Application of Parallel Processing Architecture to Computer Music Synthesis

by

Thomas R. Hegg, Jr.

Submitted in partial fulfillment of the requirements for the Degrees of

Bachelor of Science

and

Master of Science

at the

Massachusetts Institute of Technology

July 1983

© Thomas R. Hegg, Jr.

The author hereby grants to MIT permission to reproduce and to distribute copies of this thesis document in whole or in part.

Signature of Author ................................................................. Thomas R. Hegg, Jr.

Department of Electrical Engineering and Computer Science, July 7, 1983

Certified By .................................................................

Thesis Supervisor (Academic)

Certified by .................................................................

Company Supervisor (Cooperating Company)

Accepted by .................................................................

Chairman, Department Committee on Graduate Students

Archives

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

OCT 5 1983

LIBRARIES
Application of Parallel Processing Architecture to Computer Music Synthesis

by

Thomas R. Hegg, Jr.

Submitted to the
Department of Electrical Engineering and Computer Science
on July 7, 1983 in partial fulfillment of the requirements
for the Degrees of Bachelor of Science and Master of Science

Abstract

Computer music synthesis refers to the technique of generating music by means of a computer program which produces the digitized samples of the desired audio waveform. Over the past several years, many systems have been developed with the desire to perform in real-time the vast amounts of computation these programs require; these systems have been hampered, however, either by the inflexibility they offer to the implementation of new synthesis techniques, or by the computational limits imposed by their architecture.

This thesis investigates an innovative parallel processing approach which seeks to exploit the recent introduction of high-speed VLSI digital signal processing chips. A parallel tree architecture is developed, along with a special synthesis language designed to facilitate its implementation on the parallel processing architecture. Finally, a compilation strategy is presented, which efficiently partitions the task among the processing elements, schedules inter-processor communication, and corrects for delays incurred as a result of such communication, thus ensuring the computational equivalence of the parallel task to the original program specification.

Thesis Supervisor: Jack B. Dennis

Company Supervisor: Surendar S. Magar
Acknowledgments

I would like to thank Professor Dennis for his support, encouragement, and above all his patience during the preparation of this thesis. I would also like to thank Surendar Magar and all the folks at Texas Instruments for providing me with the opportunity to carry out this research, and for creating the friendly atmosphere that made this work possible. Special thanks must go to Professor Leiserson for his timely help and revelations. Conversations with Curtis Abbott at Lucasfilm and Miller Puckette and John Stautner at M.I.T.'s Experimental Music Studio were especially helpful in clarifying certain issues that arose during the course of the work. And finally, many thanks to my sister Susan, for all those fine spaghetti dinners....
To my Mother and Father -- who else?!
Table of Contents

Chapter One: Introduction

1.1 An Introduction to Computer Music Synthesis Techniques 9
   1.1.1 The Music 11 Synthesis Language 9
   1.1.2 Synthesis Techniques 12
1.2 Brief Overview of Past Research and Solutions 16
1.3 The Proposed Parallel Architecture 17

Chapter Two: Parallel Processing Architecture 19

2.1 Processing Element Architecture 19
   2.1.1 TMS320 Features 19
   2.1.2 Synchronization Signals Generated by the PE for Synchronous Operation 22
   2.1.3 Implementation of Three Rates of Computation 23
   2.1.4 Inter-Processor Communication (IPC) Ports 24
2.2 Single Cluster Configuration - Overview of Operation 24
   2.2.1 Synchronization of PE's in a Single Cluster Configuration 24
   2.2.2 Inter-Processor Communication in the Single Cluster Configuration 27
   2.2.3 Implications of the Synchronous Architecture on the Compiler 29
   2.2.4 Limits on the Number of PE's in a Single Cluster 33
2.3 Extension to Multiple Cluster (Tree) Architecture 33
   2.3.1 IPC Considerations 35
   2.3.2 Balanced Tree Architecture and Considerations 35

Chapter Three: Music 320 Language Definition 38

3.1 Introduction 38
3.2 Primitive Operators 38
   3.2.1 Arithmetic Operators 39
      3.2.1.1 Signed Fraction Arithmetic 39
      3.2.1.2 Integer Arithmetic 40
   3.2.2 Table Access Primitives 40
      3.2.2.1 Table Memory Read Operations 40
      3.2.2.2 Table Memory Read-then-Write 41
   3.2.3 MERGE Operations 41
3.3 Variable Types 42
   3.3.1 Definition of T-Variables 42
   3.3.2 Table Object Definition 43
      3.3.2.1 Read-Only Table Object Declaration 43
      3.3.2.2 Read-Write Table Object Definition 44
   3.3.3 Table Attributes 44
3.4 Rules Governing the Use of Primitive Operators and Variables 45
3.5 Music 320 Structure
  3.5.1 Macro Definition Section
  3.5.2 Instrument Definition Section

Chapter Four: Task Partitioning Compiler
  4.1 Graph Representation of the Music 320 Synthesis Program
    4.1.1 Translation of the Music 320 Source Program into its Graph Representation
    4.1.2 Properties of the Synthesis Graph
  4.2 Task Partitioning Algorithm
    4.2.1 Development of Partitioning Heuristics for Synthesis Programs
    4.2.2 Partitioning Algorithm for the Single Cluster Architecture
    4.2.3 Partitioning Algorithm for the Balanced Tree Architecture
  4.3 Scheduling Routine for Inter-Processor Communication
  4.4 Task Equivalency Routine
    4.4.1 K- and I-Rate Delay Compensation
    4.4.2 Initialization of Delayed A-Rate Computation Nodes
  4.5 Final Subtask Compilation
    4.5.1 Final Subtask Definition
    4.5.2 Ordering of A-, K-, and I-Rate Nodes
    4.5.3 Assignment of TMS320 Data Memory Locations to Nodes
    4.5.4 Assignment of Table Memory Locations to Table Objects
    4.5.5 Generation of A-, K-, and I-Rate Assembly Blocks
      4.5.5.1 A-Rate Code Block Generation and Final Scheduling of IPC
      4.5.5.2 K- and I-Rate Code Generation
    4.5.6 Final Assembly Code Generation
    4.5.7 Linking to the Host Processor

Chapter Five: Conclusions and Suggestions for Further Research
  5.1 Discussion of the Parallel Processing System
    5.1.1 Problems with the Parallel Processing Architecture
    5.1.2 Problems with the Music 320 Synthesis Language
    5.1.3 Problems with the Compilation Strategy
  5.2 Conclusion

Appendix A: Min-Max Tree Partitioning Algorithm

References
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Example Music 11 Instrument</td>
<td>12</td>
</tr>
<tr>
<td>1-2</td>
<td>Subtractive Synthesis</td>
<td>13</td>
</tr>
<tr>
<td>1-3</td>
<td>Additive Synthesis</td>
<td>13</td>
</tr>
<tr>
<td>1-4</td>
<td>FM Synthesis</td>
<td>15</td>
</tr>
<tr>
<td>1-5</td>
<td>Waveshaping Synthesis</td>
<td>15</td>
</tr>
<tr>
<td>2-1</td>
<td>Processing Element Architecture</td>
<td>20</td>
</tr>
<tr>
<td>2-2</td>
<td>TMS320 Internal Architecture</td>
<td>21</td>
</tr>
<tr>
<td>2-3</td>
<td>K-Frame Process</td>
<td>23</td>
</tr>
<tr>
<td>2-4</td>
<td>Single Cluster Architecture</td>
<td>25</td>
</tr>
<tr>
<td>2-5</td>
<td>PE Subtask for Synchronous Parallel Computation</td>
<td>26</td>
</tr>
<tr>
<td>2-6</td>
<td>IPC Scheduling and Slot Timing</td>
<td>28</td>
</tr>
<tr>
<td>2-7</td>
<td>Indeterminacy of Delays in Uncorrected IPC</td>
<td>31</td>
</tr>
<tr>
<td>2-8</td>
<td>IPC Delay Correction</td>
<td>32</td>
</tr>
<tr>
<td>2-9</td>
<td>Tree Architecture</td>
<td>34</td>
</tr>
<tr>
<td>2-10</td>
<td>Intermediate IPC Links in the Tree Architecture</td>
<td>36</td>
</tr>
<tr>
<td>3-1</td>
<td>Variable Coercion</td>
<td>47</td>
</tr>
<tr>
<td>3-2</td>
<td>Example Music 320 Program</td>
<td>54</td>
</tr>
<tr>
<td>4-1</td>
<td>A Directed Graph</td>
<td>58</td>
</tr>
<tr>
<td>4-2</td>
<td>Graph Representation of a Music 320 Fragment</td>
<td>60</td>
</tr>
<tr>
<td>4-3</td>
<td>Graph Representation of Constants, Parameters, and Table Objects</td>
<td>61</td>
</tr>
<tr>
<td>4-4</td>
<td>Graph Representation of Forward and Table Attribute Dependencies</td>
<td>62</td>
</tr>
<tr>
<td>4-5</td>
<td>A Directed Tree Graph</td>
<td>66</td>
</tr>
<tr>
<td>4-6</td>
<td>Graph Representation of the OSCIL Macro</td>
<td>68</td>
</tr>
<tr>
<td>4-7</td>
<td>Typical Orchestra File Organization</td>
<td>69</td>
</tr>
<tr>
<td>4-8</td>
<td>Placing Cuts on Tree Graph Edges to Isolate Subtasks</td>
<td>70</td>
</tr>
<tr>
<td>4-9</td>
<td>Possible Spanning Tree for the Graph in Figure 4-1</td>
<td>73</td>
</tr>
<tr>
<td>4-10</td>
<td>IPC Node Insertion</td>
<td>78</td>
</tr>
<tr>
<td>4-11</td>
<td>Compensating for IPC Delays</td>
<td>80</td>
</tr>
<tr>
<td>4-12</td>
<td>Implementation of Correction FIFO's</td>
<td>90</td>
</tr>
</tbody>
</table>
Chapter One

Introduction

Computer music synthesis refers to the technique of generating music by means of a computer program which produces the digitized samples of the desired audio waveform. Since its conception in 1957 by Max Mathews at Bell Labs [14, 18], however, it has been haunted by one over-riding problem: namely, that the long turn-around time involved in running a sound synthesis program on a standard computer system severely limits any sort of interactive feedback with the machine. Since the sound samples are computed at a considerably lower rate than the rate at which they must be sent to the digital-to-analog converters to be heard, they must be stored in some high-density/fast-access storage medium for later playback. As a result, composers who wish to "tweek" certain synthesis parameters to achieve some desired effect may find themselves waiting as much as one minute for every second of sound produced until the results of the adjustment may be heard. Not only does this impair the composer's ability to create a desired sound, but it also immediately rules out the possibility of live performance for a computer synthesis work, in which the parameters of the piece may be controlled by actual performers.

Only in the last several years has this problem been addressed in the development of special purpose hardware for digital real-time music synthesis. These architectures have typically taken on two forms: one type of architecture involves time-multiplexing hardware realizations of popular sound generating and modifying blocks to provide many times the original function; the other is based on fast bit-slice microprocessors whose data paths have been optimized for signal processing computation. As will be seen in Section 1.2, each of these approaches exhibits its own deficiency: the former is often plagued by its inflexibility (since the signal processing is done in hardware), while the latter is limited in the amount of signal processing it is capable of performing, even at moderate sampling rates. These architectures effectively situate themselves at opposite ends of a flexibility/capability trade-off curve.

With the advent of VLSI digital signal processing chips, most notably the Texas Instruments TMS320, a solution which avoids such a trade-off is not only possible, but also economically attractive. By taking advantage of the high degree of parallelism inherent in audio signal processing
algorithms, multiple configurations of processing elements based on these chips would afford a powerful and yet flexible means of real-time synthesis. This thesis develops such an approach, both in the realm of a parallel processing architecture for music synthesis, and in the realm of a compiler for a music synthesis language which is capable of efficiently partitioning the task among the individual processing elements.

In order to provide the necessary background information surrounding the topic of real-time digital music synthesis, the next two Sections will be devoted to introducing the computational techniques used in computer music synthesis, and to briefly reviewing past research and solutions to the problem.

1.1 An Introduction to Computer Music Synthesis Techniques

This Section will summarize two areas of computer music synthesis which will be of relevance for this thesis: the structure of the synthesis program, and the computational techniques used by synthesists to produce sounds of varying complexity.

1.1.1 The Music 11 Synthesis Language

An idea of the structure of the software environment used in most computer music compositions may be gained by examining that of the Music 11 sound synthesis language [22]. A Music 11 program consists of an orchestra file, which specifies the signal processing aspect of the sound generation, and a score file, which, as its name implies, contains a list of commands which control the timing of note events. Since the orchestra program represents the bulk of the computational burden, we shall concentrate on how the signal processing for music synthesis is specified in Music 11.

The orchestra file is divided into two sections. The first is a header section, which defines the sampling rate, the control rate (to be explained), and the number of channels of sound output. The second is the instrument definition section. Instruments are defined in the following format:
INSTR <instrument id number>

{body of instrument definition}

ENDIN

The instrument id number serves to identify the instrument for the purposes of event initialization from the score file. An event is a communication from the score to the orchestra signalling some musical change (e.g. a new note, a change in the amount of reverberation). A score command would then consist of an instrument number, the time (in seconds) at which the instrument is to begin the event, the length of the event, and any other expressive parameter the composer may wish to include (e.g. the pitch, timbre, and loudness of the event). These parameters are passed to the instrument block by p variables (e.g. p3, p6, etc.).

The body of an instrument definition is built by commands which implement popular signal processing functions, which are known as unit generators. Typical unit generators include oscillators, filters, envelope generators, adders, multipliers, delay lines, and reverberation units. Unit generators may be thought of as blocks which may be interconnected through the use of signal variables. In Music 11 there are three types of signal variables, designated by a single-letter prefix, which correspond to three different rates of computation:

- A-rate variables (e.g. a1, a4, a12) are variables which are re-computed at the audio sampling rate.

- K-rate variables (e.g. k1, k10) are variables which are computed at the slower control rate, and are used to control slowly varying aspects of the audio signal (such as loudness and timbral contours).

- I-rate variables (e.g. i2, i8) are variables which are computed only once at the beginning of an event (i.e. as a result of an event initialization from the score), and are used to perform pre-computations on those variables needed throughout the duration of the event.

Unit generators are implemented in a straightforward fashion. Three typical unit generator commands found in Music 11 are described below. The oscillator unit generator is the most handy way to produce a signal to begin music synthesis. The general format for the oscillator in Music 11 is
xout  OSCIL  xamp, xfreq, itbl

where  xout is the variable representing the output of the oscillator,
xamp is the amplitude of the oscillator output,
xfreq is the frequency (in Hz) of the oscillator output,
and  itbl is a table (or array) originating in the score file
which contains one cycle of the desired waveshape.

Note that in a Music 11 command, the output of a unit generator is listed first, followed by the
command name and the list of input variables. Another popular function is the second-order
recursive filter, which is useful for accentuating desired harmonics in a waveform. Its format is as
follows:

xout  RESON  xin, kcf, kbw

where  xin is the input to the filter,
kcf is the center frequency of the filter,
and  kbw is the bandwidth of the filter.

Other unit generators are mainly used to generate signals which are used to control certain
slowly-varying aspects of a sound during synthesis. One very simple such command in Music 11 is
the LINE generator, which produces an output which progresses linearly from a designated starting
value to a final value in a specified amount of time.

xout  LINE  istart, idur, iend

where  istart is the starting value of the line output,
       idur is the time it takes to go from istart to iend,
       iend is the final value of the line output.

As an example, let us consider the simple instrument in Figure 1-1, where the output of an
oscillator is connected to a filter, the output of which is sent to the sound storage device for later
playback. The variable il, which is computed once at the beginning of each note initiated by the
score, forces the center frequency of the filter to be twice the fundamental frequency of the oscillator
output; furthermore, the bandwidth is equal to the fundamental frequency. Thus, the filter
accentuates the second harmonic of the oscillator output. The LINE generator shapes the amplitude
envelope to begin abruptly, and to linearly die down to zero over the duration of the note.

INSTR 1

; The following variables are sent from the score:
; ;
; t1 - table defined in the score which contains the waveshape
; for the oscillator,
; p3 - controls the duration of the note.
; p4 - controls the loudness of the note.
; p5 - controls the pitch of the note.
;
; i1 = p5 * 2
k1 LINE p4, p3, 0
a1 OSCIL k1, p5, t1
a2 RESON a1, i1, p5
OUT a2

ENDIN

Figure 1-1: Example Music 11 Instrument

1.1.2 Synthesis Techniques

There are currently four main synthesis techniques in popular use: these are, by name, subtractive synthesis, additive synthesis, FM synthesis, and non-linear or "waveshaping" synthesis. All are for the most part concerned with ways to expressively control the harmonic content of a pitch. They are considered separately below.

Subtractive synthesis simply involves using various filtering techniques to attenuate certain harmonics and accentuate others from an already harmonically-rich source; as a matter of fact, the example Music 11 program described above is a subtractively synthesized instrument. The general structure for subtractive synthesis can therefore be summarized by the block diagram in Figure 1-2, where the output of an oscillator feeds the input to a filter.
Figure 1-2: Subtractive Synthesis

Figure 1-3: Additive Synthesis
Additive synthesis, on the other hand, uses the summed output of a number of sine-wave oscillators to generate a signal with the desired timbral characteristics, as is shown in Figure 1-3. This is probably the most powerful synthesis technique, since each harmonic may be individually controlled, both in amplitude and in frequency. Unfortunately, it is also the most expensive in terms of computational resources, and is therefore not as popular as the other techniques. It is most often used as a method for imitating the voices of traditional instruments.

FM synthesis was first brought to the attention of computer music composers by John Chowning [4]. It involves frequency modulating the output of one oscillator with another oscillator, as is shown in Figure 1-4. If both are sine-wave oscillators, then the output contains harmonics spaced about the carrier frequency at integer multiples of the modulating frequency, whose amplitudes are determined by the set of Bessel functions. If the modulating frequency is an integer multiple of the carrier, then the harmonics produced will be harmonically related to a perceived fundamental; if not, then a complex, inharmonic spectrum will result, producing a bell-like tone. The harmonic structure may be controlled in a predictable manner by varying the amount of frequency modulation. Because of its computational simplicity and the wide range of spectra it can produce, FM synthesis has become a very popular synthesis technique.

The last synthesis technique to be described is known as non-linear or "waveshaping" synthesis. It was developed almost simultaneously by Le Brun [11] and Arfib [1]; a good summary of the technique may be found in [17]. This approach usually involves passing a sine-wave source through a non-linear transfer function, effectively creating higher harmonics through distortion. The number and relative amplitudes of the harmonics for a sine-wave of fixed amplitude may be determined mathematically if the set a Chebyshev polynomials is used to construct the transfer function. The harmonic structure becomes less predictable, however, as the amplitude of the input sine wave changes; for inputs other than sine waves, or transfer functions other than Chebyshev polynomials, mathematical solutions become intractable. In spite of, or perhaps because of, this unprdictability, waveshaping synthesis has become very popular. This is due, at least in part, to its computational simplicity, since the non-linear transfer function may be implemented as a look-up table in a computer's memory, as shown in Figure 1-5.

This concludes our brief overview of computer synthesis tools and techniques. Though it is by no means an adequate survey over the whole of the field, it is only intended to provided the necessary background for the discussion of some of the issues raised in this thesis; more detailed information
Figure 1-4: FM Synthesis

Figure 1-5: Waveshaping Synthesis
may be found in the references cited and in the pages of the Computer Music Journal [5]. We may therefore turn our attention to the problem of real-time synthesis.

1.2 Brief Overview of Past Research and Solutions

Previous approaches to real-time digital music synthesis have concentrated on the hardware implementation of a single functional unit, which is then time-multiplexed to provide many times the original function. This type of system can range in relative complexity from the 4C machine designed by Pepino de Giugno at IRCAM [15], to the Systems Concepts digital synthesizer designed by Peter Samson [19], which is capable of implementing 256 oscillators and 128 "modifiers" (filters, mixers, etc.) at just under a 20KHz sampling rate. Both of these designs, however, suffer from two basic limitations. The first is that the range of signal processing chores that they are capable of implementing is limited in their respective hardwares, meaning that any synthesis technique not included in the design will run very inefficiently on these machines, if at all. For example, the Systems Concepts synthesizer was designed before the introduction of waveshaping synthesis; consequently, the implementation of the technique is clumsy, and uses up a fair number of resources [13]. The second limitation is that both of these machines require constant monitoring and updating by a host processor during the course of an event, since they have no way themselves of making any decisions based on their current state. This can present very serious problems for real-time control [16].

In answer to these difficulties, several bit-slice based processors have been built. The most commercially visible is the Digital Music Systems DMX-1000 [23], which offers all the flexibility of a high-speed minicomputer for the task of real-time music synthesis. The machine is capable of executing 253 instructions during a 50 μS sampling period (20 KHz sampling rate), and supports conditional operations, but no branching. The drawback here is that the processing capacity of the machine may easily be exceeded for many applications. For example, it is only capable of implementing (at a 20 KHz sampling rate) a maximum of 16 oscillators with envelope control. Therefore, it is really not powerful enough to perform in real-time the large, computationally burdensome orchestra programs that computer music composers run on slower mainframe computers. Moreover, it does not seem as though any single processor will ever attain the speed necessary to handle the vast amounts of signal processing required by these composers. It is only natural, then, to start looking at multiple-processor architectures to take advantage of the high degree
of parallelism inherent in music processing.

Though little has actually materialized here, much work has centered on hypothesizing what form a multiple-processor music system should assume. Cesari [3] notes that the program specification for music synthesis may be represented by a data flow graph, and proposes a real-time system based on the data flow architecture developed by Jack Dennis at M.I.T.'s Laboratory for Computer Science [6]. Because the architecture is relatively new, however, it seems unlikely that it will find its way into the hands of computer music composers for a number of years. Kahrs [8], in his discussion of the prospects of Very-Large-Scale Integration (VLSI) for computer music synthesis, suggests that each processing element in a multiprocessor configuration could be customized to emulate a popular signal processing algorithm; a variety of these building blocks would then be placed in a rectangular array, and each block could communicate with its four neighbors. He cites this as an application of Kung's work on systolic algorithms for signal processing [10]. Unfortunately, the doctrine of "designing the processor for the algorithm" denies the user the flexibility of complete programmability, with the consequence that computational resources cannot be reallocated depending on varying need. Moreover, the recent introduction on the market of several general-purpose VLSI digital signal processing chips further motivates research into how multiple configurations of processing elements based on these chips can be implemented, and how the music synthesis task would be divided among them.

1.3 The Proposed Parallel Architecture

The parallel architecture developed in this thesis attempts to solve many of the problems encountered by previous designs. The architecture is suited to performing all the computation associated with an orchestra file; a single, separate host processor can send commands to any processing element in the parallel processing environment, either to initiate a new note event, or to update a synthesis parameter in real-time. Since the communication between the host (or "score") processor and the parallel processing environment takes place on the event level, the amount of communication between the two is minimized. Furthermore, the architecture is programmable at a very basic level, meaning that new unit generators may be constructed and new synthesis techniques implemented as the need for them arises, in a straightforward manner.

The following Chapters describe the architecture, implementation language, and compilation
strategy for the parallel processing system. Chapter 2 develops a tree architecture based on the Texas Instruments TMS320 digital signal processing chip, and derives some of the implications of using such an architecture for real-time synthesis. Chapter 3 presents an orchestra description language called Music 320, which is well-suited for implementation on the parallel architecture, and is structured to facilitate the task-partitioning compilation strategy to be described in Chapter 4. The compilation strategy not only distributes the task efficiently among processing elements, but it also modifies the subtasks to compensate for the delays incurred as a result of inter-processor communication, thus ensuring the computational equivalency of the partitioned task to the Music 320 source program specification. Chapter 5 concludes with an analysis of some of the weak points of system, and describes areas of future research.
Chapter Two

Parallel Processing Architecture

This chapter deals with the architecture of the parallel processor, and examines the ramifications of implementing music synthesis programs in real-time in the parallel processing environment. The exposition will be from the ground up: Section 1 develops the architecture of a single processing element; Section 2 describes how a number of processing elements may be organized to construct a single-cluster parallel architecture; and finally, Section 3 shows how the single cluster architecture may be expanded to facilitate any number of processing elements in a tree processor configuration.

2.1 Processing Element Architecture

The basic building block of the parallel architecture is the processing element (PE). A block diagram of the PE is shown in Figure 2-1. The PE is organized around the Texas Instruments TMS320 digital signal processing chip [21], which is capable of executing 5 million instructions per second. The PE receives commands from the host processor through a two-port command buffer memory. The PE is also equipped with up to 64K words of table memory, which is used for storing oscillator waveshapes and implementing long audio delay lines. Finally, the PE may communicate with other PE's in the architecture through two inter-processor communication (IPC) ports. The use of the two IPC ports will become apparent when the single cluster machine architecture is described in a later section. For now, though, let us examine the main features of the TMS320 as they affect the synchronization of the parallel architecture.

2.1.1 TMS320 Features

An internal block diagram of the TMS320 is shown in Figure 2-2. Several features are of interest here.

- Capable of addressing 4K words of program memory. This is not a lot, but music synthesis programs are short.
Figure 2-1: Processing Element Architecture
Figure 2-2: TMS320 Internal Architecture
- 144 words of data memory. This on-chip memory is used to hold the signal variables of the synthesis program.

- Capable of addressing 8 I/O ports. These are used to interface with the command buffer, table memory, and two IPC ports. I/O is handled directly between the I/O ports and the data memory (that is, not through the accumulator); this is a handy feature which will allow us to perform IPC anywhere in the object code without having to worry about the state of the rest of the chip.

- External reset pin. A low signal on this pin forces the 320 program counter to zero; when the pin is returned to the high state, program execution starts at location zero within one clock cycle. This feature will be used to synchronize the parallel operation of the PE's.

- Data shift feature. This is a single cycle instruction which shifts data in the data RAM to the next higher data memory location.

- Two auxiliary loop counter registers. In conjunction with a "branch on loop counter not 0" instruction, these allow efficient coding of iterative computations.

Other features of the chip's internal architecture make it well-suited for performing real-time music synthesis. The on-chip 16 by 16 signed integer multiplier returns a 31-bit result in one cycle, allowing for a register-to-register multiply time of four cycles. The ALU may perform a register-to-register addition in three cycles, with the added option of overflow protection, which returns the highest or lowest signed integer as a result of an overflow or underflow, respectively. The computational power of a single PE is therefore comparable to a bit-slice processor like the one used in the DMX-1000 [23] (see Section 1.2).

2.1.2 Synchronization Signals Generated by the PE for Synchronous Operation

The parallel architecture to be described in the following Sections is synchronous in operation. Several provisions in the PE architecture are provided for this purpose.

- External clock input. This allows the instruction cycle time of all PE's to be synchronized. All PE's in the parallel architecture will operate from the same global clock.

- K-FrameStart input. Derived from the reset pin on the TMS320, this resets the PC of the TMS320 to 0, which will be the origin address of the k-frame program, described in the next Section.

- K-FrameDone acknowledge output. One bit of one TMS320 output port is used to signal completion of the k-frame routine initiated by the K-FrameStart signal. This is an open-collector output which acknowledges on active high.
- K-FrameDone acknowledge input. One bit of one TMS320 input port is used to propagate K-frameDone acknowledge signals from other PE's. K-FrameDone outputs from several PE's can be connected to a single input, which contains the pull-up resistor for the open-collector outputs. When several K-FrameDone outputs from other PE's are connected to the same K-FrameDone input of a single PE, they are effectively "wire-anded". The use of the K-FrameDone input and output signals is described in more detail in the next Section.

2.1.3 Implementation of Three Rates of Computation

The way in which a single PE may implement three rates of computation may now be described. Recall that the parallel processing environment must support computation at the sampling rate, at the slower control rate (k-rate), and at the beginning of a note event (i-rate). This is accomplished by structuring the computation to produce output samples in blocks corresponding to the number of audio-rate samples per control period. This process will be referred to as a k-frame; a program representing a k-frame process is shown in Figure 2-3.

Procedure ComputeK-Frame;

{KSMPS := A-Rate / K-Rate}

Begin
For N := 1 to KSMPS do
  begin
    ComputeA-RateVariables;
    OutputSamplesToD/A
  end

For M := 1 to NumberOfInstruments do
  If NewCommandFromHostForInstr(M)
    then begin
      FetchParametersFromCommandFIFO
      ComputeI-RateVariablesForInstr(M)
      InitializeVariables
    end
  else
    ComputeK-RateVariablesForInstr(M)

End.

Figure 2-3: K-Frame Process

23
The k-frame process begins by first computing KSMPS (= a-rate/k-rate) a-rate samples, and then polls the command buffer to see if a new set of parameters has been sent from the host processor. If so, a routine loads the parameters from the command buffer into the PE's table memory. Then for each instrument running on the PE, either a new set of k-rate variables is computed, or an initialization routine is run which computes the i-rate variables for that instrument and initializes any a- or k-rate variables specified in the synthesis program.

2.1.4 Inter-Processor Communication (IPC) Ports

Each PE is furnished with two IPC ports, each of which provides the necessary interface between a PE and an IPC bus. Each IPC port has timing associated with it which serves to broadcast a variable to another PE along a shared IPC bus. The unit of time in which a variable is placed on the bus will be called the slot time. The slot time will last for four instruction cycles.

2.2 Single Cluster Configuration - Overview of Operation

In order to facilitate the discussion of the architecture of the full parallel processing environment, it would be worthwhile to first describe the operation of a single-cluster parallel processor. A single cluster machine is shown in Figure 2-4. PE's communicate with each other using a single shared IPC bus. One PE has been designated the "father" PE - this PE uses its spare IPC port to transfer audio sound samples to a FIFO buffer and D/A conversion equipment.

Assuming for the moment that all PE's have been given a part of a synthesis task in the form specified in Figure 2-3, proper operation of the single cluster architecture centers on two issues:

- how the PE's are synchronized to perform the overall task in parallel, and
- how inter-processor communication is accomplished among PE's.

2.2.1 Synchronization of PE's in a Single Cluster Configuration

Synchronous operation of the PE's in Figure 2-4 is achieved by running all PE's from a single global clock source, and by tying all K-FrameStart inputs together, thus forcing the K-Frame computations of all PE's to begin simultaneously. The global K-FrameStart signal is generated by a command from the output FIFO, which initiates a k-frame computation when it is almost empty.
Figure 2-4: Single Cluster Architecture
The initiation of a new k-frame process is conditioned by the status of the father PE’s K-FrameDone output; if the FIFO attempts to start a new k-frame process without the consent of the father PE’s K-FrameDone output, then a system fault occurs. Output audio samples are removed from the FIFO at a rate determined by a separate sampling rate clock, which allows the real-time synthesis to be performed at less-than-maximum sampling rates.

Since the sub-task code blocks of the PE’s are probably of unequal size, the K-FrameDone signal of the father-PE must not be sent until it has been ascertained that the k-frame computation has been completed on each PE. This is accomplished with the K-FrameDone input and output signals generated by each PE in conjunction with the extra routine placed at the end of a k-frame code block shown in Figure 2-5. As each son PE completes its k-frame process, it checks to see if the K-FrameDone input is high - which it will be, as no other PE’s are attached to it. If so, it signals (high) its K-FrameDone output to the father PE. Since the open collector outputs of the son PE’s are effectively wired-AND, the father-PE will not receive a K-FrameDone input until all the son-PE’s have completed their k-frame computations. Similarly, the father-PE will not signal K-FrameDone until it has completed its k-frame computation and received the collective K-FrameDone signal from the son-PE’s. Therefore, the K-FrameDone signals serve as semaphores, to ensure that a new K-FrameStart signal is not sent until all k-frame computations on each PE have been completed.

Program SynchronousPESubtask;
{Program is initiated by K-FrameStart signal}

Begin
KFrameDoneOutput := True;
ComputeKFrame;
WaitUntilKFrameDoneInputisTrue;
KFrameDoneOutput := True
End.

Figure 2-5: PE Subtask for Synchronous Parallel Computation

26
2.2.2 Inter-Processor Communication in the Single Cluster Configuration

IPC will be limited in this thesis to the communication of a-rate variables. This means that slower rate variables (k- or i-rate, note parameters, etc.) needed by more than one PE must be replicated on each PE.

All PE's communicate with each other using the shared IPC bus. The communications scheme employed here places the burden of bus arbitration on the compiler. Let us assume for the moment that no branch instructions exist within the body of the a-rate code. Because of the repetitive nature of signal processing computation, communication among PE's is regular and predictable, and therefore slots on the IPC bus may be easily scheduled by the compiler. This can be done by inserting a two cycle I/O instruction in the TMS320 object code of the transmitting and receiving PE's such that the resulting IPC slots coincide. That is, since the k-frame computations for all PE's are initialized at the same time, slot times on the IPC bus stand in direct relation to the number of instruction cycles from the origin of the k-frame code block. Slot times are therefore scheduled by inserting IPC port I/O instructions into the k-frame code block of each PE involved in a communication in positions corresponding to the same number of instruction cycles from the origin. A two-dimensional table like that shown in Figure 2-6, showing instruction cycles along the horizontal axis and PE's along the vertical axis, is useful to conceptualize this. Note that, since I/O instructions take place directly between the data memory and the IPC port, an IPC slot can be scheduled (virtually) anywhere without having to worry about the state of the rest of the TMS320. This interleaving of IPC port instructions with the a-rate code is efficient because a PE is not dormant while other PE's are communicating on the bus, and consequently there is more time available on the bus for IPC.

This IPC strategy places an immediate restriction on the size of a-rate code blocks residing in different PE's - that is, they must all be the same number of instruction cycles in length; otherwise, successive iterations of the a-rate computation will result in the loss of synchronization of slot times. Therefore, all but the longest a-rate code block must be padded with no-operation instructions to preserve the synchronization of IPC slots.

Another restriction concerns the use of branch instructions. As will be seen in the description of the Music 320 language, a conditional assignment operator will be implemented in the primitive instruction set, which involves using conditional branch instructions to assign the value of one of two input variables to an output variable depending upon the truth or falsehood of a relation. Therefore,
Figure 2.6: IPC Scheduling and Slot Timing
not only must each branch of the conditional assignment consist of the same number of instruction cycles, but an IPC slot occurring in the midst of this operation must be scheduled in both arms of the conditional branch.

Thus, the operation of a single cluster parallel processor may be described as follows. A k-frame computation is initiated simultaneously in each PE by a request from the output FIFO. Since a-rate code blocks in each PE are constrained to contain the same number of instruction cycles, and since all PE’s run off the same clock, IPC transactions may take place among PE’s in the manner described above. As each final output sample is computed, it is sent from the father PE to the FIFO buffer along the father PE’s spare IPC port. After KSMPs a-rate samples have been computed, each PE may operate independently of the other PE’s, either to receive a new command from the host processor or to compute the next set of k-rate variables. When either process has been completed, the PE waits until its K-FrameDone input is high until it signals its K-FrameDone output. When the father PE signals K-FrameDone, then the entire cluster is ready for the next k-frame computation to begin, which is again initiated by a signal from the FIFO.

2.2.3 Implications of the Synchronous Architecture on the Compiler

Several implications of the synchronous single cluster parallel processing scheme presented above may now be discussed. One set of issues deals with the criterion for the efficient partitioning of the synthesis task. The other concerns the overall equivalency of the partitioned task to the original synthesis program in the light of the IPC strategy adopted above. Both sets of considerations will influence the nature of the compilation process described in Chapter 4.

In the summary of the operation of a single cluster architecture presented in the previous section, we simply assumed that the synthesis task had been evenly partitioned so as to approach some optimal level of efficiency. Here, this optimal level of efficiency will correspond to the maximum sampling rate that can be achieved for performance of a particular synthesis task. Due to the use of K-FrameDone semaphores which prohibit the initialization of another k-frame computation until all PE’s have completed the last one, one efficiency criterion might be to minimize the maximum length of any k-frame process. On the other hand, since the a-rate portion of the k-frame computation represents the greatest burden on the overall speed of the machine, and since the length of the a-rate code block is constrained to be equal to the largest (to maintain the synchronization of IPC slots), an alternative criterion to ensure efficient partitioning of the synthesis
task would be to minimize the maximum number of instruction cycles in any a-rate subtask. This is in fact the strategy that will be adopted in designing the partitioning algorithm.

One other comment concerns the effect of IPC scheduling on the efficiency of the final partition. While the interleaving of IPC port instructions with the a-rate object code is an efficient way to carry on IPC between PE's, the number of such I/O instructions will obviously add to the length of the code. If this number becomes large, any effort to minimize the maximum number of a-rate instruction cycles will be overwhelmed by the number of cycles due to IPC requirements. Therefore, in addition to evening out the length of the a-rate code, the partitioning algorithm in the compiler should also work to minimize the amount of IPC required among the subtasks.

It is now possible to discuss some issues concerning the interleaved IPC strategy as they relate to ensuring the equivalency of the partitioned task and the synthesis source program. When an IPC port instruction is randomly inserted in the a-rate code block, it is unknown whether the variable being broadcast (or received) was computed previous to the IPC transaction, or if it is yet to be computed. That is, there is the problem of honoring the input and output dependencies of the subtask computation: consequently, for a variable being passed from one PE to another, there is an indeterminate delay of 0, 1, or 2 samples. This situation is illustrated in Figure 2-7. It should be stressed that, in order to compensate for delays due to IPC, it is necessary that they be predictable once the partitioning of the task has been completed.

This problem of indeterminate delay times due to IPC can be solved, however, by simply constructing one level deep FIFO buffers in the TMS320 data memory. As was mentioned in the description of the TMS320, a one-cycle data memory shift instruction is available which shifts the value of one data memory location into the next higher address. By assigning two contiguous data memory locations to any variable being transmitted or received, a one level FIFO can be implemented with a single data memory shift instruction at the very end of the a-rate code block. Therefore, one FIFO buffer is implemented at both the receiving and the transmitting PE's as is illustrated in Figure 2-8. By performing IPC transactions with the buffered variable, there results a uniform delay of two samples between a variable's computation on one PE and its appearance at the other receiving PE, corresponding to a one sample delay at each buffer.

It will later be seen (in Section 4.4) that the delays incurred due to IPC along a data dependency path may be compensated for by inserting other delays elsewhere in the computation, thus pipelining
If variable $x$ is computed by PE$_2$ in: and is used by PE$_2$ in: then the resulting delay is:

<table>
<thead>
<tr>
<th>A</th>
<th>A</th>
<th>1 sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>0 samples</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>2 samples</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>1 sample</td>
</tr>
</tbody>
</table>

Figure 2-7: Indeterminacy of Delays in Uncorrected IPC
Result of computation A: stored in A, and shifted to A+1 at end of a-rate code block.

Value is communicated on IPC Bus during next sample computation.

Figure 2-8: IPC Delay Correction
the computation. In these cases also, the data memory shift instruction will be used in an analogous fashion to implement these corrective delays.

2.2.4 Limits on the Number of PE’s in a Single Cluster

The number of PE’s able to efficiently utilize a single IPC bus is limited by the number of slots available within a certain "reasonable" range of sampling rates. Medium quality music synthesis requires a sampling rate of 20 to 33 KHz. If the PE’s are running at an instruction execution rate of 5 MHz, then the total number of a-rate instruction cycles is constrained to be between 150 and 225. It is expected that a slot time lasting 4 instruction cycles is feasible for the hardware. Therefore, the number of slots is limited to approximately 20 to 30, since the delay shift operations must be placed at the end of a-rate code. Exactly how the limit on the number of available slots limits the number of PE’s using a single IPC bus depends on the efficiency of the partition and, of course, on the IPC requirements of the particular task. From the above considerations, however, it would probably be desirable to limit the number of PE’s on a bus to 6.

2.3 Extension to Multiple Cluster (Tree) Architecture

So far, the description of the parallel architecture has limited itself to the discussion of a single cluster machine. The architecture is, however, recursively expandable: any single son PE in Figure 2-4 may be replaced by a cluster. That is, the son PE uses its spare IPC port to communicate with added "grandson" PE’s, as is shown in Figure 2-9. The architecture thus takes on the structure of a tree, and it can be seen that this expansion may be extended indefinitely (at least in theory).

The synchronization of the tree processor is analogous in every respect to that of the single cluster machine. All PE’s still run off the same global clock, and the K-FrameStart is also global. K-FrameDone semaphores propagate from the lowest (leaf) PE’s up the tree to the topmost (root) PE. The subtasks assigned to each PE in the tree is the same as was used for the single cluster architecture.
Figure 2-9: Tree Architecture
2.3.1 IPC Considerations

Note that a PE may only directly communicate with father, son, or brother PE's. In order for one PE to communicate with another not connected to the same IPC bus, IPC must be scheduled through several intermediate PE's on the path. Intermediate PE's will effectively have to transfer a variable from one of its IPC busses to the other. Therefore, the IPC scheduling section of the compiler will also have to be able to route IPC paths through the tree.

In transferring a variable from one IPC bus to the other, the same issues apply regarding indeterminate delay lengths due to randomly interleaved IPC. Again, a single level FIFO buffer is implemented using two contiguous data memory locations in the TMS320, as shown in Figure 2-10. The delay due to such a communications link is therefore fixed at one sample. We may therefore state as a general rule:

Proposition 2-1: The number of samples delay incurred due to IPC between two PE's is equal to the number of PE's on the path from one to the other (inclusively).

2.3.2 Balanced Tree Architecture and Considerations

While any type of tree structure may be constructed out of PE's, a special structure that will be called a balanced tree architecture has several interesting properties that make it preferable to a general tree structure. The balanced tree architecture is defined as follows:

Definition 2-2: Let the generation number of a PE refer to the number of IPC busses from the root PE; the root is then generation 0, its sons are generation 1, all their sons are generation 2, and so on. A balanced tree architecture is defined as any tree configuration in which all PE's with the same generation have the same number of sons.

Note that the tree processor configuration in Figure 2-9 satisfies the balanced tree definition, up to generation 2.

It is immediately apparent that all PE's of the same generation also head the same number of descendant PE's. This suggests a recursive approach to the partitioning of the synthesis task which greatly simplifies the problem of distributing the task among all the PE's in the tree processor. Consider the following scheme: the root PE is "given" the entire task to divide equally among its sons; each of the sons then splits the task among its sons, and so on. The end result is that the task has been divided equally among the leaf PE's (i.e., among the last generation). The technique may easily be modified to so that non-leaf PE's are given an appropriate portion of the task as well. Not only
In order for $PE_3$ to use a variable $x$ produced by $PE_5$, an intermediate IPC link must be implemented on $PE_2$.

Figure 2-10: Intermediate IPC Links in the Tree Architecture
does this scheme greatly simplify the compilation process for arbitrarily sized (balanced) tree architectures, but it also eliminates the need for a subtask-to-PE assignment algorithm: implicit in the iterative partitioning approach is the arrangement of PE's in the parallel hierarchy.

Because of the immense simplification of the problem of task partitioning it affords, the balanced tree architecture defined above will be adopted here as the structure of our parallel music processor. With this last feature, our description of the parallel processing architecture is complete.
Chapter Three

Music 320 Language Definition

3.1 Introduction

In the preceding chapter, the architecture of the parallel music processor was defined. Described here is the specification of a synthesis language called Music 320, which, after being partitioned by the compilation process in the next chapter, can be implemented on the tree processor. Music 320 is syntactically similar to the Music 11 language [22] described in Chapter 1, and incorporates a number of concepts from data flow computation [6]. A few of its features are listed below:

- Three main classes of primitive operators constitute the instruction set. These may be used to construct any unit generator; alternatively, new primitive operators may be defined by microcoding.

- The primitive operators work on six different variable types: a-, k-, i-, and p-variables are carried over directly from Music 11. Two additional types, c- and t-variables, define and allocate constants and table memory, respectively.

- Strict rules govern the assignment of values to variables, allowing both the easy and unambiguous transformation to and from the graph representation used by the compiler, and the implementation of the program in a parallel environment.

In the following sections, the central aspects of the Music 320 language will be discussed. The first presents the primitive operators, followed by a description of the different variable types on which they operate. In the last section, the specification of the structure of Music 320 - that is, how the primitive operators are used to construct a synthesis program - is introduced, along with some examples.

3.2 Primitive Operators

There are three main classes of primitive operators used to construct synthesis programs in Music 320: arithmetic operators (add/subtract and multiply), table memory access operators, and
conditional merge operators. The syntax of the primitives resembles that of a Music 11 command, as is shown below:

\[
\langle \text{xout} \rangle \quad \{\text{PRIMITIVE OP}\} \quad \langle \text{xin1}, \text{xin2}, \ldots [\text{init}] \rangle
\]

The result variable is defined on the left, followed by the operator's name and the input variable list. Following the last input variable, an optional initialization variable designates the value to which the result is initialized upon a new note command.

The three classes of primitive operators are described separately below. All primitive operators are for 16-bit input and output variables, although some deal implicitly with fixed-point fractions (as in the case of some of the arithmetic operators) and some with unsigned integers.

### 3.2.1 Arithmetic Operators

#### 3.2.1.1 Signed Fraction Arithmetic

All input variables are treated as signed fractions; the output is corrected for overflow conditions to +1 for positive overflow and -1 for negative overflow.

\[
\text{xout ADD xin1, xin2 [init]}
\]

\textit{action:} \quad \text{xout} \leftarrow (\text{xin1} + \text{xin2})

\[
\text{xout SUB xin1, xin2 [init]}
\]

\textit{action:} \quad \text{xout} \leftarrow (\text{xin1} - \text{xin2})

\[
\text{xout MULT xin1, xin2 [init]}
\]

\textit{action:} \quad \text{xout} \leftarrow (\text{xin1} \times \text{xin2})

39
3.2.1.2 Integer Arithmetic

All input variables are treated as 16-bit unsigned integers. There is no overflow correction (i.e. mod $2^{16}$ arithmetic).

- **xout ADDM xin1, xin2 [, init]**
  
  *action*: $xout \leftarrow (xin1 + xin2) \text{ MOD } 2^{16}$

- **xout SUBM xin1, xin2 [, init]**
  
  *action*: $xout \leftarrow (xin1 \cdot xin2) \text{ MOD } 2^{16}$

- **xout MULTM xin1, xin2 [, init]**
  
  *action*: $xout \leftarrow (xin1 \times xin2) \text{ MOD } 2^{16}$

3.2.2 Table Access Primitives

3.2.2.1 Table Memory Read Operations

The first input variable is used as the 16-bit address to a *physical* location in the PE table memory to be read. The second input parameter refers to some table memory object that was defined earlier in the program. While this parameter is not used in the computation, it serves to denote the data dependency on an object, which is necessary to compilation. This will be discussed in more detail later.

- **xout TBLR xaddr, tx**
  
  *action*: $xout \leftarrow \text{tablememory @ } xaddr$
3.2.2.2 Table Memory Read-then-Write

The first input variable is used as the 16-bit physical address to a location in the PE table memory, which is assigned to xout, and is then replaced with xdata.

\[
\text{xout} \quad \text{TBLW} \quad \text{xaddr, xdata, tx}
\]

\text{action:} \quad \text{xout} \leftarrow \text{tablememory} @ \text{xaddr}
\quad \text{(tablememory} @ \text{xaddr} \leftarrow \text{xdata)}

3.2.3 MERGE Operations

The merge operator allows the conditional assignment of one of two input variables to the output variable based on the relation of a third "test variable" to zero. The general format is

\[
\text{xout} \quad \text{MERGE<cond>} \quad \text{xtest, xtrue, xfalse}
\]

\text{action:} \quad \text{xtrue if } \{\text{xtest }\langle\text{cond}\rangle 0\} \text{ is true, and}
\quad \text{xout} \leftarrow 
\quad \text{xfalse if } \{\text{xtest }\langle\text{cond}\rangle 0\} \text{ is false.}

The condition option field may contain any one of the following:

\[
< \quad \text{Less Than Zero}
\]
\[
> \quad \text{Greater Than Zero}
\]
\[
= \quad \text{Equal to Zero}
\]
\[
\leq \quad \text{Less Than or Equal to Zero}
\]
\[
\geq \quad \text{Greater Than or Equal to Zero}
\]
\[
\neq \quad \text{Not Equal to Zero}
\]

41
3.3 Variable Types

Primitive operators work on six different variable types. A variable name consists of a single letter specifying the type, followed by an integer which serves as a unique identifier. The six different letters specifying the variable type are given below:

- a -- audio rate variable (computed at the audio sampling rate).
- k -- control rate variable (computed at the slower control rate).
- i -- initialization rate variable (computed at note initialization time).
- p -- event parameter variable (passed from the host processor).
- c -- constant
- t -- table memory object (refers to an array allocated in table memory).

The a-, k-, i-, and p-type variables have been transported directly from Music 11, which was described in Chapter 1. A-, k-, and i-rate variables are assigned values using the primitive operators described in the previous section. P-variables (parameters) are assigned values by the host processor. C-variables (constants) are given values by a single assignment statement of the form

```
cx       EQU      \{expression resulting in a 16-bit integer\}.
```

While a-, k-, and i-rate variables may appear in either input or output variable fields of a primitive operator (except for fields referring to table objects), p- and c- type variables may only appear in the input field. The use of t-variables is less straightforward, and the next Section deals with their use.

3.3.1 Definition of T-Variables

The use of t-variables with primitive operators is restricted to the "tx" field of the two table access primitives but, as was mentioned, only to establish an object dependency and not a strict data dependency. Since t-variables refer to a pre-allocated block of table memory, they do not correspond to any location in the TMS320 data memory, and are not used in the actual computation of the table access primitives. In denoting an object dependency, however, they serve to indicate to the compiler,
after the partitioning has been completed, which table objects must be allocated in a particular PE's table memory space. We have yet, on the other hand, to describe the manner in which actual table objects are defined.

Furthermore, since a table object primitive uses the "xaddr" input field directly as the physical address to the PE's table memory, a way is needed to generate the physical address from the relative address of the desired array variable in table memory. This is accomplished by defining a set of table attributes to an already defined table object which, in conjunction with the arithmetic primitives, may be used to generate the offset and possibly scaled address value needed by the table access primitive. Therefore in the next two sections, first the means of defining a table object are introduced, followed by a description of the use of the table attributes.

3.3.2 Table Object Definition

Table objects are typically used for one of two purposes: either strictly as a look-up table for oscillator waveshapes, or as read-write storage for delay lines. In the first case, the table memory is initialized prior to run-time to store one cycle of some waveshape. In the second case, space is allocated in the table memory, and the values of individual elements in the array are determined during run-time. To accommodate both uses for table memory objects, two types of table objects are provided for -- read-only table objects, and read-write table objects. The formats of the declaration statements for both kinds of table objects are described separately below.

3.3.2.1 Read-Only Table Object Declaration

The format for declaring a read-only table object is given as:

\[ \text{tx} \quad R \quad \langle \text{field 1}, \langle \text{field 2}, \ldots, \langle \text{field n} \rangle \rangle \]

The \( n \) fields following the read-only declaration are used to define the waveshape stored in the table object. Field 1 is an integer designating the number of elements in the array. Fields 2 through \( n \) are used to define the waveform to be stored in the table object prior to run-time. Rather than describe all the possible waveforms that might be used, we will simply assume that the GEN function definitions used in Music 11 are available to us, and refer the reader to [22] for a complete
description.

3.3.2.2 Read-Write Table Object Definition

The format for declaring a read-write table object is given as:

\[
\text{tx} \quad \text{W} \quad \langle \# \text{ of pts.} \rangle, \text{ init}
\]

As can be seen, the read-write table declaration simply allocates the desired number of elements in the array, and provides an optional initialization field whereby the entire array may be preset to a constant prior to run-time. This would be desirable in the case of delay lines, which should be initialized to contain all zeroes.

3.3.3 Table Attributes

As was mentioned, in order to use the table access primitives to access a particular element in an array, a way must be provided to generate the physical address of that element from the corresponding relative address. More generally, it may be necessary to know several aspects of a table object's physical allocation in order to implement any one of a number of relative addressing schemes. Towards this end, we define a sub-class of table variables which function as constants referring to these aspects of a table object's physical allocation, called table attributes. The table attribute constant is of the form

\[
\text{tx(\langle attribute\rangle)},
\]

where \text{tx} is a previously declared table object, and the attribute is one of the following:
<table>
<thead>
<tr>
<th>Name of <code>&lt;attribute&gt;</code></th>
<th>Value of <code>tx(&lt;attribute&gt;)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>address of first element</td>
</tr>
<tr>
<td>last</td>
<td>address of last element</td>
</tr>
<tr>
<td>size</td>
<td>number of elements allocated</td>
</tr>
<tr>
<td>mid</td>
<td>address of midpoint</td>
</tr>
<tr>
<td>mag</td>
<td>half of &quot;size&quot;</td>
</tr>
</tbody>
</table>

Although the "first" and "last" attributes fully specify the physical location of an object in a PE’s table memory, the other three come in handy when implementing different types of addressing schemes. These will be encountered when example unit generators are built out of primitive operators in a later section.

3.4 Rules Governing the Use of Primitive Operators and Variables

Having described the primitive operators and variables, we may now present the rules which govern their use.

**ASSIGNMENT RULES**

1. All input variables to a primitive operator must have been defined in an earlier statement, or by that statement, or by a FORWARD-reference command.

2. The output variable name to a primitive operator must be unique; that is, there are no side effects (except those specified in the init field of a primitive operator).

3. The computation rate type of the output variable must be at least as fast as the fastest input variable to that operator.

Rule 1 simply requires that the input variables to a primitive operator must all have been defined by an earlier primitive operator, or by that very operator (i.e., self-loops may be created). This means that, in order to create non-trivial feedback loops in the synthesis task, there must be a way to forward-reference a variable that is to be defined later in the program. It is for this reason that a special non-computational FORWARD command is provided. The FORWARD command is used

45
in the following fashion:

\[ \text{<variable name> \ \ \ \ FORWARD \ \ \ \ <loop delay fix>} \]

The variable name which is forward-referenced may be used by any statement following the FORWARD command, provided it is defined at some later point.

The delay-fix parameter following the FORWARD command is necessitated by the fact that the primitive operators comprising the feedback loop may be partitioned among a number of PE's during compilation. It will be impossible to compensate for delays due to IPC unless a maximum possible delay is specified as the loop delay fix parameter, corresponding to the longest anticipated delay due to IPC if the feedback loop were to be split among several processors. While this may be an inconvenience, there are alternatives, which will be discussed in the concluding chapter.

Rule two restricts the occurrence of side effects to those which take place during note initialization as a result of the init field of a primitive operator. This means that there is a one-to-one correspondence between variable names and the operators that assign values to them. It will be seen that this allows the easy transformation of the synthesis program into its proper graph representation.

Rule three defines the coercion constraints of rate variables, which are shown graphically in Figure 3-1. Primitive operators with output variables of the type represented by a node in Figure 3.1 may have input variables of any type reachable from that node along a directed path. Note that constants may be defined in terms of other constants as well as by table attributes.

3.5 Music 320 Structure

The primitive operators presented above represent the smallest division of the synthesis task recognized by the task-partitioning compiler. How these primitives are used to build a synthesis task is the topic taken up here.

A Music 320 synthesis program consists essentially of three sections. The first section is simply a header which defines several run-time characteristics of the synthesis task, such as the number of a-rate samples computed per k-rate sample (KSMPS), and the number of output channels. The second section is a macro definition section in which often used unit generators are defined in terms
Figure 3-1: Variable Coercion
of primitive operators and other previously defined macros. The third section defines the actual instruments in the orchestra.

In the following two sections, the macro and instrument definition sections of the Music 320 language are described and demonstrated.

3.5.1 Macro Definition Section

Macro definitions allow the computer music synthesist to build complex unit generators out of primitive operators and previously defined macros. A macro operator, then, is a synthesis command that has been defined in the macro definition section. The general format for a macro definition is

```
DEFINE{{<model statement>}}
   <definition statement 1>
   <definition statement 2>
   .
   .
   .
   <definition statement n>}
```

The model statement is of the same format as the statement for a primitive operator, i.e.

```
xout MACRONAME xin1, xin2, ... xinN.
```

The input and output variable list is composed of dummy variables, which are used by the definition statements. The dummy variables are typed with a-, k-, or i- prefixes. The definition statements will define local variables which are bound within that particular macro definition. One definition statement will assign a value to the xout dummy variable. The MACRONAME and the output variable type uniquely name the macro definition, so that two macro definitions with the same MACRONAME and different rate dummy output variables are distinct.

The rules and features of macro operators are best brought to light by considering a few examples. The following is the very simple definition of the a-rate macro PHASOR, which corresponds to the Music 11 command of the same name:
DEFINE{(aout  PHASOR  ain, cinit)
aout  ADDM  aout, ain, cinit
}

Recall that ADDM is modulo $2^{16}$ arithmetic, so the output of PHASOR winds back around from +1 to -1 when it is viewed as a signed fraction. Aout is initialized to cinit at initialization time. Also, even though the input to PHASOR is designated as an a-rate variable, there is no reason why it couldn't be a k- or i-rate variable, since either would still satisfy the coercion constraints shown in Figure 3-1.

The PHASOR operator is very useful in the construction of oscillators, where it is used to scan some waveshape stored in table memory. In order to use it for this purpose, though, another macro operator TABLE must first be defined as follows:

DEFINE{(aout  TABLE  ain, tx)
a1  MULT  ain, tx(mag)
a2  ADD  a1, tx(mid)
aout  TBLR  a2, tx
}

TABLE scales and offsets ain, based on the attributes of the table object in the dummy input variable list, to address the block of physical memory locations in the table memory corresponding to that object. Note that two locally defined temporary variables have been used, and that these are strictly bound to this particular macro definition.

So far, macro operators have only been defined in terms of primitives. A good example of a macro definition using both macro and primitive operators is the following definition of OSCIL, which assumes that PHASOR and TABLE have been previously defined:
DEFINE{(aout
   OSCIL aAmp, afreq, tx, cinit)
   aout
   a1 PHASOR afreq, cinit
   a2 TABLE a1, tx
   aout MULT a2, aAmp
}

OSCIL is a unit generator which will generate an audio tone with a waveshape determined by the waveform stored in the table object tx, amplitude determined by aAmp, and frequency determined by afreq.

One unit generator which demonstrates the use of a MERGE primitive, a read-write table object, and the FORWARD-reference command is the macro implementation of the DELAY command in Music 11.

DEFINE{(aout
   DELAY ain, cnpts)
   t1
   W cnpts, 0
   c1 EQU 0001_16
   c2 EQU -t1(size)
   c3 EQU t1(last) - 4
   a3 FORWARD 4
   a1 PHASOR a3, t1(first)
   a2 SUB a1, c3
   a3 MERGE:= a2, c2, c1
   aout TBLRW ain, a1, t1
}

The DELAY operator defines a read-write table object within its definition with length equal to the desired number of points delay. A PHASOR is used to scan the table, but in a manner quite different from the case of OSCIL. Here, PHASOR increments by one (c1) every sample until it equals four less than the address of the last entry of the table. At this point, the output of the SUB primitive forces the input to the PHASOR to be the negative of the size of the table, thus forcing the PHASOR to reset to the starting address four samples later (due to the FORWARD-reference delay of 4). The delay-fix parameter of four corresponds to the case when the feedback loop is distributed between two PE's.

As can be seen, macro definitions may be used to define a wide variety of unit generators.
More examples of this type of construction are provided in an example at the end of this Chapter. In some cases, however, it may be more efficient to implement the unit generator in microcode. For example, it is much more efficient to implement filtering operations in TMS320 microcode, since the internal data paths of the chip are optimized for performing these functions. At any rate, it can be seen that the desired goal of flexibility can be achieved with the approach that has been presented above.

3.5.2 Instrument Definition Section

The macro definition section of the Music 320 orchestra program defines the unit generators which are to be used in the instrument definition section. An instrument definition block is a collection of primitive and macro operators which appear to the host processor as a distinct computational entity. Instruments are identified by a unique string variable, and their real-time operation is controlled by the p-variables sent from the host processor. The format for an instrument definition block is given as

\[
\text{INSTR} \quad \text{<unique string identifier>}
\]

\[
\text{<instrument statement 1>}
\]

\[
\text{<instrument statement 2>}
\]

\[
\vdots
\]

\[
\text{<instrument statement n>}
\]

\[
\text{ENDIN.}
\]

The instrument statements which compose the body of the instrument block are simply the primitive or macro operators described above. Variables defined within an instrument are local to that instrument. Instruments communicate by means of global variables. Global variables may be either a-, k-, or i-rate, and are designated by a pre-prefix "g" (e.g. ga1, gk4, gi2). A global variable is distinct from the non-global variable with of the same type and same integer identifier (that is, ga2 is distinct from variable a2).

Instruments are built with primitive and macro operators following the same rules used for constructing macros out of primitives. Thus the single-assignment rule applies equally to global
variables.

Now that the means for building instruments has been presented, all that is left to describe is the way in which computed sound samples are sent to the FIFO and D/A converters to be heard. This is accomplished with a special OUTPUT operator of the form

\[ \text{OUTPUT} \ aout1, aout2, \ldots, aoutN \]

where \( N \) is equal to the number of sound output channels specified in the orchestra header section. There is only one restriction on the use of the OUTPUT operator, and that is that it can only be used once in an orchestra. Unlike Music V or Music 11, where repeated instances of an output statement cause the signal variables to be implicitly accumulated, all instrument outputs must be explicitly summed before being output.

This completes the description of the Music 320 music synthesis language. An example orchestra program shown in Figure 3-2 will serve to illustrate many of the points discussed above.

The program header section simply sets KSMPS to 20 and specifies two channels of output audio. Assuming that the macro definition section already contains the definitions for the TABLE and OSCIL operators, it goes on to define three more operators to be used in the instrument definition section. We will also assume the availability of a new primitive operator TONE, a first-order recursive low-pass filter. The first input parameter is the input signal to be low-pass filtered, and the second specifies the cutoff frequency.

This primitive operator is used in the first new macro to be defined, ADENVG, which is a simple k-rate envelope generator. It is a bit unconventional, in that it doesn’t directly control the duration of the note; rather, the duration is controlled by the instantaneous value of \( key \), which should be some positive value (corresponding to the loudness of the note) when a "key" is depressed, and 0 when it is released. The TONE operator acts as a simple smoothing filter, with the MERGE operator specifying a different amount of smoothing for the attack and decay portions of the note. If \( key \) is controlled by a parameter value, then the host processor (which could conceivably be monitoring a keyboard for real-time performance) would send the key parameter at every instance of a key being pressed or released.

The second macro operator is a straightforward implementation of an FM oscillator (see Figure
1-4), using two OSCIL operators. The constant c1 limits the full-scale modulation index to 6. The last macro, MIX, is used to mix two input signals, balancing the two based on a third input variable kLR. Its operation is therefore analogous to a balance control on a stereo.

The first instrument to be defined is an FM instrument, where the modulating frequency is forced to be twice the carrier frequency. Separate envelope generators are used to control the amplitude envelope of the note and the modulation index, thus producing a time-varying timbral envelope. The output of the oscillator is given a global variable name so it may be used by other instruments.

The second instrument is an implementation of the waveshaping synthesis technique, as described in Section 1.1.2. Table object t1 is used by the OSCIL macro to produce a sine wave, and table object t2 is the waveshaping look-up table. Once again, separate envelope generators are used, one to control the overall amplitude envelope, and the other simultaneously controlling the amplitude and the dynamic timbral characteristics.

The last instrument is the output summing instrument, which takes the outputs of the two previous instruments and places them spatially in the stereo signal with the MIX operators.
KSMPS = 20
NCHNL = 2

; Macro Definition Section

DEFINE{(aout       TABLE       . . . )}
DEFINE{(aout       OSCIL       . . . )}

DEFINE{(kout       ADENVG   key, iattack, idecay)
         kslope    MERGE = 0key, idecay, iattack
         kout      TONE      key, kslope
    }

DEFINE{(aout       FOSCIL   aAmp, afreq, amodfreq, amodindx, itbl)
  ;
  ; c1 specifies the maximum modulation index
  ;
  c1       EQU       6_{16}
al       OSCIL     amodindx, amodfreq, itbl
a2       MULTM     a1, c1
a3       ADDM      a2, afreq
aout     OSCIL     aAmp, a3, itbl
    }

DEFINE{(aout       MIX      ain1, ain2, kLR)
  ;
  ; 7FFF_{16} is the maximum positive fraction (approx. = 1)
  ;
  c1       EQU          7FFF_{16}
k1       SUB       c1, kLR
a1       MULT      ain1, k1
a2       MULT      ain2, kLR
aout     ADD       a1, a2
    }

Figure 3-2: Example Music 320 Program
; Instrument Definition Section

INSTR     FM-instr

; p3 is the key signal to the envelope generators.
; p4 is the frequency of the note.
; p5 specifies the attack of the amplitude envelope.
; p6 specifies the decay of the amplitude envelope.
; p7 specifies the attack of the timbral envelope.
; p8 specifies the decay of the timbral envelope.
; p9 controls the overall amount of frequency modulation.

    c1   EQU  2^16
    t1   R    <sine wave look-up table>
    l1   MULTM  p4, c1
    k1   ADENVG  p3, p5, p6
    k2   ADENVG  p3, p7, p8
    k3   MULT  k2, p9
    Ga1  FOSCIL  k1, p4, k1, k3, t1

ENDIN

INSTR     Waveshaping

; p3 is the key signal to the envelope generators.
; p4 is the frequency of the note.
; p5 specifies the attack of the timbral envelope.
; p6 specifies the decay of the timbral envelope.
; p7 specifies the attack of the amplitude envelope.
; p8 specifies the decay of the amplitude envelope.

    t1   R    <sine wave look-up table>
    t2   R    <waveshaping look-up table>
    k1   ADENVG  p3, p5, p6
    k2   ADENVG  p3, p7, p8
    a1   OSCIL  k1, p4, t1
    a2   TABLE  a1, t2
    Gaa2  MULT  a2, k2

ENDIN

Figure 3-2, continued
INSTR       Output

; p4 - mixes the two instruments into channel one of the output.
; p5 - mixes the two instruments into channel two of the output.

   a1    MIX   Ga1, Ga2, p4
   a2    MIX   Ga2, Ga1, p5
       OUT    a1, a2

ENDIN

Figure 3-2, concluded
Chapter Four

Task Partitioning Compiler

Now that the music synthesis language has been defined, we may now turn our attention to the compilation techniques used to implement it on the parallel architecture described in Chapter 2. Presented here is a series of graph algorithms which partition the synthesis task among PE's in a balanced tree architecture, schedule inter-processor communication, and compensate for the delays incurred as a result of IPC to ensure equivalence of the final partitioned task to the original synthesis source program. Finally, some considerations necessary for the final compilation of sub-tasks will be discussed. Before these graph algorithms are presented, however, it is necessary to first describe the way in which a Music 320 synthesis program may be transformed into its graph representation.

4.1 Graph Representation of the Music 320 Synthesis Program

The Music 320 language was designed specifically to make the transformation from a statement-by-statement description of a synthesis program into its graph representation an easy one. We will first present some of the notation that will be used for the rest of the chapter.

A directed graph \( G(V,E) \) is a structure which consists of a set of nodes or vertices \( V = \{v_1, v_2, v_3, \ldots \} \) and a set of directed edges \( E = \{e_1, e_2, e_3, \ldots \} \). An example of a directed graph structure is shown in Figure 4-1. An edge \( e \) such that \( u \xleftarrow{e} v \) is said to be directed from \( u \) to \( v \); \( u \) is called the start-vertex of edge \( e \), and \( v \) is called the end-vertex. The outdegree \( d_{out}(v) \) of a vertex \( v \) is the number of edges which have \( v \) as their start-vertex; similarly, the indegree \( d_{in}(v) \) is the number of edges which have \( v \) as their end-vertex. A directed path is a set of edges \( \{e_1, e_2, \ldots, e_n\} \) such that the end-vertex of \( e_i \) is the start-vertex of \( e_{i+1} \). A directed path is a directed circuit if the start-vertex of the path is also its end-vertex.

In the graph representation of a Music 320 synthesis program, nodes represent primitive operators and directed edges represent data dependencies. For example, the Music 320 synthesis fragment in Figure 4-2a may be represented by the graph structure in Figure 4-2b. Note that edges
Figure 4.1: A Directed Graph
are directed from a node representing a primitive operator to those primitive operators which supply the variables necessary for its own computation; that is, in the opposite direction of data flow. Otherwise, the graph structure is analogous to the data dependency graphs described by Kuck [9], or to the data flow graphs used to describe data flow programs [6]. Nodes representing primitive operators are tagged with the rate at which they are computed (i.e. a, k, or i), and with the type of operator they represent (e.g., ADD, MULT).

In order that the graph structure may represent every detail of the Music 320 source program, nodes will also be used to represent constant variables, score parameters, and table objects. This is demonstrated in Figure 4-3. Nodes representing constants are tagged with the 16-bit value assigned to them in the synthesis program. Nodes representing score parameters are tagged with the parameter variable's integer identifier, and with the identifier of the instrument definition block in which it was defined. Nodes representing table objects will be tagged with the type of the table object (i.e. read-only or read-write), the number of words of table memory that must be allocated for it, and with information about what the table object should be initialized to prior to run time. Table object nodes will be drawn as rectangles to make their location and function in the graph easier to identify. A-, k-, and i-rate computation nodes are also tagged with the id of the instrument block in which they occur, as well as with the line number of the statement they represent.

The only remaining aspects of a Music 320 program yet to be incorporated into the graph representation are the delay-fix parameters associated with FORWARD-referenced data dependencies, and table attribute dependencies. The delay-fix parameters associated with a FORWARD-referenced variable are designated by defining a function $d(e)$ for each edge of the synthesis graph, such that $d(e)$ is equal to the delay-fix parameter if the edge represents a FORWARD-referenced data dependency, and is equal to zero otherwise. A table attribute dependency is represented by a directed edge with end-vertex at the table object to which the attribute refers. An edge representing a table attribute dependency is tagged with the attribute designated by its use in the program (e.g., first, last, size), and will be drawn with a dotted line to avoid confusion with actual dependencies on the table object itself, which occur only in the case of table access primitives. An example of the graph representation of FORWARD and table attribute dependencies is shown in Figure 4-4.
Figure 4-2: Graph Representation of a Music 320 Fragment
t1  R  1024, <initialization info>
c1  EQU  1
i1  ADD  i1, c1
i2  MULT  i1, p3
i3  TBLR  i2, t1

(a)

Figure 4-3: Graph Representation of Constants, Parameters, and Table Objects
\begin{itemize}
\item c1 \ldots
\item c2 \ldots
\item c3 \ldots
\item t1 \ W \ cnpts, 0
\item a3 \ FORWARD \ 4
\item a1 \ ADDM \ a3, t1(first)
\item a2 \ SUB \ a1, c3
\item a3 \ MERGE= \ a2, c2, c1
\item aout \ TBLRW \ ain, a1, t1
\end{itemize}

(a)

(b)

Figure 4-4: Graph Representation of Forward and Table Attribute Dependencies
4.1.1 Translation of the Music 320 Source Program into its Graph Representation

As was mentioned earlier, Music 320 was designed to facilitate its translation into its corresponding graph representation. Because of the assignment rules governing the construction of synthesis macros and instruments, there is a one-to-one correspondence between computation variable names (i.e. a, k, and i rate) and the primitive operators that generate them, within the macro or instrument definition block in which they are defined. A compiler which constructs a data structure representation of the synthesis graph would simply construct a node for each primitive operator in the instrument definition section (expanding macro operators when they are encountered), and connect its edges according to the variable names appearing in the input variable list of the statement. Since all variable names must be defined before they are used, and since a variable name may only be used once within the definition block in which it is defined, this process of constructing a data structure representation of the synthesis graph is trivial, and would probably be accomplished in a single pass through the source program.

The implementation of the FORWARD-reference command presents the only complication in the construction process described above. When such a command is encountered, an unspecified node structure is constructed; when the variable is used in a subsequent statement, the edge is connected to this unspecified node; when the variable is finally defined, the node may then be tagged with the appropriate information.

4.1.2 Properties of the Synthesis Graph

The graph representation of the synthesis program has several properties which may now be stated. First, it can easily be seen that the only nodes for which $d_{out}(v) = 0$ are those nodes representing constants, score parameters, and read-only table objects. It is also readily apparent that the only node for which $d_{in}(v) = 0$ is the OUTPUT operator node. If every variable defined in the source program is used by at least one primitive operator, then a path exists from the OUTPUT node to any other node in the synthesis graph.

A fourth and more useful property derives from the coercion constraints placed upon the computation of a-, k-, and i-rate variables. Recall that the computation rate type of the output variable must be at least as fast as the fastest input variable to that operator. We will define the subgraph $S(V_d,E_d)$ of the overall synthesis graph $G(V,E)$ to be composed of the set of nodes
representing a-rate operators \( V_a \subseteq V \), and of the set of edges \( E_a \subseteq E \{ e \in E \text{ if } u \xrightarrow{e} v \exists u, v \in V_a \} \).

**Property 1: A-Rate Subgraph Connectivity**

The subgraph \( S(V_a, E_a) \) of the overall synthesis graph \( G(V,E) \) is completely connected from the OUTPUT operator node.

**Proof:** Let us assume that every node in \( V \) was connected from the OUTPUT operator. Because of the coercion constraints governing the use of primitive operators, any edge with an end-vertex at an a-rate node must have an a-rate node as its start-vertex. By induction, then, it can be seen that any path from the output node to an a-rate node will be composed entirely of edges with start and end-vertices at a-rate nodes, or alternatively, of edges \( e \in E_a \). \( \square \)

This property is of great utility to the task partitioning algorithm presented in the next section. We saw in Chapter 2 that the partitioning algorithm should seek to minimize the maximum length of any a-rate code block residing in the PE's. Therefore, the partitioning algorithm need only concern itself with the subgraph \( S(V_a, E_a) \).

### 4.2 Task Partitioning Algorithm

It is the job of the task partitioning algorithm to divide the synthesis graph into numerous subtasks, and to assign these subtasks to the individual PE's in the parallel architecture. In developing the parallel architecture in Chapter 2, two criteria for maximizing the efficiency (i.e. sampling rate) of the resulting partition emerged, and are restated below:

- The maximum length (in instruction cycles) of any a-rate code block should be minimized.

- The amount of IPC between subtasks should be minimized.

A large body of work exists in the literature concerning one or both of these criteria, but they are generally restricted to task partitioning among a small number of PE's. Moreover, most of this work deals with the partitioning of the broadest possible range of source program material (those with non-deterministic program flow, those which implement iteration and/or recursion, etc.), and so is unsuitable for the problem presented here. By realizing, however, that synthesis programs have several properties which immeasurably simplify the job of partitioning the task under the above listed constraints, a strategy based on heuristic assumptions about the nature of synthesis programs may be developed. This topic is pursued in the following section.
4.2.1 Development of Partitioning Heuristics for Synthesis Programs

In defining the properties of a synthesis program written in Music 320, it is immediately obvious that program flow is completely deterministic, which allows the scheduling of IPC within the compilation process. A more interesting property of synthesis programs, though, is the "convergent" nature of the computation, which is especially noticeable in its graph representation; there is only one output node, and all other computation nodes branch out from it. As a matter of fact, it can be argued that a good first-order approximation of the structure of a generalized synthesis graph is a tree. A tree graph $T(V,E)$ is a directed graph with the following properties:

1. $d_{in}(v) = 1$ for every vertex $v \in V$ except for one vertex $r$ called the root node, for which $d_{in}(r) = 0$.

2. There is a unique path from $r$ to every other vertex $v \in V$.

In a tree graph, a vertex $v$ is a descendant of $u$ if there is a directed path from $u$ to $v$; $u$ might also be called an ancestor of $v$. An example of a graph demonstrating these properties is shown in Figure 4-5.

The assumption that the a-rate subgraph $S(V_{a \cdot r}, E_{a \cdot r})$ of any given synthesis graph $G(V,E)$ can be approximated by a tree structure forms the heuristic base around which the task partitioning strategy may be built. It can be defended on two counts, based on programming methodologies typically used by composers in constructing synthesis programs:

1. The four popular synthesis techniques used to construct instruments, namely FM synthesis, additive synthesis, subtractive synthesis, and waveshaping synthesis, all take on a tree structure when their computations are represented by a synthesis graph.

2. Instruments themselves, in considering the global communications among them, are typically arranged in a tree-like fashion.

The first observation can be demonstrated by considering the unit generator representation of the four synthesis techniques described in Section 1.1.2. It can be seen from Figure 4-6 that the graph representation for the oscillator also satisfies the properties of a tree graph, with the exception of a single self-loop, which may be ignored since it is ineligible for IPC. Note that the table attribute dependencies may also be ignored, since since they do not represent true data dependencies. Therefore, since the structure of the oscillator is a tree, each configuration of oscillators, adders, and multipliers shown in Figures 1-2, 1-3, 1-4, and 1-5 will conform to a tree structure when translated.
Figure 4.5: A Directed Tree Graph
into their respective graph representations.

The second observation may be defended by noting that composers typically create instrument definitions which are summed together before being output, as shown in Figure 4-7. The root instrument shown in Figure 4-7 may also contain reverberation, delays, and other signal processing associated with the mixed signal.

A task partitioning strategy which addresses the two criteria of minimizing the maximum length of any resulting a-rate subtask while also minimizing the amount of IPC stems from the fact that a cut on any edge of a tree graph completely isolates a subset of nodes from the rest of the graph. This is shown in Figure 4-8. It is also readily apparent that the resulting subgraphs are also trees, and that the number of subtrees resulting from placing several cuts on a tree graph is equal to the number of cuts plus one. If each of the subtrees are interpreted as subtasks to be assigned to individual PE's, it will be seen that each of the cuts represents a single variable which must be assigned a slot on an IPC bus. If a tree could be partitioned in such a way as to minimize the maximum number of cycles represented by the computation nodes of each subtree, then a distribution of the task among PE's would result which satisfies the two efficiency criteria mentioned above. In fact, a tree partitioning algorithm exists in the literature [2] for placing cuts on a tree graph in which each of the nodes are assigned a weight function \( w(v) \), such that the maximum total weight of any subtree is minimized. This min-max tree partitioning algorithm forms the heart of the task partitioning strategy to be presented here.

An overall summary of the task partitioning strategy may be then formulated in the following manner. A given synthesis graph will almost certainly not be an exact tree structure, but if a subset of the data dependencies which do correspond to a tree graph are considered, then the min-max partitioning algorithm may be applied to this graph, known as the spanning tree graph. If the assumption is valid that any synthesis graph may be approximated by a tree structure, then the number of edges not represented by the spanning tree graph will be small, and the resulting partition will have a minimal number of inter-task data dependencies. Thus the two efficiency criteria imposed on the compiler by the limitations of the parallel architecture will be addressed. A more detailed account of the task partitioning process is given in the next Section.
Figure 4.6: Graph Representation of the OSCIL Macro
Figure 4-7: Typical Orchestra File Organization
Figure 4.8: Placing Cuts on Tree Graph Edges to Isolate Subtasks
4.2.2 Partitioning Algorithm for the Single Cluster Architecture

In presenting the task partitioning strategy, it will first be helpful to describe the way in which it would be implemented for a single cluster parallel architecture. An outline of the process is given below:

1. The a-rate subgraph $S(V_a, E_a)$ is derived from the total synthesis task $G(V, E)$.

2. Each of the vertices $v \in V_a$ is assigned a weight function $w(v)$, corresponding to the number of instruction cycles required for the computation of that primitive operator.

3. A spanning tree subgraph $T(V_{a,t}, E_{a,t})$ is derived from $S$, with $E_{a,t} \subseteq E_a$; that is, a subset of $E_a$ is chosen such that the resulting graph $T$ has the properties of a tree given in Section 4.2.1.

4. A min-max partition of the spanning tree $T$ is performed, with the number of cuts equal to one less the number of PE's in the single cluster; each of the subtrees then defines a subtask which may be assigned to a single PE. The subtask containing the original root of $T$ (i.e., the OUTPUT operator) is assigned to the father-PE of the cluster; all other assignments are arbitrary.

The first step allows the algorithm to consider only the a-rate nodes, since it is the computational burden represented by these nodes that we seek to distribute among the PE's. The construction of this a-rate subgraph is trivial due to the property of the synthesis graph $G(V, E)$ mentioned in section 4.1.2, which states that the a-rate subgraph is connected. Thus, the subgraph $S$ is derived from $G$ by simply defining it to be composed of the set of a-rate vertices $V_a (\subseteq V)$ and the set of edges $E_a$ where \( \{e \in E_a, u \triangleleft v \triangleright u, v \subseteq V_a\} \). Since the OUTPUT operator and any read-write table objects operating at the audio rate are also defined to be a-rate nodes, these types of nodes may be present in $S$ along with the a-rate primitive operators of the synthesis task.

Step two simply assigns a weight function to each node in $S$ corresponding to the number of instruction cycles represented by the microcode for each type of primitive operator. Nodes representing read-write table objects are assigned a zero weight.

Step three defines a new graph $T(V_{a,t}, E_{a,t})$ known as the spanning tree of the graph $S(V_a, E_a)$, which contains all of the a-rate nodes of $S$, but only a subset of the edges. The elements of $E_{a,t}$ are chosen such that $T(V_{a,t}, E_{a,t})$ is a tree graph. The spanning tree $T$ may be defined by the algorithm given below, which is modelled on a depth-first search type of algorithm [20]:

71
**DFS Spanning Tree Algorithm**

All nodes \( v \in V \) and edges \( e \in E_a \) are initially unmarked; the function \( f(v) \) will be the node from which \( v \) was first reached in the steps below, and is initially undefined.

1. Let \( v \leftarrow r \) (the OUTPUT node).

2. Mark node \( v \).

3. If there are no unused edges with start-vertex \( v \), go to step 5.

4. Choose an unmarked edge \( v \rightarrow u \). Mark \( e \) "used". If \( u \) is marked, then go to step 3. Otherwise, do the following:
   
   a. Let \( E_f \leftarrow E_f + e \).
   
   b. Mark vertex \( u \).
   
   c. Let \( f(u) \leftarrow v \), \( v \leftarrow u \), and go to step 3.

5. If \( f(v) \) is defined, then \( v \leftarrow f(v) \), and go to step 3. Otherwise, \( v = r \) and the algorithm terminates.

A spanning tree that may be derived from the graph shown in Figure 4-1 is shown in Figure 4-9. The spanning tree \( T(V_a,E_f) \) constructed by this algorithm is not unique; there are many possible spanning trees for a given graph \( S(V_a,E_a) \). While the choice of spanning tree will affect the efficiency of the resulting partition, both in terms of IPC requirements and subtask distribution, let us assume for now that variations in efficiency will be small between partitions with different spanning trees. This topic will be brought up again in Chapter 5.

Finally, step four effects the partition of the spanning tree using a shifting algorithm for optimal min-max tree partitioning presented by Becker, Schach, and Perl [2], which is presented in Appendix A. The spanning tree graph \( T(V_a,E_f) \) is partitioned into \( k \) subtrees by placing \( k-1 \) cuts on the appropriate edges, such that the maximum weight of any subtree (defined to be equal to the sum of the weights of the vertices) is minimized. Each subtree, then, represents a subtask to be assigned to a single PE in the single cluster architecture with \( k \) PE's. Since the father PE of the cluster is tied directly to the output FIFO through one of its IPC ports, it will be assigned the subtask containing the OUTPUT node. All other subtask-to-PE assignments are arbitrary.
Figure 4.9: Possible Spanning Tree for the Graph in Figure 4.1
After computation nodes have been assigned to PE’s, we must of course schedule IPC in order to handle the data dependencies represented by those edges which have start and end vertices assigned to different PE’s. This is discussed in Section 4.3; let us first, however, describe the way in which this simple task partitioning strategy may be extended to the partitioning of tasks among PE’s arranged in the balanced tree architecture described in Section 2.3.

4.2.3 Partitioning Algorithm for the Balanced Tree Architecture

It was seen in the description and definition of the balanced tree architecture that, if a compilation strategy existed for partitioning a task among the PE’s of a single cluster, then a recursive implementation of this strategy could be used to partition a task among the PE’s in a balanced tree architecture. Specifically, since partitioning a spanning tree results in a number of subtrees (all but one of which is assigned to a son PE in a cluster), each subtree assigned to a son PE may itself be partitioned among PE’s of the next generation cluster. In order to effect an even partition, however, it will be necessary to ensure that the weight of the subtask assigned to the father PE of each successive cluster is approximately equal to the projected optimal weight for any PE in the tree configuration. That is, unless some corrective measure is taken, the recursive implementation of the min-max tree partition will assign large subtasks to those PE’s toward the top of the tree hierarchy, and successively smaller sized subtasks to PE’s in the lower portions of the tree.

In order to recursively partition a task, we shall re-weight the root node of the (sub)tree given to the father PE of a cluster so that when the min-max tree partition is performed, the subtask assigned to the father PE is approximately equal to the cost of the total synthesis task divided by the number of PE’s. If we define this new weight \( w_f(r) \) of the root node \( r \) to be equal to the computational weight \( w(r) \) plus a correction factor,

\[
w_f(r) = w(r) + a,
\]

(4-1)

then the value of \( a \) may be found in the following manner. We first define the following characteristics of a father PE and the subtask to be partitioned among the PE’s in its cluster:

\( k \) = the number of son PE’s in the cluster.

\( T \) = the total number of descendant PE’s (sons, grandsons, etc.) not including the father PE.

\( W_t \) = the total weight of the subtask given to the father PE to be partitioned.
Therefore,
\[ W_t/(T+1) = \text{the desired weight of each subtask.} \]

and
\[ (W_t+a)/(k+1) = \text{the approximate weight of a subtask assigned to a son PE as a result of the min-max partition.} \]

We would like to choose \( a \) such that the weight of the subtask assigned to the cluster father is equal to the desired weight of each subtask, or
\[ (W_t+a)/(k+1) - a = W_t/(T+1). \]

Solving for \( a \),
\[ a = W_t(T - k)/(k(T + 1)). \] \hspace{1cm} (4-2)

Thus, by re-weighting the root node of a subtree assigned to be partitioned among a cluster of PE's somewhere in balanced tree architecture, the resulting min-max tree partition will assign the father PE a subtask with the proper weight. Note that if the son PE's of the cluster are also the leaf PE's in the processor tree, then \( k = T \), and consequently \( a \) is zero (the single cluster condition).

The task partitioning algorithm for the balanced tree architecture may now be summarized in the following steps:

1. Initially, assign the entire spanning tree to the root PE. Let \( G \) be the number of generations of PE's in the balanced tree architecture. Also, let \( N \) be the generation number currently being considered. Initially, let \( N = 0 \).

2. As long as \( N < G-1 \), do the following:
   a. For each PE in generation \( N \), do the following:
      i. Re-weight the root node of the tree by Equations 4-1 and 4-2.
      ii. Perform min-max tree partitioning algorithm on the tree.
      iii. Assign the resulting subtrees to PE's: the sub-tree with the original root is assigned to the father PE of generation \( N \); the other subtrees are assigned to the son PE's of generation \( N+1 \) indiscriminately.
   b. Let \( N = N + 1 \).
4.3 Scheduling Routine for Inter-Processor Communication

The task partitioning strategy described above assigns groups of computation nodes with highly localized data dependencies to the PE's in the parallel architecture. Data dependency edges between two nodes not assigned to the same PE represent communications that must be scheduled on one or several IPC busses. Described here is a straightforward process for designating IPC requirements within the graph representation of the synthesis task used thus far.

IPC requirements may be represented by introducing a special class of non-computational nodes called IPC nodes. There are three varieties of IPC nodes, which specify one of three possible operations on a PE's IPC busses; also, for each of the three varieties of nodes, there are two modes, corresponding to the two IPC ports on each PE. The six types of IPC nodes are listed and described below:

- **IN\(\uparrow(a)\)** inputs a variable from the upper IPC port at slot time \(a\).
- **IN\(\downarrow(a)\)** inputs a variable from the lower IPC port at slot time \(a\).
- **OUT\(\uparrow(a)\)** outputs a variable onto the upper IPC port at slot time \(a\).
- **OUT\(\downarrow(a)\)** outputs a variable onto the lower IPC port at slot time \(a\).
- **TRAN\(\uparrow(a,b)\)** receives a variable from the lower IPC port at slot time \(a\) and transfers it to the upper IPC port at slot time \(b\).
- **TRAN\(\downarrow(a,b)\)** receives a variable from the upper IPC port at slot time \(a\) and transfers it to the lower IPC port at slot time \(b\).

In describing the use of IPC nodes, the following notation will be used extensively. First, let us represent the tree architecture as a directed tree graph \(T(V_p,E_p)\), with the nodes in \(V_p\) used to designate PE's and edges \(E_p\) used to designate the PE hierarchy. Further, let us identify each PE node in \(V_p\) with a unique integer. For the set of nodes \(V_a\) which were distributed among PE's in the task partitioning algorithm from the preceding section, we define \(p(v)\) as the identifying integer of the PE to which the computational node \(v\) was assigned. We shall also use \(PE(v)\) to designate the PE node in \(T(V_p,E_p)\), \(PE(v) \in V_p\).

The use of IPC nodes to schedule communications between PE's is relatively simple. For every data dependency edge \(e, u \xrightarrow{E_d} v, e \in E_d\), such that \(p(v) \neq p(u)\), IPC nodes are inserted along the edge \(e\) to correspond to the IPC operations necessary for the communication from \(PE(v)\) to \(PE(u)\).
Specifically, one IPC node is assigned to each processing element along the unique path from PE(v) to PE(u). Note that if PE(v) and PE(u) are not tied directly to the same IPC bus, one or several TRAN-type IPC nodes will have to be assigned to those PE's along the path. It is also apparent from the descriptions of the various IPC nodes that an IN-type IPC node will be assigned to PE(u), and an OUT-type node assigned to PE(v).

The IPC node insertion algorithm which operates on the a-rate subgraph \( S(V_{a}, E_{a}) \) may be described as follows. For each edge \( e \in E_{a} \), with \( u \xrightarrow{e} v \) such that \( p(u) \neq p(v) \), the edge \( e \) is replaced with a path of the general form:

\[
\begin{align*}
  u \xrightarrow{e} & \text{IN}^{*}(\text{BusCount}(1)) \xrightarrow{e} \\
  \text{TRAN}^{*}(\text{BusCount}(1),\text{BusCount}(2)) \xrightarrow{e} \\
  & \ldots \\
  \text{TRAN}^{*}(\text{BusCount}(n-1),\text{BusCount}(n)) \xrightarrow{e} \\
  \text{OUT}^{*}(\text{BusCount}(n)) \xrightarrow{e} v.
\end{align*}
\]

The number of \( \text{TRAN}(\cdot, \cdot) \) IPC nodes will depend on the number of IPC busses separating PE(u) and PE(v): if they are tied to the same IPC bus, then none will be needed. One IPC node is assigned to each processor along the unique path from PE(u) to PE(v). As they are introduced into the graph, IPC nodes are tagged with the number of the PE to which they are assigned. The direction of each IPC node will be determined by the direction of communication along the IPC busses between PE(v) and PE(u). An integer array BusCount is associated with each IPC bus in order to facilitate the scheduling of slots on that bus. When an IPC node is inserted into the graph, it is assigned the slot time given by the value of its BusCount variable, and then the variable is incremented. In this way, unique slot times are assigned to each IPC node for each IPC bus. This process is best described by Figure 4-10. The insertion of IPC nodes in \( S(V_{a}, E_{a}) \) yields the IPC-scheduled a-rate subgraph \( S_{a}(V_{a}, E_{a}) \).

It should be noted here that much redundancy in the scheduling of IPC slots may occur when the data produced by a node on one PE is needed by several nodes residing on several different PE's. It may happen that the same variable is scheduled on the same IPC bus more than once. This may be
Figure 4-10: IPC Node Insertion
avoided by simply checking to see whether the immediate ancestor of node \( v \) is an OUT node of the direction that would otherwise have been inserted. If this is the case, then the previously inserted IPC node path may be traced backwards to see how close to the desired destination PE the variable has been transmitted. The new IPC path could then exclude those IPC nodes representing redundant communication on the busses, and thereby keep the amount of IPC down to a minimum.

IPC nodes serve not only to schedule slot times on the IPC busses; they also represent the delays incurred by IPC. It will be recalled that variables received from or transmitted on an IPC bus are buffered by a one level FIFO buffer. IPC nodes therefore represent not only the I/O port operations performed by the TMS320, but also the implementation of these one level buffers; consequently, each node further represents a one sample delay in the flow of data from the source to the destination. This allows us to compensate for the delays incurred by IPC by placing other delays elsewhere in the computation, thus delaying the final result, but ensuring the equivalency of the parallel processed task to the computation specified in the original synthesis program. This is described in the next Section.

4.4 Task Equivalency Routine

In the preceding Section, the subtasks created by the task partitioning algorithm were isolated from one another by the inclusion of IPC nodes along the edges representing data dependencies between subtasks. It was seen that each IPC node along a path between two computational nodes represented a one sample delay in the data flow from the source node to the destination node. These delays, of course, change the nature of the music synthesis computation; unless they are compensated for, the synthesis task implemented in the parallel processing environment will not correspond to the one specified in the synthesis program. In this section, we describe an algorithm which can compensate for these delays by introducing other delays elsewhere in the computation graph, such that the new graph is computationally equivalent to the original one, except that the final output is delayed by a certain number of samples.

One way to initially approach the problem of compensating for IPC delays is to add additional delays to other edges to effectively pipeline the computation. Consider the simple example presented in Figure 4-11a, which represents the original (a-rate) synthesis graph built from the Music 320 source program. Note that edge \( e_3 \) is a FORWARD-referenced edge with a delay of four samples.
Figure 4-11: Compensating for IPC Delays
specified. Now suppose that, due to the task partitioning algorithm, $v_4$ is assigned to a different PE than the rest of the nodes in the graph, and as a result of IPC node inclusion, we find that the delay incurred along edge $e_4$ is two samples, as is represented in Figure 4-11b. Now, in order to preserve the synchrony of the data values presented to both inputs of node $v_2$, we may simply insert a delay of two samples on edge $e_2$, as is shown in Figure 4-11c. This, of course, delays the output of $v_2$ by two samples, and hence the input to $v_4$ lags the input from $v_2$ by two samples. This may be corrected for, however, by simply subtracting two samples of delay from edge $e_3$. The resulting graph in Figure 4-11d is thus computationally equivalent to the original graph in Figure 4-11a, except that the output of node $v_4$ has been delayed by two samples.

To transform the original synthesis graph into an equivalent, delayed version, the concept of retiming is introduced [12]. Retiming involves the construction of a constraint graph $C(V_a,E_a)$ from the original $a$-rate subgraph $S(V_a,E_a)$, and by solving a system of linear constraints on $C$, delays may be placed on $S(V_a,E_a)$ such that the computation represented by the retimed graph is equivalent to that of the original synthesis graph. The constraints will seek to place delays in the retimed graph which correspond to the delays imposed by IPC. The delays in the retimed graph which do not correspond to IPC delays will then have to be implemented by constructing delay lines in the PE's, which, as we have seen previously in our discussion of the TMS320, is relatively simple and efficient. With the inclusion of these extra delays, the parallel implementation of the synthesis task will be a delayed, but computationally equivalent, version of the original synthesis task.

The retiming algorithm presented below defines a lag function $r(v)$ for $v \in V_a$ with the lag value of the root (i.e., OUTPUT node) $r()$ defined to be 0. The lag $r(v)$ of a node $v$ specifies the lag between the computation of the result of node $v$ for a given sound sample and the actual appearance of the sound sample at the OUTPUT node. Since the root node will have the maximum lag of any node in $V_a$, the lag values assigned by the retiming algorithm will be negative.

Values for the function $r(v)$ are found by means of performing a simple algorithm on the constraint graph $C(V_a,E_a)$, which will be described in detail further on. For now, though, assuming that the lag values for all the $a$-rate nodes have been found, let us define a new function $w(e)$ on the edges of $E_a$ to designate the number of delays that must be placed on $e$ in order to implement these computational lags. It can be seen that $w(e)$ is defined by the following relation, with $u \rightarrow v$:

$$w(e) = d(e) + r(u) \cdot r(v) \tag{4-3}$$
with $d(e)$ designating the delay specified in the synthesis program between nodes $v$ and $u$, which was defined in the graph construction (see Section 4.1). This is easily verified from the fact that, with the computational flow proceeding from $v$ to $u$, the term $r(u) - r(v)$ corresponds to the lag between the computation of node $v$ and that of node $u$, and is therefore the number of extra delays that must be placed on $e$ along with those specified by $d(e)$ in order to implement this lag.

It can be seen that certain constraints apply to the assignment of values to $r(v)$. One such constraint is the legality constraint, which prohibits negative values of $w(e)$, since negative delays will be impossible to implement. So for $w(e) \geq 0$, we have

$$d(e) + r(u) - r(v) \geq 0$$

or

$$r(v) - r(u) \geq d(e). \quad (4-4)$$

Another constraint is imposed by the delays due to IPC, which can be determined from the IPC-scheduled synthesis subgraph $S(V_s, E_s)$. That is, if we define a function $i(e)$ on the edges in $E_a$ from the original $a$-rate subgraph $S$ to be the number of delays incurred along the data dependency represented by $e$ as a result of IPC, it can be seen that we would like to assign values to the lag function $r(v)$ such that $w(e) \geq i(e)$. It is this IPC constraint that allows the compensation for IPC delays. Rewriting, we find that

$$d(e) + r(u) - r(v) \geq i(e)$$

or

$$r(v) - r(u) \geq d(e) - i(e). \quad (4-5)$$

Notice that, since $i(e) \geq 0$, if the IPC constraint is satisfied by a retiming, then the legality constraint given in Equation 4-4 is also satisfied.

Values for the lag function $r(v)$ are found by constructing a constraint graph $C(V_a, E_a)$, which has the same structure as the $a$-rate subgraph $S(V_a, E_a)$ except that the edges in $E_a$ are assigned a constraint function $c(e)$ derived from the IPC constraint equation. In other words, for all edges $e \in E_a$,
\[ c(e) = d(e) \cdot i(e). \]  

Thus, for each edge \( e \in E_a \) with \( u \xrightarrow{e} v \), we have the constraint

\[ r(v) - r(u) \geq c(e). \]  

(4-7)

The constraint graph \( C \) will therefore describe a system of \( m = |E_a| \) linear inequalities of the form

\[ X_i - X_j \geq Y_{ij}. \]  

(4-8)

It is shown in [12] that this system of linear constraints can be solved by a dynamic programming approach, specifically by using the Bellman-Ford single-source shortest-paths algorithm. This algorithm finds the length of the shortest path from a single source (here, the OUTPUT node) to every other node in the graph, where the path length is defined to be equal to the sum of the constraint functions for each edge on the path. The path weight is well-defined as long as there are no negative weight cycles contained in the constraint graph \( C \). If the path weight is well-defined, then the system of linear inequalities represented by Equation 4-8 can be solved by simply assigning the path weight to the \( X_i \)'s. This in turn defines the lag function \( r(v) \) for each node in \( V_a \), from which the retimed edge delays \( w(e) \) may be found, using Equation 4-3.

The Bellman-Ford single-source shortest-paths algorithm is now presented, which determines the weight \( u_i \) of the shortest path from \( v_o \) (the root or OUTPUT node) to \( v_i \in V_a \). The algorithm also indicates whether the system of \( m \) inequalities in Equation 4-8 is unsolvable due to the presence of a negative-weight cycle in \( C \).
Bellman-Ford Single-Source Shortest-Paths Algorithm

1. for i ← 0 to |V_a| do u_i ← 0; 
2. for k ← 1 to |V_a| do 
   3. for e_{ij} ∈ E_c do 
      4. if u_j > u_i + w(e_{ij}) then u_j ← u_i + c(e_{ij}); 
      5. unsatisfiable ← FALSE; 
   6. for e_{ij} ∈ E_c do 
      7. if u_j > u_i + c(e_{ij}) then unsatisfiable ← TRUE; 

Upon completion of the algorithm, if unsatisfiable = TRUE, then a negative-weight cycle exists in the constraint graph, and the system of linear equalities in Equation 4-8 is unsolvable. If, on the other hand, unsatisfiable = FALSE, then the weights u_i found in the algorithm satisfy Equation 4-8, and they can be assigned as values for the lag function r(v_i), v_i ∈ V_a, to satisfy the constraints in Equation 4-7. The complexity of the algorithm can be seen to be O(|V_a||E_a|), since line 2 is executed |V_a| times, and for each iteration, line 3 is executed |E_a| times.

While a proof as to the validity of the above outlined procedure for assigning values to the r(v)'s is given in [12], a more intuitive explanation of the workings of the algorithm may be more suitable here. Assume, for the moment, that the original a-rate synthesis subgraph S(V_a,E_a) contains no cycles (i.e., no FORWARD-reference data dependencies), and that as a result, d(e) = 0 for all e ∈ E_a. From the IPC constraint given in Equation 4-5, and from the definition of c(e) in Equation 4-6, it follows that c(e) will be non-positive for all edges in the constraint graph. Therefore, the weights u_i of the shortest path from v_0 to v_i computed by the Bellman-Ford algorithm may be viewed as the negative of the longest delay from the computation represented by v_i to the computation of the OUTPUT node. By assigning u_i to r(v_i), then, we are in fact lagging the computation of node v_i to correspond to the longest delay incurred due to IPC from v_i to the OUTPUT node. If we now consider occurrences of FORWARD-referenced data dependencies, with d(e) defined (in the synthesis program) to be some positive integer, we see that the value of c(e) for these edges will be positive, and that (hopefully) the sum of the c(e)'s along the edges of the directed circuit containing e
will be positive, so that the minimum-weight paths for the nodes on the circuit will be well-defined. This of course corresponds to the programmer's desire to specify a delay-fix parameter for FORWARD-reference dependencies large enough to offset any IPC delays in the feedback circuit.

Once \( r(v) \) has been computed for all \( v \in V_a \) by the Bellman-Ford algorithm, and the retimed edge delays \( w(e) \) found by applying Equation 4-3, only one other step is required to calculate the delays that must be inserted into the IPC-scheduled synthesis graph \( S_i \) to compensate for delays due to IPC. This simply involves defining a new edge delay function \( f(e) \) to designate the number of FIFO delays that must be implemented along a data dependency represented by \( e \) in the TMS320 object code. Since \( w(e) \) represents the total number of delays required in the retimed synthesis computation, and since the number of delays already existing due to IPC is given by \( i(e) \), the FIFO edge delay function \( f(e) \) is easily seen to be

\[
f(e) = w(e) \cdot i(e)
\]  (4-9)

for all \( e \in E_a \subseteq E_r \). If \( f(e) \) is non-zero for an edge \( e \) along which IPC is scheduled (that is, for an edge \( e \) for which \( i(e) \) is non-zero), then the FIFO delay will be implemented on the edge from \( u \) to the IN IPC node, since this edge was retained in the IPC-scheduled graph \( S_r \). This is an important consideration when IPC scheduling is performed to exclude redundant IPC node insertion, to ensure that the implementation of \( f(e) \) does not affect other edge delays.

The process of compensating for delays due to IPC may now be summarized as follows:

1. Find \( i(e) \) for all \( e \in E_a \) from the IPC-scheduled subgraph \( S(V_s,E_r) \).

2. With \( i(e) \) and \( d(e) \) defined during the construction of the synthesis graph \( G(V,E) \), construct a constraint graph \( C(V_a,E_a) \) and define the constraint function \( c(e) \) from Equation 4-6.

3. Assign values to the lag function \( r(v) \) for all \( v \in V_a \) by performing Bellman-Ford single-source shortest-paths algorithm on \( C \).

4. Find the retimed edge delay function \( w(e) \) from the lag function \( r(v) \) computed above using Equation 4-3.

5. Derive the FIFO edge delay function \( f(e) \) from the \( w(e) \)'s using Equation 4-9 to determine the edge delays that must be implemented in TMS320 object code to compensate for IPC delays.
The overall delay in the retimed computation will be seen to be equal to \( |R_{\text{min}}| \), where \( R_{\text{min}} \) is the minimum lag assigned in step 3, or

\[
R_{\text{min}} = \text{Min} \{ r(u), u \in V_a \}. 
\]

4.4.1 K- and I-Rate Delay Compensation

So far we have been considering only the effects of IPC delays on the a-rate subgraph \( S_f \); the retiming procedure discussed above compensated for IPC delays by assigning lags to the a-rate computation nodes. Unfortunately, this results in a loss of synchronization between k- and i-rate computations and those a-rate computations which were assigned a non-minimum lag during the retiming. For instance, if an a-rate node \( u \) with a data dependency on a k- or i-rate node is assigned a lag \( r(u) = -3 \) by the retiming algorithm, with the minimum lag assigned to any node \( R_{\text{min}} = -5 \), the effect of the retiming is to place a delay of \( r(u) - R_{\text{min}} = 2 \) samples between the computation of \( v \) and \( u \). This results from the fact that, if we define those a-rate nodes with lag equal to \( R_{\text{min}} \) to be synchronous with the k-frame process, then nodes with a non-minimum lag will consequently lag behind the k-frame process as a result of the pipelining implemented by the retiming. Similar arguments apply to a-rate node dependencies on parameter variables sent from the host processor. These delays resulting from the retiming may be compensated for in a very simple manner; we may place other delays on k- and i-rate node dependencies such that this effect is eliminated. This is done in the following fashion, by considering the entire synthesis graph \( G(V,E) \), and perform the following operation:

For every edge \( e \in E \), with \( u \xrightarrow{e} v \), such that \( u \in V_a \) and \( v \in V_k \cup V_i \) (the set of all k-, and i-rate vertices, respectively), let

\[
f(e) = r(u) - R_{\text{min}}. \tag{4-10}
\]

These new edge delays may be implemented in the same way as the previously defined edge delays \( f(e) \), by allocating contiguous spaces in the TMS320 data memory for a FIFO buffer, and using the data shift instructions in the a-rate code to effect the shifts.
4.4.2 Initialization of Delayed A-Rate Computation Nodes

Now that compensation for IPC delays has been accounted for in a-, k-, and i-rate computations, the only issue left to consider is how a-rate nodes which have been lagged behind the k-frame are to be initialized. The problem results from the fact that initialization takes place prior to the beginning of a k-frame, which means that a-rate nodes with non-minimum lag must be initialized in the midst of a k-frame iteration. The solution for this is a bit more troublesome than that for k- and i-rate delay compensation, and requires that we use conditional branch instructions to implement the initialization of a TMS320 data memory location by monitoring the iteration loop value (contained in an internal auxiliary register). If the value to which a node \( u \) is to be initialized is placed in the data memory location immediately preceding that of where the result of \( u \) is stored, a conditional data memory shift may be placed in the a-rate code such that the side effect takes place if:

1. An initialization has occurred as a result of a command from the host processor, and

2. the level of the iteration (from 1 to KSMPS) is equal to \( r(u) \cdot R_{\text{min}} \).

Such a conditional shift operation would then initialize \( u \) at the proper time in the k-frame. Note, however, that the use of conditional branch instructions in the a-rate code is subject to the same restrictions as were placed on the implementation of the MERGE operator: namely, that both branches contain the same number of instruction cycles.

4.5 Final Subtask Compilation

The graph algorithms discussed in the previous sections have partitioned the synthesis task, scheduled IPC, and compensated for IPC delays. The only step in the compilation process left to describe concerns the details involved in generating the assembly code for each of the TMS320-based PE's; a rough outline of these details will be summarized in the sections below.

4.5.1 Final Subtask Definition

Up to this point, only a-rate computation nodes have been assigned to PE's; to completely define a subtask, we must now assign all other computation/variable nodes in the set \( V - V_a \) of the synthesis graph \( G \) to PE's. This simply involve defining subtask graphs \( P_n(V_n, E_n) \) which are derived as follows:

For \( n = 1 \) to (\# of PE's), construct \( P_n(V_n, E_n) \) out of the a-rate subtasks defined by the
task partitioning algorithm and all IPC nodes inserted for PE n, and include all other nodes \( v \in V - V_a \) in \( V_n \) which can be reached by a path consisting of edges \( e \in E - E_a \). Include these edges in \( E_n \). All node and edge markings defined either during synthesis graph construction or by the previously described compilation algorithms are copied onto \( P_n \).

The result of this straightforward process is to copy all k-rate, i-rate, constant, parameter, and table object nodes which are needed by an a-rate subtask defined in the task partitioning algorithm into the final subtask \( P_n \). This means, of course, that those nodes \( v \in V - V_a \) which are needed by more than one a-rate subtask are replicated by each final subtask; but as was mentioned previously in the discussion of the parallel architecture, this redundancy will not greatly affect the over-all performance of the real-time computation, since the a-rate computation represents the greatest computational burden.

4.5.2 Ordering of A-, K-, and I-Rate Nodes

Now that each subtask graph \( P_n \) has been defined for each PE, we may focus our attention on how each subtask is transformed so that a source program of the form shown in Figure 2-5 may be defined. From Figure 2-3, it can be seen that separate a-, k-, and i-rate code blocks must be generated and placed within the designated flow structure. This simply involves grouping a-, k-, and i-rate nodes separately, and generating the assembly code from the microcode definitions of the primitive operators. Care must be taken, however, to ensure that the nodes are ordered correctly within each of the computation rate groups, or else unexpected delays will result due to violated input and output dependencies in the computation. This ordering is made easy, however, by the fact that the computation nodes were marked with an integer during the graph construction, corresponding to the statement number in which the primitive occurred (after all macro operators were expanded). Thus, if the nodes in each grouping are arranged in ascending order according to the statement number in which it was defined, then all input and output dependencies are satisfied in conjunction with the first assignment rule, described in Section 3.4.

4.5.3 Assignment of TMS320 Data Memory Locations to Nodes

Now that different rate computation nodes have been grouped and ordered, we may consider how TMS320 data memory locations are to be assigned. For the most part, the exact location is arbitrary, but special considerations relating to the implementation of FIFO's for IPC nodes and
compensation delays will be of interest here. The case of IPC node FIFO’s has already been described: two contiguous data memory locations are reserved for the implementation of a one level FIFO. The construction of a FIFO delay for implementing the correction delay on an edge \( e \) with \( f(e) \) defined to be some positive integer is implemented in a similar fashion: \( f(e) + 1 \) contiguous locations in data memory are assigned to a computation node, with the result of the computation being placed in the lowest location, and the data dependency taken from the highest location. In the case of a computation node \( u \) with \( d_{\text{in}}(u) > 1 \) in which several of the incoming edges have been assigned a non-zero correction delay, data memory locations (as well as data shift operations) may be conserved by reserving \( F_{\text{max}} + 1 \) contiguous data memory locations for that node, where

\[
F_{\text{max}} = \text{Max}\{ f(e), \text{for all } e \text{ with endvertex } u \}.
\]

Data dependencies on \( u \) may then refer to various levels of the FIFO, depending on the correction delay \( f(e) \) assigned to that edge. Note that an IPC FIFO for an OUT node may also be incorporated into this generalized data memory assignment scheme, by simply referring the microcode output instruction to the first level of the correction FIFO. This is illustrated in Figure 4-12.

The FIFO implementation scheme presented above also applies to a delay \( f(e) \) on a parameter variable node. Constant-type nodes are always assigned a single location in data memory. The only nodes for which we have not yet discussed memory allocation are table object nodes; this is described in the following section.

4.5.4 Assignment of Table Memory Locations to Table Objects

A PE’s table memory is allocated in a very straightforward manner, with table objects being assigned blocks of table memory of size corresponding to the number of memory locations designated by the table object definition in the synthesis program. Once all table objects in a subtask graph \( P_n \) have been allocated, the values of the table attributes may be determined. Table attribute dependencies in \( P_n \) may now be implemented as constant-type nodes, i.e. replaced by true data dependencies on constant nodes generated from these values.
Figure 4-12: Implementation of Correction FIFO's
4.5.5 Generation of A-, K-, and I-Rate Assembly Blocks

A-, k-, and i-rate nodes have been grouped together and ordered in previous stages of the final subtask compilation, so we must now consider the steps involved in the final translation from the ordered graph representation of each rate of the subtask into TMS320 assembly language. Let us first examine the generation of the a-rate code block.

4.5.5.1 A-Rate Code Block Generation and Final Scheduling of IPC

The a-rate code block will consist of the concatenation of two phases of operation: the first is simply the translation of a-rate primitives into their assembly language equivalents, interleaved with IPC-port I/O instructions; the second consists of the data shift operations which are necessary for the implementation of IPC node and delay compensation FIFO's.

The interleaving of IPC-port I/O instructions within the a-rate assembly code is complicated somewhat by the fact that we have implemented the MERGE operator with conditional branches. IPC slots occurring within a MERGE operator must therefore be scheduled in both arms of the conditional branch. Another consideration emerges from the fact that all PE's except the leaf PE's in the tree architecture will be communicating over two IPC busses. Many complications resulting from trying to schedule both busses efficiently may be overcome by noting that, according to previous estimates, a slot time will occur over four instruction cycles, while an IPC-port I/O instruction requires only two instruction cycles.

In appending the data memory shift operations to the end of the a-rate code block, the only care that must be taken is to perform shifts on the upper memory locations of a FIFO before the lower ones, so that every FIFO will be properly implemented. Also note that, included in the data shift portion of the code, are the conditional shift operations required for the implementation of initialization of a-rate nodes which were lagged behind the k-frame process as a result of retiming the synthesis graph.

4.5.5.2 K- and I-Rate Code Generation

In the generation of k- and i-rate assembly code blocks, the ordered nodes must be further grouped according to the instrument block in which they were defined. Since a PE may have been assigned a subtask containing computation nodes from several different instruments, each decision
fork occurring at the end of a k-frame process shown in Figure 2-3 must contain the code corresponding to the instrument for which a possible initialization is to be determined. This is facilitated by the fact that the instrument block in which a k- or i-rate node was defined is recorded in the node. Once k- and i-rate nodes have been further grouped according to the instrument in which they were defined, the assembly code generation may proceed by translating the primitive operator nodes into their assembly code equivalents.

The i-rate code block is responsible not only for the computation of i-rate primitives, but also the initialization of primitive operators which specified an initialization variable in the last field of the Music 320 definition. In the case of k- and i-rate primitives, this simply involves a side effect on the variable's data memory location; this is also the case for the initialization of a-rate nodes which were assigned a minimum lag, since their computation is synchronous to the k-frame. The initialization of a-rate nodes not assigned a minimum lag must be implemented by placing the initialization value in the data memory location immediately preceding the variable to be initialized, as was outlined in a previous Section. Therefore, this transportation of the initialization value must be effected in the proper i-rate code block, as well as the setting of a flag to designate to the conditional shift operator residing in the a-rate code block that an initialization is to take place. This flag may be the same one that effected the fork between the k-rate and the i-rate code blocks; a semaphore will also be needed to reset the flag after the k-frame following the initialization of the instrument.

4.5.6 Final Assembly Code Generation

Now that the assembly details for each of the different rate code blocks has been described, there is little left to consider in the generation of the final assembled subtask to each processing element. The code blocks are simply placed in the context of the k-frame parallel process in Figure 2-3. A-rate code blocks for each PE are made to contain the same number of instruction cycles by inserting NOP instructions at the end of all but the longest code block. One point that may finally be considered, once the k-frame code for each PE in the parallel architecture has been assembled, is the maximum sampling rate that can be achieved in real-time operation of the synthesis task. This can be computed from the maximum number of instruction cycles involved in the worst-case computation of any k-frame subtask, where “worst case” refers to the non-deterministic nature of the program flow after the computation of a frame of a-rate samples. Once the maximum sampling rate has been determined, the settings of the sampling rate clock may be chosen to correspond to the next lowest
available cutoff frequency of the output low-pass filter(s).

4.5.7 Linking to the Host Processor

Now that the PE's have all been assigned an assembled sub-task, there is yet one more consideration that must be dealt with: the co-ordination of the host processor and the parallel processing environment. Much of this depends on exactly how the command FIFO interfacing is implemented in hardware, and so unfortunately cannot be discussed here. It is important to note, however, that the computation pertaining to an instrument may reside on several PE's, and that furthermore, due to the possible replication of an instrument's parameter values, the same parameter may also reside on several PE's. Therefore, it will be necessary to define a table of instrument parameter variables containing the location(s) to which they must be sent during real-time synthesis. It will also be necessary to synchronize the initialization process on all PE's so that the note initialization occurs after all the parameter values for a note event have been sent to the proper PE's in the parallel processing environment.
Chapter Five

Conclusions and Suggestions for Further Research

The preceding chapters have delineated an approach to implementing real-time computer music synthesis in a parallel processing environment. Although the system as presented above is complete in all theoretical aspects, there are still many unresolved issues and problems which may be encountered in its actual implementation. This concluding chapter hopes to anticipate some of the problems in the practical implementation of the parallel processing system, and to offer suggestions for possible future research.

5.1 Discussion of the Parallel Processing System

The parallel processing system described in the preceding Chapters was designed in the hope of providing a powerful, yet flexible, solution to the problem of real-time computer music synthesis. While it is believed that the solution presented provides an attractive alternative to previous digital synthesis systems, it would seem that there are still many inadequacies in the system that must be overcome before its advantages may be practically realized. What follows is an attempt to outline the major weak points in the present parallel processing approach with respect to the three main areas of its presentation: architecture, synthesis language, and compilation strategy.

5.1.1 Problems with the Parallel Processing Architecture

Probably the greatest disadvantage with the architecture presented in Chapter 2 is the fact that the TMS320 is only a 16-bit machine, which means that the parallel processor will have a difficult time maintaining even 12 bits of accuracy, especially for large instruments. It may be noted, however, that the primitive operators may be extended to provide double precision arithmetic, since the TMS320 has a 32 bit accumulator. Of course, the cost of doing this is an increase in the number of instruction cycles for the arithmetic operators: ADD and SUB would require twice as many instruction cycles, and MULT about three times as many.
Another possibility is to take advantage of the TMS320's ability to carry 32 bits of precision through repeated additions or multiply-adds into the accumulator. This would involve separate optimization of each subtask prior to the generation of object code. Of course, one might also await the development of true 32-bit signal processing microprocessor chips. In any case, if the parallel processing approach presented in this thesis is to be expected to produce high-quality digital audio, the issue of maintaining adequate precision throughout the sample computation must be dealt with.

5.1.2 Problems with the Music 320 Synthesis Language

The Music 320 language was intended to be as simple as possible, not only to facilitate the compilation process, but the explanation of the compilation process as well. It would seem, though, that some of the clumsiness of using the language might well be avoided by effecting some modifications on both the assignment rules and on certain stages of the compiler.

One of the main inconveniences stems from the fact that the programmer is required to specify the expected number of delays incurred due to IPC within a feedback loop when declaring a FORWARD-referenced variable. This admittedly involves a bit of guesswork. It may on the other hand be more convenient if, instead of specifying the delay, the programmer is able to specify a constant which will take on the value of the loop delay once the task equivalency routine has been completed. For example, in the macro implementation of the Music 11 DELAY command (see Section 3.4.1), a3 would be defined as

\[ a3 \quad \text{FORWARD} \quad c4 \]

and c3, which represents the physical address at which the incrementing PHASOR must return to the start address of the delay table, would be defined as

\[ c3 = t1(\text{last}) - c4. \]

The value of c4 would be determined during the retiming algorithm, and so would be available (along with t1(\text{last})) when the value of c3 is computed during the final subtask compilation.
5.1.3 Problems with the Compilation Strategy

The task partitioning strategy presented in Chapter 4 was based upon the heuristic assumption that a typical synthesis program, when transformed into its graph representation, may be roughly approximated by a tree structure. If a majority of the data dependency edges in the synthesis graph may be included in a spanning tree subgraph, then a partitioning of the spanning tree into subtrees (corresponding to subtasks) will in some sense minimize the amount of IPC required between PE's. If the subtrees may be generated such that the number of instruction cycles in each subtask is approximately the same, then the original synthesis task will have been evenly and efficiently distributed among all the PE's in the parallel processing architecture.

Several problems may exist in the proposed partitioning strategy, not the least of which being the robustness of the claim that most of the data dependencies of any given synthesis program may be represented by a tree structure. While the reasons given in Section 4.2.1 to defend the claim may certainly be valid, there is always the possibility that a composer may wish to run a program having a very dense graph representation. The performance of the partitioning strategy (with regards to IPC minimization) under these conditions will undoubtedly be much worse, although it may still be acceptable when it is acknowledged that the efficient partitioning of a dense graph among a nominal number of PE's (6 to 30, perhaps) is a very difficult, if not impossible, problem to solve optimally.

In the partitioning of dense synthesis graphs, the choice of spanning tree becomes more critical. Recall that the spanning tree selection in Section 4.2.2 is based on a depth-first search algorithm, which yields a spanning graph that is not unique to a given synthesis graph (unless, of course, the original synthesis graph was itself a tree). For dense graphs, the efficiency of the partition with respect to the amount of IPC scheduling necessary may be dependent on the choice of spanning tree. A possible solution might be to favor those dependency edges for inclusion into the spanning tree which are more strongly bound to a densely connected region in the synthesis graph, thus keeping computation nodes of dense regions of the graph in the same part of the spanning tree. The hope would then be that a greater portion of the non-spanning tree dependencies would occur between nodes residing on the same PE after the partition, thus reducing the amount of IPC necessary.

Another consequence of partitioning dense synthesis graphs is the possible bottlenecking of IPC around the root PE of the parallel architecture. Because of the tree structure of the architecture, many communications between PE's must be routed near the root PE, which may lead to a significant amount of added overhead due to IPC for PE's in the first few generations. This could affect the
overall speed of the machine, since the sampling rate is determined by the largest code block. Bottlenecking may be compensated for, however, by modifying the re-weighting constant \( a \) discussed in Section 4.2.3, such that a smaller portion of the spanning tree is allotted to the father PE's for the clusters in the first few generations.

One aspect of the partitioning strategy which would be of more immediate interest for further study is the efficiency of the min-max tree partitioning algorithm. The prime motivation for implementing the min-max partition is to split the task into equally sized subtasks, and it would be of great practical importance to determine how the efficiency of the partition is affected by the choice of spanning tree, the number of cuts, and the overall structure of the synthesis program. It may be advantageous to perform manipulations on the synthesis graph to produce a graph that will yield a more efficient cut. A good example of where this might be effective is in the re-association of the ADD operators used for summing instrument outputs.

Another possible problem associated with the task partitioning algorithm concerns the performance of the recursive approach to task partitioning for the balanced tree architecture. This is directly related to the above topic, since an imbalance in a min-max cut at the first generation will propagate down to all the remaining generations; one might argue intuitively that the variance in code block sizes over the entire processing tree would be proportional to both the expected variance of a single min-max partition and the number of generations in the balanced tree architecture. More satisfactory results may be obtained by simply performing a single min-max partition with N-1 cuts (N being the total number of PE's), and by developing a placement algorithm to assign the subtasks to PE's in a manner which facilitates IPC.

One last set of comments applies to the task equivalency portion of the compiler. In compensating for the delays due to IPC, the retiming algorithm is capable of placing a large number of delays on the synthesis graph, which must be implemented as FIFO's by the TMS320. It may be possible to reduce the number of delays necessary to correct for IPC, thus reducing some of the non-computational overhead. For example, when a large instrument, requiring a great amount of computation spread over a number of PE's, is summed with a smaller instrument residing on only a single PE, the uncompensated output of the larger instrument will probably lag the output of the smaller instrument. When the retiming algorithm as described in Section 4.4 is used to compensate for IPC, it will do so by placing delays in the smaller instrument on the data dependencies from a-rate nodes to k- or i-rate nodes. Clearly it would be more efficient to place a single FIFO delay at the
output of the smaller instrument. This more efficient retiming may be implemented by modifying the retiming algorithm to define the lag function for each node using an all-pairs shortest-path algorithm. The lag function for each node would then be defined as the shortest distance from that node to any k- or i-rate node, given the constraint graph constructed as before. The details of this retiming technique are described in [12]; the solution to a similar problem is also given by Gao [7] in connection with pipelining data flow programs.

5.2 Conclusion

This thesis has presented what is believed to be a viable alternative to current real-time computer music synthesis systems. In addition, some of the issues concerning how a parallel processing system might be used to take advantage of the high degree of parallelism in music synthesis computation have been addressed, and solutions to many of the problems involved were presented. The design of the parallel architecture, synthesis language, and compiler were ultimately influenced by the availability of a new high-speed microprocessor chip, and by heuristic assumptions about the nature of synthesis computation graphs. The project was also designed to be economically feasible while providing a reasonably high-power solution to the problem of real-time music synthesis.
Appendix A

Min-Max Tree Partitioning Algorithm

The min-max tree partitioning algorithm originally given in [2], which forms the basis for our task partitioning compiler, is presented here for the reader’s convenience. Let us first restate the problem: given a directed tree $T(V,E)$, with a non-negative weight $w(v)$ associated with each vertex $v \in V$, $q$ subtrees $T_1, T_2, \ldots, T_q$ may be generated by placing cuts on $k = q - 1$ edges of $T$. We shall define the weight $W(T_i)$ of each subtree $T_i$ to be the sum of the weights of its vertices. We wish, then, to place $q$ cuts on $T$ such that $\text{Max}\{W(T_i), 1 \leq i \leq q\}$ is minimized.

Before presenting the algorithm, we must define the following terminology. The \textit{down-component} of a cut $c$ is the subtree rooted at the end-vertex of the edge on which the cut is placed. Similarly, the \textit{root-component} is the subtree with the same root as the original graph $T$. The \textit{son-edge} $e_s$ of an edge $e$ is any edge for which $\text{start-vertex}(e_s) = \text{end-vertex}(e)$. Also, the son-edge of a vertex $v$ is an edge $e_s$ having start-vertex $v$.

The algorithm is a shifting algorithm, in that all cuts are originally placed on an edge incident to the root of $T$ and shifted during the course of the algorithm. A temporary root node with zero weight is constructed with a single outgoing edge to the original root to allow this initialization. The cuts may be \textit{down-shifted} or \textit{side-shifted} from one edge to another: a down-shift involves shifting the cut from an edge to one of its son-edges; a side-shift occurs when a cut is shifted from one edge to another edge having the same start-vertex.
Shifting Algorithm for Min-Max q-Partition of a Tree [2]

1. Place all \( k \) cuts on the edge incident the original root. Set \( \text{BestMin-MaxSoFar} \leftarrow \infty \), and set \( \text{BestPartitionSoFar} \) equal to the starting configuration.

2. While the root-component is not a heaviest component, perform steps 3 - 5.

3. Find a cut with a heaviest down-component and down-shift it from its current edge \( e \) to a vacant son-edge having heaviest down-component. If no such vacant edge exists, then halt.

4. Traverse the path from \( \text{start-vertex}(e) \) to the root in the bottom-up direction until a vertex \( w \), which is the root of a currently defined subtree, is encountered. For each vertex \( w \) on that path having a cut on an edge \( e \) with \( \text{start-vertex}(e) = w \), do the following:
   
   If the down-component of a cut on an edge with \( \text{start-vertex} w \) is lighter than the down-component of the vacant son-edge \( e_s \) of \( w \) on the path, then side-shift that cut to edge \( e_s \). If more than one such cut can be side-shifted, then choose that cut with the lightest down-component.

5. Set \( \text{LargestWeight} \) equal to the weight of the largest component in the current partition \( A \). If \( \text{LargestWeight} < \text{BestMin-MaxSoFar} \), then set \( \text{BestMin-MaxSoFar} \leftarrow \text{LargestWeight} \), and set \( \text{BestPartitionSoFar} \leftarrow A \).

The final value of \( \text{BestPartitionSoFar} \) gives the resulting min-max partition. A proof of the validity of the algorithm is given in [2], as well as an analysis of its complexity, which is found to be \( O(k^3 \text{rd}(T) + kn) \), where \( \text{rd}(T) \) is the number of edges in the radius of \( T \), and \( n \) is the number of vertices in \( T \).
References


A Shifting Algorithm for Min-Max Tree Partitioning.


[4] Chowning, John M.


[6] Dennis, Jack B.
Data Flow Supercomputers.

An Implementation Scheme for Array Operations in Static Data Flow Computers.

Notes on Very-Large-Scale Integration and the Design of Real-Time Digital Sound Processors.
Dependence Graphs and Compiler Optimizations.

Special-purpose Devices for Signal and Image Processing: An Opportunity for VLSI.

[11] Le Brun, M.
Digital Wave-shaping Synthesis.

[12] Leiserson, C. E., Rose, F. M., and Saxe, J. B.
Optimizing Synchronous Circuity by Retiming.

[13] Loy, Gareth D.

[14] Mathews, M.
The Technology of Computer Music.

The 4C Machine.

[16] Moorer, J. A.
Synthesizers I Have Known and Loved.

[17] Roads, C.
A Tutorial on Non-linear Distortion or Wave-shaping Synthesis.
[18] Roads, C.
Interview with Max Mathews.

[19] Samson, P.
A General Purpose Digital Synthesizer.

Graph Algorithms.

Texas Instruments, Houston Texas, 1982.

[22] Vercoe, Barry.

[23] Walraff, Dean.
The DMX-1000 Signal Processing Computer.