THE UNIVERSITY OF CALGARY

ARCHITECTURAL DESIGN OF A COMPUTING ELEMENT FOR SIGNAL PROCESSING

by

HAKAN ORBAY

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL ENGINEERING

CALGARY, ALBERTA

April, 1986

ISBN 0-315-32730-8

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "*Architectural Design of a Computing Element for Signal Processing*" submitted by Hakan Orbay in partial fulfillment of the requirements for the degree of Master of Science.

Supervisor, Dr. M.R. Smith
Dept. of Electrical Engineering

Dr. S.T. Nichols
Dept. of Electrical Engineering

Dr. G.S. Hope
Dept. of Electrical Engineering

Prof. H.D. Baecker
Dept. of Computer Science

Date 16 May 1986

ABSTRACT

The long term objective of this research is to realize a multi-stage digital signal processor. Each stage of the system consists of a processing unit designated to run a specific digital signal processing (DSP) algorithm. A uniform architecture suitable for various DSP algorithms is therefore required.

In this thesis, the development of such an architecture is discussed, with emphasis on parallel processing and resource optimization. It was shown that the separation of the address generation and the arithmetic operations leads to a structured organization of the processing unit. This organization remains essentially unchanged for various algorithms, with the exception of the address generator which is unique to an algorithm. The architecture of the arithmetic unit was optimized for signal processing operations.

A microprogrammable control unit was necessary in order to combine high performance and flexibility in the processing units. For this reason, a general purpose microprogrammable controller was constructed as a tool for the development of each processing unit. A software library was also provided on the host computer to maintain simple operation of the controller.

A 16-bit FFT processor was built to demonstrate the efficiency of the architecture and the usefulness of the general purpose controller. The parallel architecture allowed the completion of one FFT butterfly every four cycles with a

cycle time of 125 ns. The results indicated that, using this architecture, the real-time implementation of the signal processor is feasible.

ACKNOWLEDGEMENTS

I am indebted to my supervisor, Dr. M.R. Smith, for his guidence through the course of this work and for invaluable advice and patience during the preparation of this manuscript. Thanks are also due to Dr. G.S. Hope for his interest and support, to Warren Flaman for his most appreciated assistance in the construction of the printed-circuit boards and to Norman Bartley for many helpful discussions.

The financial assistance of Advanced Micro Devices Ltd., Ontario in supplying various components, the University of Calgary in awarding Graduate Assistanships and the Natural Science and Engineering Research Council of Canada in providing operating funds are gratefully acknowledged.

Finally, I would like to thank two special people, Monica and Ekrem. Their constant encouragement has made this thesis possible.

To My Mother and Father

Anneme ve Babama

TABLE OF CONTENTS

Table of Contents vii
List of Tables x
List of Figures xi

1. INTRODUCTION 1
1.1 Long-term Goals 4
1.1.1 The problem 4
1.1.2 Use of modeling 5
1.1.3 Implementation 7
1.2 Purpose and Objectives 8
1.3 Outline 8

2. MICROPROGRAMMING 10
2.0 Introduction 10
2.1 Generation of Control Information 10
2.2 Why Microprogramming? 12
2.2.1 Advantages of Microprogramming 12
2.2.2 Disadvantages of the Microprogram 14
2.2.3 Economical Considerations 14
2.3 Evolution of Microprogramming 15
2.4 Components of a Microprogrammable Control Unit 16
2.4.1 The Control Store 17
2.4.2 The Sequencer 18
2.4.3 The Condition Code Generator 20
2.4.4 The Pipeline Register(s) 21
2.4.5 The Clock Pulse Generator 22
2.5 Summary 23

3. GENERAL PURPOSE CONTROLLER 24
3.2 Organization of the General Purpose Controller 30
3.2.1 Design Considerations 31
3.2.2 The Architecture 32

3.2.3 The System Clock 35
3.2.4 Specifications of the Prototype Controller 37
3.3 Loading the WCS 39
3.3.1 Downloading Procedure 40
3.4 The Task Manager 42
3.4.1 Implementation of the Task Manager 43
3.4.2 Downloading The WCSs 45
3.4.3 System Development Support 46
3.4.4 Host Communications 47
3.4.5 Other Features 47
3.5 Software Support 48
3.6 Summary 50

4. AN ARCHITECTURE FOR SIGNAL PROCESSING 53
4.1 Processing Requirements 54
4.1.1 Basic Operations of DSP 55
4.1.2 The Benchmark Algorithm 58
4.2 Functional Requirements 59
4.2.1 Parallelism and Pipelining 60
4.2.2 Functional Blocks of the Processing Unit 63
4.3 The Arithmetic Processor 66
4.3.1 Optimization of Resources 67
4.3.2 Component Technology 72
4.3.3 Data Format 73
4.3.4 Data Flow Structure 74
4.4 Architecture of the Processing Unit 76
4.4.1 The Memory 76
4.4.2 The Scaler 78
4.4.3 I/O Interface 78
4.4.3.1 The Metastable Problem 82
4.5 An Example: The FFT Processor 85
4.5.1 Characterization of the Processing Unit 85
4.5.2 Overflow Protection in FFT 88
4.5.2.1 Word Growth in a Butterfly Operation 89
4.5.3 Software for Parallel Execution 93
4.5.4 Clock Requirements 97
4.6 Summary 98

5. CONCLUSIONS 100
5.1 The General Purpose Controller 100
5.2 The Digital Signal Processing Architecture 101
5.3 Recommendations for Future Research 103

REFERENCES 105

LIST OF TABLES

3.1. Calculation of the delay for the CJMP instruction 36
3.2. Calculation of the delay for the CONT instruction 36
4.1. Effects of the multiplicity of the resources 69
4.2. Progress of a butterfly operation 94
4.3. Modified butterfly for overlapping 95
4.4. Two overlapped butterfly operations 95
4.5. Completely overlapped butterflies 96
4.6. Calculation of the delay for the READ operation 98

LIST OF FIGURES

1.1. A signal processing system 6
2.1. Fundamental blocks of a digital machine 11
2.2. Organization of a typical sequencer 19
2.3. The microinstruction register 21
3.1. Flow of control information 24
3.2. Instruction based Structure 27
3.3. Address based Structure 27
3.4. Two-level pipeline based structure 28
3.5. One-level pipeline based structure 28
3.6. Architecture of the general purpose controller 33
3.7. Typical WCS organization 39
3.8. Bank organization within the WCS 42
3.9. The task manager 44
3.10. Sequencer and WCS board 51
3.11. Task manager board 51
3.12. WCS expansion board 52
3.13. General purpose controller 52
4.1. A perspective of designing digital signal processors 54
4.2. A pipelined parallel processing system 61
4.3. Parallel processing in different levels 62
4.4. General organization of the processing unit 67
4.5. Data flow in the arithmetic processor 75
4.6. An example of the simplified multiplier inputs 76
4.7. The architecture of the processing unit 78
4.8. The data memory 78
4.9. The scaler block 79
4.10. I/O Interface 80
4.11. Interconnection of the processing units with a system bus 81
4.12. Timing diagram of execution and data transfers 81
4.13. Timing diagram with different execution speeds 81
4.14. The metastable problem 82
4.15. Metastable problem in the control unit 84

4.16. Rectification of the metastable problem 84
4.17. The FFT processor 86
4.18. FFT processor board 99

CHAPTER 1

INTRODUCTION

In the last two decades, digital signal processing has become an increasingly important tool for science and technology as its applications spread to such diverse fields as geophysics, biomedical engineering, nuclear science and communications. Originally, signal processing has been conducted using analog equipment. With the need to process large volumes of data and the advances in the computer technology, attention has shifted to processing in the digital domain. The flexibility of the digital computers fostered experimentation with progressively more sophisticated algorithms, and as a result, new digital techniques have been developed without any apparent analog implementation. Thus, the digital signal processing (DSP) theory, which was originally regarded as an approximation to the analog signal theory, has evolved into a field of its own.

Signal processing, in general, is used to extract certain characteristics of a signal or to transform the signal into a more desirable form. For example, in EKG and EEG analysis or in speech recognition some characteristic parameters of the signal are estimated. Alternatively, noise or other interference may be removed from the signal or the signal may be modified to present it in a form which is more interpretable. Further examples of the application fields of digital signal processing are listed below.

Geophysics: Deconvolution methods are used in analyzing seismic data to aid in modeling the structure of the earth's interior, the study of earthquakes and exploration for oil.

Communications: A signal transmitted over a communication channel may be perturbed in a variety of ways, including channel distortion, fading and insertion of background noise. One of the objectives at the receiver end is to compensate for these disturbances. For example, signal processing may be used for filtering signals to remove out-of-band components, echo cancellation in voice/data channels, detection and purification of satellite signals.

Speech processing: The main problems here are speech recognition, voice identification and verification, speech waveform parametrization, encoding and compression. Spectral analysis and signal modeling techniques are used to aid in recognition of the type of the signal or to characterize the voice signal.

Sonar: High resolution spectral analysis is an important tool in various sonar activities such as generation of signal pulses or detection and analysis of echo returns. Sonar has its applications in target detection and localization, navigation and mapping.

These applications are just a sample of the multitude of the fields where DSP techniques are utilized. An excellent review of the application areas of digital signal processing is edited by Oppenheim [1].

This growing interest in digital processing has created a huge area of research in the implementation of digital signal processors, both in software and hardware. Classically, the realizations of the signal processing systems have tended towards one of two general approachs depending on the performance requirements. Inflexible, dedicated hardware systems were used where high performance was the priority (e.g. [2, 3]). Where a lower performance was tolerable, software implementations on general purpose computers were sought (e.g. [4]). Over the past decade, attempts to combine flexibility and high performance resulted in two new approachs, implementation on the supercomputers and on specialized array processors attached to a general purpose host computer. Recently, advances in the integrated-circuit technology have given rise to new high-performance components which have dramatically changed the cost/performance criteria for all digital systems. In particular, a range of programmable, configurable chips opened up a new dimension in digital signal processor implementations.

In this thesis the possibility of efficient utilization of these new components in a DSP suitable architecture is investigated. In effect, this thesis completes the first leg of a project which requires the hardware implementations of several DSP algorithms. Thus, the purpose and the scope of this thesis is closely related to the objectives of the parent project. For this reason, a brief discussion of the parent project and its goals is given before concentrating on the scope of this report within these goals.

1.1. Long-term Goals

The long term objective of this research is to design a signal detection system which implements a modeling algorithm. In this section, the signal detection problem and its possible solutions are briefly presented.

1.1.1. The problem

This technique is directed towards the well-known problem of restoring a signal $x(t)$ after it has been passed through a known transformation $h(t)$ and has been obscured by noise $n(t)$. The observed signal $y(t)$ has the form

$$y(t) = \int_{-\infty}^{\infty} x(\lambda)h(t-\lambda)d\lambda + n(t) \qquad ..(1.1)$$

Taking the Fourier transform of the above equation yields

$$Y[f] = X[f]H[f] + N[f] \qquad ..(1.2)$$

Since N[f] is indeterminable, only an approximation of X[f] can be calculated by the deconvolution of Y[f]:

$$\tilde{X}[f] = \frac{Y[f]}{H[f]} = X[f] + \frac{N[f]}{H[f]} \qquad ..(1.3)$$

Taking the inverse Fourier transform of $\tilde{X}[f]$ yields $\tilde{x}(t)$, an estimation of the input signal x(t).

The numerical solutions for eqn. (1.3) have been extensively analyzed. The main problem is the error introduced by the term N[f]/H[f] which is more significant at the higher frequencies. The results can be improved by applying a window in the

frequency domain to increase the roll-off of $\tilde{X}[f]$, but at the cost of decreasing the resolution of $\tilde{x}(t)$. An alternative solution was sought that did not compromise the obtainable resolution.

1.1.2. Use of Modeling

After removing the noise contaminated high-frequency components, modeling can be applied in the frequency domain to extrapolate the remaining spectrum. Let's assume that the spectrum $\tilde{X}$[f] is known at the intervals of Δf. $\tilde{X}$[f] can be represented by the series

$$(\tilde{X}_0, \tilde{X}_1, \ldots, \tilde{X}_{n-1}) \qquad ..(1.4)$$

where $\tilde{X}_m$ denotes the spectral component at the frequency $m\Delta f$. Now assume that there is an arbitrary frequency $k\Delta f$ where $\tilde{X}_m, m < k$ are not significantly contaminated by the noise. Each one of these spectral components can be approximated by a weighted sum of the previous p samples of the spectrum,

$$\hat{X}_m = -\sum_{j=1}^{p} a_j \tilde{X}_{m-j} \qquad m < k \qquad ..(1.5)$$

This equation is known as prediction equation. The error between the actual component and the predicted value $\hat{X}_m$ is given by

$$e = \tilde{X}_m - \hat{X}_m \qquad ..(1.6)$$

This error can be minimized across the data sequence $\tilde{X}_m$, $0 \le m < k$ in order to determine the prediction coefficients a_j. This prediction technique is known as auto-regressive (AR) modeling, and p is referred to as the model order [5].

Once the prediction coefficients have been determined, The series $(\tilde{X}_0, \tilde{X}_1, \ldots, \tilde{X}_{k-1})$ can be predicted or extrapolated to an arbitrary number of points to achieve the series

$$\hat{X}[f] \equiv (\hat{X}_0, \hat{X}_1, \ldots, \hat{X}_{n-1}, \ldots) \qquad ..(1.7)$$

It has been suggested that the spectrum $\hat{X}$[f] obtained by the modeling technique presents a better approximation of X[f] than $\tilde{X}$[f], especially when the inverse Fourier transform is applied after deconvolution. The block diagram shown in fig. 1.1 represents this algorithm.

The technique described above has been improved by using the *transient error method* [6]. The modified technique is a deterministic auto-regressive moving-average (ARMA) model that produces better results than the simple AR model.

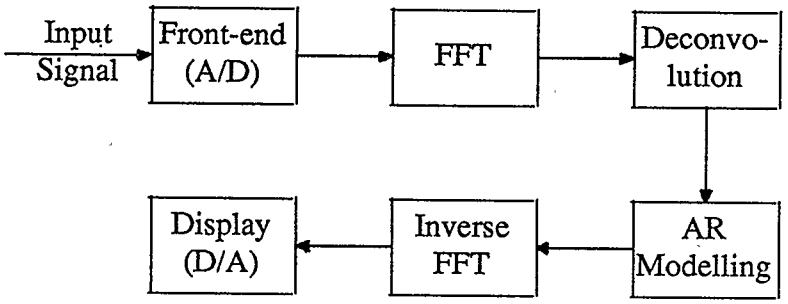


Fig. 1.1. A signal processing system

1.1.3. Implementation

In a real time application, the number of computations involved in the algorithm described above is too great to be handled by a single processor. The block diagram suggests a modularization where a processor is assigned to each block. Since each of these processors will be performing a specific job, the efficiency and the overall speed can be improved immensely by designing dedicated processors instead of using general purpose microprocessors.

These processors will execute the well-known algorithms for the forward or inverse fast Fourier transform (FFT), Burg's or Levinson's AR modeling, and deconvolution. It is observed that the data flow between the blocks can be organized so that the overall system becomes a general purpose spectrum analyzer. For example, a straight FFT can be calculated by bypassing most of the system, or one may bypass the FFT and use the modeling algorithm and last FFT block (the difference between forward and reverse FFT is trivial) to estimate the spectrum of the input signal. In the latter case, the last block could be modified to perform an interpolating FFT algorithm [7] in order to reduce the processing time.

In general, the DSP technique specifies an order of operations, such as FFT, AR modeling, etc., similar to fig. 1.1. The actual implementation of a particular technique is irrelevant to the scope of this thesis and for this reason the processing model of fig. 1.1 is adopted as an example of the implementation of a DSP algorithm. This model is referred to as the (multi-stage) signal processing system and each stage is called a processing unit.

1.2. Purpose and Objectives

The purpose of this study was to design a high performance array processing architecture particularly targeted for DSP algorithms. The feasibility of a real-time signal processor employing microprogrammed control was investigated. The results of this thesis are intended to be passed on to further research to complete the design of the processing system.

It was the objective of this thesis to build the FFT processor block as a demonstration of the proposed architecture. The control unit of this processor was built as a general purpose microprogrammable controller which could be used to control any of the processing units. It was also within the scope of this thesis to provide a development and support system for the general purpose controller. The development system and the controller have been brought to a stage that future researchers who continue to complete this project will be able to utilize these units to design and debug the implementation of other processing units within a user-friendly environment [8].

1.3. Outline

This thesis can be examined in two sections, the design of the general purpose controller and the development of a DSP architecture. Due to the variety of the material discussed in this thesis, background information about the concepts used are given when necessary, rather than grouping all this information together. However, we felt that a detailed introduction to the concept of microprogramming was

necessary. Chapter 2 provides a historical introduction to microprogramming. The advantages and disadvantages of microprogramming are summarized and the general design of the components of a microprogrammed control unit are also discussed in this chapter.

Chapter 3 deals with the construction of the general purpose controller. Common organizations of the control unit are summarized and the architecture of the controller is discussed. Then the motivation to design a support system for the controller is presented, followed by the description of the support system developed. Finally, the contents of the software library are briefly listed.

In chapter 4, an architecture for the processing units of the signal processing system is proposed. The computational and the functional requirements of a processing unit are discussed, arriving at an optimal architecture. Then the components of this architecture are defined. In the final part of this chapter, the FFT processor constructed with the proposed architecture is described.

The conclusions and suggestions for further research are presented in chapter 5.

CHAPTER 2

MICROPROGRAMMING

2.0. Introduction

As mentioned earlier, the implementation of the various processors will be based on specialized configurable components which require complex control signal sequences. Two basic approaches to generate the required control information are presented and compared in this chapter. Microprogramming, the preferred approach, is examined in detail.

The concept of microprogramming was first introduced by Wilkes in 1951[9] as an alternative method of control unit design. Since then, microprogramming gained a great deal of significance as it became a powerful tool in the hands of designers and users alike. This technique provides a highly systematical approach to design, resulting in virtually unlimited flexibility in controller applications.

2.1. Generation of Control Information

A digital machine can be represented by four basic functional units: processing unit (CPU), storage, interface and control unit as illustrated in fig. 2.1. It is evident that the control unit is responsible for the sequencing and timing of all the hardware activity within the system. Therefore, the control unit is the section where commands (instructions) are interpreted and performed by causing the execution of a

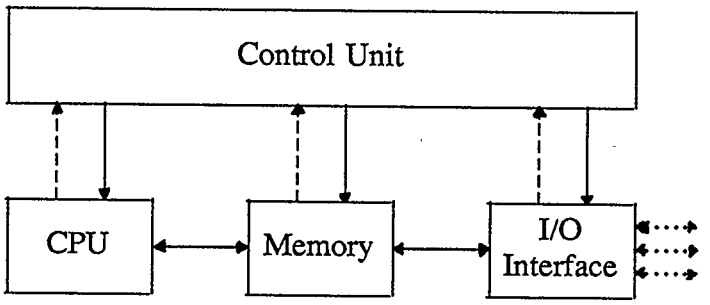


Fig. 2.1. Fundamental blocks of a digital machine

series of primitive operations such as register-to-register transfers, selection of the arithmetic-logic unit (ALU) functions, etc.

The conventional method of implementing the control unit is by designing a sequential logic network. The commands are fetched from the storage unit (memory) and converted into control information which activates the discrete logic circuit which in turn activates a series of primitive operations that constitute the command. This approach is called *hardwired* or *conventional control.* It follows that with this approach the control unit becomes the most complicated part of the digital machine. Once the design is completed implementing a certain set of commands (machine instructions), it requires a non-trivial effort to alter or enhance this set.

Microprogramming, an alternative method to conventional control, can reduce the complexity and the inflexibility of the control unit. Wilkes [10], who introduced the term microprogramming, conceived its objective as

> to provide a systematic alternative to the usual somewhat *ad hoc* procedure used for designing the control system of a digital computer.

This method is based on the observation that a complex operation such as a machine instruction can be completely specified by a series of primitive operations. The control information for these primitive operations can be directly stored in a memory element and consequently each complex operation becomes a sequence of references to the memory element. This representation of the complex operation is called a *microprogram* (or *microroutine*) and the memory unit is called the *control storage* (or *microprogram memory*). Accordingly, the primitive operations stored in the microprogram memory are referred to as *microoperation*s (or *microinstruction*s). The terms in parenthesis are alternate descriptions and in this thesis, they are used interchangeably with the preceding terms.

2.2. Why Microprogramming?

2.2.1. Advantages of Microprogramming

1. *Systematic Approach*

This is the most striking characteristic of microprogramming over conventional control. The latter approach results in a random structure limited by the designer(s) ingenuity. Microprogramming can cut the development time drastically as the random sequential logic is replaced by a pseudo-structured microprogram.

2. *Architectural Changeability.*

The characteristics of a machine such as instruction set, word size or bus width can be altered through microprogramming, arriving at totally different architectures from the same hardware resources. The significance of this feature was recognized during the early years of microprogramming as it provided a direct method of using the old software in newer machines without any modifications. This compatibility is achieved by providing a set of microprograms for the new machine which interprets and executes the older system's instruction set. Such alternate sets of microprograms which mimic another system are called *emulators.*

3. *Flexibility*

This advantage follows from the previous one but it needs to be reemphasized. During the design stage of the system, the instruction set does not need to be fixed. It can be altered and realtered to fit as specifications of the system change. In other words, experimentation with various sets in order to find the optimum instruction set is a possibility while the hardware is still in development stage. Experimentation on the user's part enables him to tailor a system to the requirements of a specific application which is described by the term *adaptability.*

4. *Diagnosability*

With microprogramming it is possible to locate the errors in the hardware much more precisely than is possible otherwise. Machine self-tests are much easier to implement and more versatile than when using conventional control. Thus diagnosis and maintenance of the hardware becomes easier and more reliable.

2.2.2. Disadvantages of Microprogramming

1. *Speed*

It is always possible to design a hardwired control which runs faster than the microprogrammed one, even with today's high speed bipolar memories. Essentially this was the reason that microprogramming did not gain wider attention during the years following its introduction..

2. *Lack of Support Systems*

Since each microprogrammed system is essentially different, it is very difficult to design a general purpose development system for microprogramming applications. Most of the sophisticated equipments on the market today use a wide set of parameters which have to be defined by the user to obtain a meaningful assembly language. Many designers are forced to prepare their own development systems and assemblers to debug a microprogramming project. In fact, this problem was encountered during this research and designing a support system became necessary. The measures we have taken will be explained later.

2.2.3. Economical Considerations

The economical feasibility of microprogrammed implementations depends on the size and the complexity of a machine. It has been established that microprogramming costs less than the conventional control except in very simple and dedicated systems [11].

2.3. Evolution of Microprogramming

Although microprogramming received some attention during the 1950s, it was not used or researched on a significant scale until the mid 1960s. The reasons for this delay can be found in the memory technology of the time period. First, the speed of memory access was significantly lower than logic speeds, and second, memory elements were very expensive. The simplicity and the flexibility offered by microprogramming using the technology of the era was more than offset by the cost and the time overhead of the memory access for each microinstruction.

With the technological breakthroughs during the 1960s, microprogramming finally became cost-effective. The first microprogrammed machines were introduced in this time period. The IBM 360 series was a milestone in the acceptance of the microprogramming. Most of the computers in this series were microprogrammed in order to achieve instruction set compatibility between machines of different capabilities and hardware organizations. Larger models of this series were hardwired for reasons of speed.

Although these machines and others of this time period were true microprogrammed machines, they did not offer the advantages of user-microprogrammability. Two basic reasons for this lack were:

1. The cost of fast random access memories (RAMs) was relatively high until 1970s.
2. Manufacturers were reluctant to let users tamper with the architecture of the system because of the effect this might have on the reliability of the machine.

The full power of microprogramming came into realization with the introduction of fast, relatively cheap RAMs. Using RAMs as the control storage enables the user to easily modify an existing instruction set in order to adapt the processor to a specific application. This type of microprogram memory is called *writable control storage (WCS)*.

Finally in the mid 1970s, several manufacturers marketed microprogrammable processors or *bit-slice* processors. With these off-the-shelf processors, it has become possible for the users to design microprogrammable processors for various applications.

Today, besides implementing an instruction set for a general purpose computer, microprogramming is also used in dedicated machines running without a software instruction set. Microprogrammed instruction sets are getting progressively more complicated. This has caused a reversed trend towards conventional control. The feasibility range of the conventional control have been increased by the introduction of the reduced instruction set computers (RISC). On the other hand, microprogramming is still an indispensable tool for the designers who struggle to achieve a degree of parallelism in processor applications.

2.4. Components of a Microprogrammable Control Unit

A general microprogrammed control unit consists of five functional blocks.

1. The Control Storage
2. The Sequencer

3. The Pipeline Register(s)

4. The Condition Code Generator

5. The Clock Pulse Generator

In this section, the functions of these blocks are examined with particular emphasis on the enhancment they provide to the efficiency of the overall controller.

2.4.1. The Control Store

As mentioned before, this is the characteristic block of any microprogrammed machine. The most common and simple form of the control store (CS) is the ordinary memory array in which there is one microinstruction per CS word. Some of the variations to this structure are:

a) There may be two microinstruction for each CS word. In this structure, the output of the CS is written to two pipeline registers simultaneously, reducing the number of memory references. Effectively this scheme halves the access time of the control store, a limiting factor of the overall speed of the digital machine.

b) The control store may be divided into sections, which are called pages. Only one section is accessible at a time and the selection of this page is normally done in hardware. A page contains only a fraction of the total memory locations thus requiring fewer address lines at the expense of extra hardware to enable switching between pages. Therefore, if jumps between pages are not frequent, this organization reduces the access time and the number of the bits necessary to specify a destination microinstruction within a page.

c) The control store may be formed with a two level structure. The upper level of the control store is a narrow memory and it is used to address the lower level control store which is a wide memory unit containing a list of common microinstructions. The lower level control store is commonly referred to as the *nanocontrol store,* and accordingly programming the lower level is called nanoprogramming. This structure reduces the total number of bits required for the control store. An example of a machine employing such a control store organization is the 68000 microprocessor.

2.4.2. The Sequencer

It was suggested that one form of controlling the microprogram flow is to append the address of the next instruction to the current one, creating a linked list of microinstructions [12]. In this structure, the sequencer is totally eliminated or replaced by a register and/or a combinational logic circuit to provide conditional branches. However, it has been established that this method is merely a form of implementing sequential logic networks [13]. A linked list of microinstructions displays some of the poor properties of a sequential logic circuit. In particular, it is not trivial to change a branch instruction. Since microprogramming is meant as an alternative to sequential logic, this structure should not be discussed as a form of microprogramming. Therefore, a true microprogrammable control unit should include a form of address generation logic, i.e. a sequencer.

The basic elements of a sequencer include an address register, stack and address multiplexer, as illustrated in fig. 2.2. In the configuration shown, the incrementer generates the sequential address and it is latched to the address register which behaves as a program counter. Absolute jumps are performed by transferring input data directly to the sequencer's output. Subroutines in the microprogram are facilitated by the stack which consists of a register file and a stack pointer. The address selected by multiplexer depends on the instruction and condition code (CC). Therefore, the sequencer provides the microprogrammer a means of structured

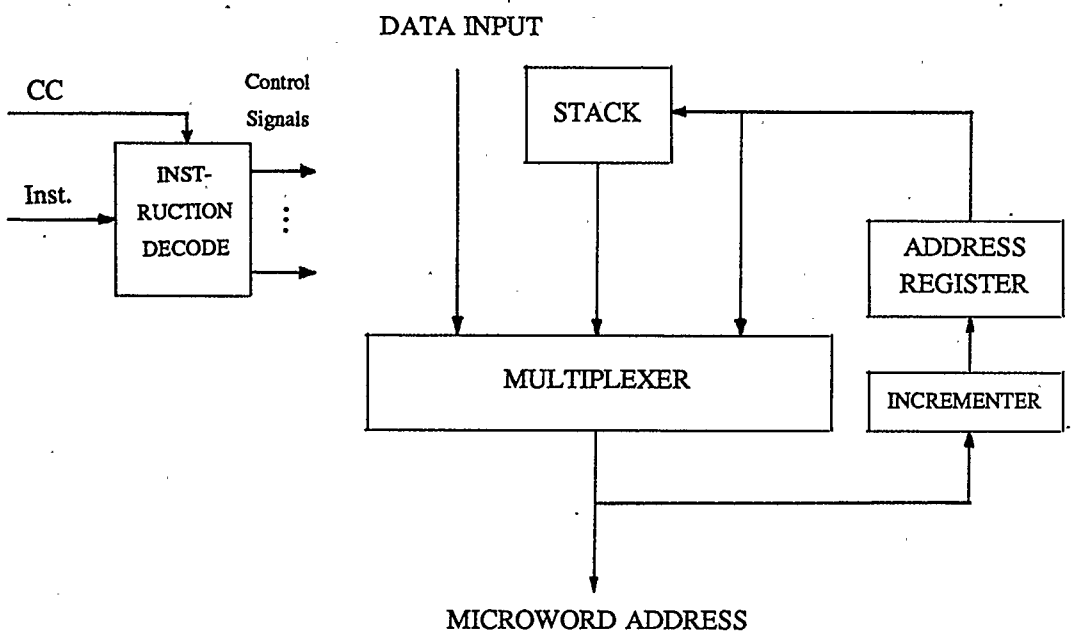


Fig. 2.2. Organization of a typical sequencer

control flow within the microprogram. Loops, subroutines, decision structures can be implemented with simple encoded microinstructions, tremendously enchancing the power of microprogramming.

Currently available sequencers provide some additional features for address generation such as an increment-by-two (SKIP) instruction or a counter/register that provides yet another source of address. A detailed discussion of variations of the basic organization (fig. 2.2) in commercially available sequencers is given by Andrews [14].

2.4.3. The Condition Code Generator

Although, some microprogrammed systems integrate the condition code generation into the sequencer, generally the condition code supplying logic is regarded as a supporting element for the sequencer and not necessarily a part of it. In its simplest form a multiplexer selects one of the several status signals generated by the system to control the conditional instructions of the sequencer. In order to have absolute power over looping instructions in the sequencer a polarity control is usually included. This device generates the correct level of condition code as required by the sequencer and eliminates the need to have all signals active high or all active low.

Several dedicated condition code generators, e.g. Am2904, are available for handling more complicated requirements such as storing the status information for future references or testing several status signals in one cycle.

2.4.4. The Pipeline Register(s)

In most microprogrammed systems, a register is placed between the outputs of the control store and the control points of the processing unit. Such a register, shown in fig. 2.3, serves several purposes. Most importantly it breaks the continuous loop between the outputs of the control store and the instruction input of the sequencer which avoids race conditions. Its other function is to hold a stable microinstruction throughout the cycle time. This register is called *microinstruction register* (MIR) or *pipeline register*. The name "pipeline" derives from the fact that

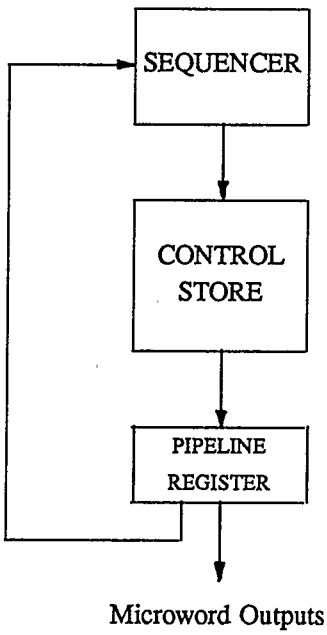


Fig. 2.3. The microinstruction register

this register actually divides the control path into two parts, isolating the sequencer and the CS from the processing unit. Thus pipelining the control store outputs decreases the minimum required clock period, improving overall speed. In this report, the more general name *pipeline register* is used for consistency.

Pipelining is not limited to the outputs of the control store, and it can be used wherever a long path is limiting the speed. However, the increased use of pipeline registers makes the programming more complicated. Different control unit organizations can be arrived by employing pipeline registers on different paths.

2.4.5. The Clock Pulse Generator

The clock pulse generator is an intrinsic part of any digital state machine. In microprogramming, unlike the sequential logic circuits, clock requirements are usually very simple. This is due to the fact that every part of the system is synchronized with the appearance of the data on the pipeline register. Usually a sequential circuit demands several phases of a clock pulse which further complicates the circuit. Normally, when converting a sequential circuit to the microprogrammed equivalent each phase becomes a clock cycle. One other advantage of the microprogramming is that it provides an easy way to control the clock period dynamically. This boosts the overall throughput if the execution of some instructions require longer time for completion than others.

2.5. Summary

The microprogramming method is superior to the hardwired control for generating complex control signal sequences, such as required by most signal processing systems. Although a microprogrammed control unit can be realized in a variety of ways, five basic blocks can always be identified. These blocks are the control store, the sequencer, the condition code generator, the pipeline registers and the clock generator.

In effect, this chapter provides an introduction to the next chapter, where several different organizations of the five basic blocks will be presented, and the design of the general microprogrammable control unit will be described.

CHAPTER 3

GENERAL PURPOSE CONTROLLER

3.0. Introduction

This chapter is dedicated to the design and implementation of a general purpose controller. The generality is inherent in a microprogrammable control unit, especially when it is viewed as a module which can generate any sequence of control signals at its output. This view is emphasized in fig. 3.1, which illustrates the flow of control information in a digital machine. The *target system* represents the unit which utilizes the outputs of the control unit. The control unit is shown as a "black box" where the outputs, the microword, are related to the inputs, the status information from the target system or an external source. The relationship, or the transfer function of the box, is fixed in a hardwired control unit, hence the operations

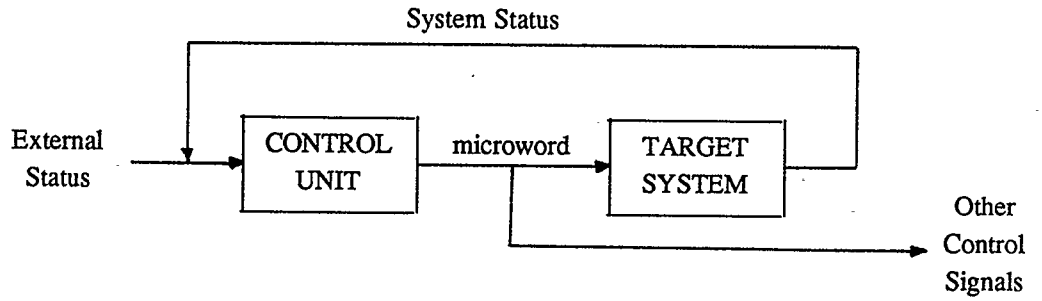


Fig. 3.1. Flow of control information

of the target system are also specified. In the case of a microprogrammable control unit, the transfer function is completely defined by the microprogram and it can be modified to accommodate any processing unit.

This representation of the control unit is particularly interesting when the multi stage signal processor is considered. Fig. 3.1 suggests that a processor can be designed as two individual modules, the control unit and the processing unit (target system). Moreover, once a general purpose controller is realized, it can be used during the development of each stage of the signal processing system. After the development of the individual processors, the controller can be duplicated to provide each processor with its own control unit.

The first part of this chapter deals with the design of the controller. Various structures are considered and the implementation of a suitable structure is described. In order to maintain the ease of use of the general purpose controller, a separate support system to load the control store is necessary. The design of the downloading unit and the software support package for the controller are discussed in the second half of the chapter.

3.1. Organization of a Control Unit

The data flow paths within the control unit can be broken into two sections by inserting a pipeline register to the path. The pipeline registers also cause one cycle delays between the two sections of the path. These delays are very important because different programming structures can be achieved by altering their location

and number.

Fig. 3.2 illustrates a basic architecture where only one register at the output of the control store is used. This pipeline register (PL) holds the microinstruction during its execution. The output of the sequencer, or the address issued to the control store, is shown as $A+n$, where n indicates the relative delay. Accordingly, the output of the WCS is $I(A+n)$, i.e. the instruction at location $A+n$. $S(A)$ denotes the status outputs of the target system in response to the instruction $I(A)$. This notation is used consistently in this section.

In this structure, the status information of the processing unit is immediately made available to the sequencer. Therefore, during the execution of a successful conditional branch instruction, the processing unit, the condition code multiplexer, the sequencer and the control store are all in series and the total delay is calculated by adding the delays in each unit. This path, a *critical* path, is usually responsible for determining the minimum possible clock period.

The organization shown in fig. 3.3 is a variation of the one described above. Since there still is one pipeline register, no improvement in speed is achieved but this scheme requires fewer register bits (typically 10-14) than the previous one (typically 40-120 bits) since only the address of the control store is registered. However, during the address setup period, the output of the control store, or the microinstruction, is unstable and it cannot be tied to sensitive control points. This fact usually more than offsets the slight advantage of the fewer register bits, rendering this organization impractical.

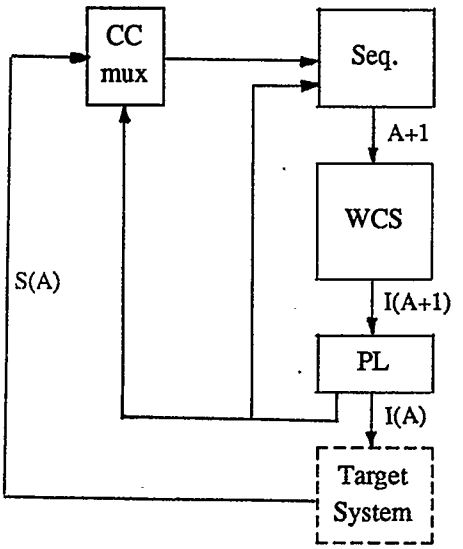


Fig. 3.2. Instruction based structure

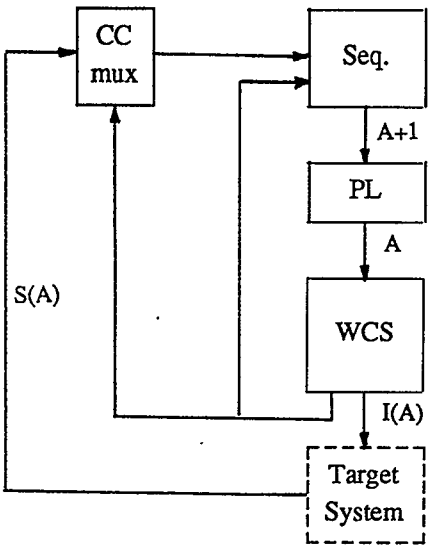


Fig. 3.3. Address based structure

Fig. 3.4 illustrates an architecture where the critical path in the previous architectures is replaced by three shorter paths. This structure isolates the delays in each device, thereby achieving maximum clock speed possible. However, programming with this structure is more complicated because the address of an instruction is generated two cycles before its execution. Unlike the previous architectures the addresses shown must be sequential for continuous execution. If branching occurs, the microinstruction pointed by the pipeline register (PL #1) cannot be executed, so the sequencer "freezes" for one cycle to discard this information. Therefore, if there are a significant number of branches in the

microprogram, the throughput of this structure may be less than that of the first structure even though the clock is faster.

At this point, the nomenclature needs to be clarified. The first two architectures are not called "pipelined", although there is a delay involved, because the function of this delay is to avoid the race conditions by breaking a continuous loop and no gain in speed is achieved. For the third organization, however, the pipelining is applied to divide a critical path into three parts, so it is classified as a two-level

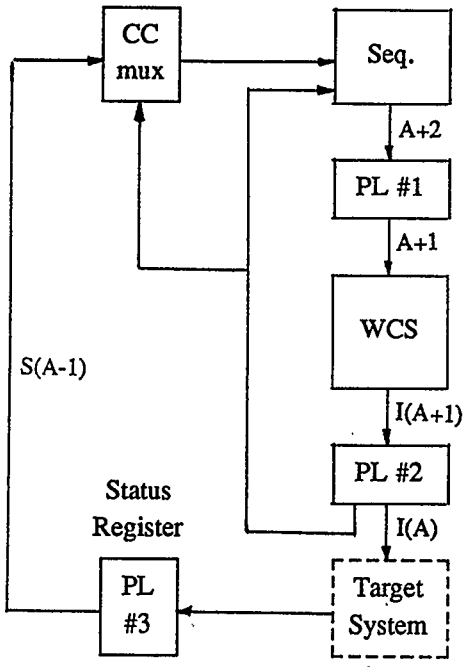


Fig. 3.4. Two-level pipeline based structure

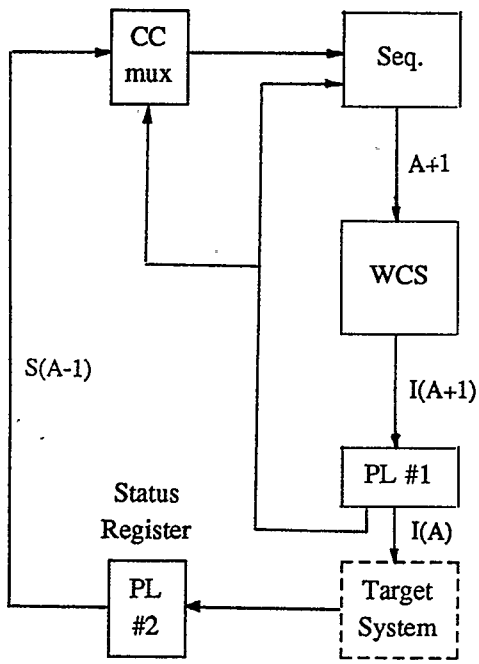


Fig. 3.5. One-level pipeline based structure

pipelined structure.

The architecture shown in fig. 3.5, an one-level pipelined organization, provides better speed and throughput than most others. It is not necessary to "freeze" the sequencer in this structure. However, unlike the first architecture, the conditional branch instruction has to be executed one cycle after the required status is generated. Therefore, while a conditional branch instruction is executed, the rest of the instruction may not be conditional. This may be a wasted part of the instruction in some cases, but it can be utilized for house-keeping functions within the processing unit.

To clarify the difference between these structures, consider the following pseudo-code for a program segment which adds two registers and performs a conditional operation based on the result.

R1 ← R1 + R2
if (CARRY is 1) then down-shift R1 by one

The microprogram for the first and the second structures will look like

```
 A    : R1 ← R1 + R2, if (not CARRY) jump A+2
A+1   : down-shift R1
A+2   : ...
```

For the third and fourth architectures the code becomes

```
 A    : R1 ← R1 + R2, continue
A+1   : if (not CARRY) jump A+3
A+2   : down-shift R1
A+3   : ...
```

Note that since the condition codes in the last two architectures are registered, the status 'CARRY' can not be examined in the same microinstruction that performs the addition. Although the microprogram appears identical for the third and the fourth architectures, there is a difference in the execution. In the third one, while the instruction at address A+1 is executed, the address A+2 is already issued to the control store. If the jump is to be performed, this address has to be overwritten by hardware measures and consequently there is an invisible one cycle delay. To summarize, assuming that the 'CARRY' condition is true, this program segment is executed in two cycles in the first and the second organizations, three cycles in the fourth and four cycles in the third organization. This example highlights the compromise made by the attempt to increase the clock speed by breaking the data flow paths.

3.2. Organization of the General Purpose Controller

In this section, the features that a controller must possess to maintain its efficiency over a range of applications are discussed. A general purpose controller has a set of requirements that are different from those of a control unit targeted for a specific processing system. These requirements should be considered before choosing the architecture for the general purpose controller among the alternatives described in the previous section.

The chosen architecture for the controller is described after the design requirements. The dynamic clock control technique is introduced and

implementation of this technique is discussed. Finally a summary of the features implemented in the controller is given.

3.2.1. Design Considerations

The most obvious characteristics of a general purpose controller are *flexibility* and *adaptability*. These terms are often used synonymously as a flexible system is usually adaptable and vice versa. By flexibility it is implied that the controller should be expandable and microprograms should be easily loaded and modified. A RAM based control store (WCS) is therefore necessary. On the other hand, an adaptable system can easily be modified in order to fit the special requirements of the applications. In order to realize adaptability, the target system should be provided with an access to the sequencing logic enabling the execution of the microinstructions addressed by the processing unit itself. This is particularly useful if an addressing scheme such as interrupt processing is to be implemented in the target system.

Another consideration is the general type of the microinstructions to be executed by the controller. Digital signal processing or other ''number-crunching'' algorithms are highly symmetric, usually consisting of several loops executed many times. This essential unsequentiality of the routines for which the controller is targeted implies that the sequencer must have a degree of *branch-efficiency*. The two-level pipeline based structure presented in the previous section is therefore ruled out since it limits the throughput while branching.

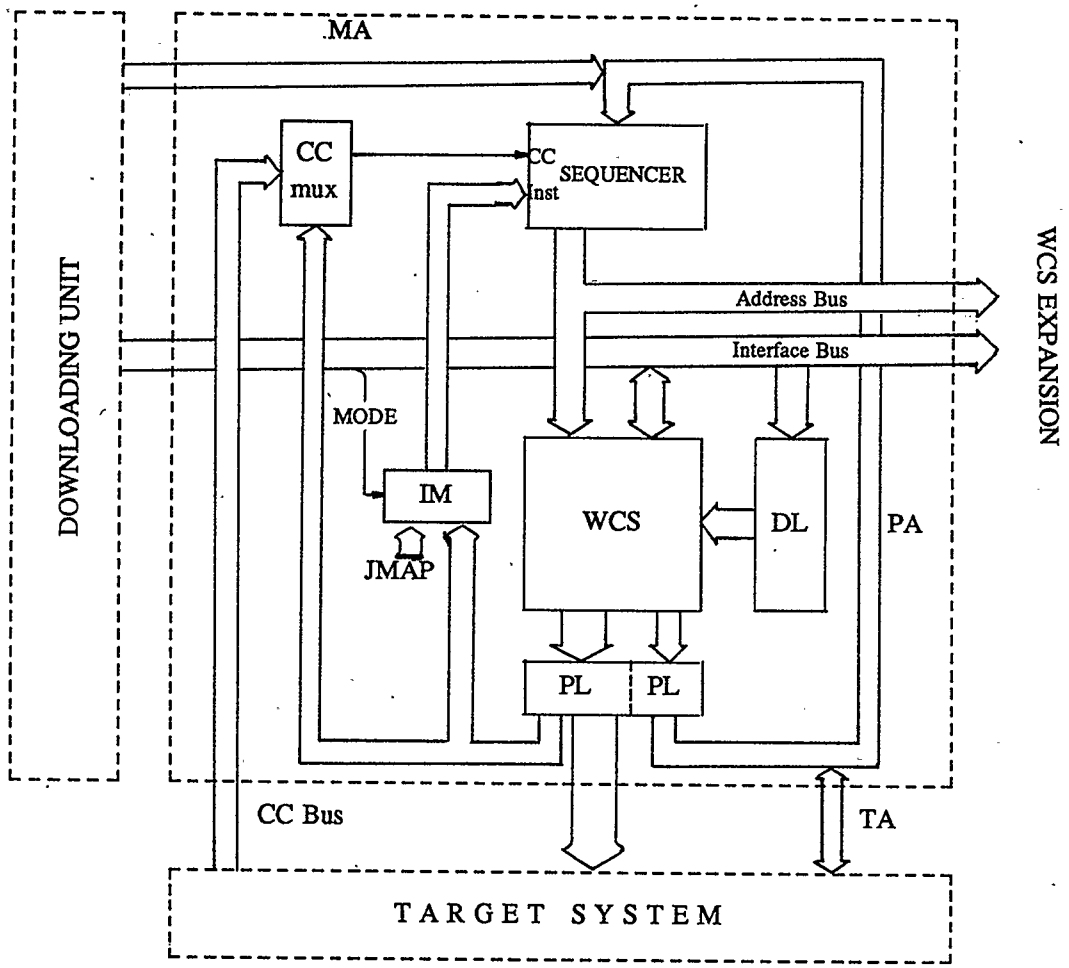
Since the general purpose controller is a separate unit, the microword outputs should be buffered before they are transmitted off the board to the target system. This fact, in addition to the output stability problem described previously, eliminates the address based architecture (fig. 3.3). Introduction of the output buffers cancel the only advantage of this structure i.e. requiring fewer register bits. *Bus-driving capability* is inherent in the other structures which employ a microinstruction register since outputs of most off-the-shelf register devices are buffered.

3.2.2. The Architecture

The block diagram of the architecture actually chosen for the general purpose controller is shown in fig. 3.6. This organization is an adaptation of the instruction based structure (fig. 3.2).

The controller was designed to have interfaces with two external systems: the target system and the downloading unit. The target system can utilize all of the microword outputs except the fields which control the sequencer and the condition code multiplexer. The status generated by the processing unit is transmitted to the controller via the CC bus. The target system can also specify the address of the next microinstruction or load parameters to the sequencer through the TA bus. This fulfills the adaptability requirement described previously.

The interface bus, the instruction multiplexer (IM), the downloading logic (DL) and the MA bus are all related to the downloading unit. The downloading unit interface is completely defined and discussed later in this chapter. The interface bus



PA, MA, TA Busses: Data sources of the sequencer
IM: Instruction multiplexer
DL: Downloading Logic

Fig. 3.6. Architecture of the general purpose controller

and the Address Bus (A bus) are extended to allow the expansion of the control store.

Now let us consider an example of the architectural adaptability of the controller apart from the TA bus. Comparing fig. 3.5 and fig. 3.6, it is clear that the one-level pipeline based architecture can also be realized by designing the status register as a part of the target system. Registering the status information (CC-bus) is particularly efficient for the ALU generated signals because of the long delay times involved with the ALU. In DSP algorithms, the ALU status signals, especially CARRY and overflow, are used only if the overflow prevention mechanism is incorporated in the software (microprogram). Other signals which are frequently examined to test the end of the loop conditions do not have significant delay times. Thus pipelining these signals is not necessary.

The key device in implementing this architecture was the Am2910A, the microprogram sequencer. This chip provides a compact sequencing logic completely fulfilling the requirements of the architecture. The Am2910A can receive the address information from three distinct sources which are reflected with the busses PA, MA and TA in fig. 3.6. In addition, the Am2910A also incorporates a 12-bit counter/register and a 9 word deep stack, complemented by a powerful instruction set [15].

The device chosen for implementing the WCS was the Am9150, a 1K by 4 bits random access memory chip. The distinctive characteristics of this unit are its fast access time (25 ns) and separate input and output ports. The isolation of the input and the output is particularly useful for tying the interface bus to the WCS since it is desirable to have low capacitance and loading effects on the WCS output bus.

Standard low power Schottky components were used to implement the rest of the controller in order to reduce the total power consumption.

3.2.3. The System Clock

Since the controller is to be used to control another system, it was not possible to completely define the clock specifications. For this reason the clock generator circuit was implemented to allow easy modification as required, permitting the user to tailor the system clock according to requirements of the particular application.

The overall clock requirements of a system are determined by whichever data flow path in the various processors has the longest delay. The delay on this path is equivalent to the minimum cycle time of the system, thus the path is referred to as the *critical path.* The usual method to find this path is to list the delay times on all potentially critical data paths and pick the longest one. Generally, the critical path is expected to be the one which utilizes most of the resources within one clock cycle. For example, let us consider the conditional branch instruction (CJMP) for the sequencer (Am2910A). The microword output selects the condition code which in turn selects one of the two possible addresses. The WCS must be accessed in the same cycle that sets up the pipeline register inputs. The estimation of the delay time is shown in Table 3.1. This delay represents the minimum clock period for any application of the controller unless the CJMP instruction is disallowed. Next, consider the very commonly used continue (CONT) instruction which increments the microprogram address by one. The delay for this instruction is 80 ns (Table 3.2),

DEVICE	PATH	PART NO.	DELAY
PL	Clock→Output	74LS374	20 ns
CC mux	Select→Output	74LS153	22 ns
Sequencer	CC→Output	Am2910	30 ns
WCS	Addr.→Output	Am9150-25	25 ns
TOTAL			97 ns

Table 3.1. Calculation of the delay for the CJMP instruction

which is significantly less than the estimated minimum clock period. The clock pulse period must be fixed to accommodate the longest delay, which limits the achievable performance level. On the other hand, it is not desirable to increase the overall system speed by disallowing certain instructions, as this would severely limit the flexibility.

The solution to this problem is the technique known as the *dynamic clock control.* In order the illustrate the improvement in the performance level consider a hypothetical system with two different instructions, *A* and *B*, with the frequency of usage 60% and 40%, respectively. Let us assume that instruction *A* requires 60 ns and *B* requires 100 ns. If the clock period is fixed, it has be 100 ns to accommodate

DEVICE	PATH	PART NO.	DELAY
PL	Clock→Output	74LS374	20 ns
Sequencer	Inst.→Output	Am2910	35 ns
WCS	Addr.→Output	Am9150-25	25 ns
TOTAL			80 ns

Table 3.2. Calculation of the delay for the CONT instruction

instruction *B*. However, if the clock rate is selectable during the execution to be either 60 or 100 ns, the average clock rate becomes

$$60 \times 0.6 + 100 \times 0.4 = 76ns$$

Therefore, simply employing a dynamic clock control technique improves the performance by approximately 25%.

Dynamic clock control was implemented on the controller using a microprogrammable clock pulse generator, the Am2925. This device is capable of dividing the frequency of its input (e.g. signal from a crystal) by a number between 3 and 10 as determined by 3 control inputs. For example, if the input oscillator has a cycle time of 15 ns, the output clock period can be selected to be 45, 60, 75, 90, 105, 120, 135 or 150 ns. In the actual implementation however, only one control input was selected by the microprogram and the other two were connected to hardware switches. This limitation is due to the fact that the 8 different selections of the clock period are somewhat redundant. Most of the time it is more convenient to classify the instructions simply as "fast" or "slow" rather than determining the exact delay for each instruction from the data books.

3.2.4. Specifications of the Prototype Controller

(1) The clock for the input of the Am2925 was generated by an external pulse generator with variable frequency. Therefore, the user has the complete control over the system speed through the external pulse generator, the hardware switches and the microprogram selected frequency division. This control is

important because the clock requirements of the target system are not known beforehand. In addition, the variable clock allows the user to experimentally determine or verify speed limits of the target system.

The system clock can be run either in the continuous mode or in the single pulse mode. Single pulse mode enables stepping through the microprograms, which is an indispensable tool for software development. Two external switches were provided to select the mode of the clock and to trigger the pulse in single pulse mode.

(2) The delay time from the clock to the pipeline outputs was 20 ns. This could easily be improved by replacing the low-power Schottky parts with high performance equivalents.

(3) The WCS provided with the sequencer is 1K deep with a word length of 64 bits. An expansion board containing 1K $\times$ 64 control storage is also provided, extending the microword width to 128 bits. Out of the 128 control outputs, 8 bits are reserved for the sequencer, the condition code (CC) multiplexer and the clock period control. An additional 12-bit field is tied to the input of the sequencer (PA). This field is also available to the target system (TA bus).

(4) The target system is expected to provide the condition code inputs, which are selected by a 4-to-1 multiplexer (CC mux).

3.3. Loading the WCS

In most machines with a writable control storage the microprogram memory is divided into the two conceptual sections shown in fig. 3.7. The loading of the RAM section (WCS) is done under the control of a microroutine stored in the ROM part. The usual method of transferring the microcode to WCS from the intermediate storage unit consists of treating the RAM module as an input/output device during loading. In most cases, particularly where the WCS is used only to provide a very fast medium for the microprogram memory, loading is done during the power-up initialization and then the ROM is disabled, allowing a faster system clock. Other machines may require interactive instructions to commence loading.

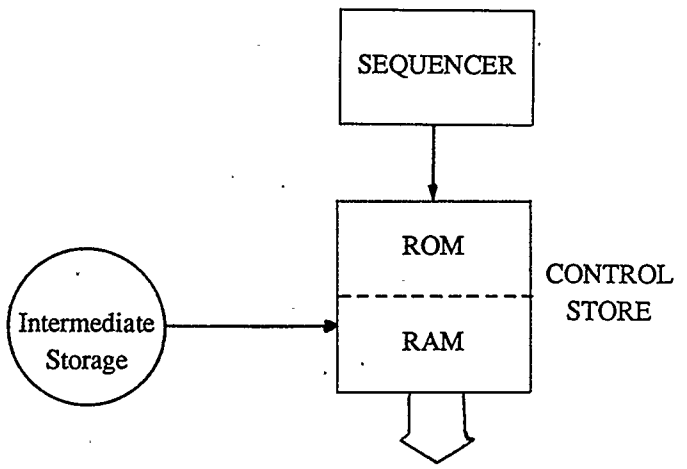


Fig. 3.7. Typical WCS organization

As far as the general purpose controller is concerned there were several shortcomings of the loading scheme described above:

(1) The necessity of a non-volatile intermediate storage such as EPROMs, cassette tape, etc. compromises the flexibility of the system. Modification of a microprogram residing in the intermediate storage can only be done on a separate device (e.g. EPROM programmer) which was simply not desirable in a development tool.

(2) Conceptually, the intermediate storage can be eliminated by establishing direct communications with a host computer. With this method the microprograms can be stored and modified in the host system and then downloaded to the WCS. However, the microroutine which facilitates the host interface must reside in the ROM section of the WCS.

An increase in flexibility can be obtained by assigning a microprocessor to the downloading task. The functions of the microprocessor system, the *downloading unit*, are then extended to include various debugging facilities; In its final form, this support unit is called the *task manager* (described later). The next section describes the downloading procedure as implied by the chosen architecture (fig. 3.6). It will be referred to later during the discussion of the downloading unit.

3.3.1. Downloading Procedure

There are two modes of operation for the controller, *RUN* and *LOAD*, which are selected by the downloading unit. In the *RUN* mode, the loading of the WCS is

disabled and the sequencer receives the program flow instructions from the pipeline register. Data input of the sequencer can be fed from any of the three busses, PA, MA or TA, as controlled by the microinstruction. In the *LOAD* mode, a special instruction is forced to the sequencer through the instruction multiplexer. This instruction (JMAP) enables the MA bus and the sequencer becomes transparent between its input and output. Consequently the downloading unit can directly address the control storage, allowing the WCS to be loaded.

The downloading unit transfers the microprogram to the WCS through the 8-bit data bus which was included in the interface bus (fig. 3.6). The data bus is bidirectional in order to allow the downloading unit to read the contents of the microprogram memory for debugging purposes. The WCS was arranged in 8-bit modules to accommodate the 8-bit data bus. These modules are called *banks*. Fig. 3.8 illustrates the bank organization of the WCS. The individual banks are selected through the BA (Bank address) bus, which also is a part of the interface bus. The bank-select logic demultiplexes the read and write strobe signals from the interface bus to the selected module, enabling one bank at a time.

Due to lack of space on the physical board, the WCS read back facility was not implemented in the prototype controller. This limitation can be justified by the fact that the microprogram transfers through the interface bus were found to be very reliable. Also the documented microprograms can just as easily be modified on the mainframe and downloaded to the WCS, avoiding the need to modify WCS using the keyboard and then remembering (reading) the changes.

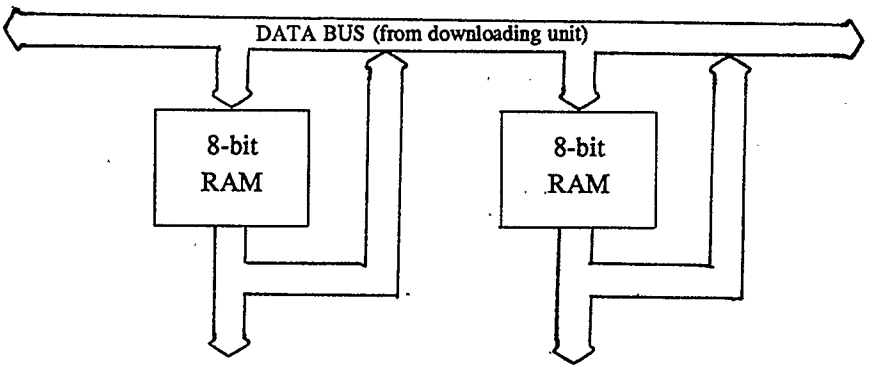


Fig. 3.8. Bank organization within the WCS

3.4. The Task Manager

The motivation for a microprocessor controlled downloading unit was established in the previous section. It was observed that such an intelligent unit could also be utilized as a development tool or an overhead task controller. The possibility of employing the downloading unit as the manager of several processors was particularly appealing when the necessities of the signal processing system (fig. 1.1) were considered. The jobs that the task manager was expected to perform are listed below:

1. Downloading the WCSs of several controllers,
2. Giving macro instructions to the controllers,
3. Passing parameters to the target system(s),
4. Initializing the processing units upon power-up in a stand-alone mode.

The last requirement arises from the expectation that the signal processing system

would be assembled as a mobile unit in a later stage. In this case, the task controller will be used to start-up the system, or it may be modified to handle the front panel (user) interface.

In the development stage of the signal processor, the most important function of the task manager is to facilitate host communications. The whole system (the manager, the controller and the target system) operates under complete control of the host computer during the development phase. Thus the discussion in the following chapters concentrates on the host control of the system. Nevertheless, stand-alone operation mode is also incorporated into the design of the task manager [16].

3.4.1. Implementation of the Task Manager

The task manager was built around an Intel 8031 microprocessor. The 8031 offers tremendous advantages over other general purpose microprocessors as it combines an UART, two counters and four 8-bit bit-addressable data ports with an 8-bit ALU, a Boolean processor and 128 bytes of on-board RAM on a single chip. The 8031 is capable of addressing 64 Kbytes of program memory (ROM) and 64 Kbytes of data memory (RAM/ROM). In addition, it supports a priority based interrupt structure and several stack operations.

Fig. 3.9 illustrates the functional organization of the task manager. As shown in the diagram, the task manager has interfaces with four different systems, the microcontroller, the host, the target system and the RAM/ROM expansion board. These interfaces represent different functions of the task manager and are described

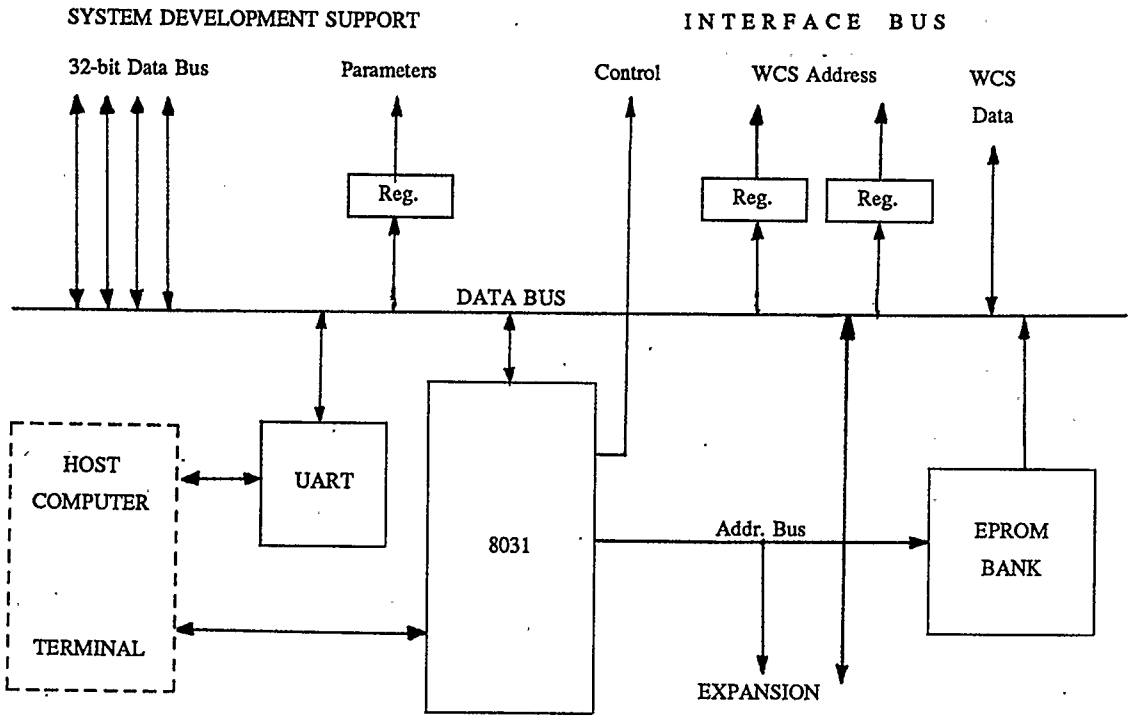


Fig. 3.9. The task manager

in the next four sections.

3.4.2. Downloading The WCSs

The downloading of the microprograms to the WCSs is done through the controller interface bus. The contents of this bus are:

1. the 12-bit WCS address field,
2. an extension of the 8031 data bus,
3. the bank-select field,
4. the 2-bit WCS-select field,
5. the WCS read/write strobes and
6. the MODE signal.

All signals are sourced by the task manager except the data bus which is bidirectional.

The downloading procedure was previously discussed without mentioning the WCS-select field which enables the task manager to download WCSs of up to four separate control units. Only the control unit that is identified by the WCS-select field will respond to the read/write strobes of the interface bus. However, the MODE signal is interpreted by *all* controllers regardless of the contents of the WCS-select field i.e. all controllers are in the same operating mode (*RUN* or *LOAD*) at any given time.

The controller interface bus was therefore optimized for downloading four WCSs, each up to 4 Kwords deep and 16 banks (128 bits) wide. Different organizations of the WCSs totaling up to 512 Kbanks may be achieved by modifying the decoding logic at the controller end.

3.4.3. System Development Support

The target system interface is one of the features that make the downloading unit a more general development tool. During the development of the multi-stage signal processor (fig. 1.1), each stage or processing unit has to be tested separately. When a stage is isolated from the others, the inputs and outputs of this stage must be simulated in order to carry out the testing. For example, during the development of the FFT board, a block of data is required to be written into the data memory. Since the front end processor may not be available, another data source is required.

The task manager is the logical choice to simulate the input and receive the output of a processing unit. A 32-bit bidirectional data port is added to the task manager for this purpose. Data is processed in 8-bit segments on this port and the necessary hardware for handshaking has to be built on the target system end. An example of the hardware and utilization of this port can be found in the next chapter.

In addition to the 32-bit data port, an 8-bit registered field was included in the target system interface to pass any required parameters, to the processing unit. This field is particularly useful to handle the global values which may not be implemented in the microprogram for practical reasons. An example of such global values is the length of the data block upon which the processing system will operate or the size (order) of the algorithm implemented.

3.4.4. Host Communications

Two full duplex asynchronous serial lines with up to 9600 baud rates were provided to maintain the host communications. One of these lines was connected to the host and the other to a terminal. The task manager is usually transparent between the host and the terminal but it can trap escape sequences to initiate the data transfers through the downloading interface or the target system interface.

All microprogram and data transfers between the host and the task manager are carried out with the *U-RECORD format* which was developed specifically for this purpose. This is a logical extension of the Motorola S-RECORD format. The data is converted from binary to ASCII to form the U-RECORD file which also includes several check values such as sum of the data bytes in the record, sum of the data bytes in the file, etc. Several different record types, including the microprogram record and the data record (for the target system), are supported. The complete specifications of the U-RECORD format can be found in the manual on the control unit [16].

3.4.5. Other Features

It was previously mentioned that the task manager may be required to run without the host support. In this case the functions of the task manager is reduced to the downloading of the control stores. The microprograms to be downloaded must reside in on-board non-volatile memories. 16 Kbytes of EPROM, expandable to full 64 Kbytes, were provided for this purpose. Approximately 6 Kbytes of this space is

needed for 8031 programs and the rest is available for storing the microprograms.

All of the peripheral devices, except the EPROMs, were mapped into the data memory space. In order to minimize the chip count a partial mapping method was employed. The data memory was divided into four equal sections which are:

1. the WCS through the downloading interface bus,
2. the 32-bit data port,
3. the on-board devices, i.e. registers and the UART and
4. the RAM.

No external RAM was provided with the task manager since the internal RAM of the 8031 was found to be sufficient for the current facilities. Nevertheless, up to 16 Kbytes of RAM can be added to the system through the expansion port.

3.5. Software Support

The software support for the general purpose controller can be classified into two groups: the microprogram development support, which consists of the *META* assembler [17], and the software library for the controller. The software library was created using C-compiler for the host computer (VAX 11/750) programs, and the *as8031* assembler [16] for the 8031 routines.

The *META* assembler is a general purpose assembler for microprogrammable systems. It compiles the source code for an arbitrary, user-defined architecture using optional mnemonics which are also user-defined. Therefore *META* is the assembler with which the user should be familiar in order to utilize the controller in a

structured, documentable manner. *META* is capable of producing output in several useful formats, including the U-RECORD format.

The software library consists of several user-friendly programs which interact with the 8031 to perform the various functions of the task manager. With these programs the task manager becomes transparent between the host and the microprogrammable control unit. Therefore the user does not have to be familiar with the details of the task manager. The contents and the functions of the software library are listed below,

dwcs

Downloads microprograms from the host to the WCS of the controller. The microprograms are normally generated by the *META* assembler.

download

Downloads data files to the memory of the target system.

upload

Uploads data from the target system to the host. It provides the data files in both hexadecimal and floating point formats.

start.exec

Start the execution of the microprograms in the WCS. This programs simply changes the operating mode of the controller from *LOAD* to *RUN*.

ehalt

Stops the execution of the microprograms.

addr.load

Sets the address where the execution of the microroutines commence.

datau

Generates a data file in U-record format from floating point data input.

3.6. Summary

The first phase of this thesis was completed with the construction of the general purpose controller and the task manager (Figs. 3.10, 3.11 and 3.12). Both systems were installed together as a unit, shown in fig. 3.13. The target system is connected to the control unit with 4 flat cables (at the left-hand side of the picture) which carry the microword, the condition code inputs and the system clock. The flat cables labeled "1" and "2" in this picture carry the 32-bit data bus and the 8-bit parameter field, respectively.

The control unit, i.e. the combination of the controller and the task manager, provides an excellent means of testing any microprogrammed processing unit, especially when used with the *META* assembler. During the development stage of a signal processing algorithm implementation, the designer does not have to be concerned about the control unit of the processor. Thus the final product can be realized much faster. The usefulness of the software library and the development system can be attested by the ease with which a microprogrammed implementation of the Burg algorithm has been developed by Nichols [8].

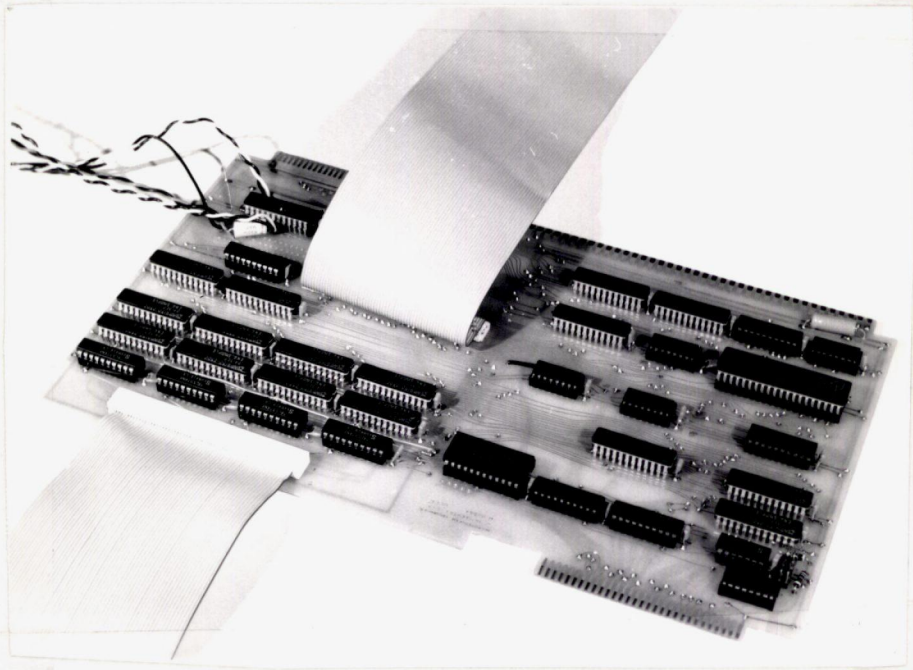


Fig. 3.10. Sequencer and WCS board

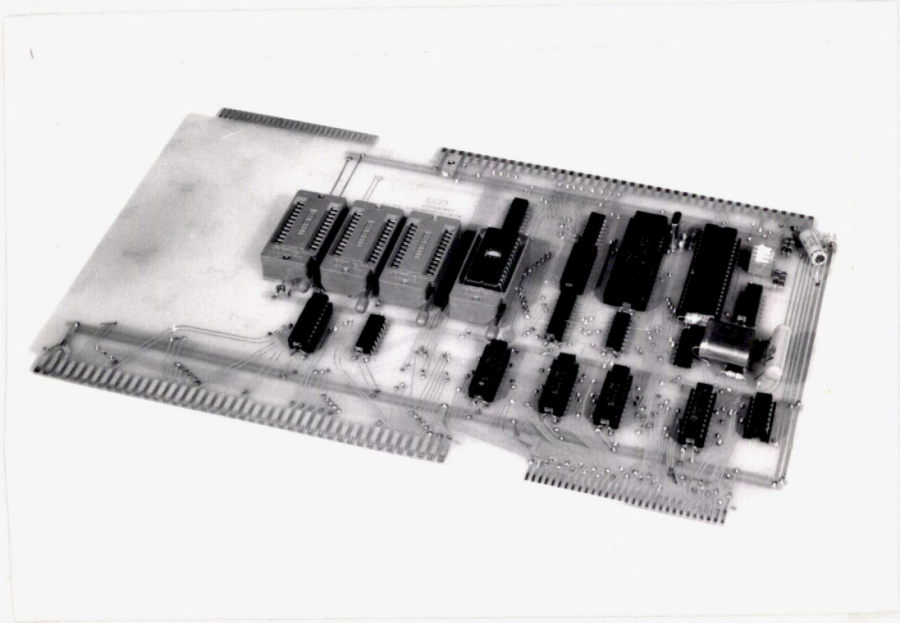


Fig. 3.11. Task manager board

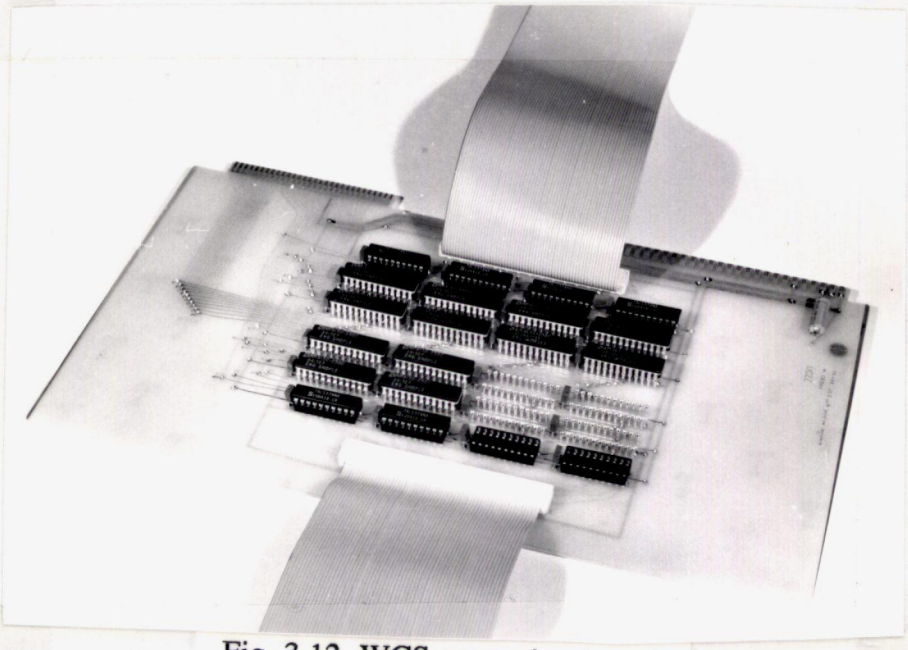


Fig. 3.12. WCS expansion board

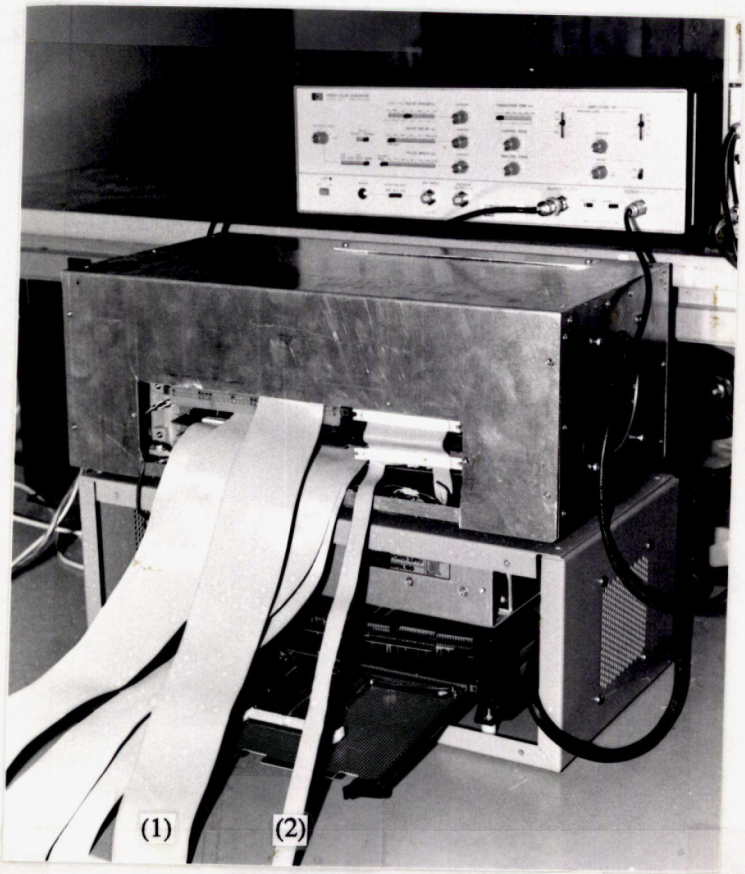


Fig. 3.13. General purpose controller

CHAPTER 4

AN ARCHITECTURE FOR SIGNAL PROCESSING

4.0. Introduction

The design of an application oriented processor consists of two major stages, the theoretical problem modeling and the implementation of the processing models. These two stages can be further partitioned as illustrated in fig. 4.1 [18]. As established in chapter 1, the purpose of this study was to design an architecture aimed at general DSP applications and therefore its scope is limited to the implementation stage. However, in order to develop an efficient hardware structure, the characteristics of the operations to be performed must be specified. Accordingly, the common computational requirements of the DSP algorithms are examined in the first section of this chapter, followed by the presentation of a benchmark algorithm, for which the architecture was optimized. Then the development of the architecture, the data structure and associated component technology are discussed.

The last part of the chapter deals with the physical implementation of the proposed architecture. A processing unit implementing a fast Fourier transform (FFT) algorithm was built. The parallel processing techniques and the efficiency of the architecture are demonstrated.

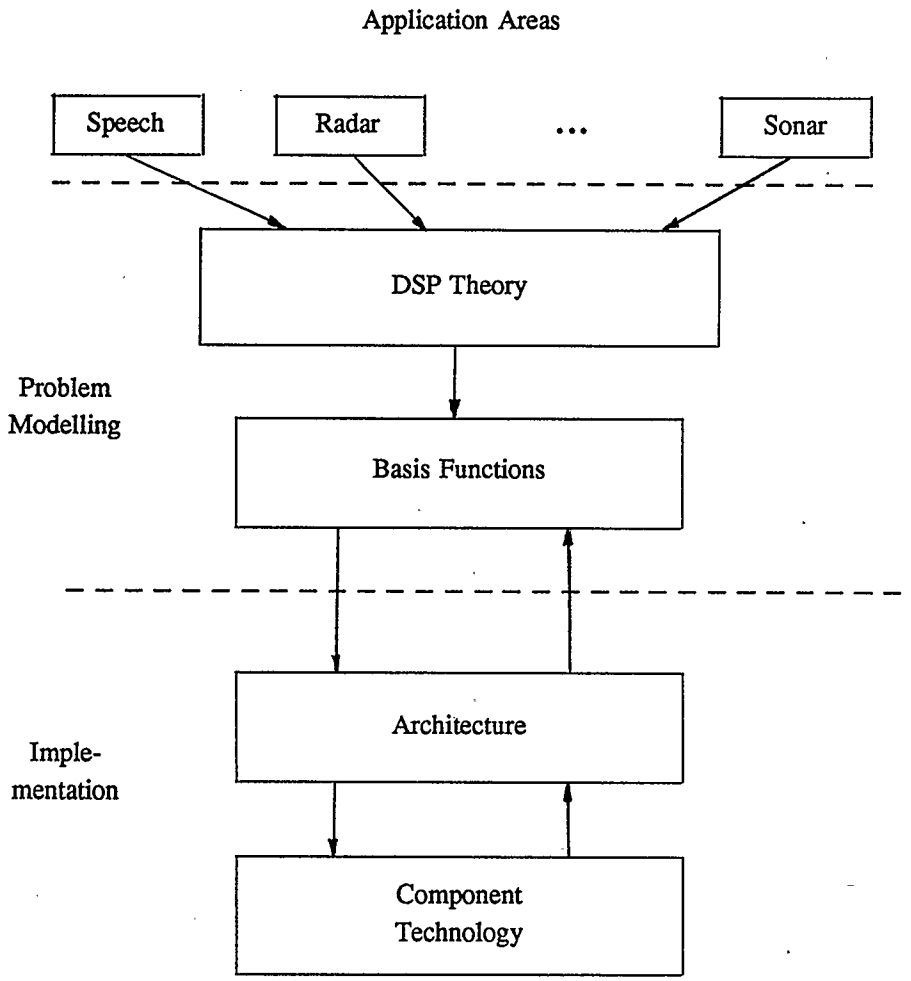


Fig. 4.1. A perspective of designing digital signal processors

4.1. Processing Requirements

In general, DSP theory specifies the order in which the signal is manipulated to transform the information of interest to a desired representation. The actual signal manipulations tend to be based on a small set of basic signal processing operations

such as convolution, correlation, difference equation calculation, DFT coefficient calculation, matrix operations etc. The purpose of this section is to develop a processing model on which the architecture will be based. This processing model is a subalgorithm common to most DSP operations and it is called the *benchmark algorithm.* A brief review of the DSP operations is given below in order to present the establishment of the benchmark algorithm.

4.1.1. Basic Operations of DSP

1. *Convolution*

In general, the convolution of two signals, $x[k]$ and $h[k]$ is given by the equation

$$y[n] = h[k]*x[k] = \sum_{k=0}^{N} h[k]x[n-k] \ , \ \ n = 0,1, \ldots ,N \qquad ..(4.1)$$

It is assumed that all signals are defined for $k = 0,1, \ldots ,N$.

2. *Correlation*

The correlation coefficients are calculated with the equation

$$\phi[n] = \sum_{k=-N}^{N} y^*[k]x[n+k] \qquad ..(4.2)$$

where [-N,N] is the window over which the signals $x[k]$ and $y[k]$ are correlated and $y^*[k]$ represents the complex conjugate of $y[k]$.

3. *Difference Equation Calculation*

The equation

$$y[n]=\sum_{k=0}^{N} a_k x[n-k] - \sum_{k=1}^{M} b_k y[n-k] \qquad ..(4.3)$$

represents a general infinite impulse response (IIR) filter where $x[k]$ is the input and $y[k]$ is the output sequences. The transfer function of the filter is defined by the parameters a_k, b_k, N and M. A particularly interesting case is where N and M are both limited to 2, giving the standard second-order filter which can be used as a building block to construct arbitrary filters.

Another important variation of eqn. 4.3 is achieved by setting all b_k's to zero. The filter then becomes a finite impulse response (FIR) filter with the difference equation

$$y[n] = \sum_{k=0}^{N} a_k x[n-k] \qquad ..(4.4)$$

4. *DFT Calculation*

Computation of the discrete Fourier transformation is one of the most widely required operations of the DSP. In general the DFT is represented by the equation

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j(2\pi/N)nk} \quad k = 0,1, \ldots ,N-1 \qquad ..(4.5a)$$

where $x[n]$ is the signal and $X[k]$ is the frequency spectrum. The inverse relation is very similar,

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X[k]e^{j(2\pi/N)nk} \quad n = 0,1, \ldots, N-1 \qquad ..(4.5b)$$

This computation is usually realized by using the well known FFT algorithms based on the butterfly computation. One of the most commonly used butterfly operations is represented by the equation

$$\begin{aligned} A' &= A + BW_N^k \\ B' &= A - BW_N^k \end{aligned} \qquad ..(4.6)$$

where A and B are complex numbers and W_N^k is equal to $e^{-j(2\pi/N)k}$ for the forward transform and $e^{j(2\pi/N)k}$ for the inverse transform.

5. *Frequency Translation*

Complex frequency demodulation is performed by the equation

$$y[n] = e^{j2\pi nk}x[n] \qquad ..(4.7)$$

This calculation is frequently required for band-selected analysis.

6. *Magnitude Calculation*

Another basic computational requirement is the calculation of the magnitude squared of the complex sequences:

$$|X[k]|^2 = X[k]X^*[k] \qquad ..(4.8)$$

7. Other Numerical Operations

In general, the DSP algorithms may require several numerical operations, such as division, logarithm, exponential and matrix manipulation. Although these operations have no significance within the purposes of this study, they are mentioned here to complete the list of basic operations.

4.1.2. The Benchmark Algorithm

An examination of the first six basic operations defined in the previous section reveals some common points among these operations. Basically all of these operations are based on complex multiplication. Furthermore, the summations in eqns. 4.1 to 4.4 can be implemented in steps of the form

$$C = C + AB \qquad ..(4.9)$$

where A, B and C are complex numbers. Eqn. 4.9 is also similar to one half of the butterfly calculation (eqn. 4.5). One minor difference is that some operations require multiplication of the data point with a constant, such as a_k, b_k, or W_N^k, while others require multiplication of two data points.

We have thus established the importance of the subalgorithm, the *fundamental operation*, where two complex numbers are multiplied and added to another. The butterfly computation, which consists of two fundamental operations, was adopted as the benchmark algorithm. The design of the architecture was based on the computational requirements of this benchmark algorithm. Then, the multiplicative coefficient of the butterfly operation, W_N^k, was generalized to accommodate the other

basic operations.

Although the basic operations 1 through 4 consist of repetitive computation of the fundamental operation, each of them has a unique addressing sequence which is reflected by the subscripts in the corresponding equations. This fact suggests that the processing unit can be divided into two basic blocks, the address generator and the arithmetic processor. The address generator will compute the necessary sequence of the data addresses to be used by the arithmetic processor which will carry out the fundamental calculation. The design of the processing units will be simplified with this composition since each arithmetic processor can be made identical for the multiprocessor system (fig. 1.1). The address generator specifies the algorithm(s) to be executed in the processing unit and must therefore be designed according to the requirements of each processor.

4.2. Functional Requirements

A processing unit is composed of several basic building blocks such as arithmetic-logic unit (ALU), multiplier, memory, etc. The architectural design of the processor involves specifying the types of the building blocks and the number of blocks of each type as well as devising an interconnecting scheme. In this section, a summary of these blocks are given in order to clarify the requirements from the architecture.

Parallel processing is an important consideration for the design of the processing unit and an understanding of this technique is essential. For this reason, a brief review of parallelism and application of the parallel processing concept to the DSP algorithms is given before the basic functional blocks of the architecture are discussed.

4.2.1. Parallelism and Pipelining

The purpose of implementing parallelism in the architecture is to achieve the maximum utilization of *all* resources and consequently higher speed and performance throughout the execution of an algorithm. The basic conditions that must be met in order to reach this objective are:

1. Complete control of each resource at any time,
2. Non-conflicting data busses for each independent resource and
3. The algorithm must be breakable into subalgorithms which can be executed simultaneously.

The first criterion can be easily fulfilled when microprogrammed control is employed. The microword outputs should control the individual devices on the system without any encoding. This type of microinstruction is called a *horizontal* microinstruction. Horizontal microwords often cause wasted space in the microprogram memory. However this is a necessary trade-off to achieve any degree of parallel processing, particularly in the early development stages. Another trade-off, as pointed out by the second criterion, is between the number of the data paths

in the system and the extent of parallelism. It is obvious that no concurrent processing is possible when all basic resources, such as ALU, memory, register file, etc., communicate on a single data bus. On the other hand, an increased number of interconnection paths pose several practical problems, such as increased RF noise, larger printed-circuit boards, longer development time etc.

Although breaking the algorithm into simultaneously executable subalgorithms is normally a software related problem, an understanding of this process is essential to incorporate the necessary facilities in the hardware. As an example, consider three separate processes constituting a larger process, as illustrated in fig. 4.2. A system broken into such concurrent processors operating on a single data stream is often referred to as a *pipelined* system. Pipelining, in general, is an important concept in parallel processing. Using pipelining registers in the interface allows two synchronous processes to run in different phases. For example, in fig. 4.2, the process P2 operates on the data block processed by P1 and while nth data block is being passed on to P2, P1 can start processing the $(n+1)$th data block.

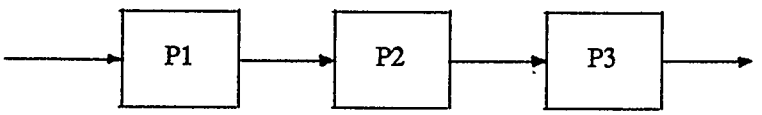


Fig. 4.2. A pipelined parallel processing system

In general, concurrent computation can be performed in several levels. An example showing the partitioning of the signal processing task of fig. 1.1 is illustrated as a tree in fig. 4.3. Each level in this tree represents a series of pipelined

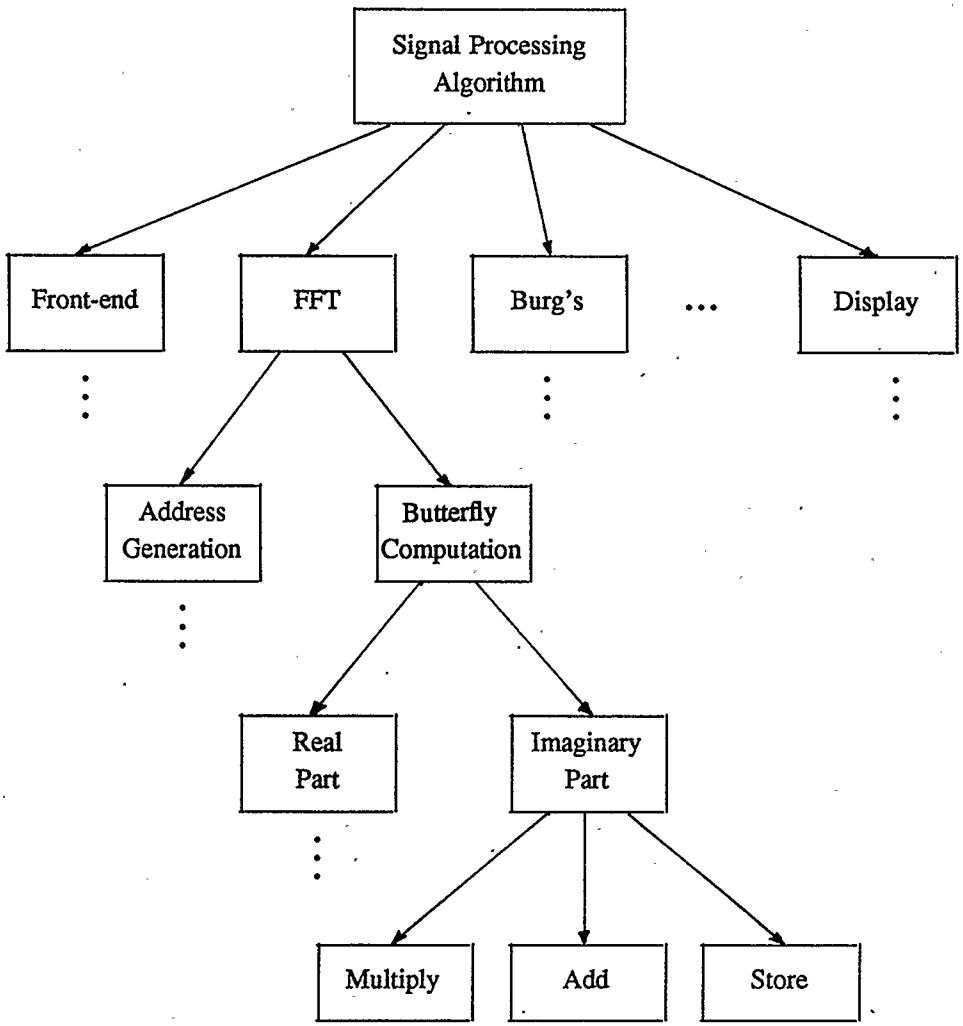


Fig. 4.3. Parallel processing in different levels

parallel processes. The stages of the signal processing algorithm form the first level of the tree. Each stage performs a basic operation of the DSP which can be separable into two subalgorithms, the address generation and the fundamental operation. Furthermore, real and imaginary parts of the data can be calculated simultaneously with concurrent instructions such as multiply, add and store.

The interface between the address generator and the fundamental operation processor constitutes a fine example of pipelining. The address generation must start first to calculate the addresses required for the first fundamental operation. The address of the first data point is transferred to a pipeline register and at the same time the generator starts calculating the next address. The arithmetic processor, which carries out the fundamental operation, can access the first data point as soon as the address is in the pipeline register. In other words, the address generator calculates the second address while the arithmetic generator processes the first data point. Throughout the rest of the execution of the current basic operation the address generator stays ahead of the arithmetic processor allowing both algorithms to run continuously.

4.2.2. Functional Blocks of the Processing Unit

The basic building blocks of the processing unit are summarized below.

1. *Memory*

A random access memory provides the storage for input and output data streams. All intermediate data blocks may also be stored in the RAM.

2. *ALU*

A standard arithmetic logic unit performs the basic functions such as boolean AND, addition, subtraction and negation.

3. *Multiplier*

The importance of the multiplication in DSP algorithms, especially in complex data processing, was established previously. Therefore, the hardware multiplier is a necessary element of any high-performance signal processing unit.

4. *Scratch-pad Register File*

A register file holds the temporary data values needed for the ALU or multiplier operations. Storing such temporary values in the main memory is inefficient because, in addition to its relative slowness, the memory access represents a bottleneck in the arithmetic processor due to a large number of read/write operations. For maximum efficiency, the register file should provide multiple input and output ports. For example, two outputs for the ALU and two for the multiplier may be needed to have both devices operating continuously, if only one register file is available.

In most multi-ALU architectures, a register file is provided for each ALU and consequently most LSI ALU chips in the market include a scratch-pad register file. Hence the register file is considered a part of the ALU block instead of a separate block. This view is followed in this study. Any referral to ALU as a resource implies the combination of the ALU and the register file.

5. *Pipeline Registers*

As discussed in the previous section, the pipeline registers are primarily used for interconnecting different resources, enabling one resource to operate at a different step than the others. Often, as in the case of address generation and the arithmetic processor, the "step" is actually several cycles and in such cases multi-level pipelining is required.

6. *Scaler*

A scaling block is necessary to implement hardware overflow protection for the DSP algorithms. Conceptually the scaler consists of two parts, the shifter and the overflow detection logic. The overflow detection logic can be omitted if the scaling is to be controlled by the software (microprogram). The shifter may also be required for numerical algorithms such as division.

7. *Address Generator*

As discussed before, the design of the address generator is unique to each processing unit. In general, the structure of the address generator is complex and it is usually implemented with general purpose ALU chips. An example is the implementation for the Burg algorithm with 3 ALUs (Am2901s) by Nichols [8]. One exception to this is the FFT processor for which a single chip FFT address sequencer is available (Am29540).

8. *Input/Output Interface*

This unit performs the data transfers between the processing unit and other systems. The specifications of the interface between processing units of the

overall system will be discussed later in this chapter.

9. *Coefficient Memory*

This memory is different from the main memory since it is used to store the coefficients only and it is not necessary for all processing units. The coefficient memory is essential for the FFT processor where a ROM is used to generate the real and imaginary parts of the constant W_N^k. A RAM may be required for processors implementing a difference equation with variable coefficients, such as adaptable filters.

The precalculation of the coefficients provides a considerable improvement in the system speed as it avoids the necessity of on-line computation of time consuming algorithms.

Fig. 4.4 illustrates how these blocks are placed within the general structure common to all processing units. As it was mentioned the address generator provides each processor with its distinct flavor. On the other hand, the architecture of the arithmetic processing block should remain the same since all algorithms are based on the same basic step, the fundamental operation. The next section concentrates on the design of this block.

4.3. The Arithmetic Processor

The elements of the arithmetic processor were introduced in the previous section. Three critical resources, the ALU, the multiplier and the data busses between the memory and the arithmetic processor, can be identified among these

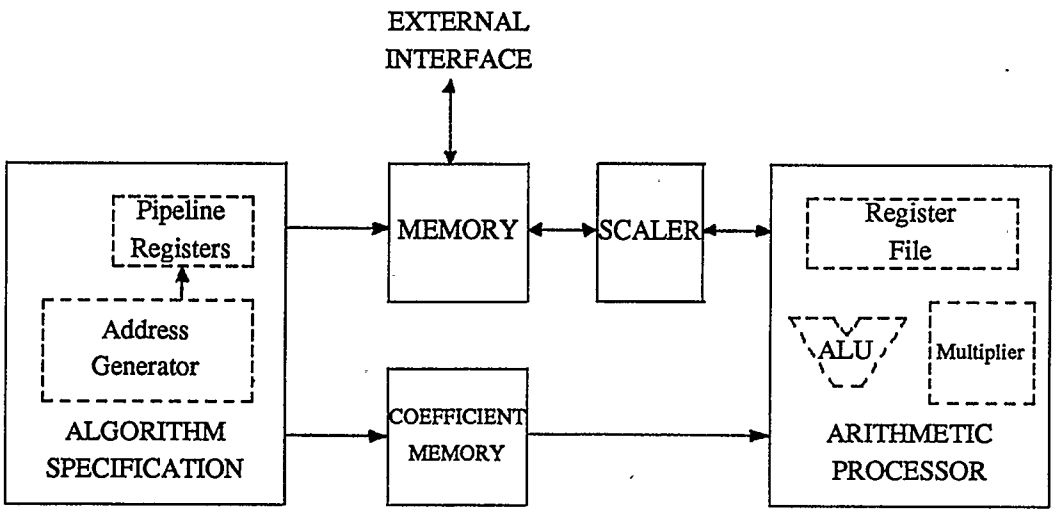


Fig. 4.4. General organization of the processing unit

elements. In this section the trade-offs involved with the increase in the multiplicity of these resources are examined and an optimum combination is sought. Then the related current component technology is reviewed and its effects on the specifications of the data structure is examined. Finally, an interconnection scheme between the elements of the arithmetic unit is presented.

4.3.1. Optimization of Resources

Conceivably, the computational power of the arithmetic processor can be enhanced by increasing the numbers of the critical resources. The first trade-off in increasing the multiplicity of the resources is between the performance and cost (complexity) of the processor.

Any optimization of this trade-off should be based on the benchmark operation which consists of two complex operations,

$$A' = A + BW^k$$
$$B' = A - BW^k \qquad ..(4.10)$$

The subscript N of the coefficient W^k is dropped since we are dealing with fixed-length data blocks. The complex equations (4.10) have to be expanded for real and imaginary parts of A' and B',

$$A'_R = A_R + (B_R W_R^k - B_I W_I^k) \qquad ..(4.11a)$$
$$B'_R = A_R - (B_R W_R^k - B_I W_I^k) \qquad ..(4.11b)$$
$$A'_I = A_I + (B_R W_I^k + B_I W_R^k) \qquad ..(4.11c)$$
$$B'_I = A_I - (B_R W_I^k + B_I W_R^k) \qquad ..(4.11d)$$

where the subscripts R and I denote the real and the imaginary parts of a complex number, respectively. From equations 4.11, it can be seen that the benchmark operation requires 4 multiplications and 6 additions (subtractions). In addition 4 values, A_R, A_I, B_R and B_I, must be read and then stored, hence the data memory must be accessed 8 times. Although the coefficients W_R^k and W_I^k represent two additional read operations, this does not pose a problem since the coefficients are read directly into the multiplier through a separate bus.

In a one bus structure, the benchmark operation takes 8 cycles to complete with the ALU staying idle for 2 and the multiplier idle for 4 cycles of the 8 cycle period.

Hence the efficiencies of the data bus, the ALU and the multiplier are 100%, 75% and 50% respectively. The bottleneck is clearly the memory access, indicating that the number of the data busses must be increased. If two busses are used instead of one, the 8 memory accesses can be completed in 4 cycles, but the benchmark operation requires 6 cycles as the ALU becomes the bottleneck resource. Table 4.1 illustrates the results of progressive increases in the multiplicity of the resources [19]. The ''cycles'' column under each resource represents the number of the cycles needed to complete the assigned part of the benchmark operation when the corresponding multiplicity (''#'' column) is provided. The total number of cycles is the greatest of the ''cycles'' columns for each line. The efficiency colomn is determined by the ratio of the required number of cycles to the total number of cycles.

TOTAL CYCLES	DATA BUSES			ALU			MULTIPLIER		
	#	Cyc.	Efficiency	#	Cyc.	Efficiency	#	Cyc.	Efficiency
8	1	8	100%	1	6	75%	1	4	50%
6	2	4	67%	1	6	100%	1	4	67%
4	2	4	100%	2	3	75%	1	4	100%
3	4	2	67%	2	3	100%	2	2	67%
2	4	2	100%	3	2	100%	2	2	100%
1	8	1	100%	6	1	100%	4	1	100%

Table 4.1. Effects of the multiplicity of the resources

Table 4.1 clarifies the performance and complexity trade-off. The arithmetic processor can be made faster and more efficient at the cost of more resources and, consequently, more complex microprograms. The design of the other blocks in the processing unit are also effected by the choice of the speed of the architecture. For example, the architecture with 8 data busses demands a storage system with 8 independent data ports, which is very complicated to implement.

The structure of the address generator is another important consideration in choosing the architecture. The butterfly operation (eqn. 4.10) can be performed ''in-place'': i.e. after the operation is completed, the values A and B are not needed and can be overwritten by A' and B' respectively. Hence three addresses are required for the benchmark operation, those of A, B and W^k. If the benchmark operation is executed in three or more cycles, only one address needs to be generated per cycle thus a single bus structure is sufficient for the address generator block. The complexity of this block increases tremendously if the benchmark operation is to be completed in less than 3 cycles.

Among the alternatives presented in Table 4.1, the architecture with two data busses, two ALUs and one multiplier was chosen. The reasons for using this ''2-2-1'' architecture are given below:

(1) The two-bus structure is natural for complex valued data processing because the data memory can be divided into two logical sections, real and imaginary, with one bus assigned to each section. The same address is issued to both sections of the memory in order to access a complex valued data. Thus the addressing

scheme is, in fact, simpler than that in a one-bus structure where two addresses are required to access the real and imaginary parts of the data.

(2) With the same reasoning, two ALUs can be employed to perform the real and the imaginary calculations separately. Thus the addition of the second ALU causes minimal complexity in design and programming while improving the speed by 33% over the "2-1-1" structure. The average efficiency of all three resources is also increased from 78% to 91%.

(3) The 2-2-1 architecture appears to be a good compromise as using two multipliers is not feasible without at least 4 data busses.

(4) From Table 4.1, the total number of cycles required to complete the benchmark operation is 4. Therefore only one address per cycle needs to be generated, simplifying the address generator as explained previously. Since only three addresses are used in the operation, the remaining cycle can be used conveniently for incrementing a counter or other house-keeping tasks in the address generator.

As it will be discussed shortly, a family of specialized devices manufactured by the Advanced Micro Devices (AMD) were the key components of this architecture. The "2-2-1" architecture was also suggested by AMD as the optimal complexity [19].

4.3.2. Component Technology

Several microprogrammable processors with pipelined organizations are available today. Among these components, the Am29500 family products are particularly well suited to the needs of the arithmetic processor. The distinctive feature of the Am29500 family components is that they employ ECL internal technology with TTL input/output levels.

The ECL technology is widely used in manufacturing high-speed integrated circuit. However, the interface between ECL parts is always troublesome; with the bus capacitances and the lead inductances having to be kept very low to ensure reliable data communications. In addition the fan-out of the ECL output is low limiting the number of devices that can be connected to a bus. These problems are solved in Am29500 family by using ECL/TTL translators at the inputs and outputs of the integrated circuit devices. The TTL technology, although slower, offers more reliable communications between ICs and the signals are less susceptible to noise. Another advantage of using the TTL signal levels is that it allows the designer to use the widely available TTL logic components, such as buffers and drivers, with these fast devices.

The two relevant devices to the arithmetic processor in the Am29500 family are the multi-port pipelined ALU/Register file, Am29501, and the multiplier, Am29517. The Am29501 is a specialized 8-bit processor which executes multiple simultaneous data operations. The bit-slice design of the ALU allows cascading any number of Am29501s to implement wider formats for data. The structure of the register file,

which contains six registers, is the key element for performing simultaneous data transfers. The registers are pipelined and a multitude of data transfers can be performed in one cycle. Another important advantage of the Am29501 is that it has 3 data ports to speed up the data transactions with the memory and the multipler. Any combination of the register operations, the ALU operations and the I/O instructions can be programmed to occur in the same cycle [15].

The Am29517 is a 16-bit by 16-bit multiplier with a pipelined organization that improves its throughput. However the pipelining may be disabled to obtain a combinational multiplying logic.

4.3.3. Data Format

The selection of the Am29517 as the multiplier limits the word width to 16 bits. Since the Am29501 is an 8-bit processor, two of these are cascaded to form a 16-bit ALU. From here on, any reference to ALU will imply 2 cascaded Am29501s.

To complete the data description, the data format must also be specified. Generally, the best data format for DSP is the floating point format. However, with the available components (Am29501 and Am29517), the floating point operations must be carried out in software and therefore is not efficient. The block floating point (BFP) arithmetic is preferred with these components because the errors introduced by the truncation and scaling are less crucial in this format than it is in standard fixed-point format [20]. In the BFP format, the data word can be viewed as a mantissa where the common exponential is stored elsewhere. The natural choice for the

interval of real numbers representable in the mantissa is [-1,1). Therefore the value of a number formed by the bits a_{15}, a_{14}, . . . , a_0 of the mantissa is

$$-a_{15}2^0 + a_{14}2^{-1} + \cdots + a_0 2^{-15}$$

where a_n can be either 1 or 0. This representation is also called the two's-complement floating point representation.

As indicated above, the selection of the data format is dictated by the current component technology and it can be changed as advances are made. Other possibilities of the data format are discussed in the next chapter.

4.3.4. Data Flow Structure

The data paths connecting the blocks of the arithmetic processor are illustrated in fig. 4.5. Six different data busses are used to facilitate the full utilization of the resources. The two memory access busses, as required by the "2-2-1" architecture, are the *R–bus*, for the real part of the data, and the *I–bus*, for the imaginary part. The *Y–bus* carries the output of the multiplier, while the *RM*, *IM* and *C* busses provide the operands for the multiplier.

In order to generalize the architecture for all DSP basic operations, the operands of the multiplier are selected by a multiplexer. The real and the imaginary parts of the data from the corresponding ALUs, in addition to the real *or* imaginary part of the coefficient, are the three inputs of the multiplexer. In the generalized form, the multiplexer is capable transferring any combination of the inputs to its output. However this results in an overly complicated multiplexer block with high chip

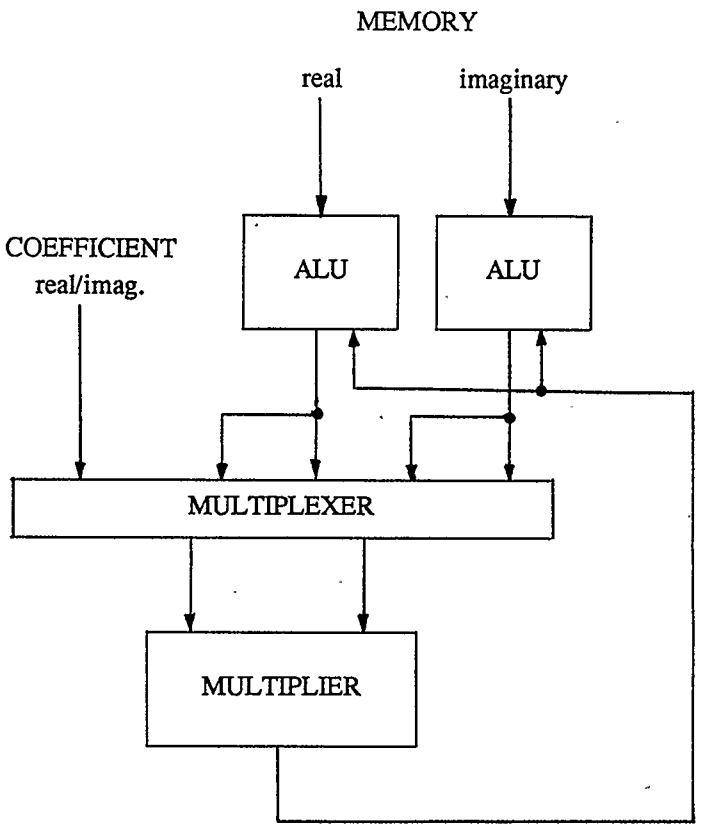


Fig. 4.5. Data flow in the arithmetic processor

count. Therefore this block should be modified later to meet the requirements of each processing unit. For example, for the FFT computation, the external multiplexing logic may be eliminated with the organization of fig. 4.6. On the other hand, the magnitude square calculation $(X_R^2 + X_I^2)$ requires both operands of the multiplier to be sourced from the same input.

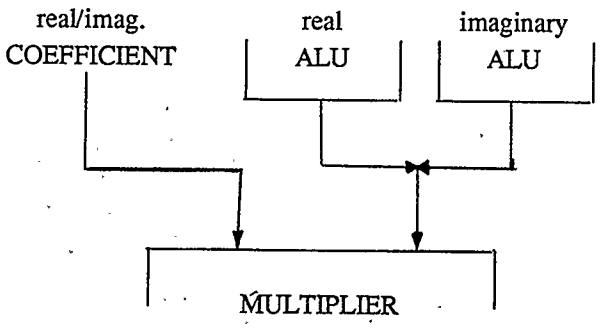


Fig. 4.6. An example of the simplified multiplier inputs

4.4. Architecture of the Processing Unit

The final architecture of the processing unit is illustrated in fig. 4.7. This figure is an evolution of fig. 4.4 to include the data flow structure of the arithmetic processor (fig. 4.5). The arithmetic processor was discussed extensively in the previous section, the design of the memory, the scaler and the I/O interface will be presented here.

4.4.1. The Memory

The memory block is composed of two identical storage segments as illustrated in fig. 4.8. The real and the imaginary parts of the complex data are stored in separate segments, both 16-bits wide. Since the segments can not be addressed separately, real and imaginary parts of the data must be accessed in the same cycle.

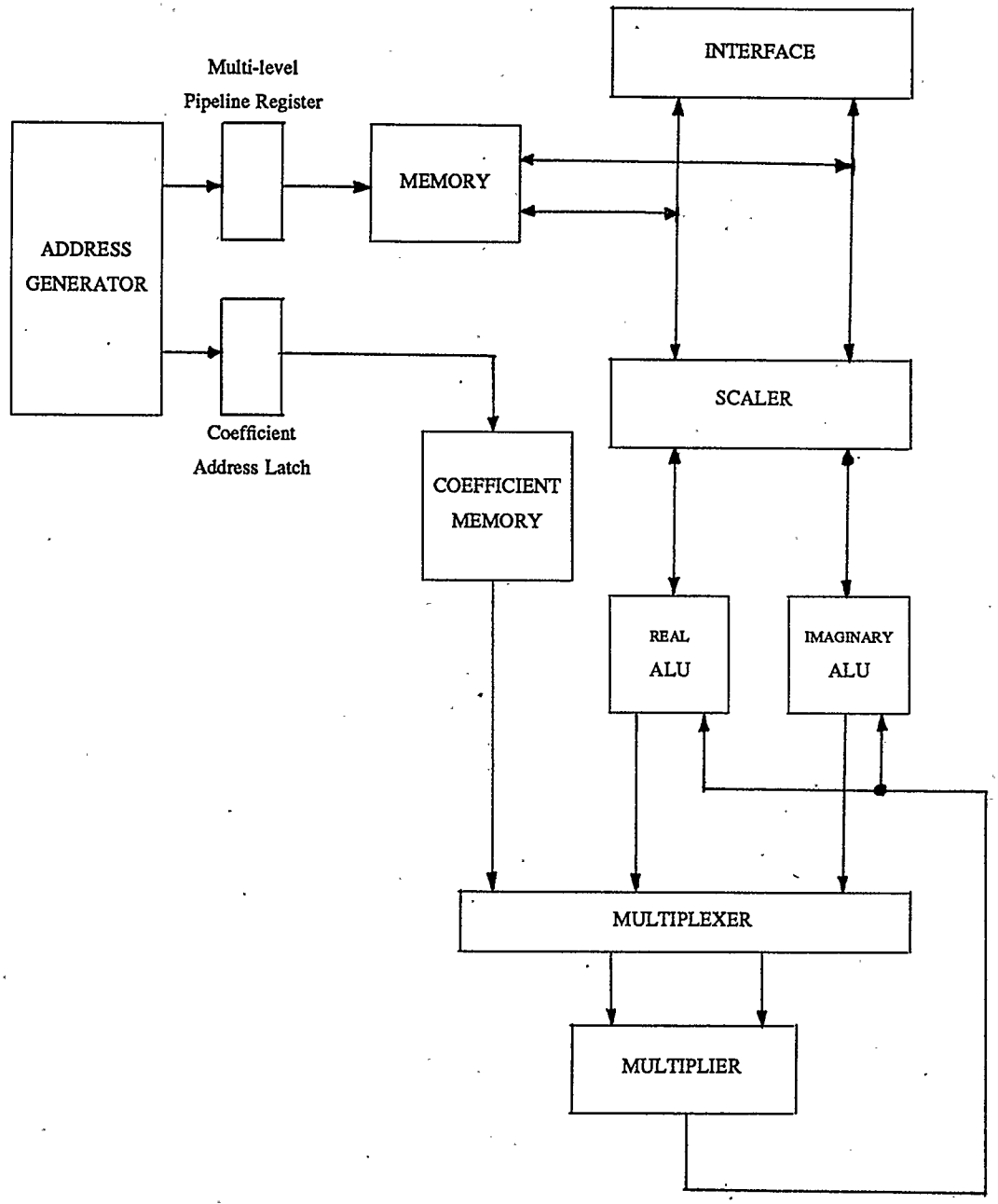


Fig. 4.7. The architecture of the processing unit

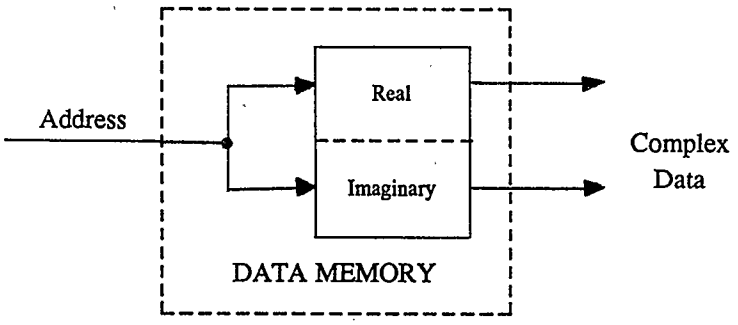


Fig. 4.8. The data memory

In this respect, the memory block can be viewed as a single 32-bit wide RAM.

4.4.2. The Scaler

The structure of the scaler block depends on the necessities of the processing unit. Fig. 4.9 illustrates a simple form of the scaler block for one data bus. The scaling is performed while data is being read from the memory based on some previous decision (i.e. overflow). For multi-bus structures, a shifter must be provided for each data bus. This simple scheme is sufficient for most applications where the BFP format is used.

4.4.3. I/O Interface

In order to carry out the isolated testing of a single processing unit during the development stage, the task manager provides the input data and receives the output data of the processing unit; this facility was described previously as the target system

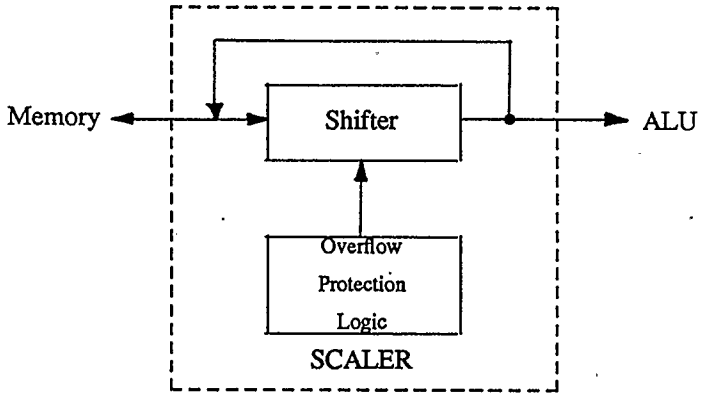


Fig. 4.9. The scaler block

development support. Hence, the I/O interface must be designed to handle the handshaking as specified by the task manager. Fig. 4.10 illustrates a simple yet effective scheme of a completely symmetrical interface. Two back-to-back registers allow the bidirectional data transfers while two flip-flops generate the necessary handshaking signals. Each flip-flop is set when data is written to the corresponding register and cleared when data is read by the other side. The data busses shown in the figure are 32-bits, i.e. real and imaginary parts are transferred in parallel when connecting to other processing units. The task manager processes the data in 8-bit segments, so that each register must be constructed with 4 individually accessible 8-bit registers to be able to test the processing unit with the help of the task manager.

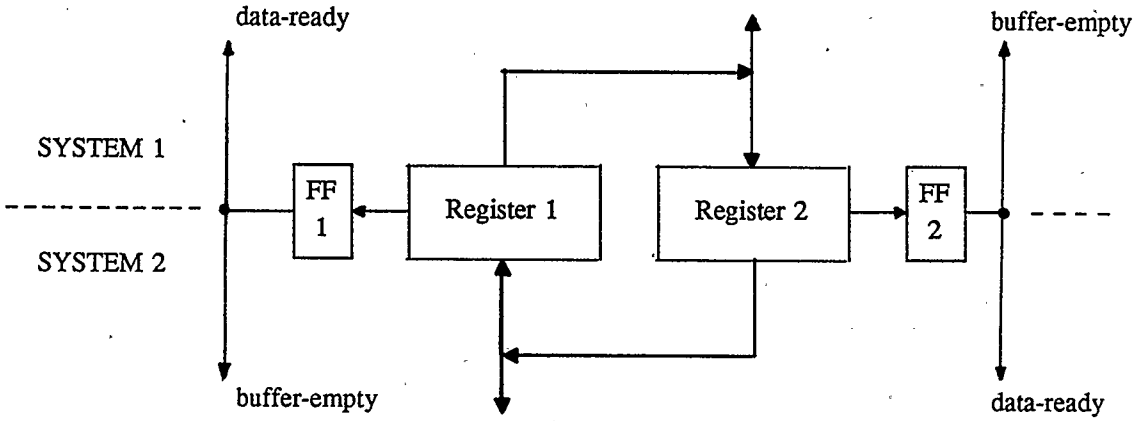


Fig. 4.10. I/O Interface

After the development stage, the same interface block can be used to tie the processing units together as required by the signal processing unit of fig. 1.1. The interconnection using a single system bus is illustrated in fig. 4.11. In order to demonstrate the timing of processing and transferring of data blocks, let's assume that a data block is processed in unit 1 (PU1) first, then in PU2 and PU3. The progress of the operation is shown in fig. 4.12, where it is assumed that all processing units require the same amount of time to carry out their respective algorithms. The numbers on the data transfer segments are those of the transmitting and the receiving processing units. The solid lines indicate that a PU is busy processing the data block indicated above the line, and the dotted lines correspond to the time period where data transfers are performed.

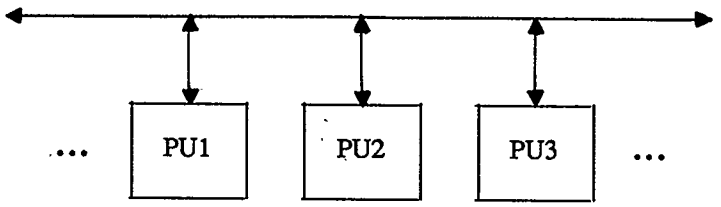


Fig. 4.11. Interconnection of the processing units with a system bus

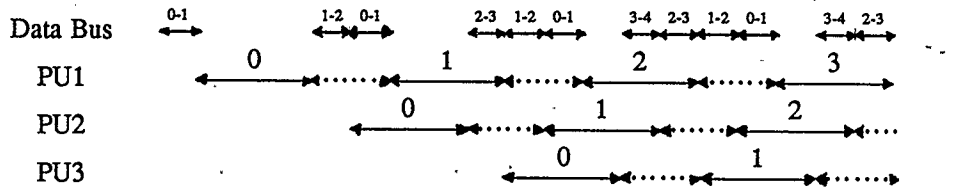


Fig. 4.12. Timing diagram of execution and data transfers

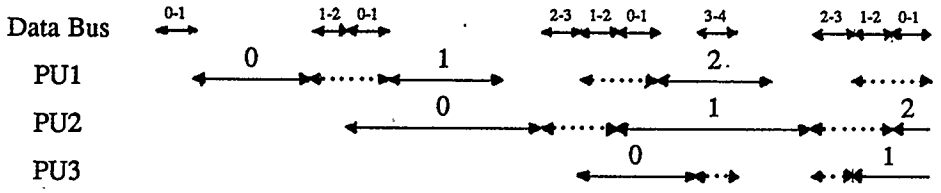


Fig. 4.13. Timing diagram with different execution speeds

In reality, the processing units require different time periods to complete an operation, and the slowest PU determines overall throughput. An example is illustrated in fig. 4.13, where it is assumed that PU2 requires more time than the others.

4.4.3.1. The Metastable Problem

A subtle problem was encountered in the implementation of the interface logic (fig. 4.9). The *metastable* problem is common to all clocked systems incorporating an asynchronous signal. It is an unavoidable condition and results in occasional system failure. To understand this problem let's consider the clocked system **S** (fig. 4.14). The output, **Y**, of this system depends on the asynchronous input event **D**. By asynchronous, it is implied that the occurrence of event **D** is not correlated to the clock pulse signal. Every clocked system such as **S** has a input set-up time, t_s, which means that the event **D** has to occur at least a time period of t_s before the clock edge. If this condition is not met, the system "crashes" because the output **Y**

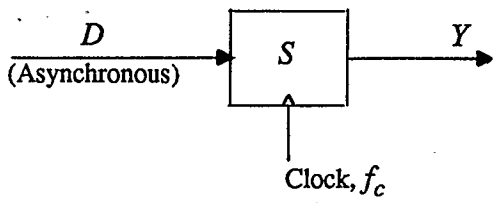


Fig. 4.14. The metastable problem

and the state of system **S** are not predictable. Since **D** is uncorrelated to the clock there is a finite probability of the occurrence of this failure. This probability can be calculated assuming **D** is totally asynchronous:

$$Failure\ Rate = \frac{set\text{–}up\ time}{clock\ period} = t_s f_c$$

where f_c is the clock frequency.

The metastable stable problem exist in the interface logic because of the status signals. The status signals *data-ready* and *buffer-empty* are tested by the controller in order to start a data transfer. However, these signals are generated by the read/write operations originating from the task manager and therefore are totally asynchronous to the controller. Fig. 4.15 illustrates the status signal path which is conceptually similar to that of fig. 4.14. The set-up time for the path shown is equal to the sum of the delays in the CC-multiplexer, the sequencer, the WCS and the set-up time of the pipeline register. From the data books the total set-up time, t_s, is estimated to be 70 ns. Hence, with a clock frequency of 5 MHz, the rate of a failure during a data transfer is

$$Failure\ Rate = 70ns \times 5MHz = 35\%.$$

The obvious deduction from this tremendously high probability is that a clock rate of 5 MHz is not attainable with this structure.

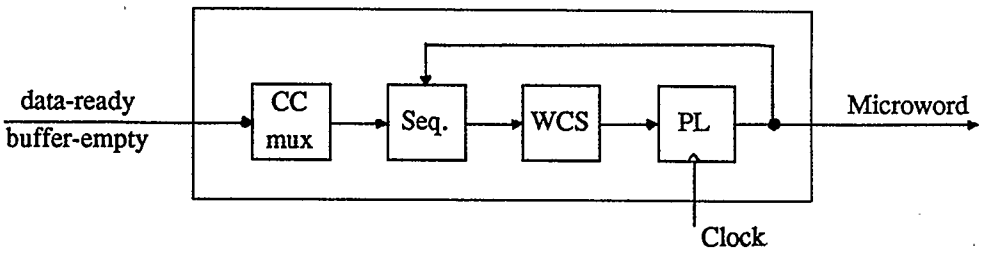


Fig. 4.15. Metastable problem in the control unit

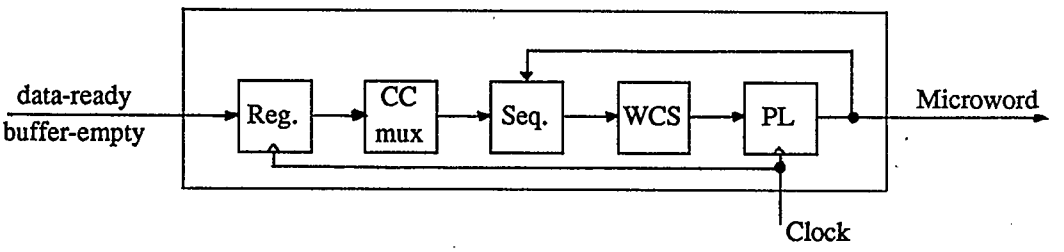


Fig. 4.16. Rectification of the metastable problem

This problem was rectified by inserting a register into the signal path, as shown in fig. 4.16. The set-up time for a register is typically 2 ns so that the failure rate with the same clock becomes

$$Failure\ Rate\ =\ 2ns \times 5MHz\ =\ 1\%.$$

Although this figure seems to be still high, it should be remembered that a register can easily recover from a metastable condition and at worst another clock edge is required to register the signal level. On the other hand, the recovery is unlikely if the clock edge occurs while the sequencer or the WCS is not stabilized.

4.5. An Example: The FFT Processor

To demonstrate the concepts developed in this chapter an FFT processor was built. In this section, first the hardware features of this processor and the overflow protection mechanism are discussed. Then the development of the software for parallel processing is presented. The block diagram of the FFT processor is given in fig. 4.17.

4.5.1. Characterization of the Processing Unit

The general architecture of the processing unit was described in the previous sections. As mentioned before, the characterization of a processing unit designated for a specific algorithm is reduced to the definition of 5 blocks,

1. The address generator(s), to specify the algorithm(s) to be executed,
2. the scaler, to protect the processor from overflow failure,
3. the multiplexer block, to supply the necessary data to the multiplier,
4. the type of coefficient memory, (if any), and
5. the pipeline registers.

In the following subsections, the FFT processor is defined by the specification of these blocks. The design of the scaler block is discussed in the next section.

4.5.1.1. The Address Generator

A single chip FFT sequencer, Am29540, was used as the address generator. Although the Am29540 is capable of generating addresses for a large variety of FFT algorithms, we will concentrate on the radix-2, decimation-in-time (DIT) FFT

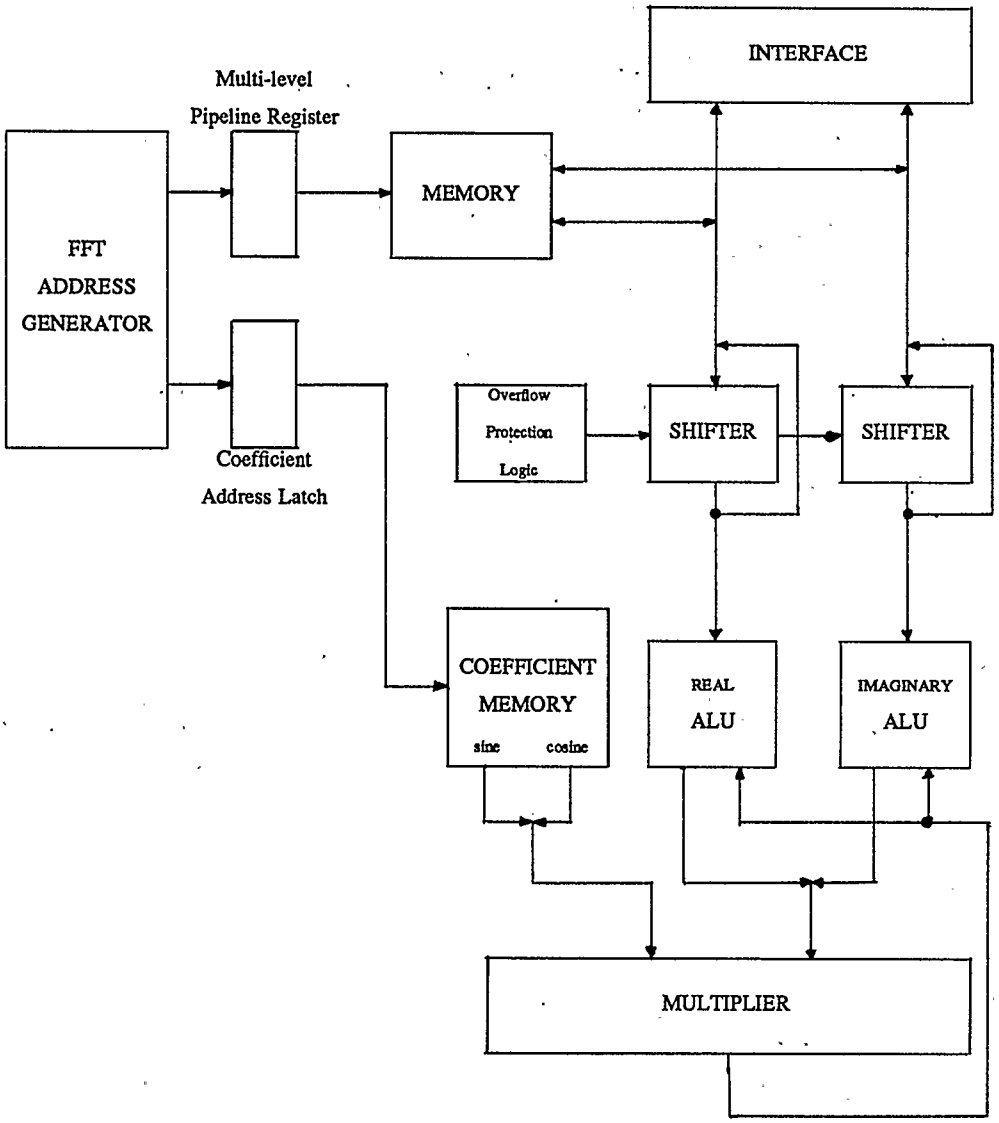


Fig. 4.17. The FFT processor

algorithm. The definition of this algorithm and other variations of FFT algorithms can be found in many sources (e.g. [21]). The Am29540 was also employed to download/upload the data block by addressing the data memory sequentially.

The transform length is one of the parameters required for the Am29540 and this is set by the task manager through the parameter register mentioned earlier. The length of the data block is necessarily a power of 2 due to the Am29540 specifications. The setting of the transform length independently of the FFT software allowed compact microprograms. Since the end-of-loop conditions were generated by the Am29540, software counters to keep track of the execution were not required.

4.5.1.2. The Multiplexer

The multiplexer block was not necessary for the FFT processor since only multiplication with a coefficient was required. The organization shown in fig. 4.6 was implemented.

4.5.1.3. The Coefficient Memory

The coefficient required for the butterfly operation is

$$W^k = e^{j\theta_k} = \cos\theta_k + j\sin\theta_k$$

Currently available sine/cosine ROMs were used to generate the real and imaginary components of the coefficient W^k. These ROMs are Am29526/7 ($-\sin\theta_k$) and Am29528/9 ($-\cos\theta_k$).

4.5.1.4. The Pipeline Registers

Two pipeline registers, each two level deep were required to hold the two data addresses for the butterfly operation. The two level depth was necessary for overlapping the execution of the butterflies.

A single latch was sufficient for the coefficient address since it does not change during the execution of a butterfly.

4.5.2. Overflow Protection in FFT

In this section we will first identify the requirements of the FFT algorithm for the overflow protection logic. From the definition of the DFT,

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk}$$

where $|x[n]| \leq \sigma$, it is clear that

$$|X[k]| \leq \sum_{n=0}^{N-1} |x[n]| < \sigma N$$

Therefore the maximum word growth in an FFT computation is limited. This upper bound of the word growth allows that immunity from overflow can be achieved by simply scaling the input data by σN. However, this method reduces the effective word length and therefore is not very efficient. An alternative method is to scale the data block only when necessary. For this purpose the word growth in one butterfly of the radix-2, DIT FFT algorithm is analyzed. The results suggest an effective method for overflow protection.

4.5.2.1. Word Growth in a Butterfly Operation

The computation of the FFT of N data points ($N = 2^m$) consists of $\log_2 N$ ($\equiv m$) passes with each pass consisting of $N/2$ butterfly operations of the form

$$A' = A + BW^k \qquad ..(4.12a)$$

$$B' = A - BW^k \qquad ..(4.12b)$$

The operands of the butterfly operation can be defined as, in both rectangular and polar coordinates,

$$A = A_R + jA_I = R_A e^{j\theta_A} \qquad ..(4.13a)$$

$$B = B_R + jB_I = R_B e^{j\theta_B} \qquad ..(4.13b)$$

$$W^k = W_R^k + jW_I^k = e^{j\theta_k} \qquad ..(4.13c)$$

where $j = \sqrt{-1}$.

As specified earlier, a data word can only represent a number in the interval $[-1,1)$, hence

$$-1 \le A_R, A_I, B_R, B_I, W_R^k, W_I^k < 1 \qquad ..(4.14)$$

Consequently, the magnitude of a complex number represented with two data words is less than or equal to $\sqrt{2}$, or

$$0 \le R_A, R_B \le \sqrt{2} \qquad ..(4.15)$$

Now consider the multiplication of B and W^k,

$$BW^k = R_B e^{j\theta_B} e^{j\theta_k}$$

$$= R_B\cos(\theta_B+\theta_k) + jR_B\sin(\theta_B+\theta_k) \quad ..(4.16)$$

which results in the rotation of B by θ_k. If $\theta_B + \theta_k$ is a multiple of 90°, B becomes purely real or purely imaginary after the rotation. This is crucial to the word growth analysis because the components of BW_k is no longer bound by the word limits:

$$-\sqrt{2} \le R_B\cos(\theta_B+\theta_k), R_B\sin(\theta_B+\theta_k) \le \sqrt{2} \quad ..(4.17)$$

From (4.12) and (4.16),

$$B_R' = A_R - R_B\cos(\theta_B+\theta_k) \quad ..(4.18a)$$

$$B_I' = A_I - R_B\sin(\theta_B+\theta_k) \quad ..(4.18b)$$

These equations can be combined with (4.14) and (4.16) to determine the limits of B_R' and B_I':

$$-1-\sqrt{2} \le B_R', B_I' \le 1+\sqrt{2} \quad ..(4.19)$$

This result shows that the absolute value of a component of B can grow from 1 to a maximum of $1+\sqrt{2}$, or approximately 2.41 times its original value. Since every data point is processed once in each pass, this result is valid for all passes.

Based on eqn. (4.19), the optimum overflow prevention strategy is to scale the data block before each pass so that the maximum absolute value of the real or imaginary part of the data block will be 1/2.41. This method, however, is not practical. An alternate solution which is implementable in BFP format is discussed

in the next subsection.

4.5.2.2. Implementation of the Scaler

A simple implementation of hardware division consists of a shifter which can divide the data by powers of 2. Two shifters are inserted in the real and imaginary data paths as shown in fig. 4.17. These shifters, implemented by Am25S10s, can shift the input data by 0, 1 or 2 bits, corresponding to division by 1, 2 or 4 respectively.

From the results of the previous section it is clear that a real or imaginary component can grow by more than 2 but less than 4 in any pass. Therefore, overflow can be prevented in a pass by keeping the absolute values of all components less than or equal to 0.25 before the pass. To achieve this condition, the data block must be tested after every pass and scaled down if the real or imaginary part of any data point is greater than 0.25. The scaling factor is determined by the following conditions:

- If any component is greater than 0.5 then scale by 4.
- If all components are smaller than 0.5 but some component is greater than 0.25 then scale by 2.

Hence the overflow prevention problem is reduced to design the logic circuit to generate these conditions.

As discussed earlier, a real number, x, is represented in two's complement fractional binary format as

$$x = -a_{15}2^0 + a_{14}2^{-1} + \cdots + a_0 2^{-15} \qquad ..(4.20)$$

where a_n is the n^{th} bit value which is 1 or 0. Following logic equivalences can be verified from eq. (4.20):

$$\begin{aligned} x \leq 0.5 &\equiv \overline{a_{15}} \bullet a_{14} \\ x < -0.5 &\equiv a_{15} \bullet \overline{a_{14}} \end{aligned}$$

where $\bullet$ represents the logical-and operation. Combining these equations yields

$$\begin{aligned} R &\equiv |x| \geq 0.5 \\ &\equiv (\overline{a_{15}} \bullet a_{14}) + (a_{15} \bullet \overline{a_{14}}) \\ &\equiv a_{15} \oplus a_{14} \end{aligned} \qquad ..(4.21)$$

where $+$ and $\oplus$ denote the logical-or and exclusive-or operations respectively. Similarly,

$$\begin{aligned} S &\equiv 0.25 \leq |x| \leq 0.5 \\ &\equiv \overline{R} \bullet (|x| \geq 0.25) \\ &\equiv \overline{R} \bullet (a_{15} \oplus a_{13}) \end{aligned} \qquad ..(4.22)$$

Equations (4.21) and (4.22) show a simple way to determine the range of a component. These tests must be repeated for the real and imaginary parts of all data points. The results, R and S, from each component are registered by a pair of flip-flops which detect a "1" at their respective inputs. The state equations for these flip-flops are

$$Y_1 \equiv Y_{1p} + R_R + R_I$$
$$Y_2 \equiv Y_{2p} + \left[Y_{1p} \bullet (S_R + S_I)\right] \quad ..(4.23)$$

where the subscript p denotes the previous state and subscripts R and I indicate real and imaginary tests respectively.

A practical approach is to test each data point as it is written back to the memory as a result of an FFT butterfly, rather than searching the entire data block after the pass is completed. In this method the flip-flops are clocked every time a data point is written into the memory. After the pass is completed, the outputs of these flip-flops are transferred to another pair of flip-flops which select the amount of shift (0, 1 or 2) to be performed by the shifters. As a result, each component will be scaled down by the same amount as it is read by the arithmetic processor for the next pass.

4.5.3. Software for Parallel Execution

The progress of a butterfly operation in the arithmetic processor is illustrated in Table 4.2. The register file operations are not shown to simplify the diagram, however some of the visible delays are associated with the register file. For example, the multiplier output is generated one cycle after the value of B is read because this value has to be written to the register file first. As observed from Table 4.2, the butterfly operation requires 9 cycles although each device is operational for only 4 cycles of this period. In order to achieve the efficiency figures derived in section 4.3.1, the butterfly operations have to be overlapped in a way that

CYCLE #	DATA BUSSES	REAL ALU	IMAG. ALU	MULTIPLIER
1	READ B			
2	READ A			
3				$B_R W_R$
4		$A_R'' = A_R + B_R W_R$		$B_R W_I$
5		$B_R'' = A_R - B_R W_R$	$A_I'' = A_I + B_R W_I$	$B_I W_I$
6		$A_R' = A_R'' - B_I W_I$	$B_I'' = A_I - B_R W_I$	$B_I W_R$
7		$B_R' = B_R'' + B_I W_I$	$B_I' = B_I'' - B_I W_R$	
8	WRITE B'		$A_I' = A_I'' + B_I W_R$	
9	WRITE A'			

Table 4.2. Progress of a butterfly operation

all devices will be operational at all times. Hence a repetition rate of 4 cycles is required.

The butterfly operation as shown in Table 4.2 can not be overlapped with a rate of 4 cycles because the ''Read B'' operation at the first cycle has to be repeated at the 5^{th} cycle and then at the 9^{th} cycle which conflicts with the ''Write A'' operation. The progress of the butterfly can be modified slightly, as shown in Table 4.3, to get rid of this problem. With this organization, the butterfly operations can be completely overlapped. Table 4.4 shows two overlapped butterflies which can be extended to fill all cycles. The operation of the whole processing unit with completely overlapped operations is shown in Table. 4.5.

CYCLE #	DATA BUSSES	REAL ALU	IMAG. ALU	MULTIPLIER
1	READ B			
2				
3	READ A			$B_R W_R$
4		$A_R'' = A_R + B_R W_R$		$B_R W_I$
5		$B_R'' = A_R - B_R W_R$	$A_I'' = A_I + B_R W_I$	$B_I W_I$
6		$A_R' = A_R'' - B_I W_I$	$B_I'' = A_I - B_R W_I$	$B_I W_R$
7		$B_R' = B_R'' + B_I W_I$	$B_I' = B_I'' - B_I W_R$	
8	WRITE B'		$A_I' = A_I'' + B_I W_R$	
9				
10	WRITE A'			

Table 4.3. Modified butterfly for overlapping

CYCLE #	DATA BUSSES	REAL ALU	IMAG. ALU	MULTIPLIER
1	READ B (1)			
2				
3	READ A (1)			$B_R W_R$ (1)
4		A_R'' (1)		$B_R W_I$ (1)
5	READ B (2)	B_R'' (1)	A_I'' (1)	$B_I W_I$ (1)
6		A_R' (1)	B_I'' (1)	$B_I W_R$ (1)
7	READ A (2)	B_R' (1)	B_I' (1)	$B_R W_R$ (2)
8	WRITE B' (1)	A_R'' (2)	A_I' (1)	$B_R W_I$ (2)
9		B_R'' (2)	A_I'' (2)	$B_I W_I$ (2)
10	WRITE A' (1)	A_R' (2)	B_I'' (2)	$B_I W_R$ (2)
11		B_R' (2)	B_I' (2)	
12	WRITE B' (2)		A_I' (2)	
13				
14	WRITE A' (2)			

Table 4.4. Two overlapped butterfly operations

CYCLE	Address Generator		Data Addr. PL		Coef. Addr. Register	DATA BUSSES	REAL ALU	IMAG. ALU	MULTIPLIER		
	Inst.	Output	Inst.	Output					X	Y	Output
1	COUNT		HOLD	A1 (-1)		READ A (-1)	B_R' (-2)	B_I'(-2)	B_R (-1)	sin (-1)	B_RW_R (-1)
2	HOLD	B	PUSH B	B2 (-2)		WRITE B' (-2)	A_R' (-1)	A_I' (-2)	B_I (-1)	sin (-1)	B_RW_I (-1)
3	HOLD	W		B1	LATCH	READ B	B_R'' (-1)	A_I'' (-1)	B_I (-1)	cos (-1)	B_IW_I (-1)
4		A	PUSH A	A2 (-2)		WRITE A' (-2)	A_R'' (-1)	B_I'' (-1)	B_R	cos	B_IW_R (-1)
5	COUNT (1)		HOLD	A1		READ A	B_R' (-1)	B_I' (-1)	B_R	sin	B_RW_R
6	HOLD	B (1)	PUSH B (1)	B2 (-1)		WRITE B' (-1)	A_R''	A_I'' (-1)	B_I	sin	B_RW_I
7	HOLD	W (1)		B1 (1)	LATCH (1)	READ B (1)	B_R''	A_I''	B_I	cos	B_IW_I
8		A (1)	PUSH A (1)	A2 (-1)		WRITE A' (-1)	A_R'	B_I''	B_R (1)	cos (1)	B_IW_R
9	COUNT (2)		HOLD	A1 (1)		READ A (1)	B_R'	B_I'	B_R (1)	sin (1)	B_RW_R (1)
10	HOLD	B (2)	PUSH B (2)	B2		WRITE B'	A_R'' (1)	A_I'	B_I (1)	sin (1)	B_RW_I (1)
11	HOLD	W (2)		B1 (2)	LATCH (2)	READ B (2)	B_R'' (1)	A_I'' (1)	B_I (1)	cos (1)	B_IW_I (1)
12		A (2)	PUSH A (2)	A2		WRITE A'	A_R' (1)	B_I'' (1)	B_R (2)	cos (2)	B_IW_R (1)
13	COUNT (3)		HOLD	A1 (2)		READ A (2)	B_R' (1)	B_I' (1)	B_R (2)	sin (2)	B_RW_R (2)
14	HOLD	B (3)	PUSH B (3)	B2 (1)		WRITE B' (1)	A_R'' (2)	A_I'' (1)	B_I (2)	sin (2)	B_RW_I (2)
15	HOLD	W (3)		B1 (3)	LATCH (3)	READ B (3)	B_R'' (2)	A_I'' (2)	B_I (2)	cos (2)	B_IW_I (2)
16		A (3)	PUSH A (3)	A2 (1)		WRITE A' (1)	A_R'' (2)	B_I'' (2)	B_R (3)	cos (3)	B_IW_R (2)

Table 4.5. Completely overlapped butterflies

The required time to compute an N-point FFT can now be estimated. An N-point FFT is performed in $\log_2 N$ passes with N/2 butterflies in each pass. Therefore, if the cycle period is t_c, the total time required for the FFT is

$$T_N = 4t_c\frac{N}{2}\log_2 N \quad \text{or,}$$

$$= 2t_c N\log_2 N \qquad ..(4.24)$$

In reality, the first and last two butterflies are not totally overlapped with the other butterflies because the pipeline has to be filled before the parallel execution and it has to be emptied after. These end effects have to be included in the total time for precision. It has been observed that the filling and emptying of the pipeline requires an additional 22 cycles, thus equation 4.24 becomes

$$T_N = (2N\log_2 N + 22)t_c \qquad ..(4.25)$$

4.5.4. Clock Requirements

The slowest device in the FFT processor is the memory, which is implemented with 4 Am9128s. Using this fact and the block diagram of the processor (fig. 4.17) the critical path on this system is indicated as the path for the *Read* operation, i.e. the path from the address register to the ALU. The delay on this path was calculated to be 112 ns (Table 4.6). To determine the minimum clock period the delay from the clock to the outputs of the pipeline register at the control unit (20 ns) must be added to this figure. Thus the minimum clock period was estimated to be 132 ns. However, the delays shown in Table 4.6 are maximum values and we were able to

DEVICE	PATH	PART NO.	DELAY
Addr. Reg.	sel.→output	Am29520	12 ns
Data Memory	Addr. setup	Am9128	70 ns
Bus Driver	input→output	74LS244	12 ns
Shifter	input→output	Am25S10	8 ns
Reg. File	data setup	Am29501	10 ns
TOTAL			112 ns

Table 4.6. Calculation of the delay for the READ operation

operate the FFT processor with a clock frequency of 8 MHz (125 ns period).

Equation 4.25 can now be determined numerically. For example, the time required to compute the discrete Fourier transform of a 1024 point data block is

$$T_{1024} = (2\times1024\times\log_2 1024 + 22)\ 125\ ns\ =\ 2.56\ ms$$

4.6. Summary

In this chapter, we have shown that a uniform structure for different processing units can be achieved by separating the address generator and the arithmetic processor. These blocks can then be operated concurrently improving overall speed and the performance of the unit. The architecture of the arithmetic processor is identical for different DSP algorithms because, in all cases, the computations of the *fundamental operation* is required. A particular DSP algorithm can be specified by defining its address generator.

An architecture for the arithmetic processor, suggested by AMD, was found to be optimal for repetitive computations of the fundamental operation. Overall

organization of the processing unit was discussed emphasizing parallel processing.

An FFT processor board, shown in fig. 4.18, was constructed to demonstrate the efficiency of the presented architecture. A very high performance level was achieved with the implementation of overlapped butterflies and concurrent real/imaginary operations. This processor completes one butterfly operation every four cycles with an experimentally determined cycle time of 125 ns.

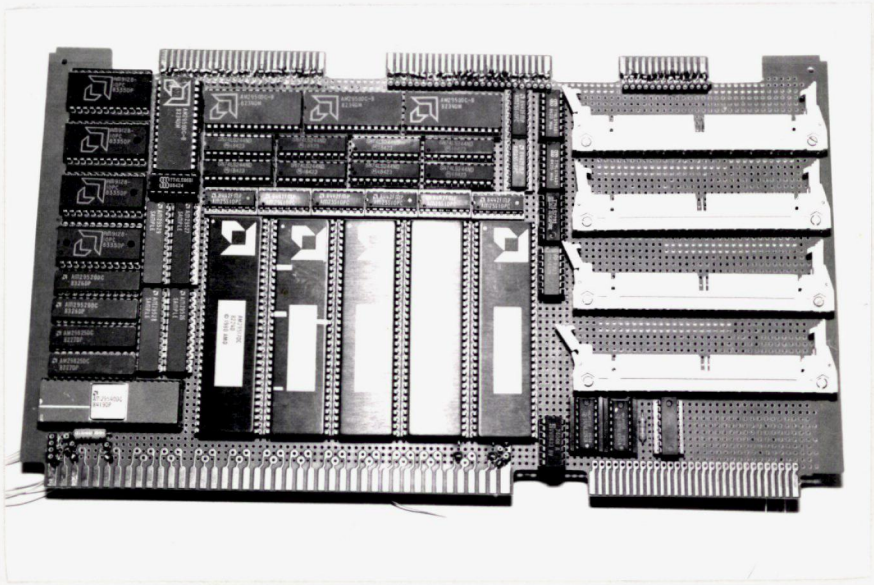


Fig. 4.18. FFT processor board

CHAPTER 5

CONCLUSIONS

The research towards the objective of designing the basic tools for microprogrammed implementations of DSP algorithms consisted of two phases:

1. Construction of the general purpose controller
2. Development of a DSP-suitable architecture

The results obtained from each phase are examined separately in the next two sections.

In the final section of this chapter, suggestions for improving various aspects of the proposed signal processing system are presented.

5.1. The General Purpose Controller

A versatile microprogrammable controller and development tool has been constructed. This unit was intended to be used during the development stage of the individual processing units within the signal processing system. It is capable of

1. Generating control signals which can be used by any processing unit in a 128-bit horizontal microword format.
2. Executing microprograms, which are stored in the host computer or the on-board memory, in continuous or single-step mode.
3. Transferring data blocks from the host computer to the processing unit and

back.

In addition, the modular design allows using the host-interface unit of the controller as the task manager. This unit is capable of downloading multiple RAM-based microprogrammable control units. A software library provides easy access to all functions of the controller from the host computer using simple commands.

Although the general purpose controller was designed to meet the requirements of DSP oriented implementations, its applications are not limited. For example, its potential application in regulating the speed of a rotating machine was recognized during the early development stage [22]. The controller is also in use by Nichols[8] to develop a processor for auto-regressive modeling using Burg's algorithm.

5.2. The Digital Signal Processing Architecture

In the second phase of this thesis, an architecture suitable for signal processing applications has been developed following suggestions made by AMD [19]. Three major points have been demonstrated:

(1) Most signal processing algorithms consist of repetitive computations of the fundamental operation with an addressing scheme unique to the algorithm. Hence, once a processing unit for one DSP operation is realized, another operation can be implemented by modifying only the address generator.

(2) The two data bus, two ALU, one multipler architecture suggested by AMD yields a very good performance level with a reasonable level of complexity.

(3) Very efficient, highly parallel processors can be achieved when sufficient data flow paths and complete control of all resources are available.

The proposed architecture have successfully been tested for the radix-2 FFT algorithm. To illustrate the performance of the FFT processor, let us consider the computation of 1024-point complex FFT. The required time for this calculation on the FFT processor was estimated to be 2.56 ms. In comparison, the TMS32010, a popular signal processing chip, requires 69.4 ms [23]. This tremendous improvement is a result of several techniques, which are

1. Parallel processing, which is reflected as overlapped butterfly operations, concurrent computations of the real and imaginary parts, and simultaneous addition and multiplications.

2. Pipelining, which improves the speed by isolating the delays in the individual components.

3. Hardware generation of data addresses, to avoid the significant amount of time required to calculate the addresses in software.

5.3. Recommendations for Future Research

The signal processing system described in chapter 1 is yet to be completed. As mentioned before, the processing units can be realized by designing an appropriate hardware address generator for the arithmetic processor developed in this thesis. The FFT processor board constructed during this research may be directly used in this system. However, since the FFT processor was designed simply as a test-bed for the architecture, the address generator of this board may be found inadequate for some applications. In particular, the input data is assumed to be pre-scrambled as required by the FFT algorithm. Hence there are no provisions for bit-reversed scrambling of the data block. Therefore, the addition of an address generator for data scrambling is recommended if this board is to be used later.

The interface between the processing units should be improved. Since the data flow is necessarily towards one direction in a pipelined system, uni-directional busses between consequent processors can be used instead of a system bus (fig. 4.11). Also the I/O registers shown in fig. 4.10 do not adequately isolate the processing units. Replacing these registers with first-in-first-out (FIFO) memories is recommended.

An important limitation of the architecture is the data word length. The 16-bit integer arithmetic may not provide sufficient precision for some applications. As discussed in chapter 4, the selection of this data format was dictated by the available components. Several new 32-bit parallel multipliers, such as Am29323, have been announced recently. The word length can therefore be extended to 32 bits as these products become available. The 32-bit ALU can be constructed by cascading four

Am29501 devices.

Another exciting new product is the Am29325, a 32-bit floating point ALU. When used with an external register file such as Am29334, also a recent announcement, this device may replace the Am29501s and the multiplier. This possibility is particularly interesting because the general outline of the "2-2-1" architecture can be preserved by using three Am29325s, possibly with one of them wired as a multiplier, tremendously increasing the flexibility, the adaptability and the precision of the resulting processing units.

REFERENCES

1. A.V. Oppenheim, *Applications of Digital Signal Processing,* Prentice-Hall (1978).

2. M.C. Pease, "An Adaptation of the Fast Fourier Transform for Parallel Processing," *J. Ass. Comput. Mach.* **15** pp. 252-264 (April 1968).

3. B. Gold and T. Bially, "Parallelism in Fast Fourier Transform Hardware," *IEEE Trans. Audio Electroacoust.* **AU-21** pp. 5-16 (Feb.1973).

4. K.J.M. Campbell, "A Microprocessor Based Spectrum Analyzer," *MSc Thesis*, Dept. of Elec. Eng., U. of Calgary, (1984).

5. J. Makhoul, "Linear Prediction: A Tutorial Review," *Proc. IEEE* **63** pp. 561-580 (Apr. 1975).

6. M.R. Smith, S.T. Nichols, R.M. Henkelman, and M.L. Wood, "Application of Parametric Modeling in Magnetic Resonance Imaging," *Submitted as a paper to IEEE Trans. Med. Images*, (1986).

7. J.D. Markel, "FFT Pruning," *IEEE Trans. Audio Electroacoust.* **AU-19, no. 4** pp. 305-311 (Dec. 1971).

8. S.W. Nichols, "Microprogrammed Implementation of AR Modeling," *MSc Thesis*, Dept. of Elec. Eng., U. of Calgary, (expected completion July, 1986).

9. M.V. Wilkes, W. Renwick, and D.J. Wheeler, "The Design of the Control Unit of an Electronic Digital Computer," *Proc. of IEE*, pp. 121-128 (June 1958).

10. M.V. Wilkes, "The Growth of Interest in Microprogramming: A Literature Survey," *Computing Surveys* **1** pp. 139-145 (Sept.1969).

11. S.S. Husson, *Microprogramming: Principles and Practices,* Prentice-Hall (1970).

12. D.K. Banerji and J. Raymond, *Elements of Microprogramming,* Prentice-Hall (1982).

13. G. Hope, *Integrated Devices in Digital Circuit Design,* Wiley (1981).

14. M. Andrews, *Principles of Firmware Engineering in Microprogram Control,* Computer Science Press (1980).

15. Data Book, *Bipolar Microprocessor Logic and Interface,* Advanced Micro Devices (1985).

16. H. Orbay and M.R. Smith, "A Development Tool for Microprogrammable Systems: General Purpose Controller," *Report #28 CO 85*, Dept. of Elec. Eng., U. of Calgary, (Sept. 1985).

17. M.R. Smith, "A METAASSEMBLER for developing microwords for a microprogrammed architecture," *Report, #19 PS 85*, Dept. of Elec. Eng., U. of Calgary, (March 1985).

18. B.A. Bowen and W.R. Brown, "Signal Processing and Signal Processors," *VLSI Systems Design For Digital Signal Processing, Vol. 1,* Prentice-Hall,

(1982).

19. J.W. Locke, *Designing Digital Signal/Array Processors with the Am29500 Family,* Advanced Micro Devices (1984).

20. A.V. Oppenheim and C.J. Weinstein, ''Effects of Finite Register Length in Digital Filtering and the Fast Fourier Transform,'' *Proc. IEEE* **60** pp. 957-976 (Aug.1972).

21. A.V. Oppenheim and R.W. Schafer, *Digital Signal Processing,* Prentice Hall (1975).

22. M.R.Smith, T.Grant, and H.Orbay, ''A Development System for High Speed Microprogrammable Sequential Controllers,'' *presented at Compint 85, Montreal, Quebec,* (Sept. 1985).

23. C.S. Burrus and T.W. Parks, *DFT/FFT and Convolution Algorithms,* Wiley (1985).