Modular decomposition techniques for stored-logic digital filters

This item was submitted to Loughborough University’s Institutional Repository by the/an author.

Additional Information:


Metadata Record: https://dspace.lboro.ac.uk/2134/34123

Publisher: © Mohamed Arif Bin Nun

Rights: This work is made available according to the conditions of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0) licence. Full details of this licence are available at: https://creativecommons.org/licenses/by-nc-nd/4.0/

Please cite the published version.
MODULAR DECOMPOSITION TECHNIQUES
FOR STORED-LOGIC DIGITAL FILTERS

BY

MOHAMED ARIF BIN NUN,
B.Sc. (University of London),
M.Sc. (Loughborough University of Technology).


Supervisor: M.E. WOODWARD, Ph.D.
Department of Electronic and Electrical Engineering.

To my wife, THYE KHIM, for her patience, sacrifice and emotional support, and to my parents for their understanding.
ACKNOWLEDGEMENTS

I am grateful to the Malaysian Government for supporting me financially throughout my studies, and to Professor J.W.R. Griffiths, Head of the Department of Electronic and Electrical Engineering, for providing the research facilities. I would also like to thank my Research Supervisor, Dr. M.E. Woodward, who introduced me to the concept of 'closed' partitions, for his help and encouragement.

In addition, I am indebted to Dr. D.J. Quarmby, Manager of the Signal Analysis Centre, and Dr. R.P. Knott from the Engineering Mathematics Department, for their invaluable specialist advice. I also appreciate Dr. R. Steele's interest in my work, and his optimistic encouragement.

During my research, my close friends and colleagues, especially Mr. F.T. Sakane, Dr. U. Somaini and Dr. M.L. Rahman, provided a healthy mixture of humourous and serious discussions, and for them I have only fond memories.

Finally, I must mention, with gratitude, the efficient service of the Staff of the University Library, and the immaculate typing of my Thesis by Mrs. B. Wright.

M.A. Bin Nun, June 1977.
Digital filtering is an important signal processing technique whose theory is now well established. At present, however, there are no well defined and systematic methods available for realising digital filters in hardware.

This project aims to develop such methods which are general and technology independent, and adopts a systems and sub-systems design philosophy. The realisation problem is approached in a new way using concepts from finite-automata theory and implementing complete digital filter sections as stored-logic units. Two methods are introduced and developed.

In the first, a complete basic second-order filter is directly modelled as a finite-state sequential machine (F.S.M.) and implemented with memory devices whose storage capacity is reduced by the application of a well known method of machine decomposition via 'closed' partitions.

To initiate a systematic analysis of the partition structure of the F.S.M. digital filter, a study is made into the algebraic decomposition structure of the basic computational units making up the filtering algorithm.

The insight gained is useful in a subsequent analysis which shows that a second-order filter section, suitably simplified and modelled as an F.S.M., may be decomposed into a parallel connection of smaller sub-machines, each of which, in turn, being composed of a 'nested' cascade interconnection of still simpler components. The overall memory requirement of the decomposed realisation is considerably less than that of the direct one.
The second method presents a technique of 'digit slicing' over a variable number base, using which a filter section may be realised as a regular interconnection of identical sub-filters of short wordlengths. The technique leads to a flexibility in hardware count and processing mode, and a modular expandibility in computational accuracy and filter order. It is suited to implementations using large-scale integrated (L.S.I.) devices.

Practical prototypes are constructed using programmable and erasable memory modules.

The methods developed provide a general theoretical basis for the hardware realisation of digital filters. It is hoped that its main usefulness lies in bridging the gap between the initial analytical description of the desired frequency characteristics and corresponding filter transfer function, and the actual hardwired practical implementation.
# Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Introduction</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0 Introduction</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.2 Motivation for project</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1.3 Design philosophy and problem formulation</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1.4 Scope of research and organisation of Thesis</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2.0 Introduction</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>2.1 Descriptions of a general digital filter</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>2.1.0 Dynamics of digital filters</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>2.2 Applications and advantages</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>2.3 Design and realisation</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>2.3.0 Mathematical design</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>2.3.0.0 Design techniques for FIR and IIR filters</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>2.3.1 Effects of finite-length registers</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>2.3.2 Considerations in the real-time hardware implementation</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>2.3.3 Existing hardware design approaches</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>2.3.4 Conclusion</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>3.0 Introduction</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>3.1 Descriptions of F.S.M's</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>3.2 Interconnections of F.S.M's</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>3.3 Problem areas in F.S.M. realisation</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>3.4 Basic algebraic concepts</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>3.5 Structural decompositions of F.S.M's</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>3.6 State reduction using S.P. partitions</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>3.7 Conclusion</td>
<td>34</td>
</tr>
</tbody>
</table>
## CHAPTER 4  FinitE-State Machine Models of Stored-Logic  Digital Filters

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0</td>
<td>Introduction</td>
<td>35</td>
</tr>
<tr>
<td>4.1</td>
<td>General approach</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>Stored-logic digital filters</td>
<td>35</td>
</tr>
<tr>
<td>4.3</td>
<td>Examples of S.L. digital filters</td>
<td>37</td>
</tr>
<tr>
<td>4.3.0</td>
<td>Second-order non-recursive section</td>
<td>37</td>
</tr>
<tr>
<td>4.3.1</td>
<td>First-order recursive section</td>
<td>38</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Second-order autonomous recursive section</td>
<td>38</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Memory storage requirements</td>
<td>41</td>
</tr>
<tr>
<td>4.4</td>
<td>F.S.M. models of digital filters</td>
<td>41</td>
</tr>
<tr>
<td>4.4.0</td>
<td>F.S.M. model of a general S.L. non-recursive second-order section</td>
<td>43</td>
</tr>
<tr>
<td>4.4.0.0</td>
<td>An application</td>
<td>43</td>
</tr>
<tr>
<td>4.4.0.1</td>
<td>State-reduction of the general F.S.M. non-recursive section</td>
<td>46</td>
</tr>
<tr>
<td>4.4.0.2</td>
<td>Partial state reduction</td>
<td>49</td>
</tr>
<tr>
<td>4.4.1</td>
<td>F.S.M. model of second-order autonomous recursive section</td>
<td>52</td>
</tr>
<tr>
<td>4.4.2</td>
<td>F.S.M. model of first-order recursive section</td>
<td>55</td>
</tr>
<tr>
<td>4.4.2.0</td>
<td>Decomposition results for D.F.2 with different feedback coefficient values</td>
<td>58</td>
</tr>
<tr>
<td>4.5</td>
<td>Discussion</td>
<td>66</td>
</tr>
<tr>
<td>4.6</td>
<td>Conclusions</td>
<td>70</td>
</tr>
</tbody>
</table>

## CHAPTER 5  Partition Structures of Stored-Logic Arithmetic Circuits

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0</td>
<td>Introduction</td>
<td>72</td>
</tr>
<tr>
<td>5.1</td>
<td>F.S.M. model of a general arithmetic circuit</td>
<td>72</td>
</tr>
<tr>
<td>5.2</td>
<td>Radix-$2^N$ adders</td>
<td>74</td>
</tr>
<tr>
<td>5.2.0</td>
<td>Example</td>
<td>75</td>
</tr>
<tr>
<td>5.2.0.0</td>
<td>S.P. partitions of radix-$2^3$ half-adder</td>
<td>75</td>
</tr>
<tr>
<td>5.2.1</td>
<td>The general modulo $2^N$</td>
<td>82</td>
</tr>
<tr>
<td>5.2.1.0</td>
<td>Generation of S.P. partitions</td>
<td>82</td>
</tr>
<tr>
<td>5.2.1.1</td>
<td>Loop-free realisation of adders modulo $2^N$</td>
<td>85</td>
</tr>
<tr>
<td>5.2.1.2</td>
<td>Memory storage reduction</td>
<td>87</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Generation of the carry digit</td>
<td>88</td>
</tr>
</tbody>
</table>
5.3 Radix-2\(^N\) parallel multipliers

5.3.0 Example

5.3.1 The general N × N bit multiplier

5.3.1.0 Example

5.3.2 Decomposition of the F.S.M. multiplier

5.3.3 Improved model of N-bit parallel multiplier

5.4 Conclusion

CHAPTER 6 NOVEL METHOD OF MODULO 2\(^N\) MULTIPLICATION USING CONSTRAINED OPERANDS

6.0 Introduction

6.1 Observations

6.2 Modulo 2\(^N\) multiplication using 'forced' operands and product correction

6.2.0 Example

6.3 Internal algebraic structure of reduced modulo 2\(^N\) multipliers

6.3.0 Example

6.3.1 The group under modulo 2\(^N\) reduced multiplication

6.3.2 Application of theoretical results

6.4 General comparison with the direct implementation of modulo 2\(^N\) multipliers

6.5 Conclusions

CHAPTER 7 DECOMPOSITION STRUCTURES OF MODULO M ADDERS AND MULTIPLIERS, AND OF A SIMPLIFIED MODEL OF A SECOND-ORDER DIGITAL FILTER

7.0 Introduction

7.1 Partition structure of modulo M adder

7.1.0 Generation of the basic S.P. partitions

7.1.1 General form of \(\pi_c\)

7.1.2 The partition lattice of the general \(\mod M\) adder

7.2 S.P. partitions for a \(\mod M\) multiplier

7.2.1 Sub-lattice of multiplier's S.P. partitions
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>7.3</strong> Decomposition structures of digital filters</td>
<td>157</td>
</tr>
<tr>
<td>7.3.0 Notation</td>
<td>157</td>
</tr>
<tr>
<td>7.3.1 Simplified models of non-recursive filters</td>
<td>158</td>
</tr>
<tr>
<td>7.3.2 Homomorphic images of ((DF)_M)</td>
<td>162</td>
</tr>
<tr>
<td>7.3.3 Parallel connection of ((DF)_b) and ((DF)_c)</td>
<td>166</td>
</tr>
<tr>
<td>7.3.4 Cascade decomposition structure of (\mod p^a) digital filter</td>
<td>173</td>
</tr>
<tr>
<td>7.3.4.0 Notation</td>
<td>173</td>
</tr>
<tr>
<td>7.3.4.1 Analysis of cascade structure of ((DF)_p^a)</td>
<td>174</td>
</tr>
<tr>
<td>7.3.5 Lattice of homomorphic images of ((DF)_M)</td>
<td>181</td>
</tr>
<tr>
<td><strong>7.4</strong> Conclusions</td>
<td>183</td>
</tr>
</tbody>
</table>

**CHAPTER 8 MODULAR PARTITIONING OF BASIC SECOND-ORDER DIGITAL FILTER**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.0 Introduction</td>
<td>184</td>
</tr>
<tr>
<td>8.1 General modular partition theory</td>
<td>185</td>
</tr>
<tr>
<td>8.1.0 Sequence elements represented as sequences</td>
<td>185</td>
</tr>
<tr>
<td>8.1.1 Extraction of a basic convolution unit</td>
<td>187</td>
</tr>
<tr>
<td>8.1.2 The primitive convolution cell</td>
<td>189</td>
</tr>
<tr>
<td>8.1.3 Effect of modular partitioning on frequency analysis</td>
<td>190</td>
</tr>
<tr>
<td>8.1.3.0 Frequency characteristics</td>
<td>190</td>
</tr>
<tr>
<td>8.1.3.1 Digit frequency response templates</td>
<td>191</td>
</tr>
<tr>
<td>8.2 Reprint of the article entitled &quot;A modular approach to the hardware implementation of digital filters&quot;.</td>
<td>194</td>
</tr>
<tr>
<td>8.3 Negative values of filter data</td>
<td>203</td>
</tr>
<tr>
<td>8.3.0 Constant bias of filter input</td>
<td>203</td>
</tr>
<tr>
<td>8.3.1 Distributed correction</td>
<td>205</td>
</tr>
<tr>
<td>8.3.2 Example</td>
<td>207</td>
</tr>
<tr>
<td>8.3.3 Circuit implementation of correction scheme</td>
<td>210</td>
</tr>
<tr>
<td><strong>8.4</strong> Conclusions</td>
<td>210</td>
</tr>
<tr>
<td>CHAPTER</td>
<td>9</td>
</tr>
<tr>
<td>---------</td>
<td>---</td>
</tr>
<tr>
<td>9.0</td>
<td>Introduction</td>
</tr>
<tr>
<td>9.1</td>
<td>General filter system</td>
</tr>
<tr>
<td>9.2</td>
<td>Functional and circuit description of filter sub-systems</td>
</tr>
<tr>
<td>9.2.0</td>
<td>p.R.O.M. module and data registers</td>
</tr>
<tr>
<td>9.2.0.0</td>
<td>Programming the p.R.O.M.</td>
</tr>
<tr>
<td>9.2.1</td>
<td>Accumulator</td>
</tr>
<tr>
<td>9.2.1.0</td>
<td>Circuit implementation of accumulator</td>
</tr>
<tr>
<td>9.2.2</td>
<td>Buffer logic</td>
</tr>
<tr>
<td>9.2.3</td>
<td>Clock and control unit</td>
</tr>
<tr>
<td>9.2.3.0</td>
<td>System clocks</td>
</tr>
<tr>
<td>9.2.3.1</td>
<td>Timing pulses</td>
</tr>
<tr>
<td>9.3</td>
<td>System performance</td>
</tr>
<tr>
<td>9.4</td>
<td>Conclusion</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>10</th>
<th>A UNIFIED FILTER REALISATION APPROACH USING PROGRAMMABLE STORED-LOGIC CONVOLUTION MODULES</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0</td>
<td>Introduction</td>
<td></td>
<td>227</td>
</tr>
<tr>
<td>10.1</td>
<td>Basic implementations of digit-convolution module</td>
<td></td>
<td>228</td>
</tr>
<tr>
<td>10.2</td>
<td>Novel implementation using complementary convolution module</td>
<td></td>
<td>228</td>
</tr>
<tr>
<td>10.3</td>
<td>The $Y \sim Y$ module in the parallel modular realisation</td>
<td></td>
<td>230</td>
</tr>
<tr>
<td>10.4</td>
<td>Consequence of the concept of complementary $Y \sim Y$ convolution module</td>
<td></td>
<td>231</td>
</tr>
<tr>
<td>10.5</td>
<td>Application to time-varying digital filters</td>
<td></td>
<td>233</td>
</tr>
<tr>
<td>10.6</td>
<td>General digital filter systems</td>
<td></td>
<td>233</td>
</tr>
<tr>
<td>10.6.0</td>
<td>General second-order section</td>
<td></td>
<td>234</td>
</tr>
<tr>
<td>10.6.1</td>
<td>General high-order filters</td>
<td></td>
<td>235</td>
</tr>
<tr>
<td>10.7</td>
<td>Recent proposals for the hardware implementation of digital filters</td>
<td></td>
<td>236</td>
</tr>
<tr>
<td>10.8</td>
<td>Conclusions</td>
<td></td>
<td>237</td>
</tr>
</tbody>
</table>
CHAPTER 11 REVIEW AND RECOMMENDATIONS

11.0 Introduction 238
11.1 Review of main results 238
11.2 Possible directions for development 240
11.3 Conclusions 242

REFERENCES
CHAPTER 1

INTRODUCTION

1.0 Introduction.

In this Chapter we describe the background of and the motivation for the research project, state and discuss the nature and scope of the investigation, and finally outline the organisation of the various chapters of the Thesis.

1.1 Background.

The trend in the field of communication and signal processing is towards the digital format for representing, transmitting and operating on signals, the increasing use of pulse-code modulation (P.C.M.) and delta-modulation (Δ - M.) being two familiar examples. This may be attributed mainly to the ever growing complexity and flexibility of digital computers and the rapid advance in the technology of medium and large-scale (M.S.I. and L.S.I.) integrated circuits.

In a general digital signal processor, one of the main components is the digital filter, which is basically a "black box" which processes a digital input signal according to a computational algorithm to produce a digital output having some specified characteristics. Among its many applications, a digital filter is widely used for waveform shaping and spectral analysis and synthesis of signals. Some of the advantages of a digital filter over its analogue counter-part are its arbitrary guaranteed accuracy,
predictable and reproducible performance, flexibility in parameter changes, and the possibility of time-multiplexing its major components.

From its early start in the mid-60's the theory in the analysis and design of digital filters is now well advanced and fairly complete, and comprehensive discussions on it may be found in many excellent text books and special issues that are now available. As such, in the next Chapter, we give only a brief review of the general theory and place more attention to discussing the problem of implementing digital filters in hardware. In contrast to its well developed theory, the practical aspect of digital filtering is far from satisfactory. Until a few years ago, with the exception of the classic paper by Jackson et al, most published papers concentrated only on the off-line simulation of digital filters on general-purpose computers.

The past few years, however, have seen a growing number of papers on the real-time hardware implementation of digital filters. The design techniques seem to differ from each other, but they invariably share a common philosophy, viz, a binary number representation is assumed for the arithmetic operations, and the hardware implementation is implicitly accepted as only an exercise in switching circuit techniques and combinational logic design.

1.2 Motivation for project.

Consequently, we feel that there is a theoretical gap between the analytical design of digital filters and their final realisations in real-time hardware, and a need for a systematic
realisation technique. If it is to be useful, any method developed should preferably be user-oriented and result in hardware structures that are modular and flexible for easy construction, testing, maintenance and reliable operation.

1.3 Design philosophy and problem formulation.

In our investigation, we decide to adopt the system and sub-system approach in the design philosophy, in which the hardware structures of digital filters are analysed from their input-output behaviour. Thus a macroscopic view is taken, rather than the conventional microscopic one in which logic elements and parts are put together to make up the overall filter circuit.

Furthermore, we feel that a powerful tool with which to analyse such filter systems is the concept of finite-state sequential machine (F.S.M.) or finite automata \(^{10,11,12}\), (also see Chapter 3), which is a useful model of the dynamics of discrete-parameter systems.

As it happens, in the period 1960-65, a structural theory of sequential machines which is generally unified and complete was developed by Hartmanis \(^{12}\). Using this theory, it is possible in general to decompose an F.S.M. into an interconnection of smaller and simpler sub-machines. The application of Hartmanis' theory to our filter systems is obviously attractive since a structural decomposition implies a modular system architecture. Furthermore, Howard \(^{13}\) showed that the decomposition theory is still applicable when an F.S.M. is realised as a table look-up unit implemented using a semiconductor read-only memory (R.O.M.), an L.S.I. device
that is rapidly becoming a popular alternative to random-logic.\textsuperscript{14,15}

Consequently, besides being an exploratory study in implementing digital filters using the systems approach in general, our research project investigates in particular the feasibility of realising a digital filter section as a table look-up unit and modelling its dynamics as a finite-state sequential machine in order to discover any structural property.

We term such a filter a \textit{stored-logic (S.L.) digital filter}, and consider its implementation using semiconductor memories.

1.4 \textbf{Scope of research and organisation of Thesis.}

The results of our initial investigations along the lines proposed are described in Chapter 4, and we report some success in simplifying the memory requirement of an S.L. digital filter. We are not able, however, to generalise the technique used here to filters having arbitrary coefficients, especially with recursive filters, due to non-linearities introduced by arithmetic round-off.

To gain further insight into the algebraic structure of these S.L. filters, we then apply the F.S.M. modelling technique to the arithmetic components which make up the filter. This is not reverting to the traditional approach since the subsequent analytical treatment, which is discussed in Chapter 5, is still on the systems level. Analysing arithmetic circuits based on modulo $2^N$ arithmetic (see ref. 16 for a discussion on modulo arithmetic), we derive interesting loop-free decomposition structures for adders and multipliers modulo $2^N$ which require considerably less memory storage in their implementation when compared with that required if a direct
table look-up is used.

We then extend the analysis to "real" arithmetic units, where one has to account for the carry output in the case of a general N-bit adder, and the double-length product of an N-bit multiplier.

In Chapter 6 we outline a novel approach to the implementation of modulo $2^N$ multipliers based on a transform which maps a sub-set of the multiplication table onto the Cartesian product of modulo 2 and $2^{N-1}$ adders. Although the results are not directly relevant to the synthesis of stored-logic filters, we have included the Chapter because we feel that it is interesting and useful in its own right.

In Chapter 7 the theory on the algebraic F.S.M. decomposition of general stored-logic modulo arithmetic units and digital filter sections is developed in which the concept of a lattice of partitions on machine states (see Chapter 3) plays a central role. We show that for a general modulo M adder, its decomposition structure can be completely described. Although we are unable to do the same for the corresponding modulo M multiplier, we have managed to describe completely one possible sub-structure.

The next three chapters, 8.9 and 10, take on a more practical tone. In Chapter 8, an attractive and novel modular hardware architecture for a second-order digital filter section is introduced. This uses the concept of digit slicing, which leads to what we term a sub-filter module. We show that a digital filter section may be realised as a regular interconnection of such modules, which are all identical in structure.
Using this technique we have also constructed a practical prototype 8-bit second-order digital filter section, in which the sub-filter module is implemented using a semiconductor programmable and erasable read-only memory (p.R.O.M.). Details of the circuit construction and testing are documented in Chapter 9. Useful indications are obtained on the tradeoff between hardware complexity and processing speed.

Following this, we propose in Chapter 10 two ways with which the technique of digit-slicing and the concept of stored-logic sub-filter modules can be successfully incorporated into a general system architecture to achieve flexible and relatively inexpensive real-time digital filtering. In this chapter we also discuss briefly the state-of-the-art of practical digital filters and signal processors and suggest probable trends.

Finally, we conclude the Thesis with Chapter 11, in which the investigation that has been carried out is reviewed.

1.5 Conclusions.

The research project is an attempt to provide a theoretical framework for the methodical implementation of real-time digital filters. The problem is approached in a novel way using a systems design philosophy in general, and the concept of finite automata in particular.
2.0 Introduction.

The theory of digital filtering is briefly reviewed in this chapter, and we discuss the problems involved in implementing digital filter hardware to process real-time signals. We also survey the different approaches to the problem that have been proposed in the literature.

2.1 Descriptions of a general digital filter.

A digital filter is basically a computational algorithm by which an input number sequence \( \{x_n\} \) is transformed into an output number sequence \( \{y_n\} \). When used in a digital signal processing system, as shown in Fig. 2.0, \( \{x_n\} \) is the time and amplitude quantised version of an analogue signal input. If so required, \( \{y_n\} \) may be converted back into the analogue form.

The filter algorithm is the following linear difference equation,

\[
y_n = \sum_{k=0}^{N} a_k x_{n-k} - \sum_{k=1}^{N} b_k y_{n-k}
\]

\( \text{...(2.0)} \)

where \( a_k \)'s and \( b_k \)'s are termed the filter coefficients.

The filter described by equation (2.0) is known as a general recursive filter. In many cases, the output \( y_n \) is explicitly determined only by the present and past input values, i.e. all \( b_k \)'s = 0. The corresponding filter is then known as a non-recursive one.
Fig. 2.0  Block representation of a digital signal processing system.

Fig. 2.1  The direct form of a general digital filter.
In spectral analysis, due to the convenience of algebraic manipulation, a digital filter is alternatively described by its z-transform transfer function $H(z)$, where

$$H(z) = \frac{\sum_{k=0}^{N} a_k z^{-k}}{1 + \sum_{k=1}^{N} b_k z^{-k}} \quad \ldots \quad (2.1)$$

where $z^{-1}$ is the unit delay operator.

A canonical circuit realisation of (2.1) is shown in Fig. 2.1, known as the direct form. Due to accuracy requirements, the following cascade and parallel forms are preferred, i.e.

$$H(z) = a_0 \prod_{i=1}^{M} \frac{1 + a_{1i} z^{-1} + a_{2i} z^{-2}}{1 + b_{1i} z^{-1} + b_{2i} z^{-2}} \quad \ldots \quad (2.2)$$

and

$$H(z) = \gamma_0 + \sum_{i=1}^{M} \frac{\gamma_{0i} + \gamma_{1i} z^{-1}}{1 + \beta_{1i} z^{-1} + \beta_{2i} z^{-2}} \quad \ldots \quad (2.3)$$

where $M$ is the integer part of $(N+1)/2$, and $\gamma_0 = a_0/b_n$.

The corresponding circuit realisations are shown in Figs. 2.2(a) and (b), in which the basic building block is the second-order or biquadratic section, which is shown in Fig. 2.3 and described by the following relationship, i.e.,

$$y_n = \sum_{k=0}^{2} a_k x_{n-k} - \sum_{k=1}^{2} b_k y_{n-k} \quad \ldots \quad (2.4)$$
Fig. 2.2 The cascade form (a) and the parallel form (b) of a digital filter.
Fig. 2.3 The general second-order section.
2.1.0 Dynamics of digital filters.

The operational and functional behaviour of a digital filter can be analysed either in the time or frequency domain.

In the former, we use the impulse response $h(n)$, which is the filter output response to a discrete-time impulse at $k = 0$ (a digital impulse at $k = k_0$ is a signal $x(k)$ such that $x(k) = 1$ when $k = k_0$ and $x(k) = 0$ when $k \neq k_0$).

If $h(n) = 0$ for $N_1 < n < N_2$, with $N_1 > N_2$, the associated filter is called a finite impulse response (FIR) filter. An infinite impulse response (IIR) filter is one in which either $N_1 = \infty$ or $N_2 = -\infty$ or both.

Given an input sequence $g(n)$ and the filter impulse response $h(n)$, the output $f(n)$ is obtained by the discrete-time convolution operation defined by

$$f(n) = \sum_{k=-\infty}^{\infty} h_k g_{n-k} \quad \text{(2.5)}$$

Alternatively, a filter may be described by its frequency response, which is the value of $H(z)$ when evaluated on the unit circle, i.e. $|z| = 1$, in the complex $z$-plane. When the frequency response is expressed in polar form, its magnitude and its angle as a function of frequency is called the amplitude and the phase response respectively.

Other aspects of digital spectral analysis such as Discrete Fourier Transform (D.F.T.), and the algorithm for its efficient computation called the fast Fourier Transform (F.F.T.) may be found in the recommended references.
2.2 Applications and advantages.

Digital filters are extensively used in data reduction and system simulation experiments, and as integral parts of communication or signal processing systems. Specific applications include character extraction in speech processing and biomedical engineering, the study of new signal processing systems via computer simulation, e.g. vocoders, speech codecs, bandwidth compression schemes, the removal of interference noise and the compensation for perturbation in the transmission channels of communication systems.

A digital filter has the following advantages over its analogue counterpart;

(a) Theoretically, it can be designed to an arbitrarily high accuracy which is reproducible due to the absence of drift and component tolerance.

(b) It is very flexible as the overall performance can be modified by simply altering the filter coefficients.

(c) The time-multiplexing of the main hardware units is possible, leading to simple filter banks.

(d) There is no problem of impedance matching and also no restriction on critical frequencies.

(g) Many practical signals today are already in digital form anyway.

(h) Its configuration is not highly cross-connected and is thus suitable for integrated circuit technology, which is currently developing at a tremendous rate.

On the other hand digital filtering has its special problems
in design and implementation. These will be discussed in the following sections.

2.3 Design and realisation.

To the system designer, his problem is basically to produce a realisation which approximates as "closely" as possible a given specified filter response (time, frequency, group-delay etc.) in a prescribed manner. This realisation may be an off-line software routine or a real-time hardware implementation.

There are three distinct stages that he has to go through, viz., that of

(a) mathematical design assuming infinite precision arithmetic,
(b) circuit or configuration design accounting for the effects of finite register lengths, and
(c) real-time hardware architecture and device implementation.

2.3.0 Mathematical design.

In general, the "filter design problem" is one in mathematical approximation and consists simply of finding the values of the coefficients \( a_k \)'s and \( b_k \)'s such that the response of the corresponding filter approximates, in a prescribed manner, a desired characteristic. The theory on the design techniques is well developed and excellent documentation of established and proven methods may be found in the literature (e.g. References 1,2,17,18). Also new designs are constantly being published. As such, we will mention only briefly the main design procedures for both FIR and IIR filters.
2.3.0.0 Design techniques for FIR and IIR Filters.

There are three well known classes of design methods.

The first is the window method, which is based on the expansion, in a Fourier series form, of the periodic (in frequency) frequency response \( H(e^{j\omega}) \) of any digital filter, i.e.,

\[
H(e^{j\omega}) = \sum_{n=-\infty}^{\infty} h(n)e^{-j\omega n}
\]

where \( h(n) \) are the Fourier coefficients. It is also easily shown that \( h(n) \) is identical to the impulse response of a digital filter. To obtain a realisable FIR filter, a finite weighting sequence \( w(n) \) is used to modify \( h(n) \) to control the convergence of the Fourier series. Some well known windows as these \( w(n) \)'s are called, are the rectangular, "generalised" Hamming, and Kaiser windows.

The second method is that of frequency sampling in which an FIR filter is expressed in term of its D.F.T. coefficients. The continuous frequency response is thus approximated by sampling, in frequency, at \( N \) equidistant points around the unit circle. The continuous frequency response is then evaluated as an interpolation of the sampled frequency response.

In the third method, the design problem is regarded as a Chebyshev approximation problem and consists of minimising the maximum absolute value of a weighted error of approximation \( E(e^{j\omega}) \) (see page 126 of Ref.1 for its definition).

For the IIR filters, there are two main classes of design techniques. In the first class, one first designs an appropriate continuous time analogue filter. The design obtained is then
digitised to determine its digital equivalent using procedures like the mapping of differentials to finite differences, the impulse invariant, the bilinear transform and the matched z-transform techniques. The second class is the open form approach using modern optimisation algorithms, like the minimum mean square and minimum absolute error methods, equiripple techniques and time domain optimisation.

2.3.1 Effects of finite-length registers.

In a theoretical realisation it is assumed that infinite precision arithmetic is used. In practical realisations, however, (especially with special-purpose implementations), data words can only be stored in registers having finite lengths. Thus the filter data, coefficients and the results of intermediate operations have to be either truncated or rounded-off. These quantisation effects affect the overall filter performance in various ways, depending on the type of arithmetic used, the type of quantisation and the exact filter structure. (Comprehensive discussions on these effects are given in the review papers by Oppenheim and Weinstein¹⁹ and Liu²⁰).

The first of these effects is the error introduced as a result of the A/D conversion of the filter input. This quantisation effect, however, is not usually regarded by digital filter designers as an integral part of filter design.

The second effect is when the filter coefficients are quantised, leading to the restriction of the possible values of the poles and zeros of the filter transfer function to a finite set. Consequently the actual filter response will differ slightly from the theoretically
derived one. Some common approaches to the problem consist of computing the frequency response directly using the quantised coefficients, performing an optimised search over the grid of allowed pole/zero positions around the ideal positions, and to find general structures which are less sensitive to coefficient inaccuracies.

The quantisation of the results of the arithmetic operations of multiplications and additions is the third effect of finite register lengths. Its analysis depends on whether truncation or round-off is used, whether we implement the operations with fixed-point or floating point arithmetic, and on whether we represent negative numbers in the sign-magnitude, 1's or 2's complement form. In many situations, the rounding effect at each multiplier is statistically modelled as a discrete stationary white-noise source uniformly distributed in amplitude between $\pm (1/2)2^{-b}$, $b$ being the product register's bit length. Also each source has a transfer function to the output.

For recursive filter structures, the quantisation of the multiplicative products produce stable periodic or non-zero constant outputs when the inputs are zero or constant. These outputs are called small-scale limit cycles.

Another problem is when the result of some arithmetic operations overflows and falls outside the permitted set of representable values resulting in an incorrect in-range value. When this occurs in the feedback loops of certain second-order sections, stable and persistent full-scale oscillations result. They are known as large-scale limit cycles. Methods exist with which one may determine
the scale factors of signal levels at certain points in the filter structure to prevent overflow and still maintain a maximum signal/round-off noise ratio.

2.3.2 Considerations in the real-time hardware implementation.

Although conceptually one simply interconnects adder, multiplier and storage units in order to mechanise the filter algorithm, in practice one is confronted with a bewildering array of factors and constraints in the choice of system or circuit structure and component and device technology. To achieve an efficient and economical system, the designer must consider initial costs, hardware complexity with respect to construction, testing and maintenance, power dissipation, space requirements, system modularity and flexibility etc. All these factors depend on specific needs and applications.

Assuming that the would-be designer has obtained his filter coefficients, and an estimate of the bits required for the input, coefficient and internal data words, he must also realise that in real-time digital filtering, the filtering algorithm must be computed within the sampling period $T$ of the input signal, with the maximum allowable value of $T$ depending on the bandwidth or Nyquist frequency (see Ref. 1) of the signal. Operational speed thus has to be balanced by system cost and complexity.

To obtain the system structure one has to decide on the number representation and the type of arithmetic to use. Floating-point arithmetic gives a larger dynamic range than that of fixed-point, but it requires a more complex hardware due to the need to align mantissas. The circuit complexity also depends on the particular
representation of negative numbers. One must also remember that although the use of 2's complement arithmetic makes additions and subtractions easy, multiplication is much more convenient using the sign-magnitude representation.

One must also define the basic functional units making up the adders, multipliers and data stores, their interconnections, and their processing modes, i.e. word parallel or bit serial processing (see Lewin \textsuperscript{21} for more details on number representations and arithmetic hardware). Basic units may also be time-multiplexed, and one could also increase system throughput by the incorporation of pipelining.\textsuperscript{22,23}

The basic adding unit is the full-adder (F.A.)\textsuperscript{21}; a single one is used in serial addition, and an N-bit parallel addition may be achieved by connecting N F.A.'s in cascade. Fast additions employ the familiar carry look-ahead\textsuperscript{21} technique.

Multipliers are the most important, complex and expensive units. They range in structures from the simplest shift-and-add ones, through the serial-parallel varieties, to the fast two-dimensional array multipliers. An ultra-fast array combines the carry-save technique with a "tree" arrangement of adder rows. (See Chapter 8 of Ref. 1 and also Refs. 24 & 25).

Data are stored in either bistable shift registers and/or M.S.I. and L.S.I. memories. These memories are either static or dynamic which requires refreshing\textsuperscript{26}, and are further classified into read-only memories (R.O.M's) and read-and-write memories (commonly referred to as R.A.M's which, strictly speaking, can be also taken to mean random access memories).
Apart from these main units, extra circuitry is required for overflow detection and correction, intersection scaling, data quantisation and system control.

For a given architecture, an appropriate device technology must be matched to it. Each particular technology is characterised by

(a) the physical size of the basic device, e.g. a logic gate,
(b) its power dissipation, and
(c) its switching speed.

(a) and (b) usually determine the scale of integration, i.e. the number of devices per chip, while the ratio of (b) to (c) is roughly constant for a given technology and is often used as a figure of merit.

At present two proven technologies are the bipolar saturated transistor-transistor logic (T.T.L.), the linear emitter-coupled logic (E.C.L.), and the unipolar metal-oxide semiconductor (M.O.S.) technology, having typical power-delay values of $(10 \text{ mW} - 15 \text{ nS})$, $(60 \text{ mW} - 1 \text{ nS})$ and $(0.2 \text{ mW} - 300 \text{ nS})$ respectively. In general bipolar circuits have achieved much higher speeds while M.O.S. chips have attained a much higher degree of circuit integration.

Among the newer technologies are the use of sapphire substrates and the bipolar integrated-injection logic (I.I.L.) which promises a high packing density.

Lastly, the trend in digital system design is rapidly moving from the use of discrete gates and simple logic packages to that of medium-scale (M.S.I.) and large-scale integrated (L.S.I.) techniques.
2.3.3 **Existing hardware design approaches.**

In contrast to the mathematical design theory, there is no systematic design technique for the real-time hardware implementation of digital filters. There are as many hardware structures as there are authors, and the continuing rapid change in integrated circuit technology makes the problem of the device implementation of these structures a dynamic one.

In this section we discuss the major classes of design approach. In the author's opinion, the first three are becoming established designs due to their efficiency, simplicity and modularity, and have attracted the attention and enthusiasm of a host of workers in the field.

The first design approach was proposed in the classic 1968 paper by Jackson et al. The corresponding filter structure uses serial arithmetic and features a sign-magnitude serial-data and parallel coefficient multiplier as shown in Fig. 2.4(a). The problem of excessive propagation delays and the need to quantise double-length products to single-length registers was solved by efficient pipelining and simple logic. A typical adder cell of the modified multiplier is shown in Fig. 2.4(b). Jackson et al. also presented a simple multiplexing method for multichannel or multifunction processing, using a R.O.M. to store the coefficients. The scheme was used in implementing an all-digital touch-tone receiver consisting of high-pass, low-pass, band-stop and band-pass filters of various orders from the 1\(^{st}\) to the 6\(^{th}\), using multiplexed first and second-order sections. The sampling rate is 10K samples/sec., input quantisation is 7 bits, and 40 serial adders and 400 bits of shift-
register storage were used.

Of a more recent vintage is the design proposed by Croisier et al., and further analysed and developed by Little and Peled and Liu, which promises high speed, fairly low power and low package count. The design substitutes a table look-up R.O.M. for the bit-wise multiplication of the filter coefficients by the data, with the filter output obtained by the operation of adding and shifting. A second-order section implemented in this way is shown in Fig. 2.5 and requires a 32 x 8-bit R.O.M. This basic circuit has a 20 MHz bit-rate, package count of 20 I.C's and dissipates 9.6 W. For a 12-bit input this section can handle up to 800 kHz bandwidth signals. A parallel version of the technique requires 60 I.C's, consumes 24 W, and allows a signal bandwidth of 10 MHz. A general comparison with Jackson's approach is shown in Table 2.1.

Lockhart took a different approach by combining delta-modulation encoding (instead of P.C.M.) with digital filtering. The versions described by Croisier and Liu use R.O.M's for their mechanisation. The design required simple and inexpensive hardware, is particularly appropriate to applications involving analogue-digital interfacing, and has found favour with researchers working on speech signals.

We now mention briefly the work by other authors. In 1967, Sypherd used R.O.M's for multiplication and multiple-input additions. Gabel described a simple architecture using a time-shared multiplier/adder unit in which the filter coefficients are represented in a simplified floating-point form. Trân-Thông and Liu used differential pulse-code modulation (D.P.C.M.) for the signal encoding and evolved a design for a D.P.C.M. filter. A look-up table
Fig. 2.4  Jackson's basic serial-parallel multiplier (a) and pipeline cell (b)

Fig. 2.5  Read-only memory second-order section (after Croisier et al).
<table>
<thead>
<tr>
<th>Filter type (12-bit word-lengths)</th>
<th>Bede and Liu</th>
<th>Jackson et al.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No. of I.C's</td>
<td>Power dissipation (W)</td>
</tr>
<tr>
<td>8th-order, parallel, 1 MHz word-rate (w.r.)</td>
<td>72</td>
<td>28</td>
</tr>
<tr>
<td>8th-order, cascade, 250 kHz w.r.</td>
<td>33, memory</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>Size = 128 x 8 bits</td>
<td></td>
</tr>
<tr>
<td>2n-order, multiplexed, 128 channels, each 8 kHz w.r.</td>
<td>18, memory</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Size = 512 x 8 bits</td>
<td></td>
</tr>
<tr>
<td>10th-order, mux., 96 ch., 8 kHz, w.r.</td>
<td>54</td>
<td>22</td>
</tr>
</tbody>
</table>

Table 2.1. Hardware and performance comparison of Bede and Liu, and Jackson et al. methods.
technique using R.O.M's was proposed by Nussbaumer in which he modified the familiar "quarter squares" multiplication method to reduce the overall number of additions and squarings.

Another attractive approach is to replace multiplications by simple shifting using multiplexers by restricting the values of the filter coefficients to only integer powers of two or zero. The design leads to simple and very fast filters, e.g. Tomozawa used the approach to process real-time colour television signals. Van Gerwen et al. published an excellent theoretical and experimental study of this approach, and introduced a filter consisting of a transversal part and a simple recursive network.

Other approaches are the bit-level counting technique of Zohar's (which is still a conceptual entity), and the use of logarithmic arithmetic as suggested by Hall et al. and Kingsbury and Rayner. As far as the author knows, no hardware details of the latter technique have been published.

Finally, custom-design digital filter chips and packages are now slowly making their appearances commercially, e.g. the Pye TMC Ltd.'s pM.O.S. dual second-order filter chip and the 3-chip M.S.I./L.S.I. digital filter set by Advanced Micro Devices Inc., which employs low-power Schottky bipolar technology.

2.3.4 Conclusion.

We have reviewed briefly the theory and design of digital filters and the problems involved in their real-time hardware implementation, and also surveyed the state-of-the-art of the existing hardware design approaches.
CHAPTER 3

ELEMENTARY STRUCTURE THEORY
OF
FINITE-STATE SEQUENTIAL MACHINES

3.0 Introduction.

The theory of finite automata or finite-state sequential machines (F.S.M's) is a special case of general systems theory in which the input, state and output variables only assume discrete values, and the functional relationship between them is described by abstract algebra. An F.S.M. is a useful mathematical model for digital computers, processors, the behaviour of nerve networks, language structures and information-transmission systems to name a few.

The "structure theory" for F.S.M's concerns the realisation of an F.S.M. from a set of smaller component sub-machines, the interconnection and the "information" flow between these components. The theory provides a direct link between algebraic relationships and physical realisations of machines.

In the following sections we introduce briefly the basic ideas, concepts, terminology and results of this theory. The books by Booth, Kohavi and Hartmanis are excellent introductory texts.

3.1 Descriptions of F.S.M's.

Definition 3.0. A Mealy type sequential machine M is a quintuple \((S, I, O, \delta, \lambda)\) where \(S, I, O\) are finite nonempty sets
of states, inputs and outputs respectively, and \( \delta, \lambda \) are the transition (next state) and the output functions given by

\[
\delta : S \times I \to S \quad \text{and} \quad \lambda : S \times I \to O.
\]

When the output is a function of the present state only, i.e. \( \lambda : S \to O \), then the machine is known as a Moore type. When, in many cases, we are not interested in the output, the corresponding machine is called a state machine defined by the triplet \((S, I, \delta)\). The block representation of the Mealy type F.S.M. is shown in Fig. 3.0. The behaviour of an F.S.M. is commonly represented by a flow table or a state graph. Each row of the flow table represents a machine state, while the columns correspond to the inputs. The table entries indicate each state and output transition. The nodes of the corresponding state graph represent the states, while the arrow between nodes \( s_1 \) and \( s_2 \), labelled by the ordered pair \((x, o)\), \( x \in I, o \in O \), indicates that \( \delta(s_1, x) = s_2 \) and \( \lambda(s_1, x) = o \).

These two representations are illustrated in Figs. 3.1(a) and (b) for the Mealy machine \( M = (P, Q, R, (a, b), (0, 1), \delta, \lambda) \).

In elementary machine decompositions, an important concept, which relates the behaviour of two machines, is that of machine homomorphism, which is an operation-preserving transformation.

Definition 3.1. The sequential machine \( M' = (S', I', O'; \delta', \lambda') \) is a homomorphic image of the machine \( M = (S, I, O, \delta, \lambda) \) iff there exist three onto mappings;

\[
h_1 : S \to S', \quad h_2 : I \to I' \quad \text{and} \quad h_3 : O \to O'
\]

such that
Fig. 3.0. Block representation of a finite-state sequential machine.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>(a)</th>
<th>(b)</th>
<th>(\text{Present States})</th>
<th>(\text{Next States})</th>
<th>(\text{Output})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P)</td>
<td>(Q)</td>
<td>(P)</td>
<td>0</td>
<td>0</td>
<td>(b/0)</td>
</tr>
<tr>
<td>(Q)</td>
<td>(R)</td>
<td>(R)</td>
<td>0</td>
<td>1</td>
<td>(a/0)</td>
</tr>
<tr>
<td>(R)</td>
<td>(P)</td>
<td>(Q)</td>
<td>1</td>
<td>0</td>
<td>(a/1)</td>
</tr>
</tbody>
</table>

Fig. 3.1. Flow table (a) and state graph (b) representations of an F.S.M.
The triple \((h_1, h_2, h_3)\) of mappings is referred to as a homomorphism of \(M\) onto \(M'\).

**Definition 3.2.** A state machine \(M' = (S;I;\delta')\) is a homomorphic image of \(M\) iff there exist two onto mappings;

\[ h_1 : S \rightarrow S', \ h_2 : I \rightarrow I' \]  

such that

\[ h_1[\delta(s,a)] = \delta'[h_1(s), h_2(a)] \]

When \(h_2\) and \(h_3\) are identity mappings, the homomorphism is called a state homomorphism. When two machines are identical except for a renaming of the states, inputs and outputs, we have an isomorphism between them.

**Definition 3.3** Two machines \(M = (S,I,O,\delta,\lambda)\) and \(M' = (S';I';O';\delta';\lambda')\) are isomorphic iff there exist three one-to-one mappings;

\[ f_1 : S \rightarrow S', \ f_2 : I \rightarrow I' \text{ and } f_3 : O \rightarrow O' \]  

such that

\[ f_1[\delta(s,x)] = \delta'[f_1(s), f_2(x)] \]
\[ f_3[\lambda(s,x)] = \lambda'[f_1(s), f_2(x)] \]

### 3.2 Interconnections of F.S.M's.

When decomposing a machine \(M\) into, or realising it from its component sub-machines, it is important to know the possible ways of interconnecting them.
Definition 3.4(a). The serial connection of

\[ M_1 = (S_1, I_1, 0_1, \delta_1, \lambda_1) \] and \[ M_2 = (S_2, I_2, 0_2, \delta_2, \lambda_2) \]

for which \[ 0_1 = I_2, \] is the machine \( M \), denoted by \( M_1 + M_2 \), where

\[ M = M_1 + M_2 = (S_1 \times S_2, I_1, 0_2, \delta, \lambda) \]

such that

\[ \delta\left((s_1, s_2), x\right) = \begin{cases} \delta_1(s_1, x), & \delta_2\left(s_2, \lambda_1(s_1, x)\right) \end{cases} \]

and

\[ \lambda\left((s_1, s_2), x\right) = \lambda_2\left(s_2, \lambda_1(s_1, x)\right). \]

The serial connection for state machines, however, is slightly different.

Definition 3.4(b). Given two state machines

\[ M_1 = (S_1, I_1, \delta_1), \quad M_2 = (S_2, I_2, \delta_2) \]

with \( I_2 = S_1 \times I_1 \), and an output set \( 0 \) and an output function \( \lambda : S_1 \times S_2 \times I_1 \to 0 \), then the serial connection of \( M_1 \) and \( M_2 \) is the machine

\[ M = (S_1 \times S_2, I_1, 0, \delta, \lambda) \]

where

\[ \delta\left((s_1, s_2), x\right) = \begin{cases} \delta_1(s_1, x), & \delta_2\left(s_2, (s_1, x)\right) \end{cases} \]

and

\[ \lambda : S_1 \times S_2 \times I_1 \to 0. \]

These two different serial connections are shown in the schematic diagrams in Figs. 3.2(a) and (b).

Definition 3.4(b). The parallel connection of \( M_1 \) and \( M_2 \) is the machine \( M = M_1 \times M_2 = (S_1 \times S_2, I_1 \times I_2, 0_1 \times 0_2, \delta, \lambda) \), where
Fig. 3.2. Serial connections of (a) general F.S.M's and (b) state machines.

Fig. 3.3. An F.S.M. realised with binary variables.
\[ \delta[(s_1, s_2), (x_1, x_2)] = [\delta_1(s_1, x_1), \delta_2(s_2, x_2)] \]

and

\[ \lambda[(s_1, s_2), (x_1, x_2)] = [\lambda_1(s_1, x_1), \lambda_2(s_2, x_2)] \]

### 3.3 Problem areas in F.S.M. realisation.

Two major problems in the realisation and physical implementation of F.S.M's are those of **state reduction** and **state assignment**.

The former concerns the concept of equivalence between the states of machines, and also between two machines.

**Definition 3.5.** For two machines \( M_1 = (S_1, I, O, \delta_1, \lambda_1) \) and \( M_2 = (S_2, I, O, \delta_2, \lambda_2) \) having the same input and output alphabets, \( s_1 \in S_1 \) and \( s_2 \in S_2 \) are said to be equivalent iff

\[ \overline{\lambda_1}(s_1, \overline{x}) = \overline{\lambda_2}(s_2, \overline{x}) \]

where (for Mealy type) \( \overline{x} \) is any finite non-null input sequence and \( \overline{\lambda_1}, \overline{\lambda_2} \) are the extended output functions of \( M_1 \) and \( M_2 \) respectively (see pp. 22-23 of Ref. 12).

**Definition 3.6.** Two machines of the same type, \( M_1 \) and \( M_2 \), are equivalent iff each \( s_1 \) in \( S_1 \) has an equivalent state \( s_2 \) in \( S_2 \) and vice versa.

**Definition 3.7.** A machine \( M \) is reduced iff state \( s_1 \) equivalent to state \( s_2 \) implies that \( s_1 = s_2 \).

It is easily shown that among all the machines equivalent to a given machine \( M \), there exists a unique equivalent reduced machine \( M_R \) which has the minimum number of states. Basically, for any
given finite input sequence, M and \( M_R \) will give the same output sequence. While there are standard techniques in the minimisation of machine states, it is also possible to apply structure theory to the state reduction problem as will be described in Section 3.6.

The second problem arises because in practice the inputs, states and outputs of a machine M are invariably represented by binary variables. Thus we may write

\[
S = \{(y_1, \ldots, y_n)\}, \text{ the set of all } n\text{-tuples on } \{0,1\},
\]

\[
I = \{(x_1, \ldots, x_m)\}
\]

and \( O = \{(z_1, \ldots, z_r)\} \).

Also, each state and each output binary variable is a function of \( \{y_1, \ldots, y_n, x_1, \ldots, x_m\} \). The block diagram of an F.S.M. expressed in this manner is shown in Fig. 3.3.

The state assignment problem is the selection of "desirable" binary codes to represent the internal machine states. Although this usually means the use of fewest number of components, e.g. logic gates, the relevant criteria are most often determined by the dynamics of technology. In concept, however, it is reasonable to assume that we can obtain economical state assignments and simplify the logic circuits in the physical implementation if we can reduce the number of present-state and input variables on which the next-state variables depend.

Since the structure theory for F.S.M's deals with the general understanding of functional dependence and the realisations of machines from smaller components, it may be regarded as an approach to the state assignment problem. (Other approaches are listed in page 36 of Ref. 12).
3.4 **Basic algebraic concepts.**

Two key mathematical ideas that are important tools in machine decompositions are the concept of partitions on a set, and that of an algebraic lattice.

**Definition 3.8.** A partition $\pi$ on $S$ is a collection of disjoint subsets of $S$ whose set union is $S$, i.e. $\pi = \{B_\alpha\}$ such that

$$B_\alpha \cap B_\beta = \emptyset \text{ for } \alpha \neq \beta, \text{ and } \cup \{B_\alpha\} = S.$$  

The $B_\alpha$'s are called blocks of $\pi$ and the block containing $s$ is written as $B_\pi(s)$. Also we write $s \equiv t(\pi)$ iff $s$ and $t$ are contained in the same block of $\pi$.

Partitions may be combined by the "product" or "+", and the "sum" or "*" operations as follows.

(i) $\pi_1 \cdot \pi_2$ is the partition on $S$ such that $s \equiv t(\pi_1 \cdot \pi_2)$ iff $s \equiv t(\pi_1)$ and $s \equiv t(\pi_2)$.

(ii) $\pi_1 + \pi_2$ is the partition such that $s \equiv t(\pi_1 + \pi_2)$ iff there exists a sequence in $S$, $s = s_0, s_1, s_2, \ldots, s_n = t$, for which either $s_i \equiv s_{i+1}(\pi_1)$ or $s_i \equiv s_{i+1}(\pi_2)$.

As an example, let $S = \{A, B, C, D, E, F, G, H, I\}$, and

$\pi_1 = \{\overline{A}, \overline{B}; \overline{C, D}; \overline{E, F}; G, H, I\}$ and $\pi_2 = \{\overline{A, F}; \overline{B, C}; \overline{D, E}; G, H; I\}$.

Then we have,

$\pi_1 \cdot \pi_2 = \{\overline{A}; \overline{B}; \overline{C}; \overline{D}; \overline{E}; \overline{F}; G, H; I\}$ and

$\pi_1 + \pi_2 = \{\overline{A, B, C, D, E, F}; G, H, I\}$.

Partitions may also be ordered by the "larger than or equal", 
i.e. ≤ relation. We say that \( \pi_a \leq \pi_b \) iff every block of \( \pi_a \) is contained in a block of \( \pi_b \), i.e. \( \pi_a \cdot \pi_b = \pi_a \) and \( \pi_a + \pi_b = \pi_b \).

**Definition 3.9.** A lattice is a partially ordered set \( L = (S, \leq) \) in which every pair of elements have a least upper bound (l.u.b.) and a greatest lower bound (g.l.b.) (See pp. 6-7 of Ref. 12, or Herstein for definitions of the underlined terms).

Alternatively, a lattice \( L \) is defined as a triplet \( L = (S, \cdot, +) \) where "\( \cdot \)" and "\( + \)" are binary operations satisfying certain postulates (page 7 of Ref. 12).

The set of all partitions on a set, for example, is a lattice and that g.l.b. \( (\pi_1, \pi_2) = \pi_1 \cdot \pi_2 \), and l.u.b. \( (\pi_1, \pi_2) = \pi_1 + \pi_2 \).

**Definition 3.10.** If \( L = (S, \cdot, +) \) is a lattice, and \( T \subseteq S, T \neq \emptyset \), then \( L' = (T, \cdot, +) \) is a sub-lattice of \( L \) iff \( x \) and \( y \in T \) implies that \( x \cdot y \) and \( x + y \in T \).

**Definition 3.11.** A lattice \( L_1 = (S_1, \cdot, +) \) is homomorphic to \( L_2 = (S_2, \cdot, +) \) iff there exists an onto mapping \( h : S_1 \rightarrow S_2 \), such that \( h(x \cdot y) = h(x) \cdot h(y) \) and \( h(x + y) = h(x) + h(y) \).

Thus, \( L_2 \) is very simply a "coarse" version of \( L_1 \). If \( h \) is a one-to-one onto mapping, then we say that the two lattices \( L_1 \) and \( L_2 \) are isomorphic.

### 3.5 Structural decompositions of F.S.M's.

The basis of machine decompositions is the modification of
the homomorphism concept to involve only one machine.

Definition 3.12. A partition $\pi$ on the set of states of the machine $M = (S, I, O, \delta, \lambda)$ has the substitution property (S.P.) iff $s \equiv t(\pi)$ implies that

$$\delta(s, a) = \delta(t, a)(\pi)$$

for all $a$ in $I$.

In other words, for each input, blocks of $\pi$, defined as above, will be mapped into blocks of $\pi$. These blocks may now be regarded as the states of a new machine defined by $\pi$ and $M$.

Definition 3.13. Let $\pi$ be an S.P. partition on the set of states of $M$. Then the $\pi$-image of $M$ is the state machine $M_\pi = (\{B_\pi\}, I, \delta_\pi)$ with

$$\delta_\pi(B_\pi, x) = B_\pi' \quad \text{iff} \quad \delta(B_\pi, x) \subseteq B_\pi'.$$

It is easily shown that there is a one-to-one correspondence between state homomorphisms and S.P. partitions. Also, if $\pi_1$ and $\pi_2$ are S.P. partitions, so are the partitions $\pi_1 \cdot \pi_2$ and $\pi_1 + \pi_2$.

We will use the following theorem quite often.

Theorem 3.0. The set of all S.P. partitions on the set of states of an F.S.M. $M$ forms a lattice $L_M$, under the natural partition ordering. Also $L_M$ contains the trivial partitions $\pi(0)$ and $\pi(1)$.

Proof. (See page 41 of Ref. 12).
The lattice $L_M$ is useful because it displays visually all the important multiple series-parallel state behaviour realisations, and because algebraic lattice properties are reflected in machine properties and vice versa.

A general procedure in finding all the S.P. partitions of a machine consists of two steps;

(i) For every pair of states $s$ and $t$, compute the smallest S.P. partition $\pi_{s,t}$ which identifies the pair.

(ii) Find all possible sums of the $\pi_{s,t}$'s. These sums constitute all the S.P. partitions.

Details of this procedure may be found in the recommended texts.

To consolidate the ideas we have discussed so far, we consider the machine $M$ shown in Fig. 3.4. Using the above procedure we find the following set of S.P. partitions:

\[
\begin{align*}
\pi(0) &= \left\{ \overline{1}; \overline{2}; \overline{3}; \overline{4}; \overline{5}; \overline{6}; \overline{7}; \overline{8} \right\}, \\
\pi_1 &= \left\{ 1,2; \overline{3},\overline{4}; \overline{5},\overline{6}; \overline{7},\overline{8} \right\}, \\
\pi_2 &= \left\{ 1,2,3,4; \overline{5},\overline{6},\overline{7},\overline{8} \right\}, \\
\pi_3 &= \left\{ \overline{1}; \overline{2}; \overline{3}; \overline{4},\overline{5}; \overline{6}; \overline{7}; \overline{8} \right\}, \\
\pi_4 &= \left\{ 1,2; \overline{3},\overline{4},\overline{5},\overline{6}; \overline{7},\overline{8} \right\}, \\
\pi_5 &= \left\{ \overline{1}; \overline{2}; \overline{3},\overline{6}; \overline{4}; \overline{5}; \overline{7}; \overline{8} \right\}, \\
\pi_6 &= \left\{ \overline{1}; \overline{2}; \overline{3},\overline{6}; \overline{4},\overline{5}; \overline{7}; \overline{8} \right\}, \\
I &= \left\{ 1,2,3,4,5,6,7,8 \right\}.
\end{align*}
\]
<table>
<thead>
<tr>
<th>STATE</th>
<th>I/P</th>
<th>O/P</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Fig. 3.4. P.S.M. M.

Fig. 3.5. Lattice $L_M$ of M.
The corresponding lattice $L_M$ is shown in Fig. 3.5. Also if we consider $\pi_4$ say, the $\pi_4$-image of $M$ is shown in Fig. 3.6.

The concept of S.P. partitions is very useful in the serial and parallel decompositions of an F.S.M. into its components, as shown in the following theorems.

**Theorem 3.1.** The F.S.M. $M$ has a non-trivial serial decomposition of its state behaviour iff there exists a nontrivial S.P. partition $\pi$ on the set of states of $M$.

Proof. (Pages 45-46, Ref. 12).

If the largest block of $\pi$ has $k$ states, then $M$ is defined by $\pi$ and $\tau$, where $\tau$ is a $k$-block partition such that

$$\pi \cdot \tau = \pi(0) \quad \ldots (3.0)$$

For example, for the machine shown in Fig. 3.7 the partition $\pi = \{1,2; 3,4,5\}$ has S.P. One possible $\tau$ is then $\tau = \{1,3; 2,4; 5\}$ because

$$\{1,2; 3,4,5\} \cdot \{1,3; 2,4; 5\} = \pi(0).$$

**Theorem 3.2.** The F.S.M. $M$ has a non-trivial parallel decomposition of its state behaviour iff there exist two nontrivial S.P. partitions $\pi_1$ and $\pi_2$ on $M$ such that

$$\pi_1 \cdot \pi_2 = 0 \quad \ldots (3.1)$$

Proof. (Pages 48-51, Ref. 12).

3.6 State reduction using S.P. partitions.

Another application of S.P. partitions is in finding the
Fig. 3.6. A $\pi_4$ image of M.

A = (1,2)
B = (3,4,5,6)
C = (7,8)

Fig. 3.7. An F.S.M. to demonstrate serial-decomposition.
reduced machine that is equivalent to $M$. The technique is based on the following.

**Definition 3.14.** For a machine $M$, we define $\pi_R$ to be the partition on $M$ such that $s \equiv t$ ($\pi_R$) iff state $s$ is equivalent to state $t$.

It can be shown (page 55, Ref. 12) that $\pi_R$ has S.P. and $M_{\pi_R}$ with the output

$$\lambda_R(B_{\pi_R}, x) = \lambda(s, x) \quad \text{for } s \text{ in } B_{\pi_R}$$

is the reduced equivalent of $M$.

Thus once the S.P. lattice is obtained $M_{\pi_R}$ is easily found by deciding which S.P. partition is $\pi_R$. An easy method for this is based on the following.

**Theorem 3.3.** If $M$ is an F.S.M., then $\pi_R$ is the maximal output consistent (O.C.) partition with S.P. Also S.P. partition $\pi$ is O.C. iff $\pi \in \pi_R$. (A partition $\pi$ on the states of $M$ is O.C. iff $s \equiv t$ ($\pi$) implies $\lambda(s, x) = \lambda(t, x)$ for all inputs $x$.

**Proof.** (Page 56, Ref. 12).

It is also easily shown that the O.C. S.P. partition form a sub-lattice of the S.P. lattice. Furthermore it is easy to test a partition to see if it is O.C. Thus once the S.P. lattice is given, $M_{\pi_R}$, the reduced equivalent of $M$ may be determined in a straightforward way.
3.7 Conclusion.

We have presented a very brief introduction to the main concepts in the structural theory of decompositions of finite-state sequential machine. A more detailed treatment of the theory, which includes advanced concepts like partition-pair algebra and state-splitting may be found in any of the references given.
CHAPTER 4

FINITE-STATE MACHINE MODELS
OF
STORED-LOGIC DIGITAL FILTERS

4.0 Introduction.

In this chapter, we investigate the feasibility of applying the structure theory of finite-state sequential machines (F.S.M's) to the implementation of digital filters. The results we obtain give us a valuable insight into the problem of using this direct modelling technique to realise general filter sections.

4.1 General approach.

We have seen in Chapter 2 that the conventional way to implement the basic second-order section, shown in Fig. 2.3 and described by equation (2.4), is to use adder, multiplier and delay units. Also, we know from Chapter 3 that by using the theory of state partitions with the substitution property, it is possible to decompose an F.S.M. into an interconnection of "smaller" machines.

In contrast to the conventional method we propose to realise a basic biquadratic section as a table look-up or stored logic unit. This unit is subsequently modelled as an F.S.M. which is then analysed using the method of S.P. partitions.

4.2 Stored-logic digital filters.

Conceptually, the method of table look-up is the most straight-forward way to realise combinational switching functions in general
and arithmetic circuits$^42$, $^43$, $^44$ in particular. (Maclean and Aspinall$^42$ used this technique to design a practical decimal adder in as early as 1957).

A general table look-up arithmetic unit is shown in Fig. 4.0(a), in which the arithmetic function $g$ is a function of $n$ independent variables $q_k$'s, $k = 1, 2, \ldots, n$, each $q_k$ being an $i$-valued variable. Consequently, there are $m$ possible values of $g$, where $m = (i)^n$. Every value of $g$ is precomputed and stored in a memory or storage unit. A particular value of $g$ is accessed by the corresponding $n$-tuple $(q_1, q_2, \ldots, q_k, \ldots, q_n)$ which forms the memory address.

In practice, data are usually represented in the binary form, in which case $i = 2$, and $q_k = 0$ or 1. Also $g$ will now be represented by $z$ bits. The resulting table look-up circuit is now as shown in Fig. 4.0(b), and it is usual to characterise this memory circuit by its capacity $M$ given by,

$$M = (2^n) \times z \text{ word-bits (W-b)} \quad \ldots(4.0)$$

At present, the table look-up operation is normally implemented using semiconductor bipolar or M.O.S. L.S.I. read-only or read-and-write memory chips. A typical organisation of a read-only memory (R.O.M.) is shown in Fig. 4.1.

Since the delay time is dependent only on the access time of the memory store, circuits designed using the look-up technique are obviously fast in operation and easy to construct, test and maintain. Furthermore, the architecture of any digital system designed this way is independent of device technology since the introduction of memory stores of larger capacity and faster access time will only
Fig. 4.0. General table look-up arithmetic units for (a) \( i \)-valued and (b) binary variables.

Fig. 4.1. Functional organisation of a read-only memory (R.O.M.).
result in a more efficient use of the basic system architecture.

Although R.O.M's have been incorporated in the hardware structures of digital filters (recall Section 2.3.3), they are used only for the partial computation of the overall filter algorithm.

Our approach, however, is different, in that we propose to implement a complete second-order digital filter section as a look-up table. Using equation (2.4) we first precompute the section output for every combination of present input and past inputs and/or outputs. The resulting output values are then written into a suitable memory store. In operation, the present input and past inputs and/or outputs act as addresses of the memory to access the relevant filter output.

A digital filter implemented in this manner will be termed a stored-logic (S.L.) digital filter.

4.3 Examples of S.L. digital filters.

We now illustrate the approach by deriving the S.L. forms of a few typical digital filter structures.

4.3.0 Second-order non-recursive section.

Consider a second-order non-recursive section whose data and coefficients are represented by 2 bits, and whose coefficient values are

\[ a_0 = 1, \quad a_1 = 3 \quad \text{and} \quad a_2 = 2. \]

We now compute the maximum value of the filter output, \( y_{n, \text{max}} \), by setting each \( x_{n-k}, \quad k = 0,1,2 \), to its maximum value of 3. Using equation (2.4) with \( b_1 = b_2 = 0 \), we find that, since
\[ y_{n_{\text{max}}} = (1 \times 3) + (3 \times 3) + (2 \times 3) = 18, \]

we require 5 bits to represent the filter output.

This filter which we will label D.F.1 is shown in Fig. 4.2(a), and its input-output relationship which is to be stored is given in Table 4.0. The address inputs consist of \( x_n, x_{n-1} \) and \( x_{n-2} \), and the data to be written into the look-up memory are given in the last four columns. The corresponding S.L. filter is shown in Fig. 4.2(b), and requires a \( 64 \times 5 \) word-bit storage module.

4.3.1 First-order recursive section.

Consider the first-order recursive filter labelled D.F.2 shown in Fig. 4.3(a) whose feedback coefficient \( b_1 = 5/8 = 0.101_2 \). The input \( x_n \) is represented by 2 bits, while 3 bits are used for \( b_1 \) and the output \( w_n \). Also, the 6-bit product \( b_1 \times w_{n-1} \) is quantised to 3 bits.

The values of \( w_n \) for all possible combinations of present input \( x_n \) and past output \( w_{n-1} \) are shown in Table 4.1. The S.L. form of D.F.2 is given in Fig. 4.3(b), in which a store of \( 32 \times 3 \) word-bits is used.

4.3.2 Second-order autonomous recursive section.

This section D.F.3 is shown in Fig. 4.4(a) in which \( b_1 = 2 \times 2^{-2} \) and \( b_2 = 3 \times 2^{-2} \) simplify subsequent analyses, we let the input be zero and the past outputs \( w_{n-1} \) and \( w_{n-2} \) have non-trivial initial values*.

The data and coefficients are represented by 2 bits, while the sum of the double-length products, \( b_1 \times w_{n-1} \) and \( b_2 \times w_{n-2} \).

* See Appendix 4.0 for a further explanation.
Fig. 4.2. Conventional (a) and stored-logic (b) realisations of D.F.l.
<table>
<thead>
<tr>
<th>Past inputs</th>
<th>Present input $x_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_{n-1}$</td>
<td>$x_{n-2}$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Filter output $y_n$

**Table 4.0.** Input-output relationship of D.F.1
(all data to be represented in binary).
Fig. 4.3. Conventional (a) and stored-logic (b) realisations of D.F.2.

Fig. 4.4. Conventional (a) and stored-logic (b) realisations of D.F.3.
| Previous output, \( w_{n-1} \times 2^{-3} \) | Present input \( x_n \times 2^{-3} \) | \( \) | \( \) | \( \) | \( \) |
|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 |
| 1 | 1 | 2 | 3 | 4 |
| 2 | 1 | 2 | 3 | 4 |
| 3 | 2 | 3 | 4 | 5 |
| 4 | 3 | 4 | 5 | 6 |
| 5 | 3 | 4 | 5 | 6 |
| 6 | 4 | 5 | 6 | 7 |
| 7 | 4 | 5 | 6 | 7 |

rounded output \( w_n' \)

Table 4.1. Input-output relationship of D.F.2.
are quantised to 2 bits. The state output relationship of D.F.3 is shown in Table 4.2 and its S.L. form is given in Fig. 4.4(b), which requires a $16 \times 2$ W-b store.

4.3.3 Memory storage requirements.

The examples we have discussed demonstrate the implementation of a few typical filter sections as stored-logic units. This direct approach suffers from the following problems:

(a) In practical sections a tremendous amount of storage will be required, (a second-order 7-bit non-recursive section, for example, requires a memory store of over two million words).

(b) Possible redundancies in the stored-table entries are difficult to determine.

(c) It is also not easy to detect any structure or pattern that may exist between the stored data.

4.4 F.S.M. models of digital filters.

We will now describe how the S.L. filters (D.F.1-3) that we discussed in the previous section may be modelled by F.S.M's. In the traditional approach, the design of a table look-up circuit is considered to be completed as soon as the input (address) - output relationship has been determined. We hope to extend the design problem by analysing look-up tables via F.S.M. models to achieve a reduction in the memory requirement and a systematic decomposition procedure for general S.L. filters.
<table>
<thead>
<tr>
<th>Past outputs $w_{n-1} \times 2^{-2}$</th>
<th>Double-length output $w_n \times 2^{-4n}$</th>
<th>Quantised output $w'_n$, by truncation;round-off</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_{n-2}$</td>
<td>$b_1 \times w_{n-1} \ b_2 \times w_{n-2}$</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 1</td>
<td>0 3</td>
<td>0 3</td>
</tr>
<tr>
<td>0 2</td>
<td>0 6</td>
<td>1 2</td>
</tr>
<tr>
<td>0 3</td>
<td>0 9</td>
<td>2 2</td>
</tr>
<tr>
<td>1 0</td>
<td>2 0</td>
<td>0 1</td>
</tr>
<tr>
<td>1 1</td>
<td>2 3</td>
<td>1 1</td>
</tr>
<tr>
<td>1 2</td>
<td>2 6</td>
<td>2 2</td>
</tr>
<tr>
<td>1 3</td>
<td>2 9</td>
<td>2 3</td>
</tr>
<tr>
<td>2 0</td>
<td>4 0</td>
<td>1 1</td>
</tr>
<tr>
<td>2 1</td>
<td>4 3</td>
<td>1 2</td>
</tr>
<tr>
<td>2 2</td>
<td>4 6</td>
<td>2 3</td>
</tr>
<tr>
<td>2 3</td>
<td>4 9</td>
<td>3 3</td>
</tr>
<tr>
<td>3 0</td>
<td>6 0</td>
<td>1 2</td>
</tr>
<tr>
<td>3 1</td>
<td>6 3</td>
<td>2 2</td>
</tr>
<tr>
<td>3 2</td>
<td>6 6</td>
<td>3 3</td>
</tr>
<tr>
<td>3 3</td>
<td>6 9</td>
<td>3 3</td>
</tr>
</tbody>
</table>

Table 4.2. State-output relationship of D.F.3.
F.S.M. model of a general S.L. non-recursive second-order section.

The above model is derived very simply by redrawing the standard configuration of the non-recursive filter to that shown in Fig. 4.5 such that it now corresponds to the familiar Mealy machine described by the 5-tuple \((S, I, O, \delta, \lambda)\). [See Definition 3.0], via the following mappings:

\[
\begin{align*}
    h_1 &: X_{n-1} \times X_{n-2} \rightarrow S \\
    h_2 &: X_n \rightarrow I \\
    h_3 &: Y_n \rightarrow O \\
    h_s &: a \rightarrow \delta \\
    h_o &: b \rightarrow \lambda
\end{align*}
\]

where \(X_{n-i}\), \((i = 0, 1, 2)\), is the set of all possible values of \(x_{n-i}\),

\(Y_n\) is the set of all possible values of the filter output \(y_n\),

\(a:\((x_{n-1}, x_{n-2}), x_n) \rightarrow (x_n, x_{n-1})\)

and \(b\) is the filter algorithm described by equation (2.4).

Thus, the "internal state" of the F.S.M. filter is represented by the outputs of the two delay elements.

An application.

We now apply the modelling technique to D.F.1, and thus obtain the flow table shown in Table 4.3, in which the states, represented by the ordered pairs \((x_{n-1}, x_{n-2})\)'s, have been appropriately labelled. For simplicity, the corresponding state diagram is drawn only for the inputs 0 and 2, as shown in Fig. 4.6.
Fig. 4.5. F.S.M. model of a general second-order non-recursive filter.
<table>
<thead>
<tr>
<th>Present state ordered pair label</th>
<th>Input 0</th>
<th>Input 1</th>
<th>Input 2</th>
<th>Input 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 A</td>
<td>A E I M</td>
<td>0 1 2 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 B</td>
<td>A E I M</td>
<td>2 3 4 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 2 C</td>
<td>A E I M</td>
<td>4 5 6 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 3 D</td>
<td>A E I M</td>
<td>6 7 8 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 E</td>
<td>B F J N</td>
<td>3 4 5 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 F</td>
<td>B F J N</td>
<td>5 6 7 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 2 G</td>
<td>B F J N</td>
<td>7 8 9 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 3 H</td>
<td>B F J N</td>
<td>9 10 11 12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 0 I</td>
<td>C G K Ø</td>
<td>6 7 8 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 1 J</td>
<td>C G K Ø</td>
<td>8 9 10 11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 2 K</td>
<td>C G K Ø</td>
<td>10 11 12 13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 3 L</td>
<td>C G K Ø</td>
<td>12 13 14 15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 0 M</td>
<td>D H L P</td>
<td>9 10 11 12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 1 N</td>
<td>D H L P</td>
<td>11 12 13 14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 2 Ø</td>
<td>D H L P</td>
<td>13 14 15 16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 3 F</td>
<td>D H L P</td>
<td>15 16 17 18</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3. Flow table of F.S.M. equivalent of D.F.1.
Fig. 4.6. State diagrams for the F.S.M. model of D.F.1.
with respect to inputs (a) 0 and (b) 2 respectively.
This F.S.M. equivalent of D.F.1 is so "rich" in S.P. partitions that it is impractical to generate them manually. Instead, a computer program in Fortran 1900 which was written by one of the author's colleagues was used. To obtain some idea of the size of the S.P. partition set, it suffices to say that the program found 120 basic partitions while from the first level sums alone, over 300 partitions were obtained.

It can be seen however that the next state time function is the simplest possible, since

$$d'_1(t+1) = d_1(t) \text{ and } d'_2(t+1) = d_2(t) = d'_1(t)$$

where $d_1, d_2$ and $d'_1, d'_2$ are the inputs and outputs of the delay elements $D_1$ and $D_2$ respectively. Also $d_1 = x_n, d'_1 = d'_2 = x_{n-1}$ and $d_2 = x_{n-2}$.

Nevertheless, the existence of S.P. partitions is still useful if some of them are output consistent (O.C.) as well, in which case it is possible to minimise the F.S.M. It may then be necessary to code the state variables *.

From Table 4.3 the following is the largest O.C. partition,

$$\tau = \{A, \overline{B}, \overline{C}, \overline{Dl}, \overline{HM}, J, E, F, \bar{G}, \bar{K}, \bar{L}, \bar{N}, \emptyset, F\}.$$ 

$\tau$, however, is not S.P. since the blocks $\overline{Dl}$ and $\overline{HM}$ implies that $AC, EG, IK, M\emptyset$ and $BD, FH, JL, NP$ must be "identified" thus leading to the partition $\pi$, where

$$\pi = \{AC, BD\bar{K}, \overline{FH\emptyset}, EG, JL, NP\}$$

and already with this initial implication, $\pi \neq \tau$. Thus $\tau$ is not preserved for inputs, and hence the F.S.M. given in Table 4.3 is a reduced machine.

* See Appendix 4.1.
State-reduction of the general F.S.M. non-recursive section.

Consider a general second-order non-recursive filter in which each \( x_{n-1} \) and \( a_i \) may assume any value from the set \( Z_R' \), where

\[
Z_R = \{ x \mid x \text{ integer, } 0 \leq x < R \}.
\]

The corresponding F.S.M. equivalent will then have \( R \) possible input values, and \( R^2 \) internal states, which are all the possible combinations of the ordered-pair \( (x_{n-1}, x_{n-2}) \). The general form of the flow table for this F.S.M. is shown in Table 4.4.

**Definition 4.0.** We define \( \tau(i) \) to be the partition on \( S \), the set of states of the above F.S.M., such that two ordered-pairs are in the same block of \( \tau(i) \) only if their \( i \)th components are identical.

It is easily seen that \( \tau(i) \) consists of \( R \) blocks, each containing \( R \) states or ordered-pairs.

**Lemma 4.0.** \( \tau(1) \) has the substitution property.

**Proof.** Consider any two distinct states, \( p_1 \) and \( p_2 \), in the same block \( B_k \) of \( \tau(1) \) i.e.

\[
p_1 = (k, g) \quad \text{and} \quad p_2 = (k, h).
\]

Using \( \delta \) as defined in Section 4.4.0, the next states of \( p_1 \) and \( p_2 \) for any particular input \( x_n = j \) are

\[
\delta[p_1, j] = \delta[(k, g), j] = (j, k) \quad \text{and}
\]

\[
\delta[p_2, j] = \delta[(k, h), j] = (j, k) \quad \text{respectively}.
\]
Table 4.4 Flow table for the F.S.M. equivalent of a general non-recursive second-order filter.
Thus, for a given input $p_1$ and $p_2$ have the same next state and consequently no further implication of S.P. partition blocks is possible. Since $p_1$ and $p_2$ are arbitrary states in $B_k$, it follows that all the states in $B_k$ will be mapped to the same next state. Also, as $B_k$ is an arbitrary block, therefore $\tau(1)$ has the substitution property.

As an example, see the state graphs in Fig. 4.6 for $x_n = 0$ and $x_n = 2$.

**Lemma 4.1.** Any non-trivial partition $\tau \not\leq \tau(1)$ cannot be output-consistent.

**Proof.** Consider $\tau'$ the smallest form of $\tau$. This will have one block $b_k$ containing two distinct elements, while the remaining are just one-element blocks. Let $p_1$ and $p_2$ defined as in Lemma 4.0 be in $b_k$. For a particular input $x_n = q$, the corresponding filter output will be given by

$$y_{n1} = a_0 q + a_1 k + a_2 g \quad \ldots (4.1)$$

and

$$y_{n2} = a_0 q + a_1 k + a_2 h \quad \ldots (4.2)$$

If $\tau'$ is output-consistent (O.C.) then we must have $y_{n1} = y_{n2}$ which implies that, since $k$ and $q$ are fixed, $a_2 g = a_2 h$. This is only possible if $g = h$. By construction however $g \neq h$. Therefore $\tau'$ is not O.C. Since in general $\tau$ must contain at least one block with two elements, no $\tau$ can be O.C.

**Lemma 4.2.** Any non-trivial O.C. partition on $S$ cannot have the substitution property.
Proof. As a consequent of Lemma 4.1 we see that for any partition on $S$ to be O.C. any pair of states, $s_1$ and $s_2$, in a block must have different values of their first components, i.e.

$$s_1 = (k_1, g_1) \text{ and } s_2 = (k_2, g_2), \quad k_1 \neq k_2$$

For any particular input $x_n = j$, the $j$-successors of $s_1$ and $s_2$ are given by

$$\delta(s_1, j) = \delta((k_1, g_1), j) = (j, k_1)$$

and

$$\delta(s_2, j) = \delta((k_2, g_2), j) = (j, k_2).$$

Consequently, $s_1$ and $s_2$ are mapped to the same block of $\tau(l)$, and hence the transitions to next states of $s_1$ and $s_2$ do not lead to the same output, i.e.

$$\lambda(s_1, j) \neq \lambda(s_2, j).$$

This means that any O.C. partition we start with will not be "preserved" even for the next immediate input. Therefore no O.C. partition can be S.P.

The previous Lemmas lead naturally to the following Theorem.

**Theorem 4.0.** For a general second-order non-recursive digital filter in which each of the data and coefficients comes from the set $Z_R$, the corresponding F.S.M. model is already in the minimal form.

### 4.4.0.2 Partial state reduction.

Although it has now been shown that the F.S.M. equivalent
of a second-order non-recursive section is inherently minimised, a simplification is still possible.

Suppose we represent the outputs of the F.S.M. equivalent of D.F.I shown in Table 4.3 in radix-4 arithmetic, i.e. \((14)_{10}\) say is written as \((0, 3, 2)_{4}\). If we consider only the least significant digits for the moment, the modified output table shown in Table 4.5 will be obtained. From it, we find the following O.C. partition \(\tau_d\) given by

\[
\tau_d = \{A, C, J, L; B, D, I, K; E, G, N, P; F, H, M, \emptyset\}
\]

which is also S.P. Thus we may regard the blocks of \(\tau_d\) as the states of the reduced equivalent of the machine whose output table is shown in Table 4.6. Furthermore, this reduced machine has the following useful S.P. partition

\[
\pi = \{Q, R; S, T\}
\]

To realise this reduced machine we require a partition \(\tau_a\) such that \(\pi \cdot \tau_a = \tau_d = \{Q, R, S, T\}\). One such \(\tau_a\) is the non-S.P. partition \(\{QS, \overline{RT}\}\). The initial F.S.M. (which incorporate the remaining output digits) is now easily implemented by using a partition \(\tau_b\) to distinguish between the states in the blocks of \(\tau_d\), i.e. we require that

\[
\tau_d \cdot \tau_b = \pi(O) = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, \emptyset, P\}
\]

The block diagram of the overall realisation is shown in Fig. 4.7 which results in a saving of about one third of the nominal storage of the direct form shown in Fig. 4.5.

It is possible to achieve further savings if the component
### Table 4.5. Output (least significant digit) table of D.F.I.

<table>
<thead>
<tr>
<th>Present state</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td>B</td>
<td>2 3 0 1</td>
</tr>
<tr>
<td>C</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td>D</td>
<td>2 3 0 1</td>
</tr>
<tr>
<td>E</td>
<td>3 0 1 2</td>
</tr>
<tr>
<td>F</td>
<td>1 2 3 0</td>
</tr>
<tr>
<td>G</td>
<td>3 0 1 2</td>
</tr>
<tr>
<td>H</td>
<td>1 2 3 0</td>
</tr>
<tr>
<td>I</td>
<td>2 3 0 1</td>
</tr>
<tr>
<td>J</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td>K</td>
<td>2 3 0 1</td>
</tr>
<tr>
<td>L</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td>M</td>
<td>1 2 3 0</td>
</tr>
<tr>
<td>N</td>
<td>3 0 1 2</td>
</tr>
<tr>
<td>Ø</td>
<td>1 2 3 0</td>
</tr>
<tr>
<td>P</td>
<td>3 0 1 2</td>
</tr>
</tbody>
</table>

### Table 4.6. Flow table of reduced F.S.M. equivalent of D.F.I.

<table>
<thead>
<tr>
<th>Present state</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>Q,0 S,1 R,2 T,3</td>
</tr>
<tr>
<td>R</td>
<td>Q,2 S,3 R,0 T,1</td>
</tr>
<tr>
<td>S</td>
<td>R,3 T,0 Q,1 S,2</td>
</tr>
<tr>
<td>T</td>
<td>R,1 T,2 Q,3 S,0</td>
</tr>
</tbody>
</table>

\[
\tau_d = \{A,C,J,L; B,D,I,K; E,G,N,P; F,H,M,Ø\} \\
= \{Q,R,S,T\}
\]

Table 4.6. Flow table of reduced F.S.M. equivalent of D.F.I.
Fig. 4.7. Cascade realisation of F.S.M. model of D.F.1.
\( \tau_b \) of the F.S.M. in Fig. 4.7 is simplified, by applying similar analyses to the second and third digits respectively.

### 4.4.1 F.S.M. model of second-order autonomous recursive section.

This is the filter D.F.3 described in Section 4.3.2 (Figs. 4.4(a) and (b)). Its F.S.M. equivalent, shown in Fig. 4.8, is obtained by the following mappings

\[
\begin{align*}
    h_1 : W_{n-1} \times W_{n-2} &\rightarrow S \\
    h_2 : \left[ W_n \right]_t (\text{or } r) &\rightarrow 0 \\
    \delta : \left[ W_n', W_{n-1} \right](\Lambda) &\rightarrow \left[ W_{n-1}, W_{n-2} \right](\Lambda + 1)
\end{align*}
\]

where \( \left[ W_n \right]_t \), \( \left[ W_n \right]_r \) are the sets of truncated and rounded filter outputs respectively, and \( \Lambda \) is a particular time instant.

For these two forms of quantisation, the corresponding flow tables and state graphs are shown in Tables 4.7(a) and (b), and Figs. 4.9(a) and (b) respectively.

It is interesting to note that the state graphs illustrate quite clearly the existence of limit cycles. For example, in Fig. 4.9(a), if the F.S.M. is initiated at state M then, after two state transitions, the F.S.M. will be alternating between states J and G resulting in the periodic output \( \{w_n'\} = \ldots 1,2,1,2,1,2,\ldots \). The machine could also settle to a constant amplitude limit cycle if, for instance, it is started at state N. Then, after three transitions, with outputs 2,3,3, the machine stays in P with the corresponding output of 3.
Fig. 4.8. F.S.M. model of D.F.3.

Fig. 4.9. State graphs for D.F.3 with output (a) truncation and (b) rounding-off.
<table>
<thead>
<tr>
<th>Present state</th>
<th>Next state</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>A</td>
<td>A 0</td>
</tr>
<tr>
<td>0,1</td>
<td>B</td>
<td>A 0</td>
</tr>
<tr>
<td>0,2</td>
<td>C</td>
<td>E 1</td>
</tr>
<tr>
<td>0,3</td>
<td>D</td>
<td>I 2</td>
</tr>
<tr>
<td>1,0</td>
<td>E</td>
<td>B 0</td>
</tr>
<tr>
<td>1,1</td>
<td>F</td>
<td>F 1</td>
</tr>
<tr>
<td>1,2</td>
<td>G</td>
<td>J 2</td>
</tr>
<tr>
<td>1,3</td>
<td>H</td>
<td>J 2</td>
</tr>
<tr>
<td>2,0</td>
<td>I</td>
<td>G 1</td>
</tr>
<tr>
<td>2,1</td>
<td>J</td>
<td>G 1</td>
</tr>
<tr>
<td>2,2</td>
<td>K</td>
<td>K 2</td>
</tr>
<tr>
<td>2,3</td>
<td>L</td>
<td>Ø 3</td>
</tr>
<tr>
<td>3,0</td>
<td>M</td>
<td>H 1</td>
</tr>
<tr>
<td>3,1</td>
<td>N</td>
<td>L 2</td>
</tr>
<tr>
<td>3,2</td>
<td>Ø</td>
<td>P 3</td>
</tr>
<tr>
<td>3,3</td>
<td>P</td>
<td>P 3</td>
</tr>
</tbody>
</table>

(a) truncated and (b) rounded-off.

Table 4.7. Flow tables of F.S.M. D.F.3 with output
Now consider the F.S.M. described by Table 4.7(a), in which we find that the largest O.C. partition is $\tau_1$, where

$$\tau_1 = \{ \text{ABE, CFIJM, DGHKN, LOP} \} .$$

This O.C. partition however is not S.P. as may be seen by considering the pair of states C and F in the second block of $\tau_1$. By applying the transition function $\delta$, we find that

$$\delta(C, I_o) = E \quad \text{and} \quad \delta(F, I_o) = F$$

where $I_o$ is the zero input. We see now that E and F are in different blocks of $\tau_1$. Therefore $\tau_1$ is not preserved.

$\tau_1$, however, may be refined to $\tau_2$,

where $\tau_2 = \{ \text{ABE, C, F, K, IJM, DGH, N, LOP} \}$

$$= \{ a, b, c, d, e, f, g, h \}$$

which can be shown to be S.P.

Consequently, the F.S.M. in Table 4.7(a) may be reduced to that shown in Table 4.8, in which some of the possible S.P. partitions are

$$\pi_1 = \{ \text{abcd, efgh} \}$$

$$\pi_2 = \{ \text{abef, cdgh} \}$$

$$\pi_3 = \{ \text{abgh, cdef} \}$$

$$\pi_4 = \{ \text{ab, cd, ef, gh} \}$$

One possible realisation of $M$, the reduced machine is to use $\pi_4$ and $\tau_2$ because

$$\pi_4 \cdot \tau_a = \tau_2.$$
The corresponding block diagram is shown in Fig. 4.10 which requires two $2 \times 1$ w-b memories and an $8 \times 3$ W-b memory.

$M_{12}$ may be simplified further when we assign binary variables to the internal states. One such assignment is shown in Table 4.9 which is the binary coded form of Table 4.8. From Table 4.9 we observe that the $Y_2, Y_1$ columns are identical to those of $y'_2, y'_1$. Thus we may eliminate two delay elements and use $y_2$ and $y_1$, as control variables, required only to specify the initial state of the F.S.M. model of D.F.3.

The final realisation is shown in Fig. 4.11 requiring only an 8-word store, thus representing a considerable simplification over the direct form shown in Fig. 4.8.

4.4.2 F.S.M. model of first-order recursive section.

The above filter is D.F.2 which we described in Section 4.3.1 (Figs. 4.3(a) and (b)), and characterised by Table 4.1. By letting $\mathbb{W}_{n-1}$, the set of delayed output values, represent the state set of the corresponding F.S.M. model we obtain the flow table shown in Table 4.10. The direct realisation is shown in Fig. 4.12, in which a $32 \times 6$ W-b memory is required.

From the state and output table we find that the following partition $\pi_1$ has S.P. as well as being output-consistent, i.e.

$$\pi_1 = \{A, BC, D, EF, GH\}.$$

This leads to the reduced F.S.M. shown in Table 4.11, which, since it has five states, still require three binary variables in the state coding. Nevertheless, a modest simplification of
### Table 4.8. Flow table for $M_{\tau_2}$

<table>
<thead>
<tr>
<th>Present state</th>
<th>Next state</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>d</td>
<td>2</td>
</tr>
<tr>
<td>e</td>
<td>f</td>
<td>1</td>
</tr>
<tr>
<td>f</td>
<td>e</td>
<td>2</td>
</tr>
<tr>
<td>g</td>
<td>h</td>
<td>2</td>
</tr>
<tr>
<td>h</td>
<td>h</td>
<td>3</td>
</tr>
</tbody>
</table>

### Table 4.9. State-assignment of $M_{\tau_2}$

<table>
<thead>
<tr>
<th>Present state</th>
<th>Next state</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_2$</td>
<td>$y_1$</td>
<td>$y_0$</td>
</tr>
<tr>
<td>a $\rightarrow$ 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>h $\rightarrow$ 1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Fig. 4.10. Cascade realisation of reduced F.S.M. model of D.F.3.

Fig. 4.11. Final simplified implementation of D.F.3.
Table 4.10. State and output table for F.S.M. equivalent of D.F.2. (Output scaling $\times 2^{-3}$)

<table>
<thead>
<tr>
<th>Present state</th>
<th>Input 0</th>
<th>Input 1</th>
<th>Input 2</th>
<th>Input 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A,0</td>
<td>B,1</td>
<td>C,2</td>
<td>D,3</td>
</tr>
<tr>
<td>B</td>
<td>B,1</td>
<td>C,2</td>
<td>D,3</td>
<td>E,4</td>
</tr>
<tr>
<td>C</td>
<td>B,1</td>
<td>C,2</td>
<td>D,3</td>
<td>E,4</td>
</tr>
<tr>
<td>D</td>
<td>C,2</td>
<td>D,3</td>
<td>E,4</td>
<td>F,5</td>
</tr>
<tr>
<td>E</td>
<td>D,3</td>
<td>E,4</td>
<td>F,5</td>
<td>G,6</td>
</tr>
<tr>
<td>F</td>
<td>D,3</td>
<td>E,4</td>
<td>F,5</td>
<td>G,6</td>
</tr>
<tr>
<td>G</td>
<td>E,4</td>
<td>F,5</td>
<td>G,6</td>
<td>H,7</td>
</tr>
<tr>
<td>H</td>
<td>E,4</td>
<td>F,5</td>
<td>C,6</td>
<td>H,7</td>
</tr>
</tbody>
</table>

$\pi_1 = \{\overline{A}, \overline{BC}, \overline{D}, \overline{EF}, \overline{GH}\}$

$= \{P, Q, R, S, T\}$.

Table 4.11. Flow table of reduced D.F.2.

<table>
<thead>
<tr>
<th>Present state</th>
<th>Input 0</th>
<th>Input 1</th>
<th>Input 2</th>
<th>Input 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>P,0</td>
<td>Q,1</td>
<td>Q,2</td>
<td>R,3</td>
</tr>
<tr>
<td>Q</td>
<td>Q,1</td>
<td>Q,2</td>
<td>R,3</td>
<td>S,4</td>
</tr>
<tr>
<td>R</td>
<td>Q,2</td>
<td>R,3</td>
<td>S,4</td>
<td>S,5</td>
</tr>
<tr>
<td>S</td>
<td>R,3</td>
<td>S,4</td>
<td>S,5</td>
<td>T,6</td>
</tr>
<tr>
<td>T</td>
<td>S,4</td>
<td>S,5</td>
<td>T,6</td>
<td>T,7</td>
</tr>
</tbody>
</table>
Fig. 4.12. Direct F.S.M. model of D.F.2.

Fig. 4.13. Cascade realisation of reduced F.S.M. equivalent of D.F.2.
this reduced machine is possible by using its sole S.P. partition
\[ \pi_2 = \{ \overline{F}, \overline{QRST} \} \] in serial with \( \tau_a \) such that
\[ \pi_2 \cdot \tau_a = \pi(0) = \{ \overline{A}, \overline{BC}, \overline{D}, \overline{EF}, \overline{GH} \}. \]
Hence, one possible \( \tau_a \) is \( \{ \overline{PQ}, \overline{R}, \overline{S}, \overline{T} \} \). The cascade realisation of this F.S.M. filter, shown in Fig. 4.13, uses an \( 8 \times 1 \) W-b memory and a \( 32 \times 5 \) W-b memory for its look-up tables.

4.4.2.0 Decomposition results for D.F.2 with different feedback coefficient values.

The same modelling and decomposition techniques that we have discussed so far will now be applied to the basic first order recursive filter section for various values of the feedback coefficient \( b_1 \), from \( b_1 = (0.001)_2 \), i.e. \( \frac{1}{8} \), to \( b_1 = (0.111)_2 = \frac{7}{8} \). The F.S.M. equivalent of the section having \( b_1 = k/8 \) will be labelled \( M_k \).

The flow tables for the \( M_k \)'s, \( k = 1, 2, 3, 4, 6, 7 \) are shown in Tables 4.12(a) to (f). The number of possible input values is not the same for all the \( M_k \)'s because the maximum input in each F.S.M. is so chosen as to prevent section overflow (see Chapter 2).

Alongside each flow table, the corresponding set of basic S.P. partitions is given, as well as a subset of those partitions generated from higher level sums. The partitions in this subset are chosen for their convenient and useful number of blocks and block sizes, and are selected by the manual inspection of a very much larger collection of possible S.P. partitions generated using the computer program mentioned in Section 4.4.0.0.

The \( M_k \)'s are first analysed for output-consistent S.P.
Present state & Input (× 2⁻³) & \begin{tabular}{ccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
0 + A & A,0 & B,1 & C,2 & D,3 & E,4 & F,5 & G,6 \\
1 B & A,0 & B,1 & C,2 & D,3 & E,4 & F,5 & G,6 \\
2 C & A,0 & B,1 & C,2 & D,3 & E,4 & F,5 & G,6 \\
3 D & A,0 & B,1 & C,2 & D,3 & E,4 & F,5 & G,6 \\
4 E & B,1 & C,2 & D,3 & E,4 & F,5 & G,6 & H,7 \\
5 F & B,1 & C,2 & D,3 & E,4 & F,5 & G,6 & H,7 \\
6 G & B,1 & C,2 & D,3 & E,4 & F,5 & G,6 & H,7 \\
7 + H & B,1 & C,2 & D,3 & E,4 & F,5 & G,6 & H,7 \\
\end{tabular} & Useful S.P. partitions:

\[ \pi_1 = \{ABCD, \overline{EFGH}\} \]

\[ \pi_2 = \{AB, \overline{CD}, \overline{EF}, \overline{GH}\} \]

\[ \pi_3 = \{AC, \overline{BD}, \overline{EG}, \overline{FH}\} \]

\[ \pi_4 = \{AD, \overline{BC}, \overline{EH}, \overline{GF}\} \]

(a) Machine M₁

Tables 4.12(a) to (f). Flow tables and useful S.P. partitions for \( M_k \)'s, the F.S.M. equivalents of first-order recursive filters.
partitions in order to determine the possibility of machine
minimisation. The reduced equivalent machines \( m_k \)'s are described
by Tables 4.13(a) to (f). These reduced F.S.M's are in turn
analysed for useful S.P. partitions which may lead to parallel
or cascade realisations. These implementations are illustrated
in Figs. 4.14(a) to (f).

As a result of the analysis described above, the following
observations are made:

(i) Machine \( M_1 \). \( \pi_1 \) is O.C. Hence \( M_1 \) is reduced to \( m_1 \),
(denoted by \( M_1 \xrightarrow{R} m_1 \)), where \( m_1 \) is a 2-state machine.

(ii) Machine \( M_2 \). \( \pi_9 \) is O.C. Therefore we have \( M_2 \xrightarrow{R} m_2 \),
which is a 3-state machine.

(iii) \( M_3 \). \( \pi_1 \) is O.C. Hence \( M_3 \xrightarrow{R} m_3 \), in which \( m_3 \) is a
4-state machine. Although \( m_3 \) possesses the S.P. partition
\( \pi = \{ PQR, S \} \), (see Table 4.13(c)), it is not useful because its
largest block contains three states. Consequently, the successor
component alone in the corresponding cascade realisation will
require two binary variables to code its states.

(iv) \( M_4 \). The S.P. partition \( \pi_1 = \{ \overline{A}, BC, DE, FG, H \} \) is
O.C. Therefore \( M_4 \xrightarrow{R} m_4 \), a 5-state machine. Also \( m_4 \) has the
S.P. partition \( \pi_2 = \{ \overline{F}, QRST \} \), and using \( \tau \) such that \( \pi_2 \cdot \tau = \pi'(0) = \pi_1 \),
i.e. \( \tau = \{ \overline{PQ}, \overline{R}, \overline{S}, \overline{T} \} \), the cascade realisation shown in Fig. 4.14(d)
is obtained.
Present state | Input
---|---
0 | 1 | 2 | 3 | 4 | 5
A | A,0 | B,1 | C,2 | D,3 | E,4 | F,5
B | A,0 | B,1 | C,2 | D,3 | E,4 | F,5
C | B,1 | C,2 | D,3 | E,4 | F,5 | G,6
D | B,1 | C,2 | D,3 | E,4 | F,5 | G,6
E | B,1 | C,2 | D,3 | E,4 | F,5 | G,6
F | B,1 | C,2 | D,3 | E,4 | F,5 | G,6
G | C,2 | D,3 | E,4 | F,5 | G,6 | H,7
H | C,2 | D,3 | E,4 | F,5 | G,6 | H,7

(b) Machine $M_2$

**Basic S.P. partitions**

$\pi_1 = \{AB, \overline{C}, \overline{D}, \overline{E}, \overline{F}, \overline{G}, \overline{H}\}$

$\pi_2 = \{A, B, \overline{CD}, \overline{E}, \overline{F}, \overline{G}, \overline{H}\}$

$\pi_3 = \{A, \overline{B}, \overline{CE}, \overline{D}, \overline{F}, \overline{G}, \overline{H}\}$

$\pi_4 = \{A, \overline{B}, \overline{CF}, \overline{D}, \overline{E}, \overline{G}, \overline{H}\}$

$\pi_5 = \{A, \overline{B}, \overline{C}, \overline{DE}, \overline{F}, \overline{G}, \overline{H}\}$

$\pi_6 = \{A, B, \overline{C}, \overline{DF}, \overline{E}, \overline{G}, \overline{H}\}$

$\pi_7 = \{A, B, \overline{C}, \overline{EF}, \overline{F}, \overline{G}, \overline{H}\}$

$\pi_8 = \{A, B, \overline{C}, \overline{D}, \overline{EF}, \overline{G}, \overline{H}\}$

**Useful S.P. partitions**

$\pi_9 = \{AB, \overline{CDEF}, \overline{GH}\}$

$\pi_{10} = \{AB, \overline{CD}, \overline{EF}, \overline{GH}\}$

$\pi_{11} = \{AB, \overline{CE}, \overline{DF}, \overline{GH}\}$

$\pi_{12} = \{AB, \overline{CF}, \overline{DE}, \overline{GH}\}$
<table>
<thead>
<tr>
<th>Present State</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>A,0</td>
</tr>
<tr>
<td>B</td>
<td>A,0</td>
</tr>
<tr>
<td>C</td>
<td>B,1</td>
</tr>
<tr>
<td>D</td>
<td>B,1</td>
</tr>
<tr>
<td>E</td>
<td>C,2</td>
</tr>
<tr>
<td>F</td>
<td>C,2</td>
</tr>
<tr>
<td>G</td>
<td>C,2</td>
</tr>
<tr>
<td>H</td>
<td>D,3</td>
</tr>
</tbody>
</table>

(c) $M_3$

S.P. partitions

$\pi_1 = \{\overline{AB}, \overline{CD}, \overline{EF}, \overline{H}\}$

$\pi_2 = \{\overline{AB}, \overline{CD}, \overline{EF}, \overline{G}, \overline{H}\}$

$\pi_3 = \{\overline{AB}, \overline{CD}, \overline{FG}, \overline{E}, \overline{H}\}$

$\pi_4 = \{\overline{AB}, \overline{CD}, \overline{EG}, \overline{F}, \overline{H}\}$
<table>
<thead>
<tr>
<th>Present state</th>
<th>Input</th>
<th>Present state</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>A,0</td>
<td>B,1</td>
<td>C,2</td>
</tr>
<tr>
<td>B</td>
<td>B,1</td>
<td>C,2</td>
<td>D,3</td>
</tr>
<tr>
<td>C</td>
<td>B,1</td>
<td>C,2</td>
<td>D,3</td>
</tr>
<tr>
<td>D</td>
<td>C,2</td>
<td>D,3</td>
<td>E,4</td>
</tr>
<tr>
<td>E</td>
<td>C,2</td>
<td>D,3</td>
<td>E,4</td>
</tr>
<tr>
<td>F</td>
<td>D,3</td>
<td>E,4</td>
<td>F,5</td>
</tr>
<tr>
<td>G</td>
<td>D,3</td>
<td>E,4</td>
<td>F,5</td>
</tr>
<tr>
<td>H</td>
<td>E,4</td>
<td>F,5</td>
<td>G,6</td>
</tr>
</tbody>
</table>

(d) $M_4$

<table>
<thead>
<tr>
<th>Present state</th>
<th>Input</th>
<th>Present state</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>A,0</td>
<td>B,1</td>
<td>C,2</td>
</tr>
<tr>
<td>B</td>
<td>B,1</td>
<td>C,2</td>
<td>D,3</td>
</tr>
<tr>
<td>C</td>
<td>C,2</td>
<td>D,3</td>
<td>E,4</td>
</tr>
<tr>
<td>D</td>
<td>C,2</td>
<td>D,3</td>
<td>E,4</td>
</tr>
<tr>
<td>E</td>
<td>D,3</td>
<td>E,4</td>
<td>F,5</td>
</tr>
<tr>
<td>F</td>
<td>E,4</td>
<td>F,5</td>
<td>G,6</td>
</tr>
<tr>
<td>G</td>
<td>F,5</td>
<td>G,6</td>
<td>H,7</td>
</tr>
<tr>
<td>H</td>
<td>F,5</td>
<td>G,6</td>
<td>H,7</td>
</tr>
</tbody>
</table>

(e) $M_6$

\[
\begin{align*}
\pi_1 &= \{A, BCDEFGH\} \\
\pi_2 &= \{A, B, CD, E, F, G, H\} \\
\pi_3 &= \{A, B, CDEFGH\} \\
\pi_4 &= \{A, B, C, D, E, F, GH\} \\
\pi_5 &= \{A, B, CD, E, F, GH\}
\end{align*}
\]
### Tables 4.13(a) to (f).

**Reduced equivalent machines**

$m_k$'s of $M_k$'s.
### Present State

<table>
<thead>
<tr>
<th>Present State</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>( \overline{A} \rightarrow P )</td>
<td>( P,0 )</td>
</tr>
<tr>
<td>( \overline{BC} \rightarrow Q )</td>
<td>( Q,1 )</td>
</tr>
<tr>
<td>( \overline{DE} \rightarrow R )</td>
<td>( Q,2 )</td>
</tr>
<tr>
<td>( \overline{FG} \rightarrow S )</td>
<td>( R,3 )</td>
</tr>
<tr>
<td>( \overline{H} \rightarrow T )</td>
<td>( R,4 )</td>
</tr>
</tbody>
</table>

(d) \( m_4 \)

### Present State

<table>
<thead>
<tr>
<th>Present State</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A \rightarrow P )</td>
<td>( P,0 )</td>
</tr>
<tr>
<td>( B \rightarrow Q )</td>
<td>( Q,1 )</td>
</tr>
<tr>
<td>( CD \rightarrow R )</td>
<td>( R,2 )</td>
</tr>
<tr>
<td>( F \rightarrow S )</td>
<td>( R,3 )</td>
</tr>
<tr>
<td>( \overline{F} \rightarrow T )</td>
<td>( S,4 )</td>
</tr>
<tr>
<td>( \overline{GH} \rightarrow U )</td>
<td>( T,5 )</td>
</tr>
</tbody>
</table>

(e) \( m_6 \)

### Present State

<table>
<thead>
<tr>
<th>Present State</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A \rightarrow P )</td>
<td>( P,0 )</td>
</tr>
<tr>
<td>( B \rightarrow Q )</td>
<td>( Q,1 )</td>
</tr>
<tr>
<td>( CD \rightarrow R )</td>
<td>( R,1 )</td>
</tr>
<tr>
<td>( F \rightarrow S )</td>
<td>( S,3 )</td>
</tr>
<tr>
<td>( \overline{EF} \rightarrow T )</td>
<td>( T,4 )</td>
</tr>
<tr>
<td>( \overline{GH} \rightarrow U )</td>
<td>( U,6 )</td>
</tr>
</tbody>
</table>

(f) \( m_7 \)
(v) \( M_6 \). \( \pi \) is O.C. Hence \( M_6 \stackrel{R}{\rightarrow} m_6 \), a 6-state machine.
As \( m_6 \) does not possess any useful S.P. partition, only a direct realisation is possible as shown in Fig. 4.14(e).

(vi) \( M_7 \). Its reduced equivalent \( m_7 \), which contains seven states, is obtained using the S.P. partition \( \{ A, \overline{B}, \overline{C}, \overline{D}, EF, \overline{G}, \overline{H} \} \).
This reduced machine \( m_7 \) possesses no useful S.P. partitions.

4.5 Discussion.
Some interesting features of the F.S.M. models of non-recursive and recursive stored-logic digital filters have been brought out as a consequence of our analysis.

We see that with the second-order non-recursive filter, although its F.S.M. equivalent is already in the minimal form further simplifications are possible if the filter output is represented as a multi-digit number with each digit regarded as a separate output for analysis.

With the autonomous 2-bit second-order recursive section, the direct realisation in Fig. 4.8 is simplified quite considerably, using S.P. partitions, to that shown in Fig. 4.11. In this example, there are still useful S.P. partitions after the state minimisation process. One of the problems encountered when the complete section is analysed directly is that for the same filter but with different values of the coefficients \( b_1 \) and \( b_2 \), the corresponding flow tables and state graphs are considerably different from one another.
Consider, for example, when the coefficients are \( b_1 = 3 \times 2^{-2} \) and \( b_2 = 2 \times 2^{-2} \). The state-output relationship is given by Table 4.14,
Figs. 4.14(a) - (f). Implementations of reduced machines $m_k$'s.
<table>
<thead>
<tr>
<th>Past outputs ((x \cdot 2^{-2}))</th>
<th>(b_1 \cdot w_{n-1})</th>
<th>(b_2 \cdot w_{n-2})</th>
<th>Double-length output (w_n) ((x \cdot 2^{-4}))</th>
<th>Quantised output (w'_n), by truncation</th>
<th>round-off</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>9</td>
<td>4</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>9</td>
<td>6</td>
<td>15</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4.14. State-output relationship of second-order autonomous recursive section D.F.4 with coefficients \(b_1 = 3\), and \(b_2 = 2\).
and the flow table and partial state graph of the F.S.M. model are shown in Tables 4.15(a) and (b) and Figs. 4.15(a) and (b) respectively. It is seen directly that they are very different in structure to Tables 4.7(a) and (b) and Figs. 4.9(a) and (b).

The same dependence of state structure on filter coefficient values is also true for first-order recursive filters as evidenced by the variety of different realisations shown in Fig. 4.14(a) to (f). We also note that simplifications of the F.S.M. models of the first-order sections are mainly due to state reductions. Among the machines analysed only M₄ and M₅ have reduced equivalents, m₄ and m₅, that could be simplified further via S.P. partitions. Furthermore, it is also observed that state reduction becomes increasingly difficult with increasing values of b₁, the feedback coefficient. This is illustrated in Fig. 4.16(a). A similar result is also obtained when the word-length is increased to 4-bits (see the graph in Fig. 4.16(b)).

One difficult problem with both types of recursive digital filters is the inherent non-linearity of the system as a result of output and state quantisation, either by truncation or round-off.

As an example, consider the reduced F.S.M. model of the first-order section whose flow table is given in Table 4.11, and two input sequences \( \{q₁\} \) and \( \{q₂\} \) given by

\[
\{q₁\} = 1,0,0,0,..........
\]

and

\[
\{q₂\} = 2,0,0,0,..........
\]
Table 4.15. Flow table of F.S.M. equivalent of filter shown in Table 4.14.
Fig. 4.15. State graphs for D.F.4 with output (a) truncation and (b) rounding-off.
Fig. 4.16. Effect of coefficient values on state reduction of F.S.M. models of (a) a 3-bit and (b) a 4-bit recursive filters.
The corresponding state transitions and output sequences for initial state $T$, say, are

$$\{s_1\} = T; S, R, Q, Q, \ldots$$
$$\{o_1\} = 5, 3, 2, 1, 1, \ldots$$

$$\{s_2\} = T; T, S, R, Q, Q, \ldots$$
$$\{o_2\} = 6, 4, 3, 2, 1, 1, \ldots$$

Let $\{q_3\}$ be the sum of the input sequences $\{q_1\}$ and $\{q_2\}$, i.e.

$$\{q_3\} = 3, 0, 0, \ldots$$

The corresponding state and output sequences, $\{s_3\}$ and $\{o_3\}$, are given by

$$\{s_3\} = T; T, S, R, Q, Q, \ldots$$
$$\{o_3\} = 7, 4, 3, 2, 1, 1, \ldots$$

Clearly $\{o_3\} \neq \{o_1\} + \{o_2\}$.

The consequence of this non-linear effect is that any result of the analysis of recursive digital filters with short word-lengths cannot be easily generalised to large word-length recursive sections.

4.6 Conclusions.

In general, the structure theory of finite-state sequential machines is conceptually attractive in the simplification of digital filters realised as stored-logic units. In practice the main problem is to generalise the method such that it may be applied, without
resorting to exhaustive manual or even computer search, to digital filters of both the recursive and non-recursive types, having varying word-lengths and coefficients. Even if a complete listing of S.P. partitions is possible, it is still extremely difficult to select the best subset of these partitions which will lead to a good realisation. Furthermore, it is not desirable to have to perform a complete analysis for every different filter specification.

In view of this the non-recursive section appears to be the most promising candidate for a general analysis.
APPENDIX 4.0

In the example in Section 4.3.2, the input to D.F.3 is assumed to be zero in order to simplify the subsequent analysis. The non-trivial initial values of the ordered-pair \((w_{n-1}, w_{n-2})\) may be set up by presetting the relevant delay registers. Alternatively, one can assume that prior to our analysis, D.F.3 has received the appropriate input sequence to 'send' the filter to a particular initial ordered-pair. As illustrated in Fig. A.4.0, a unique input sequence can always be found to connect the trivial state ordered-pair \((0,0)\) to any other ordered-pair.

Consequently, in the state diagram shown in Fig. 4.9(a), the starting states C,M,D,F,K and N which lead to limit-cycle oscillations may be reached from the trivial state \((0,0)\), i.e. A, by the application of the input sequences; \{2, -1\}, \{3\}, \{3, -1\}, \{1, 1\}, \{2, 1\} and \{1, 3\} respectively.

The above discussion assumes that the output is truncated, but the treatment is similar when the output is rounded-off instead.
Fig. A.4.0. State diagram of D.F.J for non-trivial input sequences of length ≤ 2, (with output truncation).
APPENDIX 4.1

Consider a general state ordered-pair \((s_1, s_2)\) of the F.S.M. model of D.F.l, and let its \(x\)-successor be \((s'_1, s'_2)\). From the discussion in Section 4.4.0.0, we can easily see that \(s'_1 = I\) and \(s'_2 = s_1\). Therefore the next-state function \(\delta\) simply consists of the two identity mappings,

\[
\begin{align*}
    s_1 &\rightarrow x, \\
    s_2 &\rightarrow s_1.
\end{align*}
\]

Thus in practice the implementation of \(\delta\) consists of the direct connection of \(x\) to \(s_1\) and \(s_1\) to \(s_2\) via the two delay registers.

Suppose now there exists an \(n\)-block O.C. partition which has also S.P. Then the F.S.M. equivalent of D.F.l may be reduced to an \(n\)-state machine. In such a case some combinational logic may be required for the \(\delta\) state transition mapping.
5.0 Introduction.

In the previous chapter we have encountered the limitation of the direct modelling of the complete digital filter as a finite-state sequential machine. To resolve some of the questions that were brought out there, we investigate in this chapter the application of S.P. partition techniques to the analysis of the arithmetic units that make up the filter algorithm. It is hoped that an insight into the algebraic structure of the overall section will be gained as a result of knowing the partition structures of its component units.

A general F.S.M. model is first introduced which will then be used as a basis for the structural analysis of N-bit adder and N-bit by N-bit multiplier modules.

5.1 F.S.M. model of a general arithmetic circuit.

Consider the case of an arithmetic function, g, of two variables or operands A and B, where

$$A, B = Z_M, \quad Z_M = \{x : x \text{ integer}, \ 0 \leq x \leq M-1\}.$$  

For our applications, the range of g is $Z_C$, where

$$|Z_C| > |Z_M|, \quad Z_C = \{x : x \text{ integer}, \ 0 \leq x \leq C-1\},$$  

C-1 being the maximum value of g(A,B).
This function is represented by the combinational or stored-logic circuit enclosed in the broken lines in Fig. 5.0.

The corresponding F.S.M. model for this arithmetic circuit is obtained by first separating $g(A,B)$ into two components $G_u$ and $G_t$ such that the elements of the former are identical to those of one of the operands, say $B$. For completeness we regard $G_t$ to be linked to $B$ by an imaginary feedback. The mappings below follow naturally.

$$
\begin{align*}
h_1 : A & \rightarrow I \\
h_2 : B, G_t & \rightarrow S \\
h_3 : G_u & \rightarrow O \\
h_4 : g_u & \rightarrow \lambda \\
h_5 : g_t & \rightarrow \delta
\end{align*}
$$

where $g_u : A \times B \rightarrow G_u$

and $g$ is now written as $g : A \times B \rightarrow (G_u, G_t)$.

Thus, an arithmetic function $g$ described by the four-tuple $(A,B,Z_C,g)$ may now be modelled by an F.S.M. $(S,I,O,\delta,\lambda)$.

* A discussion on the motivation behind and the theoretical constraints of the above model is given in Appendix 5.0.
Fig. 5.0. F.S.M. model of a general arithmetic circuit.

Fig. 5.1. An F.S.M. radix-$2^N$ 'half-adder'.
5.2 Radix - $2^N$ adders.

Conventionally, when two numbers are to be added, binary arithmetic is invariably used, and $N$-bit additions are realised using $N$ modulo 2 adders (i.e. the familiar half and full adders) connected in cascade. One of the disadvantages is that the final sum is obtained only after the internally generated "carries" have propagated through the whole word-length.

The trend towards the widespread use of large-scale integrated (L.S.I.) digital circuits is leading to the hardware design of arithmetic circuits based on radices greater than 2. The immediate consequences are the reduction of packages, the simplification of interconnections, and a relatively fast circuit operation because "carries" are now between groups of digits, the size of the group depending of the radix used and the degree of parallelism required.

We will study here the specific case when the radix is of the form $2^N$, $N$ a non-zero integer. Using the general model in Fig. 5.0, the F.S.M. model of a radix - $2^N$ "half adder" is easily derived by letting

$$ A = I, \quad B = S, \quad C_0 = G_0^x \quad \text{and} \quad C_1 = G_u $$

where

- $C_0$ is the modulo $2^N$ sum of $A$ and $B$,
- $C_1$ is the carry-out of the "half adder",

and

$A$ and $B$ are the two $N$-bit numbers that are to be added.

The block diagram of this radix - $2^N$ half-adder is shown in Fig. 5.1.
5.2.0 Example.

Consider the addition of two 3-bit numbers A and B, (i.e. \( N = 3 \)). The corresponding modulo \(2^3\) sum and the carry tables are shown in Tables 5.0(a) and (b) respectively. These tables may be now regarded as the state and output tables, respectively, of the F.S.M. equivalent of this radix - \(2^3\) half-adder, and implemented using memory modules as look-up tables as shown in Fig. 5.2. The sum and carry circuits require a \(64 \times 3\) and a \(64 \times 1\) W-b memory stores respectively.

This direct implementation, however, will not be practical for operands having large word lengths.

5.2.0.0 S.P. partitions of radix - \(2^3\) half-adder.

For the moment consider only the modulo \(2^3\) sum table (i.e. Table 5.0(a)) that is realised by the state machine shown in Fig. 5.2, with the elements of A and B being regarded as the set of machine inputs and internal states respectively.

This F.S.M. possesses the following S.P. partitions,

\[ \pi_1 = \{0,2,4,6; 1,3,5,7\} \quad \text{and} \quad \pi_2 = \{0,4; 2,6; 1,5; 3,7\} \]

A cascade realisation is thus possible using either \( \pi_1 \) or \( \pi_2 \) in conjunction with a non-S.P. partition \( \tau_1 \) or \( \tau_2 \) respectively, such that

\[ \pi_1 \cdot \tau_1 = \pi_2 \cdot \tau_2 = \pi(0), \text{ the zero partition.} \]

Possible values of \( \tau_1 \) and \( \tau_2 \) are

\[ \tau_1 = \{0,1; 2,3; 4,5; 6,7\} \quad \text{and} \quad \tau_2 = \{0,1,2,3; 4,5,6,7\}. \]
Table 5.0. (a) Modulo $2^3$ sum and (b) carry output tables for radix $-2^3$ "half-adder".

<table>
<thead>
<tr>
<th>B (state)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

(a) Modulo $2^3$ sum and (b) carry output tables for radix $-2^3$ "half-adder".
Fig. 5.2. Direct memory realisation of a radix $-2^3$ 'half-adder'.

Fig. 5.3. Cascade memory realisations of modulo $2^3$ adder using (a) $\pi_1$, $\tau_1$ and (b) $\pi_2$, $\tau_2$. 
Also, the machine input set may be partitioned in a similar way. The state tables for the component machines of the realisation using $\pi_1$ and $\tau_1$ are shown in Tables 5.1(a) and (b) while those of the realisation using $\pi_2$ and $\tau_2$ are given in Tables 5.2(a) and (b).

The corresponding block diagrams of these two possible cascade realisations are shown in Figs. 5.3(a) and (b), with corresponding memory storage of $\{(4 \times 1) + (64 \times 2)\} \text{ W-b}$ and $\{(16 \times 2) + (64 \times 1)\} \text{ W-b}$ respectively, i.e. 132 and 96 bits. (It is useful to note here the advantages of using a successor component having as few blocks as possible).

A much better realisation, however, will be to use $\pi_2$ and $\tau_2$ to obtain $\pi(0)$, with $\pi_2$, in its turn, being derived from $\pi_1$ and $\tau'_1$, where

$$\pi_1 \cdot \tau'_1 = \pi_2, \quad \text{i.e. } \tau'_1 = \{0415; 2637\}.$$

The state table for this 'successor' component is drawn in Table 5.3, and the overall realisation of the modulo $2^3$ addition using $\pi_1$, $\tau'_1$ and $\tau_2$ is shown in Fig. 5.4. This realisation uses three memory modules, of overall capacity of 84 bits, and compares favourably with the two realisations discussed previously. Each memory circuit is a single output store and the interconnection pattern between the memory store is highly regular.

This particular form is known as a loop-free implementation and will now be discussed in detail.
Table 5.1. State tables for component machines of cascade realisation shown in Fig. 5.3(a).
<table>
<thead>
<tr>
<th>State</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
</tr>
<tr>
<td>Q</td>
<td>Q</td>
<td>P</td>
<td>S</td>
<td>R</td>
</tr>
<tr>
<td>R</td>
<td>R</td>
<td>S</td>
<td>Q</td>
<td>P</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
<td>R</td>
<td>P</td>
<td>Q</td>
</tr>
</tbody>
</table>

\[ \pi_2 = \{0, 4; 2, 6; 1, 5; 3, 7\} \]

\[ = \{ P; Q; R; S \} \]

\[ = \{ E; F; G; H \}. \]

\[ \tau_2 = \{0, 2, 1, 3; 4, 5, 6, 7\} \]

\[ = \{ p; q \} = \{ e; f \}. \]

Table 5.2(a). State table of predecessor component of cascade realisation shown in Fig. 5.3(b).
| State | \( p \) | \( q \) | \( p \) | \( q \) | \( p \) | \( q \) | \( p \) | \( q \) | \( p \) | \( q \) | \( p \) | \( q \) | \( p \) | \( q \) | \( p \) | \( q \) | \( p \) | \( q \) | \( p \) | \( q \) | \( p \) | \( q \) | \( p \) | \( q \) | \( p \) | \( q \) | \( p \) | \( q \) | \( p \) | \( q \) |
|-------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
| \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) | \( p \) |
| \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) | \( q \) |

Augmented input

Table 5.2(b) State table of successor component of cascade realisation in Fig. 5.3(b).
\[ t_1' = \{ 0415; 2637 \} \]
\[ = \{ M, N \} = \{ m, n \}. \]

\[ \pi_2 = \{ 0, 4; 2, 6; 1, 5; 3, 7 \} \quad \text{(see Table 5.2(a)).} \]
\[ = \{ P, Q, R, S \} = \{ E, F, G, H \} \]

Hence
\[ t_1' = \{ P, R; Q, S \} = \{ E, G; F, H \}. \]

<table>
<thead>
<tr>
<th>State</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>B</th>
<th>B</th>
<th>B</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>I</td>
<td>I</td>
<td>J</td>
<td>J</td>
<td>I</td>
<td>I</td>
<td>J</td>
<td>J</td>
</tr>
<tr>
<td>M</td>
<td>M</td>
<td>N</td>
<td>M</td>
<td>N</td>
<td>M</td>
<td>N</td>
<td>N</td>
<td>M</td>
</tr>
<tr>
<td>N</td>
<td>N</td>
<td>M</td>
<td>N</td>
<td>M</td>
<td>N</td>
<td>M</td>
<td>M</td>
<td>N</td>
</tr>
</tbody>
</table>

Table 5.3. State table for successor component of the machine realisation of \( \pi_2 \) from \( \pi_1 \) and \( t_1' \).
Fig. 5.4. Loop-free memory realisation of modulo $2^3$ adder using $\pi_1$, $\tau_1'$ and $\tau_2$.

Fig. 5.5. Generalised realisation of modulo $2^N$ adders.
5.2.1 **The general modulo $2^N$ adder.**

The decomposition technique in the example may be generalised for any value of $N$ by using the following result, the proof of which may be found in pages 379–380 of Reference 71.

**Theorem 5.0.** If there exists a set of S.P. partitions \{ $\pi_1, \pi_2, \ldots, \pi_n$ \} for an F.S.M. $M$ such that $\pi_1 \geq \pi_2 \geq \ldots \geq \pi_n$, and $\pi_n = \pi(0)$, then $M$ is realisable as a serial loop-free connection of $n$ components $m_1, m_2, \ldots, m_n$ in which $m_i$ is a predecessor of $m_j$ iff $\pi_i \geq \pi_j$. All the components operate concurrently.

5.2.1.0 **Generation of S.P. partitions.**

We have seen in Chapter 3 that the generation of all possible S.P. partitions is initiated by identifying (i.e. put in the same block) all possible pair combinations of the states.

For the case of modulo $2^N$ adders, however, it is sufficient to consider only the identification of 0 and the integer $d$, where

$$d \in \mathbb{N}' = \{1, 2, \ldots, 2^N - 1\}.$$ 

This is because the first row and the first column of the modulo $2^N$ addition table merely duplicate the inputs and the present states respectively. Furthermore, each successive column, going from column 1 to column $2^N - 1$, is identical to its predecessor except that the top entry is shifted to the bottom, and every entry is shifted up by a unit step.

Thus, the identification of 0 and $d$ automatically implies the identification of $(0 + k)$ and $(0 + k) + d$ for all $k$'s where
1 \leq k \leq 2^N - 1. As a consequence, all elements that are d units apart will be identified, and hence for an arbitrary \( a_i \),

\[ 0 \leq a_i \leq 2^N - 1, \] then \( a_i \) and \( a_i + kd \) (modulo \( 2^N \)) will be in the same partition block.

The following lemmas will now be proved. (A useful aid to the proofs is to regard the \( a_i \)'s to be placed consecutively on the circumference of a circle, the 'distance' between \( a_i \) and \( a_{i+1} \) being of 'unit' length).

**Lemma 5.0.** If \( d \) is odd, there are no S.P. partitions apart from the trivial ones \( \pi(I) \) and \( \pi(0) \), the 'identity' and 'zero' partitions respectively.

**Proof.** If \( d = 1 \), then all state elements that are a unit distance from each other will be identified thus leading to a partition block which contains all the state elements, in other words \( \pi(I) \).

Consider now the general case in which \( d = (2q + 1), \)

\[ 1 \leq q \leq 2^{N-1} - 1. \]

For an arbitrary \( a_i \), any state of the form \( kd, k = 1, 2, \ldots \), will be identified with it. That this will eventually lead to \( \pi(I) \) is clearly seen by the following observation.

Consider the case when, starting from \( a_i \) and going around the circle, \( a_i \) is picked up again after \( k \) steps of \( d \) units each, i.e.

\[ a_i + k(2q + 1) \equiv a_i \pmod{2^N}, \]

i.e.

\[ (2q + 1)k \equiv 0 \pmod{2^N}. \]
As the above implies that $2^N$ divides $(2q + 1)k$, and also $(2q + 1)$, being odd, is relatively prime to $2^N$, then $2^N$ must divide $k$, i.e.

$$k = g \cdot 2^N, \quad g = 0, 1, \ldots$$

Since $k > 0$, then $g$ must be greater than zero, and hence the first solution for $k$ is when $g = 1$, leading to $k = 2^N$. Therefore $2^N$ different states will be identified before any starting state is repeated.

**Lemma 5.1.** If $d = 2^p$, $p = 1, 2, 3, \ldots N$, there exists a set of S.P. partitions $\{\pi_1, \pi_2, \pi_3, \ldots \pi_N\}$. Any $\pi_p$, derived from $d = 2^p$, contains $2^p$ blocks of equal size, and if the elements in any one block are arranged in ascending magnitude, adjacent elements will differ by $2^p$ units.

**Proof.** Following similar argument as in Lemma 1, we obtain

$$(2^p)k \equiv 0 \pmod{2^N}$$

i.e. $$k = g \cdot \frac{2^N}{2^p}$$

As $2^p$ always divides $2^N$, repetition of any initial state can occur before the full cycle of $2^N$ steps can be completed.

It follows that the number of elements, $m(\pi_p)$, in one block of $\pi_p$ is given by

$$m(\pi_p) = \frac{2^N}{2^p} = 2^{N-p},$$

and the number of blocks of $\pi_p$, $\hat{\pi}(\pi_p)$, is given by
Lemma 5.2. Let \( d,D \) be two integers, \( 1 < d,(D) \leq 2^N \).

If \( d \) divides \( D \), then \( \pi_d \geq \pi_D \).

Proof. Let \( a_i \) and \( a_j \) be two elements in a block of \( \pi_D \).

Then, by construction, we have

\[
a_j \equiv a_i + kD \pmod{2^N}.
\]

Since \( D \) is divisible by \( d \), i.e. \( D = \ell d, \ell \) an integer, then

\[
a_j \equiv a_i + k\ell d \pmod{2^N},
\]

implying that \( a_i \) and \( a_j \) are also contained in a block of \( \pi_d \).

5.2.1.1 Loop-free realisation of adders modulo \( 2^N \).

The following theorem follows naturally from the three lemmas we discussed in the previous section.

Theorem 5.1. The F.S.M. model of a general modulo \( 2^N \)
adder possesses \( N \) S.P. partitions \( \pi_1, \pi_2, \ldots, \pi_p, \ldots, \pi_N \)
such that

\[
\pi_1 \geq \pi_2 \geq \ldots \geq \pi_p \geq \ldots \geq \pi_N = \pi(0).
\]

In the implementation of the adder, any of these partitions
\( \pi_p \) can be used with any non-S.P. partition \( \tau_p \), so long as

\[
\pi_p \cdot \tau_p = \pi(0).
\]
A more economical implementation, however, will be to use all the S.P. partitions in a systematic way as follows.

Consider $\pi_{N-1}$. A valid realisation will be to use $\tau_{N-1}$ given by

$$\pi_{N-1} \cdot \tau_{N-1} = \pi_N = \pi(0).$$

From Lemma 5.1 we obtain

$$m(\pi_{N-1}) = \frac{2^N}{2^{N-1}} = 2.$$

Hence $\tau_{N-1}$ will have to be a 2-block partition in order to distinguish between the elements of each block of $\pi_{N-1}$.

$\pi_{N-1}$, in turn, is realised from $\pi_{N-2}$ in the same manner, i.e. using again another 2-block partition $\tau_{N-2}$, such that

$$\pi_{N-2} \cdot \tau_{N-2} = \pi_{N-1}.$$

By repeating this procedure for the remaining S.P. partitions, we arrive at the following iterative relationship,

$$\pi(0) = \tau_{N-1} \cdot \pi_{N-1}$$
$$\downarrow$$

$$\pi_{N-1} = \tau_{N-2} \cdot \pi_{N-2}$$
$$\vdots$$

$$\pi_{p} = \tau_{p-1} \cdot \pi_{p-1}$$
$$\vdots$$

$$\pi_{2} = \tau_{1} \cdot \pi_{1}.$$

Consequently, one can implement an adder modulo $2^N$ as a set of loop-free interconnected component machines as described in Theorem 5.0. As these sub-machines operate concurrently, there
is no carry propagation at all.

If the 'input' A is assigned the same binary code as for the 'present state' B, the resulting hardware implementation using memory modules is as shown in Fig. 5.5.

5.2.1.2 Memory storage reduction.

It will now be shown that the storage required for the loop-free form of adders modulo $2^N$ is considerably less than that required in the direct realisation.

If $M_o$ is the memory storage* of the direct form and $M_r$ the overall storage of all the sub-machines, then

$$M_o = N \times 2^{2N},$$

where $N$ is the word-length of each of the operands $A$ and $B$.

Since $A$ and $B$ are coded in the same way, then

$$M_r = (2^2) + (2^2)^2 + (2^2)^3 + \ldots + (2^2)^N \quad \ldots(5.0)$$

$$= p + p^2 + p^3 + \ldots + p^N \quad \ldots(5.1)$$

where $p = 2^2$. If we multiply (5.1) by $p$, we get

$$pM_r = p^2 + p^3 + p^4 + \ldots + p^{N+1} \quad \ldots(5.2)$$

By subtracting (5.1) from (5.2), we obtain

$$M_r(p-1) = p^{N+1} - p \quad \text{or}$$

$$M_r = \frac{p(p^N - 1)}{p-1}$$

$$= \frac{4}{3} (2^{2N} - 1).$$

* For simplicity of subsequent explanation, the unit for memory storage is understood to be 'word-bits'.
Thus, the reduction ratio $R = \frac{M_r}{M_o}$ is given by

$$R = \frac{M_r}{M_o} = \frac{4}{3} \left( \frac{2^{2N} - 1}{N \times 2^{2N}} \right),$$

and if $2^{2N} \gg 1$, then we have

$$R = \frac{4}{3} \cdot \frac{1}{N},$$

which is a considerable reduction. Also, as can be seen from the graph in Fig. 5.6, this reduction improves, i.e. becomes smaller as the word length $N$ is increased.

5.2.2. Generation of the carry digit.

When $A$ and $B$ are added modulo $2^N$, a table can be drawn to show when a carry digit has to be generated. One such table for $N = 3$ is shown in Table 5.0(b), which is also the output table for the F.S.M. model.

For a general $N$, let the rows and columns of the output table be denoted by $i$ and $j$ respectively, $(i, j = 0, 1, 2, \ldots, (2^N - 1))$. It was observed that below the diagonal described by,

$$i, k \text{ for all } i = 0, 1, \ldots, 2^N - 1 \quad \text{and} \quad k = (2^N - 1) - i,$$

the table entries are all '1's. Again this is clearly illustrated in Table 5.0(b). This fact suggests a simple method of realising the output table.

A carry is generated, i.e. $C_1 = 1$, only if

$$A + B > 2^N - 1, \quad \text{i.e.} \quad A > 2^N - 1 - B.$$
Fig. 5.6. Effect of loop-free decomposition on overall memory storage.

Fig. 5.7. Carry-out circuit of radix $- 2^3$ 'half-adder'.

Fig. 5.8. Stored-logic realisation of a radix $2^N$ 'full-adder'. 
Since the right hand side of the inequality is simply the one's complement of B, then if

\[ B = \sum_{k=0}^{N-1} b_k 2^k, \]

we can write this one's complement \( B' \) as

\[ B' = \sum_{k=0}^{N-1} \bar{b}_k 2^k, \]

where \( b_k = 0 \) or 1 and \( \bar{b}_k \) is the logical negation of \( b_k \).

Thus the output table may be realised using an N-bit inverting network and a standard N-bit M.S.I. binary comparator. This form of realisation for \( N = 3 \) is shown in Fig. 5.7.

Of course, the generation of the carry output may be incorporated in the general loop-free design as discussed in Section 5.2.1.1. by regarding the addition to be modulo \( 2^{N+1} \) instead of \( 2^N \) and assuming the last input bit and state bit to be at a constant '0' value.

### 5.2.3 Addition of "carry-in" digit.

For a full radix-\( 2^N \) adder design, the "carry-in" digit from the previous full adder must be incorporated. If \( S_N \) is the modulo \( 2^N \) sum of A and B and \( C_i \) the "carry-in" digit, then the addition of \( S_N \) and \( C_i \) is carried out in exactly the same way as described in Section 5.2.1.

This time, however, since \( C_i \) is only a one-bit variable, the required storage \( M_r \) is much less and is given by

\[ M_r = 2^2 + 2^3 + \ldots + 2^{N+1} \]

\[ = 4 (2^N - 1) = 4 \cdot 2^N \text{ if } 2^N \gg 1. \]
Therefore, the corresponding ratio \( R = \frac{M}{N_0} \) is given by

\[
R = \frac{4 \cdot 2^N}{N \times 2^{N+1}} = \frac{2}{N}.
\]

The carry-out circuit for this part is relatively simple since a carry is generated only if \( S_N = 2^N - 1 \) and \( C_i = 1 \). Thus we would require only an \((N+1)\)-input AND gate. The block diagram of the complete full adder is shown in Fig. 5.8.

5.3 Radix-\(2^N\) parallel multipliers.

We will now investigate the modelling of a parallel \(N \times N\) bit multiplier by an F.S.M. Two models will be presented, the first being the straightforward application of the general model shown in Fig. 5.0, while the second is derived by regarding the radix-\(2^N\) full multiplication as being equivalent to two multiplications, modulo \(2^N\) and modulo \(2^N - 1\) respectively, operating in parallel.

5.3.0 Example.

Consider the multiplication of two 3-bit numbers \(A\) and \(B\), giving a 6-bit product \(P\). The direct look-up table is given in Table 5.4, and the corresponding memory module implementation, requiring 384 storage bits, is shown in Fig. 5.9.

The F.S.M. model of this multiplier is obtained by separating the direct table into two simpler component tables as shown in Tables 5.5(a) and (b). The latter is simply a modulo \(2^3\) multiplication table, while the former consists of the values for the most significant 3 bits of the product \(P\). These Tables (a) and
(b) may now be regarded as the output and state tables of the equivalent F.S.M. respectively.

5.3.1 **The general N x N bit multiplier.**

The F.S.M. model of the general N x N bit parallel multiplier will now be derived.

When two N-bit numbers, A and B, are multiplied, the result P is a 2N-bit product, i.e. if

\[
A = \sum_{i=0}^{N-1} a_i 2^i \quad \text{and} \quad B = \sum_{i=0}^{N-1} b_i 2^i
\]

where \(a_i = 0 \text{ or } 1\) and \(b_i = 0 \text{ or } 1\), then

\[
A \times B = P = \sum_{j=0}^{2N-1} p_j 2^j, \quad p_j = 0 \text{ or } 1 \quad \ldots (5.3)
\]

This product \(P\) can be expressed as a 2-digit number in the radix \(2^N\) as follows:

\[
P = \sum_{j=0}^{2N-1} p_j 2^j
\]

\[
= \sum_{m=N}^{2N-1} p_m 2^m + \sum_{k=0}^{N-1} p_k 2^k
\]

\[
= p_1 (2^N)_1 + p_o (2^N)_o \quad \ldots (5.4)
\]

where

\[
p_1 = \sum_{m'=0}^{N-1} (p_{m'}) 2^{m'} \quad \text{and} \quad p_o = \sum_{k'=0}^{N-1} (p_{k'}) 2^{k'} \quad \ldots (5.5)
\]
<table>
<thead>
<tr>
<th>A</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>18</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>25</td>
<td>30</td>
<td>35</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>6</td>
<td>12</td>
<td>18</td>
<td>24</td>
<td>30</td>
<td>36</td>
<td>42</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
<td>14</td>
<td>21</td>
<td>28</td>
<td>35</td>
<td>42</td>
<td>49</td>
</tr>
</tbody>
</table>

Table 5.4. Direct multiplication table for a 3 bit \( \times \) 3 bit parallel multiplier.
Fig. 5.9. A 3-bit parallel stored-logic multiplier.

Fig. 5.10. F.S.M. model of a 3-bit parallel multiplier.
Table 5.5. (a) Output and (b) state tables for the F.S.M. model of a 3-bit parallel multiplier.
The direct look-up table represented by equation (5.3) is now separated into two simpler tables, the $P_I$ and $P_O$ tables represented by equations (5.4) and (5.5), where $P_I$ is the $N$ most significant bits of the real product $P$, and $P_O$ is the $N$-bit modulo $2^N$ product. The F.S.M. model is obtained by regarding $P_I$ and $P_O$ as the $G_u$ and $G_z$ respectively of the general model as shown in Fig. 5.0.

5.3.1.0 Example.

For the 3-bit multiplier we discussed before, let $A = 7$ and $B = 5$. Thus we have

$$A \times B = 7 \times 5 = 35$$

$$= (4 \times 8) + 3$$

$$= 4 \times (2^3)^1 + 3 \times (2^3)^0$$

Here $P_I = 4$ and $P_O = 3$. This particular operation is illustrated in Fig. 5.10.

5.3.2 Decomposition of the F.S.M. multiplier.

Considering the $P_O$ table given in Table 5.5(b), the following S.P. partitions are found,

$$\pi_1 = \{0, 2, 4, 6; 1, 3, 5, 7\}.$$

$$\pi_2 = \{0, 4; 1, 5; 2, 6; 3, 7\}$$

Following a similar argument to that used in deriving the loop-free realisation of adders modulo $2^n$, the complete multiplier is realised by using two non-S.P. partitions $\tau_1$ and $\tau_2$ such that,
\[ \pi_1 \cdot \tau_1 = \pi_2 \text{ and } \pi_2 \cdot \tau_2 = (0). \]

Possible values for \( \tau_1 \) and \( \tau_2 \) are

\[ \tau_1 = \{0, 4, 1, 5; 2, 6, 3, 7\} \]
\[ \tau_2 = \{0, 1, 2, 3; 4, 5, 6, 7\}. \]

The \( P_0 \) table may first be rearranged according to \( \pi_1 \) and \( \pi_2 \), as shown in Tables 5.6(a) and (b), and the \( \pi_1 \) and \( \pi_2 \) images of the F.S.M. multiplier, viz. \( M_{\pi_1} \) and \( M_{\pi_2} \) respectively are obtained by considering operations only between partition blocks. The corresponding 'state' tables are shown in Tables 5.7(a) and (b).

Also, the two successor components derived from the non-S.P. partitions \( \tau_1 \) and \( \tau_2 \) are shown in Tables 5.8 and 5.9 respectively.

It is interesting to note that \( M_{\pi_1} \) and \( M_{\pi_2} \) are isomorphic to a modulo 2 and a modulo 4 multipliers respectively. This observation leads to the following theorem.

**Theorem 5.2.** A partition \( \pi \) that has S.P. for a modulo \( 2^N \) addition table also has S.P. for a modulo \( 2^N \) multiplication table, where \( \pi \) is defined as in Lemma 5.1.

**Proof.** Let \( a_i \) and \( a_j \) be two elements in a block of \( \pi \) and assume that \( a_j > a_i \). Then we have

\[ a_j - a_i = kd, \quad d = 2^p. \]

Multiplying \( a_i \) and \( a_j \) by an arbitrary element \( b, \ 0 \leq b \leq 2^N-1 \), we get

\[ c_j \equiv b \times a_j, \quad c_i \equiv b \times a_i \quad (\text{modulo } 2^N), \]
Table 5.6. The modulo $2^3$ multiplication table organised by (a) $\pi_1$ and (b) $\pi_2$.
$\tau_1 = \{0,2,4,6; 1,3,5,7\} = \{A,B\} = \{E,F\}$

$\tau_1 = \{0,4,1,5; 2,6,3,7\} = \{a,b\} = \{e,f\}$

<table>
<thead>
<tr>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>

$M_{\tau_1}$

(a)

$\tau_2 = \{0,4; 1,5; 2,6; 3,7\}$

= \{P, Q, R, S\}

= \{I, J, K, L\}

$\tau_2 = \{0,1,2,3; 4,5,6,7\} = \{p,q\} = \{m,n\}$

<table>
<thead>
<tr>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>Q</td>
<td>P</td>
<td>Q</td>
<td>R</td>
</tr>
<tr>
<td>R</td>
<td>P</td>
<td>R</td>
<td>P</td>
</tr>
<tr>
<td>S</td>
<td>P</td>
<td>S</td>
<td>R</td>
</tr>
</tbody>
</table>

$M_{\tau_2}$

(b)

Table 5.7. State tables for the (a) $\tau_1$ and (b) $\tau_2$ images of a 3-bit F.S.M. multiplier.
Augmented input
\[ \tau_1 = \{A; B\} = \{E; F\} \]
\[ = \{P, R; Q, S\} = \{I, K; J, L\} \]
\[ \tau_1 = \{a; b\} = \{e; f\} \]
\[ = \{P, Q; R, S\} = \{I, J; K, L\} \]

Also
\[ F \rightarrow (A, a) \quad Q \rightarrow (B, a) \]
\[ R \rightarrow (A, b) \quad S \rightarrow (B, b) \]

Table 5.8. State table of successor machine \( M_{\tau_1} \).
<table>
<thead>
<tr>
<th>State</th>
<th>P</th>
<th>P</th>
<th>P</th>
<th>P</th>
<th>P</th>
<th>P</th>
<th>P</th>
<th>Q</th>
<th>Q</th>
<th>Q</th>
<th>Q</th>
<th>Q</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>I</td>
<td>I</td>
<td>J</td>
<td>J</td>
<td>K</td>
<td>K</td>
<td>L</td>
<td>L</td>
<td>I</td>
<td>I</td>
<td>J</td>
<td>J</td>
<td>K</td>
</tr>
<tr>
<td></td>
<td>m</td>
<td>n</td>
<td>m</td>
<td>n</td>
<td>m</td>
<td>n</td>
<td>m</td>
<td>n</td>
<td>m</td>
<td>n</td>
<td>m</td>
<td>n</td>
<td>m</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>p</th>
<th>p</th>
<th>p</th>
<th>p</th>
<th>p</th>
<th>p</th>
<th>p</th>
<th>q</th>
<th>p</th>
<th>p</th>
<th>q</th>
<th>p</th>
<th>q</th>
<th>p</th>
<th>q</th>
<th>p</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>R</th>
<th>R</th>
<th>R</th>
<th>R</th>
<th>R</th>
<th>R</th>
<th>R</th>
<th>S</th>
<th>S</th>
<th>S</th>
<th>S</th>
<th>S</th>
<th>S</th>
<th>S</th>
<th>S</th>
<th>S</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>I</td>
<td>I</td>
<td>J</td>
<td>J</td>
<td>K</td>
<td>K</td>
<td>L</td>
<td>L</td>
<td>I</td>
<td>I</td>
<td>J</td>
<td>J</td>
<td>K</td>
<td>K</td>
<td>L</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td></td>
<td>m</td>
<td>n</td>
<td>m</td>
<td>n</td>
<td>m</td>
<td>n</td>
<td>m</td>
<td>n</td>
<td>m</td>
<td>n</td>
<td>m</td>
<td>n</td>
<td>m</td>
<td>n</td>
<td>m</td>
<td>n</td>
<td>n</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>p</th>
<th>p</th>
<th>p</th>
<th>p</th>
<th>p</th>
<th>q</th>
<th>q</th>
<th>q</th>
<th>q</th>
<th>q</th>
<th>p</th>
<th>q</th>
<th>q</th>
<th>q</th>
<th>p</th>
<th>q</th>
</tr>
</thead>
</table>

| State | q | p | p | p | q | q | q | q | q | q | p | q | q | q | p | q |

Table 5.9. State table of successor machine $M_{r_2}$. 
\[ i.e. \quad b \times a_j = q_j 2^N + c_j \quad \text{and} \]
\[ b \times a_i = q_i 2^N + c_i \]
\[ \therefore \quad c_j - c_i = b(a_j - a_i) + (q_i - q_j) 2^N \]
\[ = b(kd) + (q_i - q_j) 2^N \quad \ldots (5.6) \]

Since \( d = 2^p \) and \( 2^N \) is divisible by \( 2^p \), the R.H.S. of equation (5.6) is divisible by \( 2^p \). Consequently \( 2^p \) divides \( c_j - c_i \), i.e. we may write
\[ c_j - c_i = k' 2^p, \quad i.e. \]
\[ \delta(a_j, b) \equiv \delta(a_i, b)(\pi_p). \]

Thus \( c_j \) and \( c_i \), the 'next states' of \( a_j \) and \( a_i \) respectively, are in the same block of \( \pi_p \) and therefore \( \pi_p \) has S.P.

It must be mentioned here that as a consequence of the above theorem, it does not mean that the S.P. partitions for the modulo \( 2^N \) multiplication are confined only to those having the form as in \( \pi_p \). The partition \( \{0; 1,2,3,4,5,6,7\} \) for instance, has S.P. Eventually, however, since one has to consider the interconnection of both adders and multipliers, then the use of compatible S.P. partitions for both will eliminate the need for coding and decoding between partitions having different structures in terms of the number of partition blocks and their sizes.

The \( P_1 \) or 'output' table is more difficult to analyse than the \( P_0 \) or 'state' table. This is because it contains no two rows that are identical, thus leading to output-consistent partitions, which may lead to a state reduction. There is also no obvious internal structure or pattern that we can exploit. Consequently,
the problem is also difficult to generalise to an arbitrary N.

Although the $P_1$ or output function can be implemented using conventional combinational logic techniques, a better method is presented in the following section.

5.3.3 Improved model of N-bit parallel multiplier.

A model will now be derived for the $N \times N$ bit multiplication which result in both $P_0$ and $P_1$ tables having regular algebraic structures.

The product $P$ is first written in two different ways, as a 2-digit number in the radices $2^N$ and $2^N-1$ respectively, i.e.

$$P = P_1 2^N + P_0 \quad \cdots (5.7)$$

and

$$P = Q_1 (2^N-1) + Q_0 \quad \cdots (5.8)$$

The implementation of $P_0$ has already been discussed and that of $Q_0$ will be analysed in detail in Chapter 7. Our immediate problem now is to determine $P_1$ in equation (5.7) knowing only $P_0$ and $Q_0$.

Equation (5.7) may be subtracted from equation (5.8) to give

$$P_1 2^N - Q_1 (2^N-1) = Q_0 - P_0 \quad \cdots (5.9)$$

The L.H.S. of (5.9) may be written as

$$P_1 2^N - P_1 + P_1 - Q_1 (2^N-1)$$

$$= P_1 (2^N-1) + P_1 - Q_1 (2^N-1) \quad \cdots (5.10)$$

* Another obvious solution is to regard the $N \times N$ bit multiplication as that of modulo $2^{2N}$, i.e. one simply extends the range of "modulo multiplication" by letting $N \rightarrow 2N$. 

Substituting (5.10) into (5.9), we get

\[ P_1 = Q_o - P_o + (Q_1 - P_1)(2^N-1) \]

\[ = R_o + K(2^N-1) \] \hspace{1cm} \ldots(5.11) \]

where \( R_o = Q_o - P_o \) and \( K = Q_1 - P_1 \).

The maximum value of \( P_1 \), \( P_{\text{max}} \) say, is given by

\[ P_{\text{max}} = A_{\text{max}} \times B_{\text{max}} \]

\[ = (2^N-1)(2^N-1) \text{ since } A, B \text{ are } N\text{-bit numbers.} \]

\[ \therefore \quad P_{\text{max}} = (2^N-1)2^N - (2^N-1) \]

\[ = (2^N-1)2^N - 2^N + 1 \]

\[ = (2^N-2)2^N + 1 \]

Hence \( P_{1\text{max}} = (2^N-2) \)

\[ \therefore \text{ Using equation (5.11), since } P_1 \leq P_{1\text{max}} \text{, then} \]

\[ R_o + K(2^N-1) \leq (2^N-2) \] \hspace{1cm} \ldots(5.12) \]

Since by definition \( 0 \leq P_o \leq 2^N-1 \) and \( 0 \leq Q_o \leq 2^N-2 \), then, since \( R_o = Q_o - P_o \), it is at its maximum value when \( Q_o = 2^N-2 \) and \( P_o = 0 \).

\[ \therefore \quad R_{o\text{max}} = 2^N-2 \]

Substituting \( R_{o\text{max}} \) into equation (5.12), we obtain

\[ (2^N-2) + K(2^N-1) \leq (2^N-2) \]

\[ \therefore \quad K = 0. \]
Similarly, $R_o$ is minimum when

$$Q_o = 0 \quad \text{and} \quad P_o = 2^{N-1}, \quad \text{and hence}$$

$$R_{o\min} = -(2^{N-1})$$

Since the minimum value of $P_1$ is 0, then, $K$ in equation (5.11) must be 1. Consequently, equation (5.11) may be written either as

$$P_1 = R_o = Q_o - P_o \quad \ldots (5.13)$$

or

$$P_1 = R_o + 1 \times (2^{N-1})$$

$$= Q_o - P_o + (2^{N-1}) \quad \ldots (5.14)$$

Equation (5.14) is more general and will 'cover' equation (5.13), because adding $(2^{N-1})$ to any number $x$, $0 \leq x \leq 2^N-2$, will not alter its value provided any end-round carry is taken into account, since

$$(2^{N-1}) + x = (2^{N-1}) + 1 + (x-1)$$

$$= 2^N + (x-1),$$

where $2^N$ is now the overflow or carry bit. Adding this to the least-significant bit of $(x-1)$, we obtain

$$(x-1) + 1 = x.$$
i.e. $P_1 = Q_o + P'_o$ 

...(5.14) 

where $P'_o$ is the additive inverse of $P_o$ with respect to $(2^N-1)$. 

We have now, in effect, modelled the $N \times N$ bit multiplication as two simpler multiplications in parallel, viz. that of modulo $2^N$ and modulo $(2^N-1)$. Consequently, the original F.S.M. multiplier can be regarded as two simpler F.S.M. multipliers operating in parallel. 

The corresponding block diagram of this parallel realisation is shown in Fig. 5.11. As the modulo $2^N$ product is already in the binary form no decoding is necessary. $P_1$ is easily obtained from $Q_o$ and $P_o$ by using a conventional $2^N$ adder with an end-round carry. Although there will be two representations for zero, this is not a problem since we know that

$$P = (2^N-1)2^N + P_o$$ does not happen since $P_1 \leq (2^N-2)$. 

5.4 Conclusions. 

Stored-logic radix $2^N$ full adders and multipliers are analysed on a systems level by modelling them as finite-state sequential machines. The algebraic structures of these F.S.M's are then analysed using S.P. partitions. For the F.S.M. models of both the modulo $2^N$ sum and product, respectively, of two $N$-bit numbers, theorems have been derived showing that these F.S.M's possess cascade loop-free decomposition structures. The corresponding implementations require substantially less memory storage than those of the direct form, and this advantage improves with increasing word-length $N$. 


Fig. 5.11. Improved model of a general $N \times N$ bit parallel multiplier.
Two models have been found for the radix - $2^N$, i.e. $N \times N$ bit, parallel multiplier. The second model is extremely useful because although it requires some simple additional circuitry, it enables the $N$-bit most significant half of the multiplication product to be determined in a systematic way for a general $N$. 
APPENDIX 5.0

Some observations on the F.S.M. model of stored-logic arithmetic circuits.

Although the stored-logic arithmetic circuits that we are modelling as F.S.M's are strictly combinational switching circuits (thus necessitating the imaginary feedback from \( G_2 \) to \( B \) in Fig. 5.0), the approach enabled the application of the useful results from the structure theory of machine decompositions. Furthermore, some familiar arithmetic units do have real feedbacks, e.g. digital accumulators. Thus the model is quite general.

There is however a theoretical constraint. At the particular 'instant' that the product \( A \times B \) is required it has to be assumed that the result of a 'previous' multiplication is such that its less significant half \( G_2 \) must be equal to \( B \). This implies that from a starting state \( s_i \) there is an input sequence \( \bar{x} \) such that

\[
\delta(s_i, \bar{x}) = B.
\]

This constraint is only academic since in practice, this condition is satisfied all the time.
APPENDIX 5.1.

carry input. Multiplication by \(-1\) is just the bit-by-bit inversion of the data word, and multiplication by \(2^{-j}\), to normalise the transform, is only a cyclic shift of the word \(j\) places to the left.

![Fig. 1 d against k](image)

The system shows relatively high error correction against loading performance. Fig. 1 shows, for a 16-channel system, the minimum weights obtained for different values of \(k\), the number of channels in use. Since the system is a linear code, these values can be taken as the minimum distance of the code \(d(0)\). The number of errors that can be corrected is then \((d/2) - 1\); for this code, all minimum weights are even. The performance is independent of \(p\), the modulo number. To obtain this performance, the carriers must be selected properly, otherwise a much lower bound will be obtained; namely, \(d = 4\) for half rate, \(d = 8\) quarter rate, and so on.

In short, a multiplexing system has been described that compares very favourably with existing systems. Unfortunately, for maximum performance, the carriers must be properly selected. At the receiver end, the decoding procedure for correcting more than the trivial 1-error case is very complex.

E. INSAM  
1st May 1974

Electronics Department  
Chelsea College  
London SW6 5PR, England

References

HALFADDERS MODULO \(2^n\) USING READ-ONLY MEMORIES

Indexing terms: Adders, Digital arithmetic, Read-only storage, Sequential machines

Halfadders modulo \(2^n\) are regarded as finite-state sequential machines, and are implemented with read-only memories. The application of the theory of ‘closed’ partitions is shown to lead to considerable savings in the memory storage required, which improves with increasing word lengths, and gives a very regular interconnection pattern and parallel operation.

Introduction: With the advent of m.s.i./l.s.i. techniques, the trend in the design of logic systems is moving from the discrete-logic-gate level towards the system and subsystem level. Thus there is a case for investigating the hardware realisation of arithmetic operations in number systems having radices greater than two. In this letter, half adders modulo \(2^n\), where \(N\) ranges through the set of positive integers, are considered as basic arithmetic modules, and are studied to find whether they contain useful algebraic structures that can lead to practical and economical hardware implementations.

Table 1 MODULO-2\(^3\) SUM TABLE

<table>
<thead>
<tr>
<th>(A)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>(B)</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 2 CARRY TABLE

<table>
<thead>
<tr>
<th>(A)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>(B)</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Example: Consider adding two numbers \(A\) and \(B\) modulo 8, i.e. \(N = 3\). The modulo-sum-and-carry tables are shown in Tables 1 and 2, respectively. Two read-only memories acting as table-lookup units may be used as the hardware. As the two operands are used to address \(2^{(3+2)}\) memory locations for each of the two r.o.m.s, and, since each location for the modulo sum and carry tables contains a 3-bit and a 1-bit word, respectively, the memory storage required will correspondingly be 192 and 64 bits. For large word lengths, however, this direct implementation will lead to excessive storage.

Fortunately, the memory storage can be reduced considerably by applying the theory of ‘closed’ partitions\(^5,6\) to decompose the direct realisation into an interconnection of smaller and simpler substructures. This is done by regarding Tables 1 and 2 as the state transition and output tables, respectively, of a finite-state machine (f.s.m.), having \(A\) and \(B\) as the ‘input’ and ‘internal state’. Partitioning the machine states,\(^3\) we find the following nontrivial ‘closed’ partitions:

\[
\pi_1 = (0, 2, 4, 6/1, 3, 5, 7) \quad \pi_2 = (0, 4/2, 6/1, 5, 7, 3)
\]

A ‘serial’ decomposition is possible using either \(\pi_1\) or \(\pi_2\) in conjunction with a nonclosed partition \(\lambda_1\) or \(\lambda_2\), respectively, provided that \(\pi_1 \lambda_1 = \pi_2 \lambda_2 = \pi(0)\), where

\[
\pi(0) = (0/1 \mid 2/3 \mid 4/5 \mid 6/7)
\]

is the ‘zero’ partition, and the \(\lambda\) signifies a partition multiplication. Notice, however, that, since \(\pi_1\) is ‘greater’ than \(\pi_2\), \(\pi_2\) in turn, can be derived from \(\pi_1\) and \(\lambda_1\), where \(\lambda_1\) is another nonclosed partition such that \(\pi_1 \lambda_1 = \pi_2\). Thus we obtain

\[
\lambda_2 = (0, 2, 1, 3/4, 6, 5, 7) \quad \lambda_1 = (0, 4, 1, 5/2, 6, 3, 7)
\]

The hardware for this form of realisation is shown in Fig. 1, where the ‘input’ has been given the same assignment as the ‘internal state’. The overall memory storage of the three r.o.m. modules used is only 84 bits for the modulo-8 sum, compared with the 192 bits obtained previously. Also, each of the modules is a single-output r.o.m. and the interconnection pattern is very regular. The above implementation is known as the loopfree realisation of an f.s.m.\(^4\)
CHAPTER 6

NOVEL METHOD OF MODULO $2^N$ MULTIPLICATION
USING CONSTRAINED OPERANDS

6.0 Introduction.

In the previous chapter, we saw that a modulo $2^N$ multiplier, modelled as an F.S.M., may be realised as a cascade loop-free interconnection of submachines. As indicated before, this is just one possible structure, and it would be more useful if there exist one or more parallel decompositions. Besides, each sub-machine in the loop-free decomposition does not appear to possess a structure regular enough to be generalised. The modulo $2^N$ multiplication table, as it stands however, is not as easy to analyse as the modulo $2^N$ addition table.

In this chapter, a novel technique is presented in which the multiplication table may be modified in such a way that it is then possible to determine a definite algebraic structure that may be generalised to arbitrary N. Its features make it an interesting alternative to the loop-free configuration. This approach was initiated by the following observation.

6.1 Observations.

Consider the modulo $2^3$ multiplication table discussed in Section 5.3.0 and shown in Table 5.4. Some of the possible S.P. partitions for this multiplier are *

* It may be noted that $\pi_2$ and $\pi_3$ cannot be derived using the loop-free structure described by Theorems 5.1 and 5.2.
\[ \pi_1 = \{ 0, 4; 2, 6; 1, 5; 3, 7 \} \]
\[ \pi_2 = \{ 0, 4; 2, 6; 1, 3; 5, 7 \} \]
\[ \pi_3 = \{ 0, 4; 2, 6; 1, 7; 3, 5 \} . \]

In the above partitions, it is observed that their first two blocks are identical, i.e. \( (0, 4) \) and \( (2, 6) \). Also, if we form the partition product of any two of the above partitions we will obtain \( \pi_4 \) where
\[ \pi_4 = \{ 0, 4; 2, 6; 1; 3; 5; 7 \} . \]

If we now restrict the values of the operands \( A \) and \( B \), and the multiplication product to the set \( (1, 3, 5, 7) \), then the following partitions may be derived from \( \pi_1, \pi_2, \pi_3 \), i.e.
\[ \pi_a = \{ 1, 5; 3, 7 \} \]
\[ \pi_b = \{ 1, 3; 5, 7 \} \]
\[ \pi_c = \{ 1, 7; 3, 5 \} . \]

Any two of the above S.P. partitions may be used in a parallel realisation to obtain the modified modulo 2^3 multiplication table, e.g.
\[ \pi_a \cdot \pi_b = \pi'(0) = \{ 1, 3, 5, 7 \} . \]

These observations suggest that parallel decompositions are possible if the original multiplier is modified by restricting the operands only to odd values. The corresponding table is shown in Table 6.0. The actual product may then be obtained using a simple correction circuit.
Table 6.0. Reduced modulo $2^3$ multiplication table.

<table>
<thead>
<tr>
<th>$\times$</th>
<th>16</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
<th>11</th>
<th>13</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>9</td>
<td>15</td>
<td>5</td>
<td>11</td>
<td>1</td>
<td>7</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>15</td>
<td>9</td>
<td>3</td>
<td>13</td>
<td>7</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>5</td>
<td>1</td>
<td>15</td>
<td>13</td>
<td>11</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>11</td>
<td>13</td>
<td>15</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>11</td>
<td>1</td>
<td>7</td>
<td>13</td>
<td>3</td>
<td>9</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>13</td>
<td>7</td>
<td>1</td>
<td>11</td>
<td>5</td>
<td>15</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
<td>13</td>
<td>11</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.1. Reduced modulo $2^4$ multiplication table.
6.2 Modulo $2^N$ multiplication using 'forced' operands and product correction.

The approach proposed is based on a reduced multiplier, which is the original modulo $2^N$ multiplier whose operands and product are constrained to take on only the odd values from the set

$$Z_N = \{ 0, 1, 2, \ldots, 2^N - 1 \}$$

Hence, if $A, B$ and $A', B'$ are the operands to the original and reduced multipliers respectively, then $A', B' \in Z_D$, where

$$Z_D = \{ x : x \text{ odd integer, } 1 \leq x \leq 2^N - 1 \}.$$  

These modified operands may be derived from the originals by the mappings $g_A$ and $g_B$, where

$$g_A : A \rightarrow A' = A + c$$

and

$$g_B : B \rightarrow B' = B + d$$

such that

$$c = \begin{cases} 
0 & \text{for } A \text{ odd} \\
1 & \text{for } A \text{ even} 
\end{cases} \quad d = \begin{cases} 
0 & \text{for } B \text{ odd} \\
1 & \text{for } B \text{ even} 
\end{cases}$$

In other words, whenever any of the original operands is even, we 'forced' it to be odd by adding a '1' to it. In practice, this simply means that the least significant bits (L.S.B's) of $A'$ and $B'$ are assumed to be '1' all the time. Similarly for the product $P'_o$ of the reduced modulo $2^N$ multiplier. The remaining $N-1$ bits of

* The meaning of "reduced" here is different from that defined in Chapter 3 in the context of state minimisation.
A' and B' are identical to those of A and B.

Multiplying these 'forced' operands, we obtain,

\[ P' = A' \times B' = (A+c)(B+d) \mod 2^N \]

\[ \equiv AB + dA + cB + cd \mod 2^N. \]

Hence the required product \( P_0 \equiv A \times B \mod 2^N \) is given by

\[ AB \equiv (A'\times B') - (dA + cB + cd) \mod 2^N \quad \cdots (6.0) \]

This congruence relationship expresses our proposed multiplication scheme, in which \((A' \times B')\) describes the reduced multiplication and \(C = (dA + cB + cd)\) the correction required to obtain the actual product. The block diagram of the overall configuration is shown in Fig. 6.0.

The various values of \(C\) corresponding to all possible combinations of \(a\) and \(b\), the L.S.B's of \(A\) and \(B\) respectively, are given below.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A+B+1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

This leads to a very simple correction circuit consisting of a modulo \(2^N\) adder which is just the conventional \(2^N\) adder with the carry digit excluded, and two gating circuits, each consisting of \((N-1) 2\)-input AND gates and one inverter. The output \(C\) from this correction unit is then subtracted from \(P'_0\) to obtain the actual product \(P_0\). This correction process is shown in Fig. 6.1. Further
Fig. 6.0. Block diagram of modulo $2^N$ multiplication using 'forced' operands and product correction.
Fig. 6.1. Correction circuit for multiplication product.
simplification in its detailed realisation is possible and will be described in the following example.

6.2.0 Example.

Consider first a modulo $2^4$ multiplier whose operands have been constrained to only odd values. The corresponding multiplication table is shown in Table 6.1. The F.S.M. model of this reduced multiplier has the following basic S.P. partitions.

- $\pi_1 = \{ 1, 3, 9, 11; 5, 7, 13, 15 \}$
- $\pi_2 = \{ 1, 5, 9, 13; 3, 7, 11, 15 \}$
- $\pi_3 = \{ 1, 7, 9, 15; 3, 5, 11, 13 \}$
- $\pi_4 = \{ 1, 7; 3, 5; 9, 15; 11, 13 \}$
- $\pi_5 = \{ 1, 9; 3, 11, 5, 13; 7, 15 \}$
- $\pi_6 = \{ 1, 15; 3, 13; 5, 11; 7, 9 \}$

By forming the higher level partition sums, it may be shown that these partitions, $\pi_1 - \pi_6$, are the only possible S.P. partitions for the F.S.M. model of the reduced modulo $2^4$ multiplier. The corresponding partition lattice is shown in Fig. 6.2.

There are a variety of ways with which this reduced multiplier may be implemented using the above partitions. For example, any two of the 4-block partitions $\pi_4$, $\pi_5$ and $\pi_6$ will lead to a parallel realisation. The case of using $\pi_4$ and $\pi_5$ is shown in Fig. 6.3(a),

* In our discussion, it is assumed that $A'$ or 'input', and $B'$ or 'state' of the reduced multiplier are assigned the same partition code.
in which 64 bits of memory are required. Alternatively, any one
of these 4-block partitions, \( \pi_5 \) say, along with a relevant non-S.P.
partition \( \tau_5 \), will realise a cascade configuration, as in Fig. 6.3(b),
requiring a 96-bit memory store.

Similarly, with the 2-block partitions \( \pi_1, \pi_2 \) and \( \pi_3 \), a cascade
form is possible if \( \pi_1 \) is used in conjunction with a non-S.P. partition
\( \tau_1 \) as shown in Fig. 6.3(c). Also, any two of them, \( \pi_2 \) and \( \pi_3 \) say,
may be operated in parallel to realise \( \pi_5 \) since \( \pi_2 \cdot \pi_3 = \pi_5 \). This
parallel form shown in Fig. 6.3(d), is then used with \( \pi_4 \), as in
Fig. 6.3(a) to realise the reduced multiplier which now require only
40 bits of memory storage.

An even better realisation has been found in which \( \pi_3 \) and \( \pi_2 \)
(or \( \pi_1 \)) in parallel realise \( \pi_5 \), and the same \( \pi_3 \) in cascade with a
non-S.P. \( \tau_3 \) implement \( \pi_4 \) (or \( \pi_6 \)). The resulting partitions \( \pi_5 \) and
\( \pi_4 \) are then operated in parallel. Unlike the scheme shown in Fig. 6.2(a),
these components now share a common variable between them. This hybrid
or composite configuration, shown in Fig. 6.4, requires a storage
of only 24 bits, which is a considerable reduction from the 192 bits
required in the direct realisation.

It may be shown, by deriving the logical functions of the components
represented by \( \pi_2, \pi_3 \) and \( \tau_3 \) that the binary assignment shown in
Table 6.2 is a good one. The blocks of \( \pi_5 \) and \( \pi_4 \) are encoded by the
variables \((y_2, y_1)\) and \((y_2, y_3)\) while those of \( \pi_3 \) and \( \pi_2 \) by the
variables \( y_2 \) and \( y_1 \) respectively as shown below.
Fig. 6.2. Partition lattice of F.S.M. model of modulo $2^4$ reduced multiplier.

Figs. 6.3(a) - (d). Possible decompositions of modulo $2^4$ reduced multiplier.
Fig. 6.4. Hybrid (composite) realisation of modulo $2^4$ reduced multiplier.
<table>
<thead>
<tr>
<th>Operands A, B</th>
<th>Binary number representation</th>
<th>Actual binary assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(x_3) (x_2) (x_1) (x_0)</td>
<td>(y_3) (y_2) (y_1) (y_0)</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 1</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1 1</td>
<td>0 1 1 1</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 1</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>7</td>
<td>0 1 1 1</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td>9</td>
<td>1 0 0 1</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>11</td>
<td>1 0 1 1</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>13</td>
<td>1 1 0 1</td>
<td>1 1 0 1</td>
</tr>
<tr>
<td>15</td>
<td>1 1 1 1</td>
<td>1 0 1 1</td>
</tr>
</tbody>
</table>

permanent '1's

Table 6.2. Binary codings for operands of reduced modulo 2^4 multiplier.
If the original operands \( A \) and \( B \) come from an external environment, then they are invariably coded in the binary number representation. Consequently, except for their L.S.B's, (which are kept at '1') the 'forced' operands \( A' \) and \( B' \) are also in the binary format. In this situation the partition variables \( y_1, y_2 \) and \( y_3 \) have to be encoded from the binary number code represented by the variables \( x_1, x_2 \) and \( x_3 \). Similarly, the output of the reduced multiplier has to be decoded back to the conventional binary format.

The logical relationship for this encoding and decoding process however is simple. From Table 6.2 we find that

\[
\begin{align*}
y_1 &= x_1 \\
y_2 &= x_2 \oplus x_1 \\
y_3 &= x_3
\end{align*}
\]

Similarly, for the decoding, we obtain

\[
\begin{align*}
x_1 &= y_1 \\
x_2 &= y_2 \oplus x_1 = y_2 \oplus y_1
\end{align*}
\]
and

\[ x_3 = y_3. \]

If the reduced multiplier is in an "internal" processor then the partition variables may be used directly.

The functional structures of the \( n_2 \), \( n_3 \) and \( T_3 \) components are obtained by first reorganising the multiplication table in the ways shown by Tables 6.3(a) and (b), and Table 6.5.

Consider first the \( n_2 \) and \( n_3 \) images of the F.S.M. model of the multiplier, whose state tables are shown in Tables 6.4(a) and (b) respectively. It is observed that in each of the two cases, the operation between the partition blocks is simply that of the Exclusive-OR function. Hence the \( n_2 \) and \( n_3 \) components are straightforward to implement.

With the table organised by \( T_3 \), we observe that if the operation between blocks is considered, this would have resembled that of the Ex-OR were it not for the entries shown in the dotted boxes. Assigning the variable \( y_3 \) to the blocks of \( T_3 \), we obtain,

\[ \tau_3 : (1,7,3,5) \rightarrow 0 \]
\[ (9,15,11,13) \rightarrow 1 \]

From this Table 6.5, it is seen that if the Ex-OR function is to be used for the operation between the blocks of \( T_3 \), the output of this successor component has to be modified by logically inverting it whenever both \( A' \) and \( B' \) come from the set \( (3,5,11,13) \). An Ex-OR output is assumed if either or both \( A' \) and \( B' \) are from the set \( (1,7,9,15) \). This information is easily obtained since these two blocks are also the blocks of \( n_3 \).
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>5</th>
<th>9</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>13</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>1</td>
<td>5</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>15</th>
<th>11</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>15</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>11</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>7</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>3</td>
<td>15</td>
<td>11</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>7</th>
<th>9</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>7</td>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>1</td>
<td>15</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>15</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>9</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>5</th>
<th>11</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>5</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>3</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>13</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>11</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

(b)

Table 6.3. Reduced modulo $2^4$ multiplication table reorganised by (a) $\pi_2$ and (b) $\pi_3$. 
Table 6.4. (a) \( \pi_2 \) and (b) \( \pi_3 \) images respectively of reduced multiplier.

\[
\begin{array}{cc|cc}
E & F & U & V \\
E & E & F & U \\
F & F & E & V \\
\end{array}
\]

(1, 5, 9, 13) \( \rightarrow \) E \hspace{1cm} (1, 7, 9, 15) \( \rightarrow \) U
(3, 15, 11, 7) \( \rightarrow \) F \hspace{1cm} (3, 5, 11, 13) \( \rightarrow \) V

(a) \hspace{2cm} (b)

Table 6.5. Multiplication table reorganised by \( \tau_3 \).

\[
\begin{array}{cccc|cccc}
& 1 & 7 & 3 & 5 & 9 & 15 & 11 & 13 \\
1 & 1 & 7 & 3 & 5 & 9 & 15 & 11 & 13 \\
7 & 7 & 1 & 5 & 3 & 15 & 9 & 13 & 11 \\
3 & 3 & 5 & 9 & 15 & 11 & 13 & 1 & 7 \\
5 & 5 & 3 & 15 & 9 & 13 & 11 & 7 & 1 \\
9 & 9 & 15 & 11 & 13 & 1 & 7 & 3 & 5 \\
15 & 15 & 9 & 13 & 11 & 7 & 1 & 5 & 3 \\
11 & 11 & 13 & 1 & 7 & 3 & 5 & 9 & 15 \\
13 & 13 & 11 & 7 & 1 & 5 & 3 & 15 & 9 \\
\end{array}
\]

since \( \pi_3 \cdot \tau_3 = \pi_4 \),

\( \tau_3 = \{1, 7, 3, 5; \ 9, 15, 11, 13\} \)
The detailed circuit structure of the component machines of the reduced modulo $2^4$ multiplier has now been derived and is shown in Fig. 6.5. Furthermore, using equation (6.0) the corresponding correction circuit may be obtained as shown in Fig. 6.6.

6.3 Internal algebraic structure of reduced modulo $2^N$ multipliers.

We have already seen that in the particular cases of modulo $2^3$ and modulo $2^4$ multipliers, the restriction of the multiplicative operands to only odd values led to the discovery of a variety of useful S.P. partitions on the input and state sets of their corresponding F.S.M. models. That these partitions could not have been predicted by Theorems 5.1 and 5.2 implies that algebraic structures other than the cascade loop-free form are possible.

In this section the algebraic structure of a general reduced modulo $2^N$ multiplier is investigated in depth in order to determine its general nature and pattern.

6.3.0 Example.

Consider the modulo $2^4$ multiplier discussed in Section 6.2.0, and in particular the reduced multiplication table reorganised by $\pi_2$, i.e. Table 6.3(a). There it was shown that the operation between the blocks of $\pi_2$ is analogous to the logical Exclusive-OR operation which, in turn, is simply the familiar modulo-2 addition. Consequently, the $\pi_2$-image of the reduced modulo $2^4$ multiplier is identical to a modulo-2 adder.

Table 6.3(a) also possesses another interesting feature. Consider first the multiplication operation between elements of the block
Fig. 6.5. Logic circuit implementation of modulo $2^4$ reduced multiplier.
Fig. 6.6. Implementation of correction circuit for modulo $2^4$ multiplier.
(1,5,9,13), which is described by the upper left-hand quadrant. The structure of this quadrant, which is redrawn in Table 6.6, becomes obvious if the following mapping is performed.

\[
\text{modulo } 2^4 \text{ multiplication } \rightarrow \text{modulo } 2^2 \text{ addition} \\
\text{confined to elements (1,5,9,13)}
\]

<table>
<thead>
<tr>
<th>1</th>
<th>(\rightarrow)</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>(\rightarrow)</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>(\rightarrow)</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>(\rightarrow)</td>
<td>3</td>
</tr>
</tbody>
</table>

By comparing Table 6.6 with the modulo 4 addition table shown in Table 6.7, it may be deduced that this particular quadrant of the reduced multiplier table is isomorphic to a modulo 4 adder. The upper right and lower left-hand quadrants may be analysed in a similar manner. The lower right-hand quadrant, although similar in structure, differs from Table 6.6 by two row shifts upwards. Hence, as shown in Fig. 6.7, the reduced modulo 2^4 multiplier can be regarded as a parallel connection of a modulo 2 adder and a modulo 4 adder incorporating the row shift mentioned.

A more elegant structure may be obtained if we consider the 4-block partition \(\pi_4\) and reorganise the multiplication table according to its block as shown in Table 6.8. The corresponding \(\pi_4\)-image, (see Table 6.9), is isomorphic to a modulo 4 adder via the mapping

\[
\begin{align*}
\times_{16} & \rightarrow \times_4 \\
P & \rightarrow 0 \\
Q & \rightarrow 1 \\
R & \rightarrow 2 \\
S & \rightarrow 3
\end{align*}
\]
### Table 6.6. Part of modulo 2\(^4\) multiplication table.

<table>
<thead>
<tr>
<th>(\times 16)</th>
<th>1</th>
<th>5</th>
<th>9</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>13</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>1</td>
<td>5</td>
<td>9</td>
</tr>
</tbody>
</table>

### Table 6.7. A modulo 4 addition table.

<table>
<thead>
<tr>
<th>(\times 4)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

### Table 6.8. Reduced multiplication table organised by \(\pi_4\).

<table>
<thead>
<tr>
<th>(\times 16)</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>9</th>
<th>15</th>
<th>11</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>7</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>15</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>15</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>5</td>
<td>[9, 15]</td>
<td>11</td>
<td>13</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>3</td>
<td>[15, 9]</td>
<td>13</td>
<td>11</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>15</td>
<td>11</td>
<td>13</td>
<td>1</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>9</td>
<td>13</td>
<td>11</td>
<td>7</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>13</td>
<td>1</td>
<td>7</td>
<td>3</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>11</td>
<td>7</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>15</td>
</tr>
</tbody>
</table>

### Table 6.9. \(\pi_4\)-image of reduced multiplier.

- \((1,7) \rightarrow P\)
- \((3,5) \rightarrow Q\)
- \((9,15) \rightarrow R\)
- \((11,13) \rightarrow S\)
Fig. 6.7. Pseudo-parallel decomposition of modulo $2^4$ reduced multiplier.

Fig. 6.8. Implementing reduced multiplier (a) as a parallel connection of modulo 2 and modulo 4 adders, (b) with the modulo 4 adder further decomposed.
Also, the entries in Table 6.8 are grouped into blocks of four elements, a typical block being shown enclosed by the broken lines.

It is observed that the structure of every such block is identical to that of a modulo 2 adder. Hence the resulting implementation, shown in Fig. 6.8(a), consists of a modulo 2 and a modulo 4 adders operating in parallel, with neither component requiring any correction.

In addition, this modulo 4 adder may be further decomposed using the loop-free technique described by Theorem 5.1. The final realisation of the reduced modulo $2^4$ multiplier shown in Fig. 6.8(b) is thus obtained which requires a memory store of 24 bits.

Although this approach led to an overall circuit configuration and memory requirement identical to those shown in Figs. 6.4 and 6.5, it will now be shown to be more systematic and directly applicable to the general modulo $2^N$ reduced multipliers than the approach used in Section 6.2.0.

6.3.1 The group under modulo $2^N$ reduced multiplication.

Consider the general modulo $2^N$ multiplier whose two operands (and hence product) are constrained to odd values, i.e. to values from the set $Z_D$, and denote this multiplier as the tuple $(Z_D, \otimes_2^N)$. We will now prove the following lemmas.

**Lemma 6.0.** $(Z_D, \otimes_2^N)$ is closed.

**Proof.** Consider $a, b \in Z_D$. Using the familiar division algorithm, the real product $a \times b$ may be expressed as

$$ a \times b = q2^N + r, \text{ where } 0 \leq r < 2^N. $$
Since $a, b$ are odd and hence not divisible by $2$, so is their product. Thus the right-hand side of the above equation is not divisible by $2$. Hence $r$, the modulo $2^N$ product of $a$ and $b$ must also be odd and in the range $0$ to $2^N - 1$, i.e. $r \in \mathbb{Z}_D$.

**Lemma 6.1.** \((\mathbb{Z}_D, \otimes)\) is associative, commutative and has identity $1$.

**Proof.** This follows naturally from the properties of the original modulo $2^N$ multiplier.

**Lemma 6.2.** Every element $d \in \mathbb{Z}_D$ has a multiplicative inverse w.r.t. modulo $2^N$ multiplication, i.e. $x \in \mathbb{Z}_D$ may be found such that $dx \equiv 1 \pmod{2^N}$.

**Proof.** Consider $d \in \mathbb{Z}_D$. From Lemma 6.0, $d \times d$ is also in $\mathbb{Z}_D$ and similarly for $d \times dx \ldots \times d = d^k$, $k$ an integer. Since $\mathbb{Z}_D$ is a finite set, then for a particular $k$, say $k'$, there must be a repetition, i.e.

$$d^{k'} = q2^N + d$$

i.e.

$$d(d^{k'-1} - 1) = q2^N.$$  

Since $d$ is coprime to $2$ and hence $2^N$, the above equation implies that $(d^{k'-1} - 1)$ is divisible by $2^N$. Therefore we can write

$$d^{k'-1} \equiv 1 \pmod{2^N}$$

or

$$d \times d^{k'-2} \equiv 1 \pmod{2^N},$$

i.e. for any $d \in \mathbb{Z}_D$, we can find its inverse, which is $d^{k'-2}$. 
Lemma 6.3. \( \mathbb{Z}_D \) contains \( 2^{N-1} \) elements.

Proof. All the even elements from the original \( \mathbb{Z}_N \) are of the form

\[
0, 2 \times 1, 2 \times 2, \ldots, 2 \times \epsilon, \ldots, 2 \times (2^{N-1} - 1), \quad \epsilon = 0, 1, \ldots, 2^{N-1} - 1.
\]

\( \therefore \) There are \((2^{N-1} - 1) + 1 = 2^{N-1}\) even elements. Consequently, the number of odd elements which we denote by \( |\mathbb{Z}_D| \) is given by

\[
|\mathbb{Z}_D| = 2^N - 2^{N-1} = 2^{N-1}.
\]

The above four lemmas lead naturally to the following theorem with which the overall structure of the general reduced modulo \( 2^N \) multiplier may be described.

**Theorem 6.0.** The set of odd integers, i.e. \( \mathbb{Z}_D = \{ x : x \) odd integer, \( 1 \leq x \leq 2^{N-1}\} \), forms an Abelian group \^ under multiplication modulo \( 2^N \). This group, which we denote by \( \mathbb{G}(2^N) \), has order \( |\mathbb{G}(2^N)| = 2^{N-1} \).

### 6.3.2 Derivation of detailed algebraic structure of reduced modulo \( 2^N \) multipliers.

Before we present our main results, we state below some well known results and concepts in algebraic number theory that we will be using.

**Definition 6.0.** Let \( x \in \mathbb{Z}_M \) where \( \mathbb{Z}_M = \{ x : x \) integer, \( 0 \leq x < m \} \) and suppose \( x^{\varphi(m)} \equiv 1 \) (modulo \( m \)), where \( \varphi(m) \) is the number of elements coprime to \( m \). Also let \( x^n \not\equiv 1 \) (modulo \( m \)) for

* References 68-70 are excellent introductions to the concepts and terminology of elementary group theory.
n ≤ θ(m). Then x is called a primitive root of 1 modulo m or simply a primitive root of m.

Theorem 6.1. If \( Z_m \) has a primitive root of 1 modulo m

if

a) \( m = p^N \),

b) \( m = 2p^N \) or

c) \( m = 1, 2, 4 \), i.e. \( m = 2^0, 2^1, 2^2 \).

where \( p \) is any odd prime.

Proof. See Theorem 2.25 of Reference 47.

We see that (c) can be used directly to describe the structure of our reduced multipliers for moduli \( 2^0, 2^1 \) and \( 2^2 \), by determining the relevant primitive root and using the following Theorem 6.2.

Theorem 6.2. If \( Z_N \) has a primitive root of 1 modulo \( p^N \),

then \( G(p^N) \) is a cyclic group where \( Z_N = \{ x : x \text{ integer, } 0 < x < p^N \} \), and \( G(p^N) \) is the group, consisting of the set of elements of \( Z_N \) that are coprime to \( p^N \), and the modulo \( p^N \) multiplication.

Proof. Suppose \( Z_N \) has x as a primitive root of \( p^N \).

Then x must be coprime to \( p^N \) and hence \( x \in G(p^N) \). Let \( H \) be the subgroup of \( G(p^N) \) generated by x. Then \( |H| = O(x) = \theta(p^N) \), i.e. \( |H| = |G(p^N)| \), where \( |H| \) and \( O(x) \) are the orders of \( H \) and the element x respectively. Since \( H \leq G(p^N) \) we must have \( H = G(p^N) \), i.e. \( G(p^N) \) is cyclically generated by x.

Theorem 6.3. If G is an Abelian group of order \( p^N \) for some prime \( p \) and natural number q, then

\[
G = H_1 \times H_2 \times \ldots \times H_i \times \ldots \times H_q
\]
where each $H_i$ is cyclic of order $p_i^k$ and $\sum k_i = N$.

Proof. See Theorem 5.1.11 of Ref. 48.

From Theorems 6.1 and 6.2 we now know that for each of the moduli $2^0$, $2^1$ and $2^2$, the complete multiplication table of the corresponding reduced multiplier can be generated by a single element $x \in \mathbb{Z}_N$, if $x$ is a primitive root of $2^N$, $N = 0, 1, 2$.

In our following two lemmas and one theorem we extend the analysis to cases where $N \geq 3$ and will show that the table of a general reduced modulo $2^N$ multiplier, $N \geq 3$, can still be described completely but this time two elements $Z_i, Z_j \in \mathbb{Z}_N$ are required to generate it.

**Lemma 6.4.** $3^{2^n} = (2^{n+2})(x(n)) + 1$ for $n \geq 1$ where $x(n)$ is an odd number for all $n$.

Proof. $3^1 = 9 = 8 + 1 = 2^3 + 1 = (2^{1+2}) \times 1 + 1$.

So the expression is true for $n = 1$, where $x(1) = 1$. We now assume that it is true for $n = k$, i.e.

$$3^{2^k} = 2^{k+2} x(k) + 1.$$ 

Writing the expression $3^{2^{k+1}}$ we get

$$3^{2^{k+1}} = 3^{2^k} \cdot 2 = (3^{2^k})^2.$$ 

Substituting for $3^{2^k}$, we have

$$3^{2^{k+1}} = \left[2^{k+2} x(k) + 1 \right]^2 = (2^{k+2} x(k))^2 + 2(2^{k+2} x(k)) + 1$$

$$= 2^{2k+4} (x(k))^2 + 2^{k+3} x(k) + 1$$

$$= 2^{k+3} \left[2^{k+1} (x(k))^2 + x(k) \right] + 1$$
where we have let \[2^{k+1} (x(k))^2 + x(k)\] be equal to \(x(k+1)\), which is odd since \(x(k)\) is odd and \(2^{k+1} (x(k))^2\) is even. Therefore the Lemma is proved by induction.

(We note that \(x(1) = 1\) and that \(x(n)\) is defined recursively using the expression:

\[x(n) = 2^n \left\{ \left(x(n-1)\right)^2 + x(n-1) \right\}.\]

**Corollary.** The element 3 in \(\mathbb{Z}_{2^N}\) has order \(2^{N-2}\) in \(G(2^N)\).

**Proof.** Since 3 is in \(G(2^N)\), and the order of \(G(2^N)\) is \(2^{N-1}\) (see Lemma 6.3), then 3 has order \(2^k\) in \(G(2^N)\) such that \(0 < k \leq N-1\). Thus we may write

\[3^{2^k} \equiv 1 \pmod{2^N}.\]

Using Lemma 6.4, we can substitute for \(3^{2^k}\) thus obtaining,

\[2^{k+2} x(k) + 1 \equiv 1 \pmod{2^N}\]

i.e. \[2^{k+2} x(k) \equiv 0 \pmod{2^N}\]

implying that \(2^{k+2}\) is divisible by \(2^N\) since \(x(k)\) is odd. Thus we obtain

\[2^{k+2} = q2^N, \quad q = 0, 1, \ldots\]

The first (or least non-zero) value of \(k\) to satisfy the above equation is when \(q = 1\), resulting in \(k+2 = N\), and hence \(k = N-2\). Therefore 3 has order \(2^{N-2}\) in \(G(2^N)\).

**Lemma 6.5.** There are four elements, \(\pm 1\) and \(2^{N-1} \pm 1\), in \(G(2^N)\) having order 2.
Proof. Let \( x \in G(2^N) \) have order 2, i.e.
\[
x^2 \equiv 1 \pmod{2^N}
\]
or
\[
x^2 - 1 \equiv 0 \pmod{2^N}.
\]
Since \( x \) is odd, we can write
\[
x = 2q + 1.
\]
Therefore
\[
(2q+1)^2 - 1 \equiv 0 \pmod{2^N}
\]
i.e.
\[
4q^2 + 4q + 1 - 1 \equiv 0 \pmod{2^N}
\]
\[
2^2 q(q+1) \equiv 0 \pmod{2^N}
\]
or
\[
q(q+1) \equiv 0 \pmod{2^{N-2}}.
\]
In this congruence, we see that if \( q \) is even then \( q+1 \) is odd and vice versa.

Case (i). Let \( q \) be even. This implies that \( q \) is divisible by \( 2^{N-2} \), i.e.
\[
q = 0, \ 1 \times 2^{N-2}, \ 2 \times 2^{N-2}, \ldots, \ \ell \times 2^{N-2}, \ldots
\]
where \( \ell \) is an integer.

\[
\therefore \quad x \text{ can be written as}
\]
\[
x = 2 \left\lfloor \frac{\ell 2^{N-2}}{2} \right\rfloor + 1
\]
For \( \ell \) even, i.e. \( = 2u \), say, we obtain
\[
x = 2 \left\lfloor 2u \cdot 2^{N-2} \right\rfloor + 1
\]
\[
= 2 \left\lfloor u \cdot 2^{N-1} \right\rfloor + 1
\]
\[
= u \cdot 2^N + 1.
\]
Therefore
\[ x \equiv 1 \pmod{2^N} \] \hspace{1cm} \text{...(6.0)}

If \( \ell \) is odd, i.e. \( = 2v + 1 \) say, then:
\[
x = 2 \left[ (2v+1)2^{N-2} \right] + 1
\]
\[
= 2 \left[ v \cdot 2^{N-1} + 2^{N-2} \right] + 1
\]
\[
= v \cdot 2^N + (2^{N-1} + 1)
\]

Therefore
\[ x \equiv 2^{N-1} + 1 \pmod{2^N} \] \hspace{1cm} \text{...(6.1)}

Case (ii) Let \( q \) be odd, thus implying that \((q+1)\) is divisible by \(2^{N-2}\), i.e.
\[ (q+1) = \ell 2^{N-2} \hspace{1cm} \ell = 0,1, \ldots \]
or
\[ q = \ell 2^{N-2} - 1. \]
\[ \therefore \]
\[ x = 2 \left[ \ell 2^{N-2} - 1 \right] + 1 \]
\[ = \ell 2^{N-1} - 1. \]

For \( \ell \) even, i.e. \( \ell = 2u \), we obtain
\[ x = 2u \cdot 2^{N-2} - 1 = u \cdot 2^N - 1. \]
\[ \therefore \]
\[ x \equiv -1 \pmod{2^N} \] \hspace{1cm} \text{...(6.2)}

For \( \ell \) odd, i.e. \( = 2v + 1 \), then
\[ x = (2v+1)2^{N-1} - 1 \]
\[ = v \cdot 2^N + (2^{N-1} - 1). \]
\[ \therefore \]
\[ x \equiv 2^{N-1} - 1 \pmod{2^N} \] \hspace{1cm} \text{...(6.3)}
Consequently, from equations (6.0) - (6.3), we obtain the result that if $x$ is to have order 2 then

$$x \equiv 1, \ 2^{N-1} + 1, -1 \text{ and } 2^{N-1} - 1 \pmod{2^N}.$$  

**Corollary.**  
(a) The values $x_h \equiv 1$ and $x_i \equiv 2^{N-1} + 1$ (modulo $2^N$) may be expressed as powers of 3 (modulo $2^N$).

(b) The values $x_j \equiv -1$ and $x_k \equiv 2^{N-1} - 1$ (modulo $2^N$) cannot be powers of 3.

**Proof.** Using Lemma 6.4 and letting $n = N-2$ and $N-3$ respectively, we obtain the expressions

$$3^{2^{N-2}} = 2^N x(N-2) + 1 \quad \ldots \ (6.4)$$

and

$$3^{2^{N-3}} = 2^{N-1} x(N-3) + 1 \quad \ldots \ (6.5)$$

The values in (a) above may be written as

$$x_h = 2^N \times q_h + 1 \quad \ldots \ (6.6)$$

and

$$x_i = 2^{N-1} q_i + 1 \quad \ldots \ (6.7)$$

where $q_h$ and $q_i$ are integers. We may now, by comparing the coefficients of the terms $2^N$ in equations (6.4) and (6.6), and the terms $2^{N-1}$ in equations (6.5) and (6.7), deduce that if we let $q_h = x(N-2)$ and $q_i = x(N-3)$ then

$$x_h = 3^{2^{N-2}}$$

and

$$x_i = 3^{2^{N-3}}$$

thus proving (a).
The values \( x_j \) and \( x_k \) may be written as

\[
x_j = \left[ 2^N x^{(N-2)} + 1 \right] - 2 = x_h - 2 \quad \ldots (6.8)
\]

and

\[
x_k = \left[ 2^{N-1} x^{(N-3)} + 1 \right] - 2 = x_i - 2 \quad \ldots (6.9)
\]

In the above equations, we know that \( x_h \) and \( x_i \), being powers of 3, must be divisible by 3. However 2 is not. Therefore \( x_j \) and \( x_k \) are not divisible by 3, and since this is a necessary condition for them to be powers of 3, then we have proved (b).

Consider now the set \( K \) given by

\[
K = \left\{ k_0', k_1', \ldots, k_i', \ldots, k_{2^{N-2}-1}' \right\}, \quad 0 \leq i < 2^{N-2}
\]

where \( k_i \in G(2^N) \) and \( k_i \equiv 3^i \mod 2^N \). For any pair \( k_i, k_j \in K \), we therefore have

\[
k_i \times k_j \equiv 3^i \times 3^j \mod 2^N
\]

\[
\equiv 3^{(i+j)}
\]

\((i+j)\) may be written as \( q2^{N-2} + r \), where \( q = 0 \) or 1 and \( 0 \leq r < 2^{N-2} \).

\[
\therefore k_i \times k_j \equiv 3^{q2^{N-2}} 3^r \mod 2^N
\]

\[
\equiv 3^r \mod 2^N \quad \text{if} \quad q = 0.
\]

If \( q = 1 \), we use the Corollary to Lemma 6.4. to obtain

\[
k_i \times k_j \equiv (Q2^N + 1)3^r \mod 2^N, \quad Q \text{ an integer}
\]

\[
\equiv 3^r \mod 2^N.
\]

Since \( 0 \leq r < 2^{N-2} \), \( k_r = (k_i \times k_j) \in K \).
Therefore K is closed and hence is a subgroup of $G(2^N)$.

Consider now the set $L$, given by

$$L = (l_0, l_1) \text{ where } l_0, l_1 \in G(2^N)$$

and $l_i \equiv x^i \mod 2^N$, for $i = 0, 1$.

The value of $x$ is either $x \equiv -1 \mod 2^N$ or $x = 2^{N-1} - 1$ (See Corollary (b) of Lemma 6.6).

To show that $L$ is closed and hence $L \subseteq G(2^N)$, it is sufficient to demonstrate that $x \times x = x^2 \equiv 1 \mod 2^N$ according to Lemma 6.5. Therefore $x^2 \in L$.

We are now in a position to present a detailed description of the algebraic structure of our reduced modulo $2^N$ multiplier, $N \geq 3$, via the following Theorem.

**Theorem 6.4.** The group $G(2^N)$, as described by Theorem 6.0, for $N \geq 3$, is isomorphic to the direct product group $K \times H$, where $K$ and $H$ are cyclic groups of orders $2^{N-2}$ and 2 respectively.

**Proof.** $G(2^N)$, $N \geq 3$, cannot be cyclic since if it were, it would have a primitive $2^N$ root and this contradicts Theorem 6.1. So by Theorem 6.3, $G(2^N) = H_1 \times H_2 \times \ldots \times H_q$, $q \geq 2$. Let $K$ be the subgroup generated by 3. Then $|K| = 2^{N-2}$. So $K$ is $H_1$ say, since $K$ is cyclic (being generated by a single element), and $G(2^N)$ is not. By order considerations then $G(2^N) = H_1 \times H_2$, where $H_2$ is cyclic of order 2. This gives our result.

Since $L$ is not a subset of $K$, we also have $H_2 = L$. 
6.3.2 Application of theoretical results.

The subgroups $K$ and $L$ may now be used to organise $G(2^N)$ by forming the relevant cosets in the usual way.

Let the cosets w.r.t. $K$ be $V_0$ and $V_1$ and those w.r.t. $L$ be $W_0, W_1, \ldots, W_i, \ldots, W_n$, $n = (2^{N-2} - 1)$, given by

$$V_0 = (v_{o,0}, v_{o,1}, v_{o,2}, \ldots, v_{o,i}, \ldots, v_{o,n}) = K$$
$$V_1 = (v_{1,0}, v_{1,1}, \ldots, v_{1,i}, \ldots, v_{1,n})$$

and

$$W_0 = (w_{o,0}, w_{o,1}) = L$$
$$W_1 = (w_{1,0}, w_{1,1})$$
$$\vdots$$
$$W_i = (w_{i,0}, w_{i,1})$$

where $v_{d,i} \equiv x^d \cdot 3^i$ modulo $2^N$, $d = 0 \text{ or } 1$, $0 \leq i \leq n$

and $w_{i,d} \equiv 3^i \cdot x^d$, in which the value of $x$ is either $x_j$ or $x_k$ as described by Corollary (b) of Lemma 6.5.

For example, if $N = 4$, then the elements of $G(2^4)$ are $1, 3, 5, 7, 9, 11, 13, 15$, and hence $K = (3^0, 3^1, 3^2, 3^3) = (1, 3, 9, 11)$ modulo $2^4$, and $L$ is either $(1, 7)$ or $(1, 15)$. Consequently, the relevant cosets are

$$V_0 = K, \text{ and } V_1 = (7, 3^0, 7, 3^1, 7, 3^2, 7, 3^3), \text{ for } x = 7$$
$$= (7, 5, 15, 13),$$

* Since the multiplication operation is commutative the elements of $K$ (or $L$) can be multiplied by any particular element of $G(2^N)$ either from the left or from the right.
and
\[ W_0 = L, \quad W_1 = (3^1.1, 3^1.7) = (3,5), \]
\[ W_2 = (3^2.1, 3^2.7) = (9,15), \quad \text{and} \quad W_3 = (3^3.1, 3^3.7) = (11,13). \]

In general, each element \( g_{\ell,m} \) of \( G(2^N) \) can be represented by the modulo \( 2^N \) product of powers of \( x \) and \( 3 \) via the following congruence, i.e.
\[ x^\ell \cdot 3^m \equiv g_{\ell,m} \quad \text{modulo} \quad 2^N, \quad 1 \leq g_{\ell,m} < 2^{N-1}, \]
where \( \ell = 0 \) or \( 1 \) and \( 0 \leq m < n. \)

From the corollary to Lemma 6.4, and the results in Lemma 6.6, the components \( 3^m \) and \( x^\ell \), will go through \( 2^{N-2} \) and \( 2 \) values respectively before repeating themselves. Therefore this will generate \( (2^{N-2}) \times 2 = 2^{N-1} \) different values of \( x^\ell \cdot 3^m \) (modulo \( 2^N \)). Since there are also \( 2^{N-1} \) different elements of \( G(2^N) \), the above congruence describes \( g_{\ell,m} \) uniquely.

Let us now express \( G(2^N) \) in terms of its cosets, i.e.
\[ G(2^N) = \{ V_0, V_1 \} \quad \text{using subgroup} \quad K, \]
and
\[ G(2^N) = \{ W_0, W_1, \ldots, W_i, \ldots, W_n \} \quad \text{using the subgroup} \quad L. \]

If we consider any two elements of \( G(2^N) \), say \( g_{\ell',m'} \) and \( g_{\ell'',m''} \), then, as shown in Appendix 6.0, their product \( P \) is given by
\[ P \equiv g_{i,j} \quad \text{modulo} \quad 2^N, \]
where
\[ (\ell' + \ell'') \equiv i \quad \text{modulo} \quad 2 \]
and

\[(m' + m'') = j\mod 2^{N-2} \]

Since the subscripts \(l\) and \(m\) denote the cosets w.r.t. the subgroups \(K\) and \(L\) respectively, then we know that

\[g_{l',m'} \in V_{l'}, \quad \text{and also} \quad E \in W_{m'}, \]

and

\[g_{l'',m''} \in V_{l''}, \quad \text{and also} \quad E \in W_{m''}.\]

Furthermore, their modulo \(2^N\) product \(g_{i,j}\) belongs to both cosets \(V_i\) and \(W_j\), where \(i\) and \(j\) are the modulo 2, and modulo \(2^{N-2}\) sums of \(l', l''\), and \(m', m''\) respectively.

Consequently, if we denote the operation between any two cosets \(w.r.t. K\) by \(\Box_{\underline{}}\) and that between any two \(w.r.t. L\) by \(\Box_{\underline{}}\), then it is not difficult to see that

\[V_{l'} \Box_{\underline{}} V_{l''} = V(l' + l'') \mod 2 \quad \ldots (6.10)\]

and

\[W_{m'} \Box_{\underline{}} W_{m''} = W(m' + m'') \mod 2^{N-2} \quad \ldots (6.11)\]

In other words, if each coset is mapped onto its corresponding index, i.e.

\[V_{l'} \rightarrow l', \quad V_{l''} \rightarrow l'' \]

and

\[W_{m'} \rightarrow m', \quad W_{m''} \rightarrow m''.\]

* i.e. the modulo \(2^N\) multiplication of any member of any one coset with any member of the same or any other coset.
then operations between cosets may be mapped onto modulo addition operations between indices as shown by the two commutative diagrams below.

(a) 
\[ V_{k}, V_{k} \xrightarrow{\square_{k}} V_{i} \]
\[ i = (k' + k'') \pmod{2} \]

(b) 
\[ W_{m}, W_{m} \xrightarrow{\square_{m}} W_{j} \]
\[ j = (m' + m'') \pmod{2^{N-2}} \]

Finally, the complete reduced modulo \(2^N\) multiplication may be described in a compact way as follows.

Let \( f \) and \( f' \) be the mappings given by

\[ f : g_{\lambda, m} \longrightarrow (\lambda, m) \]

and

\[ f' : \bigoplus \pmod{2^N} \longrightarrow (\bigoplus \pmod{2}, \bigoplus \pmod{2^{N-2}}) \]

where

\[ g_{\lambda, m} \in G(2^N), \quad \lambda \in \{0,1\}, \quad m \in \{0,1,2,\ldots, 2^{N-2} - 1\} \]
and □ is the parallel component-wise operation between any two ordered-pairs \((\ell', m')\) and \((\ell'', m'')\), i.e.

\[
(\ell', m') □ (\ell'', m'') = \left[ \begin{bmatrix} \ell' \oplus_2 \ell'' \end{bmatrix}, \begin{bmatrix} m' \oplus_{2^{N-2}} m'' \end{bmatrix} \right] = (i, j).
\]

Thus, for any two elements of \(G(2^N)\), say \(g_{\ell', m'}\) and \(g_{\ell'', m''}\), we have the following useful commutative diagram.

\[
\begin{array}{ccc}
g_{\ell', m'} & \otimes_{2^N} & (g_{\ell', m'})(g_{\ell'', m''}) \mod 2^N \\
f_g & & f_g \\
(\ell', m') & \otimes & (\ell'', m'') \\
\Box & & \Box \\
(\ell', m') □ (\ell'', m'') = \left[ \begin{bmatrix} \ell' \oplus_2 \ell'' \end{bmatrix}, \begin{bmatrix} m' \oplus_{2^{N-2}} m'' \end{bmatrix} \right] = (i, j)
\end{array}
\]

It is now easily seen that the mapping-pair \(f_g, f_p\) transforms the original reduced multiplier into two adders, modulo 2 and modulo \(2^{N-2}\) respectively, operating in parallel.

To illustrate this isomorphism between the multiplier and the adder-pair, consider again the case when \(N = 4\). We have already seen on page 132 that two possible organisations of \(G(2^4)\) into sets of cosets are,

\[
(a) \quad G(2^4) = \{(1,3,9,11) ; (7,5,15,13)\}
\]

and

\[
(b) \quad G(2^4) = \{(1,7); (3,5); (9,15); (11,13)\}.
\]
Using these cosets, the modulo $2^4$ multiplication table may be 'rearranged' as shown in Tables 6.10 and 6.12, and the corresponding operations between cosets are shown in Tables 6.11 and 6.13. The tables illustrated there are easily seen to be identical to the addition tables modulo 2 and modulo $(2^4 - 2 = 4)$ respectively.

Consider multiplying, modulo 16, the number 9 by 11. Using the mappings shown in Tables 6.11 and 6.13, and the commutative diagram (c), we may substitute additions modulo 2 and modulo 4 for our original modulo 16 multiplication as shown below.

\[
\begin{array}{ccc}
9 & \times_{16} & 11 \\
\downarrow & f & \downarrow \\
(0,2) & \oplus & (0,3) \\
\downarrow & \square & \downarrow \\
(0,2) \square (0,3) = \left[\begin{array}{ccc}
0 & 0 & 2 \\
0 & 0 & 3
\end{array}\right] = (0,1)
\end{array}
\]

Instead of 3, we can also use 5 to generate the subgroup of order 4, and 15 may be chosen for the subgroup of order 2 thus obtaining $K = (1,5,9,13)$ and $L = (1,15)$ respectively. The corresponding tables reorganised by these subgroups are shown in Tables 6.14 and 6.15 respectively. The sets of cosets are now $\{(1,5,9,13); (15,11,7,3)\}$ and $\{(1,15); (5,11); (9,7); (13,3)\}$ and the tables for the operations between these cosets can be derived in the way discussed previously.

Other possible pairs of $K$ and $L$ are $\{(1,3,9,11); (15,13,7,5)\}$, $\{(1,15); (3,13); (9,7); (11,5)\}$ and $\{(1,5,9,13); (7,3,15,11)\}$, $\{(1,7); (5,3); (9,15); (13,11)\}$. 
Table 6.10. Reduced multiplication table organised by subgroup 
(1,3,9,11).

<table>
<thead>
<tr>
<th>A'</th>
<th>16</th>
<th>1</th>
<th>3</th>
<th>9</th>
<th>11</th>
<th>7</th>
<th>5</th>
<th>15</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>9</td>
<td>11</td>
<td>7</td>
<td>5</td>
<td>15</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>9</td>
<td>11</td>
<td>1</td>
<td>5</td>
<td>15</td>
<td>13</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>11</td>
<td>1</td>
<td>3</td>
<td>15</td>
<td>13</td>
<td>7</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>B'</td>
<td>11</td>
<td>11</td>
<td>1</td>
<td>3</td>
<td>9</td>
<td>13</td>
<td>7</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>5</td>
<td>15</td>
<td>13</td>
<td>1</td>
<td>3</td>
<td>9</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>15</td>
<td>13</td>
<td>7</td>
<td>3</td>
<td>9</td>
<td>11</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>13</td>
<td>7</td>
<td>5</td>
<td>9</td>
<td>11</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>7</td>
<td>5</td>
<td>15</td>
<td>11</td>
<td>1</td>
<td>3</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.11. Operation between blocks.

<table>
<thead>
<tr>
<th>B'</th>
<th>2</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

(1,3,9,11) \rightarrow 0
(7,5,15,13) \rightarrow 1
Table 6.12. Reduced multiplier table organised by subgroup (1,7).

<table>
<thead>
<tr>
<th>16</th>
<th>1 7</th>
<th>3 5</th>
<th>9 15</th>
<th>11 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 7</td>
<td>3 5</td>
<td>9 15</td>
<td>11 13</td>
</tr>
<tr>
<td>7</td>
<td>7 1</td>
<td>5 3</td>
<td>15 9</td>
<td>13 11</td>
</tr>
<tr>
<td>3</td>
<td>3 5</td>
<td>9 15</td>
<td>11 13</td>
<td>1 7</td>
</tr>
<tr>
<td>5</td>
<td>5 3</td>
<td>15 9</td>
<td>13 11</td>
<td>7 1</td>
</tr>
<tr>
<td>9</td>
<td>9 15</td>
<td>11 13</td>
<td>1 7</td>
<td>3 5</td>
</tr>
<tr>
<td>15</td>
<td>15 9</td>
<td>13 11</td>
<td>7 1</td>
<td>5 3</td>
</tr>
<tr>
<td>11</td>
<td>11 13</td>
<td>1 7</td>
<td>3 5</td>
<td>9 15</td>
</tr>
<tr>
<td>13</td>
<td>13 11</td>
<td>7 1</td>
<td>5 3</td>
<td>15 9</td>
</tr>
</tbody>
</table>

Table 6.13. Operation between blocks.

<table>
<thead>
<tr>
<th>4</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>( \times )</td>
<td>( 2 )</td>
<td>( 16 )</td>
<td>( 1 )</td>
<td>( 5 )</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 5 )</td>
<td>( 9 )</td>
<td>( 13 )</td>
</tr>
<tr>
<td>( 5 )</td>
<td>( 5 )</td>
<td>( 9 )</td>
<td>( 13 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 9 )</td>
<td>( 9 )</td>
<td>( 13 )</td>
<td>( 1 )</td>
<td>( 5 )</td>
</tr>
<tr>
<td>( 13 )</td>
<td>( 13 )</td>
<td>( 1 )</td>
<td>( 5 )</td>
<td>( 9 )</td>
</tr>
<tr>
<td>( \otimes )</td>
<td>( \otimes )</td>
<td>( \otimes )</td>
<td>( \otimes )</td>
<td>( \otimes )</td>
</tr>
<tr>
<td>( \otimes )</td>
<td>( \otimes )</td>
<td>( \otimes )</td>
<td>( \otimes )</td>
<td>( \otimes )</td>
</tr>
<tr>
<td>( \otimes )</td>
<td>( \otimes )</td>
<td>( \otimes )</td>
<td>( \otimes )</td>
<td>( \otimes )</td>
</tr>
<tr>
<td>( \otimes )</td>
<td>( \otimes )</td>
<td>( \otimes )</td>
<td>( \otimes )</td>
<td>( \otimes )</td>
</tr>
</tbody>
</table>

Table 6.14. Modulo 2⁴ reduced multiplication table organised by subgroup (1,5,9,13).

Table 6.15. Multiplication table organised by subgroup (1,15).
If the reduced modulo $2^N$ multiplier is modelled as a finite-state machine, we recall that the sets of values that the 'forced' operands $A'$ and $B'$ can take are regarded as the 'input' set and the 'internal state' set respectively, and are both equal to $\mathbb{Z}_D$. In such a case, the sets of cosets w.r.t. the subgroups $K$ and $L$ are equivalent to S.P. partitions on the input and state sets of the F.S.M. model respectively. If these partitions are denoted by $\pi_K$ and $\pi_L$, we also observe that as a direct consequence of the isomorphism between $G(2^N)$ and $K \times L$, we have $\pi_K \cdot \pi_L = \pi'(0) = \mathbb{Z}_D$.

Furthermore, the tables in which the 'inputs' and 'states' are the cosets w.r.t. to $K$ and $L$ can now be looked upon as the homomorphic $\pi_K$-image and $\pi_L$-image, respectively, of the F.S.M. model of the reduced multiplier.

In practice, the results that we have derived are easily applied to the implementation of the general modulo $2^N$ reduced multiplier. The subgroup $K$ is first generated by simply forming, modulo $2^N$, the successive powers, up to the $(2^{N-2} - 1)$th, of 3 or 5, e.g.

$$K = \left\{3^0 = 1, 3^1, 3^2, \ldots 3^{2^{N-2} - 1}, 3^{2^{N-2}} = 1\right\},$$

either manually or by means of a straightforward computer program for large values of $N$. The corresponding $\pi_K$-image, being isomorphic to a modulo $2^{N-2}$ adder, may now be structurally decomposed using the loop-free technique for adders as in Section 5.2.1.1. The generation of $L$ is trivial.

* Unlike the direct loop-free structure of the multiplier (See Chap. 5.), the loop-free configuration of a general modulo $2^N$ adder is composed of sub-machines or components whose algebraic structures are regular, and are easily described and generalised.
For example, if \( N = 5 \), i.e. \( 2^N = 32 \), then the two possible forms of \( K \) are,

\[
K(3) = (1, 3^1, 3^2 = 9, 3^3 = 27, 3^4 = 17, 3^5 = 19, 3^6 = 25, 3^7 = 11, 3^8 = 1)
\]

and

\[
K(5) = (1, 5^1, 5^2 = 25, 5^3 = 29, 5^4 = 17, 5^5 = 21, 5^6 = 9, 5^7 = 13, 5^8 = 1).
\]

Similarly, for the subgroup \( L \), we have

\[
L(15) = (1, 15) \quad \text{and} \quad L(31) = (1, 31).
\]

We thus have the S.P. partitions,

\[
\pi_{K(3)} = \{1, 3, 9, 27, 17, 19, 25, 11; 5, 15, 13, 7, 21, 31, 29, 23\}
\]

\[
\pi_{K(5)} = \{1, 5, 25, 29, 17, 21, 9, 13; 3, 15, 11, 23, 19, 31, 27, 7\}
\]

\[
\pi_{L(15)} = \{1, 15; 3, 13; 9, 7; 27, 21; 17, 31; 19, 29; 25, 23; 11, 5\}
\]

\[
\pi_{L(31)} = \{1, 31; 3, 29; 5, 27; 7, 25; 9, 23; 11, 21; 13, 19; 15, 17\}.
\]

6.4 General comparison with the direct implementation of modulo \( 2^N \) multipliers.

In this Chapter we have been mainly occupied in the theoretical derivation of an algebraic structure for the general modulo \( 2^N \) multiplier which is found to be an interesting alternative to the loop-free configuration described in Chapter 5. As such we have not made a detailed comparison of our proposed method of implementing a modulo \( 2^N \) multiplier with that of the direct approach in which the first \( N \) bits of the partial products are summed using rows of full-adders. Some general observations, however, may be made.

In both approaches, the number of full-adders (F.A's) required can give some indication of the overall hardware complexity. With
the direct method we can easily work out that the number of F.A's needed is $\sum_{n=1}^{N-1} n$. With the proposed approach, we would need

$\{(N-2) + 1\}$ F.A's for the modulo $2^{N-2}$ and modulo 2 adder-pair, along with $(N-1)$ F.A's for each of the two $(N-1)$-bit adders used in the correction circuit, giving a total of $3(N-1)$ F.A's.

The effect on the full-adder requirement with increasing wordlength $N$ is shown in the graph in Fig. 6.9. We see that with the method proposed, the full-adder count increases linearly with $N$, while that of the direct approach is proportional to $N^2$. For $N > 6$, the proposed implementation technique requires considerably fewer full-adders.

Furthermore, with the direct approach the propagation delay through the circuit, apart from the ripple delays through each row of F.A's, is dependent on $N$. With our method, however, the system delay is basically constant, and is the sum of the delays through the first correction adder, a circuit for encoding into partition blocks, the adder-pair, a circuit for decoding from the partition blocks, and the final correction adder.

6.5 Conclusions.

A novel method of implementing a general modulo $2^N$ multiplier has been presented, and consists of constraining the operands to odd values for a modified or reduced multiplier. The output of this reduced multiplier is then corrected to obtain the actual modulo $2^N$ product.

The algebraic structure of the reduced multiplier has been analysed in detail. As a result, it was shown that a reduced modulo $2^N$
Fig. 6.9. Complexity in full-adder requirement for (a) direct implementation and (b) proposed implementation of modulo $2^N$ multiplier.
multiplier is isomorphic to two adders, modulo $2^{N-2}$ and 2 respectively, operating in parallel.

Finally, it was observed that when compared with the direct way of implementing modulo $2^N$ multipliers, the proposed approach leads to a circuit which requires considerably less full-adder units and possesses a basically constant system propagation delay.
APPENDIX 6.0

Let \( g_{k',m'} \) and \( g_{k'',m''} \) be any two elements from \( Z_2 \). Then, their product \( P \) may be expressed by

\[
P = g_{k',m'} \times g_{k'',m''} = (x^{k'} \ mod \ 2^m) \times (x^{k''} \ mod \ 2^m').
\]

i.e.

\[
P = x^{k' + k''} \ mod \ 2^m + m'
\]

where \( r_k \equiv k' + k'' \mod 2 \), and hence \( r_k = 0 \) or 1,

\[
r_m \equiv m' + m'' \mod 2 \]

and \( q_k = 0 \) or 1 and \( q_m = 0 \) or 1 since the maximum value of \( (k' + k'') = 1 + 1 = 2 \), and that of \( (m' + m'') = (2^{N-2} - 1) + (2^{N-2} - 1) = 1 \times 2^{N-2} + (2^{N-2} - 2) \) respectively.

Consider now the case when \( k', k'' \) and \( m', m'' \) are such that

\[q_k = q_m = 1.\]

Then we have

\[
P = x^{2^{-N-2}} \mod \ 2^m, \quad r_m = K \times x^{2^{-N-2}} \mod \ 2^m
\]

where \( r_k = r_m \).

From Lemmas 6.4 and 6.6, we know that

\[x^2 \equiv 1, \quad \text{and} \quad 2^{-N} \equiv 1 \mod \ 2^N\]
\[ K = (Q_x 2^N + 1)(Q_w 2^N + 1), \quad Q_x, Q_w \text{ integers}, \]
\[ = Q_x Q_w 2^N 2^N + (Q_x + Q_w) 2^N + 1 \]
\[ = F 2^N + 1 \]

where
\[ F = (Q_x Q_w 2^N + Q_x + Q_w) \cdot \]
\[ \therefore P = \left[ F 2^N + 1 \right] \left[ r_x w_m \right] \]
\[ = (F, x w_m) 2^N + x w_m \]

i.e.
\[ P \equiv x w_m \text{ modulo } 2^N \]
\[ r_x w_m \text{ modulo } 2^N \]

Since the term \( x w_m \) may be written as
\[ r_x w_m = Q 2^N + g r_x, r_m \]

\[ \therefore P \equiv g r_x, r_m \text{ modulo } 2^N. \]

Using these results, we see that the two elements \( g r_x, r_m \) and
\( g r_x, r_m' \) are mapped, under modulo \( 2^N \) multiplication, to the element
\( g_{i,j} \in \mathbb{Z}_p \) such that
\[ i \equiv (l' + l'') \text{ modulo } 2 \quad \ldots (A.6.0) \]

and
\[ j \equiv (m' + m'') \text{ modulo } 2^{N-2}. \quad (A.6.1) \]

The cases for the remaining possible values of \( q_x \) and \( q_m \) may
be treated in a similar way to derive results that are identical to
equations (A.6.0) and (A.6.1).
CHAPTER 7

DECOMPOSITION STRUCTURES OF MODULO M ADDERS AND MULTIPLIERS, AND OF A SIMPLIFIED MODEL OF A SECOND-ORDER DIGITAL FILTER

7.0 Introduction.

In this chapter we extend and generalise the main ideas developed in Chapters 4 and 5.

After a brief analysis of the partition structures of both modulo-M adders and multipliers, we will show how the non-recursive second-order digital filter can be simplified such that the resulting model is easier to analyse.

It is then shown that this simplified filter may be decomposed into a parallel and/or a 'nested' cascade interconnection of submachines. A partition lattice of these submachines is developed and is shown to be related in a simple way to the familiar lattice of integers under the 'factor' relation.

7.1 Partition structure of modulo M adder.

The generation of the set of S.P. partitions for a mod-M adder modelled as an F.S.M. is described. The lattice structure of this F.S.M. is then developed and is shown to be related in a simple way to the lattice of the divisors of M under the 'factor' relation.

7.1.0 Generation of the basic S.P. partitions.

The general mod-M addition table is shown in Table 7.0. Its modelling into an F.S.M. and the algebraic analysis of the resulting
model are similar to those discussed in Sections 5.1 and 5.2.

Furthermore, in order to generate all the basic S.P. partitions of the F.S.M. mod-M adder, the basic arguments presented in Section 5.2.1.0 for the mod-$2^N$ adder are still applicable.

Thus we may say that it is sufficient to only 'identify' the state 0 and every other state $C$, where

$$C \in \{x : 0 \leq x, \text{ integer} \leq M-1\}.$$  

We will also show later that even with this simplification, only certain values of $C$ need to be considered.

When 0 and $C$ are identified, we automatically identify every other element $x$ with $(x + C) \mod M$. The resulting pairs will in turn lead to similar implications.

Consider first the pairing of 0 with $C$. One particular chain of implied pairs is

$$0, C \rightarrow C, (C + C) \rightarrow 2C, 2C + C \rightarrow \ldots (k-1)C, kC .$$

Using the transitive property of partitions, all the above pairs have to be put in the same block.

We thus see that for the pair 0, $C$ we have the linked or chain connection of all the multiples of $C$, i.e.

$$0 \rightarrow C \rightarrow 2C \rightarrow \ldots \rightarrow kC .$$

When $kC \equiv 0 \mod M$, then the identification of all the elements in the block containing 0 and $C$ will be complete.

Thus, for a given $C$, we may apply the same argument to the implied pair $x$ and $(x + C) \mod M$, to obtain

$$x \rightarrow x + C \rightarrow x + 2C \rightarrow \ldots x + kC$$
Table 7.0. General mod-M addition table.

<table>
<thead>
<tr>
<th>+</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 7.1. Mod 12 addition table.
and hence for completion of the identification of the corresponding block, we have

\[ x + kC \equiv x \mod M \]  \hspace{1cm} \text{(7.0)}

or

\[ kC = 0 \mod M, \text{ i.e. } kC = pM \]  \hspace{1cm} \text{(7.1)}

where \( p \) is an integer.

As equation (7.1) tells us that \( k \) does not depend on \( x \), we see that every block generated this way will each contain \( k \) elements.

In general, for a given \( C \), a basic S.P. partition is generated by first forming the block corresponding to the pair \( 0 \) and \( C \), and to repeat the process for every pair \( x \) and \( (x + C) \) not contained in the preceding blocks. The resulting set of such blocks is then, by construction, a basic S.P. partition on the sets of \( M \) states of the F.S.M. adder. This partition, which we call \( \pi_C \), consists of

\[ \#(\pi_C) = M/k \text{ blocks, with each block containing } m(\pi_C) = k \text{ elements,} \]

\( k \) being obtained from equation (7.1).

E.g. Let \( M = 12 \) and \( C = 3 \) with the mod-12 addition table shown in Table 7.1. The initial pair \( 0 \) and \( 3 \) leads to the sequence

\[ 0 \rightarrow 3 \rightarrow 6 \rightarrow 9 \rightarrow 0 \text{ etc.,} \]

giving the first block \((0,3,6,9)\).

The initial pair \((0,3)\) also implies the pairs \((1,4)\) and \((2,5)\).

Consequently, we have the chain sequences

\[ 1 \rightarrow 4 \rightarrow 7 \rightarrow 10 \rightarrow 1 \text{ etc. and } 2 \rightarrow 5 \rightarrow 8 \rightarrow 11 \rightarrow 2 \text{ etc.,} \]

thus resulting in the blocks

\((1,4,7,10)\) and \((2,5,8,11)\).
For a given modulus $M$, the number of blocks $\#(\pi_C)$, and the number of elements in each block $m(\pi_C)$, of the basic S.P. partition $\pi_C$ depend on the actual values of $C$ and $M$.

In general, let the greatest common divisor of $C$ and $M$ be $d$. Hence we have

$$C = c'd \quad \text{and} \quad M = m'd$$

where $c'$ and $m'$ are now coprime.

Substituting these values in equation (7.1) we obtain

$$kc'd = pm'd$$

i.e.

$$kc' = pm'$$

The number of steps $k$ is given by

$$k = \frac{pm'}{c'}$$

Since the number of steps must by definition be an integer, then the right-hand side of equation (7.5) must also be an integer. Therefore $pm'$ must be divisible by $c'$, and since $c'$ does not divide $m'$, then $p$ must be a multiple of $c'$, say $p = qc'$.

$$k = \frac{qc'm'}{c'} = qm'$$

$k$ is least when $q = 1$. Consequently, in the generation of $\pi_C$, the repetition that was mentioned in the previous section occurs at the $k$ th. step where
\[ k = 1 \cdot m' = (M/d) \]

\[ \therefore \quad \text{We conclude that} \]

\[ k = m(\pi_C) = (M/d) \]

and

\[ \#(\pi_C) = (M)/(M/d) = d \]

Consider as examples, the cases below.

(i) \( C = 1 \), i.e. \( d = 1 \)

(ii) \( C = M \), i.e. \( d = M \).

With these two cases, it is easily shown that the \( \pi_C \)'s are the trivial S.P. partitions \( \pi(1) \) and \( \pi(0) \) respectively.

(ii) \( C \) and \( M \) are co-prime.

Here the greatest common divisor of \( C \) and \( M \) is obviously 1.

Therefore like in case (i), \( \pi_C = \pi(1) \).

e.g. \( M = 12, \ C = 5 \). Then we have the following chain

\[ 0 \rightarrow 5 \rightarrow 10 \rightarrow 15(=3 \mod 12) \rightarrow 8 \rightarrow 1 \]

\[ + 6 \rightarrow 11 \rightarrow 4 \rightarrow 9 \rightarrow 2 \rightarrow 7 \rightarrow 0 . \]

Similarly for \( c = 7 \) and 11.

(iii) \( C \) divides \( M \).

In this case \( d = C \), and hence

\[ k = (M/d) = (M/C) = m(\pi_C) \]

and

\[ \#(\pi_C) = d = C . \]

e.g. \( M = 12, \ C = 4 \). Thus \( d = C = 4 \).
k = m(π₄) = (12/4) = 3 and

\#(π₄) = d = C = 4.

Consequently, we get

\[ \pi₄ = \{0, 4, 8; 1, 5, 9; 2, 6, 10; 3, 7, 11\}. \]

In the general case, let \( M = 12 \) and \( C = 8 \) say. By the direct method we have the chain sequence

\[ 0 \rightarrow 8 \rightarrow 4; 1 \rightarrow 9 \rightarrow 5; 2 \rightarrow 10 \rightarrow 6; 3 \rightarrow 1 \rightarrow 7. \]

\[ \pi₈ = \{0, 8, 4; 1, 9, 5; 2, 10, 6; 3, 11, 7\}. \]

The main result in this section is that, in the generation of the basic partitions \( \pi₈ \), we need to consider, apart from the trivial cases of \( C = 1, C = M \) and \( C \) coprime to \( M \), only those values of \( C \) that have different values of \( d \).

This greatly simplifies the generation of the lattice of S.P. partitions of a mod-M adder.

7.1.2 The partition lattice of the general mod M adder.

A simple method is presented with which the complete partition lattice of the general adder modulo \( M \) may be derived from simply knowing the divisors of \( M \).

We begin by analysing the nature of the partition 'sums' and 'products' of pairs of \( \pi₈ \)'s derived as discussed in the previous section.

Lemma 7.0. If \( d, D \) are divisors of \( M \), and \( d \) divides \( D \), then

\[ \pi₈ \geq \pi_D. \]
Proof. Let $d < D$, and $a, b$ be any two elements in a block of $\pi_D$. Then from the results in Section 7.1.1, we have

\[ b = a + QD, \quad Q \text{ an integer.} \]

Since $d$ divides $D$, i.e. $D = qx$ say, we obtain

\[ b = a + Q(qx) = a + Q'x, \quad Q' = Qq. \]

This means $a$ and $b$ are also in the same block of $\pi_d$. Since this applies to any pair in any block of $\pi_D$, then

\[ \pi_d > \pi_D. \]

When $d = D$, we have the trivial case $\pi_d = \pi_D$.

Corollary. If $d_0, d_1, \ldots, d_i, d_{i+1}, \ldots, d_n$ are divisors of $M$ such that $d_i$ divides $d_{i+1}$, then $\pi_i \geq \pi_{i+1}$ and hence

\[ \pi_0 \geq \pi_1 \geq \ldots \geq \pi_i \geq \pi_{i+1} \geq \ldots \geq \pi_n. \]

Proof. The result is obtained by applying the Lemma to successive pairs, i.e. $(d_0, d_1); (d_1, d_2); \ldots, (d_{n-1}, d_n)$.

Since using equation (7.5), a block of $\pi_{d_i}$ contains $M/d_i$ elements, and that of $\pi_{d_{i+1}}$ contains $M/d_{i+1}$ elements and is furthermore contained in a block of $\pi_{d_i}$, then a block of $\pi_{d_i}$ will contain

\[ (M/d_i)/(M/d_{i+1}) = (d_i)/d_{i+1} \text{ blocks of } \pi_{d_{i+1}}. \]

Lemma 7.1. If $d_1, d_2$ are divisors of $M$ and they do not divide each other, then

\[ (i) \quad \pi_{d_1} + \pi_{d_2} = \pi_d \quad \text{and} \quad (ii) \quad \pi_{d_1} \cdot \pi_{d_2} = \pi_D \]

where $d$ and $D$, also divisors of $M$, are the greatest common
divisor (g.c.d.) and the least common multiple (l.c.m.) respectively of \(d_1\) and \(d_2\).

Proof. Let \(d'\) divide both \(d_1\) and \(d_2\), this implies, from Lemma 7.0, that

\[
\pi_{d'} \geq \pi_{d_1} \quad \text{and} \quad \pi_{d'} \geq \pi_{d_2}.
\]

From case (iii) in Section 7.1.1, we know that \(\pi_d\), has \(d'\) blocks, each containing \((M/d')\) elements. As \(d'\) increases, so will the number of blocks, while the number of elements of each gets fewer. In other words \(\pi_d\) 'decreases'. Finally, when \(d'\) attains its greatest value, i.e. \(d\), the corresponding \(\pi_d\) will be the 'smallest'.

Thus \(\pi_d\) is the least upper bound (l.u.b.) of \(\pi_{d_1}\) and \(\pi_{d_2}\), and using the result in page 7 of Ref. 12, we can write

\[
\pi_d = \text{l.u.b.} (\pi_{d_1}, \pi_{d_2}) = \pi_{d_1} + \pi_{d_2}.
\]

To prove (ii) of the Lemma, we let \(D'\) be a common multiple of \(d_1\) and \(d_2\). Again, from Lemma 7.0, we can say that

\[
\pi_{D'} \leq \pi_{d_1} \quad \text{and} \quad \pi_{D'} \leq \pi_{d_2}.
\]

In the way similar to that for the proof of (i), it can be seen that when \(D'\) is minimum, i.e. \(D\), then \(\pi_D\) will be the 'largest' to satisfy the simultaneous inequality.

\[
\pi_D = \text{g.l.b.} (\pi_{d_1}, \pi_{d_2}) = \pi_{d_1} \cdot \pi_{d_2}.
\]

As an example, let \(M = 12\), \(d_1 = 3\) and \(d_2 = 2\), from which we have

\[
d = \text{g.c.d.} (3,2) = 1, \quad \text{and} \quad D = \text{l.c.m.} (3,2) = 6.
\]
Using the results in Section 7.1.0, we obtain

\[ \pi_3 = \{0, 3, 6, 9 ; 1, 4, 7, 10 ; 2, 5, 8, 11\} \]

and

\[ \pi_2 = \{0, 2, 4, 6, 8, 10 ; 1, 3, 5, 7, 9, 11\} \]

Forming their sum and product we have

\[ \pi_3 + \pi_2 = \pi(1) = \pi_1 \]

\[ \pi_3 \cdot \pi_2 = \{0, 6 ; 1, 7 ; 2, 8 ; 3, 9 ; 4, 10 ; 5, 11\} = \pi_6 \]

We now need the following definition.

**Definition 7.0.** A M-integer lattice is the set \( S_D \) of all the divisors of \( M \), \( M \) an integer, which is partially ordered by the relation 'is a factor of', and the operations between pairs of \( d_x, d_y \in S_D \) of finding their greatest common divisor and least common multiple, denoted, respectively by \( \square \) and \( 0 \) say, i.e.

\[ \text{g.c.d.} (d_x, d_y) \xrightarrow{\text{d}_{\square}} d_x \square d_y, \]

and

\[ \text{l.c.m.} (d_x, d_y) \xrightarrow{\text{d}_{0}} d_x 0 d_y. \]

This 'factor' relation can be conveniently represented by a Hasse diagram, as shown in Figs. 7.0(a) - (c) for \( M = 8, 18 \) and 60.

Using Lemmas 7.0 and 7.1 we may now state the following theorem.

**Theorem 7.0.** The set \( S_\tau \) of S.P. partitions of a mod \( M \) adder partially ordered by the partition inequality \( \leq \), is isomorphic to the M-integer lattice, the isomorphism being described by the one-to-one mappings \( h_i, h_j \) and \( h_k \) given by
Theorem 7.0 is merely a formal statement of the principal results discussed in Lemmas 7.0 and 7.1, and presents us with a very simple method of constructing the lattice of S.P. partitions of a mod M adder from just knowing the divisors of M.

As an example Fig. 7.1(a) shows the partition lattice of a mod 12 adder, and the isomorphic lattice of the divisors of 12 is shown in Fig. 7.1(b).

In practice, the partition lattice of the adder may be obtained directly by regarding the divisors d's as subscripts for the corresponding partition Π_d's, and geometrically reorientating the M-integer lattice as shown in Fig. 7.2 for M = 6.

7.2 S.P. partitions for a mod M multiplier.

In contrast to that of the mod M adder, the partition structure of a mod M multiplier is difficult to describe completely due to an apparent lack of a convenient regularity. As such, we have only been able to give a complete description of the lattice made up of a subset of the possible S.P. partitions. The knowledge of this sub-lattice, however, is sufficient for our subsequent search for useful decomposition structures of stored-logic digital filters.
Fig. 7.0. Some $M$-integer lattices.

Fig. 7.1. Lattices of (a) S.P. partitions of a mod 12 adder, and (b) the divisors of 12.

\begin{align*}
\tau_2 &= \{0, 2, 4, 6, 8, 10; 1, 3, 5, 7, 9, 11\} \\
\tau_3 &= \{0, 3, 6, 9; 1, 4, 7, 10; 2, 5, 8, 11\} \\
\tau_4 &= \{0, 4, 8; 1, 5, 9; 2, 6, 10; 3, 7, 11\} \\
\tau_6 &= \{0, 6; 1, 7; 2, 8; 3, 9; 4, 10; 5, 11\}
\end{align*}
Fig. 7.2. Diagrammatic derivation of the partition lattice of a mod 6 adder from the corresponding integer lattice.

\[ \Pi_2 = \{0, 2, 4; 1, 3, 5\}, \quad \Pi_3 = \{0, 3; 1, 4; 2, 5\} \]

Fig. 7.3. Complete S.P. partition lattices of typical mod-M multipliers showing the relevant sub-lattices (in broken lines).
7.2.1 Sub-lattice of multiplier's S.P. partitions.

The following theorem is basically a generalisation of Theorem 5.2 to the general mod-M multiplier.

Theorem 7.1. The lattice of S.P. partitions of a modulo M adder is a sub-lattice of the S.P. partitions of a mod M multiplier.

Proof. Consider the S.P. partition $\pi_d$ of a mod M adder, d a divisor of M, and let $x$ and $y$ be any two elements of a block of $\pi_d$.

Multiplying each by an element $a \in \mathbb{Z}_M$ we obtain

$$ax \equiv b \mod M$$

and

$$ay \equiv c \mod M$$

Subtracting equation (7.8) from (7.7), we get

$$b - c \equiv a(x - y) \mod M$$

or

$$b - c = a(x - y) + qM, \quad q \text{ an integer.}$$

(7.9)

Since $x$ and $y$ comes from a block of $\pi_d$, then if say $x > y$, then $x = y + ld$, $l$ an integer. Also, because $d$ divides $M$, we can write $M$ as $pd$, $p$ an integer.

Equation (7.9) can now be written as

$$b - c = a(ld) + q(pd)$$

$$= (a{l} + qp)d = Q'd$$

where $Q' = (a{l} + qp)$.

$$b = c + Q'd,$$

which means that the products $b$ and $c$ are still in a block of $\pi_d$.

* Recall Section 3.4.
Hence $\pi_d$ is preserved under the modulo $M$ multiplication operation.

Furthermore, since

$$\pi_{d_1} + \pi_{d_2} = \pi_{d_3} \quad \text{and} \quad \pi_{d_1} \cdot \pi_{d_2} = \pi_{d_4}$$

where $d_1, d_2, d_3, d_4$ are all divisors of $M$, the lattice is also preserved. Hence the result.

Some examples of these sub-lattices are shown in Fig. 7.3.

The ideas and experience gained in the preceding sections were found to offer a helpful insight in the analysis into the decomposition structures of digital filters.

7.3 **Decomposition structures of digital filters.**

The general second-order non-recursive digital filter, suitably transformed and modelled, is shown to be systematically decomposable. Also, the lattice of the component sub-machines is developed. This lattice provides a simple representation of the operation of subsets of these sub-machines.

7.3.0 **Notation.**

The symbols used in the subsequent discussion are briefly explained below.

If $I$ and $J$ are positive integers, with $I > J$ say, then we can write $I$ as

$$I = kJ + p \quad \cdots (7.10)$$

where $k, p$ are integers such that

$$0 \leq k \leq I/J \quad \text{and} \quad 0 \leq p < J.$$
i.e. k and p are the quotient and remainders respectively, obtained when I is divided by J.

We denote k by \( Q_J(I) \) and p by \( R_J(I) \). Sometimes, for \( R_J(I) \) we may also use \( I \mod J \) or \( (I)_J \) instead.

\[ \therefore \text{ Equation (7.10) may be written as} \]
\[ I = J \cdot Q_J(I) + R_J(I) \quad \ldots(7.11) \]

Also, if \( a, b, c, d \) are positive integers such that

\[ a + b \equiv c \mod J, \quad \text{and} \quad a \times b \equiv d \mod J, \]

then we denote c and d as

\[ c = R_J(a+b) \quad \text{and} \quad d = R_J(a\times b). \]

Finally, if \( G \) is the \( n \)-component vector \( \{g_1, g_2, \ldots, g_n\} \), then

\[ Q_J(G) = \{Q_J(g_1), Q_J(g_2), \ldots, Q_J(g_n)\} \]

and

\[ R_J(G) = \{R_J(g_1), R_J(g_2), \ldots, R_J(g_n)\}. \]

7.3.1 Simplified models of non-recursive filters.

Our subsequent analyses will be greatly assisted if we first derive a simplified version of the original non-recursive second-order section as follows.

If the actual filter has the coefficients \( a_i \) and data \( x_{n-i} \), with its output \( Z_n \) given by

\[ Z_n = \sum_{i=0}^{2} a_i x_{n-i} , \]

then its simplified version, which we call a modulo-\( d \) filter or \( (DF)_d \), is one with coefficients \( a_i \) and data \( x_{n-i} \) given by
\[(a_i)_d = R_d \left[ a_i \right] \text{ and } (x_{n-i})_d = R_d \left[ x_{n-i} \right]\]

respectively, and whose output \((Z_n)_d\) is given by

\[\begin{align*}
(Z_n)_d &= R_d \left[ \sum_{i=0}^{2} (a_i) (x_{n-i})_d \right] \\
&\quad \text{...}(7.12)
\end{align*}\]

i.e. the filter output is now operating in modulo-\(d\) arithmetic.

Using the general ideas developed in Chapter 4, we may now model \((DF)_d\) as a finite-state sequential machine (F.S.M.). Thus we may describe \((DF)_d\) by the quintuple

\[\begin{align*}
(DF)_d &= (S_d, I_d, O_d, \delta_d, \lambda_d) \\
&\quad \text{...}(7.13)
\end{align*}\]

where, if \(s_d \in S_d\), \(i_d \in I_d\) and \(o_d \in O_d\), then

\[\begin{align*}
\delta_d \{s_d; i_d\} &= \delta_d \left\{ \left( (x_{n-1})_d, (x_{n-2})_d \right) ; (x_n)_d \right\} \\
&= \left( (x_{n-1})_d, (x_{n-2})_d \right) \\
&\quad \text{...}(7.14)
\end{align*}\]

and

\[\begin{align*}
\lambda_d \{s_d; i_d\} &= (Z_n)_d \\
&\quad \text{...}(7.15)
\end{align*}\]

(Obviously if \(d = W\), where \(W\) is the maximum value of the output of the original filter, then \((DF)_W\) is identical to this filter).

In a practical filter system, the coefficients and data are each, in the simple case, constrained to a maximum positive integer...
value of \( (M-1) \) say. In this case, only its output need to be reduced modulo-\( M \) in order to derive the corresponding modulo-\( M \) filter, i.e. \((DF)_M\).

Consequently, it is not unreasonable to consider \((DF)_M\) as a 'good' simplified model of our original second-order section.

The block diagram of \((DF)_M\) is shown in Fig. 7.4. Its state transition table is shown in Table 7.2, while, for given values of \( a_0, a_1 \) and \( a_2 \), the corresponding output table may be easily constructed in a way similar to that described in Chapter 4.

If a stored-logic approach is adopted, only the output matrix need to be realised as a look-up table. Since each \( x_{n-i} \) can have \( M \) possible values, a store of \( (M)^3 \) words will be required. Furthermore, the output \( Z_M \) also has \( M \) possible values.

Consequently, the overall stored-logic capacity is \( (M)^3 \times q \) word-bits\(^*\), where \( q \) is the integer \( \geq \log_2 M \).

\(^*\) The unit 'word-bit' is more general than the commonly used 'bit' to denote the storage capacity of a memory unit. This is because \( (M)^3 \) need not be a power of 2 and the generalised unit anticipates the time when programmable logic arrays (P.L.A's) will be used as commonly as R.O.M's are today.
Fig. 7.4. F.S.M. model of a general modulo-M second-order non-recursive digital filter.
Table 7.2. Flow table for the F.S.M. equivalent of a modulo-M digital filter.
7.3.2 Homomorphic images of (DF)M.

As will be shown, the concept of a homomorphic image* of an F.S.M. is a powerful aid in the structural decomposition of the general modulo M digital filter.

Let b and c be factors of M and the corresponding F.S.M. filters operating in the arithmetic modulo b and modulo c be denoted by (DF)_b and (DF)_c respectively.

Using equation (7.13), we obtain the quintuples

\[(DF)_b = (S_b, I_b, O_b, \delta_b, \lambda_b)\]

and

\[(DF)_c = (S_c, I_c, O_c, \delta_c, \lambda_c).\]

Theorem 7.2. If b divides c, then (DF)_b is a homomorphic image of (DF)_c, with the homomorphism defined by

\[h_1 : I_c \rightarrow R_b(I_c) = I_b\]

\[h_2 : S_c \rightarrow R_b(S_c) = S_b\]

\[h_3 : O_c \rightarrow R_b(O_c) = O_b\]

such that

\[R_b[\delta_c\{s_c ; i_c\}] = \delta_b\{R_b(s_c) ; R_b(i_c)\} = \delta_b\{s_b ; i_b\}\]

...(7.16)

and

\[R_b[\lambda_c\{s_c ; i_c\}] = \lambda_b\{R_b(s_c) ; R_b(i_c)\} = \lambda_b\{s_b ; i_b\}\]

...(7.17)

* See Definition 3.1 in Chapter 3, and also Ref. 12 for the significance of homomorphic images in general.
where \( s_c \in S_c, i_c \in I_c, s_b \in S_b \) and \( i_b \in I_b \).

Proof. Using the results in the previous section, we can write \( s_c \) and \( i_c \) as \( (x_{n-1})_c, (x_{n-2})_c \) and \( (x_n)_c \) respectively. Similarly for \( s_b \) and \( i_b \).

Expanding the left-hand side of equation (7.16) we have

\[
R_b \left[ \delta_c \left\{ s_c ; i_c \right\} \right] = R_b \left[ \delta_c \left\{ \left[ (x_{n-1})_c, (x_{n-2})_c \right] ; (x_n)_c \right\} \right]
\]

\[
= R_b \left[ (x_n)_c, (x_{n-1})_c \right]
\]

\[
= \left[ (x_n)_b, (x_{n-1})_b \right]
\]

\[
= \delta_b \left\{ s_b ; i_b \right\},
\]

which is the right-hand side of equation (7.16).

To simplify the proof of (7.17), we let, with no loss in generality, \( a_2 = 0 \).

\[
y'' = (a_0)_c (x_n)_c + (a_1)_c (x_{n-1})_c \quad \text{... (7.18)}
\]

and

\[
y' = (a_0)_b (x_n)_b + (a_1)_b (x_{n-1})_b \quad \text{... (7.19)}
\]

where

\[
(a_i)_b = R_b \left[ (a_i)_c \right] \quad \text{and} \quad \langle x_{n-1} \rangle_b = R_b \left[ (x_{n-1})_c \right],
\]

\( i = 0, 1 \).

Subtracting (7.19) from (7.18) and rearranging terms, we obtain,

\[
y = y'' - y'
\]

\[
= \left\{ (a_0)_c - (a_0)_b \right\} (x_n)_c + \left\{ (x_n)_c - (x_n)_b \right\} (a_0)_b
\]

\[
+ \left\{ (a_1)_c - (a_1)_b \right\} (x_{n-1})_c + \left\{ (x_{n-1})_c - (x_{n-1})_b \right\} (a_1)_b
\]

\( \text{... (7.20)} \)
If \( a \in \{0, 1, \ldots, c-1\} \), we may express it using equation (7.11) as,

\[ a = bQ_b(a) + R_b(a) \]

i.e.

\[ \{a - R_b(a)\} = bQ_b(a) \quad \cdots (7.21) \]

We observe now that in the R.H.S. of equation (7.20), every term in the curly bracket is of the form \( \{a - R_b(a)\} \) which, from equation (7.21), implies that it is divisible by \( b \).

Therefore \( y \) itself is divisible by \( b \) and may be written as

\[ y = y'' - y' = qb \quad , \quad q \text{ an integer} \]

or

\[ y'' = qb + y' \quad \cdots (7.22) \]

If \( y'' \) and \( y' \) is now written in the form shown in equation (7.11), the above equation may be expressed as

\[ cQ_c(y'') + R_c(y'') = qb + bQ_b(y') + R_b(y') \]

or

\[ R_c(y'') = b\left[q + Q_b(y') - kQ_c(y'')\right] - cQ_c(y'') + R_b(y') \quad \cdots (7.23) \]

We have said, however, that \( b \) divides \( c \), i.e. let \( c = kb \) say.

\[ \therefore R_c(y'') = bG + R_b(y') \]

where \( G = \left[q + Q_b(y') - kQ_c(y'')\right] \).

\[ \therefore R_b \left[R_c(y'')\right] = R_b(y') \quad \cdots (7.24) \]

One may easily work out that \( R_c(y'') \) and \( R_b(y') \) are actually \((Z_n)_c\) and \((Z_n)_b\) respectively, as described by equation (7.12).
Furthermore, we may express them in the form given by equation (7.15). As a result we can now express equation (7.24) as
\[ R_b \left[ \lambda_c \left[ s_c ; i_c \right] \right] = \lambda_b \left[ s_b ; i_b \right] \]
thus proving equation (7.17).

Consequently, the triple mappings \( (h_1, h_2, h_3) \) are preserved for both state and output transitions.

Finally, to show that it is necessary that \( b \) divides \( c \), we first observe that \( c \) and \( 0 \) are both divisible by \( c \), i.e.
\[ R_c (c) = 0 \quad \text{and} \quad R_c (0) = 0. \]
Also,
\[ R_b (0) = 0. \]
If \( c \) is not a multiple of \( b \), then
\[ c = b Q_b (c) + R_b (c) \]
where \( R_b (c) \neq 0 \).

\[ \therefore \quad \text{Although} \quad R_c (c) = R_c (0) = 0, \]
\[ R_b (c) \neq R_b (0) = 0. \]

Thus, the element \( a \equiv c \equiv 0 \mod c \) has two distinct images under the mapping \( R_b (a) \), in which case the mapping is not a morphism.

As an example, let \( b = 3 \) and \( c = 6 \). In order to simplify the illustration, we will consider only the state transition or flow table.

That for \( (DF)_6 \) is shown in Table 7.3, in which the row states are reordered to demonstrate the homomorphism. In this table we also include the images of the states of \( (DF)_6 \) w.r.t. the mapping \( h_2 \), e.g., the particular subset of state-pairs \( [(1,2), (1,5), (4,2), (4,5)] \).
is mapped to the single state-pair \([1, 2]\) of \((DF)_3\), the homomorphic image of \((DF)_6\).

The flow table for this homomorphic image is shown in Table 7.4.

In general, a homomorphic image of a modulo-\(M\) filter is a "coarse" version of it which still retains its essential characteristics.

7.3.3 Parallel connection of \((DF)_b\) and \((DF)_c\).

We now analyse the parallel operation of two homomorphic image filters \((DF)_b\) and \((DF)_c\) in which \(b\) does not divide \(c\) and vice versa, but have the greatest common divisor \(d\), i.e.

\[ b = b'd \quad \text{and} \quad c = c'd \quad \text{say,} \]

where \(b'\) and \(c'\) are coprime.

With \((DF)_b\) and \((DF)_c\) described by the quintuples as in Section 7.3.2, let \((DF)_p\) be their parallel connection. Then, if Definition 3.4b in Chapter 3 is applied, \((DF)_p\) is given by

\[
(DF)_p = (DF)_b \| (DF)_c = [(S_b \times S_c), (I_b \times I_c), (O_b \times O_c), \delta_p, \lambda_p]
\]

where

\[
\delta_p \{ (s_b, s_c) ; (i_b, i_c) \} = \{ \delta_b(s_b, i_b), \delta_c(s_c, i_c) \} \quad \text{...(7.26)}
\]

and

\[
\lambda_p \{ (s_b, s_c) ; (i_b, i_c) = \{ \lambda_b(s_b, i_b), \lambda_c(s_c, i_c) \}
\]

We will now determine the relationship between \(p\) and the pair \(b, c\).
Input \((x_n)_6\)

Present state

\[ s_6 = \left[ (x_{n-1})_6 , (x_{n-2})_6 \right] \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>02</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>
Table 7.3. Flow table for modulo-6 digital filter (DF)₆.
Table 7.4. Flow table for the $R_3^{[x_n]}_6$ homomorphic image.

<table>
<thead>
<tr>
<th>Present state</th>
<th>Input $R_3^{[x_n]}_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_3^{[s_n]}_6$</td>
<td>0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 1</td>
<td>0 0</td>
</tr>
<tr>
<td>0 2</td>
<td>0 0</td>
</tr>
<tr>
<td>1 0</td>
<td>0 1</td>
</tr>
<tr>
<td>1 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 2</td>
<td>0 1</td>
</tr>
<tr>
<td>2 0</td>
<td>0 2</td>
</tr>
<tr>
<td>2 1</td>
<td>0 2</td>
</tr>
<tr>
<td>2 2</td>
<td>0 2</td>
</tr>
</tbody>
</table>
First, consider the element $\gamma \in \{0,1,2,\ldots, k-1\}$ and define the mapping $\psi$ as

$$\psi : \gamma \rightarrow \left[ R_b(\gamma), R_c(\gamma) \right] \quad \cdots (7.28)$$

i.e. $\gamma$ is reduced modulo $b$ and modulo $c$ concurrently.

**Lemma 7.2.** If $k = \frac{bc}{d}$, then $\psi$ is a one-to-one mapping.

**Proof.** For an arbitrary $\gamma$, we first form the sequence

$$\{\gamma; \gamma+1; \gamma+2; \ldots; \gamma+i; \ldots\} \mod k.$$

Since $\gamma + k \equiv \gamma \mod k$, the above sequence will first repeat at the $k$th step.

As $\gamma$ is incremented, so will its image pairs $R_b(\gamma)$ and $R_c(\gamma)$. Furthermore, we have

$$R_b(\gamma) + q'b \equiv R_b(\gamma) \mod b$$

and

$$R_c(\gamma) + q''b \equiv R_c(\gamma) \mod c.$$

Consequently, the pair $\{R_b(\gamma), R_c(\gamma)\}$ repeats when $q'$ and $q''$ are such that

$$q'b = q''c \quad \cdots (7.29)$$

Applying equation (7.25), we have

$$q'b'd = q''c'd,$$

i.e. $q'b' = q''c'$,

which says that $q'b'$ is a multiple of $c'$.

As $b'$ and $c'$ are co-prime, this is only possible if $q'$ itself is a multiple of $c'$, i.e. $q' = tc'$ say.

'. We can write $q'b$ as
(tc')b = t(c/d)b.

The smallest integer value of q'b is when t = 1, giving us

q'b = (bc)/d.

Therefore, the pair \{R_b(\gamma), R_c(\gamma)\} will first repeat itself at the (bc)/d th step. Since \gamma first repeats at the k th step, then we have

k = (bc)/d.

Each of the values \{\gamma; \gamma+1; \ldots; \gamma+(k-1)\} mod k has a unique \psi-image in the sequence

\left\{ \left[ R_b(\gamma), R_c(\gamma) \right]; \left[ (R_b(\gamma)+1), (R_c(\gamma)+1) \right]; \ldots; \left[ (R_b(\gamma)+k-1), (R_c(\gamma)+k-1) \right] \right\} \mod b, \mod c.

We are now in a position to state the following theorem.

Theorem 7.3. The filter (DF)_k is isomorphic to (DF)_p, the parallel connection of (DF)_b and (DF)_c, with the mapping \psi given by

\[ \psi : \left\{ \begin{array}{l}
i_k \mapsto \left[ R_b(i_k), R_c(i_k) \right] \\
s_k \mapsto \left[ R_b(s_k), R_c(s_k) \right] \\
o_k \mapsto \left[ R_b(o_k), R_c(o_k) \right] \end{array} \right. \]

such that

\[ \psi \left[ \delta_k \left\{ s_k; i_k \right\} \right] = \delta_p \left\{ \psi(s_k); \psi(i_k) \right\} \quad \ldots(7.30) \]

and

\[ \psi \left[ \lambda_k \left\{ s_k; i_k \right\} \right] = \lambda_p \left\{ \psi(s_k); \psi(i_k) \right\} \quad \ldots(7.31) \]

Proof. Considering the state transition function first, we expand the left-hand side of equation (7.30), thus obtaining
If we apply equation (7.16) of Theorem 7.2, we will have

\[
\delta_k \{ s_k ; i_k \} = \delta \{ s_b ; i_b \}
\]

and

\[
\delta_k \{ s_k ; i_k \} = \delta \{ s_c ; i_c \}.
\]

\[
\psi \{ \delta_k \{ s_k ; i_k \} \} = \psi \{ s_b ; i_b , s_c ; i_c \} \quad \cdots (7.33)
\]

Applying equation (7.26) to the R.H.S. of (7.33) we finally obtain

\[
\psi \{ s_k ; i_k \} = \delta \{ (s_b, s_c); (i_b, i_c) \}
\]

and hence the proof.

The proof of equation (7.31) may be obtained in a similar way.

The resulting isomorphism described by Theorem 7.3 is shown diagrammatically in Fig. 7.5.

Of course if \( k = (bc)/d = M \), then the parallel connection of \((DF)_b \) and \((DF)_c \) realises \((DF)_M \).
Fig. 7.5. Decomposition of $(DF)_k$, $k = (bc/d)$ into a parallel connection of $(DF)_b$ and $(DF)_c$. 
7.3.4 Cascade decomposition structure of a modulo $p^a$ digital filter.

In general, a modulo $M$ non-recursive digital filter may be realised as a cascade connection of its homomorphic image $(DF)_d$, which may be regarded as a 'predecessor' component, and a 'successor' component.

In particular, it is usual in practice for $M$ to be of the form $p^a$, where $p$ is a prime and $a$ an integer. In such a case, a detailed analytical description of the cascade decomposition of $(DF)_p^a$ can be derived, thus characterising completely the structures of the 'predecessor' and 'successor' components, and also the combinational mapping between them.

7.3.4.0 Notation.

Let $\alpha \in \{0,1,\ldots, p^a-1\}$ be an input, state-component, or output element of $(DF)_p^a$, and $d$ be an integer $< a$.

Dividing $\alpha$ by $p^d$ and $p^{a-d}$ in turn we obtain the following

$$\alpha = p^d Q_d(\alpha) + R_d(\alpha) \quad \ldots(7.34)$$

and

$$\alpha = p^{a-d} Q_{a-d}(\alpha) + R_{a-d}(\alpha) \quad \ldots(7.35)$$

where

$$0 \leq R_d(\alpha) < p^d \quad , \quad 0 \leq R_{a-d}(\alpha) < p^{a-d}$$

and

$$0 \leq Q_d(\alpha) < p^{a-d} \quad ,$$

i.e. $Q_d(\alpha) \in \{0,1,2,\ldots, p^{a-d}-1\}$.

In the subsequent discussion, we often interchange the notations
\[ Q^a_d(\alpha) \oplus Q^r_d(\alpha), \quad Q^d_d(\alpha) \oplus Q^r_d(\alpha) \]

and
\[
Q^a_d(\alpha) \oplus Q^r_d(\alpha), \quad \text{and similarly for } R^a_d(\alpha), R^d_d(\alpha)
\]

and \( R^a_d(\alpha) \).

Finally, if we multiply equation (7.35) by \( p^d \), we get
\[
p^d_a = p^d \left[ p^a Q^r(\alpha) + R^r(\alpha) \right]
\]
\[
= p^a Q^r(\alpha) + p^d R^r(\alpha) \quad \ldots(7.36)
\]

Since we are operating in modulo \( p^a \) arithmetic, however, we thus have
\[
p^d_a \equiv p^d R^r(\alpha) \mod p^a. \quad \ldots(7.37)
\]

### 7.3.4.1 Analysis of cascade structure of \((DF)_p^a\)

In the following, we will show that a \((DF)_p^a\) can be decomposed into a cascade connection of two image filters \((DF)_p^d\) and \((DF)_p^a - d\), with a simple combinational mapping between them.

Now let \((a_i)^*\), \((x_{n-i})^*\) and \((z_n)^*\), \( i = 0,1,2, \) be the coefficients, data and output of \((DF)_p^a\). Thus, the filter algorithm is given by
\[
(z_n)^* \equiv \sum_{i=0}^{2} (a_i)^* (x_{n-i})^* \mod p^a \quad \ldots(7.38)
\]

(From now on, we will assume that it is understood that \((z_n)^*\) is computed in modulo \( p^a \) arithmetic).

If we now express \((x_{n-i})^*\) in the form shown in equation (7.34), we get
\[
(z_n)^* = \sum_{i=0}^{2} (a_i)^* \left\{ p^d Q^r(x_{n-i})^* + R^r(x_{n-i})^* \right\}
\]
or

\[(z_n)^* = \sum_{i=0}^{2} \left\{ (a_i)^* p^d \right\} Q' (x_{n-i})^* + (a_i)^* R' (x_{n-i})^*. \]

Using equation (7.37) to replace

\[\left\{ (a_i)^* p^d \right\} \text{ by } p^d R''(a_i)^* \]

we obtain

\[(z_n)^* = \sum_{0}^{2} p^d R''(a_i)^* Q' (x_{n-i})^* + (a_i)^* R' (x_{n-i})^* \]

\[= \left\{ p^d \sum_{0}^{2} E \right\} + \left\{ \sum_{0}^{2} F \right\} \]

\(\ldots (7.39)\)

where

\[E = R''(a_i)^* Q' (x_{n-i})^*, \text{ and} \]

\[F = (a_i)^* R' (x_{n-i})^*. \]

In equation (7.39) above, we express the terms

\[\frac{2}{0} E \text{ and } \frac{2}{0} F \]

in the forms given by equations (7.35) and (7.34) respectively.

\[\therefore (z_n)^* = p^d \left[ p^{a-d} Q'' \left\{ \sum_{0}^{2} E \right\} + R'' \left\{ \sum_{0}^{2} E \right\} \right] \]

\[+ p^d Q' \left\{ \sum_{0}^{2} F \right\} + R' \left\{ \sum_{0}^{2} F \right\} \]

\(\ldots (7.40)\)

Writing \[Q' \left\{ \sum_{0}^{2} F \right\} \] in the form given in equation (7.35), we get

\[p^d Q' \left\{ \sum_{0}^{2} F \right\} \]

\[\equiv p^d \left\{ p^{a-d} Q'' \left[ Q' \left\{ \sum_{0}^{2} F \right\} \right] + R'' \left[ Q' \left\{ \sum_{0}^{2} F \right\} \right] \right\} \]

\[\equiv p^d R'' \left[ Q' \left\{ \sum_{0}^{2} F \right\} \right]. \]
Substituting this value into equation (7.40) we obtain

\[
(z_n)^* \equiv p^d \left[ R'' \left\{ \sum_0^2 \left\{ \frac{2}{0} E \right\} + R'' \left\{ Q' \left\{ \sum_0^2 F \right\} \right\} \right\} \\
+ R' \left\{ \frac{2}{0} F \right\} \right] \mod p^a \quad \ldots(7.41)
\]

If we now substitute the actual expressions for \( E \) and \( F \), we obtain from equation (7.41)

\[
(z_n)^* = p^d \left[ R'' \left\{ \sum_0^2 R''(a_1)^* Q'(x_{n-1})^* \right\} \\
+ R'' \left\{ Q' \left\{ \sum_0^2 (a_1)^* R'(x_{n-1})^* \right\} \right\} \\
+ R' \left\{ \frac{2}{0} (a_1)^* R'(x_{n-1})^* \right\} \right] \ldots(7.42)
\]

In the above equation, since the term in the final curly brackets is computed in modulo \( p^d \) arithmetic, we may replace

\((a_1)^* \) by \( R'(a_1)^* \)

\ldots\ldots(7.43)

From equation (7.34) in Section 7.3.4.0 we know that \( Q'(x_{n-1})^* \)

comes from the set
\[ \{0,1,2,\ldots, p^{a-d} -1\}, \]

which is the same one that \( R''(a_i)^* \), by definition (i.e. equation (7.35)), comes from.

In equation (7.43) above, we observe that each of the terms in the curly brackets is identical in form to that given by equation (7.12) in Section 7.3.1 which described a general modulo filter.

Therefore, we can say that two basic sub-machines of \( (DF)_{pa} \) are actually the modulo filters \( (DF)_{pd} \) and \( (DF)_{pa-d} \), whose respective outputs \( (z_n)' \) and \( (z_n)'' \) are given by

\[
(z_n)' = R' \left[ \sum_{i=0}^{2} R'(a_i)^* R'(x_{n-i})^* \right] \tag{7.44}
\]

and

\[
(z_n)'' = R'' \left[ \sum_{i=0}^{2} R''(a_i)^* Q'(x_{n-i})^* \right] \tag{7.45}
\]

Let

\[
f = R'' \left[ Q' \left[ \sum_{i=0}^{2} (a_i)^* R'(x_{n-i})^* \right] \right] \tag{7.46}
\]

.'. The output of \( (DF)_{pa} \) is given by

\[
(z_n)^* \equiv p^d \left[ (z_n)'' + f \right] + (z_n)' \mod p^a
\]

\[
\equiv p^d K + (z_n)' \mod p^a \tag{7.47}
\]

where

\[
K \equiv R'' \left[ (z_n)'' + f \right] \mod p^a
\]

i.e. \( 0 \leq K < p^{a-d} \)

\( \therefore K = Q'(z_n)^* \).
Also \(0 \leq (z_n)' < p^d\).

Equation (7.47) is the describing equation for our original modulo \(p^a\) digital filter.

The block diagram of the corresponding circuit realisation is shown in Fig. 7.6, in which stored-logic units are used to implement the relevant functions.

The storage capacity of the three components of \((\text{DF})_{pa}\) are shown below.

\[
(\text{DF})_{pd} : (p^d)^3 \log_2(p^d) \text{ word-bits}
\]

\[
(\text{DF})_{pa-d} : (p^{a-d})^3 \log_2(p^{a-d}) \text{ word-bits}
\]

\[
\text{f-combinational} : (p^d)^3 \log_2(p^{a-d}) \text{ word-bits}.
\]

The mapping \(\phi\) shown in Fig. 7.6 transforms the input \((x_n)^*\) of \((\text{DF})_{pa}\) into the pair shown below, i.e.

\[
\phi(x_n)^* = \left[Q'(x_n)^*, R'(x_n)^*\right].
\]

If \(s^*, i^*, o^*\) are the state, input and output elements of \((\text{DF})_{pa}\), and \(s_c, i_c, o_c\) those of its equivalent realised as a cascade realisation of \((\text{DF})_{pd}\) and \((\text{DF})_{pa-d}\) then we have

\[
s_c = \left[Q'(x_{n-1})^*, R'(x_{n-1})^*\right], \left[Q'(x_{n-2})^*, R'(x_{n-2})^*\right]
\]

\[
i_c = \left[Q'(x_n)^*, R'(x_n)^*\right]
\]

and

\[
o_c = \left[Q'(z_n)^*, R'(z_n)^*\right].
\]

The cascade decomposition technique that we have presented here may of course be applied to each of the components \((\text{DF})_{pd}\) and \((\text{DF})_{pa-d}\).
Fig. 7.6. Cascade realisation of $(DF)_{pa}^d$ from $(DF)_{pd}^d$.

$(DF)_{pa-d}$ and a combinational matrix.
to simplify them still further, until we reach the point when the two components are each a mod-p filter.

The consequence of this chain of decomposition levels is that it allows one to select the pair of component machines that is most suited, in terms of stored-logic capacity, to available devices.

To have an idea of the effect of the cascade decomposition on the storage capacity of the overall realisation, let \( a = 2d \) and consider only the first level decomposition.

Then we have the component filters \((DF)_{pd}\) and \((DF)_{pa-d} = (DF)_{p2d-d} = (DF)_{pd}\).

\[\ldots\] All three components of \((DF)_{pa}\) have identical stored-logic capacity equal to

\[ (p^d)^3 \log_2(p^d) \text{ word-bits}, \]

resulting in an overall capacity of

\[ 3(p^d) \cdot d \log_2(p) \text{ word-bits.} \]

The direct stored-logic implementation of \((DF)_{pa}\) will require

\[ (p^a)^3 \log_2(p^a) = (p^{2d})^3 2d \log_2(p) \text{ word-bits.} \]

\[\ldots\] The ratio of the capacity required for the cascade realisation to that of the direct stored-logic implementation is

\[ \frac{3(p^{3d}) \cdot d \log_2 p}{(p^{6d}) \cdot 2d \log_2 p} = \frac{3}{2} \left( \frac{1}{p^{3d}} \right) \text{ word-bits.} \]

As an example, let \( p = 3, a = 3 \) and \( d = 2 \). Then the modulus \( M = p^a = 27, p^d = 3^2 = 9, \) and \( p^{a-d} = 3^1 \). Also let the resulting
have the coefficient values

\[(a_0)^* = 21, (a_1)^* = 17 \quad \text{and} \quad (a_2)^* = 16.\]

Let the data values at a particular sampling instant be

\[(x_n)^* = 15, (x_{n-1})^* = 11 \quad \text{and} \quad (x_{n-2})^* = 24.\]

The direct approach will yield

\[
(z_n)^* \equiv \sum_{i=0}^{2} (a_i)^* (x_{n-i})^* \\
\equiv 21 \times 15 + 17 \times 11 + 16 \times 24 \mod(3^3) \\
\equiv 18 + 25 + 6 \mod(3^3) \\
\equiv 22 \mod(3^3).
\]

Using the decomposition technique developed, we first obtain

\[
\phi(x_n)^* = \phi(15) = (1, 6) = (Q'(15), R'(15)) \\
\phi(x_{n-1})^* = \phi(11) = (1, 2) = (Q'(11), R'(11)) \\
\phi(x_{n-2})^* = \phi(24) = (2, 6) = (Q'(24), R'(24)) \\
R'(a_0)^* = 3, \quad R'(a_1)^* = 8, \quad R'(a_2)^* = 7 \\
\text{and} \quad R''(a_0)^* = 0, \quad R''(a_1)^* = 2, \quad R''(a_2)^* = 1.
\]

From equations (7.44), (7.45) and (7.46) we have

\[
(z_n)' = R'(\{3\times6 + 8\times2 + 7\times6\}) \\
= R'(\{0 + 7 + 6\}) = 4
\]
\[
(Z_n)^{\prime\prime} = R''\left\{(0 \times 1) + (2 \times 1) + (1 \times 2)\right\}
= R''\left\{0 + 2 + 2\right\} = 1
\]

and
\[
f = R''\left(Q\left[ (21 \times 6) + (17 \times 2) + (16 \times 6) \right]\right)
= R''\left(256\right) = R''(28) = 1.
\]

\[
K = R''\left((Z_n)^{\prime\prime} + f\right) = R''[2] = 2
\]

and finally from equation (7.47) we obtain
\[
(Z_n)^{\star} \equiv 3^2 \cdot 2 + 4.
\]

\[
\phi(z_n) = (2, 4)
\]

If we apply the \(\phi\) function to the \((z_n)^{\star}\) obtained via the direct approach, we also get
\[
\phi\left((z_n)^{\star}\right) = [22] = (2, 4).
\]

7.3.5 Lattice of homomorphic images of \((DF)_M\).

From the ideas developed in Sections 7.3.1 to 7.3.4, we see that a general modulo-\(M\) filter \((DF)_M\) may be decomposed into a parallel and/or cascade connection of submachines, i.e. its homomorphic images.

The relationship between pairs of these images can be compactly and visually represented by a lattice developed below.

Let \(d, d_1, d_2, D\) be factors of \(M\) and the corresponding modulo filters be \((DF)_d\), \((DF)_{d_1}\), \((DF)_{d_2}\) and \((DF)_D\).

Also let \(d = \text{g.c.d.} (d_1, d_2)\) and \(D = \text{l.c.m.} (d_1, d_2)\), and \(F_D\) be the set of all unique images of \((DF)_M\), i.e.
$F_D = \{ h : h = (DF)_d, \text{ where } d \text{ is a factor of } M \}$. 

Now let us define a relation '$\ll$' on $F_D$ to mean that if

$$(DF)_b \ll (DF)_c,$$

then $$(DF)_b \text{ 'is a homomorphic image of' } (DF)_c.'$$

'. We have

$$(DF)_d \ll (DF)_{d_1} \text{ and } (DF)_d \ll (DF)_{d_2}.$$ 

Since $d$ is the greatest common divisor of both $d_1$ and $d_2$, then

$(DF)_d$ is the greatest (in terms of input, state component, and output symbols) modulo F.S.M. filter that is common to both $(DF)_{d_1}$ and $(DF)_{d_2}$.

Similarly

$$(DF)_{d_1} \ll (DF)_D \text{ and } (DF)_{d_2} \ll (DF)_D.$$ 

As $D$ is the least common multiple of $d_1$ and $d_2$, then $(DF)_D$ is the smallest modulo filter that $(DF)_{d_1}$ and $(DF)_{d_2}$ are the images of.

$$(D = \text{l.c.m. } (d_1, d_2) = \frac{d_1 \cdot d_2}{d}.$$ 

'. $(DF)_D$ is identical to $(DF)_k$ in Section 7.3.3). 

Thus, the set $F_D$ is partially ordered by '$\ll$' and hence

$$(F_D, \ll) \text{ is a lattice, which has a least upper bound } (DF)_D, \text{ and a greatest lower bound } (DF)_d \text{ for every pair of images } (DF)_{d_1} \text{ and } (DF)_{d_2}. \text{ It is not difficult to see that this lattice of homomorphic images is identical to the lattice of divisors of } M \text{ with the 'factor' relation discussed in Section 7.1.2.}$$
7.4 Conclusions.

The F.S.M. models for general modulo-M adders and multipliers have been successfully analysed for S.P. partitions. The lattice of these partitions for the adder is related in a simple way to the well known lattice of the divisors of M with the 'factor' relation, and was shown to be a sub-lattice of that for the mod-M multiplier.

The general non-recursive second-order digital filter has been suitably transformed and modelled to make it more amenable to algebraic partition analysis.

This simplified model was shown to be structurally decomposable into a parallel and/or a nested cascade connection of submachines, whose lattice is identical to that of the divisors of M mentioned previously.

In general, in the author's opinion, the simplified model of the filter section is not unrealistic, since in practice it may be regarded as being 'embedded' in the actual section. Furthermore, although the decomposed realisations of $D_2^M$ require input and output combinational mappings, which, if implemented with stored-logic devices, will restrict the wordlengths of the filter's data and coefficients, this may be overcome by developing practical filter sections of short wordlengths.

In the next chapter we will see how this may be achieved.
CHAPTER 8

MODULAR PARTITIONING OF BASIC
SECOND-ORDER DIGITAL FILTER

8.0 Introduction.

In this and subsequent chapters an approach different from that discussed in previous chapters is developed to partition the basic second-order digital filter.

The design philosophy is initiated by the fact that, as explained in Chapter 2, a general digital filter of a high order is realised, not directly, but as a parallel or cascade connection of basic second-order sections, each being identical in structure.

The open question then arising is whether or not it is possible to apply a similar idea to the basic second-order section itself and factor or partition it into a systematic interconnection of smaller, preferably structurally identical, modules.

In response to this question, we have successfully extracted a basic computational unit from the algorithm of the general second-order filter. This unit, which we have termed the digit convolution module has many desirable features in terms of hardware realisation. Furthermore, we have also derived the simplest elementary form of the convolution module which we have called the primitive convolution cell.

The proposed modular approach also has a useful consequence in the frequency domain analysis of digital filters.

In the following discussion, the general theory is presented first, followed by a detailed study of a special case which will be useful in practical implementations. A short discussion on the handling of negative sample values is also given.
8.1 General modular partition theory.

In this section we show how the digit convolution module is extracted, and the technique logically extended to derive the primitive convolution cell. The concept of digit templates for frequency analysis will also be explained.

8.1.0 Sequence elements represented as sequences.

In the purely analytical design and analysis of digital filters, and even in their off-line simulations on general-purpose computers, there is the tendency to regard each element of the input and impulse response sequences of a digital filter, i.e. \( \{x_n\} \) and \( \{A_n\} \) respectively, as a single conceptual entity.

In the conventional approach, there is also the assumption that once the filter coefficients have been derived, the theoretical design problem is completed. The subsequent hardware implementation is then regarded as essentially an exercise in switching circuit theory, with hardware designed at the bit level.

As an attempt to bridge the gap between formal filter design and practical hardware realisations with a systematic theory, we propose the following approach.

We observe, first of all, that number elements are most frequently represented as the sum of weighted digits, i.e. to say, if \( N \) is a natural number, then

\[
N = \sum_{k=0}^{L-1} n_k w_k \quad ,
\]

where the \( n_k \)'s and \( w_k \)'s are the digits and weights respectively.

The most common form of this weighted digit representation is
one in which the \( w_k \)'s are integer powers of a fixed number or base, \( R \) say.

Then we have

\[
N = \sum_{k=0}^{L-1} n_k R^k, \quad 0 \leq n_k < R \quad \ldots \ldots (8.1)
\]

In both cases, \( N \) may be represented as an \( L \)-tuple digit vector, i.e.,

\[
N \equiv (n_{L-1}, n_{L-2}, \ldots, n_k, \ldots, n_1, n_0) \quad \ldots \ldots (8.2)
\]

(In equation (8.2), it is implicitly understood that any vector element \( n_k \) say is weighted accordingly by \( R^k \)).

Consider now, for simplicity, just the non-recursive part of the second-order section. If \( Z_n \) is the corresponding output, then

\[
Z_n = \sum_{i=0}^{2} A_i X_{n-i} \quad \ldots \ldots (8.3)
\]

The filter impulse response is given by

\( \{A_i\} = A_0, A_1, A_2 \)

and at a particular sampling period \( nT \), the present and past input samples are given by the sequence

\( \{X_{n-1}\} = X_n, X_{n-1}, X_{n-2} \)

If the vector representation in equation (8.2) is applied to the elements of the sequences \( \{A_i\} \) and \( \{X_{n-1}\} \), we now see that each of their elements is itself a sequence, i.e.,

\[
A_i = \{a_{i,L^n-1}, a_{i,L^n-2}, \ldots, a_{i,1}, \ldots, a_{i,0}\}
\]
and

\[ X_{n-i} = \{X_{n-i, L'-1}, \ldots, X_{n-i, 2}, \ldots, X_{n-i, 0}\}. \]

Thus, while the overall filtering operation consists of the convolution between the sequences \( \{A_i\} \) and \( \{X_{n-i}\} \), the internal computation during a sampling period \( T \) is actually composed of operations between digit sequences. The detailed nature of these internal operations will now be presented.

### 8.1.1 Extraction of a basic convolution unit

The filter algorithm described by equation (8.3) is normally carried out as shown in the block diagram in Fig. 8.0. If, however, we use the vector representation for the data and coefficient words, we arrive at the block diagram shown in Fig. 8.1. There, we have shown the operation between the \( \ell'' \)th digits of the digit vectors of the \( A_i \)'s, and the \( \ell' \)th digits of the vectors of the inputs \( X_{n-i} \)'s.

As shown in Fig. 8.1, using these digits, we then form the typical partial convolution given by

\[ Z_{n, \ell', \ell''} = \left\{ \sum_{i=0}^{2} (A_i, \ell'')(X_{n-i, \ell'})(R'')(R')^{\ell'} \right\} \ldots \ldots (8.4). \]

The overall or actual convolution product is finally obtained by summing over all such typical partial convolutions, thus obtaining,

\[ Z_n = \sum_{\ell''=0}^{L''-1} \sum_{\ell''=0}^{L''-1} (R')^{\ell'} (R'')^{\ell''} \left\{ \sum_{i=0}^{2} (A_i, \ell'') (X_{n-i, \ell'}) \right\} \ldots \ldots (8.5), \]

where the \( A_i \)'s and \( X_{n-i} \)'s are expressed as \( L'' \)-tuple and \( L' \)-tuple digit vectors respectively.

If we now compare the term in the curly brackets in either equations
Fig. 8.0. Direct implementation of algorithm of non-recursive second-order filter.
$z_{n, \ell', \ell''} = \left[ \sum_{i=0}^{2} \left( A_{i, \ell''} \right) \left[ X_{n-i, \ell'} \right] \right] (R')^{\ell''} (R')^{\ell'}$

Fig. 8.1. Block diagram of 'internal' computation of filter algorithm and the extraction of a typical convolution unit.
(8.4) or (8.5), i.e.,

$$\left\{ \sum_{i=0}^{2} (A_{i, l''}) (X_{n-i, l''}) \right\} \quad \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldOTS
Fig. 8.2. Conceptual projection of second-order filter structure onto digit convolution modules of decreasing complexity.
to looking at an object through the wrong end of a telescope.

As will be explained in Section 8.2, the digital designer, by the proper choice of \( R'' \) and \( R' \), can have complete control over the hardware complexity of his D.C. module, tailoring it according to existing technology, component availability, processing speeds, etc.

From equation (8.5), we see that the original convolution is now the sum of digit convolutions. Consequently, the basic second-order section can be realised as a regular interconnection of D.C. modules, each being identical in architecture. At any sampling instant, the filter output is obtained by summing weighted outputs of these D.C.M's. The block diagram of this modular realisation is shown in Fig. 8.3.

The practical features and applications of our proposed approach are discussed in detail in Section 8.2, when we apply the partitioning technique to the case when \( R \) is an integer power of 2.

8.1.2 The primitive convolution cell.

By carrying the modular partition technique to its logical conclusion, we can derive the most elementary form of the D.C. module. The resulting unit may then be regarded as an 'atomic' building block of the filter algorithm.

The simplest form of the D.C. module described in equation (8.6) is when the fixed bases for the digit vectors of the data and coefficient words are both chosen to be 2, i.e. \( R' = R'' = 2 \).

In such a case, the data and coefficients of a typical D.C. module are simply two-valued words, i.e.,

\[
\begin{align*}
A_i, r'' & \in 0 \text{ or } 1 \\
X_{n-i}, r' &
\end{align*}
\]
Fig. 8.3. Modular realisation of second-order digital filter.
The structure of such a module is shown in Fig. 8.4, in which the data bits; \( X_{n,i} \), \( X_{n-1,i} \), \( X_{n-2,i} \); are gated by the coefficient bits; \( A_{o,i} \), \( A_{l,i} \), \( A_{2,i} \); and the resulting bit products summed by the full-adder.

From this module, it is a short step to arrive at an even simpler one which operates now in the unary base, i.e. by counting. We have termed such a unit a primitive convolution cell (p.C.C.), whose circuit structure we show in Fig. 8.5.

In this primitive cell, the data and coefficient bits are recirculated internally, and the bit products \( (A_{2,i})(X_{n-2,i}) \), \( (A_{l,i})(X_{n-1,i}) \), and \( (A_{o,i})(X_{n,i}) \) formed in time successions. These products enable or inhibit the clock input to the two-bit counter which simply counts the number of these products that are at logical '1's.

As a further explanation to the operation of the p.C.C. we have shown in Fig. 8.6 the contents of the data and coefficient flip-flops at successive count cycles during the filter sampling interval \( T \).

8.1.3 Effect of modular partitioning on frequency analysis.

The modular approach proposed is also useful when digital filters are analysed in the frequency domain.

8.1.3.0 Frequency characteristics.

The frequency responses (amplitude and phase or real and imaginary) of digital filters are usually obtained by using, as inputs, sampled complex exponentials of the form \( e^{j\omega T} \), where \( T \) is the sampling period.
Fig. 8.4. Base 2 digit convolution module.

Fig. 8.5. Circuit structure of primitive convolution cell.
Fig. 8.6. Bit patterns of primitive cell during successive count cycles during period $T$.
For the simple second-order non-recursive filter with impulse response \( \{A_i\} \), \( i = 0,1,2 \), we know that the filter output \( Z(nT) \) is the sum of the present and past two inputs appropriately scaled by the \( A_i \)'s, i.e.,

\[
Z(nT) = A_0 e^{j\omega nT} + A_1 e^{j\omega (nT-T)} + A_2 e^{j\omega (nT-2T)}
\]

\[
= \left( \sum_{i=0}^{2} A_i e^{-j\omega Ti} \right) e^{j\omega nT}
\]

...(8.7)

Thus, the output \( Z(nT) \) is the original input \( e^{j\omega nT} \) modified by the complex number \( H(j\omega) \), called the frequency response of the filter, given by

\[
H(j\omega) = \sum_{i=0}^{2} A_i e^{-j\omega Ti}
\]

...(8.8)

8.1.3.1 Digit frequency response templates.

If each of the coefficients \( A_i \)'s is an \( L'' \)-digit number in the radix \( R'' \), then from equation (8.8), we see that there are \( \left( R'' \right)^{3} \) possible combinations of \( A_0, A_1, A_2 \). Consequently, during the analytical design stage, one apparently has to deal with a very large number of different frequency responses. Thus, in the binary representation, i.e. \( R'' = 2 \), if \( L'' = 8 \) bits, then there is, using the direct method, a total of \( 2^8 \cdot 3 = 16 \) millions possible frequency responses.

We recall, however, for radix-\( R'' \) decomposition, that our typical digit convolution module as described in Section 8.1.1 has the impulse response \( \{A_{i',l''}\} \), where

\[
\{A_{i',l''}\} = A_{0,l''}, A_{1,l''}, A_{2,l''}
\]

\[ 0 \leq l'' < L'' \]
and

\[ A_{i,i''} \in \mathbb{Z}_{R''} = \{0,1,\ldots,R''-1\} \]

Using equation (8.8), the frequency response \( H_{i''}(j\omega) \) of a typical D.C. module is given by

\[ H_{i''}(j\omega) = \sum_{i=0}^{2} (A_{i,i''}) e^{-j\omega T_i} \quad \ldots (8.9) \]

and there are now only \((R'')^3\) different frequency responses involved, and the frequency response of any D.C.M. comes from this set.

A particular response from this set we have termed a digit frequency response template (D.F.R.T).

Analogous to the realisation of the second-order section from D.C. modules, the general frequency response of the filter can be built up simply by scaling and summing the appropriate D.F.R. templates, i.e.,

\[ H(j\omega) = \sum_{i''=0}^{L''-1} (H_{i''}(j\omega)(R''))^L'' \quad \ldots (8.10) \]

As an example, let \( A_o =16, \ A_1 = 23 \) and \( A_2 = 5 \). Let \( R'' = 3 \), and use equation (8.2) to obtain

\[ A_o = (1,2,1) \]
\[ A_1 = (2,1,2) \]
and
\[ A_2 = (0,1,2) . \]

There are thus three forms of D.C.M's having the impulse responses,

\[ \{A_{i,2}\} = \{1,2,0\} \]
\[ \{A_{i,1}\} = \{2,1,1\} \]
and
\[ \{A_{i,0}\} = \{1,2,2\} . \]
Hence the frequency response of the filter is obtained by summing the following weighted D.F.R. templates, viz.,

\[
\begin{align*}
&\left[1 + 2e^{-j\omega T}\right] 3^2 \\
&\left[2 + e^{-j\omega T} + e^{-j2\omega T}\right] 3^1, \quad \text{and} \\
&\left[1 + 2e^{-j\omega T} + 2e^{-j2\omega T}\right] 3^0.
\end{align*}
\]

For the simple case in which the $A_i$'s are represented by $B^n$ bits and each coefficient is then partitioned into $B^n$ blocks of 1 bit each, the set of non-trivial D.F.R. templates is small indeed, consisting, as shown in Fig. 8.7, of only six different frequency responses.

Although at this stage our analysis is only preliminary, there is a good indication that the concept of digit frequency response templates may prove to be useful in the off-line designs and especially in the interactive simulations of digital filters.
\{A_1, \xi^n\} = 0, 1, 0

\begin{align*}
\omega & \rightarrow \\
1/2T = f_S
\end{align*}

(i)

\{A_2, \xi^n\} = 0, 0, 1

\begin{align*}
\omega & \rightarrow \\
f_S
\end{align*}

(ii)

\{A_1, \xi^n\} = 1, 1, 0

\begin{align*}
\omega & \rightarrow \\
0
\end{align*}

(iii)

Figs. 8.7 (i)-(vi). Real (a) and imaginary (b) frequency responses of D.F.R. templates for radix R=2 partition.
\( \{A_{1, \xi^n}\} = 0, 1, 1 \)

(iv)

\( \{A_{1, \xi^n}\} = 1, 0, 1 \)

(v)

\( \{A_{1, \xi^n}\} = 1, 1, 1 \)

(vi)
8.2 Reprint of the article entitled

"A modular approach to the hardware implementation of digital filters",

by

M.A. Bin Nun and M.E. Woodward

published in

The Radio and Electronic Engineer,
A modular approach to the hardware implementation of digital filters

M. A. BIN NUN, B.Sc., M.Sc.∗
and
M. E. WOODWARD, B.Sc., Ph.D.∗

SUMMARY
Recent advances in the technology of medium and large-scale integrated circuits (m.s.i. and l.s.i.) have made possible economical hardware implementations for real-time digital filtering. A flexible design approach for such implementations is presented. The processing mode can be varied to give any hybrid structure between the purely serial and parallel realizations. This leads to a design approach which can be adjusted to suit hardware availability. The resulting structures are modular and are in line with current trends in m.s.i. and l.s.i. technology in that they lend themselves readily to implementations using semiconductor read-only or random access memories.

∗Department of Electronic and Electrical Engineering, University of Technology, Loughborough, Leicestershire LE11 3TU.
are related by the following difference equation:

\[ y_n = \sum_{i=0}^{2} A_i x_{n-i} - \sum_{i=1}^{3} B_i y_{n-i} \]  

(1)

where \( A_i \) and \( B_i \) are the filter coefficients obtainable from its transfer function.

The filter network in Fig. 2 consists of a non-recursive and a recursive part. Both are essentially the same in both structure and operation in that each may be represented by an expression of the form

\[ Y_n = \sum_{i=0}^{2} C_i U_{n-i} \]  

(2)

where, for the recursive part, \( C_0 = B_0 = 0 \).

In the subsequent discussion of the proposed design approach, it is therefore only necessary to consider the more general non-recursive part, which has the input-output relationship

\[ Z_n = \sum_{i=0}^{2} A_i X_{n-i} \]  

(3)

3 Design Approach

The proposed design approach is based on computing the filtering algorithm given by equation (3), not only in the conventional binary system, but in the general radix \( R \) arithmetic, where \( R \) is an integer power of 2, i.e.

\[ R = 2^p, \quad p = 1, 2, 3, \ldots \text{ etc} \]  

(4)

It is assumed that fixed-point arithmetic is used, and that, in order to process equation (3) to a specified accuracy, \( B' \) and \( B'' \) binary digits (bits) are required to represent each of the data and coefficient words respectively. Also, to simplify the discussion on the design approach, the data and coefficient words are assumed to be non-negative integers, i.e.

\[ 0 \leq X_{n-i} \leq 2^p - 1 \]

and

\[ 0 \leq A_i \leq 2^p - 1 \]

In practice, the data and coefficients are represented as binary fractions and the two's complement notation is most commonly used to handle negative numbers.

Since any \( B \)-bit binary number \( M \) can be represented in the form

\[ M = \sum_{r=0}^{B-1} m_r 2^r, \quad m_r = 0 \text{ or } 1 \]  

(5)

the binary forms of the data and coefficients will be

\[ X_{n-i} = \sum_{k=0}^{p-1} x_{n-i,k} 2^k \]  

(6)

and

\[ A_i = \sum_{j=0}^{p-1} a_{i,j} 2^j \]  

(7)

where \( i = 0, 1, 2, x_{n-i,k}, a_{i,j} = 0 \text{ or } 1 \)

Conventionally, equations (6) and (7) are substituted directly into equation (3) for the subsequent computation of the filter output \( Z_n \). A comprehensive discussion on the possible hardware organizations and processing modes for implementations based on binary arithmetic is given by Freeny in his tutorial paper.7

In the proposed modular approach, a \( B \)-bit binary number \( M \) is first partitioned into \( b \) blocks, each of \( p \) bits, where

\[ B = b \times p, \quad b \text{ and } p \text{ being integers} \]  

(8)

\( p = 3, \text{ and } p = 4 \text{ result in the familiar octal and hexadecimal systems respectively}.\)

Thus equation (5) may now be represented as

\[ M = (m_{B-1} 2^{p-1} + \ldots + m_{B-p+1} 2^1 + m_{B-p} 2^0)(2^p)^{i-1} \]

\[ + \ldots + (m_{p(k+1)-1} 2^{p-1} + \ldots + m_{p(k+1)} 2^1) + m_{pk} 2^0 + \ldots + (m_{p-1} 2^{p-1}) + + m_1 2^1 + m_0 2^0(2^p)^0 \]

or

\[ M = \sum_{k=0}^{B-1} M_k(2^p)^k \]  

(9)

where

\[ M_k = \sum_{k=0}^{p-1} m_{pk+h} 2^h \]  

(10)

and

\[ 0 \leq M_k \leq 2^p - 1 \]

Equations (9) and (10) simply mean that the \( B \)-bit binary number in equation (4) is now represented as a \( b \)-digit number in the radix \( 2^p \), where each digit is a \( p \)-bit binary number.
3.1 Example

Let \( M \) be the 6-bit \((B = 6)\) binary number, 1 0 1 1 0 1.
Expressing this in terms of equation (5), then,
\[
M = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0.
\]
If \( M \) is partitioned into three blocks, each of two bits \((b = 3, p = 2)\), then \( M \) can be expressed as
\[
M = (1 \times 2^3 + 0 \times 2^2) + (1 \times 2^1 + 1 \times 2^0) + (0 \times 2^1 + 1 \times 2^0)
\]
or, in terms of equation (9)
\[
M = (1 \times 2^3 + 0 \times 2^2)(2^1)^2 + (1 \times 2^1 + 1 \times 2^0)(2^1)^1 + (0 \times 2^1 + 1 \times 2^0)(2^1)^0.
\]
Thus, \( M \) is now represented as a 3-digit number in the radix \( 2^1 \), where the digits, \( M_k \) of equation (9), are 2-bit binary words, and, using equation (10) are given by
\[
M_0 = 01, \quad M_1 = 11 \quad \text{and} \quad M_2 = 10
\]

3.2 Computing in the Radix \( 2^n \)

In general, each data word may be partitioned into \( b' \) blocks each of \( p' \) bits, and each coefficient word into \( b'' \) blocks of \( p'' \) bits.

Using equation (9), equation (3) can be rewritten, in which \( Z_n \) the output of the non-recursive filter section, is expressed as a triple sum.
\[
Z_n = \sum_{i=0}^{2} \left[ \sum_{k=0}^{b'-1} A_{i,k}(2^p)^k \right] \left[ \sum_{k=0}^{b'-1} X_{n-i,k}(2^p)^k \right] \tag{11}
\]
where
\[
A_{i,k'} = \sum_{b''=0}^{p''-1} a_{i,b''} \cdot 2^k \tag{12}
\]
and
\[
X_{n-i,k'} = \sum_{b''=0}^{p''-1} x_{n-i,b''} \cdot 2^k \tag{13}
\]
for \( i = 0, 1, 2 \), and
\[
(b')(p') = B', \quad (b')(p') = B'
\]
The order of summation in equation (11) is then changed, resulting in
\[
Z_n = \sum_{k=0}^{p'-1} (2^p)^k \sum_{k=0}^{b'-1} (2^p)^k \sum_{i=0}^{2} \sum_{k=0}^{b''-1} \left( A_{i,k} \right) \left( X_{n-i,k} \right) \tag{14}
\]
Equation (14) forms the basis of the proposed modular approach to the hardware implementation of digital filters.

3.2.1 Example

Consider a second-order non-recursive filter having the coefficients
\[
A_0 = 6_{10}, \quad A_1 = 13_{10} \quad \text{and} \quad A_2 = 9_{10}
\]
Also, suppose that at a particular sampling instant the data consists of
\[
X_n = 12_{10}, \quad X_{n-1} = 5_{10} \quad \text{and} \quad X_{n-2} = 7_{10}
\]
If both data and coefficients are represented by 4-bit binary numbers, \((B' = B'' = 4)\), then
\[
A_0 = 0110, \quad A_1 = 1101, \quad A_2 = 1001
\]
and
\[
X_n = 1100, \quad X_{n-1} = 0101 \quad \text{and} \quad X_{n-2} = 0111
\]
Each of these words is now split into two blocks \((b' = b'' = 2)\), each of two bits \((p' = p'' = 2)\), say. The filter output \( Z_n \) at this particular sample instant may then be computed by the substitution of the actual values of the data and coefficients, now represented in the radix \( 2^2 \), into equation (3). This computation is illustrated by Table 1.

| Coefficient | \( A_0 \) | \( A_1 \) | \( A_2 \) | \( Z_n \) |
|-------------|---------|---------|---------|
| Data        | \( X_n \) | \( x_{n-1} \) | \( x_{n-2} \) | \( Z_n \) |
| 00 00       | \( 0000 \) | \( 0011 \) | \( 0100 \) | \( 0100 \) |
| 01 10       | \( 0001 \) | \( 0001 \) | \( 1000 \) | \( 1001 \) |
| 00 11       | \( 0110 \) | \( 0011 \) | \( 0010 \) | \( 1000 \) |
| 01 00       | \( 0100 \) | \( 0100 \) | \( 0100 \) | \( 0001 \) |

Each 4-bit partial product is the result of a 2-bit by 2-bit parallel multiplication, i.e. the data and coefficient blocks are multiplied in radix \( R = 2^2 \) arithmetic. The partial products in like rows are now added. This corresponds to the first summation of equation (14). The remaining stages of summation, as specified by equation (14) for the computation of the section output \( Z_n \), are shown in Table 2.

<table>
<thead>
<tr>
<th>Second, final summation according to equation (14)</th>
<th>Filter output ( Z_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R^3 )</td>
<td>( R^2 )</td>
</tr>
<tr>
<td>+ 01 00</td>
<td>1 0 1 0 0 0</td>
</tr>
<tr>
<td>+ 1 0 0 1</td>
<td>1 0 1 1 0 0 0 0</td>
</tr>
<tr>
<td>+ 0 0 1 0</td>
<td>1 0 1 1 0 0 0 0</td>
</tr>
</tbody>
</table>

As a result, the original filter, whose data and coefficients are represented by 4-bit binary words, is now regarded as being made up of four simpler units whose data and coefficients consist of only 2-bit binary words.

4 Possible Realizations

Two possible realizations for the computation of equation (14) are shown in Figs. 3 and 4. They differ both in hardware complexity and operating speed.

4.1 Parallel Processing

In the direct realization illustrated in Fig. 3, the second-order non-recursive section consists of a parallel interconnection of, what will be termed, sub-filter modules.

These modules, enclosed by the broken lines in Fig. 3, are organized into \( b' \) groups each group containing \( b'' \) modules, where \( b' \) and \( b'' \) are the number of partition
blocks as described by equation (11). For the overall section, \(b' \times b'\) modules would be required in all.

A typical module has the same general structure and computing algorithm as that of the overall section. Each of the data and coefficients of a module, however, are now only \(p'\) bit and \(p'\) bit words respectively.

In operation, these sub-filter modules implement the first summation in equation (14). The output of each group is obtained by adding the weighted outputs of all the modules in that particular group. Similarly, the section output \(Z_n\) is obtained by summing the weighted outputs of all the groups, as specified by the outer summation of equation (14).

In this direct realization, the output weightings are done by hard-wired shifts.

4.2 Sequential Processing

In contrast to the realization shown in Fig. 3, where \(b' \times b'\) modules operate concurrently, a single module, performing \(b' \times b'\) module computations in time succession, may be used.

This sequential mode of processing is illustrated in Fig. 4, in which a basic sub-filter module is time-shared among the data and coefficient blocks. The accumulator keeps a running sum of successive module outputs and also incorporates the required weightings to them.

The blocks of each of the data words are accommodated in a \((b', p')\) register store while those of each of the coefficients are stored in a \((b', p')\) circulating register store where a typical \((b, p)\) register is one having \(b\) stages, each stage accommodating a \(p\)-bit word, as shown in Fig. 5(a).

For every clock shift of the data registers these circulating coefficient stores go through a complete cycle of \(b\) shifts. Since the data registers have to be clocked \(b\) times, the required section output, \(Z_n\), will be obtained in \(b' \times b'\) register clock periods after the arrival of the section input, \(X_n\), at a particular sampling instant.

The data and coefficient blocks are so arranged as to be in increasing order of significance at the start of every sampling instant.

The \(B'\)-bit input, \(X_n\), is loaded in parallel into an input register of the form shown in Fig. 5(b). In the subsequent processing, the blocks of \(X_n\) are accessed sequentially, the accumulator being reset to zero prior to every sampling instant \(nT\).

The control of the overall section can consist of a counter and simple logic circuitry to account for the different clock rates of the data and coefficient registers.
4.3 Features

In the direct realization, as shown in Fig. 3, the circuit configuration of the overall filter section is highly modular. All the component units have an identical structure, and the interconnection between them is very regular. In consequence, the hardware implementation of the section is systematic and straightforward. Furthermore, testing and fault diagnosis are greatly simplified.

Since a typical module has the same computing algorithm as that of the original section, the 'feel' for the overall filtering operation is retained when interconnecting modules. Also, the hardware requirement of a module is determined only by the manner in which the original data and coefficient words have been partitioned. The structure is therefore easily adjusted to suit particular requirements and available hardware components. To illustrate this, consider a non-recursive section, whose data and coefficient words are represented by 6-bit and 4-bit binary words respectively. Then Table 3 shows the possible ways in which these words may be partitioned into blocks, according to equation (8).

Table 3

<table>
<thead>
<tr>
<th>Number of blocks</th>
<th>Data</th>
<th>Coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1 2 3 6</td>
<td>1 2 4</td>
</tr>
<tr>
<td>3</td>
<td>4 2 1</td>
<td>1 2 4</td>
</tr>
</tbody>
</table>

The structure of the basic module depends very much on the size of its component multipliers. For this particular filter section there are, altogether, 4 × 3 = 12 different multiplier sizes, which range from a 1-bit × 1-bit to a 6-bit × 4-bit configuration, with one convenient size being the 2-bit × 2-bit one. An interesting size is the 1-bit (data) × 4-bit, as it is of the type used in the familiar shift-and-add technique for multiplication.3 7 8 9

A final feature of the proposed approach is that, after the structure of the basic module has been decided upon, the actual mode of processing the filter algorithm is flexible. The parallel and sequential realizations, discussed previously and shown in Figs. 3 and 4, are just two extremes, hybrid forms being possible. For example, one hybrid realization might consist of a set of basic modules, operating concurrently, this being regarded as a basic time-shared unit for subsequent sequential processing. Another hybrid form might be one in which sets of data blocks are processed in parallel by a number of time-shared basic modules each operating sequentially.

In general, in between the parallel and the completely sequential realizations there is a spectrum of hardware structures and processing modes, the final choice being left to the system designer.

4.3.1. Example

Consider a non-recursive section having 8-bit data and coefficient words, (i.e. \( B' = B^* = 8 \)). If each of these words are partitioned into four blocks, each of two bits \((b' = b^* = 4, p' = p^* = 2)\), the resulting basic module has a word length of 2 bits. The direct realization of this section, as in Fig. 3, would require \( b' \times b^* = 16 \) of these basic modules. The completely sequential mode is shown in Fig. 6(a), while Figs. 6(b) and (c) illustrate two possible hybrid realizations. In the former, two basic modules make up the time-shared unit, while in the latter the input \( X_1 \) is split into two parallel halves, each of which is then processed sequentially. It is seen that when both examples of hybrid processing are compared with the completely sequential one, two basic modules are required. Their computing time, however, is reduced by half. The parallel mode, of course, has an even shorter computing time which, in this example, is sixteen times as fast as that of the completely sequential mode.

5 Practical Considerations

The performance of the overall filter section depends primarily on the structure of the basic module and the manner in which the computing algorithm is processed. The hardware requirement and implementation of a typical sub-filter module are described below, and the computation time for the section output is derived for the two extreme modes of processing. The trade-off between circuit complexity and operating speed is also discussed.

5.1 Hardware Implementation of Sub-Filter Module

The hardware organization of a typical module is shown in Fig. 7. The required arithmetic operations are \( p' \times b' \times p^* \) bit multiplications and two \((p' + p^*)\) bit additions. These operations may be implemented by any suitable m.s.i. multiplier and adder chips currently on the market. An attractive alternative, however, is to implement the module using semiconductor memories, (either read-only (r.o.m.) or random access (r.a.m.)), acting as stored look-up arithmetic tables.12

One way of using these memory chips is to replace each \( p' \times b' \times p^* \) bit multiplier, shown in Fig. 7, by a r.o.m. or r.a.m. of suitable storage. Variable and fixed coefficient multiplications using r.o.m.s are illustrated in Fig. 8(a) and (b). The former offers versatile operation at the expense of large memory storage when the word lengths of the data and coefficient blocks are large. The fixed coefficient multiplication requires less memory storage but is less versatile.
The configuration in Fig. 8(c), however, combines partially-variable coefficient capability with reasonable memory storage requirements. A total of $2^p$ different coefficients can be stored in the r.o.m.

For data and coefficient blocks of short lengths, i.e. $p'$ and $p^*$ small, even the complete sub-filter modules may be implemented as a look-up store using a r.o.m. of sufficiently large memory storage, as shown in Fig. 9. There is thus no necessity for the two $P$-bit ($P = p' + p^*$) adders previously required.

In general, the implementation of digital filters using l.s.i. semiconductor memories is simple, straightforward and incorporates programmability. It also offers the possibility of volume production of digital filter i.e. chips using existing manufacturing facilities. As digital filters are still not being used extensively enough, there is obviously a reluctance to custom-design and manufacture special i.e.s apart from very simple filter configurations.\(^\text{10}\) The market demand for semiconductor memories, however, is great enough to support its own technology.

### 5.2 Operating Speed of Filter Section

The minimum value of the sampling period $T$ for the basic nonrecursive section depends on the time it takes to compute the output $Z_n$ after the arrival of a particular input $X_n$.

If $t_M$ is the time to compute the output of a typical sub-filter module, then

$$ t_M = t_s + t_x $$

### Fig. 6. Processing modes using 2-bit basic modules.
(a) Completely sequential
(b), (c) Two possible hybrid forms

### Fig. 7. Hardware configuration of a sub-filter module.

### Fig. 8. R.o.m. realizations of $p'$ bit $\times$ $p^*$ bit multipliers.
(a) $P \times 2^p$ bit r.o.m.
(b) $P \times 2^p$ bit r.o.m.
(c) $P \times 2^{p+1}$ bit r.o.m.
where $t_m = \text{time to perform a } p' \times p'' \text{ bit multiplication, and}
\quad t_y = \text{time to sum three } (p' + p'') \text{ bit words.}
\nFor the realizations shown in Figs. 8(a) to (c), $t_s$ will be the access time of any particular r.o.m. used. Similarly, for the realization shown in Fig. 9, $t_M$ corresponds to the access time of the r.o.m. implementing the complete sub-filter module.

For the direct realization shown in Fig. 3, the total time, $T_p$, required to compute $Z_n$ is given by
\[ T_p = t_M + t_y + t_s \]

where $t_y = \text{time to sum the outputs of all the modules in any particular group}$ and $t_s = \text{time to sum the outputs of all the groups}$.

Details on the propagation delay during the process of addition can be found in any standard text on digital arithmetic (e.g. Ref. 8).

If equation (14) is processed sequentially (see Section 4.2, Fig. 4), the computing time, $T_q$, is given by
\[ T_q = (t_M + t_y + t_s) \times (b') \times (b'') \]

where $t_y = \text{time to add the module output at time } k \Delta t, \text{to the accumulator output } A, \text{previously, } A, \text{being the period of the register clock (see Fig. 4)}$.

In equation (17), it is assumed that the time taken to clock the accumulator output is much less than the computation time for the module output.

If $f_p, f_q$ are the maximum possible sampling frequencies for the section in the parallel and sequential realizations respectively, then
\[ f_p \leq \frac{1}{T_p} \text{ and } f_q \leq \frac{1}{T_q} \]

The computation time for hybrid realizations may be determined using the general principles discussed.

5.3 Trade-off Between Circuit Complexity and Operating Speed

The relative advantages of the various processing modes depend on their respective circuit complexity, module count and operating speeds. The parallel mode has the fastest processing speed and requires virtually no control circuitry. The number of sub-filter modules needed, however, is a maximum (being $b' \times b''$ modules in total). At the other extreme, the sequential mode requires only one module and an accumulator, but operates $b' \times b''$ times slower than the parallel realization. Also, some control logic is necessary for the proper accumulation and weighting of the module output. The hybrid mode offers a compromise by enabling the designer to select the most suitable combination of module count and processing speed to match his specific requirement.

6 General Second-order Section

As the recursive and non-recursive parts of the general second-order digital filter section (Fig. 2) have basically the same structure, the modular approach already discussed can be directly applied to realize this general section.

The resulting basic module then consists of two modules, each similar to that shown in Fig. 7. The block diagram of the direct modular realization of the general second-order section is shown in Fig. 10.

Since $Y_n$, the section output, is now in a feedback loop, it has to be truncated or rounded off to prevent the number of bits required for its representation from increasing indefinitely. Also $Y_n$ has to be scaled, usually by simple powers of two. Other general practical considerations such as overflow detection, limit cycle oscillations, and manipulation of negative numbers using the two's complement code, have been adequately discussed by previous authors.

Fig. 10. Modular organization of a general second-order filter section.
7 Conclusions

A method has been presented for the hardware design of general second-order digital filter sections. The procedure is systematic, flexible, and is in accordance with current hardware trends in that it makes use of m.s.i. or l.s.i. technology. The resulting hardware structures are modular, have uniform interconnection patterns, and variable processing modes.

The versatility and flexibility of the proposed technique should make possible the economical design of special-purpose digital filter hardware for any applications requiring real-time processing.

8 References


Manuscript first received by the Institution on 9th June 1975 and in final form on 4th December 1975. (Paper No. 1730/CC 261.)
© The Institution of Electronic and Radio Engineers, 1976
8.3 Negative values of filter data.

In our previous discussions, we have assumed, for simplicity, that the data words of the second-order section are positive. In this section we outline a simple method to account for the negative values as well. This method is very convenient to use when the complete digit convolution module is implemented as a stored-logic unit using a read-only memory (R.O.M.) as shown in Fig. 9 on page 201.

8.3.0 Constant bias of filter input.

A particular structure of a D.C.M. is one in which the coefficients are not partitioned at all, and each of the B'-bit data words is partitioned into B' blocks of 1 bit each. This corresponds to the R.O.M. digital filter proposed by Croisier et al. With such a structure, the filter input is in two's-complement representation. (The mechanism of this filter is fully discussed in References 6 and 50).

For our modular realisation using D.C. modules, we propose a simple interesting alternative in which the filter input is given a constant bias, with a constant correction (for a particular impulse response) at the output.

Let $X^{*}_{n-1}$ be the actual signal samples, and $X_{n-1}$ be the input samples of the second-order section. Also, assume that B'-bit registers are available to hold the data samples.

Before going into the filter, the signal is given a positive bias $\dagger$ of $2^{B'-1}$ thus resulting in the filter input given by

$$X_{n-1} = X^{*}_{n-1} + 2^{B'-1}.$$  

$\dagger$ Most analogue to digital convertors has this bias already built in, giving their digital outputs in the so called offset binary.
The resulting modified data are now processed according to equation (14) on page 197.

Consequently, if the signal has the range given by

\[- \left[ 2^{B'-1} \right] \leq X^*_{n-i} \leq \left[ 2^{B'-1} - 1 \right] \quad \ldots (8.11),\]

then the filter data is given by

\[- \left( 2^{B'-1} \right) + 2^{B'-1} = 0 \leq X_{n-i} \leq \left( 2^{B'-1} - 1 \right) + 2^{B'-1} = 2^B - 1.\]

The filter output \( Z_n \) is thus given by

\[
Z_n = \sum_{i=0}^{2} A_i X^*_{n-i} = \sum_{i=0}^{2} A_i 2^{B'-1}
\]

Since \( \sum_{i=0}^{2} A_i X^*_{n-i} \) is the true output, \( Z^*_n \) say, then we have,

\[
Z^*_n = Z_n - 2^{B'-1} \sum_{i=0}^{2} A_i \quad \ldots (8.12),
\]

where \( 2^{B'-1} \sum_{i=0}^{2} A_i \) is the constant correction term.

Expressing \( \sum_{i=0}^{2} A_i \) as a G-bit binary number, we have

\[
\sum_{i=0}^{2} A_i = C = \sum_{g=0}^{G-1} c_g 2^g
\]

\[.\] The output correction term is given by
Thus, the first $B'-1$ least significant bits of the filter output $Z_n$ need not be corrected. The overall scheme is shown in Fig. 8.8.

### 8.3.1 Distributed correction.

The direct correction method has the disadvantage that the sum $\sum_{i=0}^{2} A_i$ has to be computed separately and held in an extra $G$-bit register.

Also, one would like to make the correction scheme compatible and consistent with the philosophy of modularity and the concept of digit convolutions.

We will now show how the relevant segments of the correction term may be distributed and absorbed into the appropriate digit modules.

Firstly, we observe that the constant positive bias $2^{B'-1}$ is a $B'$-bit number whose first $(B'-1)$ digits are all zeros, i.e.,

$$2^{B'-1} = 1 \times 2^{B'-1} + 0 \times 2^{B'-2} + \ldots + 0 \times 2^1 + 0 \times 2^0. \quad \ldots (8.13)$$

This bias may be partitioned as shown in equations (9) and (10) on page 196, thus resulting in

$$2^{B'-1} = \left[1 \times 2^{p'-1} + 0 \times 2^{p'-2} + \ldots + 0 \times 2^1 + 0 \times 2^0\right] \left(2^{p'}\right)^{b'-1}$$

$$= \left[2^{p'-1}\right] \left(2^{p'}\right)^{b'-1} \quad \ldots (8.14).$$

Also, using equation (9) each coefficient may be written as
Fig. 8.8. Direct correction scheme.

\[ \sum_{i=0}^{z} A_i X_{n-i} = z_n \]

Fig. 8.9. One mechanisation of distributed correction for sequential mode.
Consequently, the correction term can be expressed as

\[
\left\{ \sum_{i=0}^{2^{B-1}-1} A_i \right\} 2^{B-1} = \left\{ \sum_{k''=0}^{b''-1} (2P')^k'' \sum_{i=0}^{2} (A_{i,k''}) (X_{n-i,b'-1}) \right\} (2P')^{b'-1} \quad \ldots(8.15),
\]

where the order of the double summation has been interchanged.

The filter output \( Z_n \), as expressed by equation (14) on page 197 is now written in a slightly different form as shown below, i.e.,

\[
Z_n = \left\{ \sum_{k''=0}^{b''-1} (2P')^k'' \sum_{i=0}^{2} (A_{i,k''}) (X_{n-i,b'-1}) \right\} (2P')^{b'-1} + \sum_{k'=0}^{b'-2} (2P')^k' \sum_{k''=0}^{b''-1} (2P')^k'' \sum_{i=0}^{2} (A_{i,k''}) (X_{n-i,k'}) \quad \ldots(8.16)
\]

Equation (8.12) can now be written in terms of equations (8.15) and (8.16). Thus the real filter output \( Z_n^* \) is given by

\[
Z_n^* = \left\{ \sum_{k''=0}^{b''-1} (2P')^k'' \sum_{i=0}^{2} (A_{i,k''}) (X_{n-i,b'-1}) \right\} (2P')^{b'-1} - (A_{i,k''}) (2P')^{b'-1} + \sum_{k'=0}^{b'-2} (2P')^k' \sum_{k''=0}^{b''-1} (2P')^k'' \sum_{i=0}^{2} (A_{i,k''}) (X_{n-i,k'}) \quad \ldots(8.18)
\]

Equation (8.18) above indicates that, in the modular circuit configuration shown in Fig. 3 on page 198, only the last group of \( b'' \) D.C. modules need to have the correction incorporated.
When the D.C. modules are implemented as look-up tables, their contents are stored in the two's complement form since it will then be easy to perform the correction subtraction.

8.3.2 Example.

Let the filter coefficients have the values

\[ A_0 = 5, \quad A_1 = 3 \quad \text{and} \quad A_2 = 7. \]

Also, at a particular sampling instant let the signal values be

\[ X_n^* = -6, \quad X_{n-1}^* = 2 \quad \text{and} \quad X_{n-2}^* = -5. \]

If 4 bits are used to represent both data and coefficient words, then the constant positive bias will be \( 2^4 - 1 = 8. \)

Consequently, the offset data values are

\[ X_n = -6 + 8 = 2, \quad X_{n-1} = 2 + 8 = 10 \quad \text{and} \quad X_{n-2} = -5 + 8 = 3. \]

Using equation (8.12), the actual filter output is given by

\[ Z_n^* = (5)(2) + (3)(10) + (7)(3) - 8(5+3+7) \]

\[ = 61 - 120 = -59. \]

If we apply the method of distributed correction as discussed in Section 8.3.1, then the stages in the computation of \( Z_n^* \) will be shown in Tables 8.0 and 8.1. Here we have selected \( R'' = R' = R = 2^2. \)
<table>
<thead>
<tr>
<th></th>
<th>R^3</th>
<th>R^2</th>
<th>R^1</th>
<th>R^0</th>
<th></th>
<th>R^3</th>
<th>R^2</th>
<th>R^1</th>
<th>R^0</th>
<th></th>
<th>R^3</th>
<th>R^2</th>
<th>R^1</th>
<th>R^0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coefficient</td>
<td>A_0: 0 1 0 1</td>
<td>×</td>
<td>A_1: 0 0 1 1</td>
<td>×</td>
<td>A_2: 0 1 1 1</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>X_n: 0 0 1 0</td>
<td>×</td>
<td>X_{n-1}: 1 0 1 0</td>
<td>×</td>
<td>X_{n-2}: 0 0 1 1</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0 0 1 0</th>
<th>0 1 1 0</th>
<th>1 0 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 0 1 0</td>
<td>0 0 0 0</td>
<td>0 0 1 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0 0 0 0</th>
<th>0 1 1 0</th>
<th>0 0 0 0</th>
</tr>
</thead>
</table>

Table 8.0. Internal computations in filter algorithms. (Distributed correction segments are enclosed in broken rectangles.)
Table 8.1. Summation of outputs of D.C. modules according to equation (8.18).

\[
\begin{array}{cccc}
R^3 & R^2 & R^1 & R^0 \\
\hline
0 & 1 & 0 & 0 & 0 & 1 \\
+ \\
0 & 0 & 0 & 1 & 0 & 1 \\
1 & \ldots & 1 & 1 & 1 & 0 & 0 \\
+ \\
1 & 1 & 1 & 1 & 0 & 0 \\
\end{array}
\begin{array}{cccc}
R^3 & R^2 & R^1 & R^0 \\
\hline
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
+ \\
0 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 0 \\
\end{array}
\begin{array}{cccc}
R^3 & R^2 & R^1 & R^0 \\
\hline
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
+ \\
1 & 0 & 0 & 0 & 1 & 0 & 1 \\
\end{array}
\]

Filter output
\( z_n^* \) in 2's-complement

\( Z_n \) in 2's-complement
8.3.3 Circuit implementation of correction scheme.

As described in Section 4.3 on page 199, there are many possible hardware structures and processing modes in the modular implementation of the second-order filter. In the light of this, only a general comment will be given on the incorporation of the distributed correction scheme into the final filter structure.

We recall that in Section 8.3.1 it was mentioned that for the completely parallel realisation, only the $b^n$ modules belonging to the $b'$th input partition block need to be corrected. With the completely sequential mode (Section 4.2, page 198), this corresponds to the correction being applied only during the last period of the data register clock cycle.

If the time-shared convolution module is implemented using a R.O.M. as in Fig. 9 on page 201, an extra control bit will be necessary which effectively doubles the original memory size.

One possible alternative is to retain the same memory capacity at the expense of some additional simple circuitry as shown in Fig. 8.9. Also, during the last or $b'$-th data register period an additional $b^n$ coefficient register cycles will be required. At the appropriate instants, the leading bits of the data blocks are 'forced' to logical 1's and the R.O.M. output two's-complemented.

8.4 Conclusions.

A comprehensive and systematic theory, based on the novel concept of a digit convolution module, has been proposed as an attempt to bridge the gap between the formal analytical design of digital filters and the implementation of their hardware structures. The theory,
which has been developed in some detail in this chapter, enables a
general second-order digital filter to be realised in a modular form
and in a variety of processing modes. The proposed modular approach
is also well suited to the technology of large-scale integrated
(L.S.I.) circuits.
CHAPTER 9

PRACTICAL HARDWARE IMPLEMENTATION
USING MODULAR APPROACH

9.0 Introduction.

In this chapter, the essential ideas of the modular approach are consolidated, and the practical implications of the desirable features of the proposed technique brought out by a practical example.

A detailed description is presented of the design of a non-recursive second-order digital filter and its practical hardware realisation for real-time operations.

After describing the processing system in general, we go on to the functional and circuit details of the main sub-system units. Results on simple input-output tests on the filter system are also given.

9.1 General filter system.

As the hardware implementation was restricted by a modest budget, the architecture that was adopted was mainly the result of a compromise between the need to reduce component count and the desire to keep a filter processing rate that is realistic for practical real-time signals.

Consequently, the filter consists of only a single digit convolution module operating in a sequential mode (see Section 4.2 on page 198). Also the filter data and coefficients are represented by 8-bit words. Furthermore, the complete D.C. module is implemented as a look-up
table using the Intel 1702* 256 × 8-bit programmable read-only memory (p.R.O.M.), it being the only large scale integrated (L.S.I.) chip that was readily available to the author at the time of design.

The complete filter system is shown in Fig. 9.0 with its functional sub-systems shown in Fig. 9.1.

The filter proper consists of the data registers, the p.R.O.M. module, and the accumulator.

The data registers enable each of the data \(X_n\), \(X_{n-1}\) and \(X_{n-2}\) to be processed two bits at a time. These bit-pairs form the first six bits of the p.R.O.M's address lines. The remaining two are used to select the different functions of the convolution module.

During every sampling period \(T\) secs., the filter system goes through eight cycles of internal computation. At each cycle, the bit-pairs access the relevant stored function of the convolution module. The time successive outputs from the p.R.O.M. are added by the accumulator, with each partial result being appropriately shifted to ensure the correct relative weightings between the module outputs. (This accumulator is set to zero at every sampling instant \(nT\)).

After the actual filter output \(Z_n\) has been computed, the buffer logic is enabled and \(Z_n\) is converted to the analogue form by a 12-bit digital to analogue converter† (D.A.C.).

Finally, to prevent the aliasing of the frequency spectrum of the filter transfer function, the output of the D.A.C. is band-limited

---

* See Appendix 9.0.
† Appendix 9.0.
Fig. 9.0. Second-order digital filter system.
Fig. 9.1. Functional sub-systems of second-order digital filter.
by an analogue low-pass reconstruction filter* which has its -3 dB point at 3.4 kHz.

9.2 Functional and circuit description of filter sub-systems.

In the following, the sub-systems that are described in some detail are the data registers and p.R.O.M. module, the accumulator, and the filter system control unit. In their corresponding circuit diagrams, only the essential wiring and pin connections are shown and labelled. Further details on the I.C. packages used may be found in the relevant manufacturers' manuals.51,52

9.2.0 p.R.O.M. module and data registers.

In terms of processing speed, hardware count and some flexibility of operation, it was considered reasonable to partition each 8-bit data word into four blocks of 2 bits, and each coefficient word into two blocks of 4 bits.

Thus we may write $X_{n-i}$ and $A_i$ in the form

$$X_{n-i} = X_{n-i,3}(2^2)^3 + X_{n-i,2}(2^2)^2 + X_{n-i,1}(2^2)^1 + X_{n-i,0}(2^2)^0 \quad \ldots (9.0)$$

and

$$A_i = A_{i,1}(2^4)^1 + A_{i,0}(2^4)^0 \quad \ldots (9.1)$$

Using equation (14) on page 197, we can express the output $Z_n$ of our second-order filter as

* Appendix 9.1.
\[ Z_n = \sum_{k' = 0}^{2} (2^{k'}) \left[ \sum_{k'' = 0}^{1} (2^{4})^{k''} \left\{ \sum_{i=0}^{2} (A_i, k'')(X_{n-i,k'}) \right\} \right] \]

\[ \ldots (9.2) \]

where the term in the curly brackets above defines a digit
\((R' = 2^{2}, R'' = 2^{4})\) convolution module having data and coefficient
word-lengths of 2 bits and 4 bits respectively. Furthermore it may
be easily shown that the maximum value of this convolution module can
be represented completely by 8 bits.

To see how this module may be implemented as a stored-logic unit,
we first expand the corresponding digit algorithm, thus obtaining,

\[ Z_{n,k',k''} = \left\{ \sum_{i=0}^{2} (A_i, k'')(X_{n-i,k'}) \right\} \]

\[ = (A_{0, k''})(X_{n,k'}) + (A_{1, k''})(X_{n-1,k'}) + (A_{2, k''})(X_{n-2,k'}) \]

\[ \ldots (9.3) \]

Thus, for a given filter impulse response \(\{A_i\}\), the partial
convolution output \(Z_{n,k',k''}\) is essentially a function of the triplet
of data blocks, i.e.,

\[ Z_{n,k',k''} = \phi_{k''} \left[ (X_{n,k'}), (X_{n-1,k'}), (X_{n-2,k'}) \right] \]

\[ \ldots (9.4) \]

Since each \(X_{n-i,k'}\) is 2 bit in length, the function \(\phi_{k''}\), for
a given value of \(k''\), has \(2^{2} \times 3 = 64\) possible combinations. Hence,
to store this function, we would require a memory of 64 words, each
8 bits long.

As there are two values of \(k''\), i.e., 0 and 1, as can be seen
in equation (9.1), we need another \((64 \times 8)\)-bit memory space.

Furthermore, since the P.R.O.M. that we have consists of
256 8-bit words, we can make use of the 128 locations remaining to repeat the above procedure for a different filter impulse response, \{B_i\} say.

The overall organisation of the p.R.O.M. memory space is shown in Fig. 9.2, in which the contents in locations: 0,0,\(\overline{x}_k\) ; 0,1,\(\overline{x}_k\) ; 1,0,\(\overline{x}_k\) ; and 1,1,\(\overline{x}_k\) are shown, where \(\overline{x}_k\) is the value of the triplet \((x_n,k', x_{n-1,k}', x_{n-2,k}')\) at a particular computation cycle. The first two bits of the address locations shown are the 7th and 8th address bits of the p.R.O.M.

The segmentation of the 8 address lines and the allocation of the resulting segments to the relevant variables is shown in Fig. 9.3.

The hardware implementation of the D.C. module and the data registers, and the corresponding circuit diagram are shown in Figs. 9.4 and 9.5 respectively.

At the start of every sampling instant \(nT\), with the mode control at '1', the 8-bit data \(x_n\) is loaded in parallel into data registers \(R_1\) and \(R_2\). After the first memory access, the mode control is brought to '0', and, for the remaining cycles of the internal computation, these registers shift their bit-contents serially. The other registers, \(R_{-3} - R_6\), are permanently connected in the serial mode. All the data registers \(R_1 - R_6\) are clocked once for every two internal cycles. Two bits, in turn, of each \(x_{n-i}\) are used as address lines to pins 3,2 ; 1,21 ; and 20, 19 of the p.R.O.M.

Pin 18 is connected to a control variable which alternates between '0' and '1' at every internal clock cycle. Thus, for a particular impulse response \(\{A_i\}\) say, memory locations 0 to 63, and 64 to 127 will be made available alternately.
<table>
<thead>
<tr>
<th>Memory location</th>
<th>Contents</th>
<th>Address bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\sum_{0}^{2} (A_{i,0})(X_{n-i,k'})$</td>
<td>0 0</td>
</tr>
<tr>
<td>63</td>
<td>$\sum_{0}^{2} (A_{i,1})(X_{n-i,k'})$</td>
<td>0 1</td>
</tr>
<tr>
<td>127</td>
<td>$\sum_{0}^{2} (B_{i,0})(X_{n-i,k'})$</td>
<td>1 0</td>
</tr>
<tr>
<td>191</td>
<td>$\sum_{0}^{2} (B_{i,1})(X_{n-i,k'})$</td>
<td>1 1</td>
</tr>
</tbody>
</table>

Fig. 9.2. The organisation of the 256 words of the p.R.O.M. convolution module into four basic sections.
Fig. 9.3. Segmentation of p.R.O.M. address lines.

Fig. 9.6. Time successive addition of p.R.O.M. outputs.
Fig. 9.4. p.R.O.M. convolution module and associated data registers.
Fig. 9.5. Circuit diagram of convolution module and data registers.
The M.S.B. of the address, pin 17, is connected to a manual switch, and is used to select either the impulse response \( \{A_i\} \) or \( \{B_i\} \). As such, the implementation incorporates a simple 'in-situ' programmability.

Successive outputs of the p.R.O.M. are appropriately weighted (by shifting) and added together by the accumulator.

9.2.0.0 **Programming the p.R.O.M.**

Some general information on the Intel 1702A 256 x 8 bit p.R.O.M. used in the implementation are given in Appendix 9.0.

Basically, it is made of enhancement field effect transistors (F.E.T's), with a floating gate, i.e. one which is embedded in an insulating layer of silicon dioxide.

The p.R.O.M. is programmed by injecting high energy electrons, produced by a controlled avalanche breakdown, through the oxide layer to form a charge on the gate.

The p.R.O.M. can be reprogrammed by first erasing its previous contents which is done by irradiating the chip with ultra-violet light for about 15 minutes.

For our implementation the programming is straightforward. The contents of locations 0 to 255, as shown in Fig.9.2, for given \( \{A_i\} \) and \( \{B_i\} \) are precomputed and the resulting data are punched onto a standard 8-bit paper tape. Then this tape is used as the input to a p.R.O.M. programmer (made by Data I/O Corporation), with the p.R.O.M. to be programmed placed in a socket provided for. After initiating the machine, the rest of the programming is automatic.

In operation the Intel 1702A p.R.O.M. has an access time of 1\( \mu \)S.
9.2.1 Accumulator.

The accumulator adds, in time succession, the outputs of the p.R.O.M. convolution module. Also, it weights each successive output by spatially shifting the previous partial result.

Its operation is best illustrated if we first expand equation (9.2) as follows;

\[ Z_n = \left\{ \sum_{i=0}^{2} (A_{i,0})(X_{n-i,0}) \right\}(2^4)^0(2^2)^0 + \left\{ \sum_{i=0}^{2} (A_{i,1})(X_{n-i,0}) \right\}(2^4)^1(2^2)^0 \\
+ \left\{ \sum_{i=0}^{2} (A_{i,0})(X_{n-i,1}) \right\}(2^4)^0(2^2)^1 + \left\{ \sum_{i=0}^{2} (A_{i,1})(X_{n-i,1}) \right\}(2^4)^1(2^2)^1 \\
+ \left\{ \sum_{i=0}^{2} (A_{i,0})(X_{n-i,2}) \right\}(2^4)^0(2^2)^2 + \left\{ \sum_{i=0}^{2} (A_{i,1})(X_{n-i,2}) \right\}(2^4)^1(2^2)^2 \\
+ \left\{ \sum_{i=0}^{2} (A_{i,0})(X_{n-i,3}) \right\}(2^4)^0(2^2)^3 + \left\{ \sum_{i=0}^{2} (A_{i,1})(X_{n-i,3}) \right\}(2^4)^1(2^2)^3 \\
\ldots (9.5) \]

In equation (9.5), we now write each term

\[ \left\{ \sum_{i=0}^{2} (A_{i,k''})(X_{n-i,k'}) \right\}(2^4)^{k''}(2^2)^{k'}, \quad k'' = 0,1; k' = 0,1,2,3, \]

in the form

\[ \{y_{n,k'',k'}\} \cdot 2^4k'' + 2k', \]

where \( \{y_{n,k'',k'}\} = \left\{ \sum_{i=0}^{2} (A_{i,k''})(X_{n-i,k'}) \right\} \) is the output of the p.R.O.M. convolution module.

The module outputs are added in time succession in the order shown in Table 9.0, with each output being weighted by \( 2^4k'' + 2k' \).

In the sequential accumulation, this weighting is equivalent to shifting the module output, at a particular computation cycle,
<table>
<thead>
<tr>
<th>Computation step</th>
<th>p.R.O.M. output $y_{n,k''},k'$</th>
<th>No. of bit shifts to left w.r.t. L.S.B. $(4k'' + 2k')$</th>
<th>relative to previous p.R.O.M. output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$y_{n,0,0}$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>$y_{n,1,0}$</td>
<td>4</td>
<td>4 left ($\ell$)</td>
</tr>
<tr>
<td>3</td>
<td>$y_{n,0,1}$</td>
<td>2</td>
<td>2 right ($r$)</td>
</tr>
<tr>
<td>4</td>
<td>$y_{n,1,1}$</td>
<td>6</td>
<td>4 $\ell$</td>
</tr>
<tr>
<td>5</td>
<td>$y_{n,0,2}$</td>
<td>4</td>
<td>2 $r$</td>
</tr>
<tr>
<td>6</td>
<td>$y_{n,1,2}$</td>
<td>8</td>
<td>4 $\ell$</td>
</tr>
<tr>
<td>7</td>
<td>$y_{n,0,3}$</td>
<td>6</td>
<td>2 $r$</td>
</tr>
<tr>
<td>8</td>
<td>$y_{n,1,3}$</td>
<td>10</td>
<td>4 $\ell$</td>
</tr>
</tbody>
</table>

Table 9.0. Order of addition of successive outputs of convolution module.
(4k" + 2k') bits to the left, relative to the L.S.B. of the filter output, as shown in Fig. 9.6. Alternatively, the module outputs can be alternatively shifted 4 bits to the left and 2 bits to the right of each other, as shown in the last column of Table 9.0.

Since, in our circuit implementation, the p.R.O.M. output is hardwired, we have to shift, instead, the partial results of the running sum of the module outputs. Consequently, the 4-bit left and 2-bit right shifts must now be replaced by 4-bit right and 2-bit left shift respectively.

Furthermore, we have designed the accumulator to truncate the filter output to 12 bits, by shifting out the two least significant bits of the partial result with every 4-bit right shift.

The mechanisation we have described is illustrated by the example shown in Fig. 9.7 in which the successive B-bit outputs of the p.R.O.M. are enclosed in rectangles. The last 2-bit left shift shown is not a physical shift but only a reinterpretation of the binary decimal point in the final filter output.

9.2.1.0 Circuit implementation of accumulator.

The accumulator hardware and its circuit diagram are shown in Figs. 9.8 and 9.9 respectively.

In Fig. 9.9, the three 4-bit adders add the p.R.O.M. module output to the shifted and delayed partial result.

The necessary 4-bit right and 2-bit left shifts are provided by the six dual 4 line to 1 line data multiplexers. Furthermore, one data input of each multiplexer is permanently connected to a logical '0'. After the 8th computation cycle, the select lines 2, 14 are
Accumulator register length

decimal point.

Fig. 9.7. Successive addition and truncation in filter accumulator.
Fig. 9.8. Accumulator unit.
Fig. 9.9. Circuit diagram of accumulator unit.
such that an all zeros combination is loaded into the three 4-bit parallel accumulator registers at the next sampling instant \((n+1)T\). This effectively resets the accumulator at every system sample period.

The outputs of the 4-bit adders are connected to the buffer logic as well as to a row of light-emitting diodes (L.E.D's).

### 9.2.2 Buffer logic.

The buffer logic consists simply of eight 2-input And-gates and three parallel 4-bit registers.

One input of every And-gate is tied to a common control line, while the other is connected to an accumulator output bit. These gates 'mask' the partial results, and are enabled only when the actual filter output \(Z_n\) is obtained.

At every sampling instant, \(Z_n\) is loaded into the 4-bit registers and held there until the next output \(Z_{n+1}\) has been computed.

### 9.2.3 Clock and control unit.

This functional unit provides the clock pulses for the accumulator and data registers, and the necessary timing pulses to synchronise the other sub-systems.

The basic circuit and its timing diagram are shown in Figs. 9.10 and 9.11. A simple modification to this basic unit for operating the filter with real-time signals is shown in Fig. 9.12. The corresponding timing diagram is shown in Fig. 9.13.

### 9.2.3.0 System clocks.

The basic clocks of the filter system consist of a manual
'bounce-free' switch, and a simple 1 MHz square wave generator made up of two standard monostable multivibrators.

Either of these two clocks is selected via a 2 to 1 multiplexer made up of one triple 3-input Nand I.C. package.

When used in a dynamic operation the system sampling clock comes from an external generator.

As can be seen in Fig. 9.10, the output of the 2:1 multiplexer is divided by a 4-bit counter, whose A (pin 12) and B (pin 9) outputs, (see also the timing diagram in Fig. 9.11), are used to clock the accumulator and the data registers respectively.

9.2.3.1 Timing pulses.

To ensure that the relevant timing and select signals are set up before the corresponding clock pulses, the outputs of the counter are delayed by 500 nS, by clocking the parallel 4-bit register with the pulse output of a monostable, which is triggered at every computation cycle by the A output.

The mode control of data registers R1 and R2 comes from the output of the Nand-gate N2. It is set to '1' prior to every sampling instant and reverts to '0' after the first computation cycle, remaining so until after the 8th cycle. As a consequence, the filter input \( x_n \) is converted from an 8-bit parallel word to a 2-bit sequential one by the input registers R1 and R2.

The delayed B output of the counter, and the output of the Nand-gate N1 are used as the select A,B inputs, pins 14, 2 respectively, of the 4:1 data multiplexers of the accumulator.

The output of N1 is also connected to the input 2:1 multiplexer,
mode control pins 6's of registers R1, R2

2 multiplexers' select inputs

100 pF 6.8 kΩ

monostable

from manual clock

from select switch

from auto clock

Fig. 9.10. Basic timing circuit.
Fig. 9.11. Timing diagram of basic circuit.
as shown in Fig. 9.10, such that when the filter is operating with the 1 MHz clock, this clock is inhibited just after the 8th computation cycle. This effectively halts the computation process once the actual filter output \( Z_n \) has been obtained.

To enable the system to operate on real-time signals, the 2:1 input multiplexer is rewired as shown in Fig. 9.12. As can be followed from the timing diagram given in Fig. 9.13, at every sampling instant, the external clock triggers the monostable, which in turn provides an output pulse of about 5\( \mu \)S wide. This acts as a 'window' to allow at least two clock pulses from the 1MHz generator to initiate the internal computation. The remaining pulses necessary to complete the internal processing is provided for by connecting the output of the Nand-gate N3 as shown.

Finally the And-gates of the buffer logic is controlled by the output of N1, and its registers are clocked by the \( Q \) output of the input monostable.

9.3 **System performance.**

Before the sub-systems were connected together, the basic clock and control circuit was tested by selecting the 1 MHz clock and monitoring, on an oscilloscope, the counter outputs, the delayed counter outputs and the outputs of Nand-gates N1 and N2.

The complete filter system was then assembled, and for its preliminary tests, eight manual switches were used as inputs, and the system output, [from the output of the 12-bit adders of the accumulator], was monitored by the row of L.E.D's.

After programming the p.R.O.M. with some simple filter coefficients,
Fig. 9.12. Simple modification of control unit.
Fig. 9.13. Timing diagram of modified input 2:1 multiplexer.
say \( A_0 = A_1 = A_2 = 255 \times 2^{-8} \), the accumulator unit was checked for correct operation. The manual clock was selected, and the filter was stepped up through its computation cycles. At each step the partial result in the accumulator was visually compared with the calculated value.

Following this check, the overall processing of the filter was simulated by setting up successive values of \( Z_n \) via the input switches. With every value of \( X_n \), the manual clock was selected and clocked twice, thus loading \( X_n \) into the data registers R1 and R2.

The 1 MHz clock was then selected and the computation subsequently completed automatically. After \( Z_n \), the filter output, was obtained (and displayed on the L.E.D's), the filter system was then prepared for the next input value \( X_{n+1} \).

The two basic forms of the input signal used are the digital impulse and step sequences, given by \( \{X_t\} \) where

\[
X_2 = \begin{cases} 
255 \times 2^{-8} & \text{for } t = 0 \\
0 & \text{for } t \neq 0 
\end{cases}
\]

and

\[
X_\ell = 255 \times 2^{-8} \quad \text{for all positive values of } \ell,
\]

respectively.

The system was then tested for a real-time operation by using a 10 kHz square wave as the basic sampling clock. The control unit was first modified as described in Section 9.2.3, and the modified circuit was checked comparing the outputs (see Fig. 9.12) of the monostables, and those of the Nand-gates \( N_b, N_c \) and \( N_a \) with the pulses in the timing diagram shown in Fig. 9.13.

The filter dynamic characteristics, for given impulse responses
was then tested by observing the response of the filter to a digital impulse. Also the frequency content of the resulting impulse response was measured using a Fourier Analyzer*.

The digital impulse was derived by first connecting together all the input bits of \( X_n \) to a common line, which is driven by the output of a pulse generator. A pulse input of 90\( \mu \)S wide, and a repetition rate of 30mS was used to obtain the impulse response. The repetition rate was so chosen such that successive impulse responses of the filter do not overlap in time. Also, the period is of a much larger duration than the 'time window'† used in the Analyzer measurements.

The step responses of the filter were also obtained by simply widening the pulse width of the test input to more than six times the period of the sampling clock.

Some typical results are shown in Figs. 14 to 17, for the impulse responses

\[ \{ A_i \}' = (255/256), (94/256), (35/256) \]

and

\[ \{ B_i \}' = (63/256), (127/256), (63/256). \]

Figs. 14 and 15 show the time domain responses of the system to a digital impulse, while Figs. 16(a) and (b) and Figs. 17(a) and (b) show the discrete Fourier transforms (magnitude and phase) of the time wave forms in Figs. 14 and 15 respectively.

The filter system implemented has a maximum sampling frequency of about 60 kHz.

* † General information on the Fourier Analyzer used for our experiments, and the relevant parameter settings are given in Appendix 9.2.
Fig. 14. Time impulse response of filter having coefficients \( \{A_1\}' \) (see text).

Fig. 15. Time impulse response of filter having coefficients \( \{B_1\} \).

(Scale: Vertical 2V/cm; Horizontal 0.2 mS/cm).
Fig. 16. Amplitude (a) and phase (b) responses of waveform in Fig. 14.
(Horizontal scale: 2.5 kHz/div.)
Fig. 17. Amplitude (a) and phase (b) responses of waveform in Fig. 15.

(Scale: Vertical $1 \times 10^{-1}$ V/div.)

(Scale: Vertical $45^\circ$/div.)
9.4 Conclusion.

The modular approach proposed in Chapter 8 has been applied to the practical design and hardware implementation of a second-order digital filter system operating on real-time signals.

The design of the main functional sub-systems was described in some detail, with emphasis on the features particular to the modular approach.

The filter system was successfully constructed and tested.
APPENDIX 9.0

General technical data on

(i) The Intel 1702A M.O.S. erasable and electrical programmable read-only memory, and

(ii) the Datel DAC-HY12BC 12 bit hybrid digital to analog converter.
2048 BIT ELECTRICALLY PROGRAMMABLE READ ONLY MEMORY

1702A -- ERASABLE & ELECTRICALLY REPROGRAMMABLE

- Fast Programming -- 2 minutes for all 2048 bits
- All 2048 bits guaranteed* programmable -- 100% factory tested
- Fully Decoded, 256x8 organization
- Static MOS -- No Clocks Required
- Inputs and Outputs DTL and TTL compatible
- Three-state Output -- OR-tie Capability
- Simple Memory Expansion -- Chip select input lead

The 1702A is a 256 word by 8 bit electrically programmable ROM ideally suited for uses where fast turn-around and pattern experimentation are important. The 1702A has undergone complete programming and functional testing on each bit position prior to shipment, thus insuring 100% programmability.

The 1702A is packaged in a 24 pin dual in-line package with a transparent quartz lid. The transparent quartz lid allows the user to expose the chip to ultraviolet light to erase the bit pattern. A new pattern can then be written into the device. This procedure can be repeated as many times as required.

The circuitry of the 1702A is entirely static; no clocks are required.

The 1702A is fabricated with silicon gate technology. This low threshold technology allows the design and production of higher performance MOS circuits and provides a higher functional density on a monolithic chip than conventional MOS technologies.

PIN CONFIGURATION

PIN NAMES

A0-A7 Address Inputs
CS Chip Select Input
dOUT1-dOUT8 Data Outputs

BLOCK DIAGRAM

NOTE: In the read mode a logic 1 at the address inputs and data outputs is a high and logic 0 is a low.

PIN CONNECTIONS

The external lead connections to the 1702A differ, depending on whether the device is being programmed or used in read mode. (See following table)

<table>
<thead>
<tr>
<th>MODE</th>
<th>PIN</th>
<th>12 (VCC)</th>
<th>13 (Program)</th>
<th>14 (CS)</th>
<th>15 (VBB)</th>
<th>16 (VGG)</th>
<th>22 (VCC)</th>
<th>23 (VCC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td></td>
<td>VCC</td>
<td>VCC</td>
<td>GND</td>
<td>VCC</td>
<td>VGG</td>
<td>VCC</td>
<td>VCC</td>
</tr>
<tr>
<td>Programming</td>
<td></td>
<td>GND</td>
<td>Program Pulse</td>
<td>GND</td>
<td>VBB</td>
<td>Pulsed VGG(VIL4P)</td>
<td>GND</td>
<td>GND</td>
</tr>
</tbody>
</table>
LOW COST, 12 BIT HYBRID DIGITAL TO ANALOG CONVERTERS

$24. IN 100's

12 Bit Binary or 3 Digit BCD
Pin-Programmable Outputs
Internal Reference & Output Amp.
Miniature Hermetic Glass Package
±15VDC Supply Only
Fast Settling Time

The DAC-HY12BC and DAC-HY12DC are low cost 12 bit binary and 3 digit BCD digit-to-analog converters manufactured in volume in Datel Systems' modern in-house thin film hybrid facility. A new level of performance has been achieved for 12 bit D/A converters at a price far below that of previously available models. These converters are complete, including a precision internal reference and a fast output operational amplifier. A high degree of application flexibility has been achieved with voltage and current outputs of ±2mA, ±1mA, 0 to +5V, 0 to +10V, +5V, ±5V, and ±10V, all available by external pin connection. These devices are available in a miniature 1.3 x 0.8 x 0.15 inch hermetically sealed glass package.

Nonlinearity is ±0.1% LSB for the DAC-HY12BC and ±0.2% LSB for the DAC-HY12DC. Temperature coefficient of bias is ±30ppm/°C and temperature coefficient of zero is ±5ppm/°C full scale maximum. Output settling time is 300 sec. to ±1LSB for current output and 3 sec. for voltage output with a 10V range. Input coding is complementary binary, complementary BCD, and complementary offset binary. Power supply requirement is ±15VDC at 35mA. No 5 volt logic supply is necessary.

The internal design of these hybrid converters consists of 12 weighted current sources, 2 thin film resistor networks, a precision zener reference, reference control circuit, and an operational amplifier. The current source switches consist of monolithic quad-current sources in conjunction with a NiCr thin film resistor network which is functionally laser trimmed to precisely set the 4-2-1 weighting. The superior tracking capability of the thin film resistors in conjunction with the tightly matched quad-current sources results in a differential linearity tempco of ±2ppm/°C, assuring monotonic operation over the full 0°C to 70°C temperature range. For excellent long term stability both the thin film resistor networks and the thin film substrate are passivated.

These DAC-HY12BC and DAC-HY12DC are Burr-Brown series 1201 and DACB5 which are pin for pin equivalents.

MECHANICAL DIMENSIONS INCHES (MM)

INPUT/OUTPUT CONNECTIONS

<table>
<thead>
<tr>
<th>PIN</th>
<th>FUNCTION</th>
<th>PIN</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BIT 1 IN</td>
<td>13</td>
<td>NO CONN.</td>
</tr>
<tr>
<td>2</td>
<td>BIT 2 IN</td>
<td>14</td>
<td>-15VDC</td>
</tr>
<tr>
<td>3</td>
<td>BIT 3 IN</td>
<td>15</td>
<td>VOL. OUT</td>
</tr>
<tr>
<td>4</td>
<td>BIT 4 IN</td>
<td>16</td>
<td>REF. IN</td>
</tr>
<tr>
<td>5</td>
<td>BIT 5 IN</td>
<td>17</td>
<td>BIPOLAR OFF</td>
</tr>
<tr>
<td>6</td>
<td>BIT 6 IN</td>
<td>18</td>
<td>10V RANGE</td>
</tr>
<tr>
<td>7</td>
<td>BIT 7 IN</td>
<td>19</td>
<td>20V RANGE</td>
</tr>
<tr>
<td>8</td>
<td>BIT 8 IN</td>
<td>20</td>
<td>CURRENT OUT</td>
</tr>
<tr>
<td>9</td>
<td>BIT 9 IN</td>
<td>21</td>
<td>GROUND</td>
</tr>
<tr>
<td>10</td>
<td>BIT 10 IN</td>
<td>22</td>
<td>+15VDC</td>
</tr>
<tr>
<td>11</td>
<td>BIT 11 IN</td>
<td>23</td>
<td>GAIN ADJ</td>
</tr>
<tr>
<td>12</td>
<td>BIT 12 IN</td>
<td>24</td>
<td>REF. OUT</td>
</tr>
</tbody>
</table>
Circuit diagram of reconstruction filter.
APPENDIX 9.2

The Hewlett-Packard 5451A Fourier analyzer system 54, 55. used in our experiment utilizes the HP2100A digital computer to calculate the Fourier transform of a time-varying voltage \( x(t) \), i.e. the transform given by,

\[
S_x(f) = \int_{-\infty}^{\infty} x(t) e^{-j2\pi ft} \, dt .
\]

In the digital implementation of this transform the input \( x(t) \) has to be sampled at finite, usually uniform, intervals of time \( \Delta t \) say.

Thus, we calculate instead,

\[
S''_x(f) = \Delta t \sum_{n=-\infty}^{n=+\infty} x(n\Delta t) e^{-j2\pi fn\Delta t} .
\]

This describes accurately the spectrum of \( x(t) \) up to some maximum frequency \( F_{\text{max}} \) which is dependent upon the sampled spacing \( \Delta t \).

Furthermore, in practice, only a time limited record of the input signal can be taken. Thus if the signal is 'observed' from some zero time reference to time \( T \) secs., then \( N = \text{number of samples} = T/\Delta t . \)

As a result we cannot now calculate the spectrum of \( x(t) \) at an infinite number of frequencies from \( 0 \) Hz to \( F_{\text{max}} \).

Thus, we end up with what is called the discrete finite transform (D.F.T) given by

\[
S'_x(m\Delta f) = \Delta t \sum_{n=0}^{N-1} x(n\Delta t) e^{-j2\pi (m\Delta f)(n\Delta t)} .
\]
In our experiments, the following parameters were used:

\[ N = \text{block size of sampled data} = 256 \]

\[ \Delta t = \text{sampling period of analyzer's analogue to digital converter} = 20\mu\text{s} \]

\[ F_{\text{max}} = 25 \text{ kHz} \]

\[ \Delta f = 50 \text{ Hz} \]
CHAPTER 10

A UNIFIED FILTER REALISATION APPROACH
USING PROGRAMMABLE STORED-LOGIC
CONVOLUTION MODULES*

10.0 Introduction.

We have shown in Chapter 8 how a second-order section may be realised in a modular way by using digit-convolution modules. Further to this, we propose and develop, in this chapter, a novel method of implementing the basic digit convolution module which combines the fast operating speed of a table look-up form with the flexibility of one realised from standard arithmetic units.

The proposed method has the added attractions in that it may be further generalised to enable the concept of stored-logic convolution modules to be used in a general-purpose computer, and also to digital filters with time-varying coefficients.

In this chapter we also describe the extension of the modular approach to a general second-order digital filter, which now includes the recursive part, and discuss the general mechanisation of high-order digital filters.

We conclude the chapter by briefly surveying other significant approaches to the hardware implementation of digital filters that have been proposed recently.

* Sections 10.1 to 10.4 are based on a paper to be presented at a forthcoming conference 56.
10.1 Basic implementations of digit-convolution module.

After having decided upon the word sizes of the data and coefficient blocks of the basic digit-convolution module, a digital designer is faced with essentially two basic ways with which to implement the module in hardware. These are shown in Figs. 10(a) and (b).

In the former, the digit module is built directly from standard multipliers and adders using known techniques. While this form offers maximum flexibility in terms of filter coefficients, it is rather expensive, requires considerable wiring, and its operating speed is dependent on the gate propagation delays.

The stored-logic form shown in Fig. 10(b), on the other hand, is compact with reduced wiring and power dissipation, and is extremely fast in operation. Its disadvantage is that different R.O.M's are needed for different filter transfer functions. Even if erasable and programmable R.O.M's like those described in Chapter 9 are used, it still requires a considerable amount of time to erase previously stored contents of the p.R.O.M's and to prepare the paper tapes for the updated look-up table.

10.2 Novel implementation using complementary convolution module.

The method to be described is an application of a recent proposal by the author (See Appendix 10.0).

We resolve the dilemma in the previous section by realising that instead of having to decide on one of the two forms in Fig. 10.0, we may actually use both in a unified structure to produce an effective combination.

The basic scheme shown in Fig. 10.1, in which the modular
3-component data block vector

\[ X_{n,k'}, X_{n-1,k'}, X_{n-2,k'} \]

Each \( A_{i,k''} \) is \( p'' \)-bits

Coefficient block store

q-bit control line, where \( q \) is the integer \( \geq \log_2 b'' \)

Fig. 10.0. Two basic hardware implementations of digit module (a) Direct method, and (b) using a memory unit with control inputs.
REAL-TIME ENVIRONMENT

REAL-TIME table look-up digital filter

INPUT ARRAY

OUTPUT

X(t)

V(t)

VI

FAST BUFFER CL> BUFFER

data-in

MUX.

y(t)

read/write DATA

SLOW

read/write control
coefficient-block select

non-real-time standard digital filter

* A/D converter, data registers.

† Accumulator, D/A converter and reconstruction filter.

Fig. 10.1. Complementary F ∗ Y convolution module.
second-order configuration is implemented in the sequential processing mode, consists of a slow non-real time simulated filter and a fast real-time stored-logic part.

We have termed this combination the complementary $Y\&Y^*$ convolution module. The 'slow' filter part of this module is basically similar to the circuit shown in Fig. 10(a) with two main differences, viz.,

(a) the data block vector $\bar{x}_k' = (X_{n,k'}, X_{n-1,k'}, X_{n-2,k'})$

are now simulated by a 3$p'$-bit binary counter ($p'$= bit length of a data block), and

(b) since this 'slow' filter is not working in real-time, the arithmetic unit needed to compute the module output for a given vector $\bar{x}_k'$ can be constructed as a serial configuration using slow and inexpensive components.

One such realisation is shown in Fig. 10.2, in which the multiplexers allow for the multiplications $(A_{0,k''})(X_{n,k'})$, $(A_{1,k''})(X_{n-1,k'})$ and $(A_{2,k''})(X_{n-2,k'})$ to be done in timesuccessions. Thus, for a given vector $\bar{x}_k'$ and coefficient block, the corresponding module output is obtained after three multiplexers' periods.

Associated with the slow simulated filter is its fast table look-up version operating in the real-time environment. This counterpart consists of a fast read/write memory (R.A.M.), the multiplexer for the data registers and the data block simulator, and the relevant interfaces

---

* An abbreviation of the term Yin-Yang, a term used in Chinese philosophy to indicate (the active and passive principles of the universe. From their interaction all things come into existence), 'Encyclopedia Americana', Vol. 29.
Fig. 10.2. 'Slow', serial simulation of digit-convolution modules.
from and to the real-time environment.

Before actual real-time processing, the \( Y \times Y \) module is first switched to its slow half. For every combination of the vector block simulator, the coefficient block registers go through the complete sequence of coefficient blocks, i.e. \( \{A_{i,0}\}, \ldots, \{A_{i,k}\}, \ldots, \{A_{i,b''-1}\} \).

For a particular vector \( x_k \), and coefficient block \( \{A_{i,k''}\} \), the module output

\[
Z_{n,k',k''} = \sum_{o}^{2} \{A_{i,k''}\} \{x_{n-i,k'}\}
\]

is computed and written into the R.A.M. store at the location specified by the vector \( x_k \).

Each coefficient block is associated with a particular combination of those address bits that are allocated as the control variables.

The programming mode is completed after the vector simulator has exhausted all possible combinations of \( x_k \).

The \( Y \times Y \) convolution module is now switched to its active real-time mode and now operates as a fast stored-logic digital filter.

A practical digital filter using this complementary convolution module idea, and based on the author's basic circuit designs, has been successfully constructed as a Final Year's Project.

10.3 The \( Y \times Y \) module in the parallel modular realisation.

The proposed approach can be easily applied to the direct parallel form of the modular realisation of the second-order section.

To illustrate this, consider the case when the \( B' \)-bit data words are each partitioned into two blocks, each of \( p' = B'/2 \) bits, and the \( B'' \)-bit coefficient words are each partitioned into two blocks, each
of \( p" = b"/2 \) bits.

The resulting parallel form is shown in Fig. 10.3 and consists of two groups of digit-convolution modules, each group containing two modules.

The groups may be programmed simultaneously, while in a particular group, each convolution module is programmed in turn, each module being selected by the control signals.

The non-real-time module used is identical to that described previously (see also Fig. 10.2), but is used in a slightly different way.

The particular block vector \( \bar{x}_k \), in a given group, say, is not connected in parallel to the address of the stored-logic modules. Instead, only one entry port is used, via the \( X_{n,k'} \) data block as shown in Fig. 10.3. Also the address vector used, i.e. \( \bar{x}_k' \), is obtained sequentially. Each value of \( Q_x \), the output of the data block multiplexer, is input and shifted horizontally along the registers of the groups.

After three such shifts, the correct combination is now addressing the modules. By this time also, the output for the module currently being written into would have been computed.

The remaining steps in the programming are as discussed previously.

10.4 Consequence of the concept of complementary \( Y \in Y \) convolution module.

Apart from its obvious practical usefulness, the primary consequence of the above concept is that it can be a useful tool to unify what have previously been apparently different approaches to the realisation of digital filters.
Fig. 10.3. Parallel realisation of second-order section using programmable digit-convolution modules.

* see Fig. 10.2.
In the literature\textsuperscript{23,50,1}, there is firstly the division between slow but flexible realisations using a general-purpose computer (G.P.C.) and real-time special-purpose realisations using hard-wired circuits. Further to this, even with special-purpose processors, there is the division between those built from standard arithmetic units, and those using R.O.M's as look-up tables.

We have already described how the last two forms can be combined together as one complementary unit. By generalising the concept, it is also possible to combine the general and special-purpose organisations.

The block diagram of a G.P.C. organised as a complementary \textit{YxY} convolution module is shown in Fig. 10.4.

The vector simulator and the 'slow' filter in Fig. 10.1 have now been replaced by a software routine, quite probably in assembler language, and the stored-logic module is now implemented using an allocated memory space in the computer core store.

The data block vector is taken in and acts, via the direct memory access (D.M.A.)\textsuperscript{21} input-output interface, as the address to the reserved memory space. In practice, this vector may have to be modified in order to match it with the address format of the computer's memory.

The software routine computes basically the sum of products algorithm for the module. The results of the computations are loaded into the allocated store at the address specified by the vector simulator.

This method is attractive in view of the current trend in microprocessor technology.
real-time part of G.P.C.

input

buffer

I/O D.M.A.*

interface

output

buffer

R.A.M. SPACE

IN CORE

STORE

data

in

SOFTWARE

ROUTINE

simulates data block vector and implements filter algorithm

off-line part of G.P.C.

address vector

REAL-TIME ENVIRONMENT

GENERAL-PURPOSE COMPUTER (G.P.C.)

* Direct memory access.

Fig. 10.4. Embedding of a complementary $y \ast Y$ convolution module within a general purpose computer.
10.5 **Application to time-varying digital filters.**

Many practical signal processings require the use of digital filters whose coefficients have to be varied at specified time intervals, e.g. in the simple digital modelling of speech production (Chapter 12, Ref. 1), the vocal tract is simulated by a digital filter whose coefficients vary, on average, every 10 msec.

Previous hardware filter designs based on R.O.M's cannot be used in such cases. Our proposed technique, however, can be used, provided the period of coefficient updating is greater than the time required for the completion of the programming phase.

The general scheme is shown in Fig. 10.5 which is basically the structure shown in Fig. 10.1 with the addition of the extra R.A.M. and the multiplexers MUX-2 and MUX-3.

Each R.A.M. operates alternately in real-time. While R.A.M.1 say, is filtering real-time signals, R.A.M.2 is being loaded with the look-up table of the new filter characteristics.

10.6 **General digital filter systems.**

Up to now we have illustrated our various proposals for the realisation of digital filters using a non-recursive second-order section. The methods may be directly generalised to include the recursive part as well. Once the general second-order section has been implemented, standard techniques may be employed to realise higher order filters.
10.6.0 General second-order section.

As explained in Chapter 2, the general second-order digital filter consists of both a non-recursive and a recursive part described by the difference equation

\[ Y_n = \left[ \sum_{i=0}^{2} A_i X_{n-i} - \sum_{i=1}^{2} B_i Y_{n-i} \right]^Q \]  

\[ = \left[ Z_n + W_n \right]^Q \]

where \[ Z_n = \sum_{i=0}^{2} A_i X_{n-i} \] and \[ W_n = \sum_{i=1}^{2} B_i Y_{n-i} \].

(See also Fig. 2 on page 196).

For simplicity, we assume that \( B_i \) and \( Y_{n-i} \) are represented by the same number of bits required to represent \( A_i \) and \( X_{n-i} \) respectively. Furthermore, \( B_i \) and \( Y_{n-i} \) are partitioned in the same way as \( A_i \) and \( X_{n-i} \) are.

Applying the method in Section 8.2 and using equation (14) on page 197, we obtain the following expression for the modular realisation of the general second-order digital filter, i.e.

\[ Y_n = K \left\{ \sum_{i=0}^{2} (A_i,k')(X_{n-i},k') \right. \]

\[ - \sum_{i=1}^{2} (B_i,k')(Y_{n-i},k') \} \]

\[ = K \left\{ \sum_{k'=0}^{b'-1} (2^{p'})^{k'} \sum_{k''=0}^{b''-1} (2^{p''})^{k''} \right. \]

\[ \left. \left[ \sum_{i=0}^{2} (A_i,k')(X_{n-i},k') - \sum_{i=1}^{2} (B_i,k')(Y_{n-i},k') \right] \right\} \]

\[ = K \left[ \sum_{i=0}^{2} (A_i,k')(X_{n-i},k') - \sum_{i=1}^{2} (B_i,k')(Y_{n-i},k') \right]^Q \]

\[ \left[ \cdot \right]^Q \] means that the value in the brackets is rounded-off to \( Q \) bits.
The term in the curly brackets is the **generalised digit-convolution** module, and consists of a pair of modules each having a hardware structure similar to that of the simple basic digit-convolution module.

The modular realisation of the general second-order section resulting from equation (10.1) are shown in Figs. 10.6 and 10.7.

If the generalised convolution module is implemented as a stored-logic unit, two R.O.M./R.A.M. units are needed. Of course, if memory units of sufficient storage capacity are available, the complete generalised digit-convolution module may be implemented as a look-up table.

### 10.6.1 General high-order filters.

It is well known (Chapter 2) that it is convenient in practice to realise a high-order filter as either a cascade or a parallel connection of basic second-order sections. Also, to take advantage of digital techniques, these connections are usually implemented using a single time-multiplexed basic second-order section.

Typical schemes are shown in Figs. 10.8(a) and (b). In the former, after the output for the first section has been computed it is fed back to the basic section via the input multiplexer. Outputs for successive sections are obtained in this way. The coefficients for the second-order sections are obtained from a circulating store.

The implementation of the parallel connection is shown in Fig. 10.8(b) in which the outputs for successive sections are accumulated and rounded-off.

These multiplexing techniques are well described in the literature \$^{1,5,4,7,50}\$. Also a recent paper \$^{67}\$ on a filter hardware laboratory is very instructive.
Fig. 10.5. Time-varying filter using programmable convolution modules.

Fig. 10.6. Modular realisation of the general second-order filter.
Typical $G_k'$ group implemented from recursive $\&$ non-recursive pairs of digit-convolution modules.

Fig. 10.7.
Fig. 10.8. Multiplexing a single second-order section to implement a high-order filter realised in the cascade (a) and parallel (b) forms.

(a) circulating store of [S.O.S]'s coefficients

(b)
10.7 Recent proposals for the hardware implementation of digital filters.

In recent years, most of the published proposals are either extensions or generalisations of the basic structures proposed by Croisier et al and developed by Peled and Liu. Some other approaches, however, have been suggested.

De Mori et al describes a special-purpose processor for both digital filtering and fast Fourier transformation (FFT), using emitter-coupled logic (ECL) hardware components. Its main feature is a parallel multiplier which requires about 15% less hardware than normal implementations. This is because the least significant weight bits in the partial products are not processed. Instead a correcting bias is added. This multiplier performs a multiplication and two double-precision additions simultaneously.

Peled, on the other hand, proposed a machine organisation of a dedicated digital signal processor in which the filter coefficients are represented in the specialised canonical signed-digit code. The resulting realisation requires the minimum number of add/subtract operations to implement the required multiplications and additions. It promises a significantly better performance than existing realisations using standard multiplier packages.

De Mori also proposed an interesting implementation scheme based on logic-in-memory cellular arrays. The resulting structures allow very fast filters to be designed because the time required for a single multiplication, (due to a large overlapping between the execution of the overall multiplications), gives only an additive contribution to the total time required to compute an output sample.

The iterative nature of De Mori's filter structure is most suitable
for special customed-designed L.S.I. implementation of high frequency
digital filters.

As the research on finding good hardware structures for digital
filters is actively progressing, there are certainly other interesting
ideas yet to come.

10.8 Conclusions.

An interesting realisation approach to the realisation of digital
filters using programmable stored-logic digit-convolution modules
has been proposed and developed. The scheme promises to unify what
were previously apparently different realisation approaches.

The modular approach has also been extended to the general
second-order digit filter by the concept of a generalised digit-
convolution module.

Finally, other interesting implementation proposals published
recently were briefly mentioned and commented upon.
Versatile digital arithmetic unit with rams

A common method of implementing fast arithmetic circuits is to realise them as look-up tables using semiconductor read-only memories, but they are still expensive for the general user to purchase and programme. In addition, their contents cannot be altered to suit different operating parameters. Even with field-programmable and erasable memories, it still takes time to prepare the data paper tapes and to erase previously stored contents with an ultra-violet source.

A simple and efficient alternative makes use of random access read/write memories instead. As shown in Fig. 1, the ram, when operating in real time, is addressed by the signal from the environment and outputs the relevant word to it.

The ram is volatile, so that it has to have its contents written every time the system is switched on. But because the contents are computable, the ram is easily programmed using an operand simulator (os) and slow arithmetic unit (saU). The os generates all possible combinations of input values, and the saU, which is easily designed with conventional serial arithmetic techniques, computes the required arithmetic function.

As an example, a binary digital filter is shown in Fig. 2. It has four six-bit weighting coefficients, \( A_0, A_1, A_2 \) and \( A_3 \), and its output \( Y_n \) is given by:

\[
Y_n = \sum_{k=0}^{3} x_{n-k} A_k
\]  

(1)

A table look-up realisation of the portion of the filter which is enclosed by the broken lines in Fig. 2 would require a 16-word by eight-bit ram addressed by the binary vector \( (x_0, x_1, x_2, x_3) \).

The complete circuit is detailed in Fig. 3. The os is a simple four-bit counter type SN7493, while the saU is a four-bit serial one-bit adder which is made up of a pair of full adders type SN74183 and an SN7462 two-bit adder. The filter output \( Y_n \) is computed as follows. Expressing the weighted coefficients in binary:

\[
A_k = \sum_{i=0}^{n} a_{k,i} 2^i
\]  

(2)

where \( k = 0, 1, 2, 3 \) and \( a_{k,i} = 0 \) or 1. By putting (2) in (1) and re-ordering the double summation:

\[
Y_n = \sum_{i=0}^{n} x_{n-i} a_{k,i} 2^i
\]  

(3)

Thus coefficient bits of the same significance are added in one bit time, each bit \( a_{k,i} \) being weighted by the relevant simulated data bit \( x_{n-i} \) for every combination of the os output.

In operation, the weights \( A_k \) to \( A_0 \) are entered serially, via the SN74167 two-to-one multiplexer, into the SN7491 eight-bit registers. This is done by connecting the programme clock line to S4, which is used as a manual clock. Each weight is padded by two zeroes following its most significant bit. S1 is set to one, the counters are reset to zero, and the ram address is now switched to the os output via S2. The programme clock line is reconnected to the clock, S4 set to one, and the clock is initiated.

The three-bit and four-bit counters and the associated nand logic are designed so that, after every eight clock pulses, the write enable to the ram is strobed to zero, writing in the relevant filter output which has meanwhile been computed. The next clock pulse brings the write enable back to one and clocks in the new input pair. Between the counters preventing data being written into the wrong address. After another eight pulses the process is repeated.

The system has been designed to stop automatically after the os output has reached \((1,1,1,1)\) and the necessary arithmetic corresponding to this address has been duly computed. S4 is now set to zero, disabling the clock and the counters reset to zero, thus holding the write enable to one. After switching the ram address back to the environment input, the memory is now ready for real-time application.

Digital arithmetic units built with this technique are fast in operation, with a 30 ns delay typical for the example given, and inexpensive, since rams are general purpose MSI/LSI devices, and extremely versatile, since operand parameters may be altered quickly.

The slow arithmetic unit and the ram may be used in their more traditional roles when the system is not operating in the fast mode.

M. A. Bin Nun, Department of Electronics and Electrical Engineering, University of Technology, Loughborough, Leics.
CHAPTER 11

REVIEW AND RECOMMENDATIONS

11.0 Introduction.

We recall that the purpose of the research reported in this Thesis is to develop a systematic hardware realisation theory of digital filters which will logically link their formal analytical designs and their hardwired practical implementations.

In contrast to existing techniques, a systems approach was adopted for the general investigation of possible modular architectures for the basic second-order digital filter. In particular, we proposed and developed two novel methods. In the first, we modelled the complete second-order section as a finite-state sequential machine (F.S.M.), which was then analysed using the theory of machine decomposition using S.P. partitions. In the second method, we analysed the internal computation of the filter algorithm by developing the idea of digit convolutions.

Below we review briefly the main results of our investigations and in Section 11.2 we recommend the possible directions along which the foundation presented in this Thesis may be extended.

11.1 Review of main results.

When a non-recursive second-order digital filter is modelled directly as an F.S.M., we showed that the state transition function of the resulting model is already in its simplest form. In addition, we proved that this model is also a minimally reduced machine. A partial state reduction is possible, however, if the filter output
is expressed as a multi-component word. For the recursive sections, it is possible to minimise their F.S.M. models. Some of the reduced machines also contain S.P. partitions. State minimisation, however, becomes less useful with increasing wordlengths, and the non-linearities introduced in the filter transfer function by quantisation effects make it difficult to generalise the results found.

By applying the same modelling and analysis technique to the adder and multiplier units making up the filter, we first showed that modulo $2^N$ adders and multipliers may be realised as loop-free cascade interconnections of sub-machines which require less memory space to implement than the direct stored-logic implementation. In addition, we further developed the stored-logic implementation of the conventional N-bit parallel adder, and two useful F.S.M. models of the N-bit by N-bit parallel multiplier.

We then generalised our findings to adders and multipliers modulo an arbitrary base M, and showed that the partition lattices of their F.S.M. models are easily generated from the lattice of the divisors of M under the 'factor' relation. The understanding gained was useful in showing that a second-order section, suitably and realistically simplified, possesses a regular algebraic decomposition structure. We also introduced the concept of the homomorphic images of an F.S.M. filter and their corresponding lattice.

As an interesting 'spin-off', we found that, as an alternative to the loop-free structure, a modulo $2^N$ multiplier may be implemented in a novel way which requires a low full-adder count and a propagation delay that is essentially independent of wordlengths.

Using our second method, we extracted one possible basic
computational unit for digital convolution which we termed the
digit-convolution module (D.C.M.). The second-order digital filter
may now be regarded as a regular interconnection of D.C.M's. This
modular approach favours the digital designer since it is easy to
construct, test and maintain the filter hardware, the circuit
structure is directly expandable in terms of computational accuracy,
and is also flexible in its processing modes.

The modular theory was consolidated and its essential attractive
features brought out by the construction of a practical real-time
prototype filter using semiconductor memories.

Finally, we introduced and developed the concept of the $Y \sim Y$
complementary pair of 'slow' and 'fast' digit-convolution modules
which unified what were previously apparently different approaches
to digital filter realisations.

11.2 Possible directions for development.

The basic research that we have carried out has led to a useful
theoretical framework for the implementation of digital filters.
This may be used as a foundation for further research along the
following possible directions.

As an attractive alternative to read-only memories (R.O.M's),
a study may be made on the use of the newer programmable logic arrays
(P.L.A's) to implement the homomorphic images of modulo-M filters
and digit-convolution modules as stored-logic structures. As with
P.L.A's selected minterms of the logic variables may be programmed,
their use should lead to a more efficient 'packing' of stored
information.
With the modular realisation using D.C.M's, one could investigate the application of pipelining to increase the overall throughput or computation rate of the filter section. This study may incorporate the analogue-to-digital and digital-to-analogue converters since they are also usually organised in groups of data digits.

In spite of the rapid progress made in the technology of microprocessors, they are still considered slow for most real-time work if programmed to implement the digital convolution algorithm directly. A more realistic processing rate should be possible, however, if a microprocessor is used to implement only the digit-convolution module. The overall filter is now realised as an array of microprocessors. An even superior performance may be obtained if the newer bit-slice bipolar microprocessors are used instead.

Furthermore, each microprocessor in the array may be configured into a $Y \sim Y$ complementary module pair. The resulting filter will be extremely flexible and fast. This approach is attractive as semiconductor memories for microprocessors are getting larger in capacity and faster in access time.

We also believe that the concept of the digit-convolution module is useful as a unit of hardware complexity and as a means to measure the comparative usefulness between different digital filter implementations. A theoretical study on this should result in a convenient analytical tool.

Finally, as a specialist's project, the attractive implementation of modulo $2^N$ multipliers using adder-pairs should be developed further, especially to discover whether simple algorithms exist for the necessary coding and decoding.
11.3 Conclusions.

As, at the moment, there is a tremendous activity in the search for good hardware structures for practical digital filters, it will not be long before real-time digital filters will be as common and as easy to build as active filters are today, with the added extra features that are only possible with digital processing.

If the author's findings are seen to contribute in a modest way towards that objective, it will more than recompense the effort that has gone into the research reported in this Thesis.
REFERENCES


54. Fourier Analyzer Training Manual, Application Note 140-0, (Hewlett-Packard Co.).


