extended range of optimisation moves[1], and which also permits a simple costing method. The application of the simulated annealing algorithm to the behavioural synthesis task is then considered. As simulated annealing is a general optimisation technique, little code modification is required.

1. The random perturbations of the current state of the system described in section 4.3 are known as optimisation *moves*. Where different types of moves are available, they are grouped into *move sets*.

The next two sections address the core of the synthesis method presented in this thesis. In the first, a basic move set is developed; this set covers scheduling, processor allocation and memory and interconnect optimisation. Linked to the datapath move set is the mechanism by which the quality of the solution is assessed. The costing method proposed by Devedas and Newton [Dev89] is extended, and a novel cost-multiplier system introduced.

The SAVAGE tools are described, and Paulin's differential equation example is presented as a brief illustration of the method.

5.1 Data Structures

In common with the majority of simulated annealing based applications (solution of the Travelling Salesman Problem [Aar89], VLSI block placement and global routing [Sechen88]), the solution space is presented as a grid structure, or *plane*.

The Berkeley system [Dev89], described in section 4.5, represents the datapath synthesis problem as a placement of microinstructions in a two-dimensional grid whose axes correspond to the available hardware resource and control steps respectively. Memory and communications optimisation takes place *after* the optimisation move has been applied, 'downstream' in effect, from the scheduling and allocation problem. Consequently, no optimisation moves are brought to bear directly on those synthesis subtasks. Thus the Berkeley model retains a single degree of freedom - that of placement of microinstructions in a *Resource-time*[2] space.

The model described here offers three degrees of freedom by providing separate planes for the simulated annealing algorithm to operate on. These planes correspond to

2. This terminology first occurs in Denyer [Denyer89].

a *Resource-time* space, a *Memory-time* space, and a representation of the communications infrastructure required to complete the datapath known as the *Port-connection* space. While complete decoupling of the planes is impossible, the annealing algorithm will be demonstrated to operate effectively within each.

5.1.1 Resource-time Space

Resource-time (*R-t*) space records the current schedule for each operation present in the input data flow graph, and the processor that each operation is currently allocated to. *R-t* space is a two-dimensional array indexed by processor resources available to execute dataflow operations, and the control steps in which those operations are executed.

An operation, *O*, can exist at *R-t* location *R-t[processor, c-step]* if, and only if, the following conditions are true:

(i) All predecessor nodes (*pred(O)*) execute in the range [*1 .. c-step-1*], and all successor nodes (*succ(O)*) execute at time, *t*, such that $t > c\text{-}step$.

(ii) The processor is capable of executing the operation type. (i.e. $O.type \in processor.type$[3]).

By adhering to these conditions, *R-t* space models data flow precedence and ensures processor binding correctness. The data flow fragment of figure 2.1 can be (arbitrarily) mapped[4] into *R-t* space as shown in figure 5.1.

5.1.2 Memory-time Space

Memory-time (*M-t*) space records individual signal lifetimes over all control steps in

3. Refer to sections 2.3 and 2.4.1.
4. The initial mapping of the data flow graph into *R-t* space is a function of the BUILDER module in the SAVAGE toolset, and is described in section 5.5.2.

the current schedule. *M-t* space is a two-dimensional array whose axes are indexed by memory components and control steps respectively. The grid is bounded by the total number of signals present in the data flow graph, and the latest execution time present in the schedule

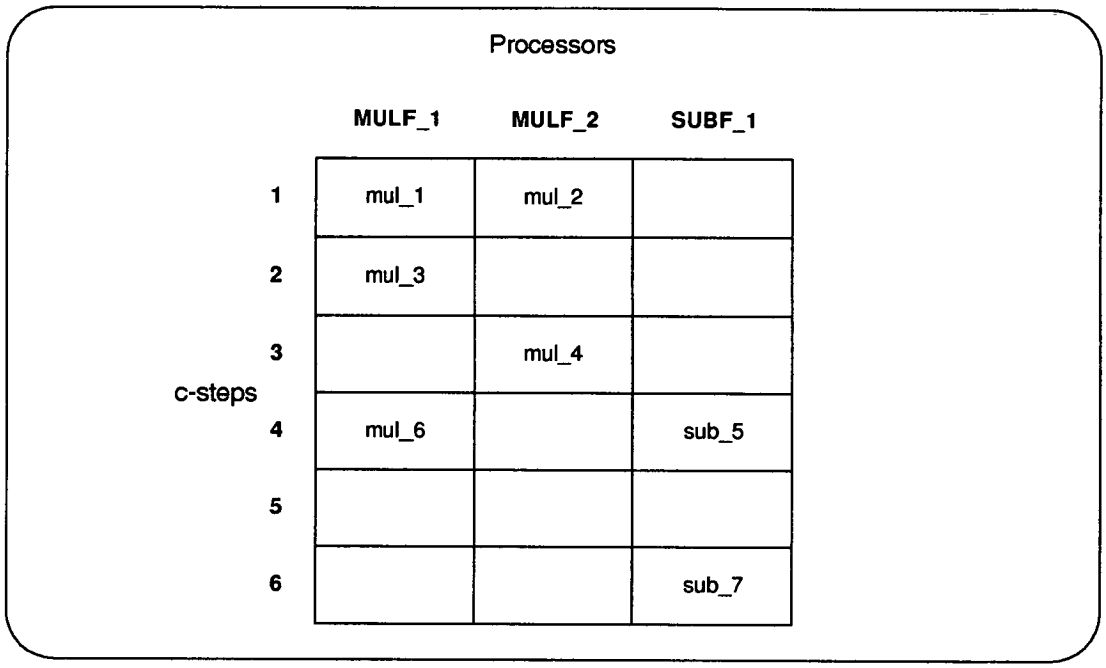


Figure 5.1 Data flow graph from figure 2.8 mapped into R-t space.

Each control step is partitioned into read and write phases. This adheres to the control model developed in section 2.6. Memory components can therefore be reused on a cycle to cycle basis while preserving data integrity. Similarly, signals may share a memory component if, and only if, the lifetimes of both signals are mutually exclusive. Thus the *R-t* space mapping of figure 5.1 produces an *M-t* space as shown in figure 5.2.

5.1.3 Port-connection Space

Port-connection (*P-c*) space records all point to point connections within the synthesised datapath. Further, *P-c* space records all bindings between signals and nets.

P-c space is a two-dimensional array whose axes correspond to all processor and memory input and output ports respectively.

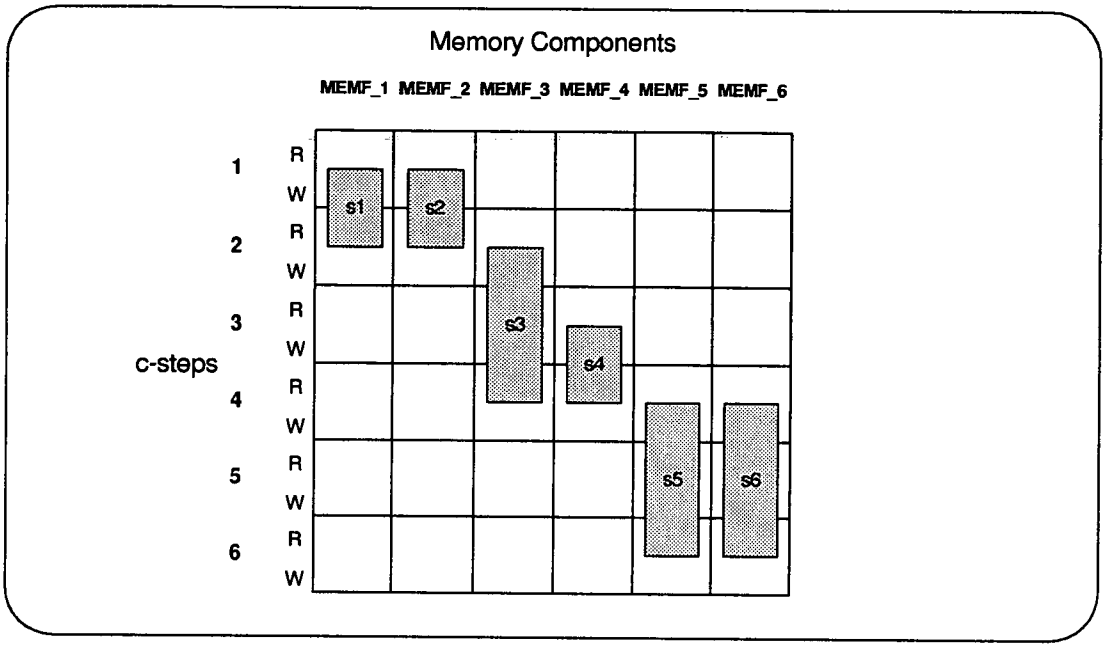


Figure 5.2 M-t space derived from R-t space of figure 5.1

Each array location contains two items:

(i) A Boolean flag. This is set if and only if there is a communications function required between the two ports.

(ii) A pointer to a net instance. If the Boolean flag is set, then the pointer indicates which net carries out the communications function.

The *P-c* space corresponding to the *R-t* and *M-t* mappings in figures 5.1 and 5.2 is shown in figure 5.3. For clarity, the pointers to the net instances are not shown.

5.1.4 Implementation Details

Each of the optimisation planes is described above as a two dimensional array. Because of their dynamic nature, however, it is impractical to implement them as conventional static array structures.

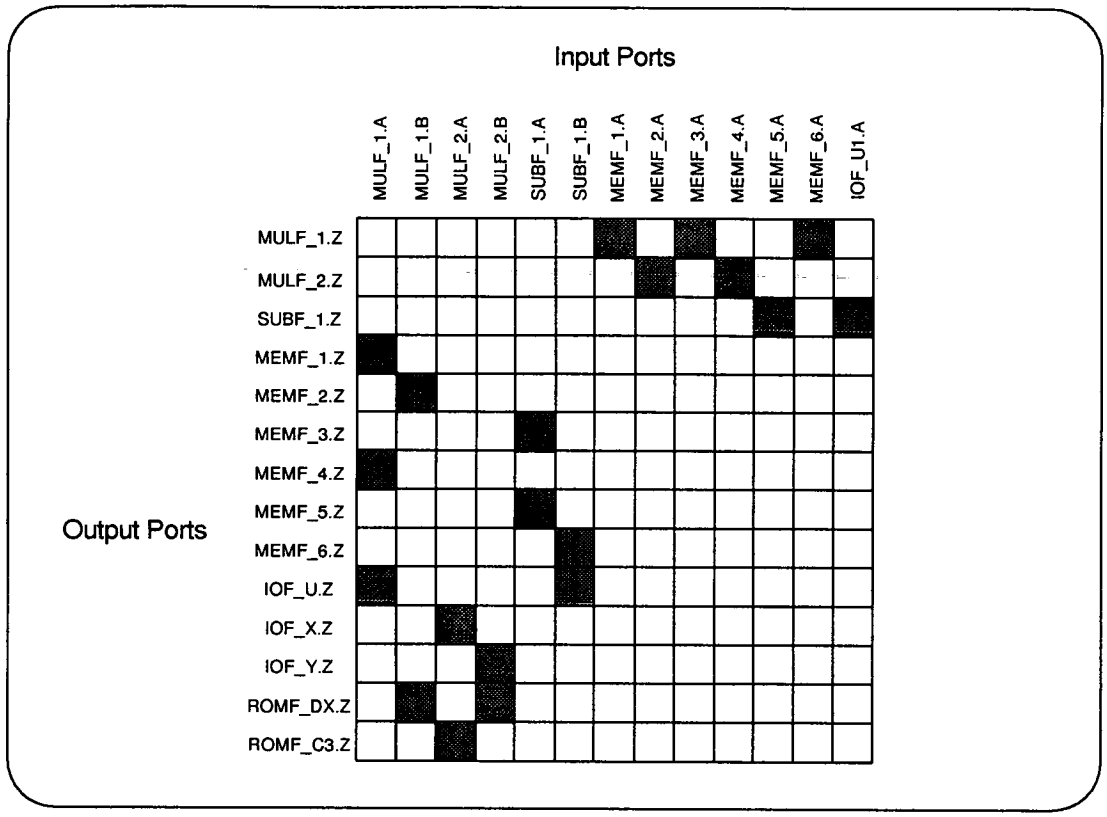


Figure 5.3 P-c space derived from R-t and M-t spaces of figures 5.1 and 5.2.

In order to maintain the maximum flexibility in the selection of scheduling and allocation moves, *R-t* space is implemented as a linked list structure. In this way, processors can be added to and deleted from *R-t* space with a minimal computational overhead. Each processor contains a pointer to another linked list corresponding to those operations currently allocated to execute on that processor. Items on this list are composed of pointers[5] to the input data flow graph, and record fields recording the current execution time. This structure is shown in figure 5.4.

Similarly, *M-t* space is maintained as a linked list of memory components. Each contains a pointer to a linked list corresponding to the signal(s) currently allocated for

5. It is important to note that these pointers are *doubly-linked*, i.e an operation allocation can be determined by beginning the search from the data flow graph.

storage. Again, the list items point to the input data flow graph. This data structure is shown in figure 5.5.

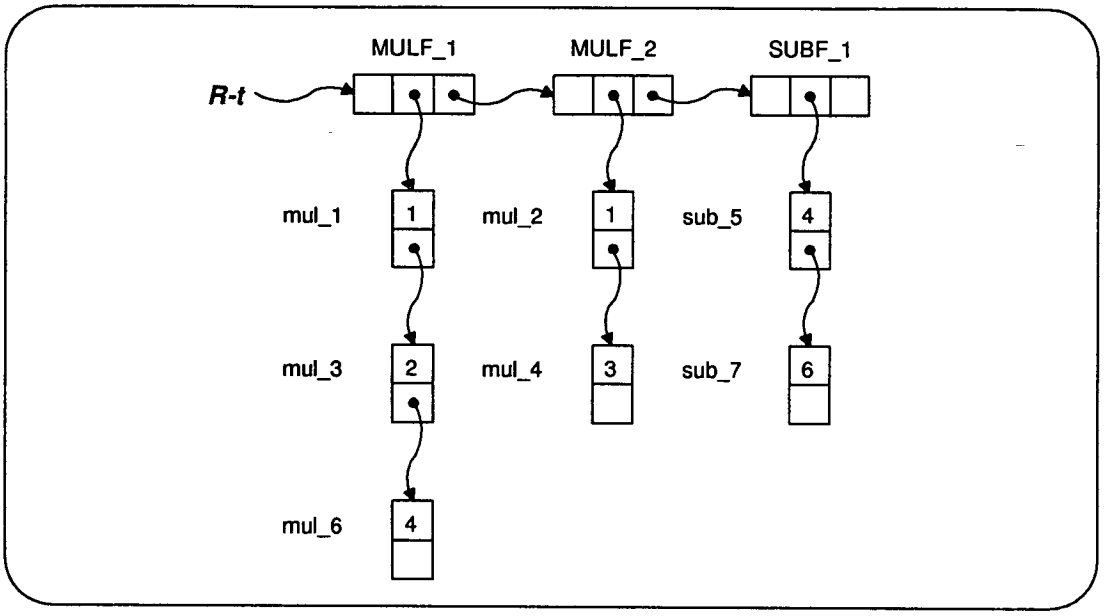


Figure 5.4 Linked list implementation of R-t space shown in figure 5.1

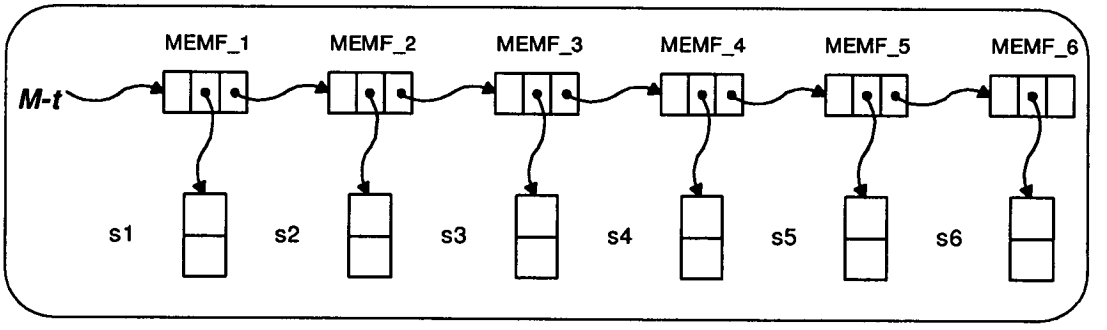


Figure 5.5 Linked list implementation of M-t space shown in figure 5.2.

P-c space is maintained as a linked list of input port records. Each input port record points to a list of output port records. Where a communications function between ports is required, a pointer to the appropriate output port is appended to the output port list of the corresponding input port. Each output port record contains another pointer to the net instance implementing the communications function. These net instances are also implemented as a linked list. Further, each output port record also contains a pointer to

the signal in the input data flow graph requiring the communications function. This structure is shown in figure 5.6 (Again for clarity, the pointers to the net instance list and to the data flow graph are not shown).

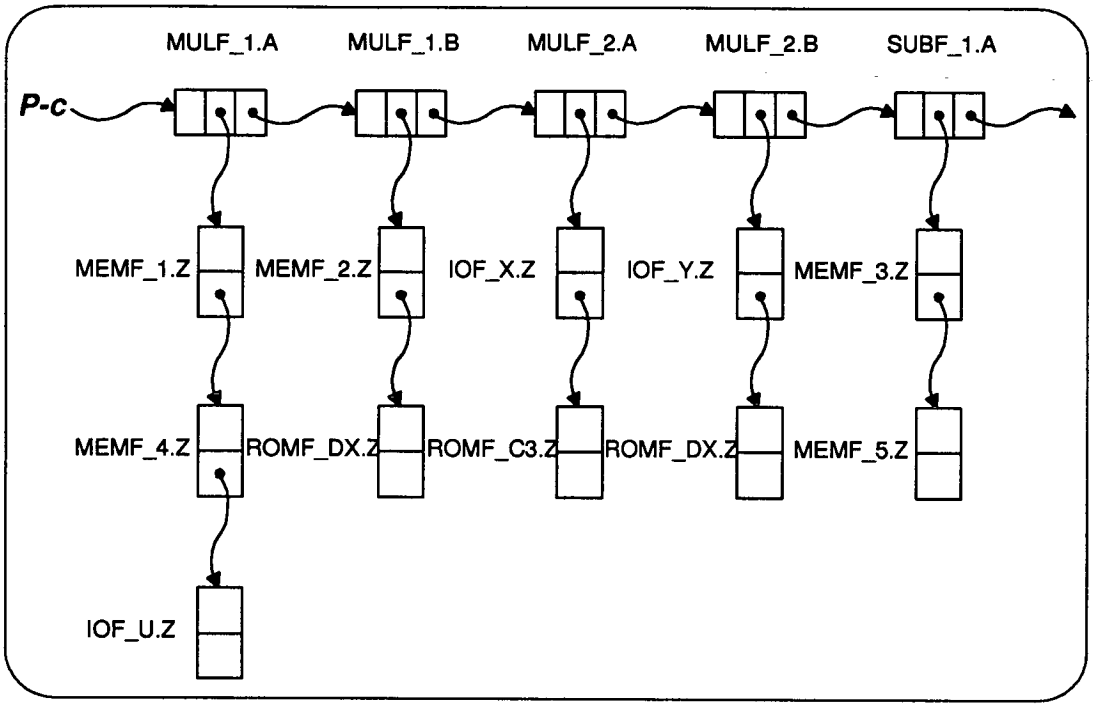


Figure 5.6 Linked list fragment of P-c space shown in figure 5.3

5.2 Core Synthesis

The core synthesis routines comprise the simulated annealing procedure and the cooling schedule used. The simulated annealing algorithm shown in figure 4.5 may be divided into two major phases:

(i) **Initialisation:** The initial system state is generated along with a starting value for the control parameter, k.

(ii) **Iteration:** The state generation and cost assessment loop continues until all stopping criteria are satisfied.

The initial system state is a function of the BUILDER module, described in section

5.5.2. BUILDER seeds operations in *R-t* space, generates an *M-t* space based upon the initial *R-t* mapping and finally computes *P-c* space. These three components form the initial system state.

5.2.1 Initial Control Parameter Value

The initial value of the control parameter, k_0, is determined using the convergence technique described in section 4.3.1. A sequence of 100 states is generated. The control parameter, k, is set to zero initially. A trial state is generated, and the quality of the newly generated state is assessed using the cost function. This value is compared with the previous cost function. The *C1* and *C2* counts are incremented accordingly. Equation 4.13 is then used to generate a new value for k. At the conclusion of the sequence, the value of k is set as the initial control parameter value.

5.2.2 State Generation

The state generation procedure applies a random perturbation to the current system state. This is a five stage process.

(i) **Optimisation plane selection:** A random variable selects either *R-t*, *M-t* or *P-c* based optimisation. Plane selection dictates the type of optimisation move to be carried out; scheduling and allocation, memory optimisation or interconnect optimisation.

(ii) **Operand selection:** If *R-t* space is chosen, then the operand is a node selected at random from the input data flow graph. If *M-t* space is chosen, then a signal is selected at random from the input data flow graph. Finally, if *P-c* space is chosen, then a component input port is selected at random from *P-c* space, or a net is selected from the net instance list.

(iii) **Build move set:** A valid move set for the selected operand is generated.

This set describes all potential movement of the operand in the selected optimisation plane.

(iv) **Operator selection and execution:** A move is selected at random from the generated set. This move is then applied to the selected operand.

(v) **Change propagation:** As a result of operator execution in *R-t* space, the *M-t* and *P-c* spaces may require updating. Similarly if the selected operation acts in *M-t* space, then *P-c* space may require updating. All resultant changes are propagated by the VALIDATE procedure, described in section 5.5.3.

5.2.3 Attaining Thermal Equilibrium

Attaining thermal equilibrium (i.e. establishing a steady-state probability distribution for all state generations) is a two stage process:

(i) Generate *maximum count threshold* and *within count* values.

(ii) Increment counters at each state generation.

Ideally, the maximum count threshold and within count values should be updated dynamically to reflect changes in the cost distribution as the control parameter is decreased. This is infeasible because of the large amount of compute time required to monitor the steady-state condition for all state generations. These parameters are therefore determined prior to the optimisation procedure.

Setting $\delta = 0.5\sigma$ (i.e. the range limit on acceptable generated costs) in equation 4.3 yields:

$$\chi = erf(0.5) = 0.38 \qquad [5.1]$$

The within count is then set to be *0.38(3n)*, where n is the number of nodes (i.e.

datapath components and data flow operations/signals) in the problem. The maximum count threshold is specified as *1 - within count(problem bound) = 0.62(3n).*

At each state generation the state cost is computed. If the cost lies within the range $(\overline{C}-\delta, \overline{C}+\delta)$, where $\overline{C}$ is the average cost, then the within count is incremented. The current number of state generations is also incremented and compared to the maximum count threshold. If the current number of state generations exceeds the maximum count threshold, then both counters are reset to zero and the process continues. If the within count value is attained prior to reaching the maximum count threshold, then thermal equilibrium has been simulated and the state generation loop terminates.

5.2.4 Control Parameter Update

The control parameter is updated in accordance with equation 4.9:

$$k' = k \cdot exp\left(-\frac{\lambda k}{\sigma}\right)$$

Experimental results [Ott84, Lun84] suggest that a suitable value for λ is $\lambda = 0.7$. Substitution in the above equation gives:

$$k' = k \cdot exp\left(-\frac{0.7k}{\sigma}\right) \qquad [5.2]$$

After each state generation sequence, the standard deviation of the cost distributions generated during the sequence is calculated. The resulting value is applied in equation 5.2, and the new value of k calculated.

5.2.5 Stopping Criterion

The stopping criterion used in the simulated annealing core is composed of two major components. The first is the comparison of three successive cost values. At low values

of the control parameter, this criterion indicates a stable datapath structure. The second is concerned with the validity of the datapath generated. The WIRED function (described in section 5.5.3) ensures that there are no wired-OR connections present in *P-c* space.

5.3 Datapath State Generation Move Sets

The state generator selects a move set from those described in the following sections. The state generation moves have been grouped into *scheduling and allocation*, *memory optimisation* and *net optimisation* moves according to the primary plane of operation (i.e *R-t*, *M-t* or *P-c* space).

5.3.1 Scheduling and Allocation

The basic scheduling and allocation move within *R-t* space corresponds to a translation of an operation over either or both axes. A node is selected at random from the input data flow graph. A valid move set for that node is then generated. In this context, a valid move does not violate data flow precedence. The valid move set is a subset of the moves described in the following subsections.

Schedule on current processor (UNARY_STEP)

This optimisation move forces a unary increase or decrease in the operation execution time; the processor allocation remains unchanged. For an operation, *O*, executing on processor, *P*, at c-step *c*, the validity of this move is subject to data flow dependencies and on the availability of *P* at c-steps, *c-1*, *c* and *c+1*. If any of these execution times is unavailable, the optimisation move proceeds using the others. If all are unavailable, the move is invalidated. Where all c-steps are available, the direction of the move is decided on the generation of a random variable (c.f. the state acceptance function, ACCEPT, presented in section 4.3). This move is illustrated in figure 5.7.

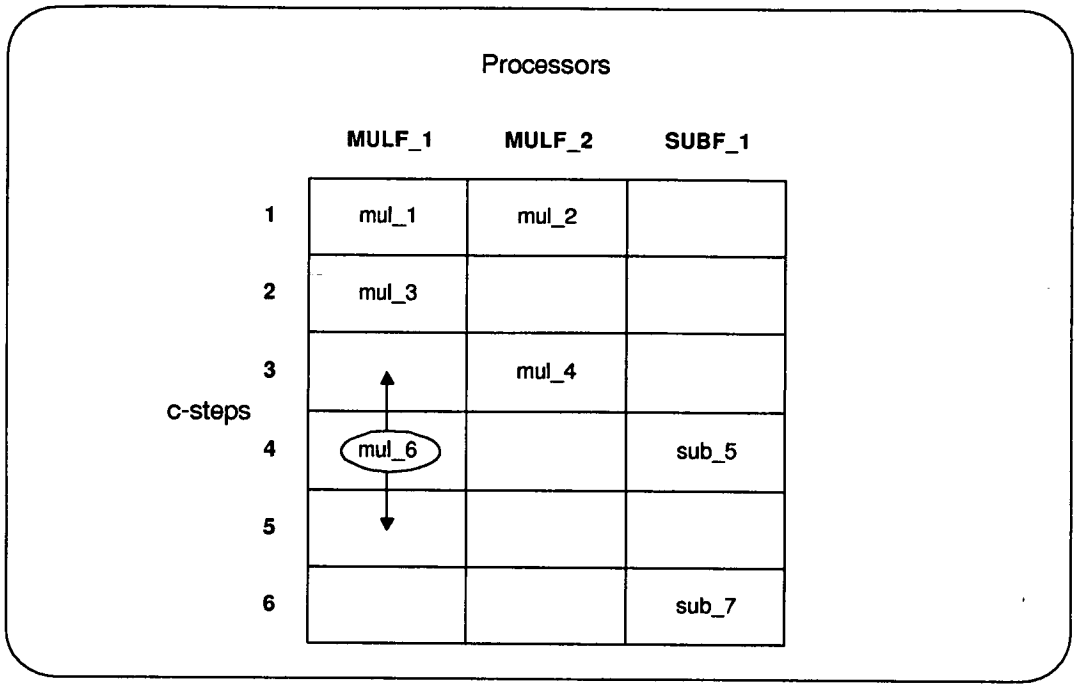


Figure 5.7 Schedule on current processor.

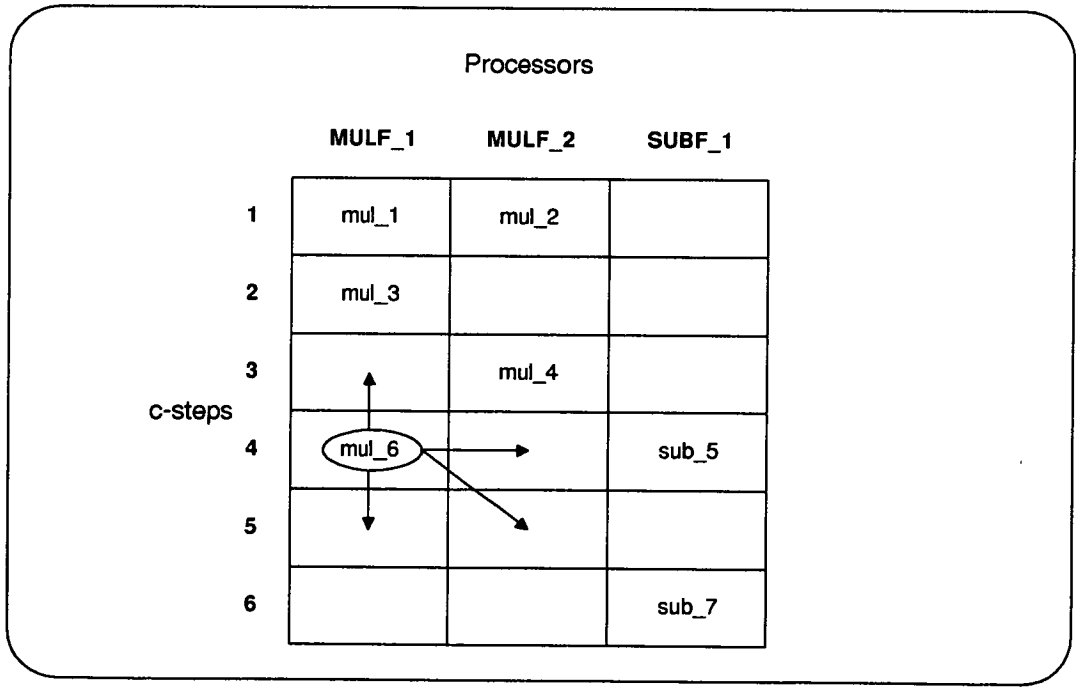


Figure 5.8 Schedule on valid processor.

Schedule on valid processor

This optimisation move may force a unary increase or decrease in the selected operation execution time and a change in processor allocation. For an operation, O, executing on processor, P, at c-step, c, all processors capable of supporting the **operation.type** and of sustaining operation execution at c-steps c-1, c and c+1 are grouped in a subset. A processor is selected at random from this subset. The operation is then allocated to the selected processor. A valid schedule for the operation is determined using the UNARY_STEP function. This move is illustrated in figure 5.8

Create processor

This optimisation move creates a new processor for the selected operation and may force a unary increase or decrease in the operation execution time. A new processor capable of supporting the **operation.type** is created and added to the Resource axis of R-t space. The selected operation is allocated to the newly created processor. A valid schedule for the operation is determined using the UNARY_STEP function. The move is illustrated in figure 5.9.

Function merge

This optimisation move creates a multi-function ALU structure from a dedicated processor. The move may force a unary increase or decrease in the execution time of the selected operation. A subset of processors not currently capable of supporting the **operation.type** is formed. Each processor must be capable of sustaining operation execution at c-steps c-1, c and c+1. A processor is selected at random from the subset, and the **operation.type** is added to the **processor.type** list. The operation is then allocated to the selected processor. A valid schedule for the operation is determined using the UNARY_STEP function. This procedure is illustrated in figure 5.10.

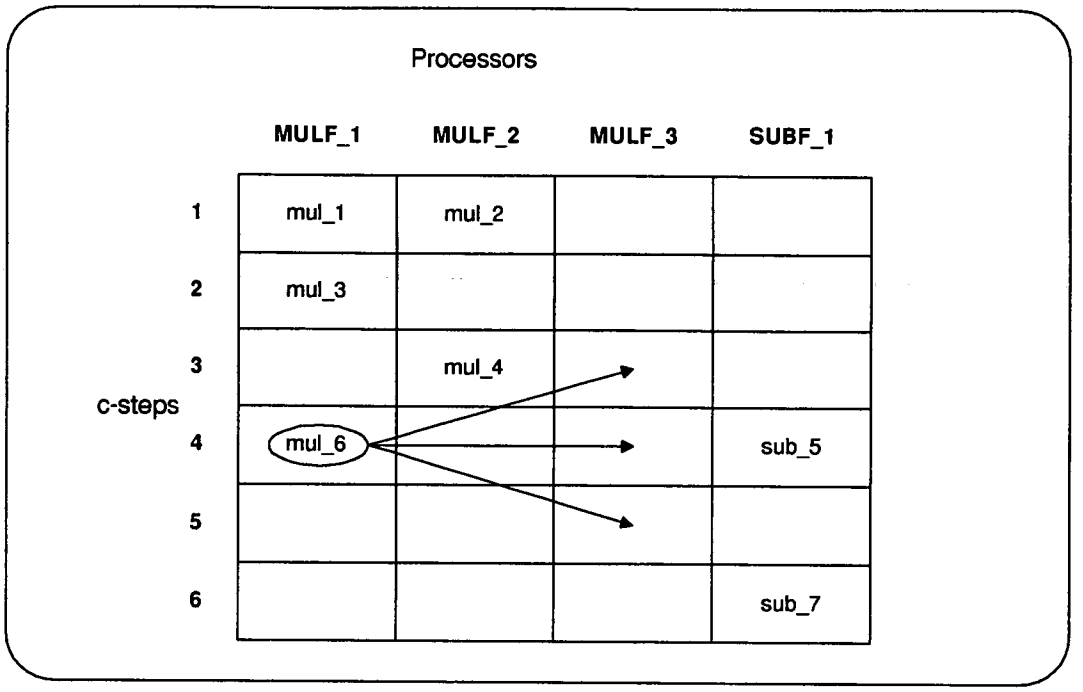


Figure 5.9 Create processor.

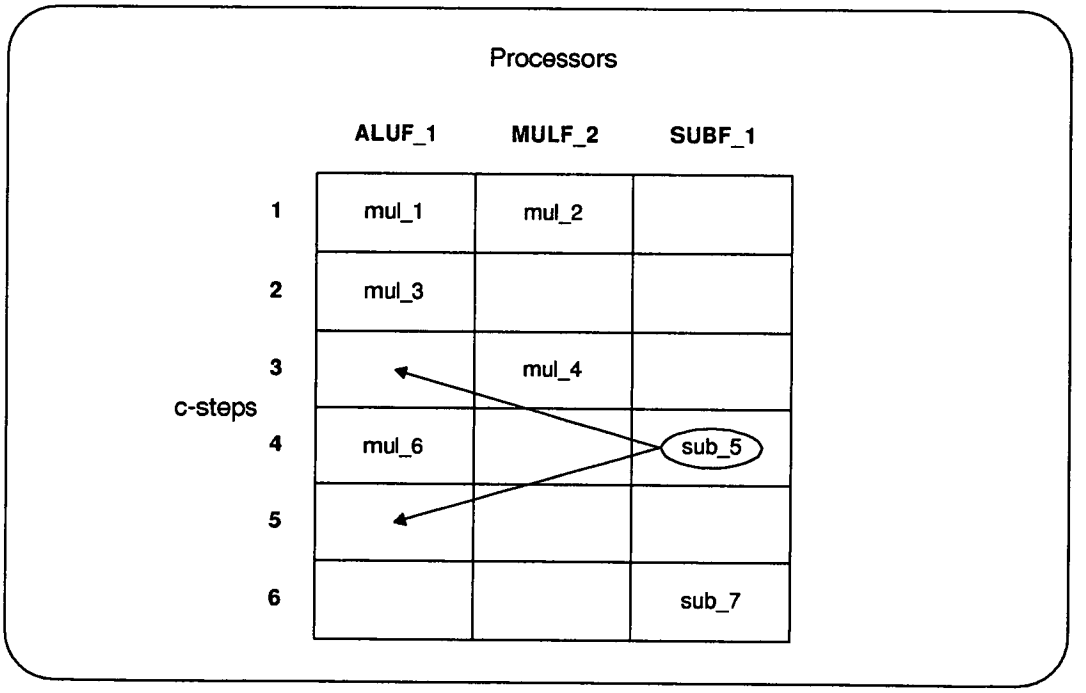


Figure 5.10 Function merge.

5.3.2 Memory Optimisation

Memory optimisation is concerned with achieving the minimum number of memory components required to store all signals declared in the data flow graph. The optimisation move merges multiple signals into a single memory component. The overall effect of this move is similar to that produced by the left-edge algorithm, described in section 3.2.2.

Signal Merge

A signal is selected at random from the input data flow graph. *M-t* space is indexed, and a subset of all memory components capable of storing the signal over its lifetime is generated. A memory component is selected at random from the subset. The signal is then allocated to the selected memory component. This process is illustrated in figure 5.11.

Create Memory Component

A signal is selected at random from the input data flow graph. A memory component of type **register**[6] is added to *M-t* space, and the selected signal allocated to the newly created register. This process is illustrated in figure 5.12.

5.3.3 Optimising *P-c* Space

Optimisation within *P-c* space takes one of three forms. The first is concerned with exercising the commutative law, and may be thought of as a 'port swap' operation, while the second mechanism is concerned with optimising the binding between the communications functionality within the datapath (i.e. point-to-point connections) to the net instances (i.e. wires, multiplexers and buses) required to implement that functionality. Although it is not directly associated with *P-c* space, the last optimisation move is a good example of the relationship between *M-t* and *P-c* space.

6. The *Net Merge* function is able to alter the **memory.type** field.

The final technique merges individual registers into register *file* structures.

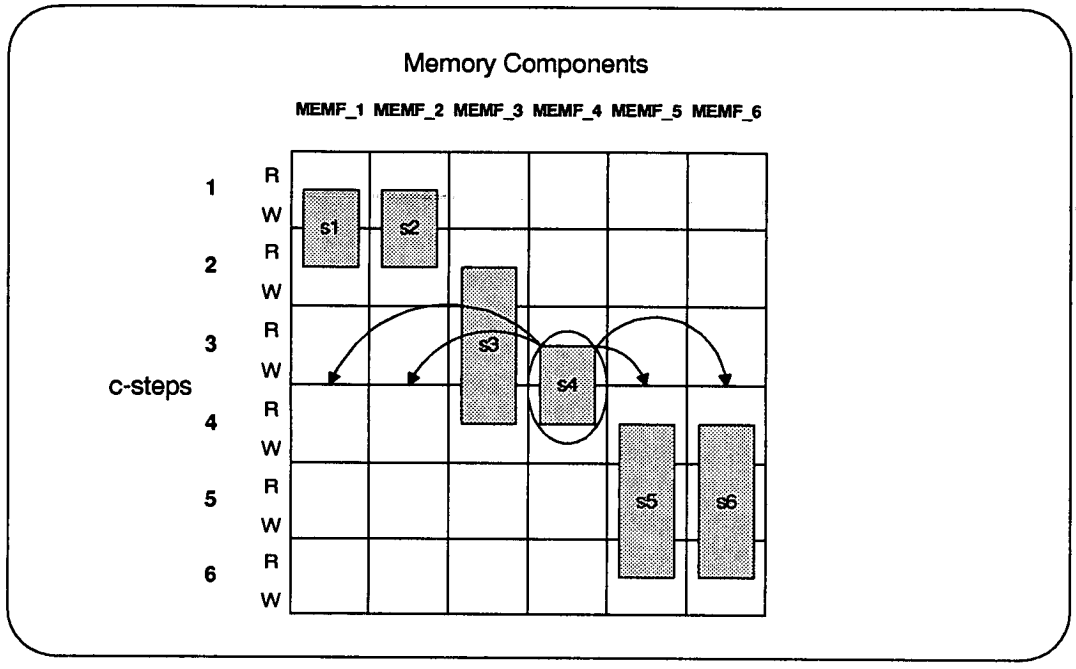


Figure 5.11 Signal merge.

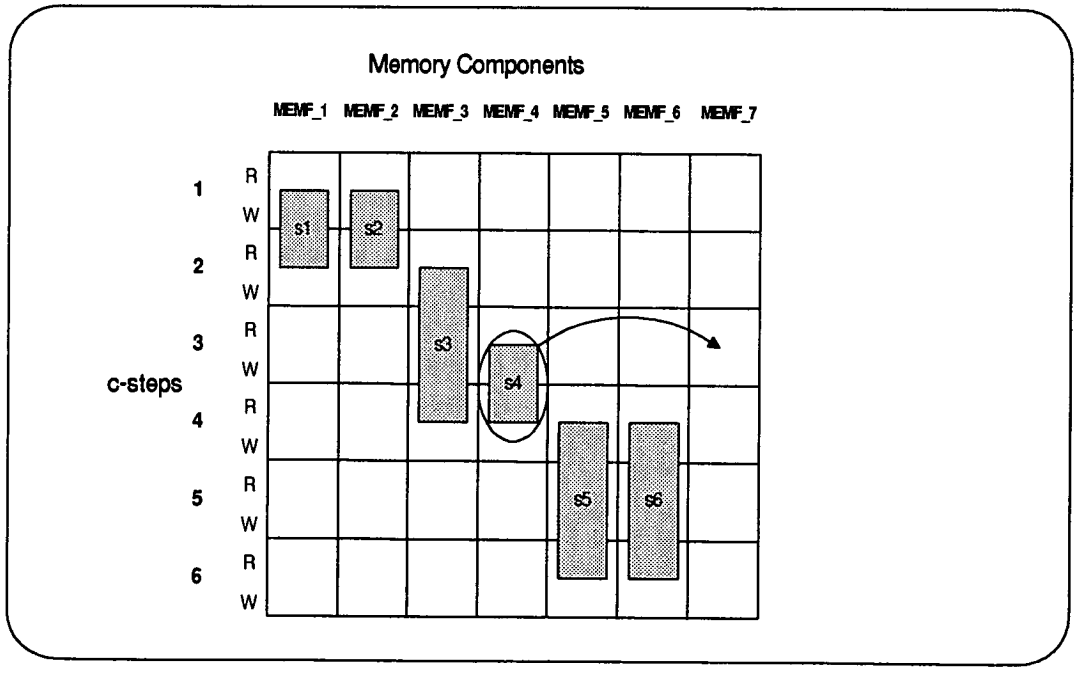


Figure 5.12 Create memory component.

Port Swap

A valid (i.e. *commutative*) operation is selected at random from the input data flow graph. By indexing *R-t* space, the input ports to the processor currently allocated to execute the operation can be determined. Once the processor input ports and the signal source ports have been located in *P-c* space, the input port connections are transposed. This is illustrated in figure 5.13.

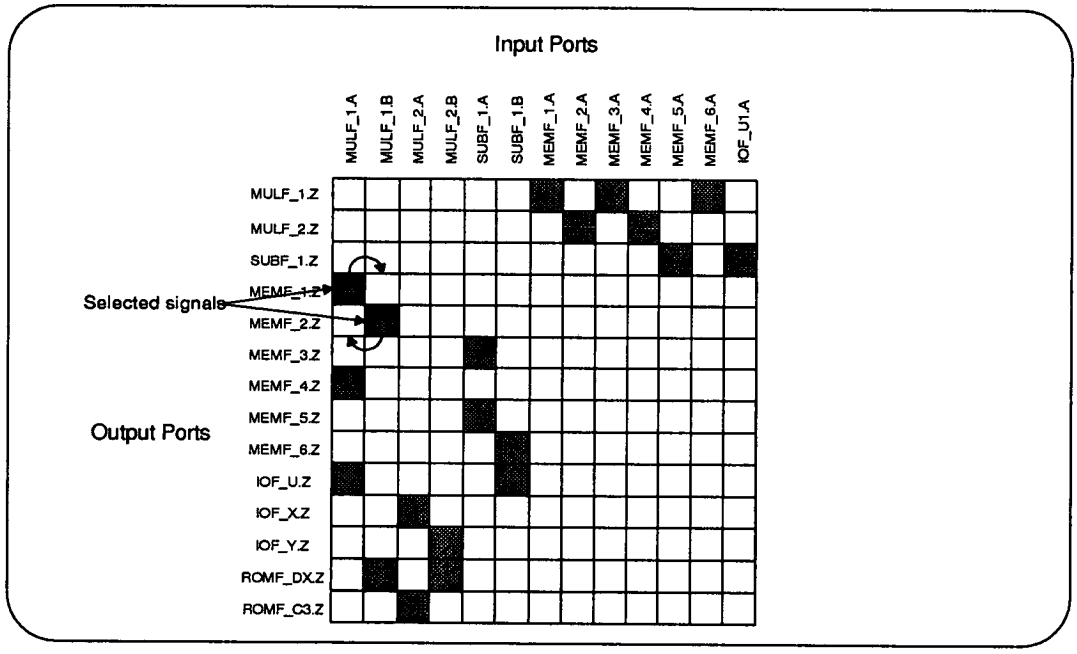


Figure 5.13 Port swap operation

Net Merge

This optimisation move replaces implicit wired-OR connection to an input port with a multiplexer instance. The optimisation moves supported are illustrated in figure 5.14.

A column-wise search through *P-c* space is carried out. Input ports with references to multiple output ports (implied wired-OR) are grouped in a subset (Input ports with a single output port connection are implemented using the default net type - `WIRE`.). An input port is selected at random from the subset. A pair of output ports are selected at random from the output port list. If both ports point to the same multiplexer instance,

then the move is invalidated. In the case where both ports point to separate wire instances then a new multiplexer instance is added to the net list and the pointers to the wire instances are replaced with pointers to the multiplexer instance (figure 5.14(a)).

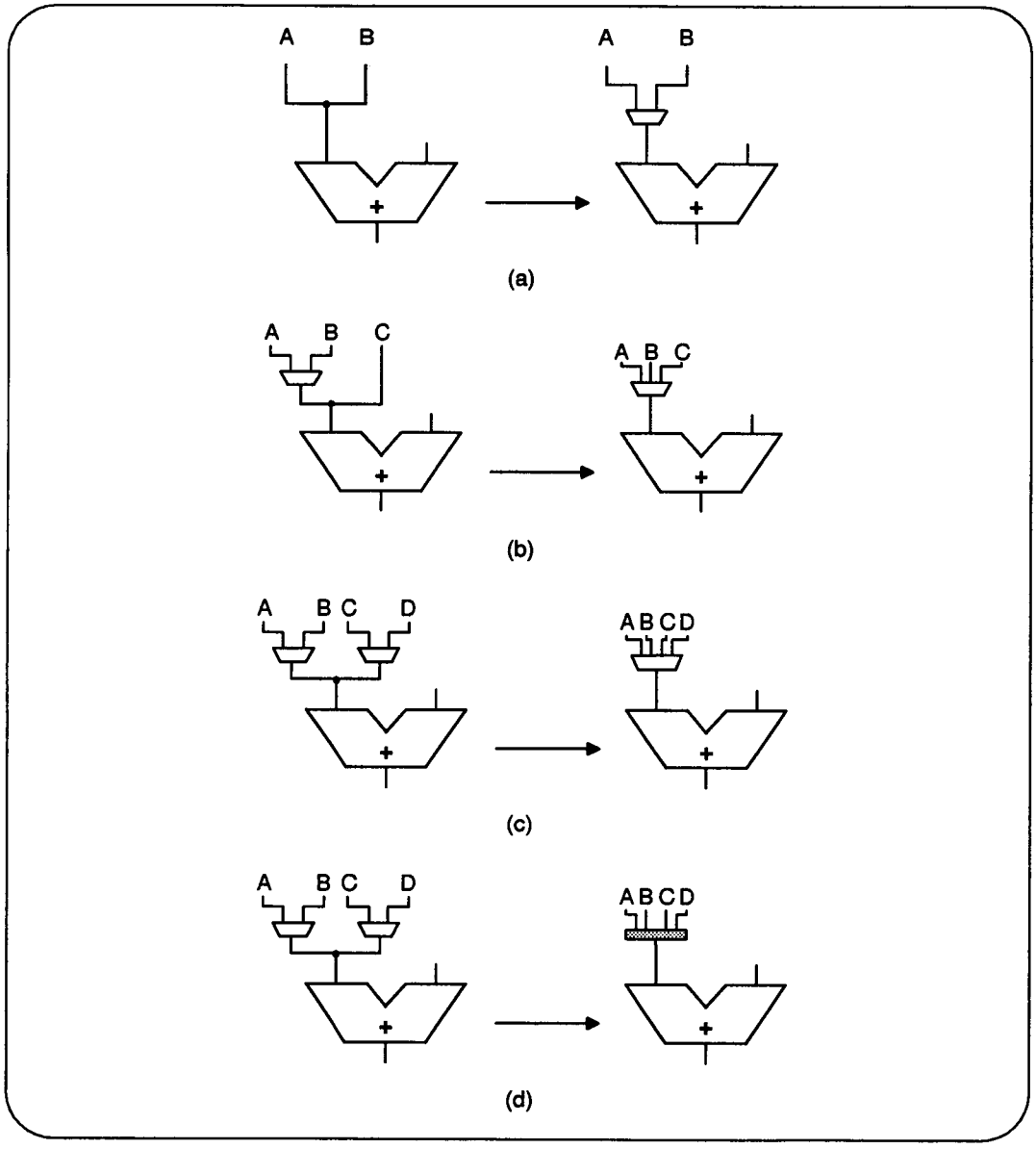


Figure 5.14 Net merge (a) wire to multiplexer, (b) wire and multiplexer to multiplexer, (c) multiplexer and multiplexer to multiplexer and (d) multiplexer and multiplexer to bus.

If either port points to a multiplexer instance, with the other pointing to a wire instance, then the wire instance is replaced with a pointer to the multiplexer (figure

5.14(b)). The cardinality of the multiplexer is also updated. In the case where both ports point to different multiplexer instances, then the two are merged into a single instance whose cardinality matches the sum of the originals (figure 5.14(c)) If the cardinality of the new multiplexer is greater than a pre-defined *bus threshold* value, then the multiplexer instance may be replaced with a bus instance (figure 5.14(d)).

Register Merge

This optimisation move merges two register (or register *file*) instances with a wired-OR connection to an input port into a single register file instance with a single wire connection to the input port. This move is invalidated if the read and write times of the signals intended for the register file are not mutually exclusive. The optimisation move is illustrated in figure 5.15.

A valid input port is selected from *P-c* space. Here, *valid* refers to input ports with more than one memory component on its output port connection list. Two memory components are selected at random. Both have their type instance set to `FILE`, and their cardinality index incremented accordingly.

5.4 Solution Quality Assessment

The costing function quantifies the quality of the newly generated state based upon three major criteria:

(i) Maximum operation execution time, extracted from the current schedule. This measures the quality of the temporal optimisation, i.e. the total execution time of the behavioural description.

(ii) A quantitative assessment of the structure of the generated datapath.

(iii) An estimation of the control overhead incurred by the schedule and datapath configuration.

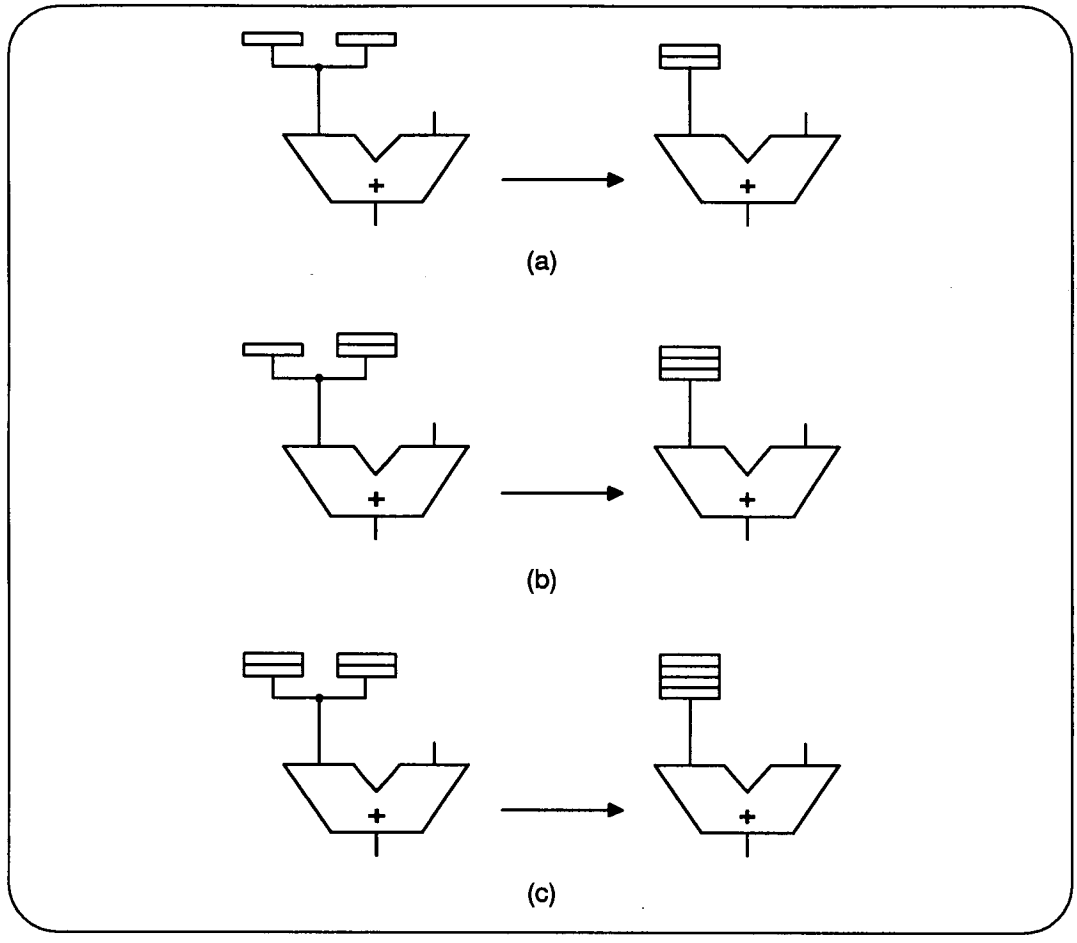


Figure 5.15 Valid register merges.

The maximum execution time is readily available as a function of the current schedule. A quantitative assessment of the structure is normally generated by attaching a value to each *type* of component present in the datapath and summing over *all* components in the datapath. Devedas and Newton [Dev89] refine this method by recognising that multiple component instances, such as register files (of cardinality n), will occupy less silicon area than a single component instance n times. A linear 'sliding scale' of cost is assigned to multiple component instances.

Many costing methods, however, overlook the routing overhead associated with a given datapath configuration. While all costing methods account for the nets (e.g.

multiplexers) present in the datapath, very few consider the *interconnect density* (i.e. the number of point-to-point connections in the datapath). Behavioural synthesis tools generally produce an optimised datapath macroarchitecture which is then targeted to a particular layout style by logic synthesis tools. For the standard cell and gate array technologies, the routing overhead can be up to 60% of the total silicon area [Sechen86]. Consequently, a measure of point-to-point connectivity is an essential component of the assessment of the datapath structure.

Within the SAVAGE system, the maximum execution time can be extracted by traversing *R-t* space searching for the maximum value of **operation.execution_time**. A similar search method is used to determine the numbers and types of datapath component. Finally, a measure of the interconnect density is obtained by summing the number of connection flags set in *P-c* space (figures 5.3 and 5.6).

The control overhead for a given datapath configuration is based on the following observations:

(i) For a 2^n input multiplexer, n control signals are required to select the appropriate input signal.

(ii) If a multi-function processor is capable of executing 2^m different operation types, then m control signals are required for function selection.

(iii) Where a register file of cardinality 2^k is instanced, k control signals are required to select the appropriate register output, in addition to the equivalent of a 2^k input demultiplexer.

(iv) For a schedule comprising of 2^t control steps, a state machine of t states is required for sequencing purposes.

To provide meaningful comparison between SAVAGE and other reported synthesis systems, however, only the figure obtained from (iii) is computed and used within the SAVAGE system.

5.4.4 A Datapath Costing Model

The SAVAGE datapath cost function is a summation of the quality criteria outlined above:

$$\begin{aligned} \text{COST (STATE)} = {} & \text{MAX\_EXECUTION} + \Sigma\ \text{COST(DATAPATH\_COMPONENTS)} \\ & + \text{INTERCONNECT\_DENSITY(PC)} \\ & + \Sigma\ \text{CONTROL\_OVERHEADS (DATAPATH\_COMPONENTS)} \end{aligned} \qquad [5.3]$$

A table lookup is used to obtain the datapath component costs. Table 5.1 indicates the *relative* costs of the various datapath components. These costs represent the number of fundamental building blocks required to implement the datapath component. For a compiled logic technology, this may be thought of as *cell units used*, while for a gate array technology it may be thought of as *gate equivalents*. The values quoted here were obtained from the LSI 1.0μ Cell-Based Products Databook [LSI91]. The figures quoted are for cell units used for each compiled datapath component.

The ALU function cost recognises the ability to share logic within a multi-function ALU. The multiplexer cost reflects the increase in multiplexer complexity as the number of multiplexer inputs increases. The bus cost is interpreted as the cost of the drive circuitry required for bus access. This cost increases linearly with the number of bus inputs.

Datapath Component	Cost
Adder	100
Subtractor	108
Multiplier	160
Comparator	140
ALU	MAX(function cost) + 30%
Register	70
n-Register File	70 *n* - 10%
n-ROM	40 *n*
n-Multiplexer	20.(10 log(*n*))
n-Bus	27*n*

Table 5.1 Datapath component costing table.

5.4.5 A Novel Cost Multiplier System

The SAVAGE datapath cost function provides a *balanced* assessment of the solution quality after a state generation. A cost multiplier system is implemented to allow the design engineer to penalise the generation of undesirable states. A cost multiplier is introduced for each quality criterion detailed above. Applying these cost multipliers to equation 5.3 gives:

$$\begin{aligned} \text{COST (STATE)} = {} & \text{C1.MAX\_EXECUTION} + \Sigma\, C_k.\text{COST(DATAPATH COMPONENT } k) \\ & + \text{C2.INTERCONNECT\_DENSITY(PC)} \\ & + \Sigma\, C_k.\text{CONTROL\_OVERHEADS (DATAPATH COMPONENT } k) \end{aligned} \qquad [5.4]$$

where C_k is the cost multiplier associated with a datapath component of type k. This novel system allows the design engineer to influence the overall architectural style of the solution datapath without directly affecting the state generation mechanism. The cost multipliers are input to the SAVAGE system, and may be changed over a sequence of simulated annealing runs.

5.5 The SAVAGE Toolset

SAVAGE is a modular software system. This software development technique offers a number of distinct advantages.

Functional partitioning: The datapath optimisation routines are maintained separately from those routines which control the search through the datapath solution space.

Software Interface: Within the SAVAGE program, interfaces between software procedures are carefully defined. This can be effectively policed by using a strongly typed programming language, such as ADA, as the implementation medium. Algorithmic tasks can easily be subdivided into manageable programming blocks, so that a significant software procedure (such as the simulated annealing core) can be developed and debugged quickly. Controlling the degree of interaction between code modules and the data structures enables changes in the solution datapath to be accurately tracked, and the correctness of the solution to be monitored throughout the optimisation procedure.

Code maintenance: By partitioning the datapath optimisation functions and search control functions into well understood code fragments each of manageable complexity, and by rigorously adhering to the interface standards, code maintenance and upgrading is more straightforward.

Ease of library expansion: Expansion of the optimisation move sets is also made straightforward by the adoption of a modular scheme.

The major components of the SAVAGE system are illustrated in figure 5.16, and are described below.

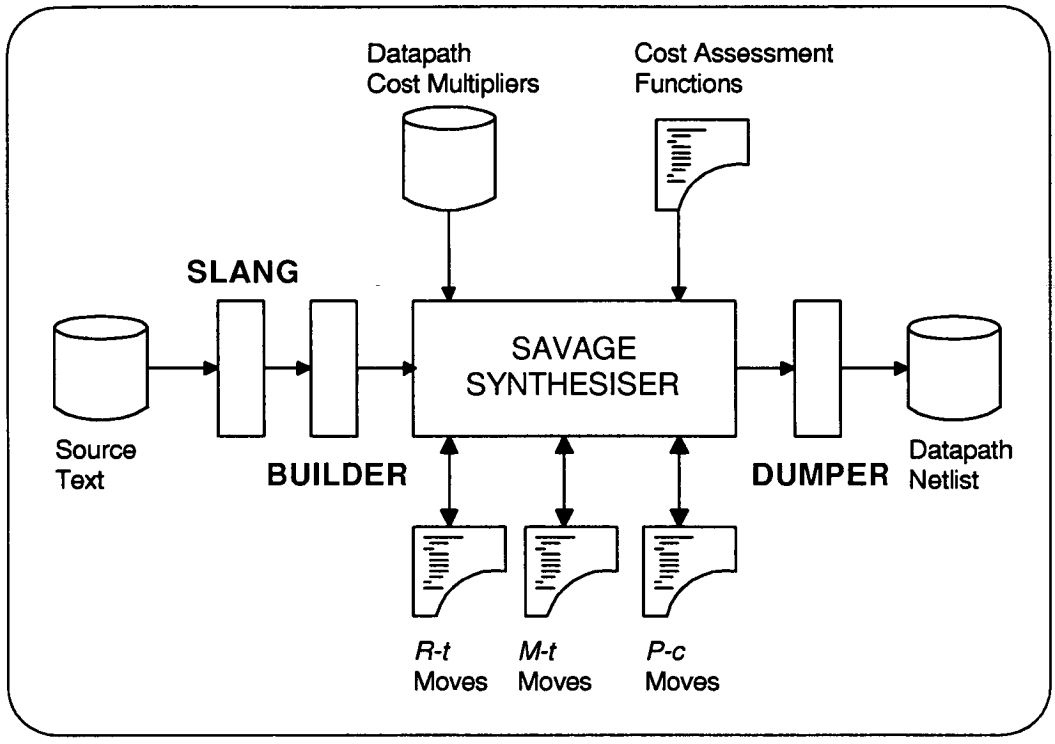


Figure 5.16 SAVAGE structure.

5.5.1 SAVAGE Synthesis Flow

The source text is processed by the SLANG (SARI LANGuage) compiler. SLANG is a subset of ADA, and the compiler produces a data flow representation similar to that described in section 2.3. The SLANG compiler was developed as part of the SAGE program [Grant90a], and is fully described in [Sey89].

After compilation, the source text has been transformed into a data flow representation. This graph is passed to the BUILDER module, which produces the initial data structure (*R-t*, *M-t* and *P-c* space).

The SYNTHESISER module comprises the simulated annealing procedure and its support functions. The state generation mechanism makes external calls to the optimisation move sets that are required. The state generation move sets and the associated costing functions are not integrated into the SYNTHESISER module. The

reason for this is primarily experimental; code maintenance and upgrades to the state generation moves can take place independently of the optimisation core.

Once optimisation is complete, the DUMPER module traverses the SAVAGE data structure extracting all datapath and connectivity information. A datapath netlist is created which adheres to the syntax specified in section 2.4.

5.5.2 BUILDER

The function of the BUILDER module is to take a compiled data flow graph and produce an initial solution space. This is a three stage operation.

(i) **Seed *R-t* space:** BUILDER establishes data flow precedence by building an ASAP schedule. Starting from the operations with the latest scheduled execution time, operations are generated randomly generated execution times bounded by the production and consumption times of the operation input and output data. BUILDER then creates a set of processors capable of executing the data flow operations. A random number of each type of processor is generated, bounded by the number of operations of the type executing concurrently and the maximum number of operations of that type present within the data flow graph. Operations are then randomly allocated among the processor set.

(ii) **Build *M-t* space**: The BUILDER creates a set of memory components (of type `REGISTER`). This size of the set is determined by the number of signals present in the data flow graph. Signals are then serially allocated to the registers.

(iii) **Complete *P-c* space:** From the initial *R-t* and *M-t* spaces, the Port-

connection space can be completed. All net instances required default to type `WIRE`.

5.5.3 Datapath Verification and Validation

The VALIDATE procedure propagates datapath changes following a state generation. VALIDATE propagates changes in *M-t* and *P-c* space if the perturbation occurs in *R-t* space, and propagates changes in *P-c* space if the perturbation occurs in *M-t* space.

If mutually exclusive signals allocated to the same memory component become overlapped as a result of a scheduling operation, then VALIDATE reallocates one of the signals to a free memory location and updates *P-c* space. Where a new port connection is required, VALIDATE sets the *P-c* flag appropriately and instances a WIRE net type. Subsequent state generations can optimise the new net instance. Similarly, where a port connection is severed, VALIDATE removes the pointer to the net instance, and where required, updates the cardinality of that instance.

The integrity of the datapath is policed by the WIRED procedure. The only illegal datapath state that can be generated is where wired-OR connections occur on a processor input port. This can be as a result of the VALIDATE or BUILDER procedures. The WIRED function checks every input port for such datapath violations as part of the cost assessment procedure after every state generation.

5.6 A Worked Example - Differential Equation Solver

The SAVAGE tools were exercised on the differential equation example first cited in Paulin [Paulin86]. An iterative algorithm is used to solve the second order differential equation below:

$$y'' + 3xy' + 3y = 0 \qquad [5.5]$$

The source text which is input to the SLANG compiler is given in figure 5.17.

```
procedure DIFF_EQ (X, U, Y : in out INTEGER; A : in INTEGER) is

  X1,Y1,U1 : INTEGER;
  DX,C3 : CONSTANT;

begin

  while (X < A) loop
      X1 := X + DX;
      U1 := U - (3*X*U*DX) - (3*Y*DX);
      Y1 := Y + (U*DX);
      X := X1; U := U1; Y := Y1;
  end loop;

end DIFF_EQ;
```

Figure 5.17 Differential equation procedure.

After compilation, the DIFF_EQ procedure is transformed into the data flow graph shown in figure 5.18.

5.6.1 A Maximum Speed Solution

In order to generate a maximum speed solution, the cost multiplier associated with the maximum data flow execution time is significantly increased. This discourages state generations with a long schedule. Similarly, as datapath area is of secondary concern in a high speed solution, the processor multipliers are relatively small. Further, a high speed solution implies fewer overall control steps, so the cost multipliers associated with those datapath components incurring control overhead, and multiplier associated with the control overhead itself (outlined in section 5.4.3) remain relatively small. ROMs are essential to the datapath function, so no cost multiplier is specified. All cost multipliers for a maximum speed solution are specified in table 5.2.

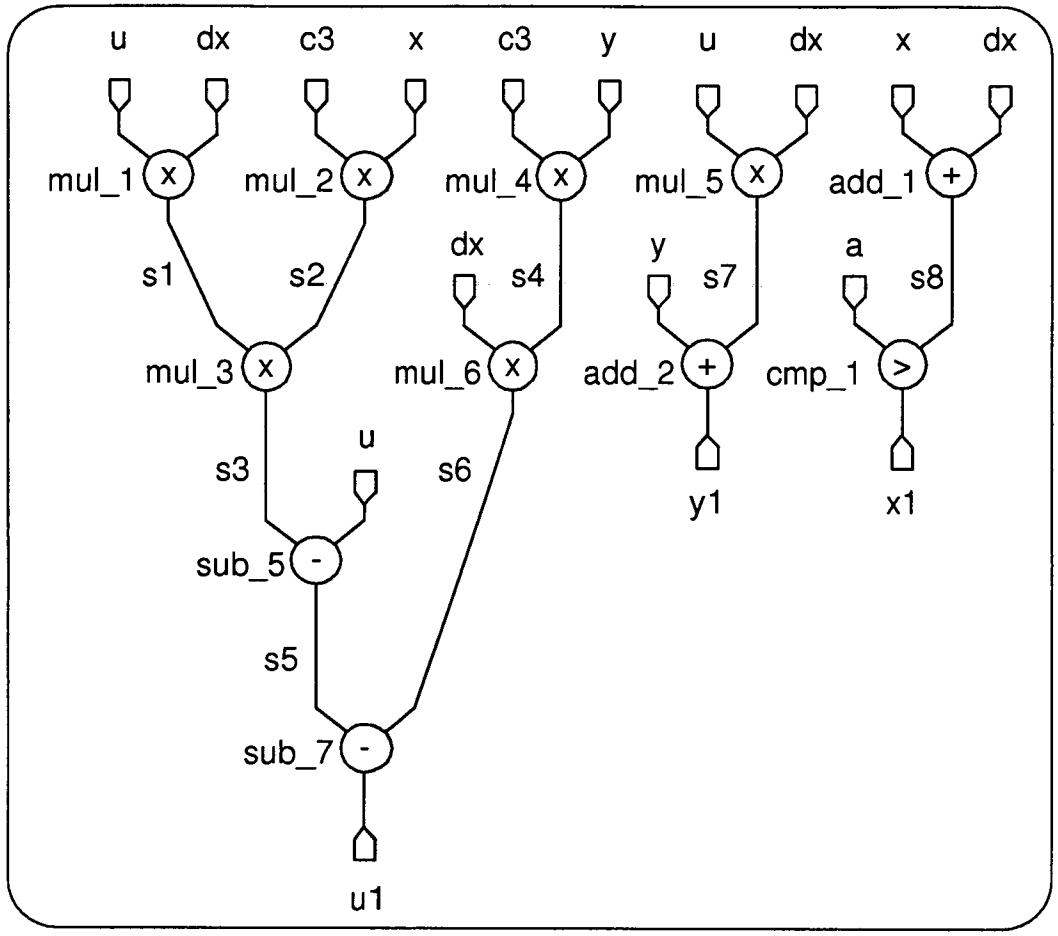


Figure 5.18 Compiled data flow graph

Adder	10	Register	30
Multiplier	10	Register File	30
Comparator	10	ROM	1
ALU	10	Maximum execution time	100
Multiplexer	30	Control Overhead	25
Tri-Buffer	45	Interconnect Density	25

Table 5.2 Cost multipliers for maximum speed solution.

Following the SAVAGE optimisation procedure, the datapath shown in figure 5.19 was generated. The datapath statistics are compared with other solutions reported in the

literature (table 5.3). No datapath schematics are reported for this maximum speed solution, so a full comparative analysis cannot be performed. It should be noted, however, that the SAVAGE system performs well by comparison, requiring the minimum number of registers and multiplexer inputs for a solution with a four control step schedule.

System	C-Steps	Processors	Registers	Mux Inputs
SPLICER [Pangrle88]	4	2x,+,-,>	6	11
CLIQUE [Fin92]	4	2x,+,-,>	5	11
ASSIGN [Fin92]	4	2x,+,-,>	5	11
HAL [Paulin89b]	4	2x,+,-,>	5	10
SAVAGE	4	2x,+,-,>	5	10

Table 5.3 Datapath statistics for maximum speed solution.

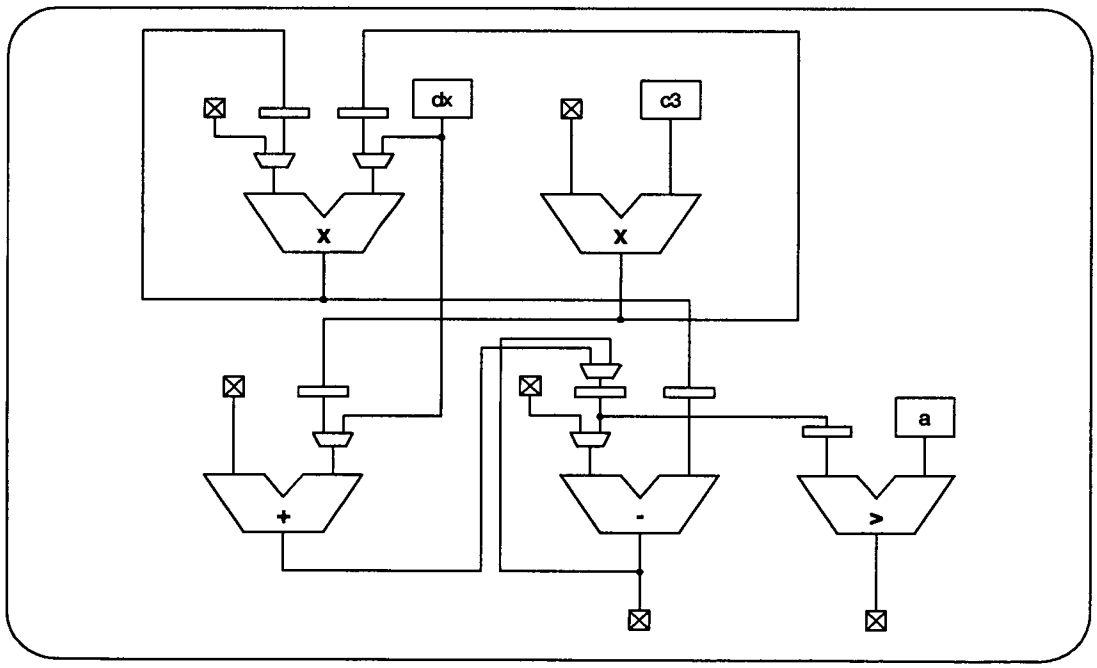


Figure 5.19 Maximum speed differential equation datapath.

5.6.2 A Minimum Area Solution

In generating a minimum area solution the cost multipliers associated with individual processors are large, while the multiplier associated with ALU structures is small, reflecting the desirable nature of states generated which map different operation types onto a single functional unit. A minimum area solution will execute in a greater number of control steps than a maximum speed solution, and so the control overheads incurred will be greater. This is reflected by the increase in the cost multipliers associated with controlled components, and the inherent control overhead itself. For the same reason, the number of point-to-point connections should be minimised; the cost multiplier associated with interconnect density is higher than that for a maximum speed solution. The full set of cost multipliers for a minimum area solution are presented in table 5.4.

Adder	50	Register	30
Multiplier	50	Register File	20
Comparator	50	ROM	1
ALU	10	Maximum execution time	5
Multiplexer	30	Control Overhead	75
Tri-Buffer	45	Interconnect Density	75

Table 5.4 Cost multipliers for minimum area solution.

The datapath generated using these multipliers is illustrated in figure 5.20(a). Comparative results for this minimum area solution are given in table 5.5. The SAVAGE solution compares favourably with those from the published literature, but the additional factors in the cost assessment function, namely the control overhead and the interconnect density figures result in a solution datapath more amenable to the prevalent compiled logic implementation technologies.

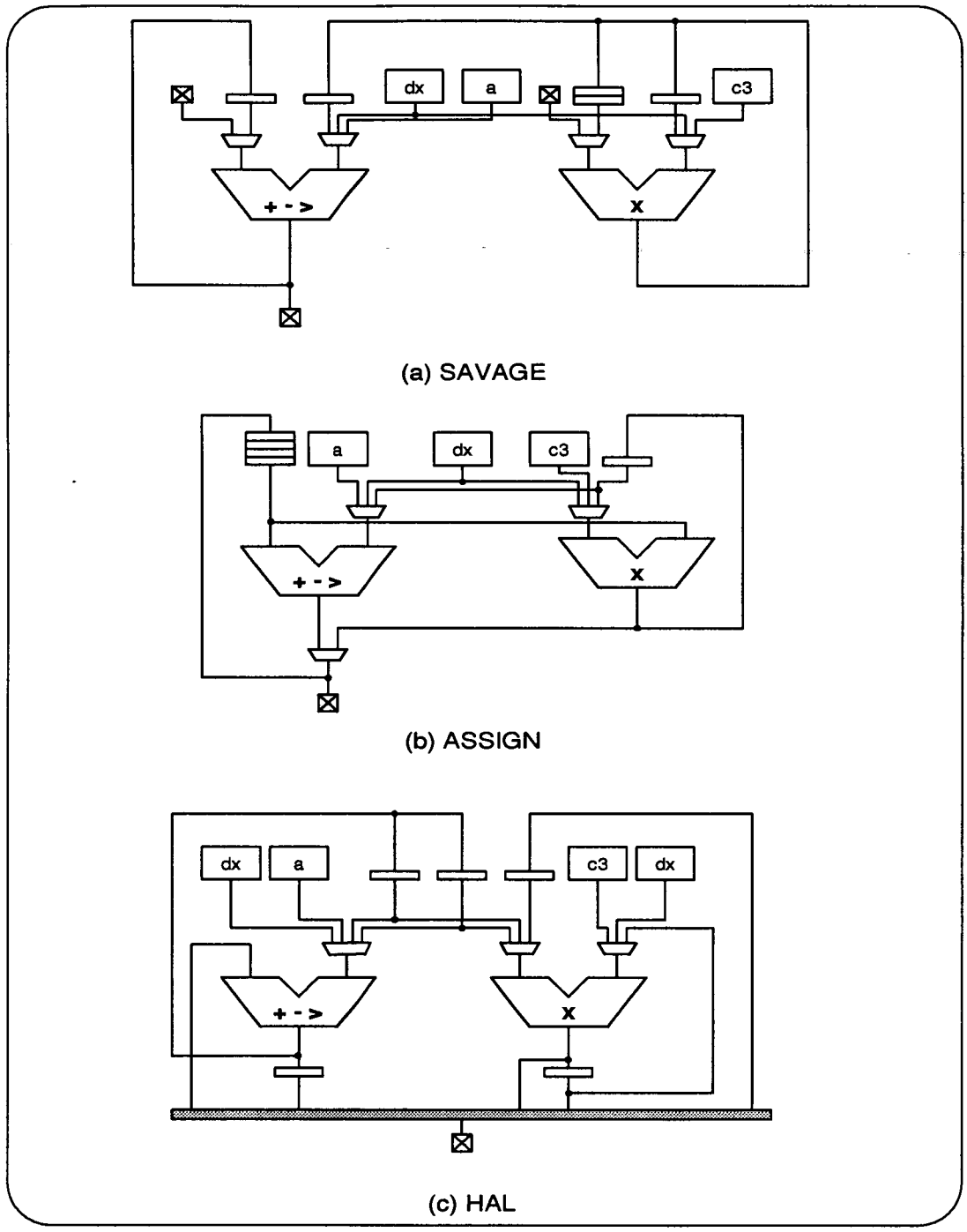


Figure 5.20 Minimum area datapaths.

System	C-Steps	Processors	Registers
ASSIGN [Fin92]	8	x,[+,-,>]	5
HAL [Paulin89b]	8	x,[+,-,>]	5
SAVAGE	8	x,[+,-,>]	5
System	**Mux Inputs**	**Control Wires**	**Point to Point Connections**
ASSIGN [Fin92]	8 (+4)[a]	9	13
HAL [Paulin89b]	11	13	22
SAVAGE	10 (+2)	8	19

a. Extra multiplexer inputs indicated as overhead for register file demultiplexing.

Table 5.5 Datapath statistics for minimum area solution.

5.7 Summary

This chapter has presented a set of synthesis tools based on the simulated annealing algorithm. A data structure was developed which enabled the optimisation routines to operate effectively in the scheduling and allocation, memory optimisation and interconnect minimisation domains. A set of modular optimisation moves was developed to describe all potential state generations from a given starting state. Further, the costing model described in Devedas [Dev89] was extended to provide a link between the datapath macroarchitecture and the physical implementation. The SAVAGE system was applied to a small-scale example, and was able to demonstrate the speed vs. area trade-off common to all synthesis problems. The generated datapaths compare favourably to those in the published literature.

The next chapter exercises the SAVAGE system on two large-scale benchmarks, and introduces mechanisms for resource constrained synthesis.

6 SAVAGE Case Studies

This chapter presents two substantial case studies. The first is a comparative study between SAVAGE and the SARI tool, SAGE (SARI Architecture Generator [Grant90b]). SAGE is primarily an interactive tool, and relies heavily on intervention from the design engineer. The test vehicle for this study is the 1-dimensional 8 point Fast Discrete Cosine Transform (FDCT). This is the most complex example currently in use as a synthesis benchmark. Aside from the large number of operations present in the data flow graph, the mixture of commutative and non-commutative operations provides added complexity.

The second case study allows the performance of the SAVAGE system to be compared with a wider selection of behavioural synthesis systems. The test vehicle here is a 5th order Wave Digital Filter.

6.1 1-Dimensional 8 point Fast Discrete Cosine Transform

The large amount of information contained within a high definition digital image poses significant problems, both in terms of memory requirement and transmission latency in applications where real time, or near real time image transmission is required. As a result, many data compression techniques have been proposed [Chen77, Wintz72, Soame82]. The Discrete Cosine Transform (DCT) operates on a series of blocks decomposed from the original image. These blocks are ranked according to their a.c. energy (a.c. energy quantifies the amount of information within a particular block). A bit assignment according to the average point variance within the block then takes place. It is here that the data compression takes place; more bits are assigned to visually important regions (i.e. regions of the image containing most information) than those of lesser interest.

Interest in the DCT algorithm realised in silicon has been driven by applications such as video conferencing and computer-based multimedia. Further, video coding and compression standards, such as JPEG (*Joint Photographic Experts Group, DIS10918 Digital Compression and Coding of Continuous-tone Still Images*) use the DCT process.

Most implementations of the DCT make use of the separability property of the 2D transform. The 2D transform is composed of a 1D n-point row transform. Rows and columns are then transposed, and the operation is completed by a 1D n-point column transform. The value of n is typically 8 or 16. Table 6.1 summarises recent DCT implementations.

Device	Transform Size	Transistor Count	Technology	Clock Rate	Notes
LSI CW702 [LSI93]	8 x 8	N/A	0.8µm CMOS	30 MHz	JPEG Core
IMSA121 [SGS90]	8 x 8	not available	1.0µm CMOS	20 MHz	
STV3200 [SGS90]	4 x 4 4 x 8 8 x 4 8 x 8 8 x 16 16 x 8 16 x 16	114.3k	1.0µm CMOS	15 MHz	
STV3208 [SGS90]	8 x 8	not available	1.0µm CMOS	27 MHz	
[Yan89]	15 x 15[a]	70k	1.25µm CMOS	30 MHz	Systolic Array
[Matt89]	8 x 8	56k	1.6um CMOS	27 MHz	Multiprocessor Architecture
[Afgha86]	16 x 16	42k	3.0µm CMOS	25 MHz	

a. This device uses the Winograd Transform, which is equivalent to a 16 x 16 DCT

Table 6.1 VLSI implementations of the DCT.

Each of the VLSI implementations in the table above is a highly optimised DSP datapath, and in the case of the CW702, the implementation is of a complete JPEG core.

The DCT, $F(k)$ of a discrete function $f(j)$, $j=0,1, \ldots, N-1$ where N is the total number of data points is :

$$F(k) = \frac{2c(k)}{2} \sum_{j=0}^{N-1} f(j) \cos\left[\frac{(2j+1)k\pi}{2N}\right] \qquad [6.1]$$

where $k = 0, 1, \ldots, N-1$, and $c(k) = \frac{1}{\sqrt{2}}$ for $k = 0$, and $c(k) = 1$ for $k = 1, 2, \ldots, N-1$.

Previously, the DCT has been implemented using a double size Fast Fourier Transform (FFT) employing complex arithmetic and operating on $2N$ coefficients. The Fast Discrete Cosine Transform (FDCT) [Chen77] alleviates the implementation problems

associated with the DCT by using only real arithmetic and operating on N data points. This results in a factor of six reduction in the algorithm complexity. The FDCT is most readily expressed in terms of an extensible flow graph. The 1-dimensional 8 point Fast Discrete Cosine Transform is shown in figure 6.1.

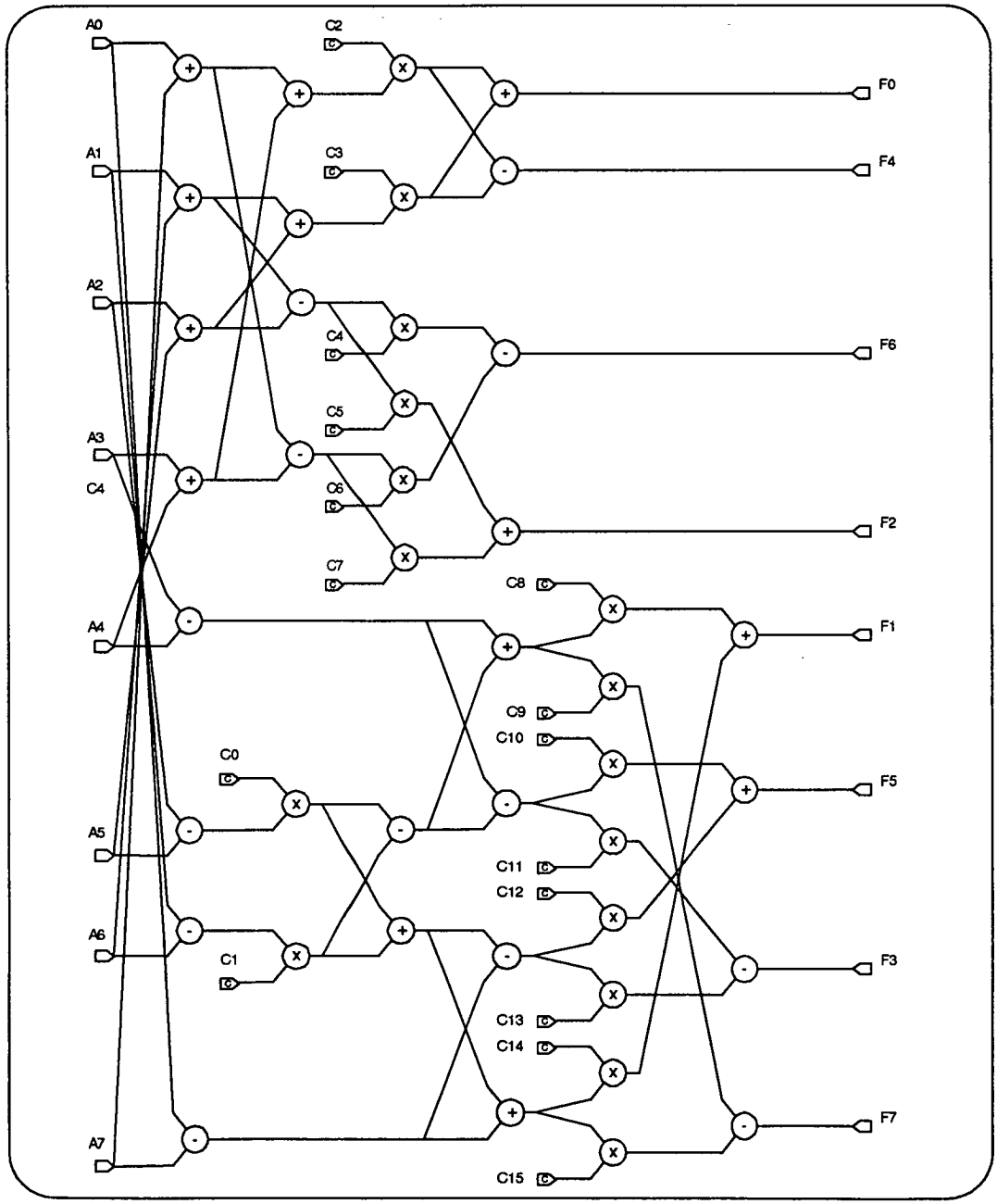


Figure 6.1 1-dimensional 8 point Fast Discrete Cosine Transform.

In order to generate a comparative study, the processor allocation available to the SAVAGE tools was constrained to be 2 adders, 2 multipliers and 2 subtractors. This provided direct comparison with a set of results generated as part of the SARI reporting procedure [SARI89]. The constraining mechanism used is the `pragma` statement. Pragmas are SAVAGE directives included in the source text which directly affect the synthesis tools. In this case, they provide a boundary condition for the BUILDER procedure and suppress the generation of new processors in the *R-t* optimisation move set. The statement:

```
pragma #MULF MAX 2
```

indicates the upper bound on the number of multiplier units that may be synthesised. The BUILDER will create an initial *R-t* space containing a maximum of two multipliers, and the CREATE_PROCESSOR procedure will be suppressed when there are two or more multipliers present in the current datapath. The SLANG code for this example is given in figure 6.2.

6.1.1 A Resource Constrained Datapath

A resource constrained datapath compromises the trade-off between a high speed solution and control overhead and interconnect density as demonstrated in the differential equation datapaths presented in the previous chapter. Correspondingly, the cost multipliers associated with these factors are relatively large compared to the cost multipliers associated with the datapath components themselves in order to minimise the impact of the constraint. The cost multipliers used to generate a constrained solution are presented in table 6.2.

```
package body FDCT_EXAMPLE is

pragma #MULF MAX 2
pragma #ADDF MAX 2
pragma #SUBF MAX 2

procedure FDCT_1D (   A0, A1, A2, A3, A4, A5, A6, A7 : in FLOAT
                      F0, F1, F2, F3, F4, F5, F6, F7 : out FLOAT) is

  COS_PI_4 : CONSTANT;
  COS_PI_8 : CONSTANT;
  SIN_PI_8 : CONSTANT;
  COS_PI_16 : CONSTANT;
  SIN_PI_16 : CONSTANT;
  COS_3_PI_16 : CONSTANT;
  SIN_3_PI_16 : CONSTANT;
  COS_5_PI_16 : CONSTANT;
  SIN_5_PI_16 : CONSTANT;
  COS_7_PI_16 : CONSTANT;
  COS_7_PI_16 : CONSTANT;
  SIN_7_PI_16 : CONSTANT;

  B0, B1, B2, B3, B4, B5, B6, B7 : FLOAT;
  C0, C1, C2, C3, C4, C5, C6, C7 : FLOAT;
  D0, D1, D2, D3, D4, D5, D6, D7 : FLOAT;

  COS_PI_4_TIMES_B5 : FLOAT; COS_PI_4_TIMES_B6 : FLOAT;
  COS_PI_4_TIMES_D0 : FLOAT; COS_PI_4_TIMES_D1 : FLOAT;

begin
  -- first pass
  B0 := A7 + A0; B1 := A6 + A1; B2 := A5 + A2; B3 := A4 + A3;
  B4 := A3 - A4; B5 := A2 - A5; B6 := A1 - A6; B7 := A0 - A7;
  -- put the expressions COS_PI_4*B5 and COS_PI_4*B6 into
  -- intermediate variables so as to avoid evaluating them twice
  COS_PI_4_TIMES_B5 := COS_PI_4*B5;
  COS_PI_4_TIMES_B6 := COS_PI_4*B6;
  -- second pass
  C0 := B3 + B0; C1 := B2+B1; C2 := B1 - B2; C3 := B0 - B3; C4 := B4;
  C5 := COS_PI_4_TIMES_B6 - COS_PI_4_TIMES_B5;
  C6 := COS_PI_4_TIMES_B6 + COS_PI_4_TIMES_B5;
  C7 := B7;
  -- third pass
  D0 := C0; D1 := C1; D2 := C2; D3 := C3; D4 := C4 + C5;
  D5 := C4 -C5; D6 := C7 - C6; D7 := C7 + C6;
  -- put the expressions COS_PI_4*D0 and COS_PI_4*D1 into
  -- intermediate variables so as to avoid evaluating them twice
  COS_PI_4_TIMES_D0 := COS_PI_4*D0;
  COS_PI_4_TIMES_D1 := COS_PI_4*D1;
  -- fourth pass
  F0 := COS_PI_4_TIMES_D0 + COS_PI_4_TIMES_D1;
  F4 := COS_PI_4_TIMES_D0 - COS_PI_4_TIMES_D1;
  F2 := SIN_PI_8*D4 + COS_PI_8*D3;
  F6 := COS_3_PI_16*D3 - SIN_PI_8*D2;
  F1 := SIN_PI_16*D4 + COS_PI_16*D7;
  F5 := SIN_5_PI_16*D5 + COS_5_PI_16*D6;
  F3 := COS_3_PI_16*D6 - SIN_3_PI_16*D5;
  F7 := COS_7_PI_16*D7 - SIN_7_PI_16*D4;
end FDCT_1D;

end FDCT_EXAMPLE;
```

Figure 6.2 SLANG description of FDCT.

Adder	30	Register File	50
Multiplier	30	ROM	1
Multiplexer	50	Maximum execution time	75
Tri-Buffer	80	Control Overhead	75
Register	30	Interconnect Density	75

Table 6.2 Cost multipliers for processor constrained datapath.

The datapath resulting from the SAVAGE synthesis procedure is illustrated in figure 6.3(a). The SAGE generated datapath is shown in figure 6.3(b). The datapath statistics are shown below in table 6.3.

System	C-Steps	Processors	Registers
SAGE [SARI89]	12	2+,2x(P)[a],2-	40
SAVAGE	12	2+, 2x(P), 2-	28
System	**Mux Inputs**	**Control Wires**	**Point to Point Connections**
SAGE	10 (+36)[b]	46	79
SAVAGE	51 (+28)	28	77

a. Indicates a pipelined multiplier.
b. Register file decoding.

Table 6.3 Datapath statistics for resource constrained example.

The two systems compare favourably in this restricted test case. The SAGE bus-based architecture trades multiple register and bus driver instances against the lower overall control overhead. By comparison, SAVAGE optimises register usage to significantly reduce the signal storage requirement.

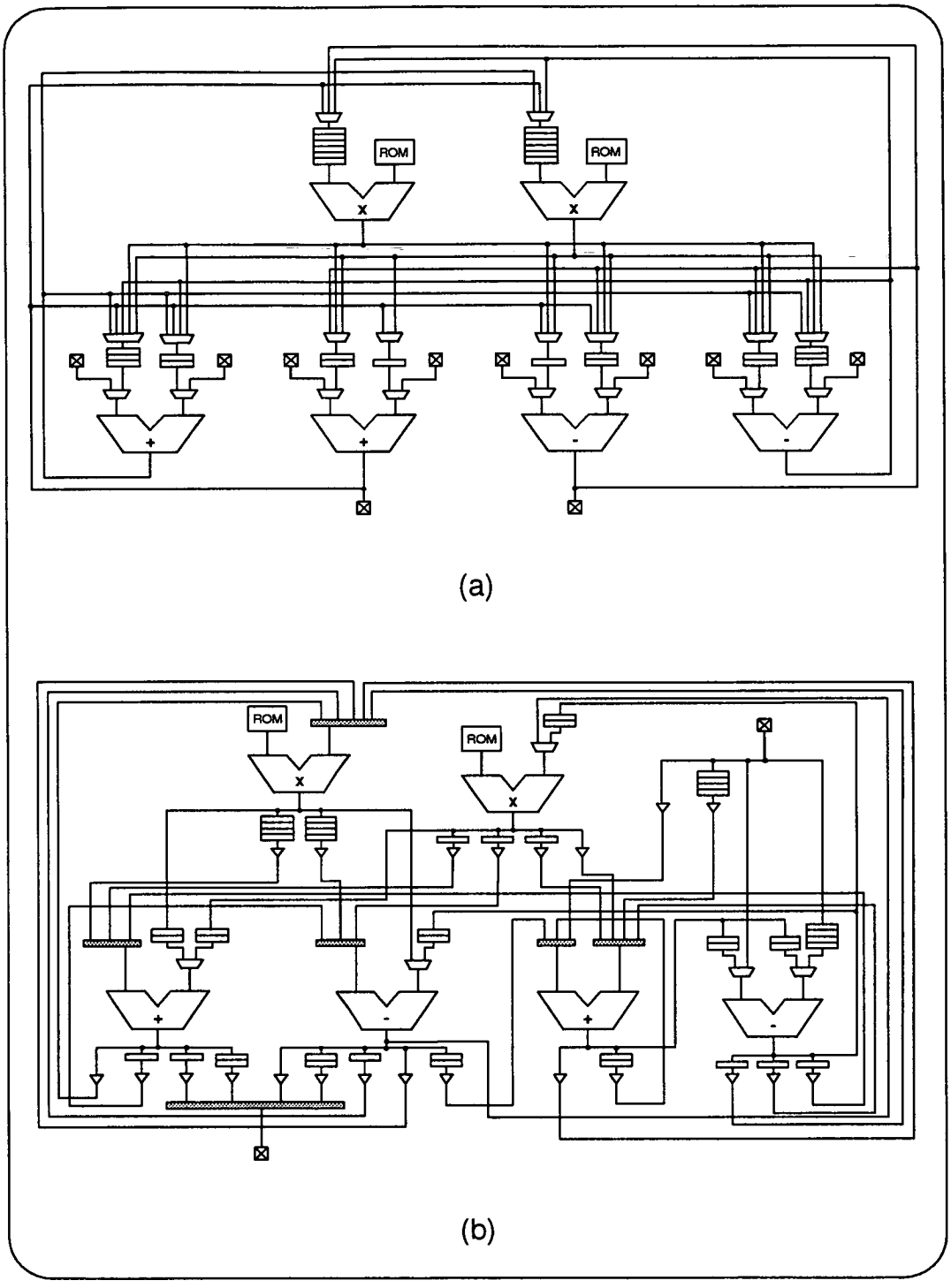


Figure 6.3 (a) SAVAGE-generated FDCT datapath (b) SAGE-generated FDCT datapath.

The major difference in the datapath architectures is the use of tri-state buses in the SAGE datapath compared with a multiplexer based approach in the SAVAGE solution.

Design methodology studies performed at the compiled logic cell library level [Sasena89], suggest that a multiplexer based approach is preferable on the basis of fewer gate equivalents required for implementation, shorter switching delays, lower overall power consumption and better circuit reliability. The studies also show that while the apparent difference between approaches can be small for designs with a regular structure, as the regularity of the design decreases, tri-state designs become more difficult to route at the cell placement stage of the standard cell design process. The studies are not exhaustive, but the results suggest that the relatively high cost multiplier associated with the bus driver components can be justified.

6.1.2 A High Speed Solution

This example removes the `pragma` constraints introduced previously. Instead, a higher speed datapath is sought. The cost multipliers for the datapath components remain unchanged (table 6.2), but solutions with long execution times are penalised more heavily. The cost multipliers for this example are presented in table 6.4.

Adder	30	Register File	50
Multiplier	30	ROM	1
Multiplexer	50	Maximum execution time	130
Tri-Buffer	80	Control Overhead	75
Register	30	Interconnect Density	75

Table 6.4 Cost multipliers for high speed datapath.

The datapath generated by the SAVAGE system is shown in figure 6.4. The datapath statistics are presented in table 6.5.

The results indicate a 33% speed improvement over the resource constrained solution, which should be set against the addition of a further multiplier. The datapath generated

reduces the number of registers required at the input to each multiplier as expected, and it should be observed that the relatively even distribution of register files at the inputs to each datapath component suggests that the solution overcomes any memory bottleneck which may be present in the solution presented in section 6.1.1. It should also be noted that this is achieved without significant increases in the control complexity and interconnect density values for the datapath.

System	C-Steps	Processors	Registers
SAVAGE	8	2+,3x(P)[a],2-	26
System	**Mux Inputs**	**Control Wires**	**Point to Point Connections**
SAVAGE	50 (+27)[b]	27	77

a. Indicates a pipelined multiplier.
b. Register file decoding.

Table 6.5 Datapath statistics for high speed solution.

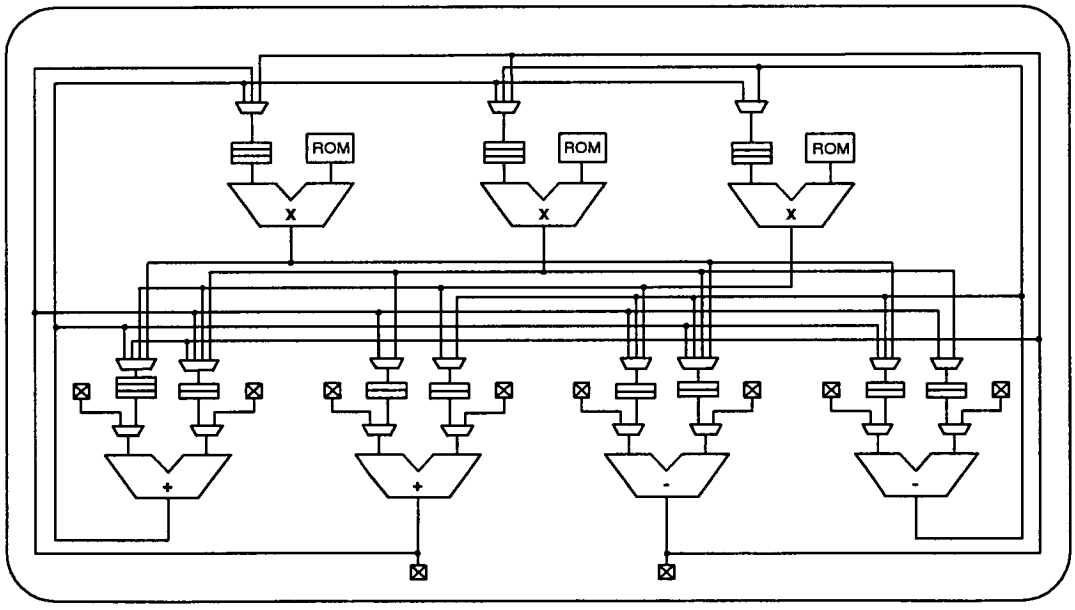


Figure 6.4 High speed FDCT datapath.

6.2 5th Order Wave Digital Filter

The 5th Order Wave Digital Filter remains an enduring synthesis example primarily because it is composed entirely of commutative operations. This allows greater flexibility in the memory and interconnect optimisation. Correspondingly, the literature is replete with datapath statistics for this design [Pangrle87, Paulin89b, Stok90, Fin92].

The first VLSI implementation of a wave digital filter was suggested in [Law77]. It was estimated that a 9th order unit element WDF could be fabricated using NMOS technology using one multiplexed two-port adaptor. With a 4 MHz clock, sampling frequencies of 13 kHz could be achieved.

The two-port adaptor provides a simple building block for the design of lattice and unit element structures. More complex three-port adaptors for ladder filter structures which use serial arithmetic are proposed in [Reek84, Petrie86].

High throughput filters for sonar and video signal processing applications have been implemented using carry-save arithmetic [Kleine88], and more recently bit-level systolic adaptors [Law90, Raj90]. Indeed, CAD research directed towards providing a high-level synthesis environment for bit-level systolic arrays is reported in [Kung85].

Three datapath variations are generated. The first two examples demonstrate the effect of processor latency on a resource constrained datapath. This effect is attributed to processor pipelining at the microarchitectural level (section 1.1). The final variation is a maximum speed solution, again using pipelined processor units. The SLANG description of the 5th Order Wave Digital Filter is given below.

```
procedure WDF (IP : in  FLOAT; OP : out FLOAT) is

  T2,T13,T26,T33,T38,T39,T18 : FLOAT;
  o1,o2,o3,o4,o5,o6,o7,o8,o9,o10,o11,o12,o13,o14,o15,o16,o17 : FLOAT;
  o18,o19,o20,o21,o22,o23,o24,o25,o26,o27: FLOAT;
  C0,C1,C2,C3,C4,C5,C6,C7,C8 : CONSTANT;

begin

  -- Initialisation
  T2 := 0.0; T13 := 0.0;
  T18 := 0.0; T26 := 0.0;
  T33 := 0.0; T38 := 0.0;
  T39 := 0.0;

  -- Filter loop
  while not (eof) loop
     o1 := IN + T2; o2 := T33 + T18; o3 := o1 + T13; o4 := o3 + T26;
     o5 := o4 + o2; o6 := o5 * C0; o7 := o5 * C1; o8 := o3 + o6;
     o9 := o2 + o7; o10 := o3 + o8; o11 := o2 + o9; o12 := o10 * C3;
     o13 := o5 + o8; o14 := o11 * C4; o15 := o9 + o13; o16 := o1 + o12;
     o17 := o14 + T18; o18 := o8 + o16; o19 := o9 + o17; o20 := o1 + o16;
     o21 := o19 + T38; o22 := o20 * C5; o23 := o18 + T33; o24 := o21 * C6;
     o25 := o17 + T18; o26 := IN + o22; o27 := o23 * C7; T28 := o24 + T38;
     OP := o25 * C8; T2 := o16 + o26; T39 := o19 + o28;
     T33 := o27 + T33; T18 := o17 + o29; T13 := o18 + o32;
  end loop;

end WDF;
```

Figure 6.5 SLANG description of 5th Order Wave Digital Filter.

The compiled flow graph is given in figure 6.6 (from [Dew85]).

6.2.1 Resource Constrained Datapaths

These test cases use the SAVAGE pragmas to constrain R-t space during synthesis as described previously in section 6.1. A further pragma is used to specify the latency of the multiply unit. The statement:

```
pragma MULF.ATT LATENCY 2
```

sets the **processor.latency** field in the processor attribute list to be 2 cycles.

This models a non-pipelined multiplier at the macroarchitecture level (i.e. the *behaviour* of the multiplier is described without any specification of the multiplier *implementation*). The BUILDER procedure and the *M-t* optimisation routines use the data in this field to modify the signal production times for all operations executing on that processor.

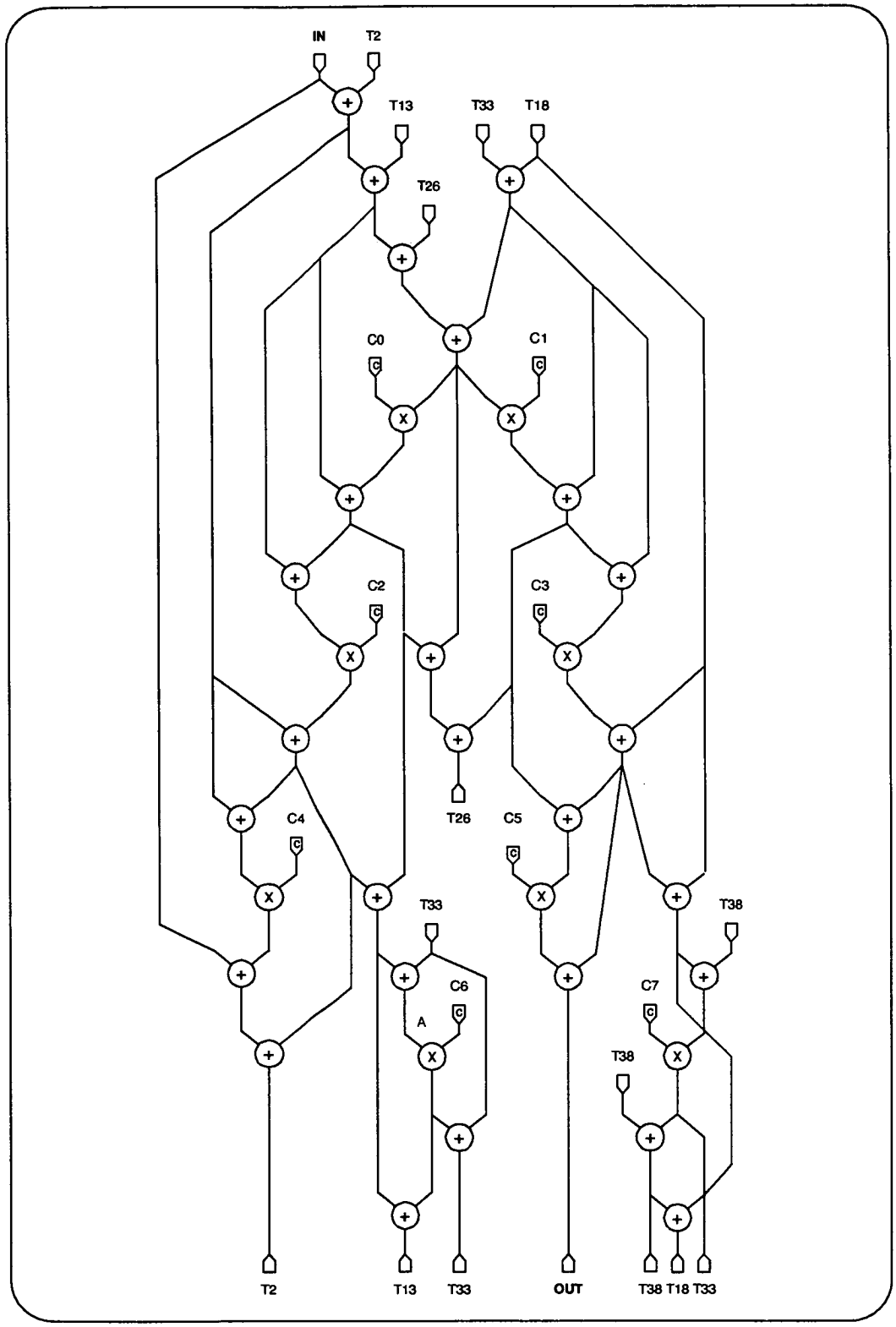


Figure 6.6 5th Order Wave Digital Filter data flow graph.

With a non-pipelined multiplier, the execution times of individual operations become a

critical factor in the overall solution quality. Correspondingly, the *maximum execution time* cost multiplier is set to a high value. Further, to generate comparative results, a pragma is used to constrain the datapath to a single multiplier unit. With a non-pipelined processor present in the datapath, overall signal lifetimes increase, and so a higher cost multiplier value is associated with individual registers, with a lower value assigned to register files reflecting the area savings gained through the use of these structures. Again, as an attempt to recognise the target implementation technology, the interconnect density value remains at a high setting. The full set of cost multipliers for this non-pipelined datapath are given in table 6.6. The generated datapath is shown in figure 6.7, with comparative datapath statistics in table 6.7.

Adder	25	Register File	35
Multiplier	25	ROM	1
Multiplexer	45	Maximum execution time	100
Tri-Buffer	55	Control Overhead	60
Register	50	Interconnect Density	80

Table 6.6 Cost multipliers for non-pipelined datapath generation.

No reported results include either generated schematics or detailed analyses of the synthesised datapaths. From the results presented, however, SAVAGE demonstrates a 30% best case improvement for the number of multiplexer inputs required within the solution datapath and a 31% best case improvement in the number of registers required to satisfy the signal storage criteria. For this non-pipelined solution, 18 control wires are necessary to implement the control function, and there are 25 point-to-point connections present within the datapath.

By resetting the latency pragma, the SAVAGE system can evaluate the alternative pipelined solution. The cost multipliers remain unchanged from the non-pipelined generation. The generated datapath is shown in figure 6.8.

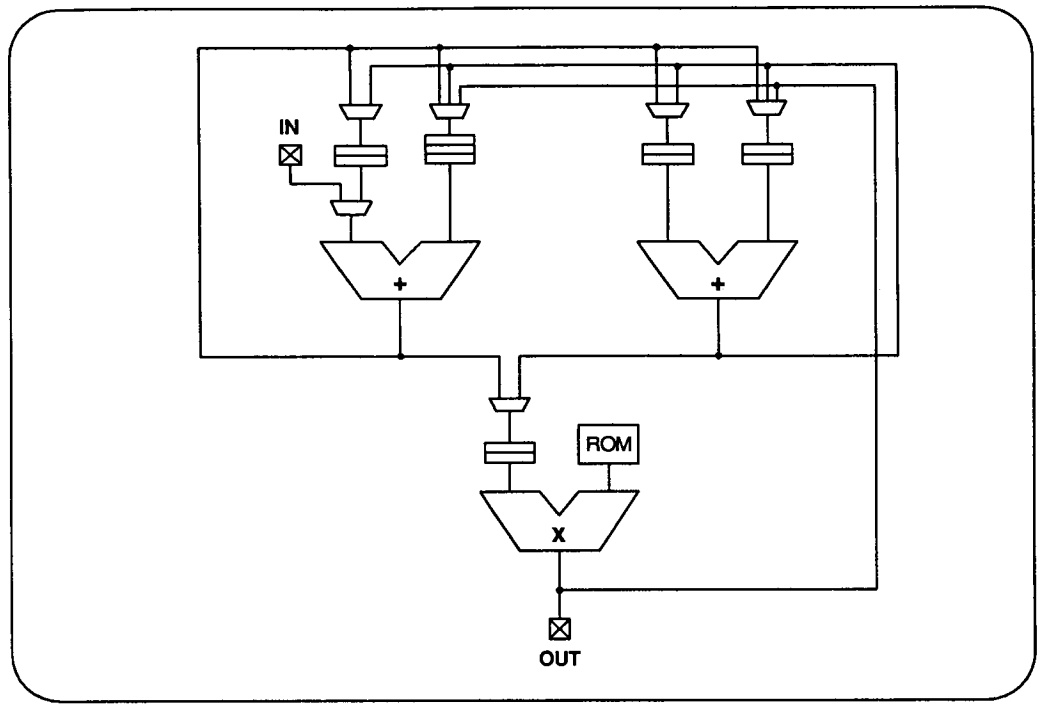


Figure 6.7 Non-pipelined resource constrained datapath.

System	C-Steps	Processors	Registers	Mux Inputs
SPLICER [Pangrle87]	21	2+, 1x	N/A	35
HAL [Paulin89b]	21	2+, 1x	12	30
ESC [Stok90]	21	2+, 1x	16	23
ASSIGN [Fin92]	21	2+, 1x	13	13 (+13)
SAVAGE	21	2+, 1x	11	14 (+11)

Table 6.7 Datapath statistics for non-pipelined datapath.

Comparative datapath results are available for this resource constrained schedule, and are presented in table 6.8. For completeness, other statistics quoting the overall schedule and the number of datapath components only are given in table 6.9. A comparative set of datapath schematics is included in figure 6.9.

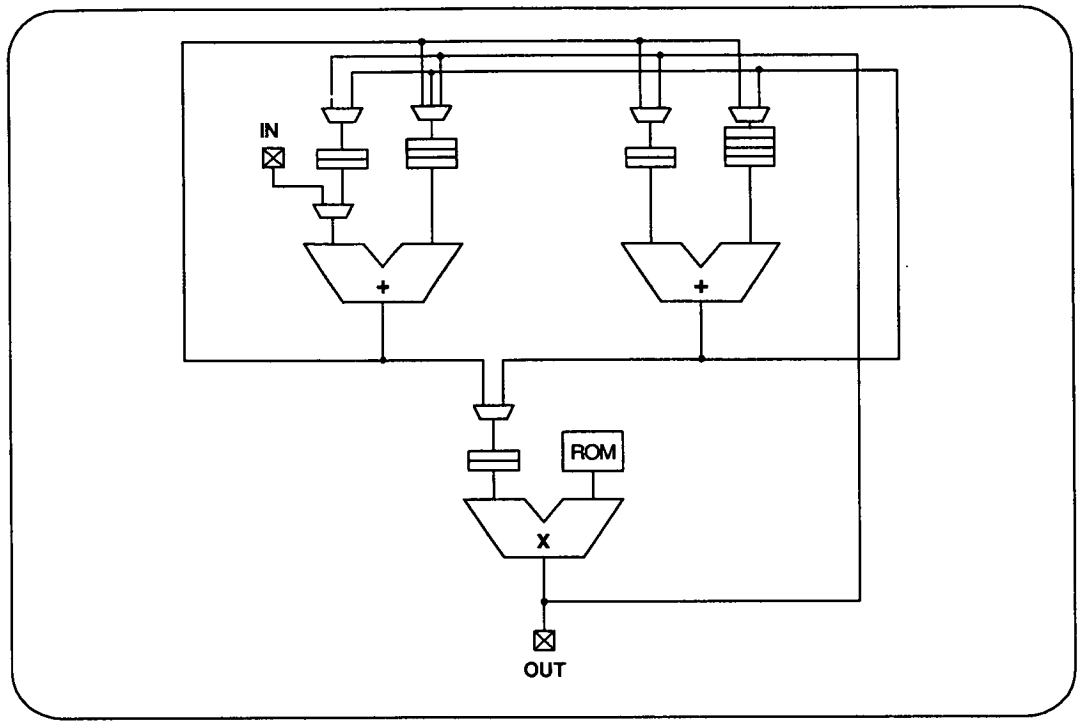


Figure 6.8 Pipelined resource constrained datapath.

System	C-Steps	Processors	Registers
ASSIGN [Fin92]	19	2+, 1x (P)	13
CLIQUE [Fin92]	19	2+, 1x (P)	14
HAL [Paulin89b]	19	2+, 1x (P)	12
SAVAGE	19	2+, 1x (P)	13
System	**Mux Inputs**	**Control Wires**	**Point to Point Connections**
ASSIGN [Fin92]	13 (+12)	20	25
CLIQUE [Fin92]	16 (+13)	22	27
HAL [Paulin89b]	-	26	39
SAVAGE	13 (+13)	19	24

Table 6.8 Comparative datapath statistics for pipelined resource constrained solution.

System	C-Steps	Processors	Registers	Mux Inputs
MABAL [Kucukc89]	19	2+, 1x (P)	10	32
MC^2 [Grant90a]	19	2+, 1x (P)	16	16 (+14)
ASYL [Mign90]	19	2+, 1x (P)	12	26
ESC [Stok90]	19	2+, 1x (P)	15	25

Table 6.9 Other comparative statistics.

The SAVAGE system achieves a comparable level of performance over all primary optimisation criteria. The best case improvement over reported results for the total number of register required is 18%. Only the HAL system outperforms SAVAGE, and does so by lowering the utilisation of individual registers, hence increasing the number required. This impacts on the communications infrastructure, and the HAL system adopts a bus based approach. While this offers a simple control solution, the number of point-to-point connections dramatically increases. This will adversely affect the synthesis from macroarchitecture to compiled logic implementation.

The best solutions generated with respect to interconnect optimisation are the ASSIGN system, reported in [Fin92], and the Eindhoven Silicon Compiler [Stok90], which both report a 4% improvement over the SAVAGE algorithm. The ASSIGN system uses a complex signal assignment and integrated multiplexer costing function to achieve high performance, while the ESC approach uses a hybridised edge colouring algorithm to optimise local interconnect (i.e interconnect associated with an *individual* processor).

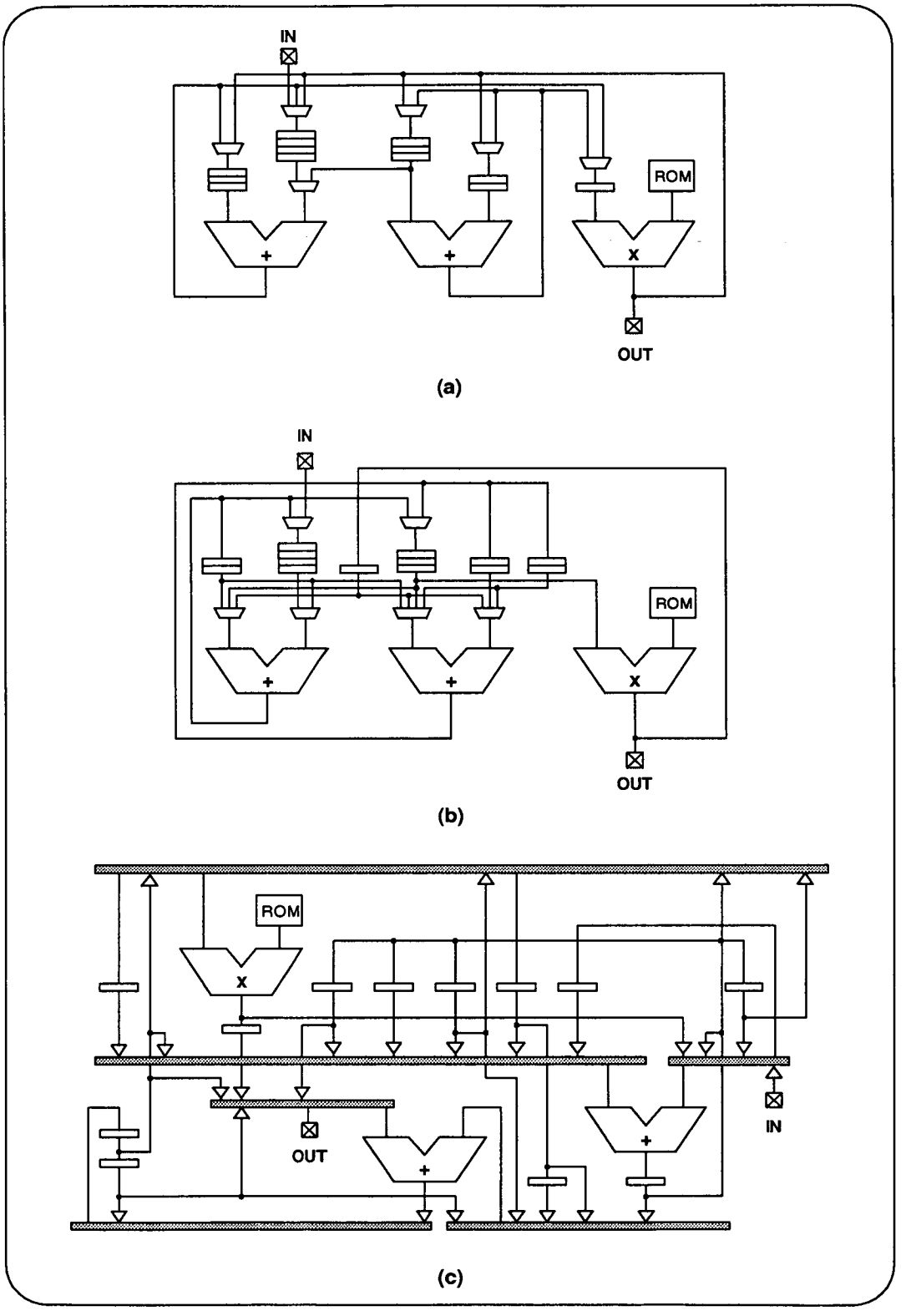


Figure 6.9 Datapath solutions for 5th Order Wave Digital Filter (a) ASSIGN[Fin92] (b) CLIQUE [Fin92] (c) HAL [Paulin89b].

6.2.2 A Maximum Speed Solution

To complete the set of Wave Digital Filter datapaths, a maximum speed solution was generated. No constraints were placed on the number or choice of functional units. For a maximum speed solution, states generated with a long overall schedule are penalised with the maximum execution time cost multiplier. For fast datapaths, the control overheads will diminish, so datapath states generated with a high control overhead are penalised. The cost multiplier values used to generate the maximum speed solution are presented in table 6.10.

Adder	25	Register File	35
Multiplier	25	ROM	1
Multiplexer	45	Maximum execution time	150
Tri-Buffer	55	Control Overhead	50
Register	50	Interconnect Density	40

Table 6.10 Cost multipliers for maximum speed solution.

Following synthesis, the datapath illustrated in figure 6.10 was produced. Again, no comparative schematics are available, and so only a comparison between the overall datapath statistics can be performed. The statistics for maximum speed solutions generated by other behavioural synthesis systems are presented in table 6.11.

System	C-Steps	Processors	Registers	Mux Inputs
SAM [Clout90]	18	2+, 2x (P)	12	27
SAW [Thomas88]	18	2+, 2x (P)	12	34
ASSIGN [Fin92]	18	2+, 2x (P)	15	17 (+13)
CLIQUE [Fin92]	18	2+, 2x (P)	14	17 (+13)
SAVAGE	18	2+, 2x (P)	13	15 (+11)

Table 6.11 Comparative datapath statistics for maximum speed solution.

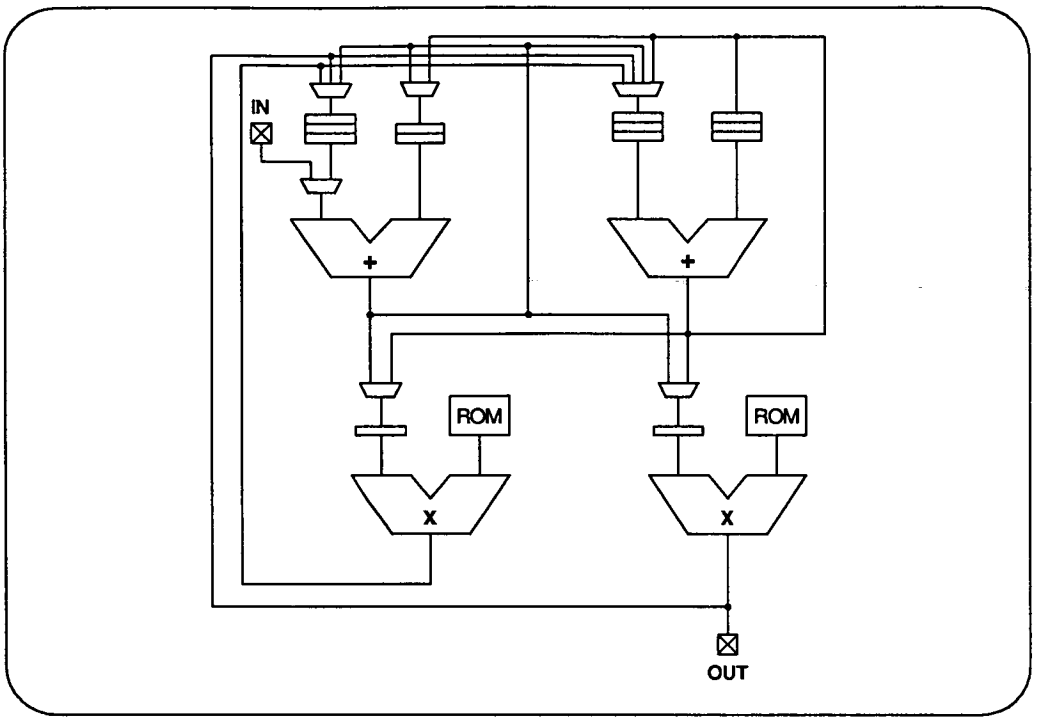


Figure 6.10 Maximum speed Wave Digital Filter Datapath.

Analysis of the results indicate that SAVAGE offers a 24% best case improvement in the number of multiplexer inputs required in the datapath communications infrastructure. Smaller relative improvements are also noted in the total number of registers required for signal storage (14% best case). The performance of the SAM and SAW systems, particularly in the register allocations attained, may be attributed to the use of manual register allocation during the synthesis procedure [Thomas88, Clout90]. Significantly, both systems use a force-based scheduler similar to that suggested by Paulin [Paulin86, Paulin89a, Paulin89b, Paulin89c]. This scheduling and allocation approach encourages the creation of single register instances as opposed to register *files*. As a result, the number of multiplexers required to implement the interconnection network increases dramatically (c.f. HAL datapath shown in figure 6.9(c)). This is reflected in the 41 point-to-point connections quoted in [Clout90] compared to the SAVAGE interconnect density figure of 29 unique connections.

6.3 Discussion and Conclusion

The SAVAGE system was exercised on two large-scale synthesis benchmarks. Resource constrained synthesis was achieved using directives present in the source text. Synthesis performance comparable to published results using other techniques was obtained in all test cases.

The performance of the SAVAGE system may be attributed to the global state generation and costing model adopted. As noted in chapter 3, graph theoretic algorithms for solution of the synthesis subtasks perform well, but can introduce local minima into the datapath solution. The performance of the synthesis systems used in comparison with the SAVAGE tools supports this contention. A global state generation technique, as advocated in this thesis, overcomes these problems. Further, the costing method developed for the SAVAGE system steers the state generation mechanism towards datapath solutions amenable to implementation in the prevalent standard cell or gate array technologies.

Finally, the SAVAGE cost multiplier system allows the design engineer to influence the overall datapath architecture without direct intervention in the synthesis procedure.

7 Synthesis of Functional Pipelines

Pipelining is a well known architectural technique for increasing the throughput of digital systems [Hwang86, Top89]. This chapter presents a pipelining algorithm amenable to simulated annealing-based synthesis. The SAVAGE tools, described in chapter 5 are extended to accommodate the generation of pipelined datapaths. The algorithm is then exercised on the large-scale synthesis benchmarks of the previous chapter. Prior to a discussion of the techniques developed, however, pipelining nomenclature and definitions are reviewed.

7.1 Pipelining Nomenclature and Definitions

A distinction is made between *structural pipelining* and *functional pipelining*. Structural pipelining is used to exploit temporal parallelism at the datapath macro- and microarchitectural levels. Functional pipelining, however, operates exclusively in the behavioural domain, and is concerned with extracting temporal parallelism from an algorithmic specification.

7.1.1 Structural Pipelining

Definition 7.1 *Processor latency, L_p, is defined as the propagation delay between data presentation at the processor inputs, and a valid data transition on the processor output.*

Consider the multiply-add-compare datapath, given in figure 7.1, which implements the function:

$$x = gtr(c, (a+b)*b) \qquad [7.1]$$

where the *gtr()* predicate returns the greater of the two input data.

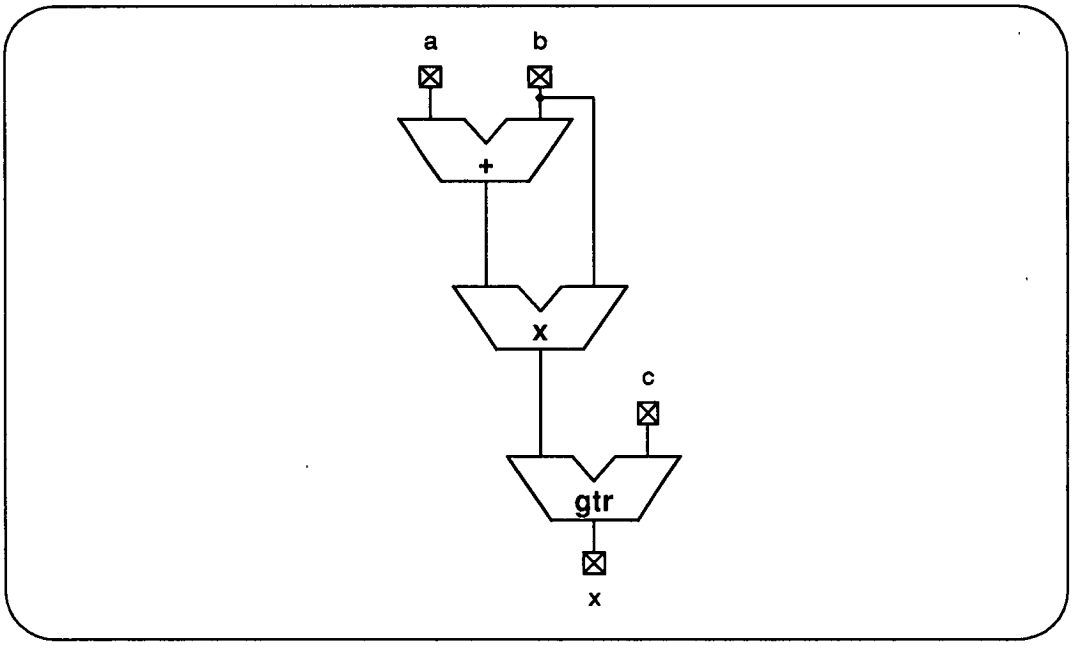


Figure 7.1 Multiply-add-compare datapath.

If the multiplier has a latency (in clock cycles) of L_m, the adder a latency of L_a, and the compare processor a latency of L_c, then the composite datapath latency, L_d, is given as:

$$L_d = L_m + L_a + L_c \qquad [7.2]$$

This represents an upper bound on the frequency of the input data. A new value must be presented to the datapath inputs once every L_d cycles. Thus, the throughput of the

datapath is given as $\frac{1}{L_d}$ cycles. Structural pipelining overlaps the individual processor operations by the introduction of a pipeline *stage* register between the component processors in the datapath. This allows the datapath processors to be operate on their input data *independently*. This basic structural pipeline is given in figure 7.2. The frequency of the pipeline cycle clock, t_c, which drives the registers is the reciprocal of the longest individual processor latency.

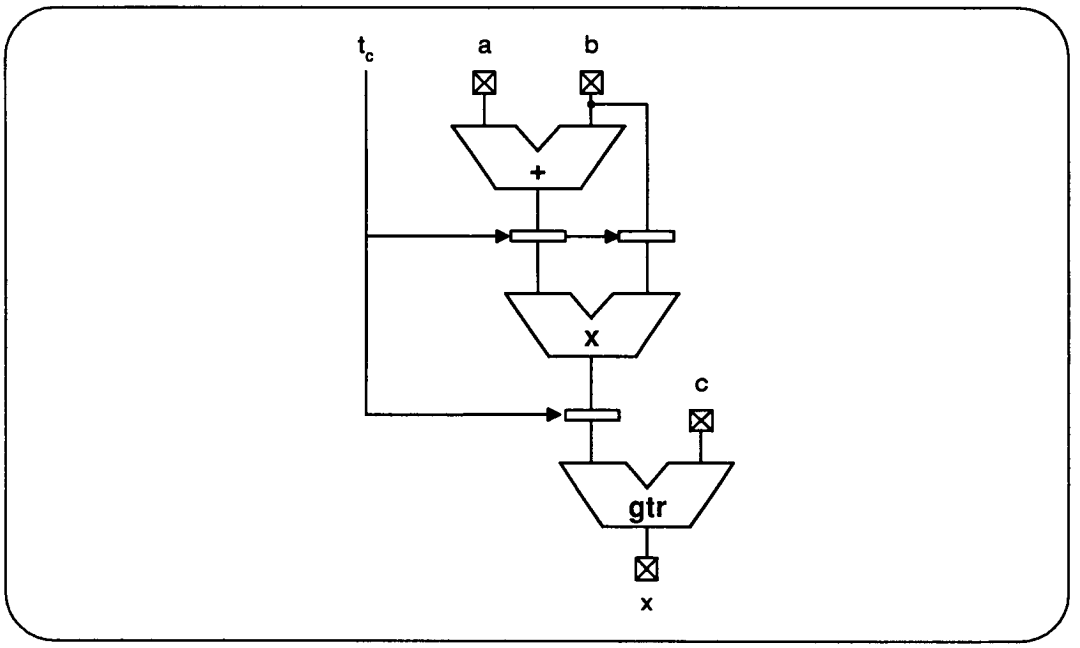


Figure 7.2 Basic structural pipeline.

Structural Pipeline Metrics

For a pipeline of length k, processing n data items, with a pipeline clock period, t_c, the *total* pipeline latency, T_k, is defined:

$$T_k(n) = k.t_c + (n\text{-}1)\, t_c \qquad [7.3]$$

where $k.t_c$ is the pipeline start-up time (i.e. the propagation delay of the first data item through the pipeline). For a non-pipelined solution (i.e. $k = 1$), equation 7.3 reduces to:

$$T_1(k) = n.tc \qquad [7.4]$$

The speed-up, $S(k)$, attained through the use of pipelining is the ratio of the non-pipelined latency to the pipelined latency. Thus:

$$S(k) = \frac{nkt_c}{kt_c + (n-1)t_c} = \frac{nk}{k+n-1} \qquad [7.5]$$

Similarly, the pipeline efficiency, $E(k)$, may be defined as the ratio of the achieved speed-up to the number of pipeline stages. This is given in equation 7.6:

$$E(k) = \frac{n}{k+n-1} \qquad [7.6]$$

Thus, a structural pipeline operates at 50% efficiency when $n = k - 1$.

7.1.2 Functional Pipelining

The aim of functional pipelining is to extract temporal parallelism from an input behaviour. In practise, this corresponds to achieving a degree of *overlap* between sequentially executing data flow graphs.

Definition 7.2 — *Pipeline latency, F_l, is specified for a given schedule as the total number of control steps required to execute the behaviour on the current resource set.*

Definition 7.3 — *Pipeline reuse time, F_r, is specified as the number of control steps between successive executions of the behaviour on the current resource set.*

Consider the data flow graph given in figure 7.3(a). An *R-t* mapping for that behaviour is illustrated in figure 7.3(b). Successive executions of this behaviour are shown shaded in the figure. The functional pipeline latency, F_l, for this mapping is six control steps. The functional pipeline reuse time, F_r, is five control steps. For *R-t* mappings with a greater number of data dependencies, and therefore a more irregular structure, a functional pipelining solution becomes more difficult to obtain.

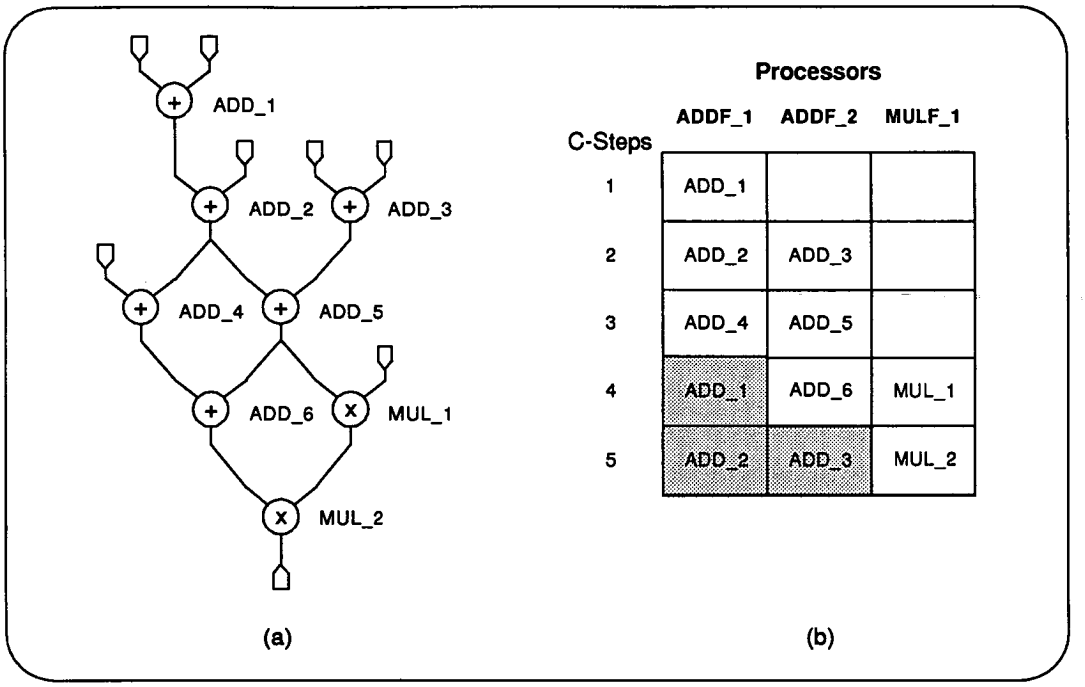


Figure 7.3 R-t mapping showing pipelined execution (a) data flow graph (b) R-t mapping.

Functional Pipeline Metrics

The *effective speed-up* achieved by functional pipelining is defined:

$$Speedup = \frac{F_r(NP) - F_r(P)}{F_r(NP)} \qquad [7.7]$$

where $F_r(P)$ is the reuse time of the pipelined solution, and $F_r(NP)$ is the reuse time of the non-pipelined solution.

7.2 A General-Purpose Functional Pipelining Algorithm

Consider the original *R-t* mapping of the example presented in figure 7.3(b). An *R-t hyperplane* can be created by overlaying a time-delayed version of the original (referred to as the *base R-t map*) as shown in figure 7.4. The value of the delay is equal to the maximum execution time present in the base schedule.

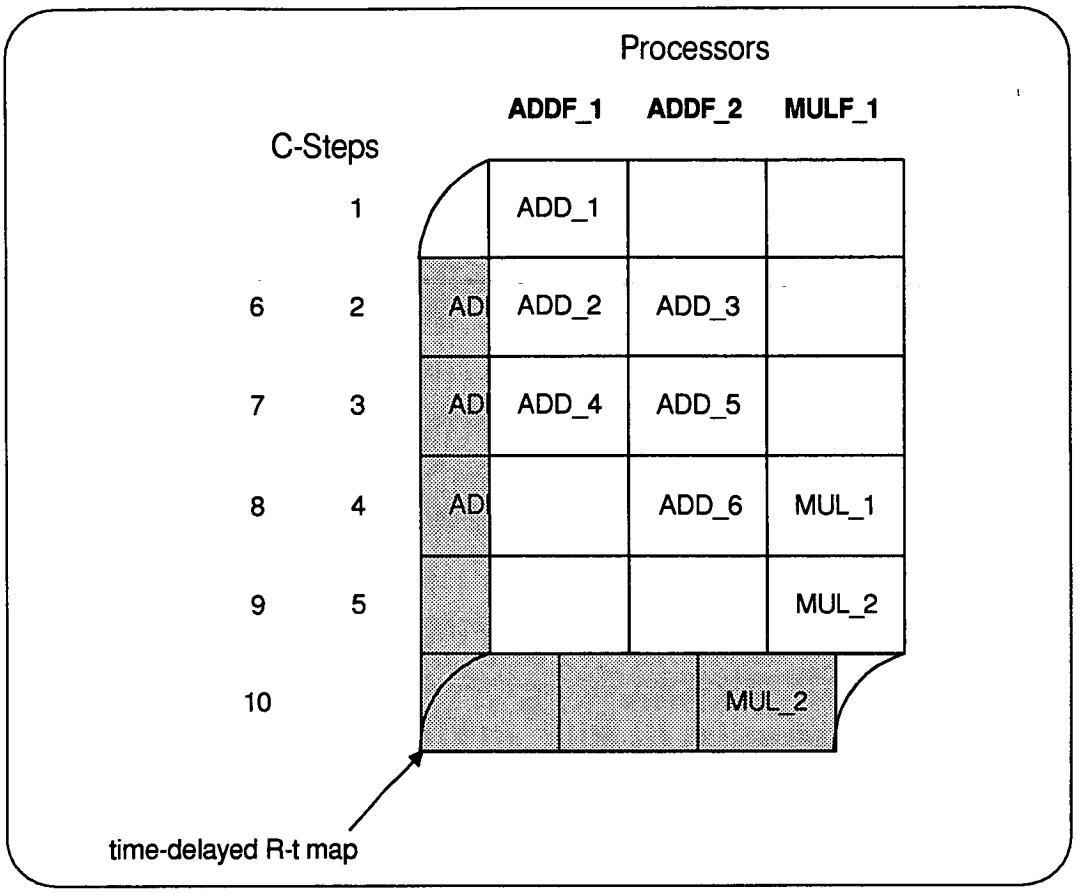


Figure 7.4 R-t hyperplanes. Delayed version is shaded.

Overlapping the execution of data flow graphs may then be represented by the transfer of operations from the base *R-t* map to the time-delayed version. As the operation is transferred to the time-delayed *R-t* map, a *phantom* operation is placed in the corresponding grid location in the base map. This phantom has no behavioural significance, but prevents any move occurring in the base *R-t* map which would cause a resource clash in subsequent executions of the behaviour. This process is illustrated in figure 7.5.

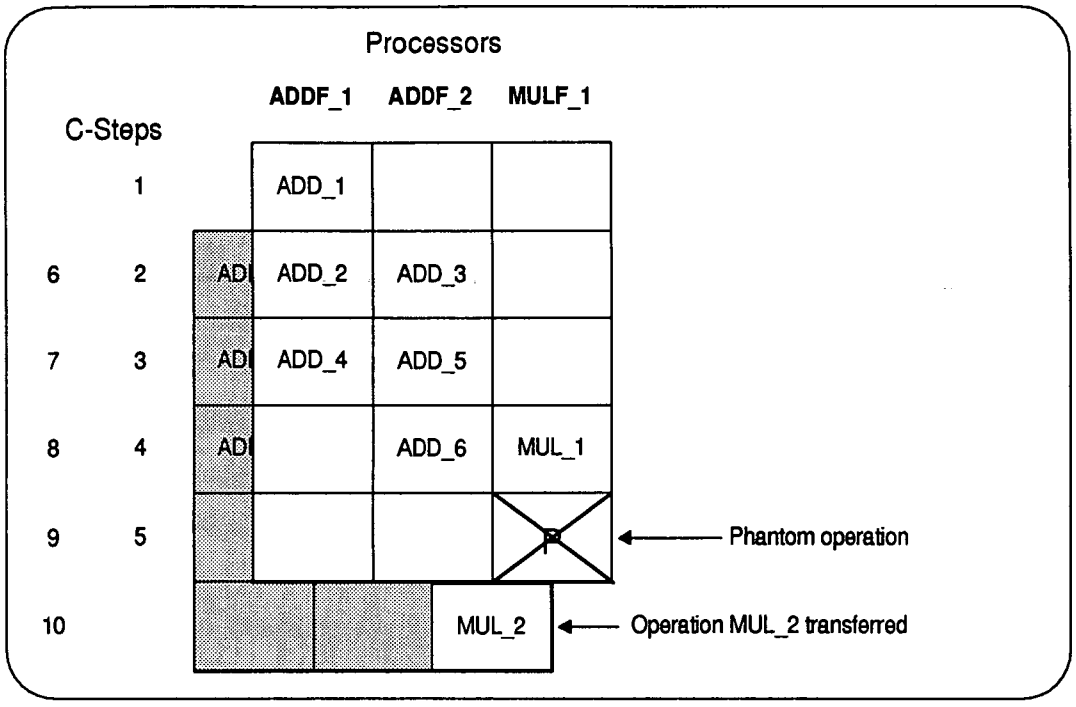


Figure 7.5 Transfer between R-t planes and the use of phantom operations.

The pipeline reuse time, F_r, corresponds to the best "fit" between base and time-delayed mappings as illustrated by the examples in figure 7.6 (successive executions of the *R-t* map are shown in gray). In many cases, however, this value will be equal to the maximum execution time present in the base *R-t* map. Thus, minimisation of F_r corresponds to a compaction of base operations within the *R-t* hyperplane. The pipeline latency, F_l, is defined as the maximum execution time of base operations in the time-delayed *R-t* space.

7.2.1 A SAVAGE Implementation

The SAVAGE toolset was extended to incorporate the optimisation algorithm described above. The extensions to the software were partitioned into two components:

(i) Pipeline generation moves.

(ii) Pipeline cost assessment.

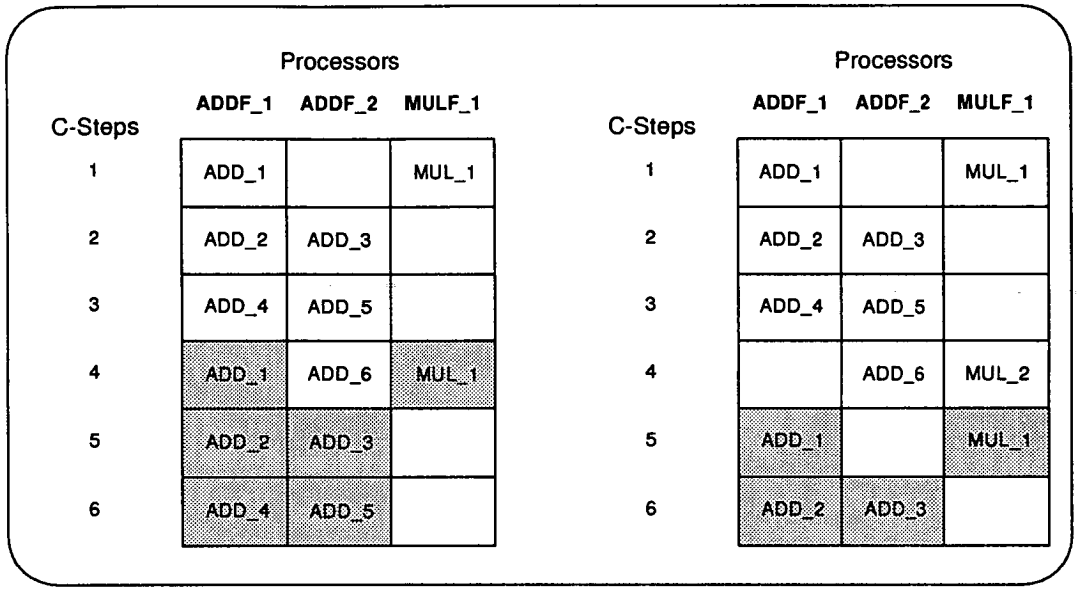


Figure 7.6 Best-fit pipeline reuse examples.

The synthesis of pipelined datapaths is performed exclusively in the *R-t* domain. Correspondingly, only optimisation moves associated with this plane are considered.

The state generation sequence proceeds as described in section 5.2.2, with the following additions: if the scheduled execution time of the operation selected is equal to the maximum execution time present in the base *R-t* mapping, then the operation is a candidate for deferral to the following execution of the behaviour. This is in keeping with the overall goal of seeking to minimise the maximum execution time present in the base *R-t* map. Where candidate a operation for deferral is identified, the `PIPE` package (described below) is added to the valid move set for that operation. Further, if the boolean flag (**operation.delayed**) associated with the selected operation is already set, then *only* the `PIPE` package supplies the valid moves.

Data Structure Addenda

A boolean flag is appended to each operation on the processor allocation lists in the linked-list implementation of *R-t* space described in section 5.1.4. The flag indicates

that the associated operation is executing in the time-delayed version of *R-t* space. The **operation.execution_time** field remains *unaffected* by the setting of the **operation.delayed** flag. Thus, the need for a second *R-t* space, and the provision of phantom operators is avoided. The execution time of an operation, and its processor allocation, are modified by the pipeline procedures. In addition, the presence of the operation in base *R-t* space prevents other (i.e. non-pipelined) optimisation moves causing resource clashes. Signal lifetimes for delayed operations are computed by adding the **operation.execution_time** field to the maximum execution time present in the base *R-t* map.

Pipeline Generation Moves

The `PIPE` package contains the valid move set for base *R-t* operations mapped into the delayed *R-t* space. The initial pipeline optimisation move for an operation in base *R-t* space is to set the **operation.delayed** flag, and effectively delay the operation by an entire schedule length. The other optimisation moves mimic the behaviour of the move set described in section 5.3.1. Only the base scheduling procedure is different. Whereas the `UNARY_STEP` function looks for a valid schedule as control steps *c-1*, *c* and *c+1* the `PIPE_UNARY_STEP` function looks for the *first available* execution times on the current processor, as illustrated in figure 7.7. This allows effective *R-t* compaction, and is permissible where the input data for the selected operation are produced in the base *R-t* map. Where the producer nodes are also mapped into the delayed *R-t* space, data flow precedence between the operations applies.

With this basic scheduling mechanism in place, variants of the non-pipelined *R-t* optimisation moves can be added to the valid move set. The valid pipeline moves are `PIPE_UNARY_STEP`, `PIPE_VALID_PROCESSOR`, `PIPE_CREATE_PROCESSOR` and `PIPE_FUNCTION_MERGE`. A final scheduling move allows operations mapped into the delayed *R-t* space to be rescheduled back into base *R-t* space.

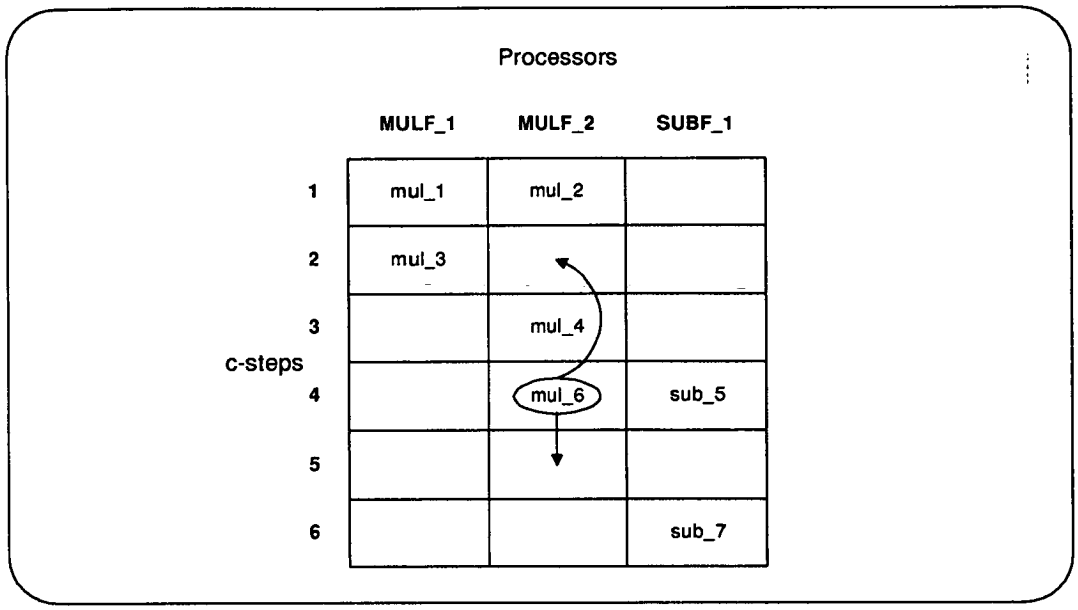


Figure 7.7 Basic pipeline scheduling move (PIPELINE_UNARY_STEP).

Pipeline Cost Assessment

The SAVAGE cost assessment function was extended to include terms for pipeline costs. Pipeline latency, F_l, replaces the MAXIMUM EXECUTION term, and the pipeline reuse time, F_r, is added to the equation. The pipeline latency terms is computed as:

$$F_l = MAX(Base\ R\text{-}t) + MAX(Delayed\ Operations) \qquad [7.8]$$

where the *MAX()* predicate returns the greatest value of **operation.execution_time** present in the input set. The pipeline reuse time, F_r, may be computed using the "best fit" method by using the algorithm given in figure 7.8.

7.3 Examples

Pipelined solution datapaths for the synthesis benchmarks introduced in the previous chapter, the 1-dimensional 8 point Fast Discrete Cosine Transform and a 5th order Wave Digital Filter, are presented.

```
function BEST_FIT (RT : in out RT_SPACE) return INTEGER is

begin
  -- Start the search at the maximum execution time in base R-t
  FR := MAX(RT); INDEX := 1; FLAG := true;
  while FLAG loop;
      -- Search over all processors
      for each processor in RT do
          if not((SLOT_AVAILABLE(FR) and OP_SCHEDULED_AT(INDEX)) or
                 (SLOT_AVAILABLE(INDEX) and OP_SCHEDULED_AT(FR))) then
                 -- R-t space will fit
                 FLAG := false;
          end if;
      end for;
      - If fit at this value, then decrement and start again
      if FLAG then
          FR := FR - 1; INDEX := INDEX + 1;
      end if;
  end loop;
  -- Return Pipeline Reuse Time
  return FR;

end BEST_FIT;
```

Figure 7.8 Best-fit pipeline reuse algorithm.

7.3.1 A Pipelined Fast Discrete Cosine Transform Datapath

The resource-constrained description of the FDCT presented in figure 6.2 was re-synthesised by the SAVAGE system. The PIPE procedure, described in section 7.2 was included in the *R-t* optimisation move set.

The specification of separate cost multipliers for F_r and F_l allows an optimisation trade-off between pipeline reuse time and overall pipeline latency. The goal of this re-synthesis was the generation of a pipelined solution with a minimised pipeline reuse time. Correspondingly, the F_r cost multiplier is assigned a higher value than that associated with F_l. By permitting potential increases in the schedule, the signal storage requirements may also increase. The cost multipliers associated with memory components are also therefore increase in order to minimise this effect. To maintain a balanced datapath solution, however, the pipeline cost multipliers are not given great precedence over the other datapath component cost multipliers. The full set of cost

multipliers used to generate a pipelined solution is presented in table 7.1.

Adder	30	Register File	60
Multiplier	30	ROM	1
Multiplexer	50	Pipeline Latency	75
Tri-Buffer	80	Pipeline Reuse	95
Register	40	Control Overhead	75
		Interconnect Density	75

Table 7.1 Cost multipliers for pipelined datapath generation.

Following synthesis, the datapath illustrated in figure 7.9 was generated. As the pipelining algorithm operates solely in the *R-t* plane, the datapath schematic is unrevealing with respect to the optimisation achieved. For comparative purposes, the *R-t* space of the non-pipelined solution (presented in section 6.1) and the pipelined *R-t* space are given in figure 7.10. Subsequent executions of the R-t mappings as presented are indicated by the shaded areas.

Comparison of the datapath topologies, as presented in table 7.2 indicates an overall gain of 4% in the number of multiplexers required to implement the pipelined solution. This minor increase may be directly attributed to the flexibility of the SAVAGE costing mechanism.

Applying equation 7.7, it can be observed that a 33% speed-up is gained through the use of functional pipelining. This is set against a 25% increase in the total pipeline latency

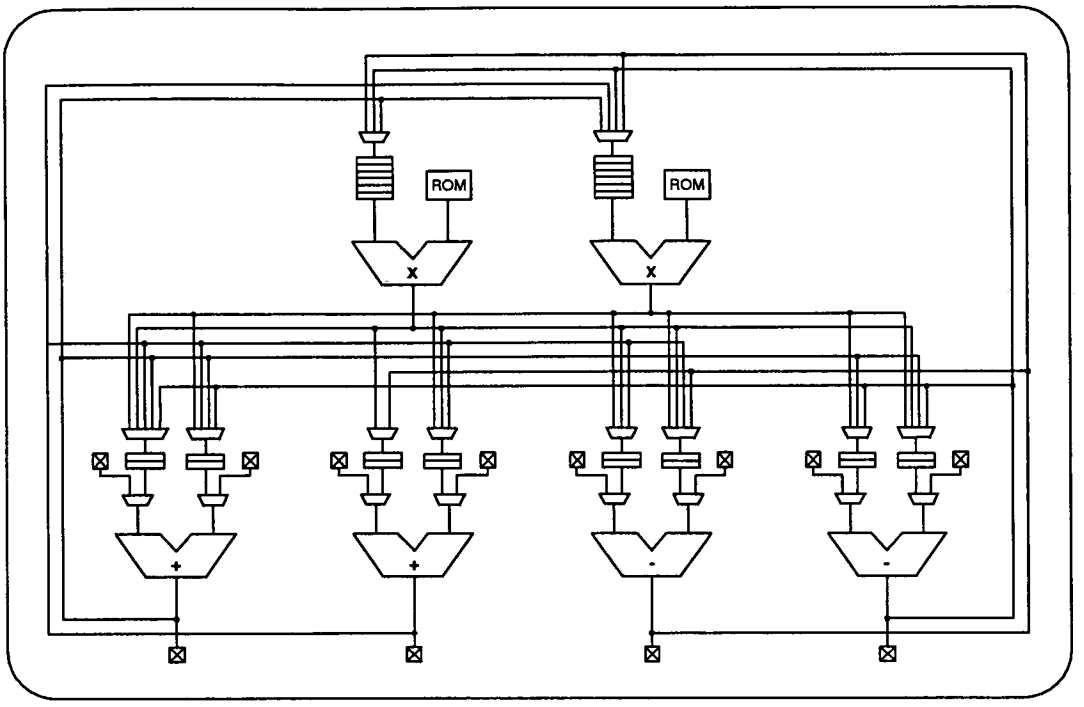


Figure 7.9 Pipelined FDCT datapath.

Processors

C-Steps

	ADDF_1	ADDF_2	MULF_1	MULF_2	SUBF_1	SUBF_2
1	ADD_4	ADD_2			SUB_7	SUB_6
2	ADD_1	ADD_3	MUL_14	MUL_13	SUB_8	SUB_5
3	ADD_10	ADD_22				SUB_21
4	ADD_27	ADD_30			SUB_29	SUB_28
5	ADD_9		MUL_34	MUL_38	SUB_11	SUB_12
6			MUL_33	MUL_32		
7			MUL_36	MUL_31	SUB_42	
8			MUL_37	MUL_35	SUB_41	
9	ADD_39	ADD_40	MUL_17	MUL_19		
10			MUL_18	MUL_16		SUB_25
11			MUL_20	MUL15		
12	ADD_26	ADD_23				SUB_24
13	ADD_4	ADD_2			SUB_7	SUB_6
14	ADD_1	ADD_3	MUL_14	MUL_13	SUB_8	SUB_5
15	ADD_10	ADD_22				SUB_21

(a) Non-pipelined

Processors

C-Steps

	ADDF_1	ADDF_2	MULF_1	MULF_2	SUBF_1	SUBF_2
1	ADD_4	ADD_2			SUB_7	SUB_6
2	ADD_1	ADD_3	MUL_14	MUL_13	SUB_8	SUB_5
3	ADD_10	ADD_22				SUB_21
4	ADD_27	ADD_30			SUB_29	SUB_28
5	ADD_9		MUL_34	MUL_38	SUB_11	SUB_12
6			MUL_33	MUL_32		
7			MUL_36	MUL_31		
8			MUL_37	MUL_35	SUB_41	
9	ADD_4	ADD_2	MUL_17	MUL_19	SUB_7	SUB_6
10	ADD_1	ADD_3	MUL_14	MUL_13	SUB_8	SUB_5
11	ADD_10	ADD_22	MUL_18	MUL_16	SUB_42	SUB_21
12	ADD_27	ADD_30	MUL_20	MUL15	SUB_29	SUB_28
13	ADD_9	ADD_39	MUL_34	MUL_38	SUB_11	SUB_12
14	ADD_40	ADD_26	MUL_33	MUL_32	SUB_25	SUB_24
15	ADD_23		MUL_36	MUL_31		

(b) Pipelined

Figure 7.10 (a) Non-pipelined R-t space (b) Pipelined R-t space for FDCT example.

System	Reuse	Latency	Registers
Non-Pipelined	12	12	28
Pipelined	8	15	28
System	**Mux Inputs**	**Control Wires**	**Point to Point Connections**
Non-Pipelined	51 (+28)[a]	28	77
Pipelined	52 (+28)	28	75

a. Register File Decoding

Table 7.2 Statistics for pipelined and non-pipelined datapaths.

7.3.2 A Pipelined Wave Digital Filter Datapath

The Wave Digital Filter example has an irregular data flow structure, and as such, is not viewed as a good candidate algorithm for functional pipelining. The resource-constrained example of two adders and a single pipelined multiplier, described in section 6.2.1, was re-synthesised in order to illustrate the generality of the SAVAGE algorithm. Again, the pipeline reuse time was specified as the primary optimisation criterion. The cost multipliers associated with the pipeline re-synthesis are presented in table 7.3.

Adder	25	Register File	45
Multiplier	25	ROM	1
Multiplexer	45	Pipeline Latency	75
Tri-Buffer	55	Pipeline Reuse	95
Register	60	Control Overhead	60
		Interconnect Density	80

Table 7.3 Cost multipliers for pipelined datapath generation.

Following synthesis, the datapath shown in figure 7.11 was generated. The optimised

R-t spaces for the pipelined and non-pipelined solutions are shown in figure 7.12. Comparative datapath statistics are presented in table 7.4.

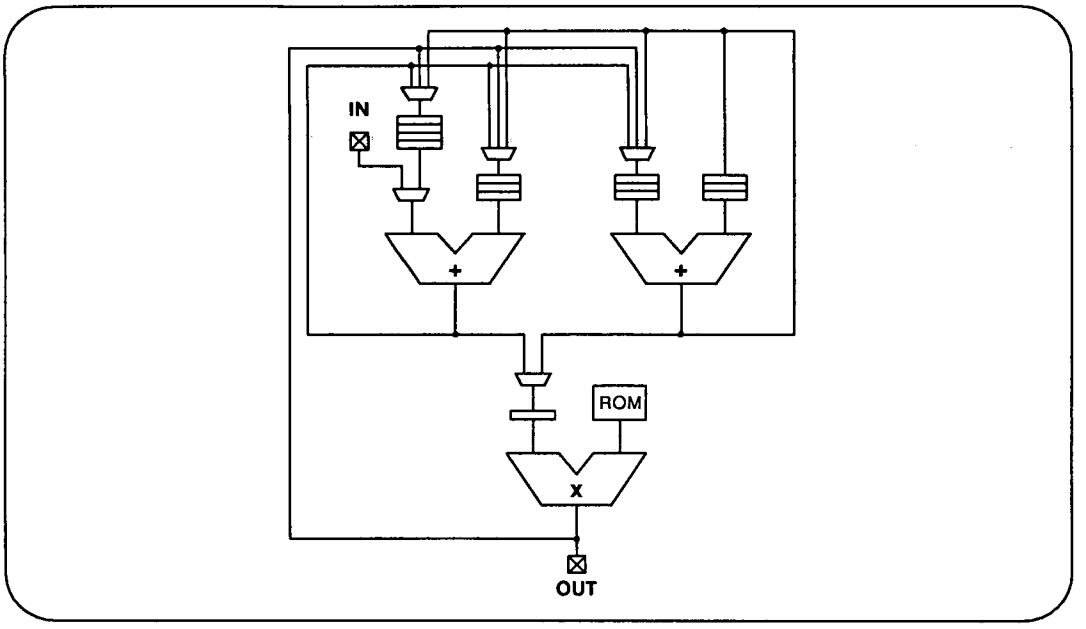


Figure 7.11 Pipelined WDF datapath.

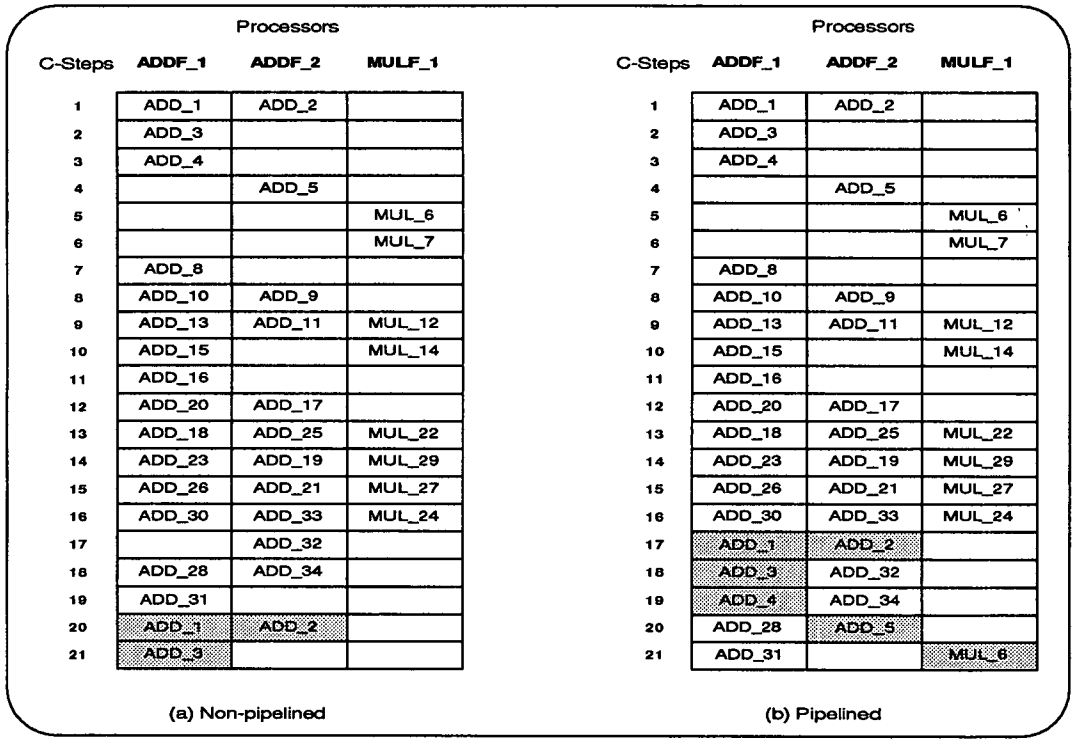
System	Reuse	Latency	Registers
Non-Pipelined	19	19	13
Pipelined	16	21	14
System	**Mux Inputs**	**Control Wires**	**Point to Point Connections**
Non-Pipelined	13 (+13)	19	24
Pipelined	13 (+14)	19	24

Table 7.4 Statistics for pipelined and non-pipelined datapaths.

The increased number of registers required is caused by the extension of the schedule to incorporate functional pipelining. The data storage required for the delayed execution of the ADD_32 operation causes a signal lifetime clash, and thus forces the

addition of a further register. The associated control overhead is reflected in the increased number of multiplexer inputs required for register file decoding.

Equation 7.7 indicates a 16% speed-up gained through the use of functional pipelining. Again, this must be set against a 10.5% increase in the overall solution latency.



Processors

C-Steps	ADDF_1	ADDF_2	MULF_1
1	ADD_1	ADD_2	
2	ADD_3		
3	ADD_4		
4		ADD_5	
5			MUL_6
6			MUL_7
7	ADD_8		
8	ADD_10	ADD_9	
9	ADD_13	ADD_11	MUL_12
10	ADD_15		MUL_14
11	ADD_16		
12	ADD_20	ADD_17	
13	ADD_18	ADD_25	MUL_22
14	ADD_23	ADD_19	MUL_29
15	ADD_26	ADD_21	MUL_27
16	ADD_30	ADD_33	MUL_24
17		ADD_32	
18	ADD_28	ADD_34	
19	ADD_31		
20	ADD_1	ADD_2	
21	ADD_3		

(a) Non-pipelined

Processors

C-Steps	ADDF_1	ADDF_2	MULF_1
1	ADD_1	ADD_2	
2	ADD_3		
3	ADD_4		
4		ADD_5	
5			MUL_6
6			MUL_7
7	ADD_8		
8	ADD_10	ADD_9	
9	ADD_13	ADD_11	MUL_12
10	ADD_15		MUL_14
11	ADD_16		
12	ADD_20	ADD_17	
13	ADD_18	ADD_25	MUL_22
14	ADD_23	ADD_19	MUL_29
15	ADD_26	ADD_21	MUL_27
16	ADD_30	ADD_33	MUL_24
17	ADD_1	ADD_2	
18	ADD_3	ADD_32	
19	ADD_4	ADD_34	
20	ADD_28	ADD_5	
21	ADD_31		MUL_6

(b) Pipelined

Figure 7.12 (a) Non-pipelined R-t space (b) Pipelined R-t space for WDF example

7.4 Discussion

A general functional pipelining algorithm was presented which is amenable to a simulated annealing-based implementation. The algorithm was shown to improve the pipeline reuse time. This improvement must be offset against the increase in solution latency. In the case of the irregular WDF example, the attained speed-up can only be justified in applications with very high data rates.

A similar iterative grid compaction method for functional pipelining is reported in [Mallon90]. For this iterative solution, however, a target reuse time must be specified as a constraint to the system. The scheduling algorithm then exhaustively searches *R-t* space to determine whether a solution satisfying the constraint exists. For practical problem instances, this may pose an unacceptable computational overhead.

The SAVAGE system does not require such a constraint on its input. The specification of separate pipeline reuse and latency cost multipliers allows designers to evaluate the trade-off between minimising pipeline reuse time and overall pipeline latency. Further, the implementation method reported here is fully integrated into the SAVAGE synthesis system, and does not require a specialist pipeline scheduler *per se*.

8 Summary and Conclusions

The behavioural synthesis task was defined as the mapping of an algorithmic specification, captured in a high-level programming language, to an optimised datapath topology capable of executing the specification at the register-transfer level. The synthesis procedure can be partitioned into a number of subtasks. The relationship between these subtasks is typified by a serial synthesis flow, with the scheduling and allocation operations occurring prior to register allocation and interconnect optimisation. These subtasks are commonly specified as a directed-graph problem.

Solution techniques for these directed-graph problems were reviewed. These are drawn from the branch of mathematics known as *algorithmic graph theory.* In many cases, however, the solution techniques belong to a class of computational problem for which no exact solution may be generated in polynomial time. This limitation is overcome by the introduction of heuristics to constrain the solution space. The use of heuristics introduces the possibility that the solution may reside in a local cost

minimum, however, thus compromising the solution quality. This observation is borne out by the appearance of local minima in the results quoted in this thesis.

A global formulation of the synthesis problem was proposed. A combinatorial approach is adopted, and a candidate optimisation technique known as simulated annealing, drawn from the field of statistical physics, was introduced. The simulated annealing algorithm demonstrates the ability to escape from local minima by accepting temporary solutions whose overall cost function is higher than that of the current state.

A behavioural synthesis method based upon the simulated annealing algorithm was developed. In common with other simulated annealing-based applications, the problem was formulated as a two-dimensional grid or *plane*. A novel feature of the data structure developed was the provision of three interconnected optimisation planes capable of independently supporting the operation of the simulated annealing algorithm. This overcomes the quasi-serial approach reported in other simulated annealing solutions to the behavioural synthesis task. A full range of primitive state generation moves was presented, ordered by optimisation plane. An extended costing method was specified which was directed towards the prevalent ASIC implementation technologies. A further innovation was the provision of a cost multiplier system, which allowed design engineers to influence the final datapath architecture without direct synthesis intervention. This intervention is supported, however, through the use of synthesis *pragmas*.

The result of this research was the SAVAGE system, a suite of modular software tools. Small and large scale synthesis benchmarks were used to exercise the tools over a wide range of optimisation criteria. The results presented offer a favourable comparison between the SAVAGE system and other behavioural synthesis tools.

SAVAGE offers reductions of up to 30% in the total number of multiplexer and register instances. Further, SAVAGE consistently generated solutions with a low interconnect density. This was identified as a key factor contributing to the overall silicon area in the datapath implementation. This improvement may be directly attributed to the global state generation and costing method adopted.

The synthesis paradigm presented in this thesis was extended by the incorporation of a general purpose algorithm capable of generating functional pipelines into the SAVAGE system. A set of modular synthesis primitives and pipeline cost assessment criteria was developed and integrated into the state generation and costing mechanisms. The algorithm produced pipelined solutions capable of operating up to 33% faster than a non-pipelined solution. The general nature of the approach was demonstrated using the Wave Digital Filter benchmark. The irregular data flow structure hinders effective functional pipelining. The SAVAGE approach yielded a 15% speed-up in spite of this. By keeping the costing factors associated with pipeline latency and pipeline reuse separate, architectural trade-offs between pipeline reuse and overall latency are possible.

8.1 Further Work

This section suggests a number of potential research areas which extend and augment the work presented in this thesis.

8.1.1 A Route to Silicon

A link is proposed between the SAVAGE toolset and the logic synthesis, placement and routing tools which implement the SAVAGE-generated datapath macroarchitecture in the target technology. The most obvious advantage of this integration is the increased accuracy of the SAVAGE cost assessment function. Technology models for datapath component area and placement and routing overheads

would be made directly available to the SAVAGE costing function. While it is impractical to suggest that physical data should be back-annotated at each state generation, a more realistic goal would be to back-annotate at each control parameter decrement.

The disadvantage of providing a link to the microarchitectural and physical synthesis tools is that SAVAGE system would no longer be technology-independent.

8.1.2 Synthesis using Structural Input

A great deal of promising research has been performed which concentrates on the provision of structural as well as behavioural input to the synthesis tools [Fin92]. In summary, an engineer may specify a partial or complete datapath on which the input behaviour should execute. This structural input can be used as the starting point for an iterative synthesis based upon a desired macroarchitecture, or could perhaps provide a mechanism for reusing synthesised datapaths.

The SAVAGE system could be updated to accept structural input. Modifications to the BUILDER module would allow a constrained initial state based upon the structural specification to replace the randomly generated *R-t*, *M-t* and *P-c* spaces. The reuse of pre-defined datapaths would impose a *hard* structural constraint, and as such, the state generation mechanism would suppress any optimisation moves which would potentially alter the datapath structure from the valid move set. A *soft* constraint. where the pre-defined datapath acts as a starting point for the synthesis procedure would not require such a reduced move set. In both instances, however, the state generation mechanism and the cost assessment criteria remain unchanged.

8.1.3 Design for Testability

Design for testability is of great concern in the large-scale integrated system designs currently being undertaken in academia and industry. A modular extension, similar to that presented in the previous chapter, is suggested. This extension is composed of three major components.

The first provides a set of cost assessment criteria targeted specifically at datapath testability. Quantifying circuit testability has been the focus of a great deal of research [Ben81, Ben84, Lala85], and a mature set of metrics are in general use. The most widespread are the concepts of *Nodal Controllability (CY)*, *Nodal Observability (OY)* and *Nodal Testability (TY)*, first suggested by Bennetts [Ben84].

These metrics could be used to build composite testability costs for datapath components at the datapath macroarchitectural level. The SAVAGE costing function could then be extended to incorporate a testability factor based on nodal *TY* values within the datapath.

The second component of a design for test synthesis module is the datapath microarchitecture component library. In order to assess *CY*, *OY* and *TY* values during state generation, accurate testability models of the datapath microarchitecture need to be generated.

Finally, datapath components incorporating design for test features such as multiplexed registers with serial scan inputs or dedicated test multiplexers to provide access to nodes with low *TY* values should be generated for use in the SAVAGE resource set. A dedicated test state generation move set is required to successfully integrate test features into SAVAGE system. It is anticipated that the development of

such a state generation move set could be modelled on the development of the pipelining module presented in the previous chapter.

8.1.4 An Architectural Script-based Design Paradigm

De Man [DeMan90] suggests a synthesis environment based around the *architectural script* concept. The architectural script specifies all levels of interaction with the software tools prior to, and during the synthesis procedure. This specification takes place at a number of levels:

(i) **System Level Constraints.** The major optimisation goals for the synthesis tools are detailed. These specify system boundaries, such as maximum execution time and total datapath area.

(ii) **Architectural Selection.** The designer can influence the final macroarchitecture of the datapath through selection of synthesis routines from an architectural library. The synthesis routines in this library are grouped according to their architectural template (e.g. bit-serial, regular array, communicating multiprocessor). This level of interaction arises directly from the CATHEDRAL experience [Clae86, DeMan88, Note88].

(iii) **Design Pragmas.** Pragmas provide a mechanism for designer intervention during the synthesis procedure. A good example of this level of interaction is a pragma which binds a subset of data flow operations to execute on a particular processor, as may be necessary with a speed critical loop structure.

(iv) **Structural Specification.** At this lowest level, the design engineer can specify a datapath macroarchitecture directly, and force the synthesis tools to optimise the mapping between the specified behaviour and the pre-defined structure.

Underlying the architectural script-based method is the need for a high quality optimisation engine. This thesis has promoted simulated annealing as such a procedure. The current SAVAGE toolset permits interaction at levels (i) and (iii) defined above, and an extension is proposed (section 8.2.2) to include interaction at level (iv). Further, the work presented in this thesis has demonstrated the ability of a modular software system, such as SAVAGE, to support the inclusion of architecture-specific optimisation routines.

The work required to explore the addition of further architecture-specific synthesis procedures within the general SAVAGE framework and to develop an extended system based upon De Man's architectural script represents the most substantial extension to the current research.

References

[Aar85] Aarts, E.H.L., and van Laarhoven, P.J.M., "Statistical Cooling : A General Approach to Combinatorial Optimization Problems," *Philips J. Res.*, Vol. 40,1985, pp. 193-226.

[Aar89] Aarts, E.H.L., and Korst, J., *Simulated Annealing and Boltzmann Machines*, John Wiley, 1989.

[Afgha86] Afghahi, M., Matsumura, S., Pencz, J., Sikstrom, B., Sjostrom, U. and Wanhammer, L., "An array processor for 2-D discrete cosine transforms," in *Proc. EUSIPCO 86*, September 1986, pp. 1283-1286.

[Aho86] Aho, A.V., Sethi, R. and Ullman, J.D., *Compilers : Principles, Techniques and Tools*, Addison Wesley Publishing Company, 1986.

[Anderson91] Anderson, S., Bruce, W.H., Denyer, P.B., Renshaw, D. and Wang, G., " A Single Chip Sensor and Image Processor for Fingerprint Verification," in *Proc CICC'91*, 1991.

[Anderson93] Anderson, S., *A VLSI Smart Sensor-Processor for Fingerprint Comparison*, Ph.D. Thesis, Department of Electrical Engineering, University of Edinburgh, 1993.

[Ben81] Bennets, R.G., Maunder, C.M. and Robinson, G.D., "CAMELOT : a computer-aided measure for logic testability," *IEE Proceedings*, Vol. 128, Part E, No. 5, 1981, pp. 177-189.

[Ben84] Bennetts, R.G., *Design of Testable Logic Circuits*, Addison Wesley Publishing Company, 1984

[Bin78] Binder, K., *Monte Carlo Methods in Statistical Physics*, Springer-Verlag, New York, 1978.

[Brayton84] Brayton, R.K., Hachtel, G.D., McMullen, C.T. and Sangiovanni-Vincentelli, A., *ESPRESSO-IIC: Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.

[Brayton88] Brayton, R.K., Camposano, R.K., De Micheli, G., Otten, R.H.J.M., and van Eindhoven, J., "The Yorktown Silicon Compiler System," in *Silicon Compilation*, Gajski *(ed)*, Addison Wesley Publishing Company, 1988, pp. 204-310.

[Broder89] Brodersen, R.W., *Architectures for Digital Signal Processing*, Oxford/Berkeley Summer Engineering Programme, 12-14 July, 1989.

[Campos89] Camposano, R. and Tablet, R.M., "Design representation for the Synthesis of Behavioural VHDL Models," in *Proc. 9th Int. Conf. Comp. Hardware Description Languages*, May 1989.

[Campos90] Camposano, R. and Bergamaschi, R.A., "Synthesis using Path-Based Scheduling : Algorithms and Exercises," in *Proc. 27th Design Automat. Conf.*, June 1990, pp. 450-455.

[Catthoor88] Catthoor, F., Rabaey, J., Goossens, G., Van Meerbergen, J., Jain, R., De Man, H. and Vanderwalle, J., "Architectural Strategies for an ApplicationSpecific Synchronous Multiprocessor Environment", *IEEE Trans. Acoustics, Speech and Signal Processing*, Vol. 36, No. 2, February 1988, pp. 265-284.

[Chen77] Chen, W. and Smith, C.H., "Adaptive Coding of Monochromatic and Colour Images", *IEEE Trans. Commun.*, Vol. 25, No. 11, pp. 1285-1292, 1977.

[Clae86] Claesen, L., Catthoor, F., De Man, H., Vandewalle, J., Note, S. and Mertens, K., "A CAD Environment for the thorough Analysis, SImulation and Characterisation of VLSI Implementable DSP Systems," in *Proc. ICCD*, 1986, pp. 72-75.

[Clout90] Cloutier, R.J. and Thomas, D.E., "The Combination of Scheduling, Allocation and Mapping in a Single Algorithm," in *Proc. 27th Design Automat. Conf.*, June 1990, pp. 71-76.

[Cook71] Cook, S.A., "The Complexity of Theorem Proving Procedures," in *Proc. 3rd ACM Symp. on the Theory of Computing*, 1971, pp. 151-158.

[Dantzig54] Dantzig, G.B., Fulkerson, D.R., and Johnson, S.M., "Solution of a large-scale travelling salesman problem," *Oper. Res. 2*, pp. 393-410, 1954.

[Decaluw89] Decaluwe, J., Rabaey, J., Van Meerbergen, J. and De Man, H., "Interprocessor Communication in Synchronous Multiprocessor Digital Signal Processor Chips," *IEEE Trans. Acoustics, Speech and Sig. Proc.*, Vol 37, No.12, December 1989, pp. 1816-1828.

[De Man88] De Man, H., Rabaey, J., Vanhoof, J., Goossens, G., Six, P. and

Claesen, L., "CATHEDRAL II - a computer aided synthesis system for digital signal processing VLSI systems", *IEE Computer Aided Engineering Journal*, April 1988, pp. 55-66.

[DeMan90] De Man, H., "High Level Synthesis Tutorial," *EDAC90*, March 1990.

[Denyer82] Denyer, P.B., Renshaw, D. and Bergmann, N.W., "A Silicon Compiler for VLSI Signal Processors," in *Proc. ESSCICR82*, 1982, pp. 215-218.

[Denyer89] Denyer, P.B., *SAGE Design Methodology*, SARI Internal Technical Report SARI-035-D, March 1989.

[Dev87] Devedas, S. and Newton, A.R., "Algorithms for Hardware Allocation in Data Path Synthesis," in *Proc. ICCAD '87*, 1987, pp. 526-531.

[Dev89] Devedas, S., and Newton, A.R., "Algorithms for Hardware Allocation in Data Path Synthesis," *IEEE Trans. CAD*, Vol. CAD-8, No.7, July 1989, pp. 768-781.

[Dew85] Dewilde, P., Deprettere, E., and Nouta, R., "Parallel and Pipelined VLSI Implementations of Signal Processing Algorithms," in *VLSI and Modern Signal Processing*, Kung, Whitehouse & Kailath *(eds)*, Prentice Hall, 1985, pp. 257-276.

[Fel70] Feller, W., *An Introduction to Probability Theory and Applications*, J. Wiley, 1970.

[Fin92] Finlay, I.W., *High Level Synthesis using Structural Input*, Ph.D. Thesis, Department of Electrical Engineering, University of Edinburgh, 1992.

[Garey79] Garey, M.R., and Johnson, D.S., *Computers and Intractibility : A Guide to the Theory of NP-completeness*, Freeman & Company, 1979.

[Gibbons87] Gibbons, A., *Algorithmic Graph Theory*, Cambridge University Press, 1987.

[Girczyc85] Girczyc, E.H., Buhr, R.J. and Knight, J.P., "Applicability of a subset of ADA for graph based hardware compilation," *IEEE Trans. CAD*, Vol. CAD-4, No.2, April 1985, pp. 134-142.

[Grant90a] Grant, D.M., and Denyer, P.B., "Memory, Control and Communications Synthesis for Scheduled Algorithms," in *Proc. 27th Design Automat. Conf.*, June 1990, pp. 162-168.

[Grant90b] Grant, P.M., "The DTI-Industry Sponsored Silicon Architectures Research Initiative", *IEE Electronics & Communications Engineering Journal*, Vol. 2 No. 3, June 1990.

[Goosse87] Goossens, G., Rabaey, J., Van de Walle, J. and De Man, H., "An Efficient Microcode Compiler for Custom DSP Processors," in *Proc. Int. Conf. on Comp. Aided Design*, November 1987, pp. 24-27.

[Hafer83] Hafer, L.J. and Parker, A.C., "A Formal Method for the Specification, Analysis and Design of Register Transfer Level Digital Logic," *IEEE Trans. CAD*, Vol. CAD-2, No.1, January 1983.

[Hashi71] Hashimoto, A. and Stevens, J., "Wire Routing by Optimizing Channel Assignment Within Large Apertures," in *Proc. 8th. Design Automat. Workshop*, 1971, pp. 155-169.

[Hilfin84] Hilfinger, P.N., *SILAGE : A Language for Signal Processing*,

University of California Technical Report, Berkeley, 1984.

[Hilfin85] Hilfinger, P.N., "A High Level Language and Silicon Compiler for Digital Signal Processing," in *Proc. Custom Integrated Circuits Conf.*, 1985, pp. 213-216.

[Holyer81] Holyer, I., "The NP-completeness of edge-colouring," *SIAM. J. Comput.*, Vol. 5, No. 4, 1981.

[Huang86] Huang, M.D., Romero, F. and Sangiovanni-Vincentelli, "An Efficient General Cooling Schedule for Simulated Annealing," in *Proc. Int. Conf. on Computer-Aided Design*, 1986, pp. 381-384.

[Huang90a] Huang, C.T., Hsu, Y.C. and Lin, Y.L., "Optimum and Heuristic Data Path Scheduling under Resource Constraints," in *Proc. 27th Design Automat. Conf.*, June 1990, pp. 65-70.

[Huang90b] Huang, C.Y., Chen, Y.S., Lin, Y.L. and Hsu, Y.C., "Data Path Allocation Based on Bipartite Weighted Matching," in *Proc. 27th Design Automat. Conf.*, June 1990, pp. 499-504.

[Hwang86] Hwang, K. and Briggs, F.A., *Computer Architecture and Parallel Processing*, McGraw-Hill Book Company, 1986.

[Johan78] Johannsen, D.L., "Silicon Compilation," *Caltech SSP Report*, California Institute of Technology, 1978.

[Johns76] Johnson, H.C., "Cliques of a Graph - Variations on the Bron - Kerbosch Algorithm," in *Int. Journal of Computer and Information Sciences*, Vol. 5, No. 3, 1976, pp. 209-238.

[Karp72] Karp, R.M., "Reducibility Among Combinatorial Problems," in *Complexity of Computer Computations*, Miller & Thatcher (eds), Plenum Press, 1972, pp. 85-103.

[Karp75] Karp, R.M., "On the Complexity of Combinatorial Problems," *Networks*, Vol. 5, 1975, pp. 45-68.

[Kirk83] Kirkpatrick, S., Gelatt, C. and Vecchi, M., "Optimisation by Simulated Annealing," *Science*, 220/4598, 1983, pp. 671-680.

[Kleine88] Kleine, U. and Noll, T.G., " Wave Digital Filters Using Carry-Save Arithmetic," in *Proc. ISCAS 88*, 1988, pp. 1757-1762.

[Kowal85] Kowalski, T.J., *An Artificial Intelligence Approach to VLSI Design*, Kluwer Academic Publishers, 1985.

[Kucukc89] Kucukakov, K. and Parker, A.C., "Data Path Design Tradeoffs using MABAL," in *Proc. 4th Int. Workshop on High-Level Synthesis*, October 1989.

[Kung85] Kung, S.Y., "VLSI Signal Processing," in *VLSI and Modern Signal Processing*, Kung, Whitehouse & Kailath *(eds)*, Prentice Hall, 1985, pp. 127-153.

[Kurdahi87] Kurdahi, F.J. and Parker, A.C., "REAL : A Program for Register Allocation," in *Proc. 24th Design Automat. Conf.*, June 1987, pp. 210-215.

[Lala85] Lala, P.K., *Fault Tolerant and Fault Testable Hardware Design*, Prentice Hall International, 1985.

[Law76] Lawley, E.L., "A note on the complexity of the chromatic number problem," *Information Processing Letters*, No.5, 1976, pp. 66-67.

[Law77] Lawson, S.S., "Computer Simulation and Implementation of a Wave Digital Filter," *I.E.E. Colloq. on Electronic Filters*, 1977.

[Law90] Lawson, S.S. and Summerfield, S., "The Design of Wave Digital Filters Using Fully Pipelined Bit-Level Systolic Arrays," *Journal of*

VLSI Signal Processing, Vol. 2, No. 1, September 1990, pp. 51-64.

[Lee89] Lee, J.H., Hsu, Y.C. and Lin, Y.L., " A New Integer Linear Programming Formulation for the Scheduling Problem in Data Path Synthesis," in *Proc. Int. Conf. on Computer Aided Design*, November1989, pp. 20-23.

[Lis88] Lis, J.S. and Gajski, D.D., "Synthesis from VHDL," in *Proc. IEEE Conf. Comp. Design*, 1988, pp. 378-381.

[LSI91] *1.0um Cell-Based Products Databook*, LSI Logic, February 1991.

[LSI93] *CW702 JPEG Core Technical Manual*, LSI Logic, 1993.

[Lun84] Lundy, M. and Mees, A., "Convergence of the Annealing Algorithm," in *Proc. Simulated Annealing Workshop*, Yorktown Heights, 1984.

[Marwed86] Marwedel, P., "A New Synthesis Algorithm for the MIMOLA Software System," in *Proc. 23rd Design Automat. Conf.*, 1986, pp. 271-277.

[Matt89] Matterne, L., Chong, D., McSweeney, B., and Woudsma, R., "A flexible high performance 2-D discrete cosine transform IC," in *Proc. Intl. Symp. Circuits and Systems*, 1989, pp. 618-621.

[Mealy54] Mealy, G.H., "A method for synthesising sequential circuits," *Bell Syst. Tech. J.*, Vol 34, 1955, pp. 1045-1079.

[Met53] Metropolis, N., Rosenbluth, A., Rosenbluth M., Teller, A. and Teller, E., "Equation of State Calculations by Fast Computing Machines," *Journal Chem. Phys.*, 21/6, 1087, 1953.

[McFarl86] McFarland, M.C., "BUD : Bottom-up Design of Digital Systems," in *Proc. 23rd Design Automat. Conf.*, July 1986, pp. 474-479.

[McFarl88] McFarland, S.J., Parker, A.C. and Camposano, R., "Tutorial on High-Level Synthesis," in *Proc. 25th Design Automat. Conf.*, July 1988, pp. 330-336.

[Mign90] Mignotte, A. and Saucier, G., "Matching Method for Concurrent Operator, Register and Multiplexer Allocation," *Synthesis Simulation Meeting and International Interchange*, 1990, pp. 215-222.

[Moore79] Moore, G.E., "Are We Really Ready for VLSI?," in *Proc. Caltech Conf. on Very Large Scale Integration*, 1979.

[Note88] Note, S., Catthoor, F., De Man, H., and Van Meergergen, J., "Hardwired Datapath Synthesis for High Speed DSP Systems with the CATHEDRAL-III Compilation Environment," in *Proc. Int. Workshop on Logic and Architecture Synthesis*, May, 1988.

[Orail86] Orailoglu, A. and Gajski, D.D., "Flow Graph Representation," in *Proc. 23rd Design Automat. Conf.*, July 1986, pp. 503-509.

[Ott84] Otten, R.H.J.M and van Ginneken, L.P.P.P, *Simulated Annealing: The Algorithm*, unpublished manuscript, 1984.

[Pangrle87] Pangrle, B.M. and Gajski, D.D., "SLICER : A State Synthesiser for Intelligent Silicon Compilation," in *Proc. IEEE Conf. Computer Design*, October 1987.

[Pangrle88] Pangrle, B.M., "Splicer: A Heuristic Approach to Connectivity Binding," in *Proc. 25th Design Automat. Conf.*, July 1988.

[Papadim82] Papadimitriou, C.H. and Steiglitz, K., *Combinatorial Optimisation : Algorithms and Complexity*, Prentice Hall, 1982.

[Park88] Park, N. and Parker, A.C., "SEHWA : A Software Package for the

Synthesis of Pipelines form Behavioural Descriptions", *IEEE Trans. CAD*, Vol. CAD-7, No.3, March 1988, pp. 356-370.

[Park89] Park, H., and Kurdahi, F.J., "Module Assignment and Interconnect Sharing in Register-Transfer Synthesis of Pipelined Data Paths," in *Proc. Int. Conf. on Computer Aided Design*, November 1989, pp. 16-19.

[Parker86] Parker, A.C., Pizarro, J.T. and Milnar, M.,"MAHA : A program for datapath synthesis", in *Proc. 23rd Design Automat. Conf.*, July1986, pp. 461-466.

[Paulin86] Paulin, P.G., Knight, J.P. and Girczyc, E.F., "HAL : A Multi-Paradigm Approach to Automatic Data Path Synthesis," in Proc. 23rd Design Automat. Conf., July 1986, pp. 263-270.

[Paulin89a] Paulin, P.G. and Knight, J.P., "Scheduling and Binding algorithms for High-Level Synthesis," in *Proc. 26th Design Automat. Conf.*, June 1989, pp. 1-6.

[Paulin89b] Paulin, P.G. and Knight, J.P., "Force Directed Scheduling for the Behavioral Synthesis of ASICs," *IEEE Trans. CAD*, Vol CAD-8, No.6, June 1989, pp. 661-680.

[Paulin89c] Paulin, P.G. and Knight, J.P., "Algorithms for High Level Synthesis," *IEEE Design and Test of Computers*, December 1989, pp. 18-31.

[Peng86] Peng, Z., "Synthesis of VLSI systems with the CAMAD design aid," in *Proc. 23rd Design Automat. Conf.*, 1986, pp. 278-284.

[Peng87] Peng, Z., *A Formal Methodology for Automated Synthesis of VLSI Systems*, Ph.D.Thesis, Department of Computer and Information Science, Linkoping University, Sweden, 1987.

[Petrie86] Petrie, N., *The Design and Implementation of Digital Wave Filter Adaptors*, Ph.D. Thesis, University of Edinburgh, 1986.

[Pot89] Potkonjak, M. and Rabaey, J., "Scheduling and Resource Allocation Algorithms for Hierarchichal Signal Flow Graphs'" in *Proc. 26th Design Automat. Conf.*, 1989, pp. 7-12.

[Rabaey88] Rabaey, J., De Man, H., Vanhoof, J., Goossens, G. and Catthoor, F., "CATHEDRAL II : A Synthesis System for Multiprocessor DSP Systems," in *Silicon Compilation*, Gajski *(ed)*, Addison Wesley Publishing Company, 1988, pp. 311-360.

[Raj90] Rajinder, J.S., Woods, R.F. and McCanny, J.V., "High Performace Systolic Two-Port Adaptor for Wave Digital Filtering Applications," in *Proc. ICASSP 90*, 1990.

[Reek84] Reekie, M., "Design and Implementation of Digital Wave Filters using Universal Adaptor Structures," in *Proc I.E.E.*, Pt. F, Vol 131, 984, pp. 615-622.

[Reif65] Reif, F., *Statistical and Thermal Physics*, McGraw-Hill, 1965.

[Ryder89] Ryder, M., *BABBLE Definition Version 1.2*, SARI Internal Technical Report SARI-034-B, February 1989.

[Saf90] Safir, A., and Zavidovique, B., "Towards a Global Solution to High Level Synthesis Problems," in *Proc. EDAC90*, March 1990, pp. 283-288.

[SARI89] *First Milestone Demonstrator*, SARI Internal Report, January 1989.

[Sasena89] Sasena, E.A., "Design Tradeoffs: Three State Drivers vs. Muxes," *LSI Logic Application Note*, LSI Logic Corporation, May 1989.

[Sechen86] Sechen, C. and Sangiovanni-Vincentelli, A., "Timberwolf3.2: a

new standard cell placement and global routing package," in *Proc. 23rd Design Automat. Conf.*, pp. 432-439, 1986.

[Sechen88] Sechen,C., *VLSI Placement and Global Routing Using Simulated Annealing*, Kluwer Academic Publishers, 1988.

[Seitz80] Seitz, C., "System Timing," in *Introduction to VLSI Systems*, Mead & Conway *(eds)*, Addison Wesley, 1980, pp. 218-262.

[Sey89] Seymour, L.P.H.K., *SLANG - The SARI Input Language*, SARI Internal Technical Report SARI-064-A, June 1989.

[SGS90] *Image Processing Databook*, SGS-Thomson, 1990.

[Soame82] Soame, T.A., "Bandwidth Compression of Images Using Transform Techniques", *GEC J. Sci. Tech.*, Vol. 48, No. 1, 1982, pp. 17-23.

[Stok88] Stok, L. and van den Born, R., "EASY : Multiprocessor Architecture Optimisation," in *Proc. Int. Workshop on Logic and Architecture Synthesis for Silicon Compilers*, McLellan *(ed)*, Grenoble, May 1988, pp. 313-328.

[Stok90] Stok, L., "Interconnect Optimisation during Data Path Allocation", in *Proc. EDAC90*, March 1990, pp. 141-145.

[Stok91] Stok, L., *Synthesis and Optimisation of Architectures for Digital Systems*, Ph.D. Thesis, Eindhoven University of Technology, 1991.

[Thomas83] Thomas, D.E., Hitchcock, C.Y., Kowalski, T.J., Rajan, J.V. and Walker, R.A., "Automatic data path synthesis," *IEEE Computer*, December 1983, pp. 59-70.

[Thomas87] Thomas, D.E., Blackburn, R.L. and Rajan, J.V., "Linking the Behavioral and Structural Domains of Representation for Digital Systems Design," *IEEE Trans. CAD*, Vol. CAD-6, No.1, January

1987, pp. 103-110.

[Thomas88] Thomas, D.E., Dirkes, E. M., Walker, R. A., Rajan, J.V., Nestor, J.A. and Blackburn, R.L. "The System Architect's Workbench," in *Proc. 25th Design Automat. Conf.*, 1988, pp. 337-343.

[Top89] Topham, N.P. and Ibbet, R.N., *The Architecture of High Performance Computers II*, MacMillan Publishers Ltd., 1989.

[Tseng83] Tseng, C.J. and Siewiorek, D.P., "FACET: A procedure for automated synthesis of digital systems," in *Proc. 20th Design Automat. Conf., 1983*, pp. 490-496.

[Tseng86] Tseng, C.J. and Siewiorek, D.P., "Automated Synthesis of Data Paths in Digital Systems," *IEEE Trans. CAD*, Vol. CAD-5, No.3, July 1986, pp. 379-395.

[Ullman75] Ullman, J.D., "NP-complete Scheduling Problems," *JCSS*, Vol 10, 1975, pp. 384-393.

[Vanhoof87] Vanhoof, J., Rabaey, J. and De Man, H., "A Knowledge-Based CAD System for Synthesis of Multiprocessor Digital Signal Processing Chips," in *Proc. VLSI87*, 1987.

[Vizing64] Vizing, V.G., "On an estimate of the chromatic class of a p-graph," *Diskret. Analiz.*, No.3, 1964, pp. 25-30.

[Wintz72] Wintz, P.A., "Transform Picture Coding", *Proc. IEEE*, Vol 60. No. 7, pp. 809-819, 1972.

[Yan89] Yan, M. and McCanny, J.V., "Architectures for computing the 2D-DCT," in *Systolic Array Processors*, McCanny, McWhirter & Swartzlander *(eds)*, Prentice Hall, 1989, pp. 411-420.

Appendix A Differential Equation Datapath

This Appendix presents the datapath netlist for the naive mapping between behaviour and structure presented in section 2.5. For reference, the datapath schematic is given again below:

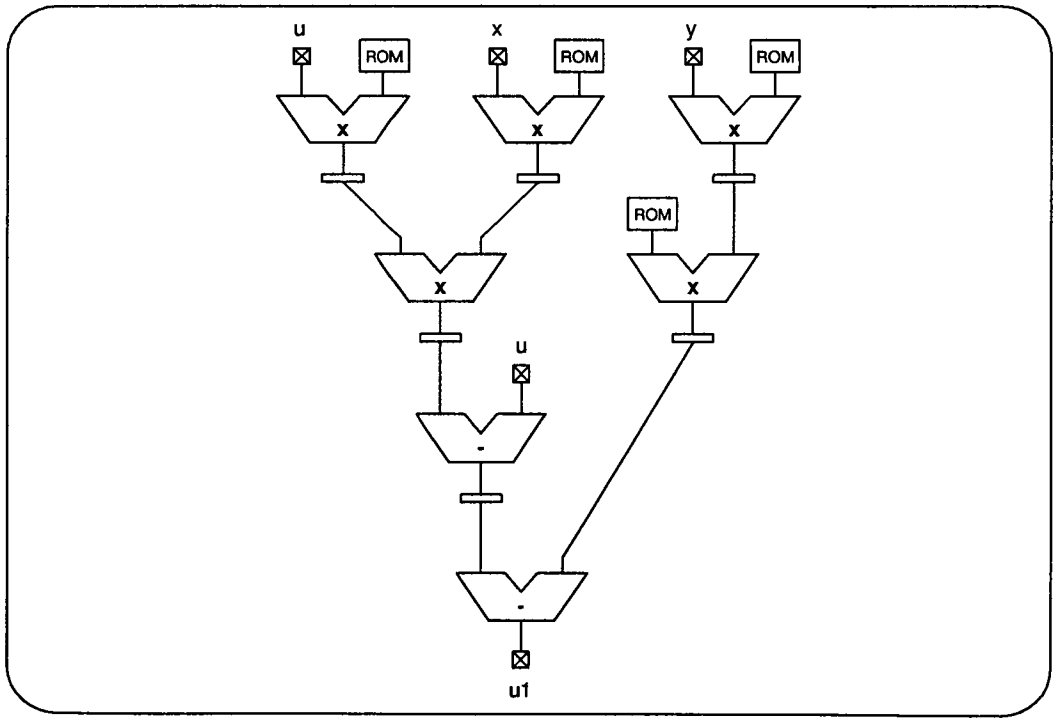


Figure A.1 Datapath produced by naive mapping axioms.

```
datapath DIFF_EQ begin

  processor MULF_1 begin
      attributes non-pipe, latency 1, area 50;
      type mul;
      commutative true;
      ports A source U;
            B source dx.1;
            Z sink s1.in;
  end processor MULF_1;

  processor MULF_2 begin
      attributes non-pipe, latency 1, area 50;
      type mul;
      commutative true;
      ports A source c5;
            B source x;
            Z sink s2.in;
  end processor MULF_2;

  processor MULF_4 begin
      attributes non-pipe, latency 1, area 50;
      type mul;
      commutative true;
      ports A source c3;
            B source y;
            Z sink s4.in;
  end processor MULF_4;

  processor MULF_3 begin
      attributes non-pipe, latency 1, area 50;
      type mul;
      commutative true;
      ports A source s1.out;
            B source s2.out;
            Z sink s3.in;
  end processor MULF_3;

  processor MULF_6 begin
      attributes non-pipe, latency 1, area 50;
      type mul;
      commutative true;
      ports A source s4.out;
            B source dx.2;
            Z sink s6.in;
  end processor MULF_6;

  processor SUBF_5 begin
      attributes area 30;
      type sub;
      commutative false;
      ports A source u;
            B source s3.out;
            Z sink s5.in;
  end processor SUBF_5;

  processor SUBF_7 begin
      attributes area 30;
      type sub;
```

```
    commutative false;
    ports A source s5.out;
          B source s6.out;
          Z sink u1;
end processor SUBF_7;

memory REGF_1 begin
    attributes area 35
    type register;
    ports  A source s1.in;
           Z sink s1.out;
    signals s1;
end memory REGF_1;

memory REGF_2 begin
    attributes area 35
    type register;
    ports  A source s2.in;
           Z sink s2.out;
    signals s2;
end memory REGF_2;

memory REGF_3 begin
    attributes area 35
    type register;
    ports  A source s4.in;
           Z sink s4.out;
    signals s4;
end memory REGF_3;

memory REGF_4 begin
    attributes area 35
    type register;
    ports  A source s3.in;
           Z sink s3.out;
    signals s3;
end memory REGF_4;

memory REGF_5 begin
    attributes area 35
    type register;
    ports  A source s6.in;
           Z sink s6.out;
    signals s6;
end memory REGF_5;

memory REGF_6 begin
    attributes area 35
    type register;
    ports  A source s5.in;
           Z sink s5.out;
    signals s5;
end memory REGF_6;

memory ROMF_1 begin
    attributes area 30
    type ROM;
    locations 1;
    ports  Z sink dx.1;
end memory ROMF_1;

memory ROMF_2 begin
    attributes area 30
    type ROM;
    locations 1;
    ports  Z sink c5;
end memory ROMF_2;

memory ROMF_3 begin
    attributes area 30
    type ROM;
    locations 1;
    ports  Z sink c3;
end memory ROMF_3;

memory ROMF_4 begin
    attributes area 30
    type ROM;
    locations 1;
    ports  Z sink dx.2;
end memory ROMF_1;

net u.1 begin
    type wire;
    source IOF_1.Z;
    sink MULF_1.A;
end net u.1;

net dx.1 begin
    type wire;
    source ROMF_1.Z;
    sink MULF_1.A;
end net dx.1;

net c5 begin
    type wire;
    source ROMF_2.Z;
    sink MULF_2.A;
end net c5;

net x begin
    type wire;
    source IOF_2.Z;
    sink MULF_2.B;
end net x;

net c3 begin
    type wire;
    source ROMF_3.Z;
    sink MULF_4.A;
end net c3;

net y begin
    type wire;
    source IOF_3;
    sink MULF_4;
end net y;

net s1.in begin
    type wire;
```

```
    source MULF_1.Z;
    sink REGF_1.A;
end net s1.in;

net s1.in begin
    type wire;
    source REGF_1.Z;
    sink MULF_3.A;
end net s1.out;

net s2.in begin
    type wire;
    source MULF_2.Z;
    sink REGF_2.A;
end net s2.in;

net s2.out begin
    type wire;
    source REGF_2.Z;
    sink MULF_3.B;
end net s2.out;

net dx.2 begin
    type wire;
    source ROMF_4.Z;
    sink MULF_6.A;
end net dx.2;

net s4.in begin
    type wire;
    source MULF_4.Z;
    sink REGF_3.A;
end net s4.in;

net s4.out begin
    type wire;
    source REGF_3.Z;
    sink MULF_6.B;
end net s4.out;

net u.2 begin
    type wire;
    source IOF_4;
    sink SUBF_5.A;
end net u.1;

net s3.in begin
    type wire;
    source MULF_3.Z;
    sink REGF_4.A;
end net s3.in;

net s3.out begin
    type wire;
    source REGF_4.Z;
    sink SUBF_5.B;
end net s3.out;

net s6.in begin
    type wire;
    source MULF_6.Z;
    sink REGF_5.A;
end net s6.in;

net s6.out begin
    type wire;
    source REGF_5.Z;
    sink SUBF_7.A;
end net s6.out;

net s5.in begin
    type wire;
    source SUBF_5.Z;
    sink REGF_6.A;
end net s5.in;

net s5.out begin
    type wire;
    source REGF_6.Z;
    sink SUBF_7.B;
end net s5.out;

net u1 begin
    type wire;
    source SUBF_7.Z;
    sink IOF_5.A;
end net u1;

I/O IOF_1 begin
    type in;
    ports Z sink u.1;
end I/O IOF_1;

I/O IOF_2 begin
    type in;
    ports Z sink x;
end I/O IOF_2;

I/O IOF_3 begin
    type in;
    ports Z sink y;
end I/O IOF_3;

I/O IOF_4 begin
    type in;
    ports Z sink u.2;
end I/O IOF_4;

I/O IOF_4 begin
    type out;
    ports A source SUBF_7.Z;
end I/O IOF_5;

end datapath DIFF_EQ;
```

Appendix B SAVAGE Optimisation Move Sets

This appendix contains the ADA code for the SAVAGE state generation move sets presented in section 5.3.

B.1 R-t Optimisation Code

This section presents the ADA package containing the optimisation moves operating in *R-t* space. The first function presented verifies the availability of execution times on a selected processor.

```
package body RT_OPTIMISE is

function SLOT_AVAILABLE_AT( C_STEP:INTEGER;
                            PROC:PROCESSOR_PTR) return BOOLEAN is
  SEARCH_LIST : ALLOCATION_PTR;
  ITEM : ALLOCATED_ITEM;
  FLAG : BOOLEAN;

begin
  SEARCH_LIST := PROC.ALLOCATION; FLAG := TRUE;
  -- Get the first item in the list
  ITEM := HEAD(SEARCH_LIST);
  -- Execute this loop until we reach the end of the allocation list
  while not(ITEM.NEXT = null) loop
     if ITEM.EXECUTION_TIME = C_STEP then
        FLAG := FALSE;
     end if;
```

```
    -- Next item on the list
    ITEM := ITEM.NEXT;
  end loop;
  return FLAG;
end SLOT_AVAILABLE_AT;
```

The UNARY_STEP function aims to schedule a selected operation on the processor that it is currently allocated to. The procedure checks that execution times at control steps *c+1* and *c-1* are available. A random variable selects the new schedule.

```
procedure UNARY_STEP (OP:in out OP_PTR; PROC:in PROCESSOR_PTR) is

  type STATE is (INC,DEC,BOTH,NEITHER);
  TEMP1,TEMP2 : OPERATION;
  FLAG : STATE;
  R : INTEGER;

begin
  TEMP1 := OP; TEMP2 := OP; FLAG := NEITHER;
  if SLOT_AVAILABLE_AT(OP.EXECUTION_TIME+1,PROC) then
     FLAG := DEC;
  if ((SLOT_AVAILABLE_AT(OP.EXECUTION_TIME-1,PROC) and (FLAG=DEC)) then
     FLAG := BOTH;
  else
     FLAG := INC;
  end if;
  case FLAG is
        when NEITHER => null;
        when DEC => begin
                    R := RANDOM(1);
                    case R is
                         when 0 => OP.EXECUTION_TIME :=
                                   OP.EXECUTION_TIME + 1;
                         when 1 => null;
                       end case;
                    end;
        when INC =>  begin
                       R := RANDOM(1);
                       case R is
                         when 0 => OP.EXECUTION_TIME :=
                                   OP.EXECUTION_TIME - 1;
                         when 1 => null;
                       end case;
                    end;
        when BOTH => begin
                       R := RANDOM(2);
                       case R is
                         when 0 => OP.EXECUTION_TIME :=
                                   OP.EXECUTION_TIME + 1;
                         when 1 => OP.EXECUTION_TIME :=
                                   OP.EXECUTION_TIME - 1;
                         when 2 => null;
                       end case;
                    end;
  end case;
end UNARY_STEP;
```

The following function aims to schedule the selected operation on a processor able to support the operation.type and having execution times available at control steps *c-1*, *c* and *c+1*. The UNARY_STEP function is used to perform the scheduling operation.

```
procedure VALID_PROCESSOR (OP:in out OP_PTR; RT:in out RT_PTR) is

  PROCESSORS, VALID_PROCESSORS : RT_PTR;
  ITEM : PROCESSOR_PTR;
  R,I : INTEGER;

begin
  PROCESSORS := RT; VALID_PROCESSORS := null;
  ITEM := HEAD(PROCESSORS);
  -- Search through all candidate processors
  while not (ITEM.NEXT = null) loop
     if ((SLOT_AVAILABLE_AT(OP.EXECUTION_TIME+1,ITEM)) and
        (IS_MEMBER(OP.TYPE,ITEM.TYPE_LIST)))then
           ADD(ITEM,VALID_PROCESSORS);
     end if;
     if ((SLOT_AVAILABLE_AT(OP.EXECUTION_TIME,ITEM)) and
        (IS_MEMBER(OP.TYPE,ITEM.TYPE_LIST)))then
           ADD(ITEM,VALID_PROCESSORS);
     end if;
     if ((SLOT_AVAILABLE_AT(OP.EXECUTION_TIME-1,ITEM)) and
        (IS_MEMBER(OP.TYPE,ITEM.TYPE_LIST)))then
           ADD(ITEM,VALID_PROCESSORS);
     end if;
     ITEM := ITEM.NEXT;
  end loop;
  --Select a valid processor at random
  R := RANDOM (LENGTH(VALID_PROCESSORS));
  ITEM := HEAD(VALID_PROCESSORS);
  for I in 1 .. R loop
     ITEM := ITEM.NEXT;
  end loop;
  -- Add the operation to the new processor's allocation list
  ADD(OP,ITEM.ALLOCATION);
  -- Remove the operation from its currently allocated processor
  -- Note : Processor allocation pointed to from data flow graph
  REMOVE(OP,OP.ALLOCATED_TO);
  -- Schedule the operation on the newly allocated processor
  UNARY_STEP(OP,ITEM);

end VALID_PROCESSOR;
```

CREATE_PROCESSOR adds a processor supporting operations of type **operation.type** to *R-t* space, and allocates the selected operation to the new processor. Again, UNARY_STEP is used to schedule the operation.

```
procedure CREATE_PROCESSOR (OP:in out OP_PTR; RT: in out RT_PTR) is

  ITEM : PROCESSOR_PTR;
```

```
begin
  -- Create a processor with the same type as that of the selected
  -- operation
  ITEM := new PROCESSOR;
  ADD(OP.TYPE,ITEM.OPERATION_TYPE_LIST);
  -- Add the operation to the new processor's allocation list
  ADD(OP,ITEM.ALLOCATION);
  -- Remove the operation from its currently allocated processor
  -- Note : Processor allocation pointed to from data flow graph
  REMOVE(OP,OP.ALLOCATED_TO);
  -- Schedule the operation on the newly allocated processor
  UNARY_STEP(OP,ITEM);
  -- Add the newly created processor to R-t space
  ADD(ITEM,RT);

end CREATE_PROCESSOR;
```

The following procedure, FUNCTION_MERGE, searches for candidate processors to become ALU units. From the candidate processors, one is selected at random, and the processor's operation_type_list is updated. The selected operation is allocated to the processor, and the UNARY_STEP function schedules the operation.

```
procedure FUNCTION_MERGE (OP:in out OP_PTR; RT:in out RT_PTR) is

  PROCESSORS, VALID_PROCESSORS : RT_PTR;
  ITEM : PROCESSOR_PTR;
  R,I : INTEGER;

begin
  PROCESSORS := RT; VALID_PROCESSORS := null;
  ITEM := HEAD(PROCESSORS);
  -- Search through all candidate processors
  while not (ITEM.NEXT = null) loop
      if ((SLOT_AVAILABLE_AT(OP.EXECUTION_TIME+1,ITEM)) and
          (not(IS_MEMBER(OP.TYPE,ITEM.TYPE_LIST))))then
             ADD(ITEM,VALID_PROCESSORS);
      end if;
      if ((SLOT_AVAILABLE_AT(OP.EXECUTION_TIME,ITEM)) and
          (not(IS_MEMBER(OP.TYPE,ITEM.TYPE_LIST))))then
             ADD(ITEM,VALID_PROCESSORS);
      end if;
      if ((SLOT_AVAILABLE_AT(OP.EXECUTION_TIME-1,ITEM)) and
          (not(IS_MEMBER(OP.TYPE,ITEM.TYPE_LIST))))then
             ADD(ITEM,VALID_PROCESSORS);
      end if;
      ITEM := ITEM.NEXT;
  end loop;
  --Select a valid processor at random
  R := RANDOM (LENGTH(VALID_PROCESSORS));
  ITEM := HEAD(VALID_PROCESSORS);
  for I in 1 .. R loop
      ITEM := ITEM.NEXT;
  end loop;
  -- Add the operation type to the processor type list
  ADD(OP.TYPE,ITEM.OPERATION_TYPE_LIST);
```

```
  -- Add the operation to the processor's allocation list
  ADD(OP,ITEM.ALLOCATION);
  -- Remove the operation from its currently allocated processor
  -- Note : Processor allocation pointed to from data flow graph
  REMOVE(OP,OP.ALLOCATED_TO);
  -- Schedule the operation on the newly allocated processor
  UNARY_STEP(OP,ITEM);

end FUNCTION_MERGE;

end RT_OPTIMISE;
```

B.2 M-t Optimisation Code

The CAN_STORE function checks the signal lifetime over all signals lifetimes allocated to the register by comparing the signal production and consumption times of all allocated signals to the selected signal. The predicates PROD() and CONS() return the production time and consumption time of the signal respectively. This signal lifetime evaluation procedure is illustrated in figure B.1.

```
package body MT_OPTIMISE is

function CAN_STORE (S : SIG_PTR; M : M_PTR) return BOOLEAN is

  SEARCH_LIST : SIGNAL_LIST_PTR;
  ITEM : SIG_PTR;
  FLAG : BOOLEAN;

begin

  SEARCH_LIST := M.SIGNAL_LIST; FLAG := TRUE;
  -- Search through signals allocated to the memory component
  ITEM := HEAD(SEARCH_LIST);
  -- Ensure that signal lifetimes do not overlap
  while not (ITEM.NEXT = null) loop
     -- Signal lifetime overlap case (i)
     if ((PROD(S) <= PROD(ITEM)) and (CONS(S) >= PROD(ITEM))) then
        FLAG := false;
     end if;
     -- Signal lifetime overlap case (ii)
     if ((PROD(S) >= PROD(ITEM)) and (PROD(S) <= CONS(ITEM))) then
        FLAG := false;
     end if;
     -- Signal lifetime overlap case (iii)
     if ((PROD(S) >= PROD(ITEM)) and (CONS(S) <= CONS(ITEM))) then
        FLAG := false;
     end if;
     -- Signal lifetime overlap case (iv)
     if ((PROD(S) <= PROD(ITEM)) and (CONS(S) >= CONS(ITEM)) then
        FLAG := false;
     end if;
  end loop;
```

```
  return FLAG;
end CAN_STORE;
```

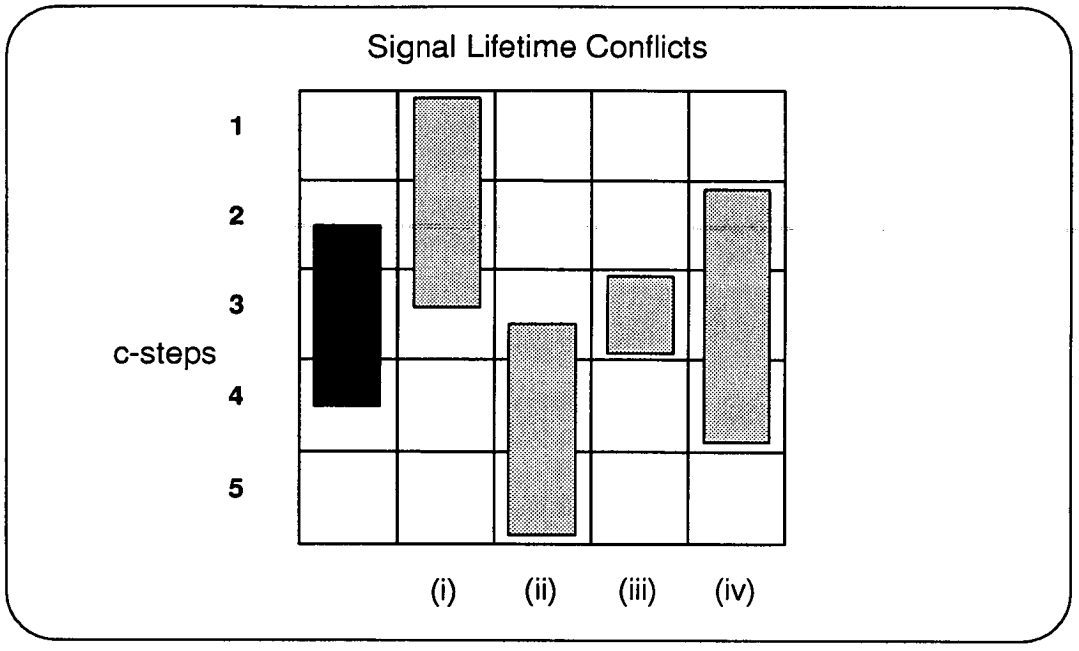


Figure B.1 Signal lifetime conflicts.

The following function traverses *M-t* space looking for registers capable of storing the selected signal. One is selected at random from the generated subset, and the signal reallocated. No error trapping is required, as there is always one register capable of storing the signal, the allocated register on entry to the procedure.

```
procedure SIGNAL_MERGE(SIG:in out SIG_PTR; MT:in out MT_PTR) is

begin
  REMOVE(SIG,SIG.ALLOCATED_TO);
  REGISTERS := HEAD(MT); ITEM := HEAD(REGISTERS); VALID_REGISTERS := null;
  -- Search through M-t space for eligible registers
  while not(ITEM.NEXT=null) loop
     if CAN_STORE(SIG,ITEM) then ADD(ITEM,VALID_REGISTERS);
  end loop;
  -- Select one at random
  R := RANDOM(LENGTH(VALID_REGISTERS)):
  ITEM := HEAD(VALID_REGISTERS);
  for I in 1 .. R loop
     ITEM := ITEM.NEXT;
  end loop;
  -- Reallocate the signal
  ADD(SIG,ITEM.SIGNAL_LIST);
end SIGNAL_MERGE;
```

CREATE_MEMORY adds a register to *M-t* space, and allocates the selected signal to the new memory.

```
procedure CREATE_MEMORY (SIG:in out SIG_PTR; MT: in out RT_PTR) is

  ITEM : MEMORY_PTR;

begin
  -- Create a register
  ITEM := new MEMORY;
  ITEM.TYPE := REGISTER;
  -- Add the signal to the register's allocation list
  ADD(SIG,ITEM.SIGNAL_LIST);
  ADD(OP.TYPE,ITEM.OPERATION_TYPE_LIST);
  -- Add the newly created register to M-t space
  ADD(ITEM,MT);

end CREATE_PROCESSOR;

end MT_OPTIMISE;
```

B.3 P-c Optimisation Code

The PORT_SWAP procedure locates the processor executing the selected operation. By traversing *P-c* space, the procedure locates the processor input ports. Both output port lists are searched for the signals bringing the operations input data. Once located, the signals are transposed on the lists.

```
package body PC_OPTIMISE is

procedure PORT_SWAP (OP:in out OP_PTR; PC:in out PC_PTR) is

  THE_PROCESSOR : PROCESSOR_PTR;
  PORT_LIST_A,PORT_LIST_B : PORT_LIST_PTR;
  ITEM1,ITEM2 : PORT_PTR;

begin
  -- Use the link into R-t space to access the allocated processor
  THE_PROCESSOR := OP.ALLOCATED_TO;
  -- Access P-c space to get the port lists
  PORT_LIST_A := FIND(THE_PROCESSOR.A,PC);
  PORT_LIST_B := FIND(THE_PROCESSOR.B,PC);
  -- Find the output ports associated with the selected operation
  ITEM1 := HEAD(PORT_LIST_A);
  while not (ITEM1.NEXT = null) loop
      exit when ((ITEM1.SIGNAL = OP.A) or (ITEM1.SIGNAL = OP.B));
      ITEM1 := ITEM1.NEXT;
  end loop;
  ITEM2 := HEAD(PORT_LIST_B);
  while not (ITEM2.NEXT = null) loop
      exit when ((ITEM2.SIGNAL = OP.A) or (ITEM2.SIGNAL = OP.B));
      ITEM2 := ITEM2.NEXT;
  end loop;
  -- Swap the input port references
  ADD(ITEM1,PORT_LIST_B);
  REMOVE(ITEM1,PORT_LIST_A);
```

```
  ADD(ITEM2,PORT_LIST_A);
  REMOVE(ITEM2,PORT_LIST_B);

end PORT_SWAP;
```

The following procedure selects a pair of output ports from an input port selected at random from *P-c* space (procedure SELECT_OUTPUT_PORTS). The net types implementing the communications function between the ports are then updated according to the rules shown in figure 5.14.

```
procedure NET_MERGE (PC:in out PC_PTR; NETS:in out NET_LIST_PTR;
                     THRESH : in INTEGER) is

  PORT1,PORT2 : PORT_PTR;
  ITEM,NET1,NET2 : NET_PTR;

begin
  SELECT_OUTPUT_PORTS(PC,PORT1,PORT2);
  NET1 := PORT1.NET; NET2 := PORT2.NET;
  -- Both ports connected with the same wire - register file
  if ((NET1.TYPE=WIRE) and (NET1=NET2)) then
     null
  -- Different wires : merge into a multiplexer
  elsif ((NET1.TYPE=WIRE) and (NET2.TYPE=WIRE) and (NET1/=NET2))then
     begin
        ITEM := new NET;
        ITEM.TYPE := MUX; ITEM.CARDINALITY := 2;
        REMOVE(PORT1.NET,NETS); REMOVE(PORT2.NET,NETS);
        PORT1.NET := ITEM; PORT2.NET := ITEM;
        ADD(ITEM,NETS);
     end
  -- One multiplexer, one wire : merge wire into multiplexer
  elsif ((NET1.TYPE=WIRE) and (NET2.TYPE=MUX)) then
     begin
        REMOVE(PORT1.NET,NETS);
        PORT1.NET := PORT2.NET;
     end
  -- ditto
  elsif ((NET1.TYPE=MUX) and (NET2.TYPE=WIRE)) then
     begin
        REMOVE(PORT2.NET,NETS);
        PORT2.NET := PORT1.NET;
     end
  -- Two separate multiplexers : merge into a single multiplexer or bus
  elsif ((NET1.TYPE=MUX) and (NET2.TYPE=MUX) and (NET1/=NET2) and
        ((NET1.CARDINALITY + NET2.CARDINALITY) > THRESH))then
     begin
        R := RANDOM(1);
        case R is
           when 0 => begin
                        ITEM := new NET;
                        ITEM.TYPE := MUX;
                        ITEM.CARDINALITY := NET1.CARDINALITY +
                                            NET2.CARDINALITY;
                        REMOVE(PORT1.NET,NETS); REMOVE(PORT2.NET,NETS);
```

```
                        PORT1.NET := ITEM; PORT2.NET := ITEM;
                        ADD(ITEM,NETS);
                      end;
            when 1 => begin
                        ITEM := new NET;
                        ITEM.TYPE := BUS;
                        ITEM.CARDINALITY := NET1.CARDINALITY +
                                            NET2.CARDINALITY;
                        REMOVE(PORT1.NET,NETS); REMOVE(PORT2.NET,NETS);
                        PORT1.NET := ITEM; PORT2.NET := ITEM;
                        ADD(ITEM,NETS);
                      end;
        end case;
    end
  end if;

end NET_MERGE;
```

Similarly, REGISTER_MERGE selects a pair of registers from an input port selected at random from *P-c* space (procedure SELECT_VALID_MERGE_PORTS), and *M-t* space is updated according to the merge functions shown in figure 5.15.

```
procedure REGISTER_MERGE (PC:in out PC_PTR; MT: in out MT_PTR) is

  PORT1,PORT2 : PORT_PTR;
  M1, M2 : MEM_PTR;

begin
  -- Get memory components for merging
  SELECT_VALID_MERGE_PORTS(PC,PORT1,PORT2);
  -- Access the M-t space
  M1 := FIND(PORT1.Z,MT); M2 := FIND(PORT2.Z,MT);
  -- 1. Both memory components belong to the same register file
  if ((M1.TYPE=FILE) and (M2.TYPE=FILE) and (M1.ID=M2.ID)) then
     null;
  -- 2. One register, one file
  elsif ((M1.TYPE=FILE) and (M2.TYPE=REGISTER)) then begin
     M2.TYPE := FILE; M2.ID := M1.ID;
     M1.CARDINALITY := M1.CARDINALITY + 1;
  end;
  -- 3. Ditto
  elsif ((M1.TYPE=REGISTER) and (M2.TYPE=FILE)) then begin
     M1.TYPE := FILE; M1.ID := M2.ID;
     M2.CARDINALITY := M2.CARDINALITY + 1;
  end;
  -- 4. Two separate register files
  elsif ((M1.TYPE=FILE) and (M2.TYPE=FILE) and (M1.ID/=M2.ID)) then
  begin
     M2.CARDINALITY := M2.CARDINALITY + M1.CARDINALITY;
     M1.ID := M2.ID; M1.CARDINALITY := M2.CARDINALITY;
  end;
  end if;
end REGISTER_MERGE;

end PC_OPTIMISE;
```

Appendix C Publications

The following papers were published during the course of the research described in this thesis, and are included in this appendix.

[1] Neil, J.P. and Denyer, P.B., "Synthesis By Simulated Annealing", *IEE Colloquium Digest 1989/125*, November 1989, pp.9/1-9/4.

[2] Neil, J.P. and Denyer, P.B., "SAVAGE : A Simulated-Annealing based VLSI Architecture Generator", *IEEE Workshop on Genetic Algorithms, Simulated Annealing and Neural Networks applied to Signal and Image Processing*, May 1990.

[3] Neil, J.P. and Denyer, P.B., "Exploring Design Space using SAVAGE : A Simulated Annealing based VLSI Architecture GEnerator", in *Proc. 33rd Midwest Symposium on Circuits and Systems*, Calgary, 1990

[4] Finlay, I.W., Neil, J.P. and Denyer, P.B., "Filter Synthesis using Behavioural Design Tools", in *Proc. 16th European Solid State Circuits Conference*, Grenoble, September 1990.

[5] Neil, J.P. and Denyer, P.B., "Simulated Annealing based synthesis of Fast Discrete Cosine Transform Blocks", in *Algorithmic and Knowledge-based CAD for VLSI*, Russell & Taylor (Ed.), Peter Perigrinus, 1991.

Synthesis by Simulated Annealing

J.P. Neil and P.B. Denyer

1. Introduction

This paper summarises work proceeding towards the development of a suite of algorithmic optimisation tools intended to operate in a directed silicon compilation environment (SAGE [1]). The tools are controlled by the stochastic computational technique known as *simulated annealing*. The paper contains a description of the scheduling and allocation problem, together with a description of the simulated annealing algorithm. There then follows a brief discussion of results obtained thus far together with some indications of future work.

2. The Scheduling and Allocation Problem

Within a silicon compiler, the aim of the scheduling task is to minimise the amount of time necessary to complete the program, subject to some limit on available hardware resources, while the allocation task deals with the minimisation of the amount of hardware resource needed.

Previous systems have addressed this problem in three general ways :

- The most straightforward technique is to set some (or no) limit on the functional units available, and then to produce a schedule.[2,3,4]
- Functional unit allocation can be done first, then a schedule can be derived. The BUD system [5] partitions operations into clusters using a metric which takes into account *potential* parallelism. Functional units are then assigned to these clusters, and then scheduling takes place.
- Scheduling and allocation can be performed simultaneously. The HAL system [6,7,8,9] uses force-directed scheduling together with a feedback loop to allow an iterative solution to be developed.

It is widely felt that neither of the first two techniques fully address the scheduling and allocation problem. Only the more complex solution offered by interrelated allocation and scheduling can offer a close to optimum solution without compromising either the schedule or resource allocations.

3. An Introduction to Simulated Annealing

Simulated Annealing is a stochastic computational technique derived from statistical mechanics for finding near globally minimum cost solutions to large optimisation problems. Kirkpatrick, Gelatt and Vecchi [10] were the first to propose and demonstrate the application of simulation techniques from statistical physics to problems of combinatorial optimisation, specifically to the problems of wire routing and component placement in VLSI design.

The following function gives the general structure of the class of algorithms called *probabilis-*

The authors are with the Silicon Architectures Research Initiative, Department of Electrical Engineering, University of Edinburgh, The Kings Buildings, Edinburgh, EH9 3JL, Scotland, U.K..

tic hill-climbing algorithms, of which simulated annealing is a special case. This class was proposed by Romeo and Sangiovanini-Vincentelli, where a number of different algorithms with the same structure were introduced [11].

```
sim_anneal (i0, T0)
{
   T = T0;
   i = i0;
   while (stopping criteria is not satisfied){
      while (inner loop criteria is not satisfied){
         j = generate(i);
         if (accept(c(j),c(i), T){
            i = j;
         }
      }
      T = update(T);
   }
}
```

where T_0 is the initial temperature of the system, i_0 is the initial network configuration and the function c() returns an assessment of the cost of a particular network. The acceptance of a new state, j, is determined by accept(), whose structure is :

```
accept (c(j),c(i),T)
{
   Δc = c(j) - c(i);
   y = f(Δc, T);
   r = random(0,1); /** random number between 0 and 1
   if ( r < y ){
      return (TRUE)
   }else{
      return (FALSE)
   }
}
```

Kirkpatrick, Gelatt and Vecchi use the following formulation of the acceptance function f() :

$$f(\Delta c, T) = \min \{1.0, \exp((-\Delta c)/T)\}$$

In particular, the implementation of f() uses a Boltzmann-like factor. Note that when Δc is negative or zero, that is, when the cost of the new state is less than or equal to the cost of the current state, then the new state is always accepted. On the other hand, when Δc is positive, the acceptance probabilities are distributed according to the Boltzmann factor.

Kirkpatrick, Gelatt and Vecchi use the following update function, update() :

$$T_n = T.\alpha(T), \quad 0 < \alpha(T) < 1$$

where T_n is the new value and T is the current value of the temperature parameter.

The inner loop criterion is implemented by specifying the number of new states generated for each stage of the annealing process.

The stopping criterion is implemented by recording the value of the cost function at the end of each stage of the annealing process. It is satisfied when the value of the cost function has remained unchanged at the end of three consecutive stages.

3.1 Simulated annealing and the scheduling and allocation problem

Devedas and Newton [12,13] report the development of a simulated annealing based scheduler when applied to the problem of optimisation of microcode instruction placement in data path synthesis. Their approach is heavily based on the success of simulated annealing as an optimisation technique for placement and routing packages, and consequently suffers from having to include a stage in the optimisation process where the problem is transformed into a suitable microinstruction format, ready for 'placement'.

The approach reported here requires no intermediate algorithmic transformations and operates on data flow information generated by the input compiler to the synthesis system.

4. Results

The work reported in this paper is primarily concerned with the development of a scheduler operating on a user-specified resource budget. The software structure is such that incremental development will allow the development of an allocation tool which will operate in concert with the scheduler in an iterative manner.

The scheduler has been tested on two significant examples, namely the 1-dimensional 8-point Fast Discrete Cosine Transform and a 5th Order Digital Elliptic Wave Filter.

Preliminary results indicate that the core of the scheduler, which resides in the generate function of the simulated annealing function has an incomplete move set. Because the annealing process is essentially random, nodes on the critical path of the data flow graph cannot be guaranteed earliest execution. A solution to this is to perform a critical path analysis, and optimise the resulting partitions separately. Current work is concerned with this development.

5. Future Directions

Results obtained thus far indicate that simulated annealing offers considerable potential as a controlling mechanism for any optimisation process, and it is intended to develop a library of scheduling, allocation and analysis tools corresponding to a wide range of system architectures. Selection of appropriate schedules and resource allocations will allow the designer to map the initial problem into a number of distinct target architectures very rapidly, and also experiment with non-intuitive combinations.

Selection of appropriate combinations of optimisation and analysis depends on 'meta-information', that is, information about architectures that cannot be generated directly from an algorithmic description of the problem. Consequently, it is intended to aid the designer in his/her selection with an expert system, able to advise on appropriate combinations.

Acknowledgements

This work was carried out as part of the Silicon Architectures Research Initiative. The use of facilities and resources is gratefully acknowledged.

The work reported here is supported by the Science and Engineering Research Council.

References

[1] Denyer, P.B., "SAGE Design Methodology", SARI Internal Technical Report SARI-035-D, March 1989.

[2] Tseng, C. and Sewiorek, D.P., "Automated Synthesis of Data Paths in Digital Systems," *IEEE Trans. Computer-Aided Design*, Vol. CAD-5, July1985, pp. 379-395.

[3] Trickey, H., "Flamel : A High-Level Hardware Compiler," *IEEE Trans. Computer-Aided Design*, Vol. CAD-6, March 1987, pp. 259-269.

[4] Mallon, D., "SAGE Operation Power Tools," SARI Internal Technical Report SARI-014-C, September 1988.

[5] McFarland, M.C., "BUD : Bottom-up Design of Digital Systems," in *Proc. 23rd Design Automat. Conf.*, July 1986, pp. 474-479.

[6] Paulin, P.G. and Knight, J.P., "Scheduling and Binding Algorithms for High-Level Synthesis," in *Proc. 26th Design Automat. Conf.*, June 1989.

[7] Paulin, P.G., "Force-Directed Scheduling for the Behavioural Synthesis of ASIC's," *IEEE Trans. Computer-Aided Design*, Vol. CAD-8, June 1989.

[8] Paulin, P.G. and Knight, J.P., "High-Level Synthesis Benchmark Results using a Global Scheduling Algorithm," presented at the International Workshop on Logic and Architecture Synthesis for Silicon Compilers, 1988.

[9] Paulin, P.G., Knight, J.P. and Girczyc, E.F., "HAL : A Multi-Paradigm Approach to Automatic Data Path Synthesis," in *Proc. 23rd Design Automat. Conf.*, July 1986, pp. 263-270.

[10] Kirkpatrick, S., Gelatt, C. and Vecchi, M., "Optimisation by Simulated Annealing," *Science*, Vol. 220, No. 4598, May 1983, pp. 671-680.

[11] Romeo, F. and Sangiovanni-Vincentelli, A., "Probabilistic Hill Climbing Algorithms : Properties and Applications," in *Proc. Chapel-Hill Conf. on VLSI*, 1985.

[12] Devedas, S. and Newton, A.R., "Algorithms for Hardware Allocation in Data Path Synthesis," in *Proc. ICCAD '87*, 1987, pp526-531.

[13] Devedas, S. and Newton, A.R., "Algorithms for Hardware Allocation in Data Path Synthesis," *IEEE Trans. Computer-Aided Design*, Vol CAD-8, No.7, July 1989, pp. 768-781.

SAVAGE:

A Simulated-Annealing based VLSI Architecture GEnerator

J.P. Neil and P.B. Denyer

Silicon Architectures Research Initiative
University Of Edinburgh
Department of Electrical Engineering
The Kings Buildings
West Mains Road
Edinburgh EH9 3JL

We present the prototype version of a flexible simulated-annealing based optimisation tool, capable of performing transformations on a data-flow graph within a directed silicon compilation environment, and mapping the transformed data-flow graph into a constrained hardware space. The use of the tool is illustrated with reference to the scheduling and allocation of the 1-dimensional 8-point Fast Discrete Cosine Transform.

1. Introduction

SAVAGE is an optimisation tool based upon the stochastic computational technique known as simulated annealing. SAVAGE is intended to form part of an interactive behavioural synthesis system, SAGE [1,2]. The aim of this work is to rapidly provide the design engineer with a number of differing architectural solutions for a given behavioural specification. These may range from a maximally parallel solution which has the shortest possible execution time, but is expensive in terms of hardware, to a serial architecture with a minimum hardware overhead, but which has a significant execution time in comparison.

We embrace De Mans architectural script-based synthesis paradigm [3]. This script-based approach has 3 levels, namely the design framework/common data model, the synthesis toolbox, and the architectural script. The work described in this paper corresponds to the development of a suitable synthesis toolbox.

This paper introduces the behavioural synthesis task. After a brief review of related work in this area, we present the software framework of the SAVAGE tool which illustrates the architectural script based method. Following a description of the scheduling and allocation models used, a design example, the 1-dimensional 8-point Fast Discrete Cosine Transform, is used to illustrate the tools capabilities. We examine the shortcomings of the current SAVAGE system, and draw some conclusions. Finally, we describe ongoing work and future directions.

2. The Behavioural Synthesis Task

Parker [4] states *"the synthesis task is to take a specification of the behaviour required of a system and a set of constraints and goals to be satisfied, and to find a structure that implements the behaviour while satisfying the goals and constraints".*

This task may be subdivided into a number of distinct steps :

(I) Transforming a behavioural description (usually written in a high level programming language such as ADA or Pascal) into some suitable internal representation. The most common approach is to represent the algorithm as two graphs; a data-flow graph whose nodes represent individual operations, and whose arcs represent communication

pathways between operations, together with a control flow graph which embodies conditional and looping constructs within the specification. These graphs can be combined.

(II) Scheduling aims to minimise the number of control steps need for the completion of the high level description. A control step broadly corresponds to a single state of a finite state machine.

(III) The allocation subtask aims to minimise the amount of hardware needed. 'Hardware' is defined here as functional units, memory elements and communication pathways.

(IV) The final major high level synthesis subtask is the derivation of a controller able to sequence operations on the datapath in a manner which corresponds to the behavioural specification.

Once these synthesis steps have been completed, logic and layout synthesis tools convert the resulting netlist into actual hardware.

The key steps in the synthesis task are the scheduling and allocation stages. It is important to note that the two subtasks are intimately related, for in order to determine a suitable degree of operational parallelism (scheduling), one has to know what functional units are available; conversely, in order to ensure judicious functional unit selection (and consequently utilisation), information from the schedule is required. Thus, there is a cyclic relationship.

2.1 Related Work

Paulin [5] classifies the following major scheduling/ allocation approaches :

1) Independent scheduling/allocation schemes.
2) Interdependent scheduling/allocation schemes.
3) Scheduling/allocation by stepwise refinement.

The simplest scheme is to schedule operations as soon as possible (ASAP). Many systems such as the Emerald/Facet System [6], the SARI tool, SAGE 2.0 [7] and the CATREE system [8] use this technique. This ASAP technique may be refined by allowing conditional postponement of operations as in the MIMOLA [9] and Flamel [10] systems.

Another major type of independent scheduler/ allocator uses list scheduling, where some pre-ordering of operations into lists via control information extracted from the data-flow and precedence graphs occurs. Scheduling then takes place into control steps. The EMUCS [11], SLICER [12] and the IMEC Cathedral-II [13] systems use this type of scheduler.

The MAHA [14] system is an example of the second technique, where scheduling and allocation are related. In this system, a critical path analysis is performed, and functional units are allocated in a first-come, first-served manner.

Scheduling and allocation by stepwise refinement is typified by the BUD-DAA system [15], by the HAL system [5,16,17,18] and the IBM Yorktown Silicon Compiler (YSC) [19]. In these systems a preliminary schedule is performed via a standard metric (for example by scheduling for the minimum number of control steps, as in the YSC). Functional units are then allocated and the schedule adjusted accordingly. This cycle is repeated until a set of cost criteria are realised. The HAL system uses a force-directed algorithm which enables specific physical information to be fed back to the scheduler allowing optimum scheduling and allocation to take place. This technique takes into account operators other than functional ones, such as memory and communication operators when determining the control step assignments. It also uses a built-in cost mechanism to allow trade-offs in functional unit, register and interconnect to be incorporated in the synthesis task to allow optimum operator scheduling.

The simulated annealing algorithm has been applied before in this field [20,21,22], but the algorithm as reported by Devedas and Newton required some explicit statement of serialism or parallelism to be included in the intermediate format. The approach

reported here differs in that no explicit statement of operation execution behaviour is required.

Further, the design environment in which SAVAGE operates requires rapid evaluation of architectural decisions. To achieve this, we have decided *not* to use SAVAGE to completely optimise and synthesise the datapath, but to apply a series of cost estimation functions to predict the hardware overhead incurred as a consequence of the scheduling and allocation functions. This enables different memory and communication strategies to be rapidly prototyped, simply by altering the cost estimation function.

There exists a discrepancy in the dimensions of the search space which the annealing function has to traverse. The time dimension is significant for complex examples, and individual nodes are allowed a significant degree of freedom in possible moves within that dimension (earliest possible schedule - latest possible schedule), whereas the hardware dimension is more restricted in terms of the moves which a single (data-flow graph) operation can make over functional units available from the library.

If the synthesis flow is viewed as a single stream where memory and communications are strongly dependant on the schedule and allocation selected [1], then the need to search the hardware resource space using simulated annealing diminishes. Consequently, we have selected to traverse the hardware resource space using deterministic methods.

3. The SAVAGE Framework

It is our intention to create a set of flexible optimisation tools. So far, we have developed a simulated annealing based core, and complemented this with a range of pre and post processing modules. (See Section 4.3)

We are currently developing a simple library based system of scheduling, allocate and cost functions, which the design engineer can select according to the problem. Such a library based approach will allow the rapid exploration of the design space.

The software structure is shown in figure 1.

4. Optimisation Procedures

Kirkpatrick, Gelatt and Vecchi [23] were the first to propose and demonstrate the application of simulation techniques from statistical physics to problems of combinatorial optimisation, specifically to the problems of wire routing and component placement in VLSI design.

```
procedure  sim_anneal  (k0,s0,inner-
                       loop)
begin
  k = k0;
  s = s0;
  repeat
    for i = 1 to inner_loop loop
      temp = generate(s);
      if (accept(c(temp),c(s), k) then
        s := temp;
      end if;
    end loop;
    update(k);
  until stopping criterion;
end sim_anneal;
```

where k_0 is a control parameter, s_0 is the initial system state and the function c() returns an assessment of the relative value of the current state based on suitable cost criteria. The acceptance of a new state is determined by accept() :

```
function accept (c(temp),c(s),k)
begin
  Δc = c(temp) - c(s);
  if ( random(0,1) < f(Δc,k)) then
    return (TRUE)
  else
    return (FALSE)
  end if;
end accept;
```

The pseudo code above contains a number of system transformation functions worthy of note :

1) The state transformation function, **generate()**.
2) The acceptance function, **f()**.
3) The control parameter update function,

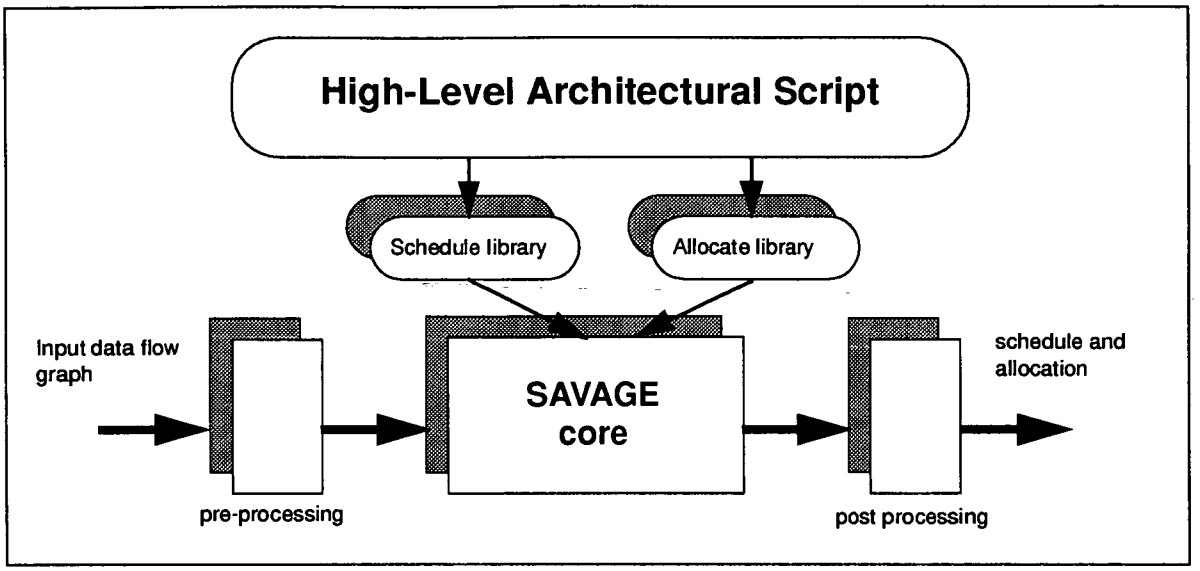


Figure 1. SAVAGE Software Structure

update().

4) The inner loop criterion.

5) The stopping criterion.

The state transformation function, generate(), is problem specific. The simpler systems use a pairwise interchange technique to generate the new configuration. This function, together with the cost function, c(), which tends to be based on the total interconnect length for placement packages based on simulated annealing, determines the final (optimal) system state.

A form of the acceptance function (Kirkpatrick, Gelatt and Vecchi) is :

$$f(\Delta c, k) = \min \{1.0, \exp((-\Delta c)/k)\}$$

With negative or zero Δc, the next state generated is always accepted. For positive Δc, the Boltzmann-like factor determines the probability of a generated state which has a higher cost than the current state being accepted as a valid state transformation.

The function update() provides a method for updating the control parameter, k. The most common form of the update function is :

$$k_{i+1} := \beta \bullet k_i \ , \qquad 0 < \beta < 1$$

The inner loop criterion is specified as the number of states generated per control parameter value. It is normally specified as some integer function of the initial system configuration.

The stopping criterion is defined as $\Delta c = 0$ over 3 annealing iterations.

In this paper, we are primarily concerned with the procedures which constitute the **generate** function.

4.1 A Scheduling Model

We select a node from the data-flow graph (or partition thereof) at random. Because of the observance of data-flow constraints, each node has an earliest possible schedule time and a latest possible schedule time. The actual execution time of the individual node will lie between these bounds.

The scheduling subtask simply corresponds to a random perturbation of the selected nodes execution time to a point within its valid schedule range. This process is shown in figure 2.

4.2 Allocation Strategy

After scheduling, a functional unit is selected based

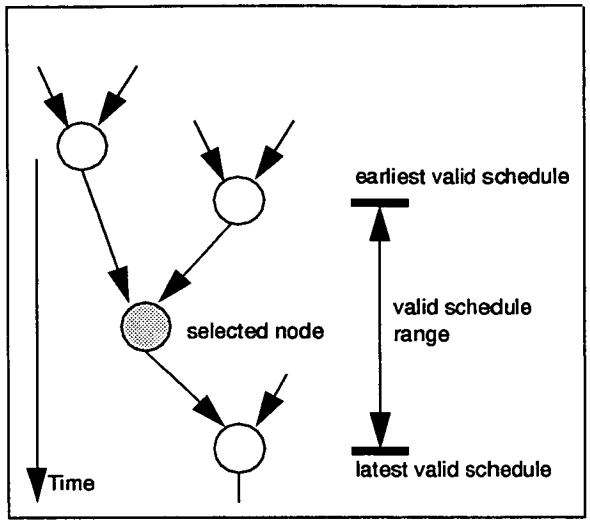


Figure 2. Scheduling Model

upon a simple load balancing criterion. This is defined as the *first-choice functional unit*. If the time slot is available on the first choice functional unit which corresponds to the new scheduled time of the selected operation, then a simple mapping between data flow graph and functional unit takes place.

If the first choice functional unit is unavailable at the scheduled time, then a search of the remaining functional units of appropriate operation class is initiated, and another unit selected. The mapping procedure is then reinvoked on the functional unit.

Should no time slots be available on any appropriate functional unit, then the operation is deferred to execute within its valid schedule range on the first choice functional unit. Should this procedure fail, a second choice functional unit is selected, and the deferment procedure repeated.

If all functional units are busy during the selected operations valid schedule range, then the user is flagged, and is prompted to either extend the valid schedule range of the particular operation (which as a side-effect, alters the schedule for all subsequent nodes in the data-flow graph), or to consider allocating an extra functional unit to the resource set. This action results in a re-allocation of operations over the new resource set.

4.3 Commentary

The scheduling and allocation procedures described above, coupled with an appropriate costing function which directs the optimisation to proceed towards an as-soon-as-possible (ASAP) solution, can produce optimised solutions for problems where a fixed hardware budget is specified, or where a speed constraint is desired.

An additional refinement is to perform some partitioning of the input data-flow graph prior to optimisation. The simplest technique to use here is a critical path analysis. This ensures that high-priority nodes (i.e. the critical path) are scheduled and allocated first. This is a good example of the flexibility of the SAVAGE system. Pre and post processing modules can be added, according to the desired architectural solution.

Further functionality is built into SAVAGE, allowing the user to develop a pipelined execution plan, once a schedule has been developed. The overall effect of this is to decrease the pipeline reuse time (input-to-input latency) with only a slight increase in cost to the pipeline propagation delay (input-to-output latency). This effect is easily achieved by selecting a suitable reuse time value, and invalidating the schedule of all nodes occurring after that time. The scheduling and allocation process is then reapplied, and a pipelined execution plan developed.

5. A Design Example

We present a typical design example used to test SAVAGE, namely the 1-dimensional 8 point Fast Discrete Cosine Transform [24].

The data flow graph generated in this example has 42 nodes, and 92 data-flow arcs. This represents a significant test of the optimisation system. The data-flow graph generated is shown in figure 3.

The SAVAGE tools was exercised on this example with a number of speed and hardware constraints. The results are presented in table 1. It is important to note that only the functional units and execution time

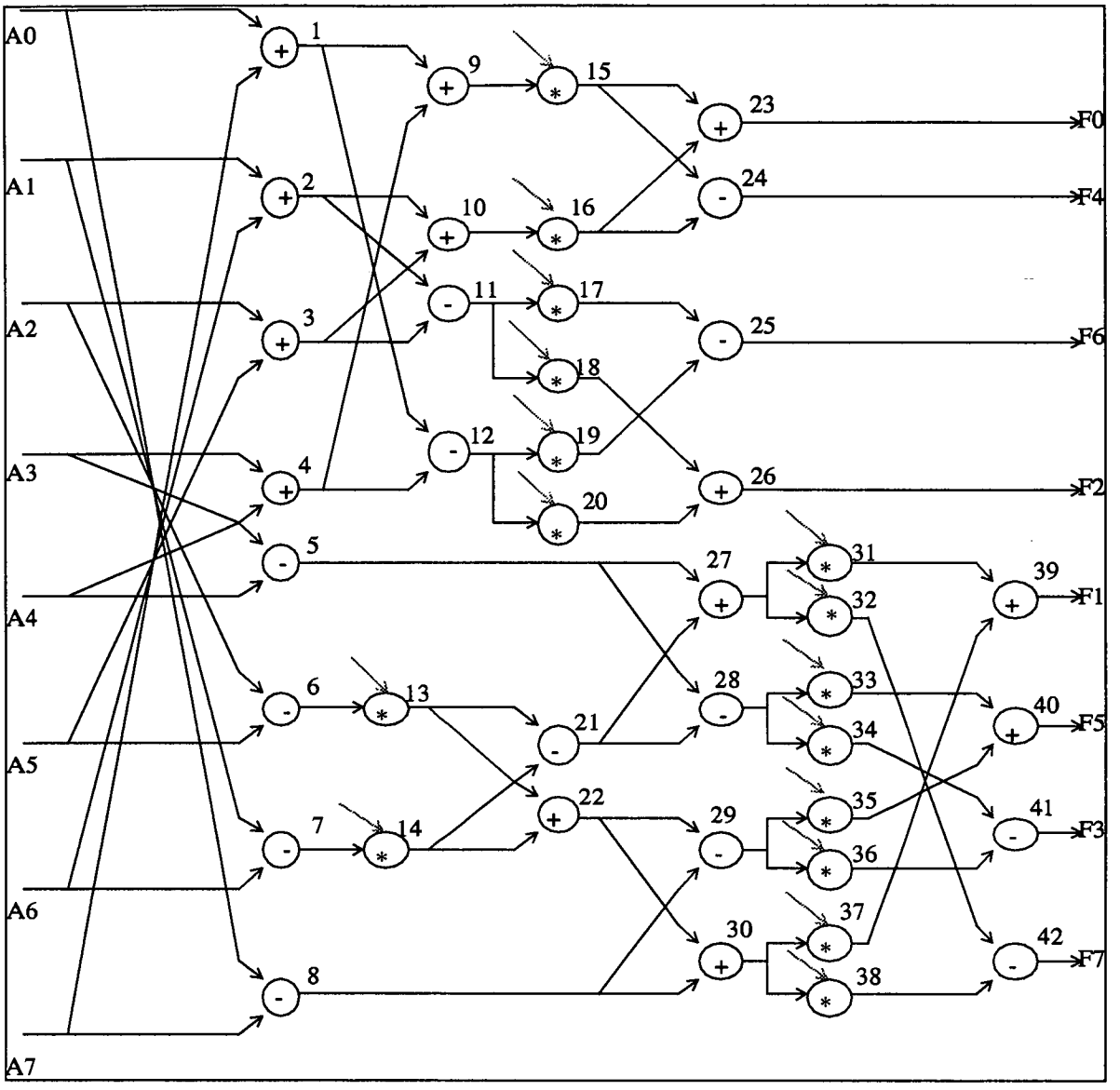


Figure 3. 1-Dimensional 8 Point Fast Discrete Cosine Transform

Cycles	*+*	*x*	*-*	*Av FU Util*	*Pipeline Reuse*	*Prop Delay*
20	1	1	1	70%	20	20
11	2	2	2	63%	11	11
8[1]	2	2	2	87%	8	15
8	3	3	3	58%	8	8
6[1]	3	3	3	77%	6	9
1.Pipelined execution plan						

Table 1. Results for 1-Dimensional 8 Point Fast Discrete Cosine Transform

are optimised. In this example, there is no direct optimisation of the registers or communication pathways used.

Table 1 shows how SAVAGE can present the designer with a number of architectural strategies ranging from a fully serial architecture to an architecture which has a pipeline reuse time corresponding to the As-Soon-As-Possible (ASAP) schedule of the FDCT block.

A good metric for the selection of an appropriate architecture is a measure of the average functional unit utilisation. As the table shows, a pipelined execution plan using 2 adders, 2 subtractors and 2 multipliers offers a considerable increase in functional unit utilisation over the other architectural strategies presented.

6. Limitations

In order to increase the versatility of the SAVAGE tool, more complex costing functions are required, which in turn require more detailed move sets for the scheduling and allocation procedures. In the prototype system, an ASAP regime suffices in order to prove the validity of the optimisation technique.

The second generation SAVAGE tool will have an enhanced set of cost functions covering scheduling strategies (ASAP, ALAP, AFAP, FDLS, etc.), more complex allocation strategies, and hardware costing functions designed to provide an approximate value of the hardware overhead in terms of registers and multiplexer inputs as a consequence of both the memory and communications architecture selected the scheduling and allocation techniques used.

7. Conclusions

We have presented a prototype optimisation system based around the stochiastic optimisation technique known as simulated annealing.

The tool has been tested on a number of significant examples, and its use has been illustrated here by searching the solution space for the 1-dimensional 8 point Fast Discrete Cosine Transform.

A number of limitations are immediately apparent. Firstly, the optimisation criteria used do not encompass memory or communication resource. Work is currently underway to correct this. Further, the strategies for optimisation are limited. Presently, only an As-Soon-As-Possible (ASAP) scheduling strategy is available.

Because of the modular nature of the SAVAGE system, however, future versions of the optimisation system will include a wide range of scheduling and allocation options. This work will complete the development of a suitable synthesis toolbox capable of supporting the architectural script based method advocated by De Man.

It is intended that SAVAGE 2 will be a prototype architectural script based system, where elements from the toolbox will be selected by heuristics determined by a number of high-level system goals, combined with the current architecture script.

Acknowledgements

This work was carried out as part of the Silicon Architectures Research Initiative. The use of facilities and resources is gratefully acknowledged.

The work reported here is supported by the Science and Engineering Research Council.

References

[1] Denyer, P.B., "SAGE Design Methodology," SARI Internal Technical Report SARI-035-D, March 1989.

[2] Mallon, D. and Denyer, P.B., "Behavioural Synthesis : An Interactive Approach," *IEE Colloquium Digest 1989/85*, May 1989, pp.2/1-2/8.

[3] De Man, H., "Tutorial On High-Level Synthesis" EDAC '90, March 1990.

[4] McFarland, S.J., Parker, A.C. and Camposano, R., "Tutorial on High-Level Synthesis," in *Proc. 25th Design Automat.*

Conf., July 1988, pp. 330-336.

[5] Paulin, P.G., "Force-Directed Scheduling for the Behavioural Synthesis of ASIC's," *IEEE Trans. Computer-Aided Design*, Vol. CAD-8, June 1989.

[6] Tseng, C. and Sewiorek, D.P., "Automated Synthesis of Data Paths in Digital Systems," *IEEE Trans. Computer-Aided Design*, Vol. CAD-5, July1985, pp. 379-395.

[7] Mallon, D., "SAGE Operation Power Tools," SARI Internal Technical Report SARI-014-C, September 1988.

[8] Gebotys, C.H. and Elmasry, M.I., "A VLSI Methodology with Testability Constraints," in *Proc. 1987 Canadian Conf. on VLSI*, October 1987.

[9] Marwedel, P., "A New Synthesis Algorithm for the MIMOLA Software System," in *Proc. 23rd Design Automat. Conf.*, July1986, pp. 271-277.

[10] Trickey, H., "Flamel : A High-Level Hardware Compiler," *IEEE Trans. Computer-Aided Design*, Vol. CAD-6, March 1987, pp. 259-269.

[11] Hitchcock, C.Y. and Thomas, D.E., "A Method Of Automatic Data Path Synthesis," in *Proc. 20th Design Automat. Conf.*, July 1983, pp. 484-489.

[12] Pangrle, B.M. and Gajski, D.D., "SLICER : A State Synthesiser for Intelligent Silicon Compilation," in *Proc. IEEE Conf. Computer Design*, October 1987.

[13] De Man, H., *et al.*, "CATHEDRAL-II : A Silicon Compiler For Digital Signal Processing," *IEEE Design and Test Magazine*, December 1986, pp. 13-25.

[14] Parker, A.C., *et al.*, "MAHA : A Program for Data Path Synthesis," in *Proc. 23rd Design Automat. Conf.*, July 1986, pp. 461-466.

[15] McFarland, M.C., "BUD : Bottom-up Design of Digital Systems," in *Proc. 23rd Design Automat. Conf.*, July 1986, pp. 474-479.

[16] Paulin, P.G. and Knight, J.P., "Scheduling and Binding Algorithms for High-Level Synthesis," in *Proc. 26th Design Automat. Conf.*, June 1989.

[17] Paulin, P.G. and Knight, J.P., "High-Level Synthesis Benchmark Results using a Global Scheduling Algorithm," presented at the International Workshop on Logic and Architecture Synthesis for Silicon Compilers, 1988.

[18] Paulin, P.G., Knight, J.P. and Girczyc, E.F., "HAL : A Multi-Paradigm Approach to Automatic Data Path Synthesis," in *Proc. 23rd Design Automat. Conf.*, July 1986, pp. 263-270.

[19] Camposano, R., "Structural Synthesis in the Yorktown Silicon Compiler," in *VLSI '87* (C.H. Sequin, ed.), New York : Elsevier, 1988, pp. 61-72.

[20] Devedas, S. and Newton, A.R., "Algorithms for Hardware Allocation in Data Path Synthesis," in *Proc. ICCAD '87*, 1987, pp526-531.

[21] Devedas, S. and Newton, A.R., "Algorithms for Hardware Allocation in Data Path Synthesis," *IEEE Trans. Computer-Aided Design*, Vol. CAD-8, No. 7, July1989, pp. 768-781.

[22] Safir, A. and Zavidovique, B.,"Towards a Global Solution to High Level Synthesis Problems", in *Proc. European Design Automat. Conf.*, March 1990, pp283-288.

[23] Kirkpatrick, S., Gelatt, C. and Vecchi, M., "Optimisation by Simulated Annealing," *Science*, Vol. 220, No. 4598, May 1983, pp. 671-680.

[24] Chen, W. and Fralick, S. "A Fast Computation Algorithm for the Discrete Cosine Transform", *IEEE Trans. Communications*, Vol. COM-25, No.9 September 1977, pp. 1004-1009.

Exploring Design Space Using SAVAGE : A Simulated Annealing based VLSI Architecture GEnerator

J.P. Neil and P.B. Denyer

Silicon Architectures Research Initiative
University Of Edinburgh
Department Of Electrical Engineering
West Mains Road
Edinburgh EH9 3JL
U.K.

We present the prototype version of a synthesis system based around an architectural script design method, and controlled by the computational technique known as simulated annealing. The system is exercised using a 5th order wave digital filter design, one of the benchmarks from the 1988 Workshop on High Level Synthesis

1.0 Introduction

As the demand for fast turnaround ASIC designs in industry increases, the complexity of the design process increases correspondingly. In response to these market demands, increasingly sophisticated Computer-Aided Design tools have become available which abstract much of the design process from the engineer.

Latest in the CAD tool field is the behavioural synthesis system, which takes a high-level algorithmic description of the desired circuit function, and follows a specific mapping process to produce a functional circuit. The constraints placed on the mapping process can often result in a simple template matching problem, where a specific algorithm is fitted onto a fairly predefined structure.

We present the prototype version of a CAD tool capable of allowing the designer to effectively explore the design space, and rapidly prototype a selection of architectural solutions to a given algorithmic specification.

SAVAGE [1,2,3] (a Simulated Annealing based VLSI Architecture GEnerator) is a library based synthesis system based around the stochastic computational technique known as *simulated annealing* [4]. It forms part of the Silicon Architectures Research Initiative (SARI) hosted at the University Of Edinburgh.[5,6,7]

The SAVAGE tool is based upon the Architectural Script design method as advocated by De Man [8] where the synthesis framework is viewed as a 3 level entity, with common data model elements at the core, operated on at a low level by a synthesis toolbox of base procedures, which are in turn controlled by an architectural script specifying their modes of operation.

Simulated Annealing has been used previously in the behavioural synthesis field, notably by Devedas and Newton [9]. The approach reported here has a number of significant differences. Firstly, SAVAGE needs no explicit statement of serialism or parallelism to execute, but more importantly SAVAGE offers the designer a greater degree of freedom by allowing interaction, not only by specifying the overall system goals, but by allowing the designer to specify the design techniques by which to achieve those goals.

The body of the paper is concerned with the development of an appropriate synthesis framework and base procedures capable of manipulating the data model. We introduce the SAVAGE tool structure in Section 2.0, and go on to develop models of scheduling, allocation, memory and communications synthesis procedures. We then present test results generated by the initial SAVAGE tool. Finally, some conclusions are offered, and directions for future work indicated.

2.0 Tool Structure

SAVAGE is based upon the same linear design flow as its parent project, SARI, where current synthesis decisions can directly affect "downstream" synthesis actions. This design flow is shown in Figure 1. The design tools reported here deal with the scheduling/

CH2819-1/90/0000-0104$01.00©1991 IEEE

allocation, memory and communication synthesis procedures. Before moving on to discuss the various design stages, we must define a method for assessing the quality of the solutions SAVAGE produces.

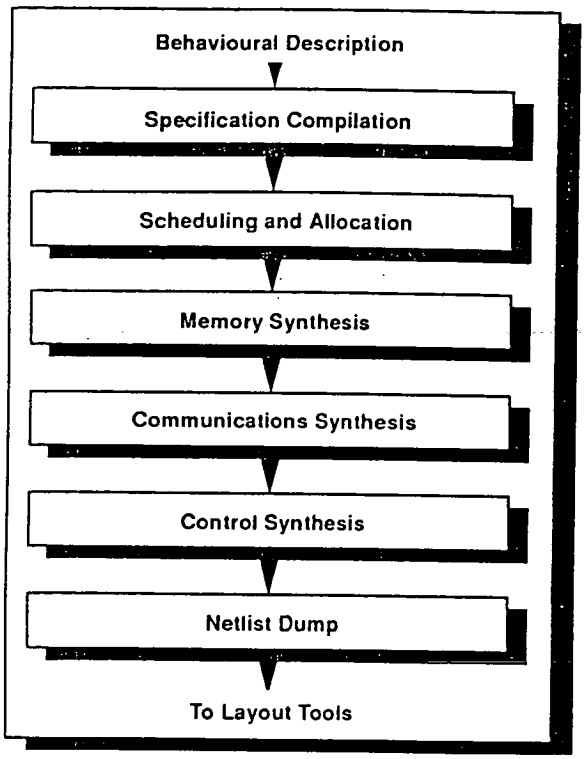


Figure 1. SARI Design Flow

2.1 Datapath Cost Assessment

Within SAVAGE, the "cost" of a particular datapath configuration is evaluated after each state generation. That is, a complete datapath is synthesised, and a weighted sum of the cost of each component, operational, memory and communication, is computed in terms of area, power consumption and total execution time.

The decision-making capability of the simulated annealing algorithm is controlled by the Metropolis criterion [10], whereby datapath states generated with a lesser cost than the previous datapath state are always accepted, whereas states generated with a greater cost are accepted with a probability function described by :

$$Accept = min\left\{1.0, e^{\frac{(-\Delta c)}{T}}\right\}$$

where Δc is the difference in cost between the two states, and T is a control parameter, which simulates the temperature of the melt in the physical annealing process. This function allows hill climbing moves to be made, and alleviates the problem of local minima, which is a characteristic of greedy heuristic search strategies.

2.2 Scheduling And Allocation Strategies

The aim of the scheduling function is to set each node within the input data flow graph to execute at such a time so that the data flow constraints are not violated. The allocation function ensures that each data flow graph operation executes on an appropriate hardware unit. These two functions are strongly interrelated, for in order to produce an efficient schedule, some knowledge about the functional unit allocation is required, whilst allocation cannot take place without an indication of parallelism within the data flow graph, which, in turn, comes from the schedule. Thus there is a circular relationship.

The scheduling function within the SAVAGE core is implemented by selecting a node at random from the data flow graph, and perturbing its execution time, subject to the data flow constraints. The perturbment can be biased in such a way so as to produce a number of different scheduling strategies (e.g. ASAP, ALAP, FDLS, etc.). This pseudo-random perturbment forms the basis of the hill climbing move set described above.

The allocation procedure operates on a simple load balancing criterion, whereby parallelism is balanced on a clock-cycle to clock-cycle basis.

These initial scheduling and allocation functions were chosen to demonstrate the SAVAGE concept, and also because of their relative ease of implementation.

2.3 Synthesis Procedures

The aims of the memory and communications synthesis procedures of a behavioural synthesis system may be stated as the provision of storage resource for data transfers which have a duration greater than the unit clock cycle, and the provision of data transfer resource from the output port of one function resource (operational/memory/external interface) to the input port(s) of other functional resources, respectively.

These definitions are made without any reference to the actual form of the components synthesised, or their communication topology; rather, they are a functional definition of memory and communication synthesis procedures in general. This is in keeping with the

architectural script synthesis paradigm where various architectural techniques can be used to provide these synthesis procedures.

The prototype version of SAVAGE reported here uses a simple left-edge memory allocation strategy, and performs a 'best-fit' grouping of signals to the resulting register locations. These registers are then grouped into common files according to the match between source and destination ports.

A clique partitioning algorithm is used to generate the minimum number of transfer pathways between connected functional resources. This connectivity information is generated in the preceding memory allocation phase. Once these transfer pathways have been defined, we impose a two-level multiplexing regime which ensures a minimum transfer pathway delay.

3.0 Test Data and Results

SAVAGE has been tested on a variety of problems drawn from the signal processing domain. In this paper, we present results obtained from trials carried out on one of the benchmarks from the 1988 Workshop on High-Level Synthesis, a 5th Order Elliptic Wave Digital Filter design (Figure 2), as popularised by Paulin [11]. This design represents a significant test for current behavioural synthesis systems, and contains 34 operation nodes with 76 communication arcs.

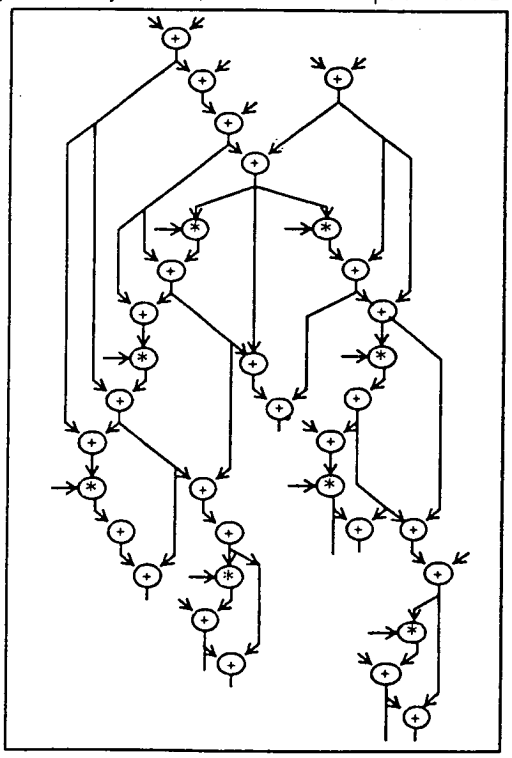


Figure 2. Wave Digital Filter Flow Graph

The SAVAGE tool was exercised on the design with 3 predefined operational resource allocations, consisting of 2 ADD units, and either a single MULT unit, a MULTP (pipelined) unit, or 2 MULTP units. These allocations were selected simply to facilitate an easy comparison between synthesis systems. These results, along with published results for other programs are shown in Table 1.

3.1 Discussion

Table 1 indicates that SAVAGE is capable of synthesising highly optimised solutions. This may be directly attributed to the global cost assessment mechanism associated with the simulated annealing algorithm. These results, however, indicate the quality of solution for a particular architectural style, namely a single level of bussing, and a 2 level multiplexer regime. The completed version of SAVAGE will allow the designer to explore a wider range of architectural options very rapidly.

System	C_steps	Mult	Registers	Mux. Inputs	Reg. files
Hal [11]	19	1P	12	26	-
Hal	21	1	12	30	-
Hal	19	2	12	28	-
Esc [12]	19	1P	15	25	-
Esc	21	1	16	23	8
Esc	19	2P	15	26	8
Splicer[13]	21	1	12	35	-
SAW [14]	19	2P	12	34	-
SPAID[15]	19	1P	19	33	6
SPAID	21	1	19	31	5
SAVAGE	19	1P	12	19 (+7)[1]	2
SAVAGE	21	1	12	21(+10)[1]	4
SAVAGE	18	2P	14	15 (+8)[1]	3

1. Figures in brackets indicate additional multiplexers required for register file decoding

Table 1. Results for Wave Digital Filter

4.0 Conclusions and Future Work

We have presented a prototype version of a powerful synthesis system which utilises the architectural script design method, and uses the simulated annealing process as its control mechanism.

Tests carried out on established benchmarks indicate that SAVAGE is capable of producing high quality solutions to behavioural synthesis problems. The

flexibility of the SAVAGE system allows a designer to rapidly evaluate the effects of high-level architectural decisions on the quality of the final solution.

The current version of SAVAGE is limited by the number of architecture synthesis modules and scheduling/allocation move sets available. The current SAVAGE was intended as a concept demonstrator. A more serious limitation is the inability to operate on hierarchical designs. This problem may be partially solved by designating null or busy periods on all resources to simulate some hierarchical resource sharing.

Future generations of the tool set will have a range of synthesis modules available capable of synthesising a number of recognised memory and communication architectures, which will be coupled with an expanded set of moves available to the scheduling and allocation tool.

Once this synthesis toolbox is in place, an architectural script interface will be implemented which will allow the designer to specify a set of system goals to be satisfied, along with an initial set of scheduling/ allocation and synthesis strategies. The final component of the script will be a set of architectural pragmas, which will allow the designer to apply 'rules-of-thumb' gained through design experience to the synthesis procedure. These measures will allow the designer to have a direct influence on the final architectural *form*, but will abstract the designer from the implementation *detail*.

Acknowledgements

This work was carried out as part of the Silicon Architectures Research Initiative. The use of facilities and resources is gratefully acknowledged.

The work reported here is supported by the Science and Engineering Research Council and the University Of Edinburgh.

References

[1] Neil, J.P. and Denyer, P.B., "Synthesis By Simulated Annealing",*IEE Colloquium Digest 1989/125*, November 1989, pp.9/1-9/4.

[2] Neil, J.P. and Denyer, P.B., "SAVAGE : A Simulated-Annealing based VLSI Architecture Generator", *IEEE Workshop on Genetic Algorithms, Simulated Annealing and Neural Networks applied to Signal and Image Processing*, May 1990.

[3] Finlay, I.W., Neil, J.P. and Denyer, P.B., "Filter Synthesis using Behavioural Design Tools" to be published in *Proc. 16th European Solid State Circuits Conference*, Grenoble, September 1990.

[4] Kirkpatrick, S., Gelatt, C. and Vecchi, M., "Optimisation by Simulated Annealing," *Science*, Vol. 220, No. 4598, May 1983, pp. 671-680.

[5] Grant, P.M., "The DTI-Industry Sponsored Silicon Architectures Research Initiative", *IEE Electronics & Communications Engineering Journal*, Vol. 2 No. 3, June 1990.

[6] Mallon, D. and Denyer, P.B., "Behavioural Synthesis : An Interactive Approach," *IEE Colloquium Digest 1989/85*, May 1989, pp.2/1-2/8.

[7] Denyer, P.B., "SAGE Design Methodology," SARI Internal Technical Report SARI-035-D, March 1989.

[8] De Man, H., "Tutorial On High-Level Synthesis" EDAC '90, March 1990.

[9] Devedas, S. and Newton, A.R., "Algorithms for Hardware Allocation in Data Path Synthesis," *IEEE Trans. Computer-Aided Design*, Vol. CAD-8, No. 7, July1989, pp. 768-781.

[10] Metropolis, N., Rosenbluth, A., Rosenbluth, M.,Teller, A. and Teller,E., "Equation of State Calculations by Fast Computing Machines", *Journal Chem. Phys.*,21/6(1953):1087.

[11] Paulin, P.G. and Knight, J.P., "Scheduling and Binding Algorithms for High-Level Synthesis," in *Proc. 26th Design Automat. Conf.*, June 1989.

[12] Stok, L., "Interconnect Optimisation During Data Path Allocation," *in Proc. EDAC '90*, pp. 141-146, March 1990.

[13] Pangrle, B.M., "Splicer: A Heuristic Approach to Connectivity Binding," in *Proc. 25th Design Automat. Conf.*, July 1988.

[14] Thomas, D.E. et al., "The System Architect's Workbench," in *Proc. 25th Design Automat. Conf.*, July 1988.

[15] Haroun, B.S. and Elmasry, M.I., "Architectural Synthesis for DSP Silicon Compilers,"*IEEE Trans. Computer-Aided Design*, Vol. CAD-8, No. 4, April 1989.

Filter Synthesis using Behavioural Design Tools

I.W. Finlay, J.P. Neil and P.B Denyer

Silicon Architectures Research Initiative
University Of Edinburgh
Department of Electrical Engineering
The Kings Buildings
West Mains Road
Edinburgh EH9 3JL

We present a 5th-Order Wave Digital Filter datapath designed using behavioural synthesis tools developed as part of the SARI project at the University Of Edinburgh. We use a simulated annealing based optimisation system to develop the execution plan for the filter, with a datapath synthesised using clique partitioning and a novel heuristically driven module selection mechanism. These tools may be characterised by their ability to rapidly search the solution space for a given behavioural specification.

1.0 Introduction

As the demand for Application Specific Integrated Circuits continues to rise in the computer manufacturing industry and elsewhere, increasingly sophisticated design tools are required to enable non-expert silicon designers to realise complete systems on a single chip. Newest in this design automation field is the behavioural synthesis tool which takes an algorithmic description of the required system behaviour and subsequently synthesises a logically correct datapath/controller system which corresponds to the required behaviour.

The work described in this paper is part of the Silicon Architectures Research Initiative [1,2] at the University of Edinburgh. An interactive behavioural synthesis system is being developed (SAGE [3]) which allows the designer to make high-level system decisions which will affect the ultimate silicon realisation, but which abstracts the synthesis procedures from the designer.

We report on the development of part of the synthesis system (SAVAGE), and demonstrate its use by synthesising a major design, namely a 5th-Order Wave Digital Filter.

We introduce the behavioural synthesis problem, and then move on to discuss the relevant subtasks. We present the experimental results gathered thus far, and finally offer some conclusions on the efficiency of the toolset.

2.0 The Behavioural Synthesis Task

Parker [4] states *"the synthesis task is to take a specification of the behaviour required of a system and a set of constraints and goals to be satisfied, and to find a structure that implements the behaviour while satisfying the goals and constraints".*

This task may be subdivided into a number of distinct steps :

(I) Transforming a behavioural description (usually written in a high level programming language such as ADA or Pascal) into some suitable internal representation. The most common approach is to represent the algorithm as two graphs; a data-flow graph whose nodes represent individual operations, and whose arcs

represent communication pathways between operations, together with a control flow graph which embodies conditional and looping constructs within the specification. These graphs can be combined.

(II) Scheduling aims to minimise the number of control steps need for the completion of the high level description. A control step broadly corresponds to a single state of a finite state machine.

(III) The allocation subtask aims to minimise the amount of hardware needed. 'Hardware' is defined here as functional units, memory elements and communication pathways.

(IV) The final major high level synthesis subtask is the derivation of a controller able to sequence operations on the datapath in a manner which corresponds to the behavioural specification.

Once these synthesis steps have been completed, logic and layout synthesis tools convert the resulting netlist into actual hardware.

The key steps in the synthesis task are the scheduling and allocation stages. It is important to note that the two subtasks are intimately related, for in order to determine a suitable degree of operational parallelism (scheduling), one has to know what functional units are available; conversely, in order to ensure judicious functional unit selection (and consequently utilisation), information from the schedule is required. Thus, there is a cyclic relationship.

The simulated annealing algorithm has been applied before in this field [5,6,7], but the algorithm as reported by Devedas and Newton required some explicit statement of serialism or parallelism to be included in the intermediate format. The approach reported here differs in that no explicit statement of operation execution behaviour is required.

Further, the design environment in which SAVAGE operates requires rapid evaluation of architectural decisions. To achieve this, we have decided *not* to use SAVAGE to completely optimise and synthesise the datapath, but to apply a series of cost estimation functions to predict the hardware overhead incurred as a consequence of the scheduling and allocation functions. This enables different memory and communication strategies to be rapidly prototyped, simply by altering the cost estimation function.

2.1 A Scheduling Model

From the design representation described above, a scheduling model based upon the stochastic computational technique known as simulated annealing[8] can be developed. In this model, a node from the data flow graph is selected at random and its execution time randomly perturbed, subject to data-flow graph constraints. This constraint is defined as the valid schedule range of the node, and is shown in figure 1.

The simulated annealing model allows scheduling moves which represent an overall increase in system cost to be accepted, dependant on a control parameter, which, in the simulated annealing system corresponds to the temperature of the physical annealing system.

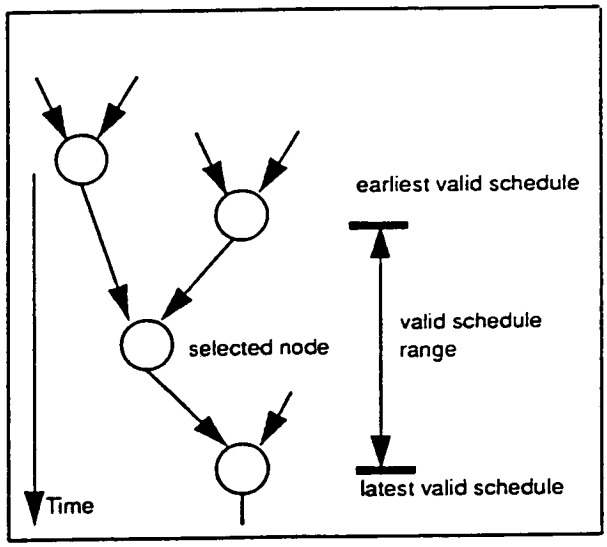


Figure 1. Scheduling Model

2.2 Allocation Strategy

After scheduling, a functional unit is selected based upon a simple load balancing criterion. This is defined as the *first-choice functional unit*. If the time slot is available on the first choice functional unit which corresponds to the new scheduled time of the selected operation, then a simple mapping between data flow graph and functional unit takes place.

If the first choice functional unit is unavailable at the scheduled time, then a search of the remaining functional units of appropriate operation class is initiated, and another unit selected. The mapping procedure is then reinvoked on the functional unit.

If all functional units are busy during the selected operations valid schedule range, then the user is flagged, and is prompted to either extend the valid schedule range of the particular operation (which as a side-effect, alters the schedule for all subsequent nodes in the data-flow graph), or to consider allocating an extra functional unit to the resource set. This action results in a re-allocation of operations over the new resource set.

2.3 Datapath Allocation

Datapath allocation is composed of two important synthesis tasks:

(I) Binding signals to memory locations

(II) Binding signal transfers to interconnections.

As McFarland stated [9], local interconnection comprising of multiplexing and bus inputs constitutes a substantial part of the design cost in terms of both speed and area. In attempting to minimise data path allocation costs most attention is paid to minimising memory requirements. Systems such as MAHA[10], SPLICER [11] and FACET [12] adopt this approach and do not consider the effects on interconnection costs when minimising the number of memory locations. Memory requirements are reduced by forcing signals which have disjoint lifetimes to share the same memory location. The 'Left Edge' algorithm [10] guarantees an optimal solution to this problem. Other approaches such as edge colouring and clique partitioning techniques are also used. HAL [13] incorporates interconnection cost into the memory minimisation by weighting the signal compatibility graph. The technique uses clique partitioning in a stepwise manner where signals with favourable weightings are partitioned and merged first. This has the added advantage of reducing the complexity of the graphs to be partitioned.

The approach taken in this paper differs from existing techniques in that it considers memory and interconnections costs jointly,and minimises them by merging wires rather than registers.

2.4 Architectural style

The interconnection style is restricted to two levels of multiplexing and a single level of bussing for each processor-memory-processor transfer path and so it is not possible to synthesise memory-memory communications. This style has two advantages. Firstly, it ensures minimum time delay through interconnect paths. Splicer and SAW [14]permit up to four levels of multiplexing and, hence, have twice the worst case delay. Secondly, as registers and register files are each connected to a single bus the interconnection topology is linear which has benefits in layout area.[15]

2.5 Synthesis Algorithm

The aim of data path allocation is to provide interconnection paths and storage locations for signal transfers between processors. Information from the scheduling and allocation phase is used to generate the necessary wiring between processors. Initially, each wire is associated with a single register. Signal transfers are assigned to wires with the appropriate source and destination. Signals can only share the same wire if their lifetimes do not overlap. If the memory location is not free for the lifetime of the signal then a new wire is created. The algorithm minimises memory locations, multiplexers and interconnection by merging wires. Any two wires can be merged provided that no signals carried by them need to be transferred at the same time. Clique partitioning is used to find all maximal merges of wires. Merges are then made on the basis of memory and multiplexer cost estimates. The memory cost estimator is based on the reduction in required memory locations resulting from the merge and how well the locations are used. The multiplexer cost estimator is based on the number of shared sources and destinations in the merge, offset by the resultant increase in the number of multiplexer and bus inputs.

3.0 Results

The optimisation tools were tested on the benchmark example for the 1988 Workshop on High-Level Synthesis, a 5th Order Elliptical Wave Digital Filter. The schedule and allocation tool was exercised with a variation in minimum c-step requirement, together with appropriate weightings for register and multiplexer overheads in the cost assessment function.

The results from the SAVAGE synthesis procedures are shown in Table 1. The synthesis procedures took a default allocation of 2 adders,with a variable number of multipliers. ('P' indicates a pipelined multiplier).The number of multiplexer inputs generated by SAVAGE is calculated in the same manner as the other systems, however, there

System	C_steps	Mult	Registers	Mux. Inputs	Reg. files
Hal	19	1P	12	26	-
Hal	21	1	12	30	-
Hal	19	2	12	28	-
Esc [16]	19	1P	15	25	-
Esc	21	1	16	23	8
Esc	19	2P	15	26	8
Splicer	21	1	12	35	-
SAW	19	2P	12	34	-
SPAID	19	1P	19	33	6
SPAID	21	1	19	31	5
SAVAGE	19	1P	12	19	2
SAVAGE	21	1	12	21	4
SAVAGE	18	2P	14	15	3

Table 1. Synthesis results

is extra decoding required in using register files which is not necessary in HAL, Splicer or SAW.

It may be seen that SAVAGE performs favourably in comparison with the other quoted systems.

4.0 Conclusions

A suite of behavioural synthesis tools has been developed which is capable of rapidly exploring the solution space of a given problem. The tools have been used to generate a number of solutions for the benchmark example from the 1988 Workshop on High-Level Synthesis. These solutions compare favourably with those solutions generated by other behavioural synthesis systems.

The modular construction of SAVAGE allows a number of different scheduling and allocation strategies to be tested before committing the selected solution for datapath synthesis and optimisation. The simulated annealing algorithm may be viewed as a general purpose heuristic capable of generating near globally optimal solutions to large problems. When this optimisation technique is coupled with powerful costing, scheduling and allocation functions, the search space can be efficiently traversed, and optimal solutions generated.

A novel approach to datapath allocation has been presented. Minimising memory and multiplexing requirements simultaneously by merging interconnections has been shown to generate datapath architectures with lower communications overheads than existing systems.

Acknowledgements

This work was carried out as part of the Silicon Architectures Research Initiative. The use of facilities and resources is gratefully acknowledged.

The work reported here is supported by the Science and Engineering Research Council and British Aerospace.

References

[1] Grant, P.M., "The DTI-Industry Sponsored Silicon Architectures Research Initiative", *IEE Electronics & Communications Engineering Journal*, Vol. 2 No. 3, June 1990.

[2] Mallon, D. and Denyer, P.B., "Behavioural Synthesis : An Interactive Approach," *IEE Colloquium Digest 1989/85*, May 1989, pp.2/1-2/8.

[3] Denyer, P.B., "SAGE Design Methodology," SARI Internal Technical Report SARI-035-D, March 1989.

[4] McFarland, S.J., Parker, A.C. and Camposano, R., "Tutorial on High-Level Synthesis," in *Proc. 25th Design Automat. Conf.*, July 1988, pp. 330-336.

[5] Devedas, S. and Newton, A.R., "Algorithms for Hardware Allocation in Data Path Synthesis," in *Proc. ICCAD '87*, 1987, pp526-531.

[6] Devedas, S. and Newton, A.R., "Algorithms for Hardware Allocation in Data Path Synthesis," *IEEE Trans. Computer-Aided Design*, Vol. CAD-8, No. 7, July1989, pp. 768-781.

[7] Safir, A. and Zavidovique, B.,"Towards a Global Solution to High Level Synthesis Problems", in *Proc. European Design Automat. Conf.*, March 1990, pp283-288.

[8] Kirkpatrick, S., Gelatt, C. and Vecchi, M., "Optimisation by Simulated Annealing," *Science*, Vol. 220, No. 4598, May 1983, pp. 671-680.

[9] McFarland, M.C., "Re-evaluating the Design Space for Register Transfer Hardware Synthesis," in *Proc. ICCAD '87*, 1987.

[10] Kurdahi, F.J. and Parker, A.C., "REAL: A Program for REgister ALlocation," in *Proc. 24th Design Automat. Conf.*, July 1987.

[11] Pangrle, B.M., "Splicer: A Heuristic Approach to Connectivity Binding," in *Proc. 25th Design Automat. Conf.*, July 1988.

[12] Tseng, C. and Sewiorek, D.P., "Automated Synthesis of Data Paths in Digital Systems," *IEEE Trans. Computer-Aided Design*, Vol. CAD-5, July1985, pp. 379-395.

[13] Paulin, P.G. and Knight, J.P., "Scheduling and Binding Algorithms for High-Level Synthesis," in *Proc. 26th Design Automat. Conf.*, June 1989.

[14] Thomas, D.E. et al., "The System Architect's Workbench," in *Proc. 25th Design Automat. Conf.*, July 1988.

[15] Haroun, B.S. and Elmasry, M.I., "Architectural Synthesis for DSP Silicon Compilers,"*IEEE Trans. Computer-Aided Design*, Vol. CAD-8, No. 4, April 1989.

[16] Stok, L., "Interconnect Optimisation During Data Path Allocation," *in Proc. EDAC '90*, pp. 141-146, March 1990.

Chapter 4

Simulated Annealing Based Synthesis of Fast Discrete Cosine Transform Blocks

J.P. Neil and P.B. Denyer

4.1 *Introduction*

This Chapter describes CAD techniques capable of synthesising Fast Discrete Cosine Transform (FDCT) Blocks from *behavioural*, or *algorithmic*, specifications. We introduce SAVAGE (a Simulated Annealing based VLSI Architecture GEnerator), a software tool developed under the auspices of the Silicon Architectures Research Initiative (SARI(Grant, 1990)) hosted at the University of Edinburgh.

SAVAGE is capable of taking a data-flow description of an input algorithm, and applying a number of synthesis steps, or transformations, to produce a hardware netlist of a datapath. The netlist description is then passed to logic synthesis and layout tools to complete the route to silicon. These application specific synthesis steps are controlled by the computational technique known as *simulated annealing.*

This Chapter reviews the design process, from the initial high-level description of the FDCT, through the various synthesis transformations, and presents a set of test results illustrating the flexibility of the SAVAGE software. Finally, some extensions to the prototype SAVAGE system are described.

4.2 *Problem Domain*

The large amount of information contained within a high definition digital image poses significant problems, both in terms of memory requirement and transmission latency in applications where real time, or near real time, image transmission is required.

As a result, many data compression techniques have been proposed (Chen and Smith, 1977, Wintz, 1972 and Soame, 1982). The Discrete Cosine Transform (DCT) operates on a series of blocks decomposed from the original image. These blocks are ranked according to their a.c. energy (a.c. energy quantifies the amount of information within a particular block). A bit assignment according to the average point variance within the block then takes place. It is here that the data compression takes place; more bits are assigned to visually "important" regions (i.e. regions of the image containing most information) than to those of lesser interest.

The Discrete Cosine Transform, F(k) of a discrete function f(j), j = 0,1, ... , N-1 where N is the set of data points is :

$$F(k) = \frac{2c(k)}{2} \sum_{j=0}^{N-1} f(j) \cos\left[\frac{(2j+1)k\pi}{2N}\right]$$

where k = 0, 1, ... , N-1 and $c(k) = \frac{1}{\sqrt{2}}$ for k = 0 and $c(k) = 1$ for k = 1, 2, ..., N-1

Previously, the DCT has been implemented using a double size Fast Fourier Transform (FFT) employing complex arithmetic and operating on 2N coefficients. The Fast Discrete Cosine Transform (FDCT) (Chen *et al*, 1977) alleviates the implementation problems associated with the DCT by using only real arithmetic and operating on N data points. This results in a factor of six reduction in the algorithm complexity.

The FDCT is most readily expressed in terms of an extensible flow graph. The 1-dimensional 8-point Fast Discrete Cosine Transform is shown in Figure 1.

4.3 *Synthesis and Simulated Annealing*

This section describes the behavioural synthesis procedure. The simulated annealing algorithm is introduced as a general purpose optimisation technique which has been applied most notably in VLSI floorplanning

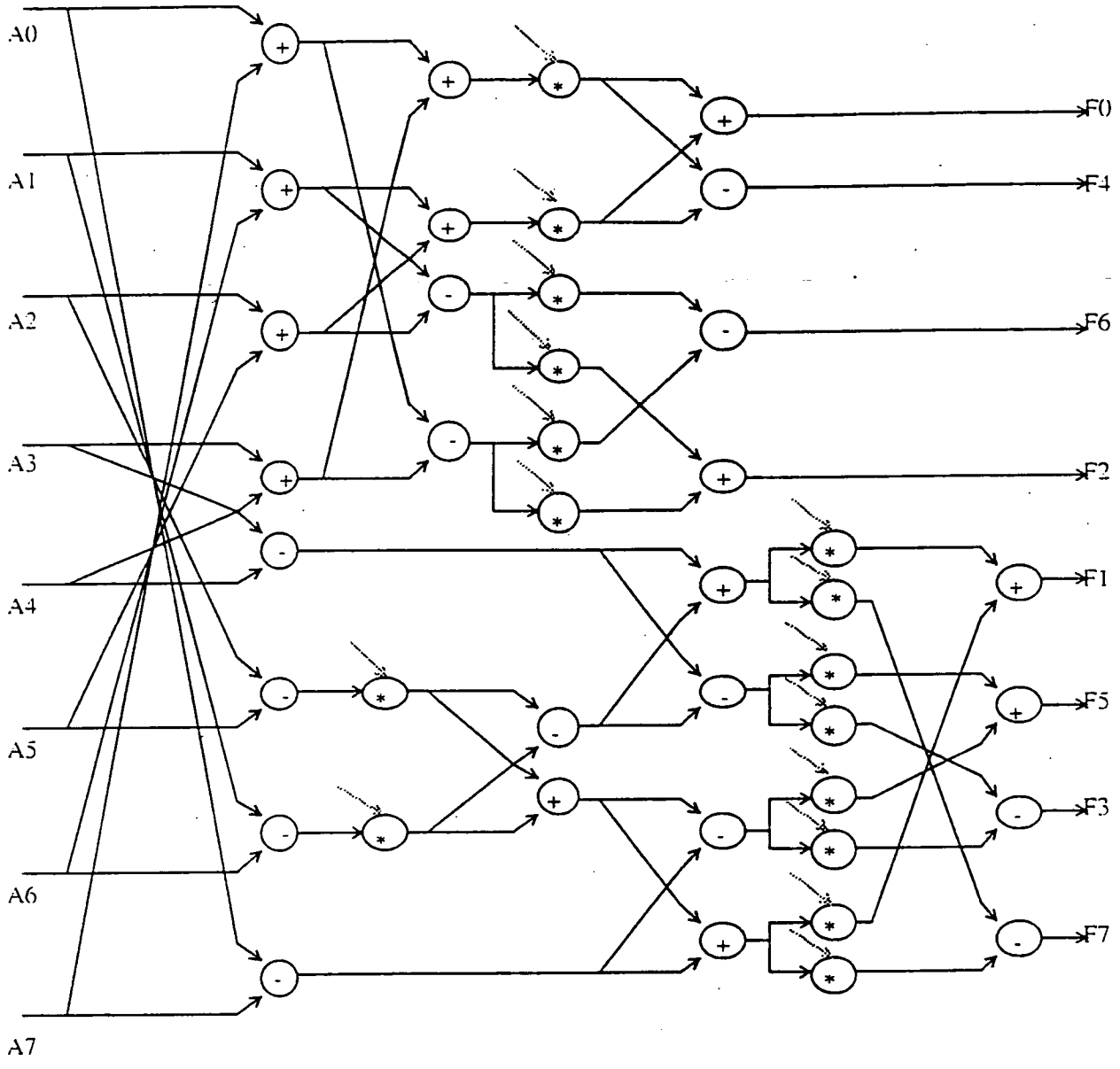


Figure 1 1-dimensional 8-point fast discrete cosine transform

problems. A formulation of the behavioural synthesis procedure is introduced which is amenable to a simulated annealing based implementation, and a relationship between the synthesis flow advocated by Denyer (Denyer, 1989) and the simulated annealing algorithm is developed.

4.3.1 The behavioural synthesis procedure

The behavioural synthesis task may be defined, at a high level, as the translation of a set of algorithmic descriptions of the required system behaviour into some suitable circuit formulation. This task may be subdivided as follows :

1) Compilation into a suitable intermediate data-structure. Current research concentrates on a relatively small core of data models, typically represented as either separated control and data flow graphs (SARI), combined control and data flow graphs (EASY (Stok and van de Born, 1988)) or tree structures (SILAGE (Hilfinger, 1984), Mimola (Marwedel, 1985)). Typical compiler optimization techniques can be applied at this stage.

2) Scheduling and Allocation. The scheduling subtask deals with the assignment of a suitable control step to individual data-flow graph operations, while the allocation subtask assigns particular data-flow graph operations to functional units. These subtasks are intimately related, for in order to determine an efficient schedule, some knowledge about the functional unit allocation is required, whilst allocation cannot take place without an indication of parallelism within the data flow graph, which, in turn, comes from the schedule.

3) Structural Synthesis. Within this step, the necessary memory and communications infrastructure required to complete the data-path, subject to the behavioural specification is generated.

4) Controller synthesis. This stage generates a suitable controller capable of sequencing data-flow operations on the specified datapath.

The version of SAVAGE reported here is a prototype system designed to investigate the scheduling and allocation stages of the behavioural synthesis procedure. Once a suitable schedule and allocation has been determined, then other software tools are invoked to complete the structural synthesis.

4.3.2 The simulated annealing algorithm

Simulated Annealing is a stochastic computational technique derived from statistical mechanics for finding near globally minimum cost solutions to large optimisation problems. Kirkpatrick, Gelatt and Vecchi (Kirkpatrick *et al*, 1983) were the first to propose and demonstrate the application of simulation techniques from statistical physics to problems of combinatorial optimisation, specifically to the problems of wire routing and component placement in VLSI design.

In general, finding the global minimum value of an objective function with many degrees of freedom subject to conflicting constraints is an NP-complete problem (Romeo and Sangiouanni-Vincentelli, 1985), since the objective function will tend to have many local minima. A procedure for solving hard optimisation problems should sample values of the objective function in such a way as to have a high probability of finding a near optimal solution and should also lend itself to efficient implementation. Recently, simulated annealing has emerged as a viable technique which meets these criteria. Rutenbar (Rutenbar, 1989) provides an elegant disposition on the subject.

The following pseudo-code function illustrates the structure of the subclass of probabilistic hill climbing (PHC) algorithms known as *simulated annealing.*

```
function sim_anneal (initial_state, k0)

I       : STATE;
K       :CONTROL_PARAM;
COUNT   :INTEGER;

begin
   K = k0;
   I = initial_state;
   while (not stopping criterion) loop
      for count = 1 to #MOVES
         generate a new state;
         compute change in system energy, ΔE;
         if (ΔE < 0)
            /* LOWER COST - ACCEPT IT */
            accept this move; update I;
         else
            /* HIGHER COST - ACCEPT IT MAYBE */
            accept with probability P = e^(-ΔE/K)
            update I iff accepted;
      end for;
   update K;
   end while;
end sim_anneal;
```

where I is a state variable (in this case the datapath state), and K is the control parameter which models the temperature in the physical annealing system.

ΔE represents the change in energy between the current state and the state produced by the random perturbation of the data flow graph. The assessment of energy or cost, is discussed in Section 4.2.3. The stopping criterion is defined as $\Delta E = 0$ over 3 control parameter decrements. This ensures that the data flow graph has assumed a minimum energy configuration. The inner loop counter #MOVES determines the number of state generations per control parameter value.

In SAVAGE, the control parameter update function is defined :

$$K_{n+1} = K_n \, \alpha(K_n)$$

where $0 < \alpha K_n < 1$

4.3.2.1 Synthesis and simulated annealing

There exist a number of synthesis systems which use simulated annealing to produce data paths. Most notable are those developed by Devedas and Newton (Devedas and Newton, 1987 and Devedas and Newton, 1989), and Safir and Zavidovique (Safir and Zavidovique, 1990).

We can formulate the scheduling and allocation problem in terms of individual data flow graph node placement within a Resource-time (*Rt*) space. Rt space can be viewed, at the simplest level, as a bounded grid whose axes represent the various hardware units available to execute data flow operations and machine execution cycles, or *c-steps*, respectively. The scheduling and allocation operations may then be defined as a node displacement in *Rt* space, subject to individual data flow graph dependencies.

We can develop a simulated annealing based synthesis model through the integration of the linear design flow (described in Section 4.1) into the generate function of the simulated annealing algorithm. Selecting the finest computational "grain" (i.e. operating on single data-flow graph nodes), we can ensure that hill climbing moves can be attained at a minimal global cost. Every state generation cycle selects a data-flow graph node at random from the node set, assigns a c-step value to it, and binds it to a particular hardware resource, as shown in Figure 2.

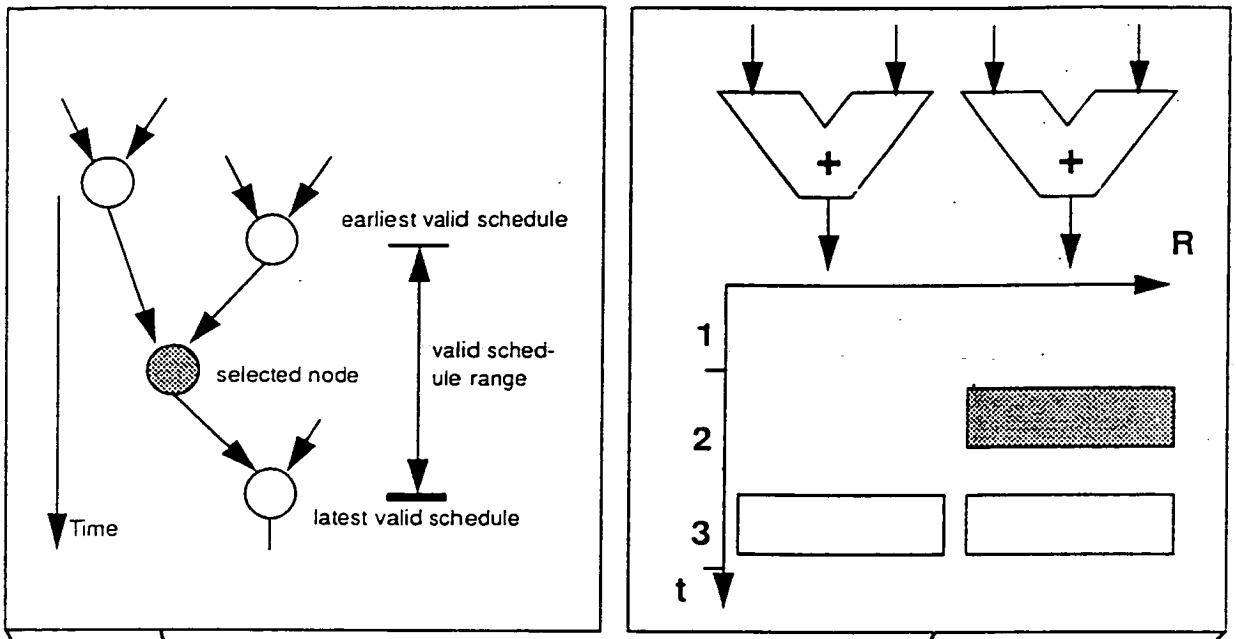
4.3.2.2 Scheduling and allocation move set development

The scheduling component of the node translation was partitioned into 3 main stages. For the selected node, the *valid schedule range* is computed first. This operation is shown in Figure 3, and represents the computation of the upper and lower bound on the temporal displacement. A sequence of possible execution times is then randomly generated within the valid

schedule range. The length of this sequence is proportional to the size of the valid schedule range. Finally, an execution time is selected at random from this sequence. This corresponds to the new execution time of the node.

By using this technique, a number of refinements were added to the basic scheduling operation. Selection of an execution time generated at random over the total valid schedule range ensured that genuine hill climbing moves were made available to the annealing procedure. The adaptive nature of the length of the execution time sequence increases the efficiency of the algorithm towards the end of the annealing run, where lower cost moves are generally achieved during the allocation phase, as most nodes have tended towards their optimum As-Soon-As-Possible (ASAP) schedule. Finally, to increase the performance of the scheduling algorithm during the early stages of the optimisation, where potential hill climbing moves have little effect on the overall quality of the final solution, the execution time sequence can be 'biased' to produce sequences of predominantly earlier execution times ($t_S' < t_S$) forcing a trend towards rapid ASAP type schedules.

The allocation move set developed was, by necessity, more deterministic in nature than the scheduling move set. In the most general view of synthesis, the module allocation procedure must ensure that a hardware component capable of executing the operation class is available at the scheduled time, t_S . A greedy heuristic allocation strategy will produce a module allocation equivalent to the maximum degree of parallelism of a particular operation class within a specific data flow graph. For practical purposes, this scheme represents a very inefficient use of available silicon area. Within SAVAGE, the allocation strategy is based on a 3 phase scheme. First, all hardware modules not supporting the operation class of the selected node are eliminated from the computation. From the remaining modules, a target hardware unit is selected based around a simple load balancing criteria. If the target module is free at t_S' , then a simple binding between node and hardware module is established. If t_S' is unavailable on the target module, then it is eliminated from the set of candidate modules, and the allocation process reinvoked. Should the allocation process fail (i.e. all candidate modules are flagged as busy during t_S'),




```
function sim_anneal (initial_state, k0)

I : STATE;
K : CONTROL_PARAM;
COUNT : INTEGER;

begin
 K = k0;
 I = initial_state;
 while (not stopping criterion) loop
  for count = 1 to #MOVES
   generate a new state;
   compute change in system energy, ΔE;
   if (ΔE < 0)
    /* LOWER COST - ACCEPT IT */
    accept this move; update I;
   else
    /* HIGHER COST - ACCEPT IT MAYBE */
    accept with probability P = e^(-ΔE/K)
    update I iff accepted;
  end for;
 update K;
 end while;
end sim_anneal;
```

Figure 2 Integrating the scheduling and allocation into the annealing algorithm

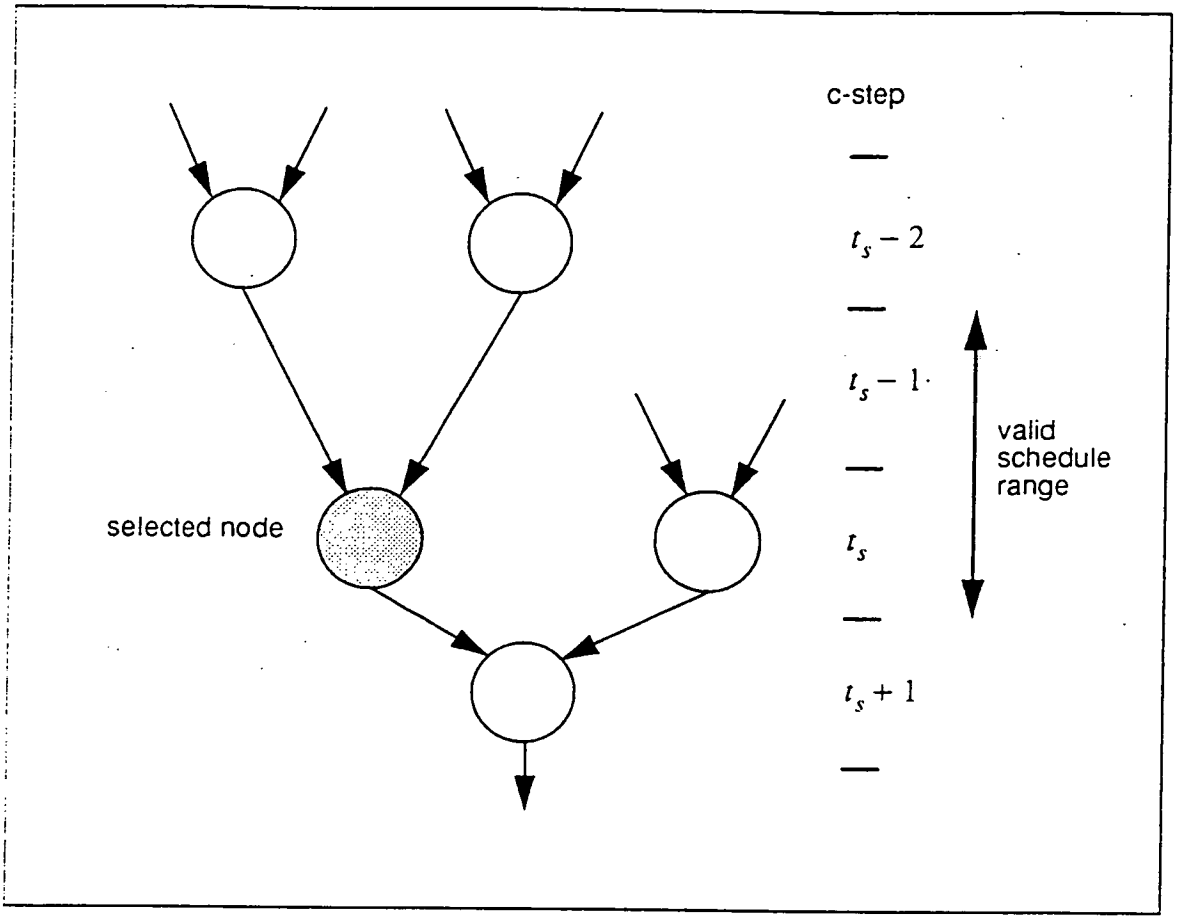


Figure 3 Computing the valid scheduling range

then in the earliest SAVAGE system, the user was prompted to either alter the valid schedule range of the node (i.e. manually alter the schedule), or allocate an extra hardware module of a corresponding class.

This manual intervention led to slow run times and a tendency towards greedy module allocation. Subsequently, the allocation strategy was revised to support operation deferment. The target module was still selected according to a load balancing criteria, but if the module was unavailable, then an extra c-step was inserted at the appropriate execution time, and the binding process invoked.

This allocation strategy allows a minimal hardware set to be used when operating under a time constraint, and ensures a global balancing of operation concurrency during the allocation phase.

SAVAGE supports a simple pipelining algorithm, similar to that described by Mallon (Mallon and Denyer, 1990). Here, the pipeline reuse time (initiation interval of successive pipeline tasks) may be specified as the timing constraint, and the pipeline latency (input to output latency of a single pipeline task) is optimised.

This pipelining operation can be viewed as a "folding" of the *Rt* space so that operations occurring after the computed pipeline reuse time are retimed to occur in free cycles in the next pipeline task.

4.3.2.3 Datapath costing

In developing a costing method for SAVAGE, a number of factors have to be considered. Firstly, as SAVAGE operates only on an incomplete part of the datapath solution space (namely the scheduling and allocation phases of the synthesis procedures), the costing functions used will not reflect the true cost of the datapath. The SAVAGE operational scenario has the user constraining one axis of the Resource-time space before the optimisation procedures are invoked. Correspondingly, the primary element of the costing function has to assess whether the datapath generated lies outwith the axis boundary specified by the user. Designs violating these boundaries are penalised heavily.

Part of the design specification for the SAVAGE software was to achieve a high utilisation of the functional units used within the solution datapath. The costing functions reflect this by penalising those designs which have functional units operating below a specific utilisation threshold (also specified by the user).

Further, datapaths generated as a result of more global perturbations to the solution - for example, where an extra control step is inserted into the schedule, and subsequent operations are retimed - are also penalised.

Thus the costing mechanism may viewed as a hierarchical structure, where gross system objectives are assigned a high importance while strategic goals, such as the attainment of a minimum functional unit utilisation occur at a lower level in the cost assessment hierarchy. Finally, library specific costing functions occur at the lowest level.

In this way, the annealing procedure is guided towards a solution which satisfies the gross system requirements quickly.

4.3.2.4 Costing mechanics

In common with Devedas and Newton, we formulate the datapath cost as a weighted sum of all hardware components within the datapath, combined with a weighted cost accounting for the total number of c-steps needed. (In the prototype SAVAGE system, structural synthesis takes place after the scheduling and allocation operations had been completed, and so the costings associated with these components were unavailable to the simulated annealing procedure.)

We extend the Devedas and Newton costing in keeping with the hierarchical costing model described above. Thus :

$$
\begin{aligned}
\text{COST}_{\text{DATAPATH}} = {} & W_1.\text{VIOLATIONS}_{\text{BOUNDARY}} \\
& + W_2.\text{VIOLATIONS}_{\text{FU\_UTILISATION}} \\
& + W_3.\text{VIOLATIONS}_{\text{RETIMING}} \\
& + W_4.\#\text{FUNCTIONAL\_UNITS} \\
& + W_5.\#\text{C-STEPS}
\end{aligned}
$$

The weightings can be varied to produce datapaths of varying architectural styles. For example, where the designer does nor explicitly wish to constrain the hardware resources available, but would prefer a solution with only a single multiplier, then the multiplier weight can be set proportionally higher, so that single multiplier solutions will have a lower global cost.

4.4 *Test results*

SAVAGE operates in a batch mode with the designer constraining either the hardware set available or the overall execution time desired.

(As SAVAGE can support simple pipelining, then the pipeline reuse time can be specified as a timing constraint)

The 1-dimensional 8-point FDCT was coded in SLANG (the SARI input LANGuage) as shown in Figure 4. The resulting data flow graph corresponds to Figure 1. The results shown in Table 1 were produced by specifying a hardware set for SAVAGE apart from those indicated as a pipelined solution, where a specific pipeline reuse time was specified.

Cycles	+	×	-	Av FU Util	Pipeline Reuse	Prop. Delay
20	1	1	1	70%	20	20
11	2	2	2	63%	11	11
8[1]	2	2	2	87%	8	15
8	3	3	3	58%	8	8
6[1]	3	3	3	77%	6	9
1. Pipelined execution plan						

Table 1 SAVAGE test results

A metric commonly used when assessing digital systems is the utilisation of each functional unit within the system. Here, it can be seen that the pipelined solution with a reuse time of 8 cycles offers the best time/hardware trade-off, and so this partially completed datapath was selected as the target for the remaining structural synthesis.

4.4.1 The structural synthesis tools

The prototype SAVAGE system used a simple left-edge algorithm to produce the memories required for the computed results and intermediate signals within the data flow graph. This algorithm produces the optimal memory *allocation*, but does not produce the optimal signal *groupings*.

```
procedure FDCT_1D    ( A0,A1,A2,A3,A4,A4,A6,A7 : in FLOAT;
                       F0,F1,F2,F3,F4,F5,F6,F7 : out FLOAT) is

  COS_PI_4 : constant    :=    0.707_106_78;        -- cos(PI/4)
  COS_PI_8 : constant    :=    0.923_879_53;        -- cos(PI/8)
  SIN_PI_8 : constant    :=    0.382_683_43;        -- sin(PI/8)

  COS_3_PI_16 :constant  :=    0.831_469_61;        -- cos(3*PI/16)
  SIN_3_PI_16 :constant  :=    0.555_570_23;        -- sin(3*PI/16)

  COS_5_PI_16 :constant  :=    SIN_3_PI_16;         -- cos(5*PI/16)
  SIN_5_PI_16 :constant  :=    COS_3_PI_16;         -- sin(5*PI/16)

  COS_7_PI_16 :constant  :=    SIN_PI_16;           -- cos(7*PI/16)
  SIN_7_PI_16 :constant  :=    COS_PI_16;           -- sin(7*PI/16)

  B0,B1,B2,B3,B4,B5,B6,B7 : FLOAT;
  C0,C1,C2,C3,C4,C5,C6,C7 : FLOAT;
  D0,D1,D2,D3,D4,D5,D6,D7 : FLOAT;

  COS_PI_4_TIMES_B5 : FLOAT;
  COS_PI_4_TIMES_B6 : FLOAT;

  COS_PI_4_TIMES_D0 : FLOAT;
  COS_PI_4_TIMES_D1 : FLOAT;

begin

  B0 := A7 + A0; B1 := A6 + A1;                    -- first pass
  B2 := A5 + A2; B3 := A4 + A3;
  B4 := A3 - A4; B5 := A2 - A5;
  B6 := A1 - A6; B7 := A0 - A7;

  -- Put the expressions COS_PI_4*B5 and COS_PI_4*B6 into intermediate
  -- variables so as to avoid evaluating them twice

  COS_PI_4_TIMES_B5 := COS_PI_4*B5;                -- second pass
  COS_PI_4_TIMES_B6 := COS_PI_4*B6;

  C0 := B3 + B0; C1 := B2 + B1;
  C2 := B1 - B2; C3 := B0 - B3;
  C4 := B4;
  C5 := COS_PI_4_TIMES_B6 - COS_PI_4_TIMES_B5;
  C6 := COS_PI_4_TIMES_B6 + COS_PI_4_TIMES_B5;
  C7 := B7;

  D0 := C0; D1 := C1;                              -- third pass
  D2 := C2; D3 := C3;
  D4 := C4 + C5; D5 := C4 - C5;
  D6 := C7 - C6; D7 := C7 + C6;

  -- Put the expressions COS_PI_4*D0 and COS_PI_4*D1 into intermediate
  -- variables so as to avoid evaluating them twice

  COS_PI_4_TIMES_D0 := COS_PI_4*D0;
  COS_PI_4_TIMES_D1 := COS_PI_4*D1;

  F0 := COS_PI_4_TIMES_D0 + COS_PI_4_TIMES_D1;     -- fourth pass
  F4 := COS_PI_4_TIMES_D0 - COS_PI_4_TIMES_D1;
  F2 := SIN_PI_8*D2 + COS_PI_8*D3;
  F6 := COS_3_PI_16*D3 - SIN_PI_8*D2;
  F1 := SIN_PI_16*D4 + COS_PI_16*D7;
  F5 := SIN_5_PI_16*D5 + COS_5_PI_16*D6;
  F3 := COS_3_PI_16*D6 - SIN_3_PI_16*D5;
  F7 := COS_7_PI_16*D7 - SIN_7_PI_16*D4;

end FDCT_1D;
```

Figure 4 SLANG description of FDCT

A greedy bus merger algorithm was used to synthesise the communications infrastructure required to complete the datapath. Here, replicated links between functional units (including the newly synthesised memories) are removed.

SAVAGE has been coupled with other datapath synthesis tools (Neil and Denyer, 1990) to synthesise a 5th Order Wave Digital Filter. Later iterations of the SAVAGE software include a complete route to datapath synthesis where structural synthesis is examined more fully.

4.5 *Conclusions*

This Chapter has described SAVAGE, a software tool capable of synthesising datapaths from behavioural descriptions. SAVAGE has been used in the development of datapaths for the 1-dimensional 8-point Fast Discrete Cosine Transform.

We have shown that given either a speed or hardware bound, SAVAGE can produce optimised solutions for both pipelined and non-pipelined designs which are comparable with those in published literature(Mallon and Denyer, 1989). SAVAGE allows the designer to rapidly explore the solution space for a given problem, and by varying the optimisation criteria produce a number of comparable datapaths. The designers own expertise is then used to select the most appropriate datapath solution.

SAVAGE has also been used to develop solutions to other synthesis benchmarks, notably the 5th Order Elliptic Wave Digital Filter, popularised by Paulin (Paulin and Knight, 1989 and Neil and Denyer, 1990).

4.5.1 Current developments

The software architecture for the SAVAGE toolset has been shown to be robust and flexible during the design cycle. Further structural synthesis move sets have been added to complete the SAVAGE route to datapath generation. This has been complemented by a corresponding increase in

complexity of the datapath costing function. Indeed, move sets have been added which support a number of different architectural styles. This expansion has led to the development of eXtended SAVAGE (XSAVAGE); this CAD tool supports the "Architectural Script" based synthesis paradigm, first introduced by De Man (DeMan, 1990 and DeMan October, 1990). XSAVAGE is characterised by a 4 level hierarchy of user interaction, namely :

1) *System Level Interaction.* This level of interaction enables us to convey system level information, such as total chip area, maximum acceptable power consumption and timing specifications to the optimisation system.

2) *Strategic Interaction.* At this level in the hierarchy, we can specify the optimisation techniques that will form the generate function within the simulated annealing core. These comprise scheduling, allocation and memory and communication synthesis strategies. Also included here is the costing information.

3) *Pragmatic Interaction.* In many synthesis systems, application specific designer knowledge cannot be included in the specification.We provide a mechanism by which designers can affect the synthesis procedures directly via architectural pragmas.

4) *Structural Interaction.* At the lowest level in the script hierarchy, interaction takes place at the component level. Partial and complete architectures can be specified through the SAVAGE Structural Description Language.

4.5.2 Concluding remarks

The previous section describes XSAVAGE, a software tool which has evolved from the SAVAGE software which was initially intended to synthesise Fast Discrete Cosine Transform blocks. XSAVAGE is a much

more powerful system, capable of producing optimised solutions of widely differing architectural styles for a given problem domain. XSAV-AGE may be classified not as a *problem-specific* synthesis system, but rather as a general synthesis *framework* capable of supporting application specific architectures.

4.6 *Acknowledgements*

This work was carried out as part of the Silicon Architectures Research Initiative. The use of facilities and resources is gratefully acknowledged. The work reported here is supported by the Science and Engineering Research Council and the University Of Edinburgh.

4.7 *References*

CHEN W., SMITH C.H. AND FRALICK S.C., "A Fast Computational Algorithm for the Discrete Cosine Transform", IEEE Trans. Commun., 1977, com25, (9), pp. 1004-1009.

CHEN W. AND SMITH C.H., "Adaptive Coding of Monochromatic and Colour Images ", IEEE Trans. Commum., 1977, com25, (11), pp. 1285-1292.

DE MAN H., "Tutorial On High-Level Synthesis" EDAC '90, March 1990.

DE MAN H., "CAD For Real Time Information Processing Systems : Challenges and Opportunities", in Proc. SASIMI '90, October 1990, Kyoto, Japan, pp. 65-72

DENYER P.B., "SAGE Design Methodology", SARI Internal Technical Report SARI-035-D, March 1989.

DEVEDAS S. AND NEWTON A.R., "Algorithms for Hardware Allocation in Data Path Synthesis," in Proc. ICCAD '87, 1987, pp. 526-531.

DEVEDAS S. AND NEWTON A.R., "Algorithms for Hardware Allocation in Data Path Synthesis," IEEE Trans. Computer-Aided Design, Vol. CAD-8, No. 7, July1989, pp. 768-781.

GRANT P.M., "The DTI-Industry Sponsored Silicon Architectures Research Initiative", IEE Electronics & Communications Engineering Journal, Vol. 2 No. 3, June 1990.

HILFINGER P.N., "SILAGE: A Language for Signal Processing", University of California, Berkley, 1984.

KIRKPATRICK S., GELATT C. AND VECCHI M., "Optimisation by Simulated Annealing," Science: Vol. 220, No. 4598> May, 1983, pp671-680.

MALLON D. AND DENYER P.B., "Behavioural Synthesis : An Interactive Approach," IEE Colloquium Digest 1989/85, May 1989, pp.2/1-2/8

MALLON D., AND DENYER P.B., "A new Approach to Pipelining Optimisation", in Proc. European Design Automat. Conf. , March 1990.

MARWEDEL P., "The MIMOLA Design System: A Design System which spans several levels", in Methodologies of Computer System Design, ed. Shriver, North Holland, 1985, pp. 223-237.

NEIL J.P. AND DENYER P.B., "Exploring Design Space using SAVAGE: A Simulated Annealing based VLSI Architecture GEnerator", in Proc. 33rd Midwest Symposium on Circuits and Systems, Calgary, August 1990.

PAULIN P.G. AND KNIGHT J.P., "Scheduling and Binding Algorithms for High-Level Synthesis," in Proc. 26th Design Automat. Conf., June 1989.

ROMEO F. AND SANGIOVANNI-VINCENTELLI A., "Probabilistic Hill Climbing Algorithms : Properties and Applications," in Proc. Chapel-Hill Conf. on VLSI, 1985.

RUTENBAR R.A., "Simulated Annealing Algorithms : An Overview," IEEE Circuits and Devices Magazine, January 1989, pp. 19-26.

SAFIR A. AND ZAVIDOVIQUE B.,"Towards a Global Solution to High Level Synthesis Problems", in Proc. European Design Automat. Conf., March 1990, pp. 283-288.

SOAME T.A., "Bandwidth Compression of Images Using Transform Techniques", GEC J. Sci. Tech., 1982, 48, (1), pp. 17-23.

STOK L. AND VAN DER BORN R., "EASY: Multiprocessor Architecture Optimisation", in Proc. Int. Workshop on Logic and Architecture Synthesis for Silicon Compilers, Grenoble, May 1988, pp. 313-328.

WINTZ P.A., "Transform Picture Coding", Proc. IEEE, 1972, 60, (7), pp. 809-819.