Application of Data Flow Architecture to Computer Music Synthesis

by

Carol Andrea Cesari

© 1981 by the Massachusetts Institute of Technology

February 1981

This work was supported in part by the National Science Foundation under the research grant MCS-7915255

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, MA 02139

This empty page was substituted for a blank page in the original document.

Application of Data Flow Architecture to Computer Music Synthesis

by

Carol Andrea Cesari

Submitted to the Department on Electrical Engineering and Computer Science
on February 2, 1981 in partial fulfillment of the requirements for
the Degrees of Bachelor of Science and Master of Science

Abstract

A computer music synthesis system is the most flexible of synthesis systems It offers a composer extensive control over the sound of his piece. A user of such a system describes his composition in some synthesis language. The computer uses this description to calculate samples of a voltage waveform that can be fed to D/A converters at a specified sampling rate. The D/As' outputs are in turn fed to loudspeakers that produce the sound of the user's composition. Real time performance is unattainable on existing computer synthesis systems due to the sequential nature of conventional computers. Unless the parallelism that is present in the sample calculation process is exploited, real time performance will remain unobtainable. This thesis presents a proposed computer synthesis system that includes a data flow machine, a computer whose architecture is highly parallel. The Music-11 synthesis system at MIT was used as a model in its design. An analysis of the algorithms used in the sample conversion process and how it would run on the data flow machine is presented. An example of how a compostion would be described in a synthesis language and how it would run on the proposed system is given.

Thesis Supervisor: Jack B. Dennis

Title: Professor of Electrical Engineering and Computer Science

Keywords: sound synthesis, voltage waveform, hardware synthesizers, real-time performance, orchestra file, score file, MUSIC-11, data flow, VAL, streams, pipelining.

This empty page was substituted for a blank page in the original document.

Acknowledgments

I would like to thank Prof. Dennis for the support, help and above all the encouragement in the preparation of this thesis. Special thanks go to the members of the Computation Structures Group for being near whenever a problem needed discussion. I express my gratitude to Barry Vercoe and the members of the Experimental Music Studio for educating me in the ways of music synthesis. I offer thanks to my family whose love has sustained me throughout my life and special thanks to my husband whose love will always nourish me and give me hope.

This work was supported in part by the National Science Foundation under the research grants MCS-7915255

This empty page was substituted for a blank page in the original document.

CONTENTS

1. INTRODUCTION 6

1.1 Hardware Synthesizers 7
1.2 Computer Synthesis Systems 9
1.3 Intent of Thesis 11
1.4 Thesis Outline 11

2. SYNTHESIS TECHNIQUES 13

2.1 Additive Synthesis 13

3. MUSIC-11 SYNTHESIS LANGUAGE 16

3.1 Orchestra File 17
3.2 Score File 21
3.3 Additional Orchestra File Information 26
3.4 Performance 30

4. DATA FLOW CONCEPTS 32

4.1 Program Graphs 32
4.2 Machine Architecture 36
4.3 VAL 42
4.4 Translation of VAL Code to Flow Graphs 44
4.5 Pipelined Flow Graphs 48
4.6 Streams 52

5. DATA FLOW IMPLEMENTATION OF MUSIC SYNTHESIS 60

5.1 Physical Layout of the Music-df System 60
5.2 Music-df Language 62
5.3 Orchestra File Conversion 66
5.4 Signal Functions 84
5.5 Function Tables 108
5.6 Sinusoidal Sum Function Table 108

1. INTRODUCTION

For several decades, sound synthesis has been used in the performance of music. Though the sound of traditional instruments can be mimicked using sound synthesis, its major attraction for many composers is the opportunity it provides to create totally new an unconventional sound.

Sound synthesis systems are the means to employ the technique of sound synthesis. Two basic types of sound synthesis are hardware synthesizers and computer synthesis systems. Both follow the basic design shown in Fig. 1.1. They all accept a set of controls from a user as input and produce a voltage waveform to be fed to one or more loudspeakers. The cones in the loudspeakers vibrate according to the voltage waveform applied to their terminals. These vibrations produce a pressure waveform that is perceived as sound. The difference between the two types of systems is the implementation of the controls and the manner in which the voltage waveform is produced.

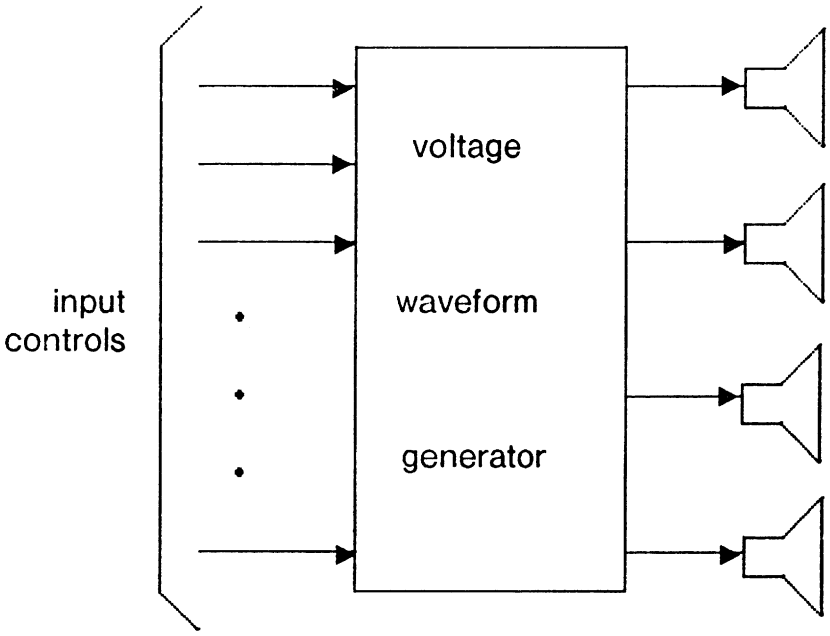
1.1 Hardware Synthesizers

Hardware synthesizers were the first synthesis systems to be built and are still in heavy use today. The major portion of their makeup is signal generators and modifiers that can be patched (cascaded) together to produce a voltage waveform. Among the list of components used are oscillators, envelope generators, mixers (signal adders) and filters. Suppose a composer wishes to produce the waveform:

$$X(t) = A_1(t)\sin\omega t + A_2(t)\sin 2\omega t$$

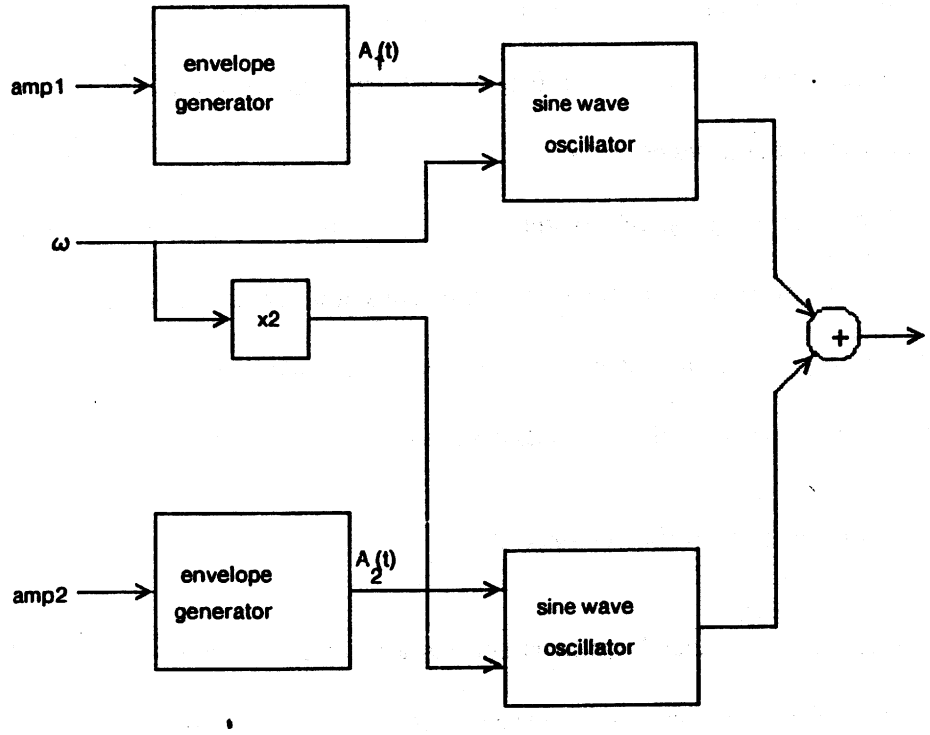
Then the components of the synthesizer would be connected as shown in Fig. 1.2. In this example, the controls that the user must provide are the amplitude (in units of voltage) to the envelope generators and fundamental frequency to the oscillators.

Originally the connections made between components were made through cords with plugs. But as synthesizer technology has progressed, a simple flick of the switch can connect two components in most synthesizers. The input to a component may be the output of another



SYNTHESIS SYSTEM

Fig. 1.1



INSTRUMENT BLOCK DIAGRAM FOR A

HARDWARE SYNTHESIZER

Fig. 1.2

component. It may also be a knob on the component that can be put at a constant setting or changed dynamically during performance by a human hand. Keyboards are generally used to control frequency (pitch). For example, in Fig. 1.2, the performer might input ω through a keyboard with one hand and dynamically adjust the amplitude input to the envelope generators by turning a knob with the other hand. The duration of the note would equal the time that the key is depressed.

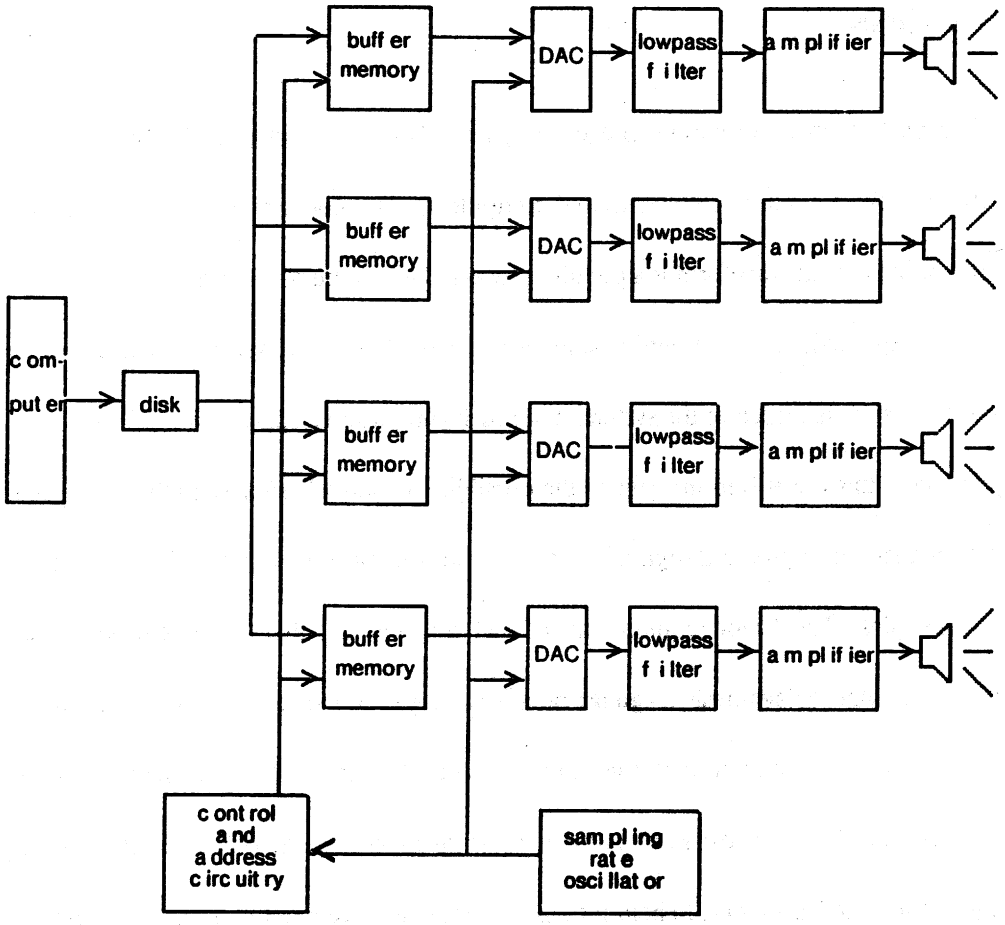
It is common practice for a performer to decide what kind of sound he wishes to produce, make the necessary connections and then play the instrument he has created by using the keyboard and manipulating the switches or knobs.

The major advantage of a synthesizer is that performance takes place in real-time due to the nature of the hardware components. The limitations however are apparent. If the performer wishes to dynamically change any other characteristics of his sound besides pitch, he must use knobs and switches. Since man has only two hands this can be difficult at times.

1.2 Computer Synthesis Systems

Most computer sound synthesis systems are some version of Fig. 1.3. A user enters a description of his composition in some computer language at the terminal. When he wishes to hear his composition, the user calls a program that takes his file as input and outputs a series of numbers that correspond to samples of a voltage waveform. Depending on whether the composer wishes mono, stereo or quadraphonic sound, samples are sent to one, two or four channels. Each channel contains buffer memory to hold the samples. The buffer contents are fed at the sampling rate to D/A converter. The filter smooths the D/A output and produces a continuous waveform. This waveform, after amplification, drives the loudspeaker to produce the musical sound.

The great advantage of such a system is the extensive control a composer has over the sound his piece. Details, down to the milisecond can be specified. Also, since signal generators and modifiers are in the form of coded functions, many instruments of arbitrary complexity can play



COMPUTER SYNTHESIS SYSTEM

Fig. 1.3

simultaneously. The major disadvantage of computer synthesis systems is that they cannot perform in real time. In order to acheive real time performance, one sample of the output waveform must be computed every 20-30 microseconds for good quality sound. Even for a simple instrument, real time performance is impossible on a synthesis system similar to the one in Fig. 1.3.

1.3 Intent of Thesis

Real time performance is unattainable on a synthesis system similar to the one in Fig. 1.1 because of the sequential nature of conventional computers. The conversion from the composer's files to samples of the output waveform is a parallel process. Unless the parallelism that is inherent in this conversion is exploited, real time performance is unobtainable. A computer whose architecture is highly parallel must be included in the system. One such computer is a data flow machine.

The purpose of this thesis is to investigate the possibility of a synthesis system that is capable of real time performance, through the design of a system using a computer whose architecture is based on data flow principles. The major requirements of such a system are that the system be easy for a composer to use and that the synthesis language be a natural way of expressing a composition. The flexibility and freedom of expression that presently exist in sophisticated computer synthesis systems should not be sacrificed. The Music-11 computer synthesis system at MIT and the Music-11 synthesis language are used as models to help incorporate these feature into the proposed system.

1.4 Thesis Outline

Chapter two presents some sound synthesis techniques to give the reader a feeling of some principles that composer might use in the composition of his piece. Chapter three details the Music-11 synthesis system and synthesis language. Chapter four explains data flow concepts and the architecture of the data flow machine that is used in this paper. VAL, a high-level language used in the expression of algorithms that are to run on a data flow machine, is presented along with streams, a

data type extension to the language. Chapter five describes the proposed real time computer synthesis system. An example of how the system would operate is presented in chapter six to expose the advantages and disadvantages of the system. Chapter six also draws some conclusions about the system and suggests further research.

2. SYNTHESIS TECHNIQUES

Musical sound can be described in terms of its physical attributes or its psychological attributes. Physical attributes include intensity, complexity (meaning the wave is not just a simple sinusoid but a sum of sinusoids), absorption and reverberation, resonance and modulation. These attributes can all be described in terms of the physical properties of the sound wave which are amplitude, frequency, period and phase. Psychological attributes include pitch, loudness, timbre, sound location and rhythm.

There is a direct correspondence between some physical and psychological characteristics. For instance, pitch depends on the fundamental frequency of the sound wave and what we perceive to be the sound location depends on absorption and reverberation. Counterintuitively, loudness is not a function of amplitude only but a function of amplitude and frequency.

Synthesis techniques help composers to produce desired psychological sound characteristics by varying parameters that can be controlled, that is the wave's physical attributes. Synthesis techniques are an area of active research. Two of the most commonly used techniques are additive synthesis and FM synthesis.

2.1 Additive Synthesis

Additive synthesis is based on the summation of harmonically related sinusoids:

$$X(t) = \sum_{k=1}^{M} A_k(t)\sin(k\omega + 2\pi F_k(t))t$$

where

$X(t)$ = output waveform as a function of time
ω = fundamental frequency in radians
k = harmonic number
$A_k(t)$ = amplitude of kth harmonic as a function of time
$F_k(t)$ = frequency deviation of the kth harmonic as a function of time

Theoretically, any waveform can be represented and hence any sound can be produced from a sum of

sinusoids. The problem lies in knowing which harmonics are present in a sound and also knowing their respective amplitudes and frequency deviations. Analysis of many different musical sounds has been done using filters that extract certain harmonics from a waveform. These studies have provided a large amount of information about different instrumental sounds. They allow us not only to reproduce sounds that are sometimes indistinguishable from the original but open the way to many new sounds.

The result of most sound analysis is the detection of which harmonics are present in a particular sound wave. Thus a big advantage of additive synthesis is that a composer can directly specify the harmonics in his waveform. The instrument block diagram in Fig. 1.2 is based on additive synthesis with M=2.

This popular technique is based on an equation that has been used for many years in radio transmission. It's use in sound synthesis was first suggested by Chowning[6]. It appears in his article in this form:

$$X(t) = A(t)\sin(\alpha t + I(t)\sin\beta t)$$

where

$X(t)$ = output waveform as a function of time
$A(t)$ = amplitude as a function of time
α = carrier frequency in radians/sec
β = modulating frequency in radians/sec
$I(t)$ = modulation index as a function of time

Using the identity:

$$\sin(\theta + a\sin\omega) = J_0(a)\sin\theta + \sum_{k=1}^{infin} J_k(a)(\sin(\theta + k\omega) + (-1)^k \sin(\theta - k\omega))$$

where $J_k(a) = k^{th}$ Bessel function at the point a, the FM equation can be expanded to:

$$X(t) = A(t)\{J_0(I(t))\sin\alpha t + J_1(I(t))[\sin(\alpha+\beta)t - \sin(\alpha-\beta)t] + J_2(I(t))[\sin(\alpha+2\beta)t - \sin(\alpha-2\beta)t] + \ldots\}$$

This last equation shows that we obtain sinusoids of frequencies α, $(\alpha\pm\beta)$, $(\alpha\pm2\beta)$, $(\alpha\pm3\beta)$ etc. As an example, if $\alpha = 2\pi*200$ and $\beta = 2\pi*100$ the resulting waveform will have sinusoids of 200, (100,300), (0,400), (-100,500) etc. Noting that $\sin(-\omega) = -\sin(\omega)$, this waveform will appear to have a fundamental frequency of 100 cps with all harmonics present. The amplitudes of these harmonics will depend on A(t), the index of modulation and the Bessel functions.

The FM synthesis technique is attractive because it allows a highly complex waveform to be expressed in one simple equation. However FM synthesis does not lend itself to analysis as easily as additive synthesis and it is difficult to develop an intuitive feel for the resulting sound. The modulation index determines the relative strengths of the fundamental and its partials. However if a composer knows which partials he would like to be present in his waveform, there is no method to derive an index of modulation that will produce the desired waveform. Thus, most most of the early studies of FM synthesis treated the FM equation empirically by plugging in different functions for A(t) and I(t) and listening to the results.

3. MUSIC-11 SYNTHESIS LANGUAGE

The EMS (Experimental Music Studio) facility at MIT uses a set-up much like the synthesis system shown in Fig. 1.3. Its processor is a DEC PDP-11/50 and the sound synthesis language is called Music-11.

Music-11, a language used for computer music composition, was developed by Barry Vercoe at MIT. The design of Music-11 is traditional in that it retains many of the concepts of hardware synthesizers. To play a composition using Music-11, the user must create two files, an orchestra file and a score file. The orchestra file is a collection of instruments described in terms of oscillators, filters, envelope generators etc. which are functions provided by Music-11. Thus, the orchestra file may be likened to the components of a hardware synthesizer and the connections made among them.

The score file is a list of notes that are to be played on each instrument. For each note the following information is needed: the instrument on which the note is to be played, the starting time of the note, the duration of the notes and several parameters that are to be passed to the orchestra file. The contents of the score file are analogous to the input controls a player gives to a hardware synthesizer while performing. The interaction between these two files may be viewed as in Fig. 3.1.

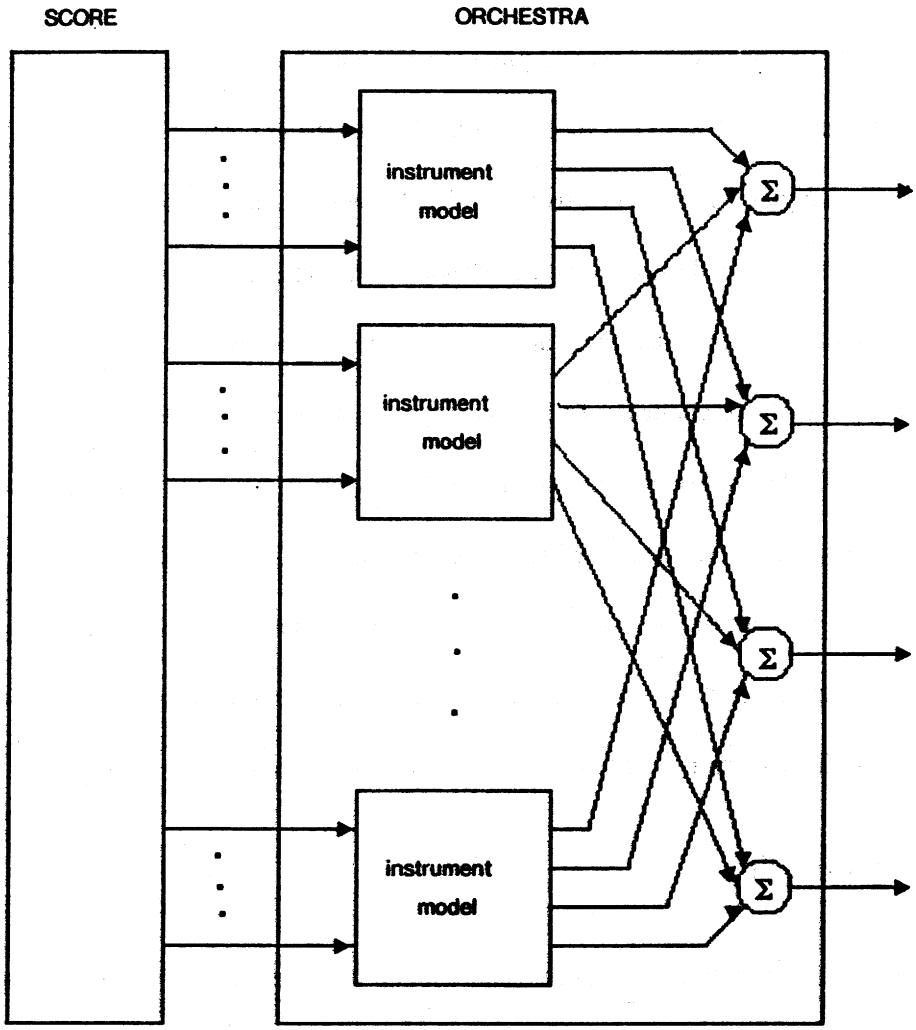
3.1 Orchestra File

The orchestra file consists of a header and one or more instrument blocks. The header has the following format:

sr = ⟨int⟩
kr = ⟨int⟩
ksmps = ⟨int⟩
nchnls=⟨int⟩

where ⟨int⟩ stands for any integer value.

The variable *sr* is the sampling rate. This means that the DAC in Fig. 1.3 will receive a number of samples every second equal to *sr*. A signal that contains *sr* values for each second in time is an



INTERACTION BETWEEN MUSIC-11 FILES

Fig. 3.1

audio signal and is said to vary at the audio rate. For good quality sound *sr* should be at least 30,000. Hardware limitations of the synthesis system impose an upper bound of 50,000.

The variable *kr* is the control rate. It is a sampling rate for signals that should vary more slowly with time than audio signals. An example of one such signal is an amplitude envelope.

The variable *ksmps* is equal to *sr/kr*. Because *ksmps* must be an integer a restriction is imposed. It is that *sr* must be divisible by *kr*. Also, *sr*, *kr* and *ksmps* must remain fixed throughout a synthesis.

The variable *nchnls* specifies the number of output channels and thus the number of loudspeakers that are used.

Instrument blocks are structured as follows:

instr *n*
<body>
endin

where *n* is an integer that serves as the instrument identifier and endin signals the end of the instrument definition. The body is a linear sequence of statements. Several types of statements are available in Music-11, however only one of them, the function statement will be considered in this discussion. Other statement types will be introduced as they are needed.

Function statements are required to have the form:

<*sig*> function *inp1,inp2,inp3...*

function can be one of the many signal generator or modifier function that Music-11 offers, perhaps a filter or an oscillator. The function's input parameters are given by *inp1, inp2 and inp3*. The number of input parameters will vary from function to function. The output of function is <*sig*> and with the exception of one function in Music-11 is either an audio or control signal. If *sig* is an audio signal then it is required to have the variable name *an* where *n* is any positive integer. If *sig* is a control signal, its variable name must be *kn*.

Function statements alone can be used to express simple instrument block diagrams such as the

one in Fig. 1.2. Suppose it is desired that the output of the envelope generators in Fig. 1.2 be linear. The line function in Music-11,

kn **line** *ia, idur, ib*

will create such a signal. Its first and third inputs define the endpoints of a linear segment. The second input is the duration in seconds of the envelope.

A sinusoidal oscillator is also needed for the instrument in Fig. 1.2 For this purpose, the function **oscil** is provided:

an **oscil** *xamp, xcps, ifn, iphs*

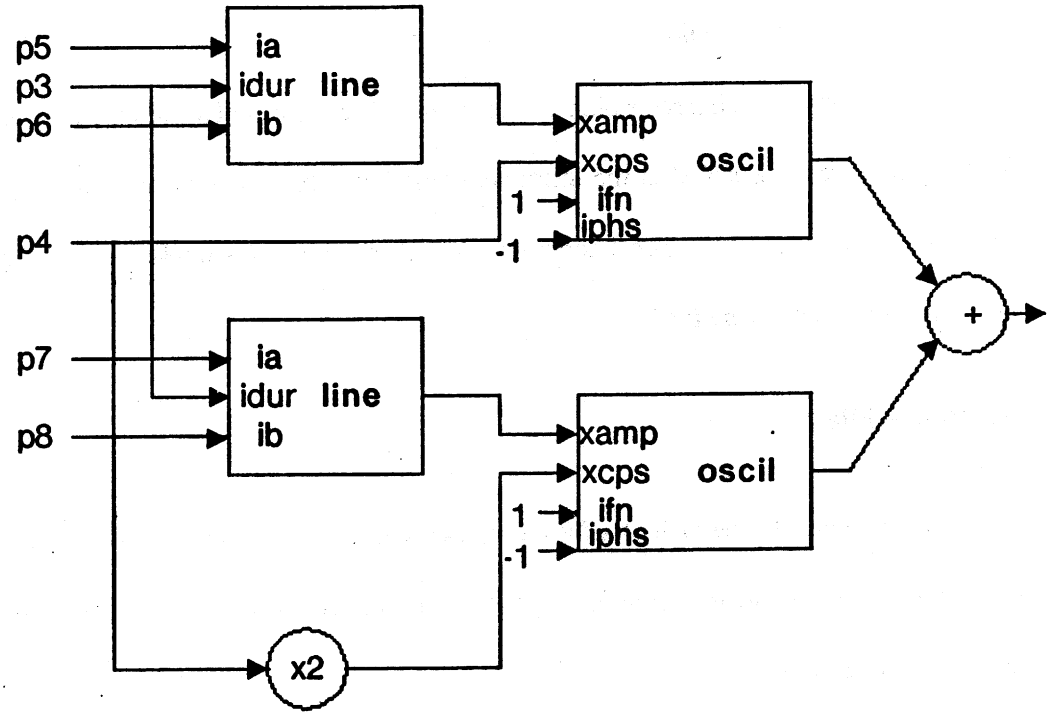
It is used as an all-purpose oscillator in Music-11 and outputs samples of a periodic wave. The third input parameter, *ifn*, is an integer identifier of an array that contains one cycle of a stored function. These arrays are called function tables. The output of **oscil** is obtained by cycling through the function table at a frequency equal to *xcps*. Each function table access is multiplied by *xamp*. Thus *xamp* determines the amplitude. The input *iphs* is explained in chapter three.

One more function is needed, the **out** function. The **out** function along with **outs** and **outq** specify to which speaker(s) the signals are to be sent. They have the following format:

out *asig*
outs *asig1,asig2*
outq *asig1,asig2,asig3,asig4*

The function **out** is used for monophonic sound. It stipulates the signal *asig* is the final output signal for the performance, the one that should be heard and it is to be sent to one speaker. In **outs**, *asig1* and *asig2* are the final output signals and each is to be sent to a different speaker thus producing stereophonic sound. The user chooses **outq** when quadraphonic sound is desired.

With these three functions in hand it is possible to construct a Music -11 diagram for the instrument in Fig. 1.2. The Music-11 diagram is shown in Fig. 3.2. Its equivalent in Music-11 code would be:



MUSIC-11 INSTRUMENT DIAGRAM

Fig. 3.2

```
instr 1
;               ia      idur    ib
k1      line    p5,     p3,     p6
;               xamp    xcps    ifn     iphs
a1      oscil   k1,     p4,     1,      -1
;               ia      idur    ib
k2      line    p7,     p3,     p8
;               xamp    xcps    ifn     iphs
a2      oscil   k2,     2*p4,   1,      -1
out     a1+a2

endin
```

Lines beginning with a semicolon are comments. The inputs to line and the frequency input to oscil are not directly specified in the instrument description because they are externally controlled. The score file provides the external control, so, *p3-p7* are parameters whose values are given in the note statements in the score file.

3.2 Score File

The score file like the orchestra file has its own statement types and is only a sequence of these statements. The final line in a score file must always be a single e, signifying the end of the file. Thus the general form is:

```
<statement>
<statement>
<statement>
.
.
.
e
```

where <statement> is one of several score statement types. The statement types that are needed to be able to write a score file to control the instrument defined in the last section are the note, stored function and tempo statements.

Note statements make up the majority of the score file and hold most of the control over the instrument. These statements follow the format:

i *p1 p2 p3 p4 p5...p128*

where

p1 - identifier of instrument for which this statement is intended
p2 - starting time in beats for this note
p3 - duration of this note in beats
p4,p5...p128 - parameters used to control the instrument

Though 128 parameters can be specified, only as many as the composer needs to control his instrument have to be included in the note statement. This is true for all score statement types.

The stored function statement is used to create an array that holds equally spaced samples of one of several functions provided by Music-11. Its format is:

f *p1 p2 p3 p4 p5 p6...p128*

where

p1 - integer identifier of the function table
p2 - creation time in beats of this function table
p3 - the function table size (restricted to be 2^n or 2^n-1 in Music-11)
p4 - integer identifier of the function generator to be used
p5,p6...p128 - input parameters of the function generator

An example of a function generator is one that computes one cycle of a sum of sinusoids. Its integer identifier is 10. The input parameters inp5, inp6... determine the relative strength of the harmonics in the sine wave. A user may specify up to 123 harmonics to be present in the wave. If *p4* is positive then the wave will be scaled to a peak amplitude of 1. If *p4* is negative, no scaling will take place. For example,

f 1 100 256 -10 4 2 0 8

causes 256 samples of one cycle of the wave

$$4\sin\omega + 2\sin 2\omega + 8\sin 4\omega$$

to be stored in an array and its integer identifier to be 1, whereas if the statement

f 1 100 256 10 4 2 0 8

is included in the score file the same array as before is constructed and then scaled so that the maximum value in the array is equal to one. For both of these examples the function table will be generated immediately before the tenth beat of the piece is performed.

The stored function and note statements express time in the unit of beats. Musically, this is a natural way to think of time progression in a composition. However the orchestra file requires that the unit of time for any of its inputs be seconds. The tempo statement in the score remedies this problem by giving guidelines on the beat to second conversion. The tempo statement takes the form:

t *p1 p2 p3 p4 p5...p128*

where

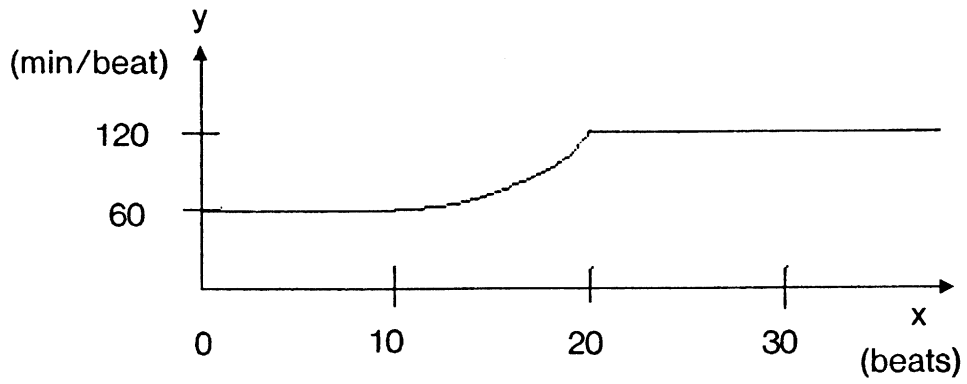
p1 - starting beat (restricted to equal zero)
p2 - initial tempo in beats/min
p3,p5,p7...p127 - starting beat of next tempo
p4,p6,p8...p128 - tempos corresponding to starting beats

It is easy to think of the input parameters as defining points on a tempo vs. beat graph. For example the statement,

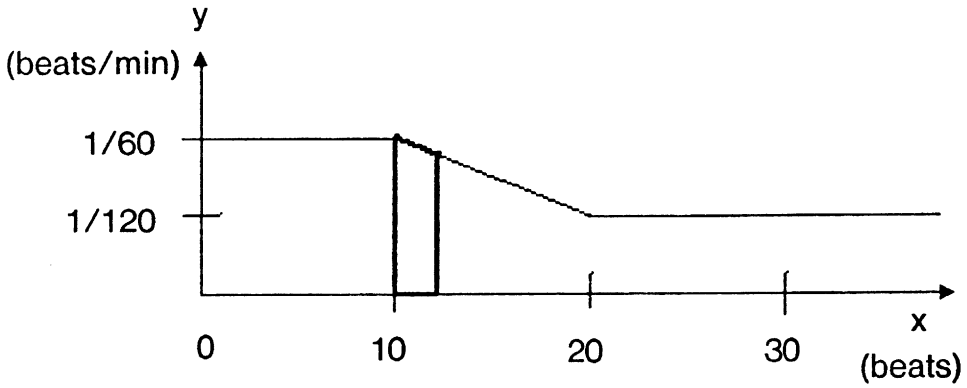
t 0 60 10 60 20 120

produces the graph in Fig. 3.3(a). The first ten beats are played at 60 beats/min. Between beats 10 and 20 an accelerando from 60 to 120 beats/min occurs. From beat 20 on, the tempo stay constant at 120 beats/min. The last defined tempo *pn* in a tempo statement always remains the tempo starting at beat *pn-1* throughout the remainder of the piece.

The curve between beats 10 and 20 depicts how accelerandi are modelled. This curve is proportional to 1/x and is constrained by Music-11's method of beat to second conversion. Music-11 carries out its beat to second conversion by first drawing a graph whose x coordinate is beats and whose y coordinate is 1/tempo = min/beats. Such a graph for the above tempo statement example is shown in Fig. 3.3(b). For a given note whose beat duration occurs between beats x_1 and x_2, the



(a)



(b)

TEMPO STATEMENT GRAPHS

Fig. 3.3

area under the graph is computed between these two points. Since the area is in units of minutes, it is divided by 60 to obtain the desired unit of seconds. For example, the duration in seconds of a note that starts at beat 10 and is 2 beats long would be equal to the outlined area in Fig. 3.3(b) divided by 60.

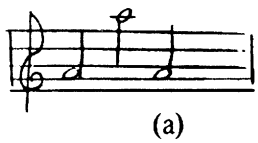
Now all the necessary tools have been presented to assemble a score file to play the instrument in Fig. 3.2. A translation of the admittedly boring score in Fig. 3.4(a) is shown in Fig. 3.4(b). Lines beginning with c are comment lines.

3.3 Additional Orchestra File Information

Now that the reader is familiar with the basics of the Music-11 language, some additional information about the orchestra file is introduced. It was not presented in the previous two sections to avoid confusing the reader.

3.3.1 Rates of the signal function input parameters

As mentioned before, the output of a signal function can be either control or audio rate. The input parameters of a signal function can also be different rates, but certain input parameters are restricted to be different rates. For instance, the oscil input parameter *ifn* is allowed to vary no faster than once each note (at the note rate). It can also be a constant, the slowest rate possible. In general an input parameter prefixed with an *i* (ie. *ifn, iphs* of oscil and *ia, ib, idur* of line) must be ote rate or a constant. An input parameter prefixed with a *k* must vary at the control rate or a slower rate. An input parameter whose first letter is *x* (ie. *xamp, xcps* of oscil) is restricted to vary at the audio rate or a slower rate. Finally, if an input parameter is prefixed with an *a* (ie. *asig* of out) it must vary at the audio rate only.



(a)

SCORE

c	p1	p2	p3	p4	p5	p6	p7	p8
t	0	60						
f	1	0	256	10	1			
c	instr	start	dur	xcps	ia	ib	ia	ib
i	1	0	2	440	10000	5000	9000	6000
i	1	2	2	880	8000	6000	4000	2000
i	1	4	2	440	10000	5000	9000	6000
e								

(b)

SCORE FILE

Fig. 3.4

3.3.2 Goto statements

Goto statements belong to an orchestra statement type that was not discussed in the section in the orchestra file. Nevertheless, these statements are extremely useful because they allow conditional branching. A goto statement may take one of the following two forms:

goto *name*

or

if *kn* <test> *km* **goto** *name*

The first goto statement causes an unconditional transfer of control to the statement whose label is *name*. The second goto statement, <test> is one of the following: =, ~=, >, <, >=, <=. If *kn* <test> *km* is true than a transfer of control to the statement labelled by *name* is made. Otherwise, the control is advanced to the next statement.

Fig. 3.5 shows an example of an orchestra file that uses goto statements. The instrument block models an instrument whose amplitude envelope depends on whether note's frequency is above or below 1000 cps. The function **expon** outputs a line segment with endpoints *ia* and *ib* like **line** except the line segment is exponential instead of linear.

3.3.3 Global variables.

All the variables in the orchestra file examples thus far have been local to the instrument block that contained them. However Music-11 allows global variables in the orchestra file. The Music-11 compiler recognizes a variable as global when it is prefixed by a *g*. An example of an orchestra file that uses a global variable is shown in Fig. 3.6. The signal function **reverb** reverberates its first input by a number of seconds equal to its second input. The global variable **gal** is used to accumulate the outputs of instruments 1 and 2. The sole purpose of instrument 3 is to reverberate the accumulated signal.

```
sr = 50000
kr = 1000
ksmps = 100
nchnls = 1

instr 1

      if p4>1000 then goto below
;                      ia     idur    ib
      k1    line     1000,  p3,     500
      goto finish
;                           ia     idur    ib
below: k1     expon    700,   p3,     400
;
;                     xamp   xcps   ifn    iphs
finish: a1    oscil    k1,    p4,    1,     -1
        out    a1
endin
```

EXAMPLE OF AN ORCHESTRA FILE CONTAINING GOTO STATEMENTS

Fig. 3.5

```
sr = 50000
kr = 500
ksmps = 100
nchnls = 2

instr 1
  .
  .
  .
;            xamp  xcps  ifn   iphs
  a1   oscil   k2,   k3,   1,   -1
  ga1 = ga1 + a1
endin


instr 2
  .
  .
  .
;            xamp  xcps  ifn   iphs
  a1   oscil   k2,   k3,   1,   -1
  ga1 = ga1 + a1
endin


instr 3
  ga1 init 0
;              asig    idlt
  a1   reverb   ga1,    2.49
  ga1 = 0
  outs  a1/2,   a1/2
endin
```

EXAMPLE OF AN ORCHESTRA FILE CONTAINING A GLOBAL VARIABLE

Fig. 3.6

3.4 Performance

When the user is satisfied with his files and wishes to hear them performed, he goes through a two step process. He must first run the program perf which outputs a sound file using the orchestra and score files as inputs. Then he must invoke the program convert with the sound file as input.

The program perf first processes the score by sorting it according to *p2* and converting the beat values of *p2* and *p3* to seconds in agreement with the directions of the tempo statement. Then the orchestra and score files are compiled so they are acceptable input to the Music-11 monitor which performs the actual sample calculation. The orchestra file is compiled into a list of function calls corresponding to each signal function. Each signal function has its own private data space in which to store the results. If the output of one function is the input of another, then the appropriate links are made between the two data spaces.

The Music-11 monitor is an event-driven program that uses the score file as its event list. The monitor's clock is initialized to zero and advances at time increments equal to one control period (1/kr). At the beginning of each control period, the monitor checks the score statement at the top of the event list to see if its clock is greater than or equal to *p2*. If so the monitor will act in accordance with the statement type. If the statement is a function statement, the monitor will create a function table by allocating an amount of space in memory equal to *p3* and fill it with samples of the appropriate function. If the statement is a note statement then the note parameters *pn, n > 3*, are read into the data space of the instrument specified by p1. After the monitor is done scanning the event list for current events, it carries out the specified function calls in the compiled orchestra file. For instance, for the instrument description in section 3.1, the monitor would call line and store the result, call oscil and store the result, call line and store the result, call oscil and store the result. Finally it would add the outputs of the two oscil functions and write the result to a soundfile. The **out** functions tell the monitor which signals are to be saved on the soundfile. Thus the monitor is outputs

calculated samples of the users's composition.

Since the signal functions are called every control period, the functions that output control rate signals produce one real value and the functions that output audio rate signals produce an array of real values of size ksmps.

The program convert uses the sound files as input. This program regulates the feeding of the buffers in Fig. 1.3 so that the samples in the sound file are fed to the DACs at the sampling rate.

4. DATA FLOW CONCEPTS

It has been shown in the previous chapter that the composition language Music-11 cannot perform in real-time on its present processor. In fact, real-time performance of a composition written in any language similar to Music-11 is unattainable on any machine whose design adheres to the Von Neumann style of architecture. To realize real-time performance, a different architectural style must be used. One such architecture is data flow.

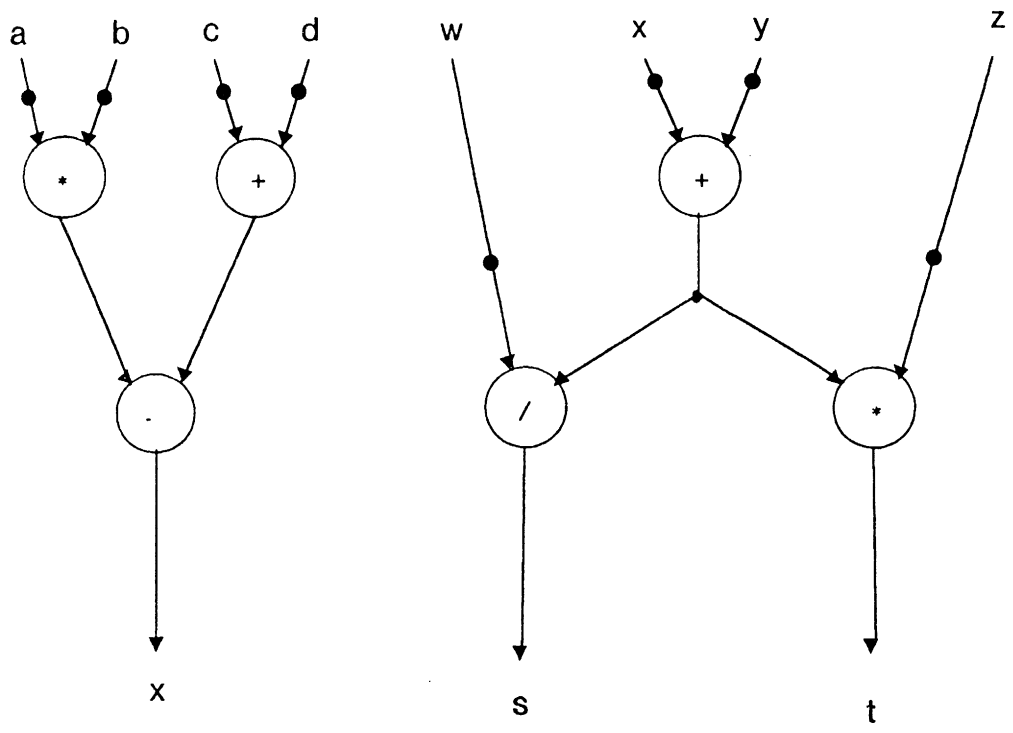
Research in data flow computer architecture was initiated to develop an alternative to the Von Neumann architecture. Whereas the conventional Von Neumann machine must execute each operation in a program in sequence, a data flow machine allows many operations to be done in parallel increasing the computation rate and thus potentially decreasing the program execution time.

Data flow machines are capable of high performance due to a unique architecture. An attempt is made in this chapter to give the reader a better than surface level understanding of the machine architecture that is used in this paper, present some principles of data flow and introduce VAL, a high level language written at MIT expressly for use with data flow machines.

4.1 Program Graphs

Data flow program graphs provide a useful tool in the analysis of parallelism within an algorithm. When an algorithm is expressed in a program graph, concurrency appears naturally. A brief discussion of program graphs and how they are interpreted follows.

Figure 4.1(a) shows an example of a simple program graph. The inputs are *a, b, c* and *d*, the output is *x*. The mathematical operators enclosed by circles are called actors, the arrows are called arcs and the black circles are referred to as tokens. Tokens represent either elementary or structure values. Elementary values may be real, integer or boolean values. Structure values are compound data types such as arrays or records.



x = (a*b)-(c + d)

s = (x + y)/w

t = (x + y)*z

(a)

(b)

DATA FLOW GRAPHS

Fig. 4.1

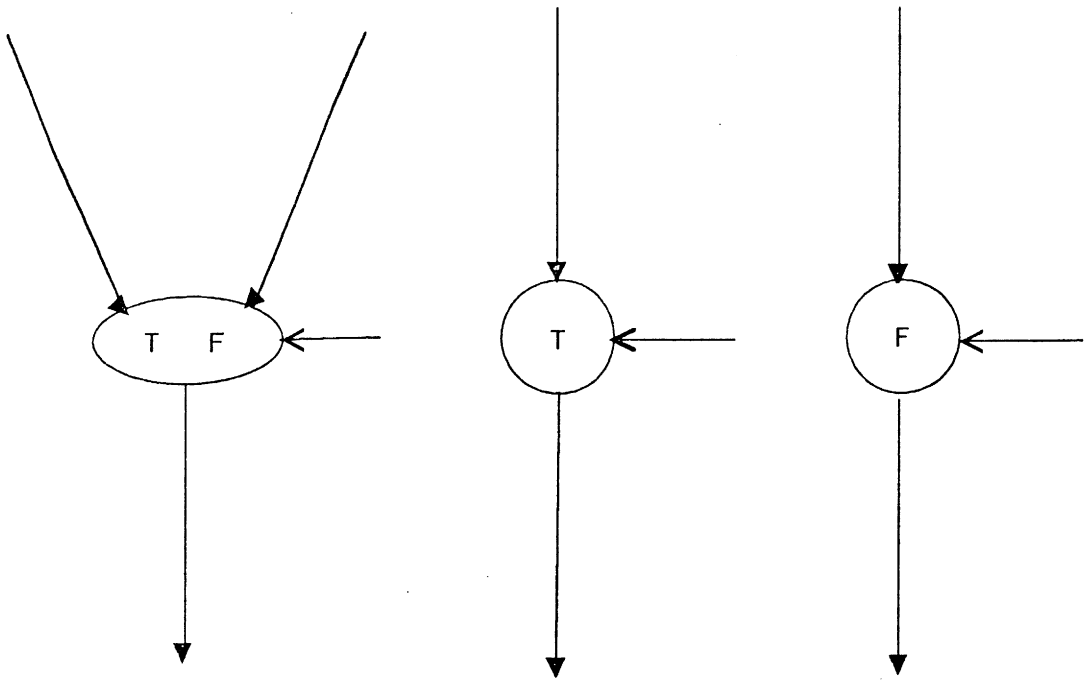
An actor may fire (perform its operation) only when all of is input arcs carry a token and its output arc is empty. Upon firing the actor consumes the input tokens and places a token containing the computed value on its output arc. The output arc may input more than one actor in which case a link provides the necessary number of tokens. For example, the link in Fig. 4.1(b) ensures that the quantity $(x+y)$ is delivered to both the divide and multiply actors.

Every actor abides by these firing rules except for three types of actors. They are the merge actor, the T-gate and the F-gate (Fig. 4.2(a,b,c)).

The merge actor has three inputs. The inputs labelled T and F may be any value. The remaining value must be a boolean value. The open arrowhead shows that a boolean value is expected as opposed to solid arrowheads which denote non-boolean values. If the boolean input contains a true token and the T input contains a token then the merge actor will fire and produce a token containing the T input value on its output arc while absorbing the boolean and T input tokens. No token was needed on the F input for this to occur. If an F token did exist, it would stay there until a false token arrived on the boolean input. In this case the actor would fire by consuming the boolean and F input tokens and placing a token carrying the F input value on its output arc.

T-gates require two inputs, a boolean input and a non-boolean input. The actor will not fire until both inputs are present. When the T-gate does fire, it duplicates the non-boolean input. If the boolean input is false the gate merely consumes the tokens and produces no output. The F-gate behaves in a complimentary fashion.

Three other operations that may not be familiar to the reader are select, append and create. They all operate on structures. Select(*struc,sel*) uses the *sel* input to locate a value within the *struc* input. The *sel* input might be an index and *struc* input an array The output can be either an elementary or structure value. Append(struc,newval,sel) returns a structure exactly like the input struc except with newval at the place in the structure specified by sel. The operation create takes no inputs and returns an empty structure. Both append and select follow the firing rules.



merge actor (a)

T-gate (b)

F-gate (c)

Fig. 4.2

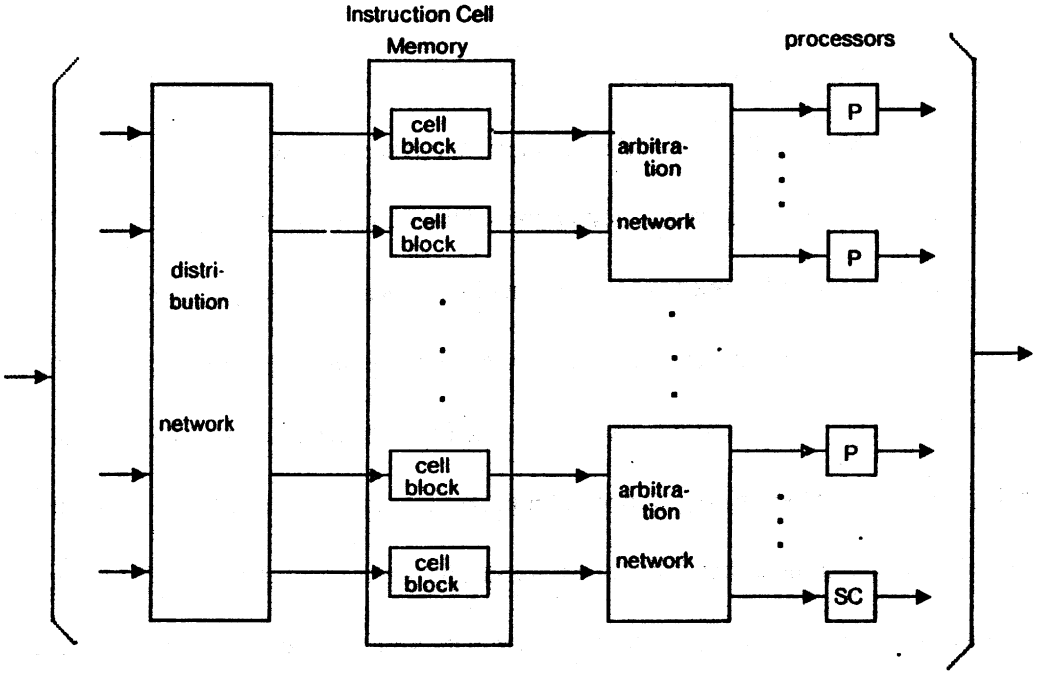
4.2 Machine Architecture

Figure 4.3 shows the architecture of the data flow machine chosen in this paper. Its design is similar to those in [12,9].

The instruction memory holds instruction blocks. Instruction blocks are made up of instruction cells. An instruction cell is a block of memory and has a unique address. It is comprised of an opcode, and space for operands and destinations. Destinations are addresses of instruction cells. The amount of space allotted to operands and destinations depends on the specific machine design.

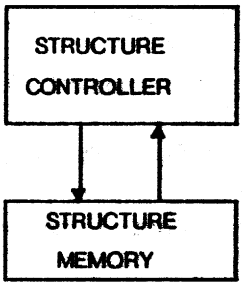
When an instruction cell contains all the operands necessary to carry out the instruction designated by the opcode and has received a specified number of acknowledge signals, it becomes enabled and fires. Upon firing, an operation packet is constructed and sent through the arbitration network. An operation packet contains the opcode, operands and destinations. The operation packet is routed to the correct processor through examination of the opcode. If the opcode specifies an operation that accepts structures as operands then the packet is sent to the structure processor (the box containing SP in Fig. 4.3). Otherwise it goes to a scalar processor (the boxes containing P). The processor performs the operation denoted by the opcode and generates result and/or acknowledge packets for the destinations designated in the instruction cell. Result packets consist of the result of the operation and address of the instruction cell for which it is bound. Acknowledge packets contain an acknowledge signal and an instruction cell address. These packets are delivered to the instruction memory through the distribution network.

The structure of the arbitration and distribution networks are discussed in [12] and will not be detailed in this paper. The design for the structure controller used in this paper is presented in [8]. It contains a structure controller and structure memory as shown in Fig. 4.4. The structure controller interprets and carries out the operation specified in the operation packets by accessing the structure memory where the structures are stored.



DATA FLOW MACHINE

Fig. 4.3



STRUCTURE PROCESSOR

Fig. 4.4

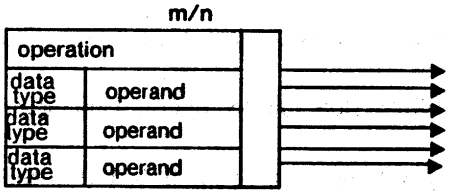
4.2.1 Instruction cell

A closer look at the instruction cell is called for. The instruction cell design proposed by [19] has been adopted in this paper. Though an instruction cell is actually a block (32 bytes) of memory, it is more convenient to represent an instruction cell as in Fig. 4.5(a). Operation denotes the opcode field in the instruction cell and the three slots marked operand characterize the operand fields. The sections labelled data type specify whether the operand is I (integer), R (real), B (boolean), S (structure) or N (not used). A *c* that resides along side an operand in a slot designates that that operand is a constant. Constants may never be changed during the course of program execution. An operand slot designated to hold a constant may initially be empty at the start of program execution but once it is filled, it can never be written on.

The letters *m* and *n* are integer values. *m* is the instantaneous number of acknowledges the instruction cell needs to fire during execution and *n* is the acknowledge reset value. After an instruction cell has fired, *m* is reset to *n*. Each time an acknowledge signal is received, *m* is decremented. When *m* is equal to zero, the instruction cell fires if all operands are present.

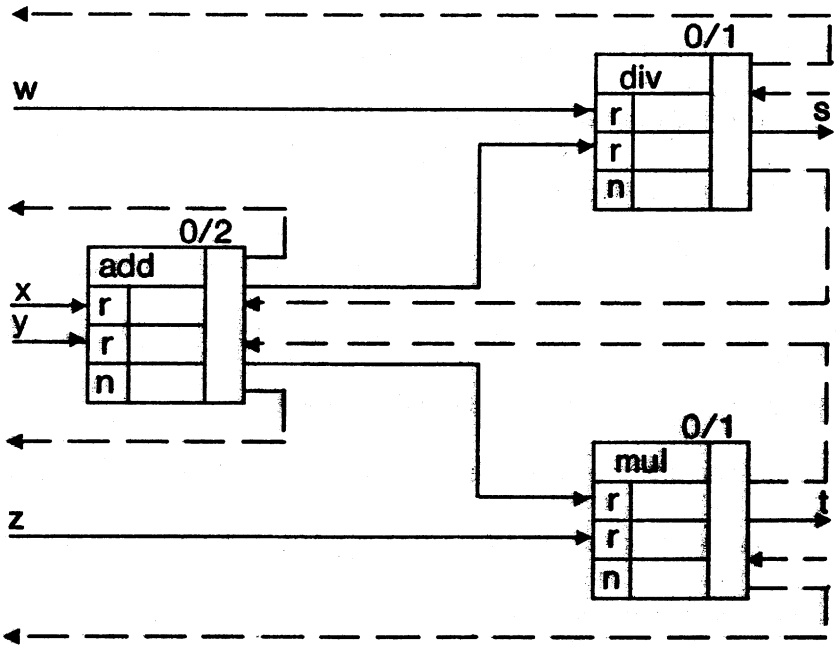
An instruction cell can send packets to up to six destinations (represented by the arrows). The six packets can be any combination of result and acknowledge packets.

Figure 4.5(b) shows how the program graph of Fig. 4.1(b) would be translated into instruction cells. For each cell in this figure *m* is the initial number of acknowledge signals needed. The solid lines represent result packets while the dotted lines signify acknowledge packets. The add instruction cell cannot fire until it receives acknowledge signals from both the div and mul instructions cells. This insures that the add will not write over an operand that has not been used. The add instruction cell sends acknowledge signals to the cells that supplied it with its operands. The fact that $n=1$ for the div and mul instruction cells implies that their results are each sent to one instruction cell. As a general rule of thumb, an instruction cell should acknowledge any cell from which it receives an



INSTRUCTION CELL PROTOTYPE

(a)



CELL GRAPH TRANSLATION OF $s = (x + y)/w, t = (x + y)*z$

(b)

Fig. 4.5

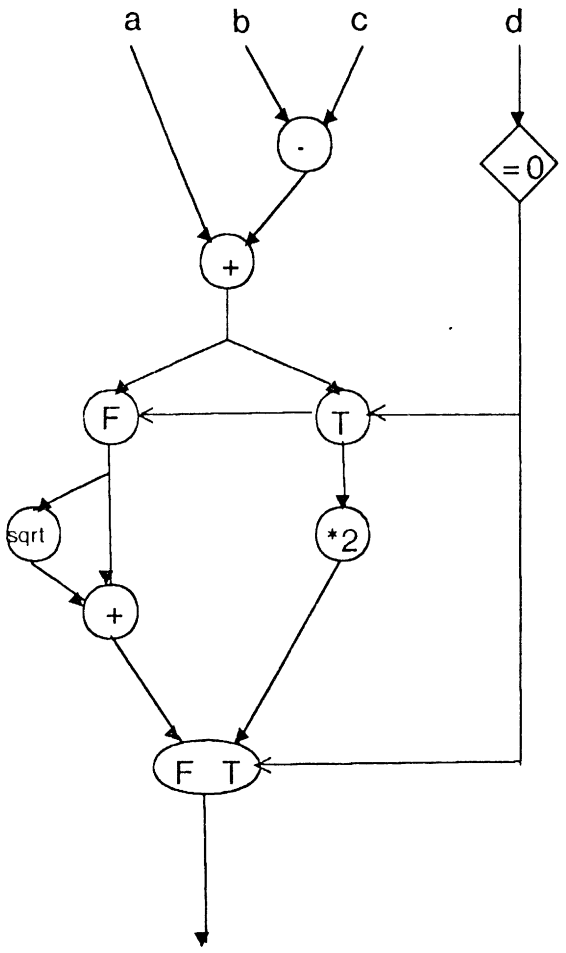
operand. However, in some cases, as we shall see later, some acknowledge signals can be eliminated.

Every operation that takes two operands on the flow graph level (ie. add) has the built in capability of a T or F gate on the instruction cell level through the use of the third operand slot. The third operand becomes a boolean input. Result and acknowledge packets can be conditionally sent based on the value of the third operand. For example, when the program graph in Fig. 4.6(a) is translated into instruction cells (Fig. 4.6(b)), the add operator and the two gates can be expressed as one instruction cell. If the third operand is not used the result and acknowledge packets are sent to their destinations unconditionally. Whenever the third operand is used to conditionally send a result, two acknowledge reset values are needed. For the add instruction cell, the true acknowledge reset value is *1* and the false acknowledge reset value is *2*. If the third operand is *true* when the add cell fires, the number of acknowledges needed is set to *1*. Similarly, if the third operand is *false* when the cell fires, the number of acknowledges needed becomes *2*.

4.3 VAL

Data flow program graphs are a valid tool to use in the expression of data flow algorithms, but writing a large program in such a graph language can be troublesome task. For this reason, the language VAL was written. The motivation behind the development of VAL was the need for a high level language in which one can write algorithms to run on a data flow machine. One of the main goals of VAL is to make concurrency easily identifiable. This is achieved by the exclusion of side effects.

It is assumed that the reader is familiar with some block-structured language. Therefore only two VAL constructs will be explained that may be foreign to the reader. They are the **let** and **for** constructs.

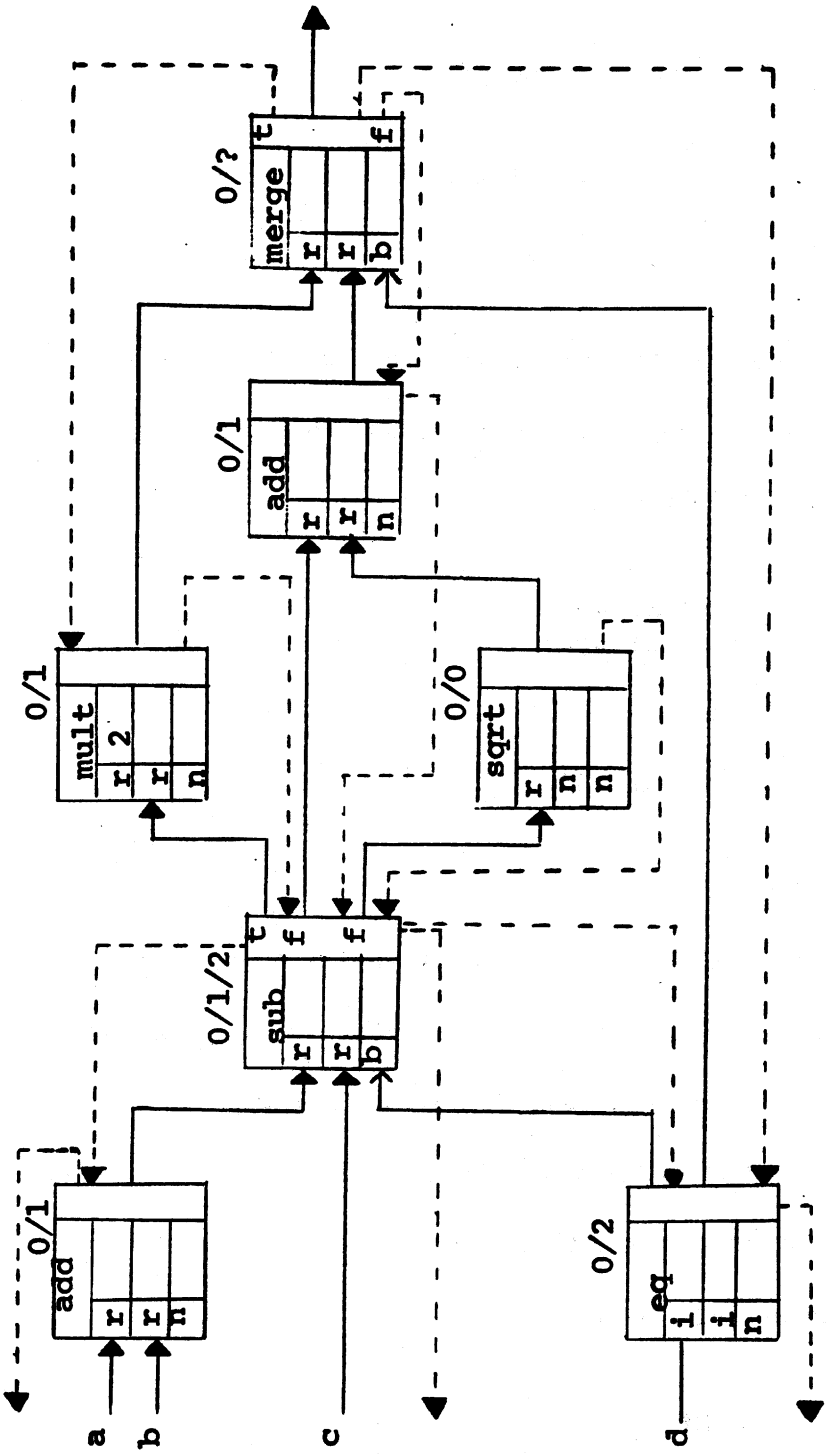


DATA FLOW GRAPH TRANSLATION OF

if d = 0 then (a + b-c) *2 else sqrt(a + b-c) + (a + b-c)

(a)

Fig. 4.6



CELL GRAPH TRANSLATION OF if d=0 then (a+b-c)*2
else sqrt(a+b-c)+(a+b-c)

(b)

Fig. 4.6

4.3.1 LET Construct

The let statement takes on the following form:

```
let
        <declaration list>
in
        <expression>
endlet
```

A declaration list is simply a list of variable declarations. In a let block, a variable may be assigned a value only in the declaration list. Once a variable has been assigned a value, no reassignment may take place. Thus, the declaration list,

```
y : real := 3.1;
x : real := y/.5;
```

is legal VAL code whereas,

```
y : real := 3.1;
y := y/.5;
```

is not acceptable VAL code.

An expression is one or more simple expressions separated by commas. A simple expression for our purposes is loosely defined as VAL code that can be evaluated down to a value. This value may be any of the accepted data types in VAL. The arity of an expression is equal to the number of simple expressions it contains. For instance *4.3* and *true* are values and therefore expressions of arity one. An expression of arity two could be *5.2/8, sin(.32).*

An example of a *let* statement is,

```
let
        x : real := 2.;
        y : real := 3.1;
in
        x*y
endlet
```

This statement is equal to 6.2.

4.3.2 FOR construct

For the present discussion, only a subset of possible **for** statements will be considered. This subset includes **for** statements that contain iteration forms. The **for** statement is presented by means of an example.

Figure 4.7 shows a **for** statement that calculates x^n. The declaration list, (lines 2-4) has the same single assignment restriction as in the **let** construct. However, variables declared in the **for** statement may be

```
for
        factor : real := x;
        powercount : integer := n;
        returnval : real ; = 1;
do
        if powercount >0
        then iter
        returnval := returnval*factor;
        powercount: = powercount-1;
        enditer
        else returnval
        endif
endfor
```

Fig. 4.7

reassigned within the **iter** statement. Only variables in the declaration list between the **for** and **do** may be reassigned.

4.4 Translation of VAL Code to Flow Graphs

Three examples of translating VAL code to flow graphs are given in this section. The first two examples are of conditional statements. The last example deals with an iteration construct.

4.4.1 Conditionals

Fig 4.8 shows the data flow graph for the VAL conditional statement,

```
if test_expression
        then true_expression
        else false_expression
endif
```

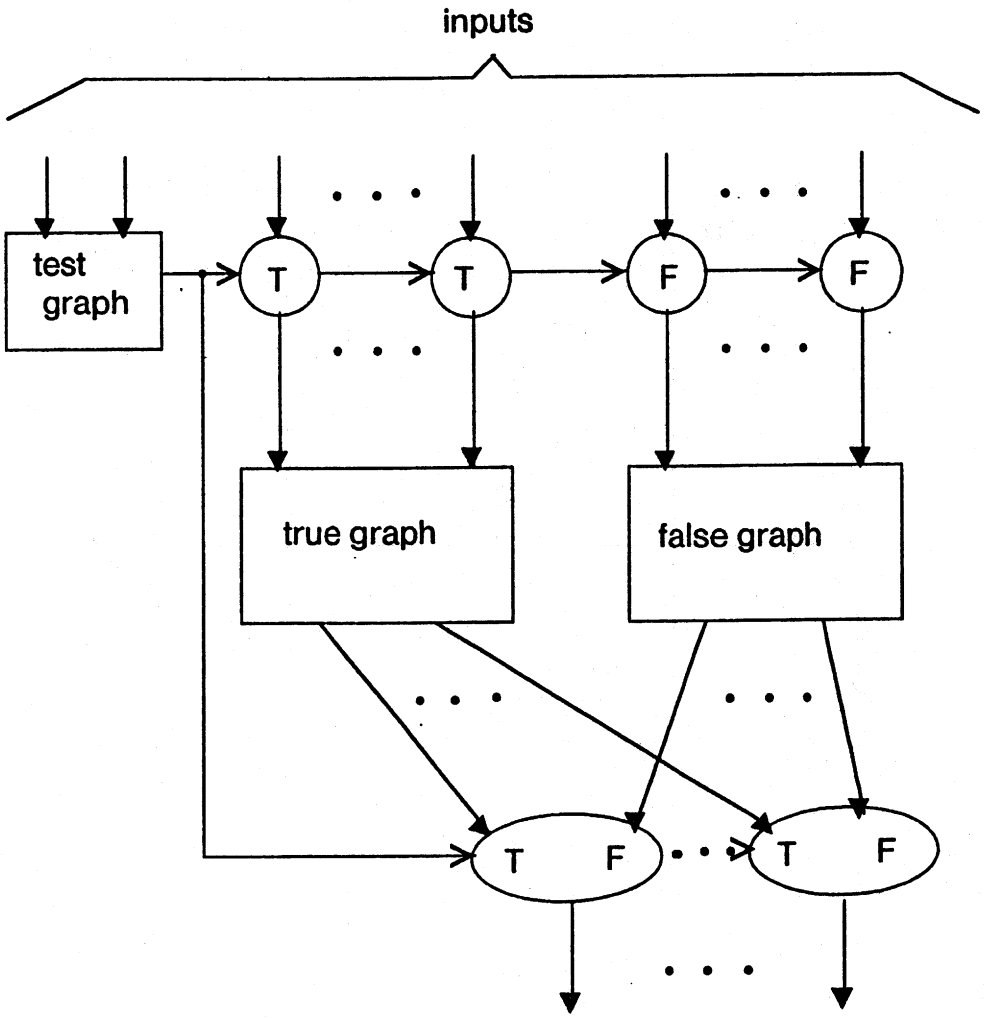
The boxes themselves contain data flow graphs. The box labelled test graph evaluates test_expression of the conditional statement and yields a boolean value of either true or false. If it is true, the inputs to the t-gates are allowed to feed into true graph and true_expression is evaluated. Otherwise the inputs to the F gates are allowed to flow through false graph in order to evaluate false_expression. The number of outputs of true graph and false graph equal the arity of the conditional expression and must equal each other. The merge gates eliminate race conditions that may occur between the true and false branches of the graph. They also insure that the nth set of outputs correspond to the nth set of inputs.

As an example of a conditional statement, the reader is referred to Fig. 4.6(a) which is a flow graph translation of

```
if  d=0
        then  (a+b-c)*2
        else  sqrt(a+b-c)+(a+b-c)
endif
```

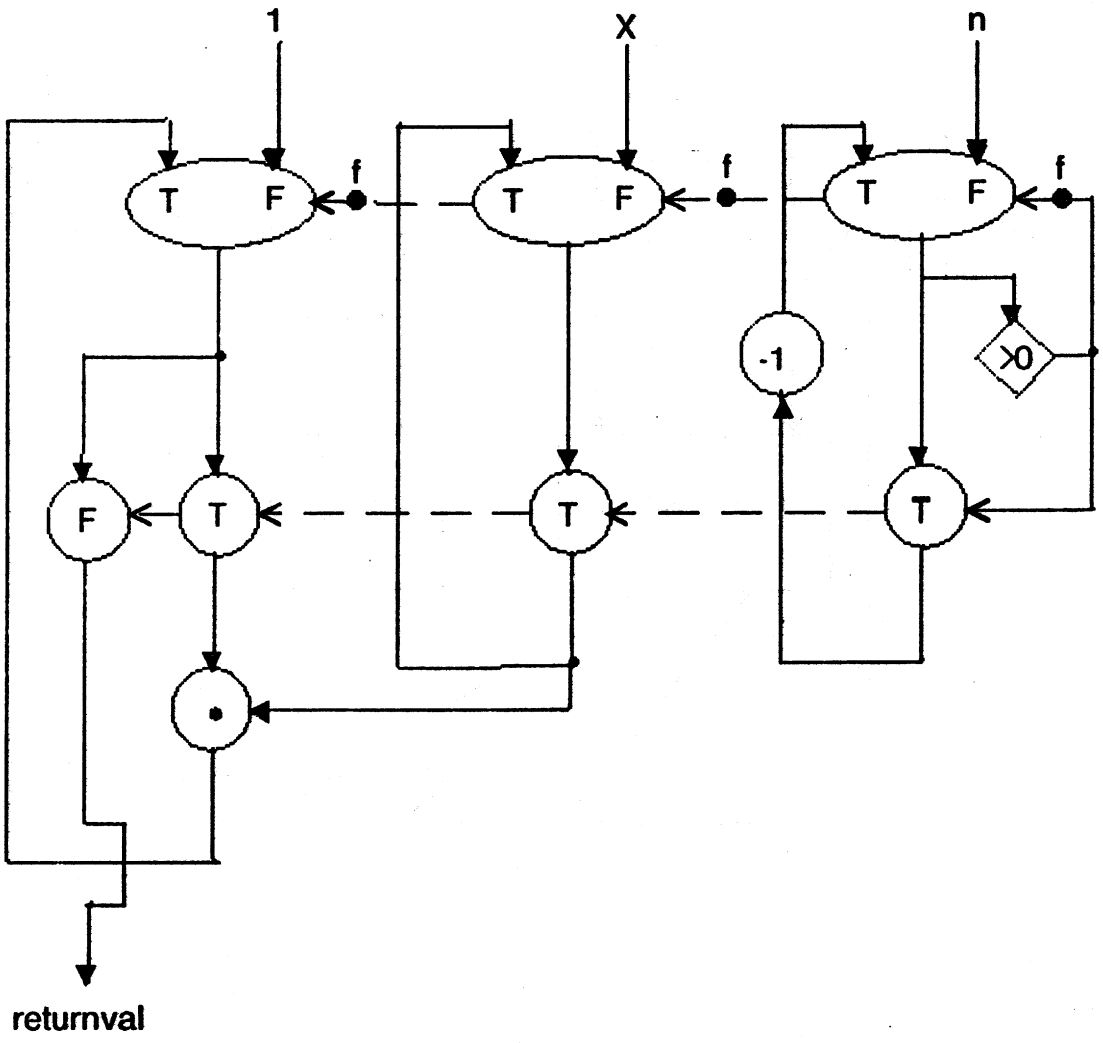
4.4.2 Iteration

As an example of translating iteration constructs, Fig. 4.9 depicts the flow graph translation of the VAL code in Fig. 4.7. There is an initial value of false on the boolean input of the merge operators, to allow the inputs to drop into the iteration loops. The ">0" operator outputs *n* consecutive true values followed by one *false* value. Each *true* value allows the tokens to cycle around their respective loops one time. When the final *false* token is produced, the T-gates swallow



DATA FLOW GRAPH TRANSLATION OF
A CONDITIONAL STATEMENT

Fig. 4.8



DATA FLOW GRAPH OF THE POWER FUNCTION

Fig. 4.9

their inputs, bringing the iteration to a halt. The F-gate allows its input to pass to its output arc, to become the output of the graph.

Note that after the iteration process has ended, *false* tokens are left on the boolean inputs of the merge operators. In other words, the graph will return to its initial state. This allows consecutive sets of tokens to flow through the graph, each set producing a correct output token. In this paper, this is a necessary feature of flow graphs, as shall be seen in chapter three.

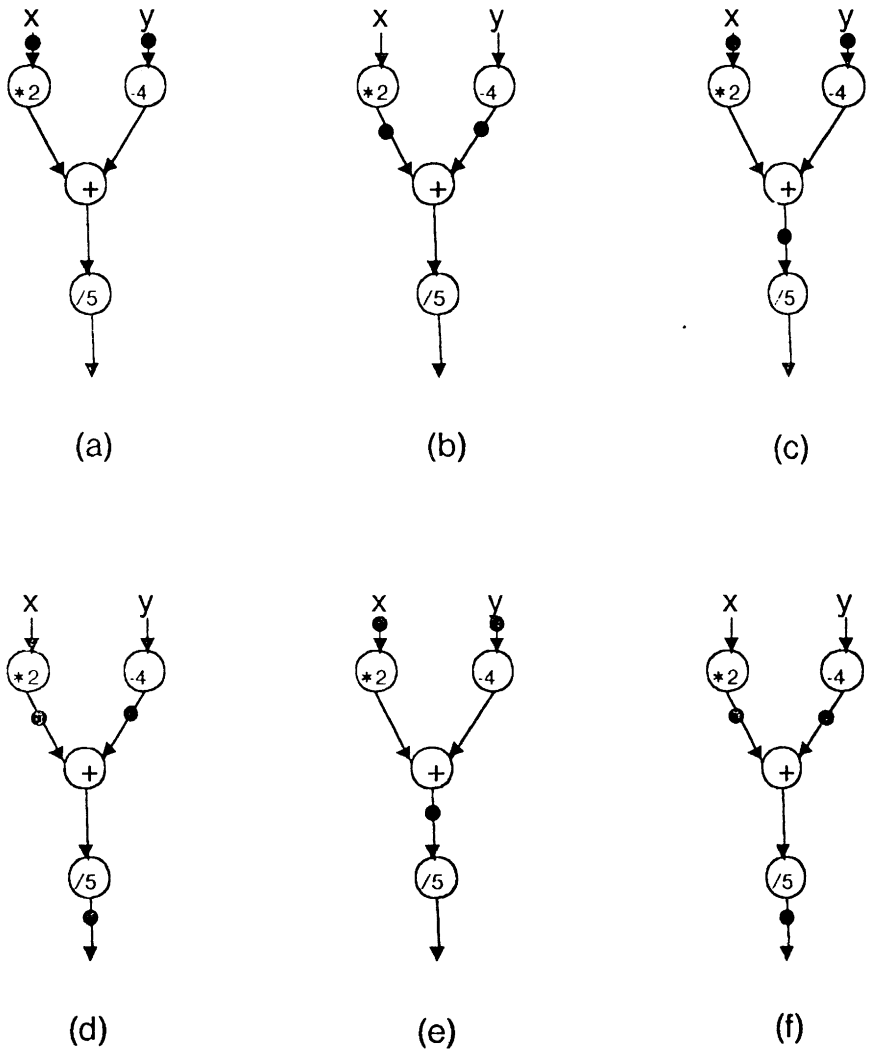
4.5 Pipelined Flow Graphs

When algorithms are expressed in data flow graphs, the operations that can be done in parallel become easily identifiable. For instance, in Fig. 4.2(b) it is clear that the add and multiply operations can fire simultaneously. When actors on the same level can perform their operations in parallel, there is horizontal concurrency. However vertical concurrency (pipelining) can also be exploited for higher performance.

Every time one or more actors fires simultaneously, a tick occurs. Thus, for Fig. 4.10(a), it takes three ticks for the inputs x and y to help produce an output on the arc labelled z. This is because the depth of the graph is 3 actors. Suppose it is desired to calculate $((2*x)+y-4)/5$ for many values of x and y. Then we can allow successive sets of x and y inputs to flow through the graph at the same time as shown in Fig. 4.10(a-f). Now an output is produced every other tick. When an output appears every second tick then the graph is maximally or optimally pipelined.

Note that new inputs cannot be put on the graph input arcs until, the *2 and -4 operators have acknowledged that they have fired. Recalling the machine architecture, acknowledge packets must traverse the same routing networks as result packets so their transit time is the same.

The graph in Fig. 4.11(a-f) is not maximally pipelined. This is because a new x input cannot be accepted until the token on the output arc of the multiplication actor has been consumed. However, a maximally pipelined graph can be obtained through the modification of the graph in Fig. 4.11. The



MAXIMALLY PIPELINED GRAPH OF ((2*x) + y-4)5

Fig. 4.10

modified graph is shown in Fig. 4.12. It contains an additional operators called identity operators. When an identity operator fires, it simply reproduces the input on its output arc. Thus the identity operator acts like a buffer in this case. The reader can verify that the resulting graph is maximally pipelined.

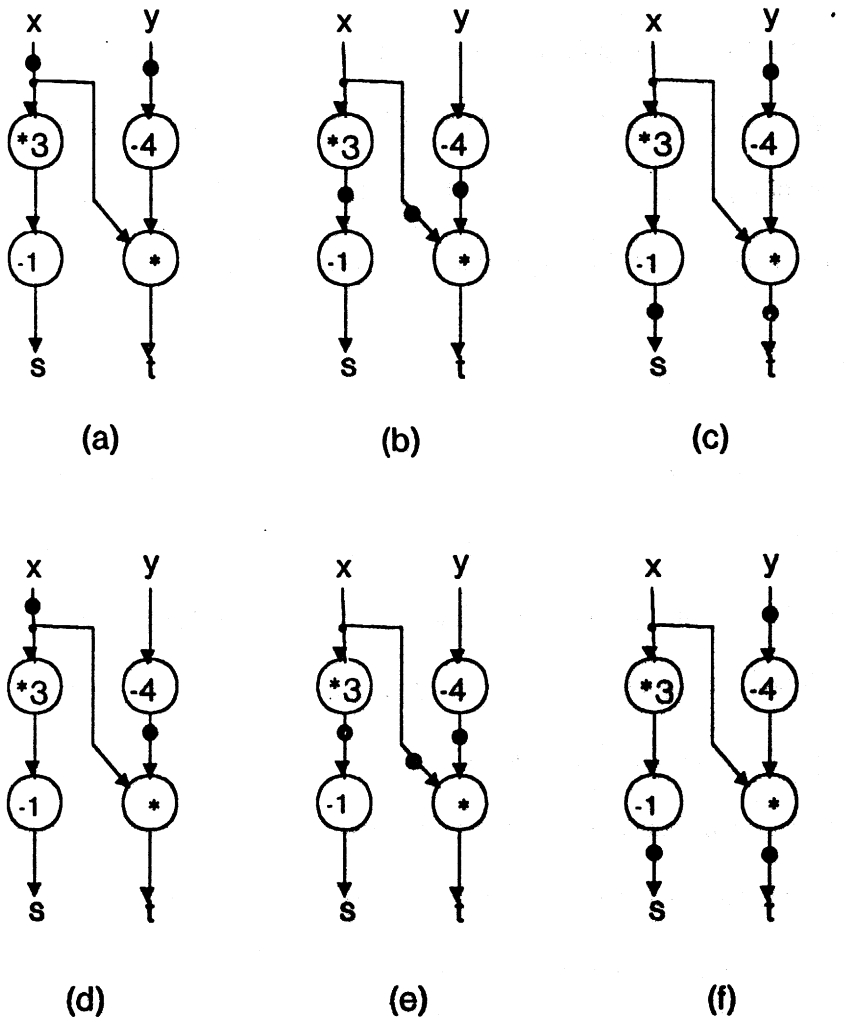
In general any data flow graph that does not contain a closed path can be maximally pipelined by adding enough identity operators so that every path in the graph holds the same number of actors. The modification of Fig. 4.11 is an example. The addition of the identity operator made the x to t path two operators long to match the length of the x to s and y to t paths.

4.6 Streams

At this point it is useful to introduce a new data type, streams. Currently, VAL does not support streams. However for the purpose of this paper, streams will be used as if they exist in VAL. The motivation behind the inclusion of stream data type is that Music-11 instrument descriptions are essentially descriptions of digital signal processing networks. It is natural to think of signals as streams of values as they flow through networks.

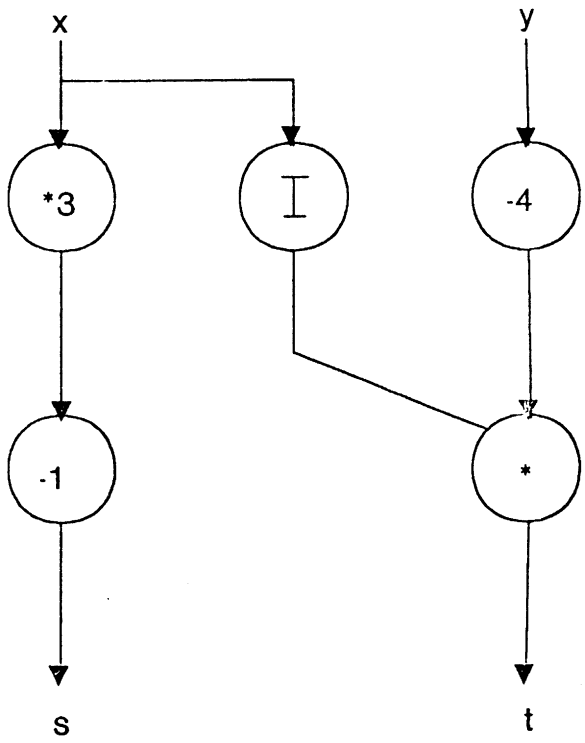
Streams are ordered sequences and can possibly be infinite. The members of a stream are restricted to be of the same data type, though there are no restraints on the data type. Hence there may be a stream of integers, a stream of arrays and even a stream of steams.

A stream will be represented by its ordered members separated by commas, enclosed by square brackets. An example of a four member stream of real is $x = [1.1, 2.2, 3.3, 4.4]$. The value *4.4* is the first member of the stream. A stream variable with a subscript of n denotes the nth member of that stream. For example $x_2 = 3.3$. Between the last member and the left bracket of every stream there is an implicit *eos* (end of stream) value. Its use will become clear in a later section.



UNPIPELINED GRAPH OF s = (x*3)-1, t = (y-4)*x

Fig. 4.11



MAXIMALLY PIPELINED GRAPH OF s = (x*3)-1, t = (y-4)*x

Fig. 4.12

4.6.1 Stream functions

There are seven stream functions available to the user. They are described below:

1) **first**(x) - returns the first member of stream x.
ex. first([1,3,5,7])=1
ex. first([[1,3][5,7]])=[1,3]

2) **rest**(x) - returns the stream x without the first member
ex. rest([1,3,5,7])=[3,5,7]

3) **consl**(y,x) - returns a stream whose last member is the value y and whose members before y are those of stream x.
ex. consl(1,[3,5,7])=[3,5,7,1]

4) **catenate**(x,z) - returns a stream whose members are the concatenation of the members of streams x and y.
ex. append([1,3,5],[2,4])=[1,3,5,2,4]

5) **stream_merge**(x,y,b) - returns a stream s whose construction is determined by the boolean stream b. If the value of b_n is *true* then s_n is chosen from stream x otherwise it is chosen from stream y. Note that the sum of the number of members in x and the number of members in y must be equal to the number of members in b.
ex. stream_merge([1,3],[2],[*true,false,true*]) = [1,2,3]

6) **stream_extend0**(x,y) - returns two streams. x and y must be of type **stream**[*type*] where *type* can be any data type except **stream**. Let x be of length n and y be of length m. If $n > m$ then x and a stream s are returned where $s_i = y_i$ for $i <= n-m$ and $s_i = 0$ for $i > n-m$. If $m > n$ then a stream s and y are returned where $s_i = x_i$ for $i <= m-n$ and $s_i = 0$ for $i > m-n$.
ex. stream_extend0([1,2,3],[5]) = [1,2,3] , [5,0,0]

7) **stream_extend1**(x,y) - similar to **stream_extend0** except that the value of *1* instead of *0* is used to extend the shorter stream.

Any function or operation that can be carried out on the data type of a stream's members can be carried out on a stream itself. When it is performed on the stream, the function or operation is applied to each member of the stream to produce an output stream. For the stream *x=[1,9,16,81]* some examples are given.

1) *sqrt(x)=[1,3,4,9]*
2) *x*2=[2,18,32,162]*
3) *x<20=[true,true,true,false]*

Both operands may be streams but when they are, they must have the same number of members. As an example, for *y=[2,4,6,8]*, *x-y=[-1,5,10,73]*.

Streams may also be used in conditional statements,

if *test(z)* **then** *x* **else** *y* **endif**

where *test* is a conditional test and *z*, *x*, *y* are streams. The value of this conditional statement is a stream *s*. The test is performed on each member of *z*. If *test*(z_n) is *true* then s_n is equal to x_n and y_n is discarded. If *test*(z_n) is *false* then s_n is equal to y_n and x_n is discarded. The three streams *x*, *y* and *z* must all be the same size. Streams *x* and *y* are required to have members of the same data type.

An important property of streams is that members of a stream may be accessed before the stream is fully constructed. This is necessary if streams are allowed to be infinite in size. An example will help to clarify. The function construct_even constructs an infinite stream of consecutive even integers whose first member is *x*. The let block in Fig. 4.13 assigns the value of construct_even(0) to *y* and produces *2*y* as its value. Because of the nature of streams the multiplication starts as soon as the first members of *y* are available and will continue as long as members of y are produced.

```
function construct_even(x : integer returns stream[integer]);

for
        outstream : stream[integer] := [ ];
        x : integer := x;
do
        if false then outstream
        else iter
        outstream := consl(outstream,x);
        x := x+2;
        enditer
        endif
endfor

endfun;
```

(a)

```
let
        y : stream[integer] := construct_even(0);
in
        2*y
endlet
```

(b)

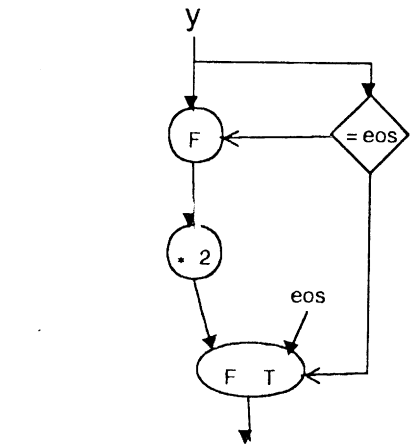
Fig. 4.13

4.6.2 Flow graph representation of streams

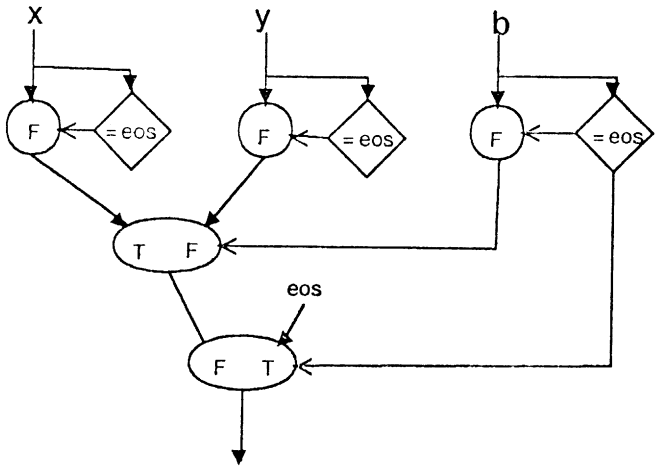
In this paper, the representation of streams in flow graphs is a sequence of tokens the last of which is always an *eos* token. So a stream multiply *y*2* would be drawn as in Fig. 4.14(a). The "=eos" condition is to test for the eos token and prevent it from becoming a input to the multiply actor. The function stream_merge is easy to translate into a flow graph (Fig. 4.14(b)). Stream conditionals are also readily translated into flow graphs (Fig. 4.14(c). It is assumed that streams *x*, *y* and *z* are all the same size.

In order to draw graphs of **stream_extend0** and **stream_extend1**, two new operations on the instruction cell level are introduced. These operations are *extend0* and *extend1*. An *extend0* cell (Fig. 4.15(a)) has the following firing rules. Let the first and second inputs to the *extend0* cell be *a* and *b* respectively. If $a \neq eos$ and $b \neq eos$ then *a* becomes the output. All acknowledge signals are sent. If $a = eos$ and $b \neq eos$ then the output is 0, the *eos* value remains in the first operand slot after firing and only acknowledge signals marked f are sent. If $a \neq eos$ and $b = eos$ then *a* becomes the output, the eos token stays in the second operand slot and only acknowledge signals marked t are sent. If $a = eos$ and $b = eos$, the output is an eos token and all acknowledges are sent. The *extend1* instruction cell behaves in a similar manner except that if $a = eos$ and $b \neq eos$ then the value 1 is the output.

The functionstream_extend0 is defined on the cell graph level in Fig. 4.16(b).

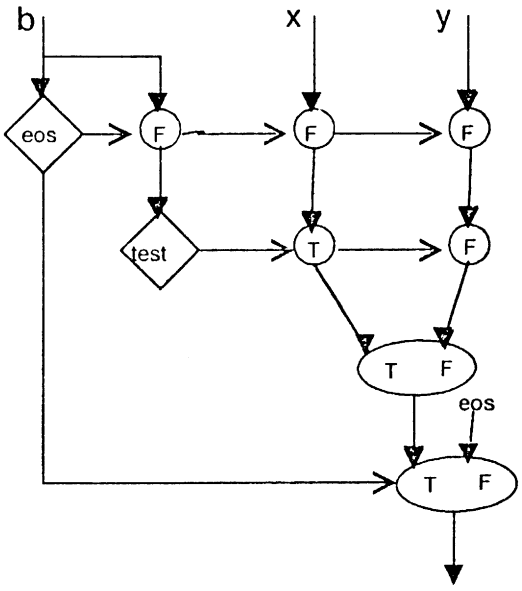


(a)



(b)

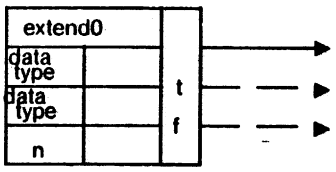
Fig. 4.14



STREAM CONDITIONAL

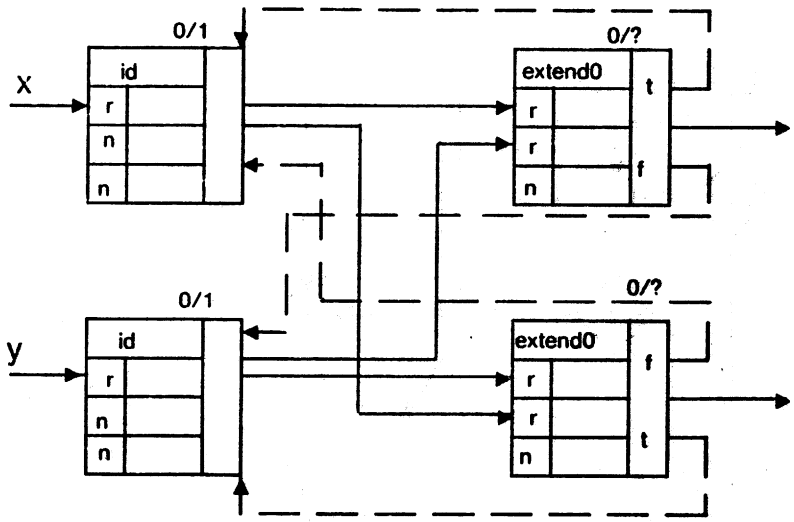
(c)

Fig. 4.14



EXTEND0 INSTRUCTION CELL

(a)



INSTRUCTION CELL TRANSLATION OF STREAM EXTEND(X,Y)

(b)

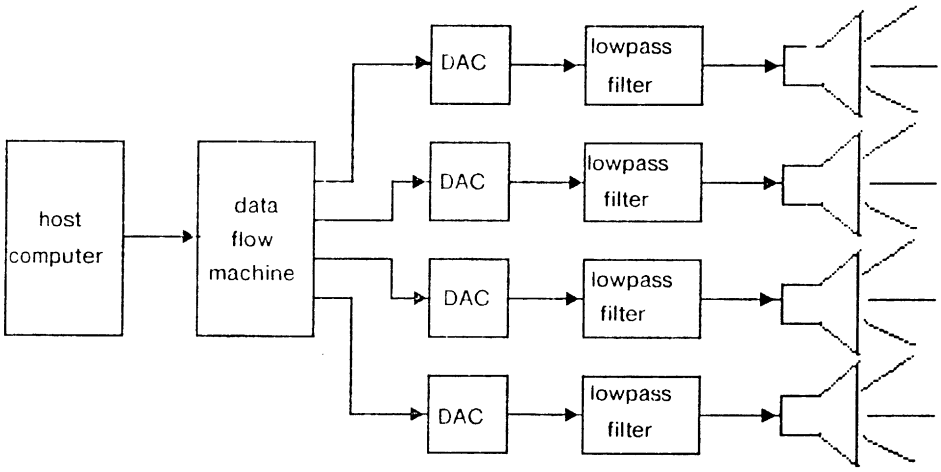
Fig. 4.15

5. DATA FLOW IMPLEMENTATION OF MUSIC SYNTHESIS

The third chapter described a representative computer synthesis system and synthesis language, both named Music-11. It was shown that a computer synthesis system similar to the Music-11 system that allowed the modelling of reasonably complex instruments cannot perform in real time due to the sequential nature of conventional computers. The potential for parallel computation in the sample calculation process is great. One can easily see the parallelism at the level shown in Fig. 3.1. Since each instrument is an independent entity, the sound outputs for each instrument can be calculated at the same time. Concurrency also exists in the models of each instrument in the orchestra file. For example if the model in Fig. 3.2 is used, the two sinusoidal components can be computed in parallel before summing. And yet another level of parallelism lies in the algorithms for many of the unit generators. A computer synthesis system could derive numerous benefits from data flow. This chapter describes a proposed implementation of a synthesis system that contains a data flow computer. This proposed system will be referred to as Music-df.

5.1 Physical Layout of the Music-df System

The Music-df system is envisioned to look like Fig. 5.1. The composer uses a language similar to Music-11 to create and edit orchestra and score files on the host computer. This computer is a machine whose architecture is conventional and could possibly be a PDP 11/50. When the user is satisfied with his files, he invokes a program that first sends the function statements of the score file to the data flow machine where the appropriate function tables are constructed. The orchestra file is compiled into executable instruction cells on the host machine and transmitted to the data flow machine. Then the host computer sorts the score file and carries out the beat to second conversion following the specifications of tempo statements in the same manner as for Music-11. After the sorting and orchestra file compilation have taken place, the note statements of the score file are sent



PROPOSED MUSIC-DF SYSTEM

Fig. 5.1

to the data flow machine and performance begins.

5.2 Music-df Language

The language that is to be used on the system will be called Music-df. It is similar to Music-11 and retains much of the syntax of the Music-11 language except for several changes in both the orchestra and score files. Some of the changes are made from necessity while others are done to make the files more readable and meaningful to someone other than their composer. The following two subsections describe the modifications that are made to the Music-11 language to derive Music-df a language that will run on the Music-df system.

5.2.1 Orchestra file modifications

In Music-df *kr* and *sr* are restricted to be defined as real numbers in the orchestra header. They must be whole numbers and as in Music-11 *sr* must be divisible by *kr*. *Ksmps* still must be defined as an integer and equal to **int**(*sr/kr*). Unlike Music-11, *kr,sr* and *ksmps* are available to the composer and can be used in expressions in the orchestra file.

The signal naming restrictions are lifted in Music-df. Control rate signals and audio rate signals need not be named kn and an, respectively. The user may choose any name he wishes. This however may pose a problem. Consider the two fragments of Music-11 code:

```
;            ia       idur    ib
k1    line   10000,   5,      5000
```

and

```
;            ia       idur    ib
a1    line   10000,   5,      5000
```

Even though the input parameters for line are the same in both cases, k1 and a1 are not equivalent signals. They vary at different rates and are calculated differently. The Music-11 monitor knows which type of signal the output should be by noticing whether the first character of the output signal

is k or a. This naming restriction no longer exists in Music-df. So, two functions exist in Music-df, kline for control rate signals and aline for audio rate signals. For every function in Music-11 that can output either an audio or control rate signal two corresponding functions are offered in Music-df. The function name is prefixed with an a or k.

Another naming restriction that is lifted is that instrument identifiers be integers. In Music-df instrument identifiers are character strings. This allows an identifier to be descriptive. It is useful when reading an orchestra file to be able to know at a glance that an instrument block probably describes a clarinet-like instrument because the instrument identifier is "clarinet".

Since the orchestra file is to be compiled into instruction cells that run on a data flow machine there are two features of the Music-11 language that must be replaced. One is the tolerance of side effects through the use of use of global variables. An important use of side effects in Music-11 occurs when each instrument is to be modified in an identical manner before becoming an output. The global variable accumulates the output signals of all the instruments so that their sum can be modified instead of having to modify each one separately. Fig. 3.6 is an example where the modification is reverberation. It is a desirable feature to be able to accumulate the instruments' outputs so Music-df retains this capability by keeping global variables while excluding side effects.

A Music-df translation of Fig. 3.6 is shown in Fig. 5.2. In Music-df the orchestra file is required to lie between a beginorchestra and an endorchestra line, making it a block in its own right. A new statement, the output statement signals the output of the instrument block in which it is contained. In Music-df, the instrument block outputs may be assigned to global variables whose scope is the orchestra block. Global variables may be used in any instrument block but they may not be modified in any way within an instrument block. Now the instruments' outputs can be summed and modified (or not modified) as a whole before becoming the argument to an out statement. The out statements are restricted to lie outside the instrument blocks in the orchestra block. The modification of the sum of the instruments' outputs may consist of several signal functions whose input parameters the

```
beginorchestra

sr=50000.
kr=500.
ksmps=100
nchnls=2

a1 = instrument one
.
.
.
;                    xamp  xcps  ifn  iphs
        a1  aoscil   k2,    k3,   1,   -1
        output a1
        endin

a2 = instrument two
.
.
.
;                    xamp  xcps  ifn  iphs
        a1  aoscil   k2,    k3,   1,   -1
        output a1
        endin

;           asig      idlt
a3  reverb  a1+a2,  2.49
outs  a3/2,  a3/2

endorchestra
```

MUSIC-DF ORCHESTRA FILE THAT CONTAINS GLOBAL VARIABLES

Fig. 5.2

composer may wish to vary from note to note. Therefore the user can specify these parameters in score file note statements whose first parameter is *out*. This is a reserved instrument name. No other instrument may be called *out*.

Another feature of the Music-11 orchestra file that cannot be translated into data flow is goto statements. The instruction cell language of a data flow machine is fundamentally different from the machine language of a conventional computer. In keeping with data flow principles it has no concept of a program counter. Therefore a goto statement is meaningless and cannot exist in data flow. The following conditional statement is offered in Music-df in lieu of goto statements,

```
if signal1 <test> signal2
then
        <truecode>
else
        <falsecode>
endif
```

where *signal1* and *signal2* may be signals of any rate and <test> is one of the following: =, ~=,>, <, >=, <=. <truecode> and <falsecode> can be any legal Music-df code that can occur in instrument blocks. As an example, the Music-11 instrument block in Fig. 3.5 would become the following Music-df instrument description:

```
instrument one

if  p4 > 1000
then
;                       ia      idur    ib
        k1      kline   1000,   p3,     500
else
;                       ia      idur    ib
        k1      kexpon  700,    p3,     400
endif
;               xamp    xcps    ifn     iphs
a1      aoscil  k1,     p4,     1,      -1
output  a1
endin
```

The following restriction is placed on the conditional statement in Music-df. Let *rate* be the

rate (ie. audio, control, note) of *signal1* or *signal2* whichever has the highest rate. Any signal (variable) that is defined in the conditional statement and used outside the conditional statement cannot have a rate slower than *rate*. The instrument description above adheres to this rule because k1 is control rate and p4 is note rate.

Finally, in Music-df once a value is assigned to a variable name it cannot be reassigned. The scope of a variable name introduced and defined in an instrument block is that instrument block alone. The scope of a variable name introduced and defined in the orchestra block but outside the instrument blocks is the entire orchestra file less any instrument blocks where the same variable name is reintroduced and defined.

5.2.2 Score file modifications

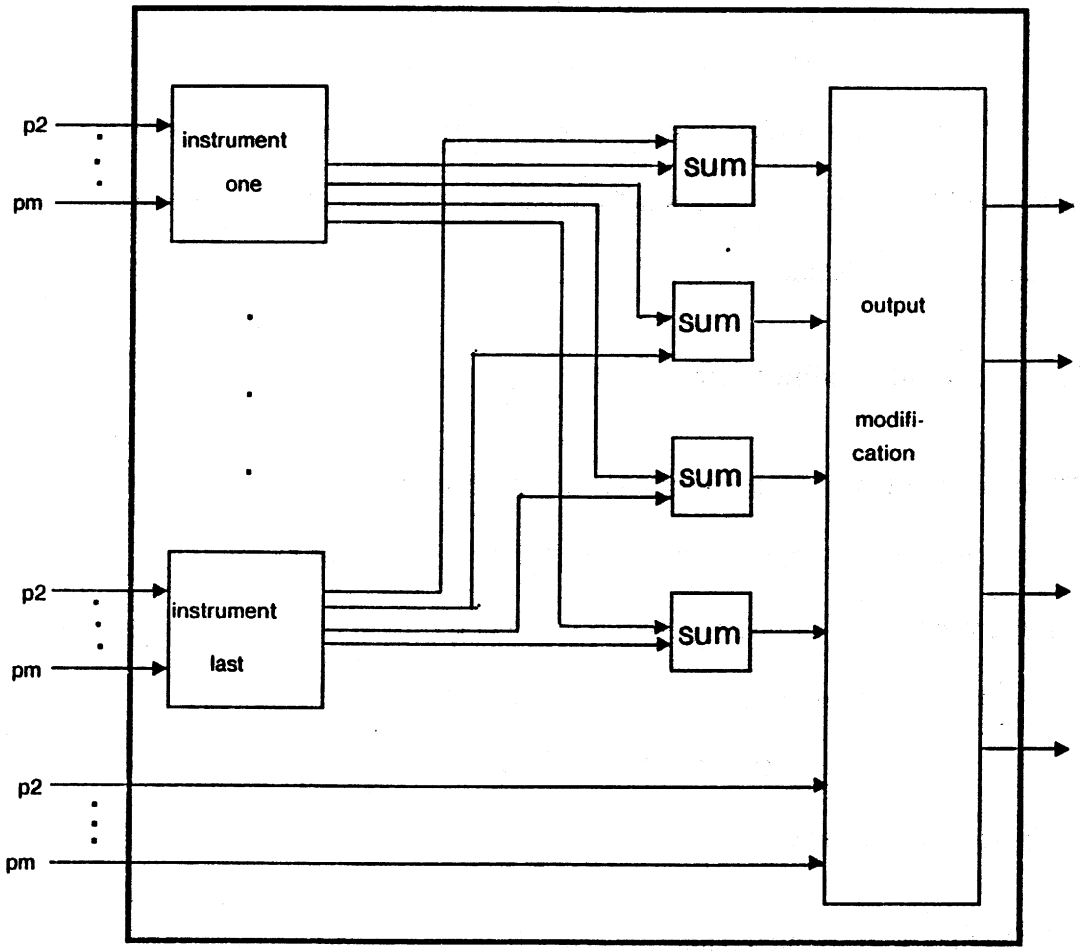
The only changes to the score file are that the single letter that specifies the score statement types in Music-11 is expanded to a word in Music-df. The letter **i** is replaced by **note**, **f** by **functiontable**, **t** by **tempo** and **e** by **end**. The only exception is the letter **c** which is continued to be used for comment line. As an example, note statements take the form,

note *p1 p2 p3 p4...p128*

in Music-df where *p1* is now a character string.

5.3 Orchestra File Conversion

The converted orchestra file in the data flow machine may be viewed as Fig. 5.3 where all the boxes contain executable instruction cells. The instrument blocks labelled instrument one and instrument n contains translations of the instrument descriptions in the orchestra file. Their inputs come from the note parameter packets sent by the host machine. The instrument outputs are then summed and modified using the parameters from the note statements whose first parameter is equal to 'out'.



ORCHESTRA FILE

Fig. 5.3

It is important to note that if the desired sampling rate of 50 Khz is to be achieved, the four output lines of the output modification box must produce a value every 20 microsec. In order for this to occur, the instruction cell translation of the orchestra file must be maximally pipelined.

5.3.1 Instrument translation

At first thought, it may seem that the instruction cell graphs in the boxes labelled instrument one and instrument last in Fig. 5.3 could be direct translations of the instrument blocks in the orchestra file. However, if they were in fact direct translations of the instrument blocks, the performance would not occur in the manner that the composer wishes if the score file contained rests. For example, consider the score,

```
c       instrument      start   duration. . .
note    one             0       2. . .
note    one             4       2. . .
note    one             8       2. . .
end
```

for a one instrument orchestra. If the instrument boxes in Fig. 5.3 contained direct translations of the instrument blocks, then three two-second notes will be played in succession. But according to the score there should be a two-second rest after the first and second notes. During these rests the instrument should remain silent. The equivalent of silence is an output of zero for each sample of the rests' durations. Thus the instrument boxes in Fig. 5.3 must contain a control structure to ensure each instrument is played (or not played) at the appropriate times.

The control structure must eventually be expressed at the instruction cell level. But it is easier to first describe the control structure in a high-level language (in this case, VAL) and then translate it into instruction cells. In order to do this, it is assumed that each instrument in the orchestra file is a VAL function that takes the parameters from the note statements in the score file as its input parameters. It is assumed that a VAL function exists for each instrument block in the orchestra file. These VAL instrument functions are direct translations of the orchestra file instrument blocks and

take the parameters from the note statements in the score file as their input parameters. The data type input parameters of the instrument functions is **stream[real]**. Thus the instrument functions accept a stream of *p2*, a stream of *p3*, a stream of *p4* etc. as their inputs. The output of each instrument function is of type **stream[stream[real]]** each stream in the stream of streams corresponding to a note in the score file. The output may be of arity one, two or four depending on mono, stereo or quadrophonic sound. A control structure that will accomodate rests is acheived by the four VAL functions in Figs. 5.4(a-d). Since *p2* and *p3* in the score file note statements specify the starting time and duration of each note, the variable names *starttime* and *noteduration* are the variable names for *p2* and *p3* in the VAL functions of Fig. 5.4.

The function convert_streams takes streams *starttime* and *noteduration* as inputs and outputs two streams. One output stream *restduration* contains real values corresponding to durations in seconds of all the rests in the score. The second stream *bval* is made up of boolean values and determines whether the next duration is to be chosen from the stream *noteduration* or the stream *restduraiton*. If the next value in *bval* is true the next duration will be taken from *noteduration*, otherwise *restduration* will yield the next duration. Thus *bval* decides whether the next duration is an actual note or a rest. Taking the above score, *starttime*=[0,4,8] and *noteduration*=[2,2,2]. Convert_streams(*starttime,noteduration*) yields two streams *restduration*=[2,2] and *bval*=[true,false,true,false,true].

The function silence expects its input *restduration* to be a stream of real values whose unit is seconds. For each member of *restduration*, silence constructs a stream of zeroes. The number of samples in this stream is equal to the number of samples in a time duration equal to that member of *restduration*. These streams are assembled into one stream to make the **stream[stream[real]]** output that silence produces.

The function repeat_bool accepts two inputs, one *bval*, of type **stream[bool]** and the other *totaltime* of type **stream[real]**. These streams should be the same size. For each $totaltime_{ii}$ a stream

```
function convert_streams(starttime,noteduration : stream[real] returns
                                                  stream[real],stream[bool]);

for
        starttime : stream[real] := starttime;
        noteduration :stream[real] := noteduration;
        bval : stream[bool]  := [ ];
        restduration : stream[real] := [ ];
        laststarttime : real :=0;
        lastnoteduration : real :=0;
do
        if  starttime = eos then restduration,bval
        else iter
        restduration,bval := if laststarttime+lastnoteduration=first(noteduration)
                    then consl(bval,true),restduration
                    else consl(consl(bval,false),true),
                         consl(restduration,first(noteduration)-laststarttime-lastntoteduration)
                endif;
        laststarttime:=first(starttime);
        lastnoteduration:=first(noteduration);
        starttime:=rest(starttime);
        noteduration:=rest(noteduration);
        enditer
        endif
endfor
endfun
```

(a)

```
function silence(restduration : stream[real] returns stream[stream[real]]);

for
        durations : stream[real] := restduration;
        quiet : stream[stream[real]] := [ ];
do
        if durations = [ ] then quiet
        else iter
        quiet := consl(quiet,for
                        quiet2 : stream[real] := [ ];
                        count : integer := int(first(durations)*kr)*ksmps;
                do
                        if count <= 0 then quiet2
                        else iter
                        quiet2 := consl(quiet2,0.0);
                        count := count-1;
                        enditer
                        endif
                endfor))
        durations := rest(durations);
        enditer
        endif
endfor
endfun
```

(b)

CONTROL STRUCTURE FUNCTIONS

Fig. 5.4

```
function repeat_bool(totaltime : stream[real] ; bval : stream[bool] returns stream[stream[bool]]);

for
        duration : stream[real] := totaltime;
        decide : stream[stream[bool]];
        bval : stream[bool] := bval;
do
        if durations = [ ] then decide
        else iter
        decide := consl(decide,for
                        decide2 : stream[bool] := [ ];
                        count : integer := int(first(durations)*kr)*ksmps;
                do
                        if count <= 0 then decide2
                        else iter
                        decide2 := consl(decide2,first(bval));
                        count := count-1;
                        enditer
                        endif
                endfor))
        durations := rest(durations);
        bval := rest(bval);
        enditer
        endif
endfor

endfun
```

(c)

```
function control(starttime,noteduration,p4. . .pm: stream[real] returns stream[stream[real]);

let
        play1,play2,play3,play4 : stream[real] := <name>(starttime,noteduration,p4. . .pm);
        bval,restduration : stream[bool],stream[real] :=
                                                convert_streams(starttime,noteduration);
        totaltime : stream[real] := stream_merge(noteduration,restduration,bval);
        quiet : stream[stream[real]] := silence(restduration);
        decide : stream[stream[bool]] := repeat_bool(totaltime,bval);
in
        stream_merge(play1,quiet,decide), stream_merge(play2,quiet,decide),
        stream_merge(play3,quiet,decide),stream_merge(play4,quiet,decide)
endlet

endfun
```

(d)

CONTROL STRUCTURE FUNCTIONS

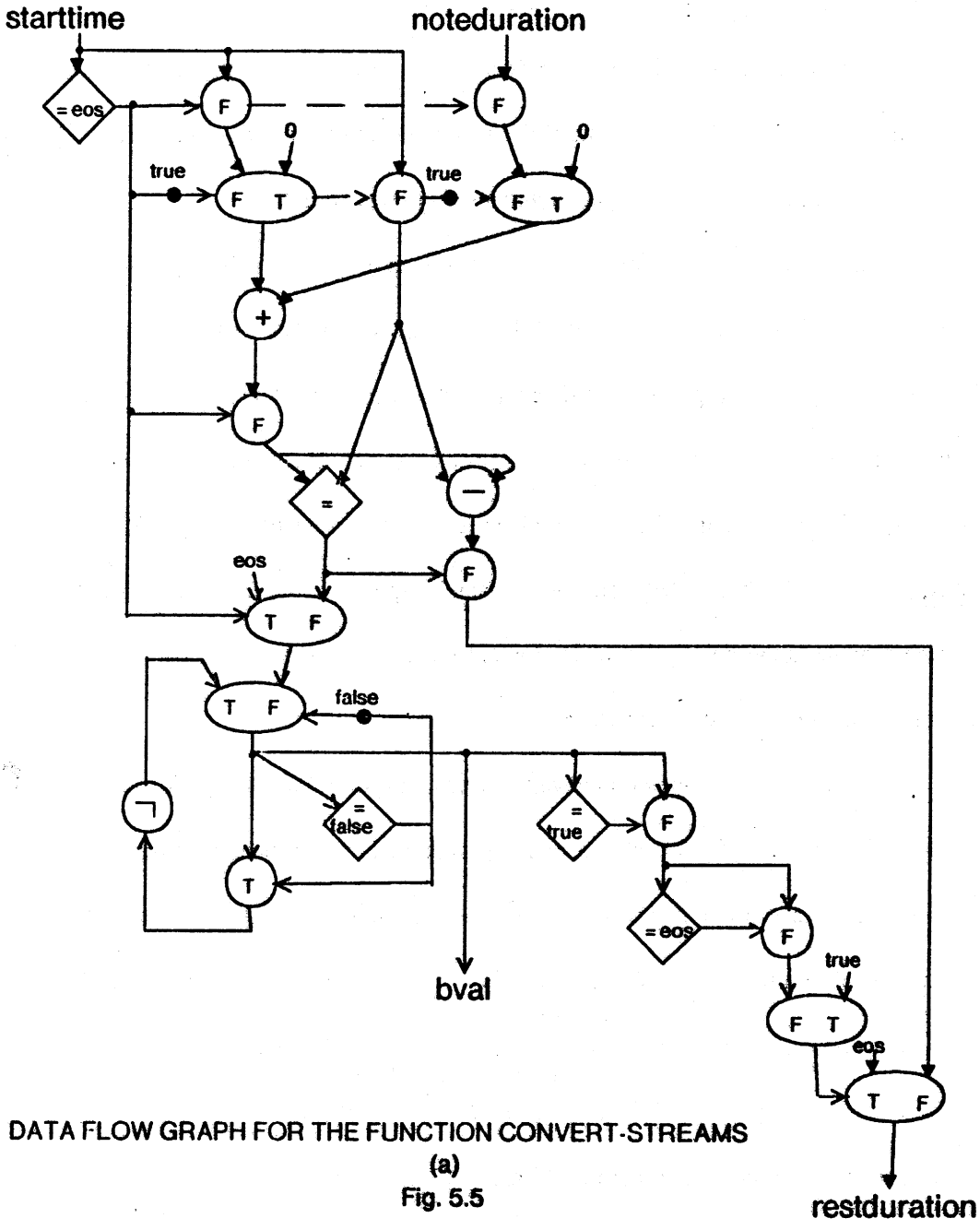
Fig. 5.4

of boolean values is constructed, whose members are all equal to $bval_n$. The size of each stream is the number of samples in a duration of $totaltime_n$. For A = [2,3], B = [true,false], kr=1, $ksmps$=1, repeat_bool(A,B) = [[true,true],[false,false,false]].

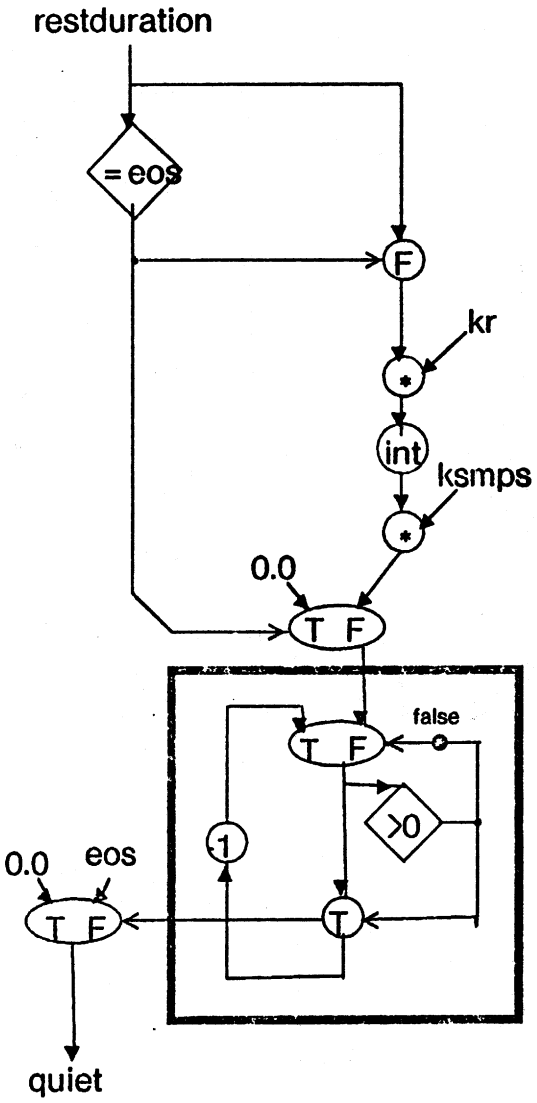
The function control (Fig. 5.4(d)) uses the output streams of convert_streams to either play the instrument or initiate rests. The function <name> is a description of the sound of the instrument whose identifier is <name>. The output of the function control is of arity four because the output of the function <name> is of arity four, signifying quadrophonic sound.

The flow graphs for these four functions are shown in Fig. 5.5(a-d). Fig. 5.5(d) is important to note because it depicts what may be called the total instrument. The total instrument consists of the function control whose output is fed through a function destream. The function destream accepts an input of **stream[stream[real]]**. Its output contains all the real values present in the input except that its data type is **stream[real]**. The flow graph for destream is depicted in Fig. 5.5(c). In subsequent figures, whenever a box labelled instrument <name> appears, it replaces Fig. 5.5(d). A box labelled <name> represents the description of instrument <name> in the orchestra file or in other words a direct translation of the instrument **block**.

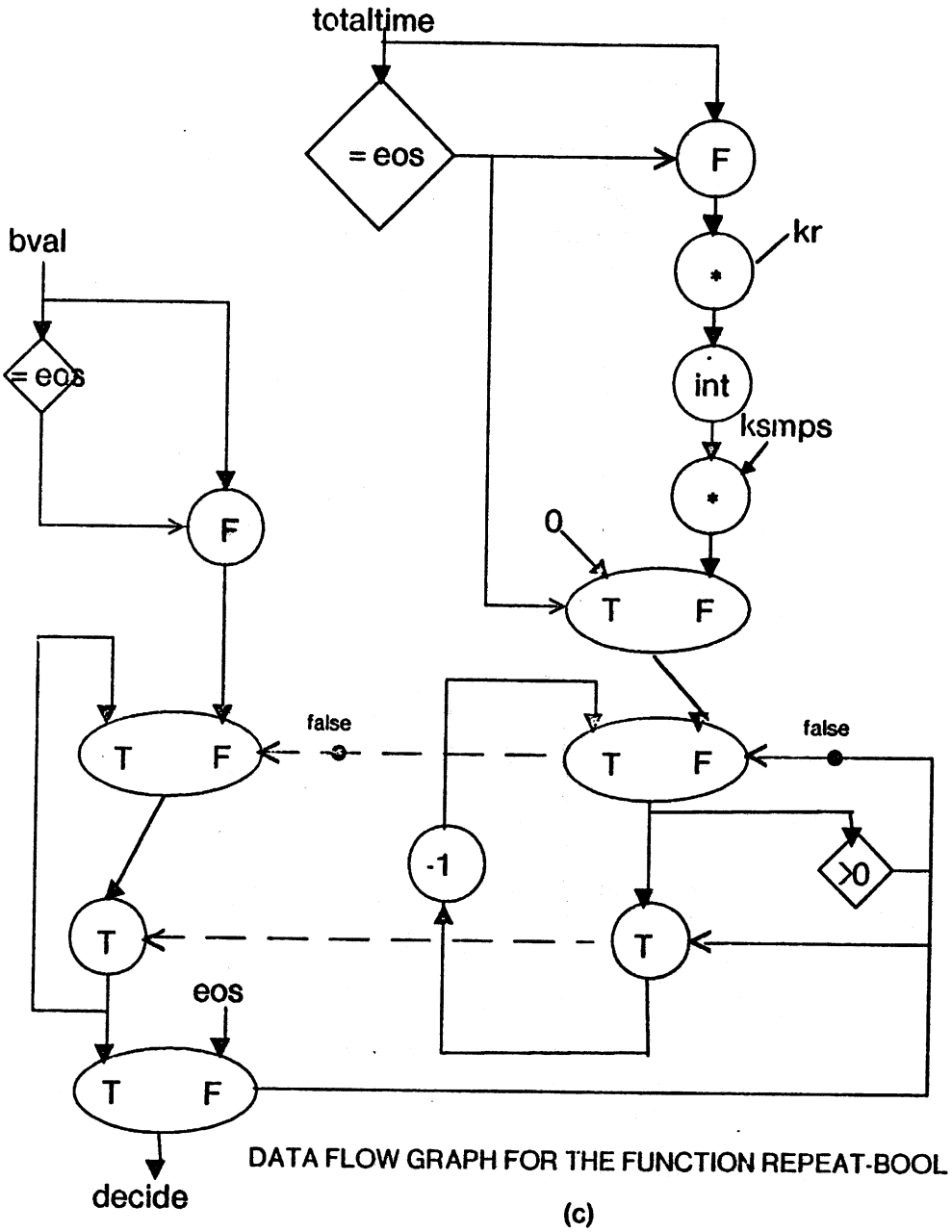
The instruction cell graphs for convert_streams and control are straightforward and will not be detailed. The cell graphs for silence and repeat_bool however bear investigation. Since these functions output audio rate signals they should be maximally pipelined. This means that the ">0" operator should output a value every other tick. A straightforward cell graph translation of these does not yield optimally pipelined functions. The boxed part of Fig. 5.5(b) is where the problem lies. A direct translation of these cells is shown in Fig. 5.6(a). The reader can see that this graph does not produce an output value every other tick. Fig. 5.6(b) shows a non-direct translation that is maximally pipelined. It is this graph portion that replaces the box labelled loop control in Fig. 5.6(a). (The loop control box is used in several more figures as a substitution for the four flow graph actors in Fig. 5.5(a).) Figs. 5.7(a,b) show maximally pipelined versions of silence and repeat_bool.



DATA FLOW GRAPH FOR THE FUNCTION CONVERT-STREAMS
(a)
Fig. 5.5



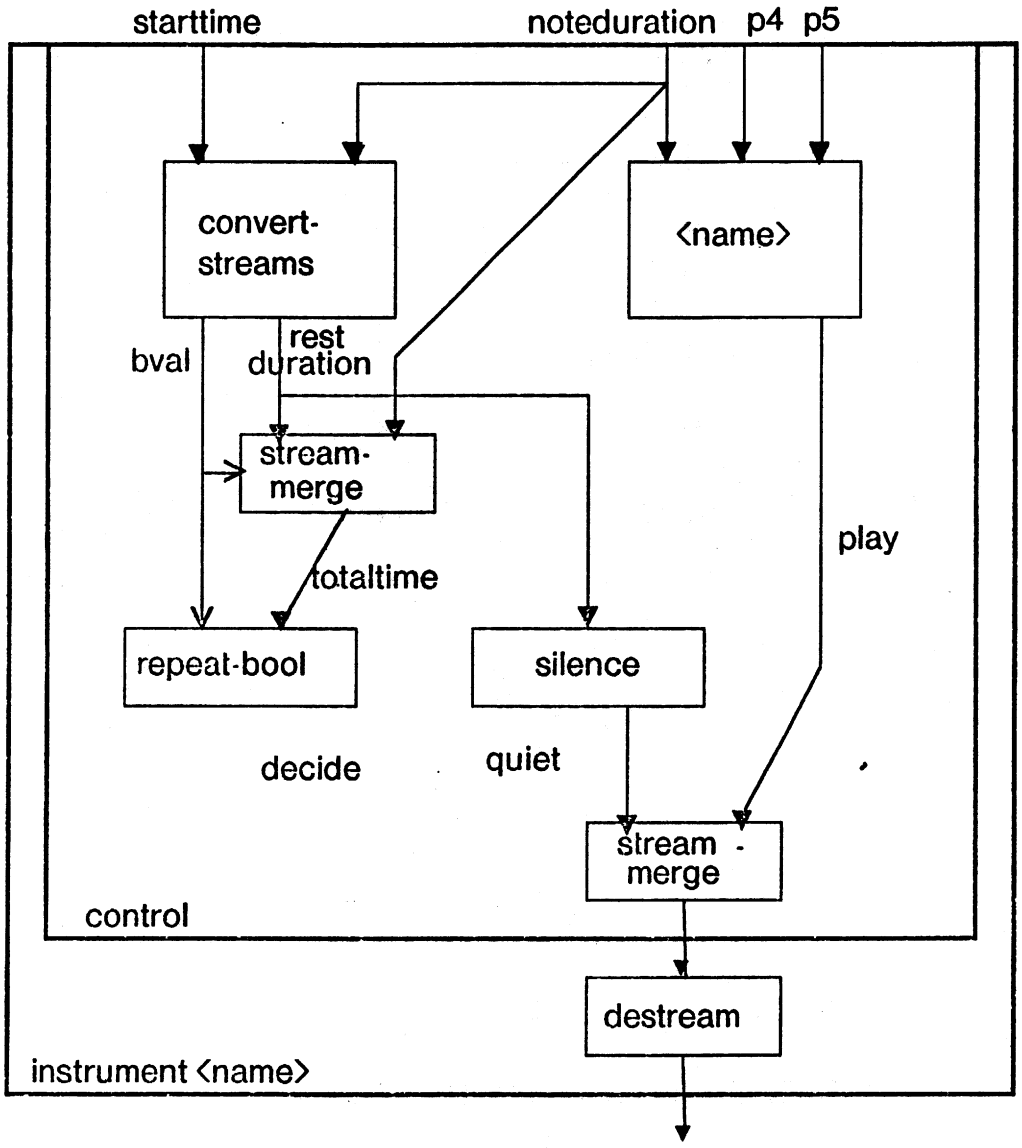
FLOW GRAPH FOR THE FUNCTION SILENCE
(b)
Fig. 5.5



DATA FLOW GRAPH FOR THE FUNCTION REPEAT-BOOL

(c)

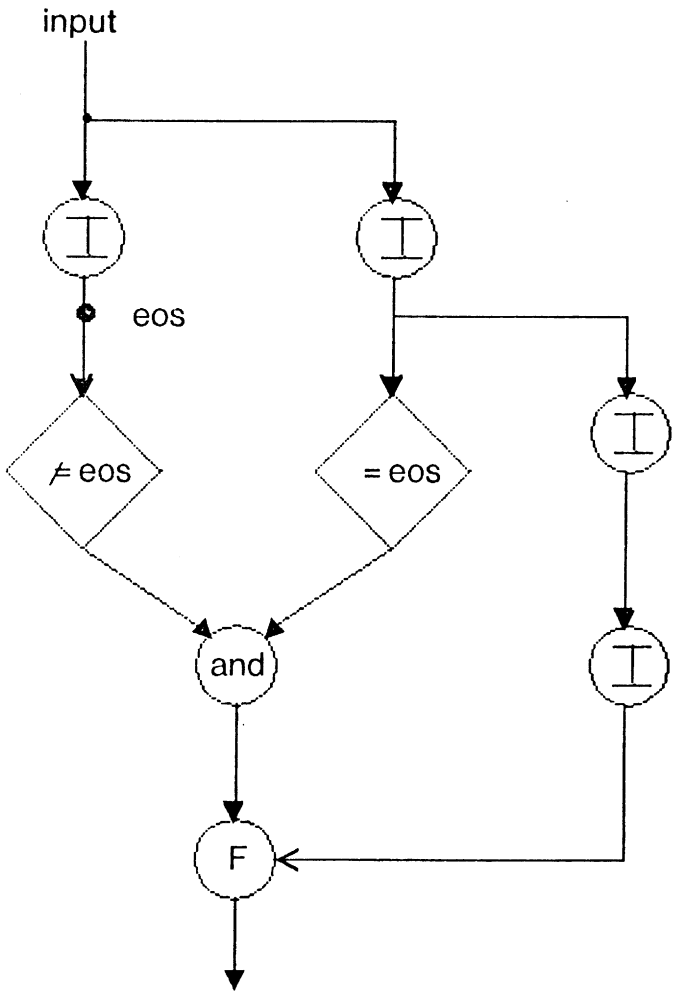
Fig. 5.5



DATA FLOW GRAPH FOR THE FUNCTION CONTROL
AND THE TOTAL INSTRUMENT

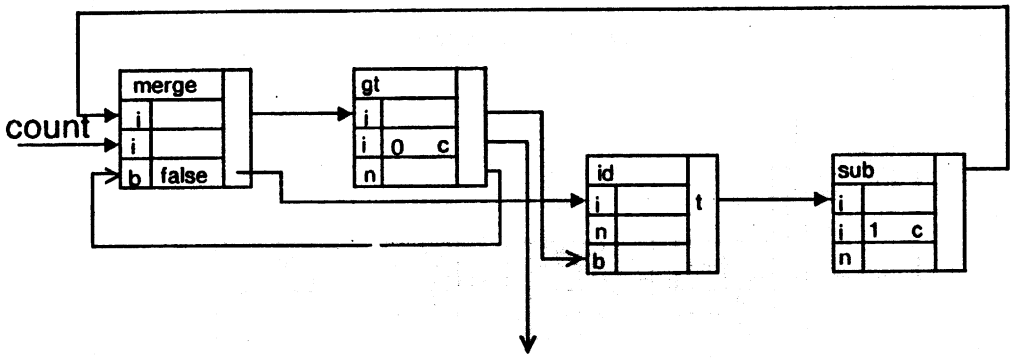
(d)

Fig. 5.5



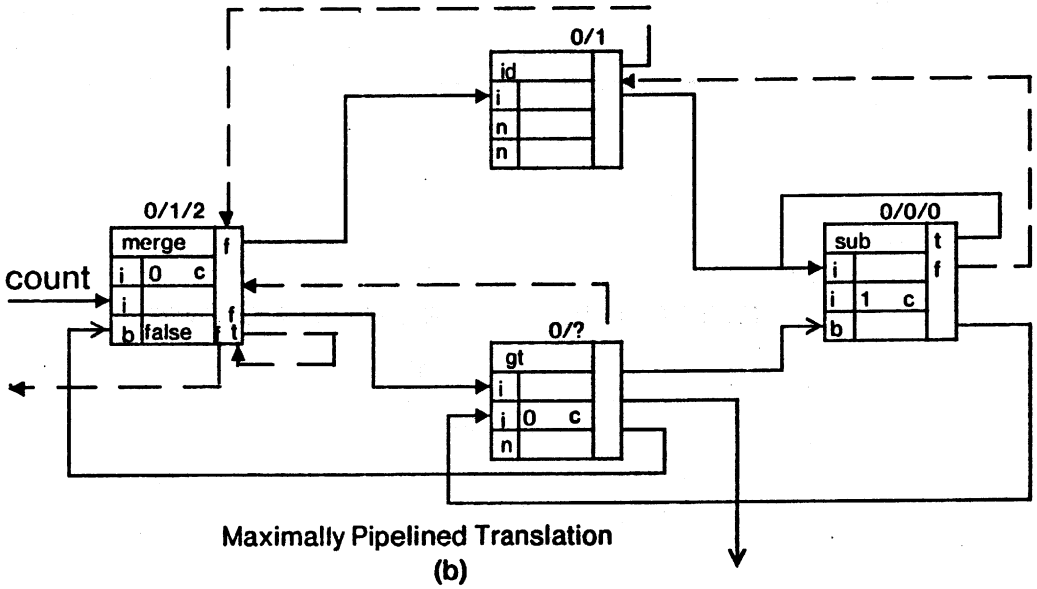
DATA FLOW GRAPH OF DESTREAM
(e)

Fig. 5.5



Direct Translation

(a)

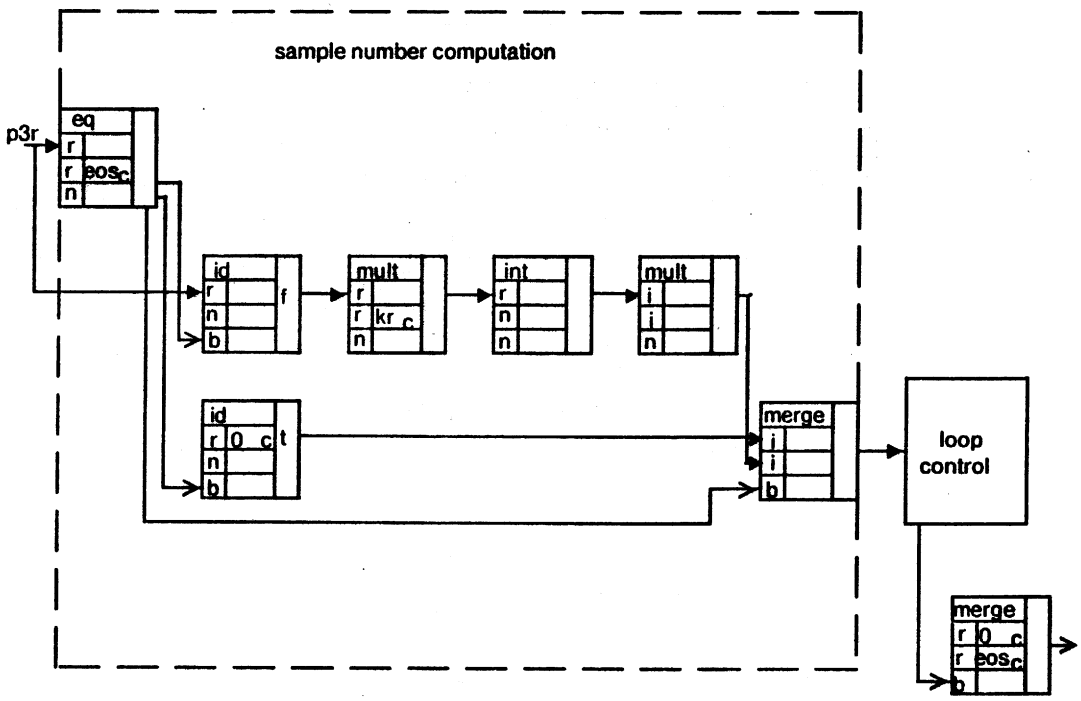


Maximally Pipelined Translation

(b)

INSTRUCTION CELL TRANSLATIONS OF LOOP-CONTROL SUBGRAPH

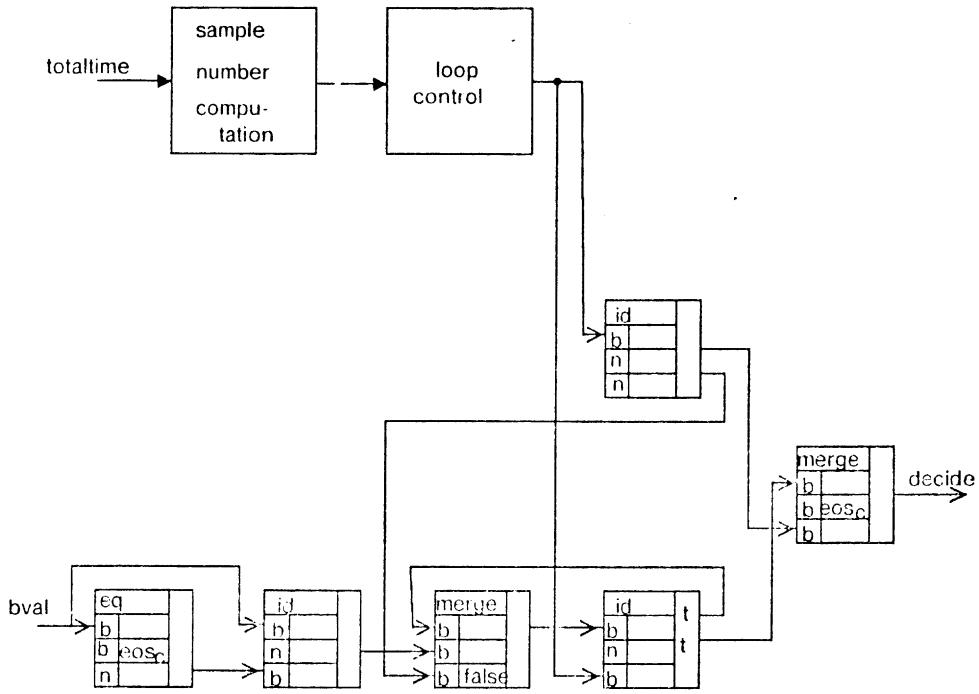
Fig. 5.6



INSTRUCTION CELL TRANSLATION OF SILENCE

(a)

Fig. 5.7



INSTRUCTION CELL TRANSLATION OF REPEAT-BOOL

(b)

Fig. 5.7

5.3.2 Instrument output summing

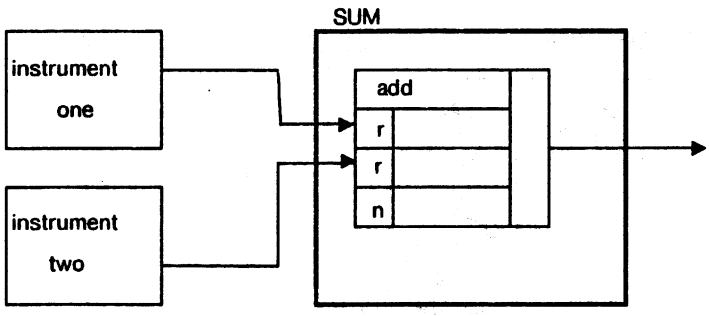
The simplest implementation of the boxes labelled sum in Fig. 5.3 would be add instruction cells (Fig. 5.8(a)). The number of add instruction cells needed would be *n-1* where *n* is the number of instruments in the orchestra file. But consider the following score file that plays a two instrument orchestra,

```
c       instrument      start   duration. . .
note    one             0       2. . .
note    one             2       4. . .
note    two             0       10. . .
end
```

Instrument two plays six beats longer than instrument one and thus outputs more samples than instrument one. If the two instruments each send their output to different speakers the piece will sound as it was intended to sound. However if the output of the instruments is summed before being sent to a speaker there is a problem. If the number of output samples from the instruments does not match up and the summing implementation in Fig. 5.8(a) is used, there will be values left in instruction cells after performance. Fig. 5.8(b) shows a solution to the problem. The output of the instrument whose output stream is shorter is extended to the length of the output stream of the other instrument, its new members equal to zero. The two streams are then added. If the two streams were to be multiplied, **extend1** operators would replace the **extend0** operators. To sum the output of *n* instruments, each instrument's output would be destreamed and *n-1* of the box labelled stream sum in Fig. 5.8(b) would be used.

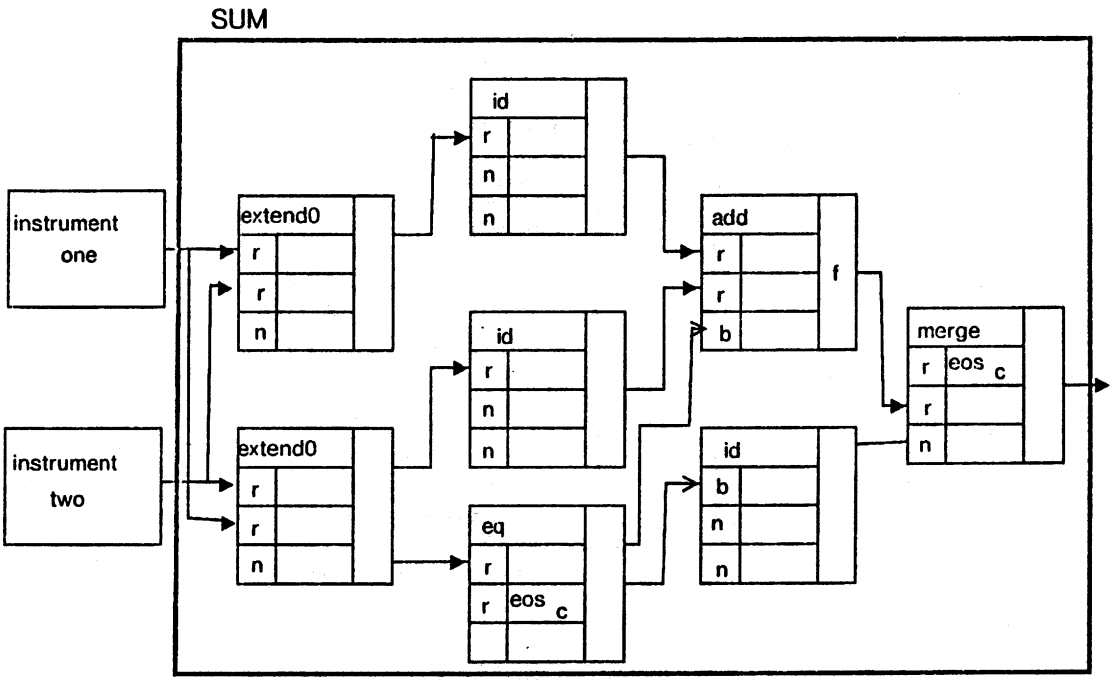
5.3.3 Orchestra output modification

The contents of the output modification box in Fig. 5.3 are shown in Fig. 5.9. The reader is reminded that *starttime* and *noteduration* are equivalent to *p2* and *p3*. Note that Fig. 5.9 assumes monophonic sound whereas Fig. 5.3 depicts an orchestra that outputs to four speakers. The input



Incorrect

(a)

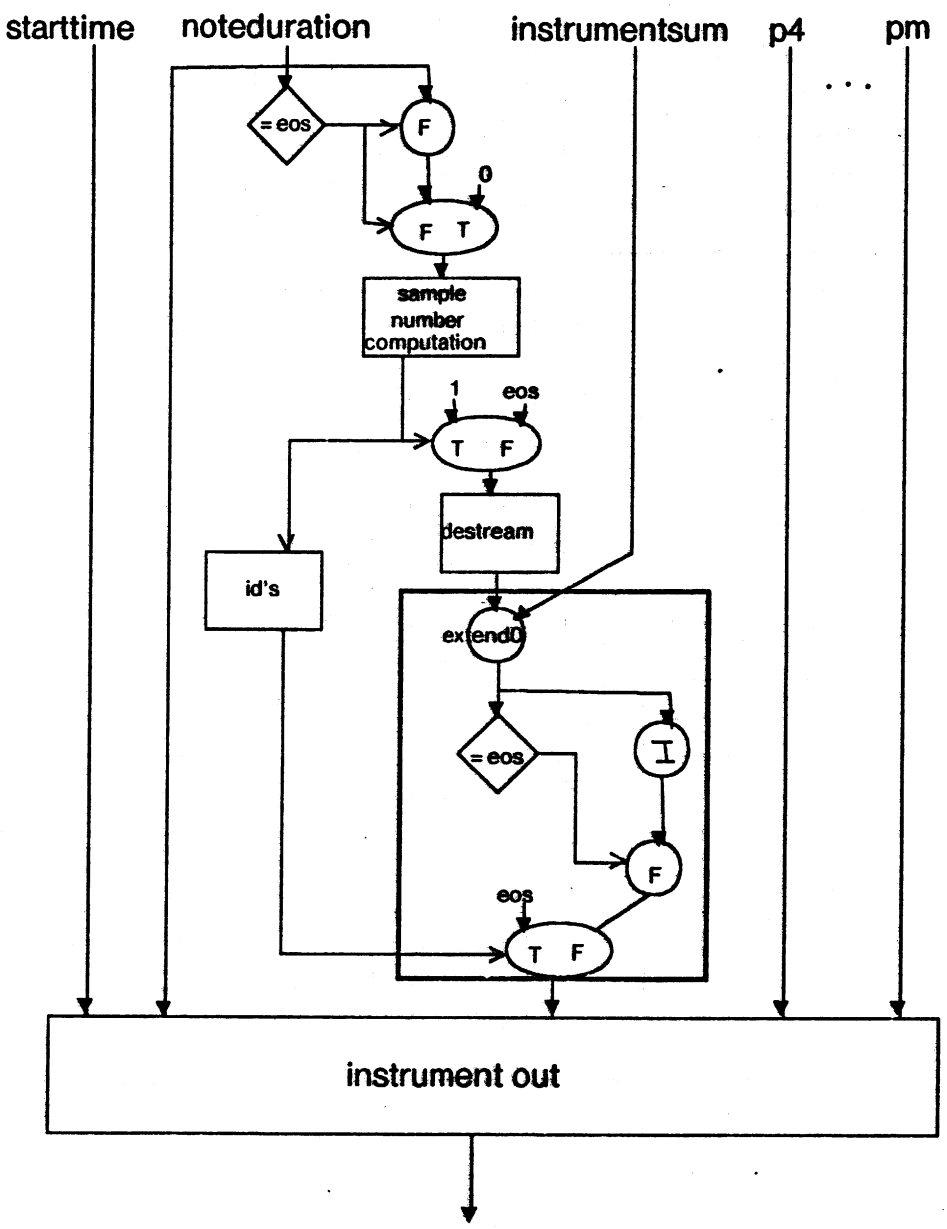


Correct

(b)

INSTRUCTION CELL TRANSLATIONS OF INSTRUMENT SUMMING

Fig. 5.8



DATA FLOW GRAPH OF OUTPUT MODIFICATION
Fig. 5.9

instrumentsum is the output of a sum box in Fig. 5.3 and its type is **stream[real]**. If it is to be used as an audio rate input to a signal function in the out instrument, it must be of type **stream[stream[real]]**. Therefore in instrumentsum is extended to contain a number of values equal to the number of samples in the total playing time of the instrument out (ie. the sum of all the *p3* values). Let *n2* be the number of members in a stream, *n* not necessarily equal for two arbitrary streams. Then it is required that $\sum_{k=0}^{n-1} p3_k$ for the instrument out be greater than or equal to $p2_{n-1}+p3_{n-1}$ for any instrument. After instrumentsum is extended, *eos* tokens are inserted in the stream, the values between the *eos* tokens corresponding to the noteduration values in the score file note statements for the instrument out. The insertion of these *eos* tokens converts instrumentsum from type **stream[real]** to type **stream[stream[real]]** so that it can now become an input to the instrument out.

If the orchestra file had four output streams (quadraphonic sound), then the boxed subgraph in Fig. 5.9 would be duplicated three times to accomodate the extension and data type conversion of four input streams.

5.4 Signal Functions

This section deals with the building blocks of the orchestra file, the signal functions. These functions can be broken down into three groups, the envelope generators, the oscillators and the signal modifiers. In the next three subsections, at least one example of each function type is worked through. For each function, VAL code is given first so that the reader can more easily understand the algorithm involved. The VAL code is followed by a flow graph representation and then an implementation using instruction cells. In deriving the instruction cell graphs, several considerations were taken into account.

In Music-11, each signal function is called every control period. As explained in the first chapter, this means that functions producing control rate signals output one real value for each function call and functions producing audio rate signals output an array of real values of size ksmps.

In Music-df, a signal function is called only once during the entire performance. Thus the output of each function when expressed in VAL is of type **stream[stream[real]]**. Each stream in this stream of streams corresponds to a note in the score file. If the output of the signal function is control rate, each stream in the output stream will contain **int(p3*kr)** real values. If the output stream is audio rate, each stream will consist of **int(p3*kr)*ksmps.**

In Music-11 most signal functions take input signals that can be one of several rates. For instance in the Music-11 statement,

an **oscil** *xamp, xcps, ifn ,iphs*

The inputs xamp and xcps may be constants, note rate, control rate or audio rate. On the other hand, the signal functions in Music-df expect an input to be the highest possible rate. For example the *xamp* variable in Music-df **aoscil** function is expected to be audio rate. If it is not audio rate, then aoscil will not yield the desired output. Oftentimes, a composer would like to use a control rate signal as the *xamp* or *xcps* input. In order that the user not have to concern himself, it is assumed that an orchestra file compiler will check the inputs to the signal functions and convert them if necessary. For this purpose, Music-df has the following conversion functions.

1) **constant_to_control**
2) **constant_to_audio**
3) **note_to_control**
4) **note_to_audio**
5) **control_to_audio**

These functions convert a data type of the left hand side the function name to a data type of the right hand side of the function name. For instance, **constant_to_control** converts a constant number to a control rate stream and **note_to_audio** converts a note rate stream to an audio rate stream. As an example, for the following lines in an orchestra file,

```
;                    ia      idur    ib
ampenvelope   kline  p5,     p3,     p6
;
;             xamp   xcps    ifn     iphs
sig           aoscil sig1,   440,    2,      -1
```

the compiler would recognize that **aoscil** expects its first two inputs to be audio rate. Thus **control_to_audio**(ampenvelope) and **constant_to_audio(440)** would become the first two inputs to **aoscil** instead of ampenvelope and 440.

It was mentioned in section 5.2 that the orchestra file must be maximally pipelined if performance is to occur in real time. Thus the signal functions which are components of the orchestra file must also be maximally pipelined. All the signal functions are translated such that they output not only audio rate signals but also control rate signals every other firing to cover the possibility that $sr = kr$. All control rate and audio rate inputs to a signal function are assumed to arrive at the functions' input cells simultaneously and at an optimally pipelined rate. Note rate signals vary so slowly that that need not flow through a graph at an optimally pipelined rate.

Finally, *kr*, *sr* and *ksmps* are treated as constants since they do not vary within an orchestra file.

5.4.1 Envelope generators

Music-11 offers seven envelope generator functions. The simplest one to study but not the least instructive is **line**. This function was briefly explained in chapter one. The orchestra function statement for **line** in Music-11 takes either of the forms,

kn **line** *ia, idur, ib*

or

an **line** *ia, idur, ib*

Thus in Music-11, **line** can output either a control or audio rate signal. For Music-df two VAL functions were written, **kline** and **aline**. Both functions produce an output of type **stream[stream[real]]**. The output of **kline** is control rate and the output of **aline** is audio rate. The VAL function **kline** is shown in Fig. 5.10(a). In Music-df, **kline** needs one more input than in Music-11. This input is *p3*. This is because **kline** are called only once during the performance and needs a way of knowing how many samples to output per note.

```
function kline(ia,idur,ib,noteduration : stream[real] returns stream[stream[real]]);

for
        outstreams : stream[stream[real]] := [ ];
        noteduration : stream[real] := noteduration;
        ia : stream[real] := ia;
        idur : stream[real] := idur;
        ib : stream[real] := ib;
do
        if noteduration = eos then outstreams
        else iter
        outstreams :=
                consl(outstreams,
                        segment(first(ia),first(idur),first(ib),kr,int(kr*first(noteduration))));
        noteduration := rest(noteduration);
        ia := rest(ia);
        idur := rest(idur);
        ib := rest(ib);
        enditer
        endif
        endfor

endfun
```

(a)

```
funtion segment(ia,idur,ib,rate : real; count : integer returns stream[real]);

for
        inc : real := (ib-ia)/(idur*rate);
        y : real := ia;
        oustream : stream[real] := [ ];
        count : integer := count;
do
        if count > 0 then iter
        outstream := consl(outstream,y);
        y := y+inc;
        count := count-1;
        enditer
        else outstream
        endif
endfor

endfun
```

(b)

VAL FUNCTIONS KLINE AND SEGMENT

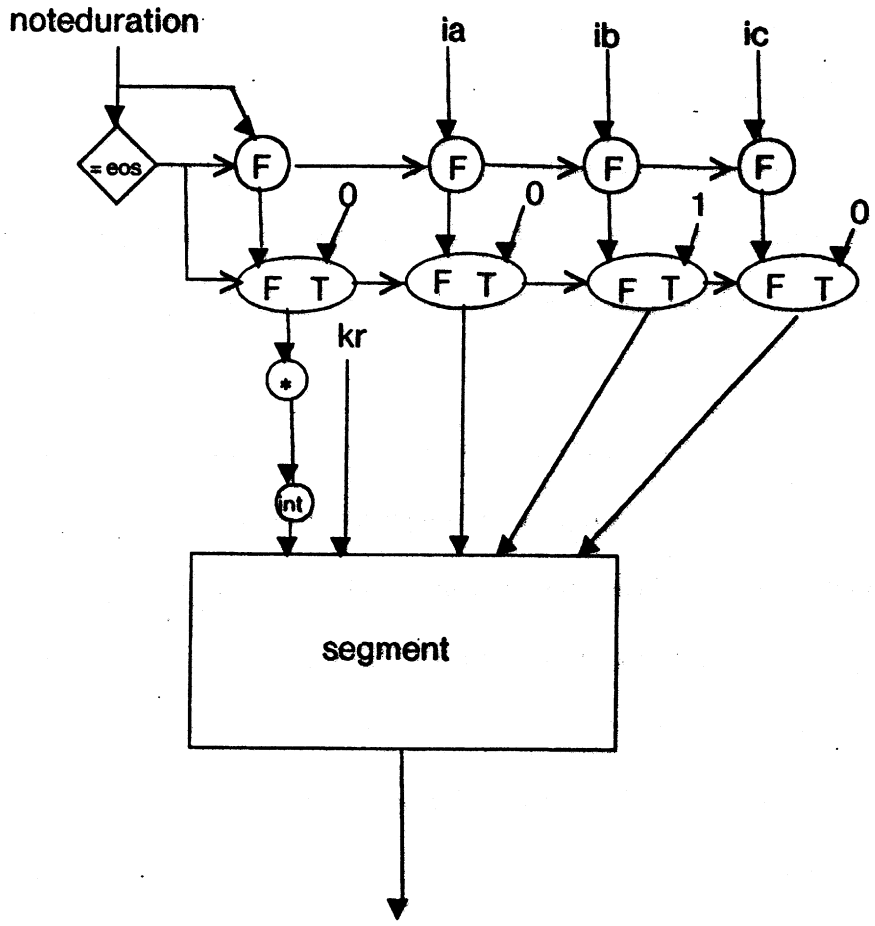
Fig. 5.10

For each member in the input streams of **kline**, the function segment (Fig. 5.10(b)) is called. The function segment outputs samples of a linear segment whose endpoints are *ia* and *ib*. The input *idur* is the amount of time that elapses between the endpoints. The input *rate* determines the rate of the output stream. If *rate=sr*, the output will be audio rate and if *rate=kr*, the output will be control rate. The length of the output stream is equal to the function input *count*. When **kline** calls segment, *count=***int***(kr*p3)*. This ensures that the size the segment's output stream is equal to the number of control samples in *p3* seconds, the note duration. Note that *idur* does not have to equal *p3*. It only serves to define the slope of the line segment. If *p3 < idur* the segment will be truncated and never reach the value *ib*. If *p3 > idur*, the segment will pass the value *ib* and continue on the same defined line.

The function **aline** is not shown but is similar to **kline** except that it calls segment with the *count* input parameter equal to **int***(kr*p3)*ksmps* and the *rate* input paramater equal to *sr*.

The flow graph translation of **kline** very straightforward as shown in Fig. 5.11(a). For each member in the stream *p3*, segment constructs a stream. After the last member of stream *p3* has travelled through the merge actor, the *=eos* condition will be true as the *eos* token appears on the input arc of the predicate operator. This causes an invocation of segment with a count of zero and the *ia, ib, idur* variables respectively equal to 0, 1 and 0. This is done to produce the final *eos* token for the **kline stream[stream[real]]** output.

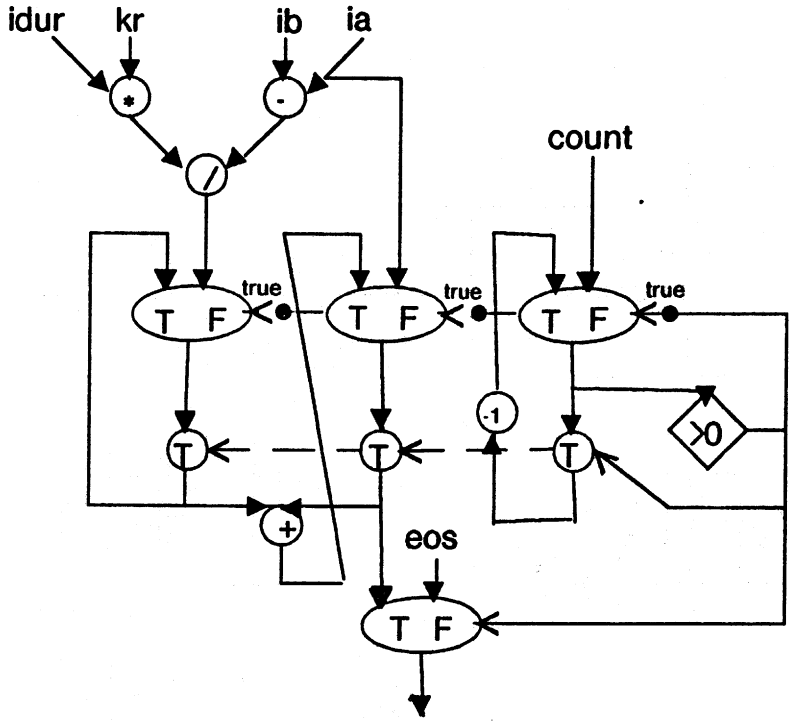
Figs. 5.11(b) and 5.12 depict the flow graph and instruction cell translations of segment. The function segment outputs either a control or audio rate signal and therefore line must be maximally pipelined. Fig 5.12 meets this requirement. The three output lines of loop-control originate from the same instruction cell. Acknowledge signals are left out of the figure to reduce clutter but it is important to note the acknowledge signals of several instruction cells so that the reader is convinced that Fig. 5.12 works correctly. Instruction cell 3 acknowledges cell 1 when its third input is *false*. It cannot fire until it has received an acknowledgement from the add instruction cells. Instruction cell 4



DATA FLOW GRAPH FOR THE FUNCTION KLINE

(a)

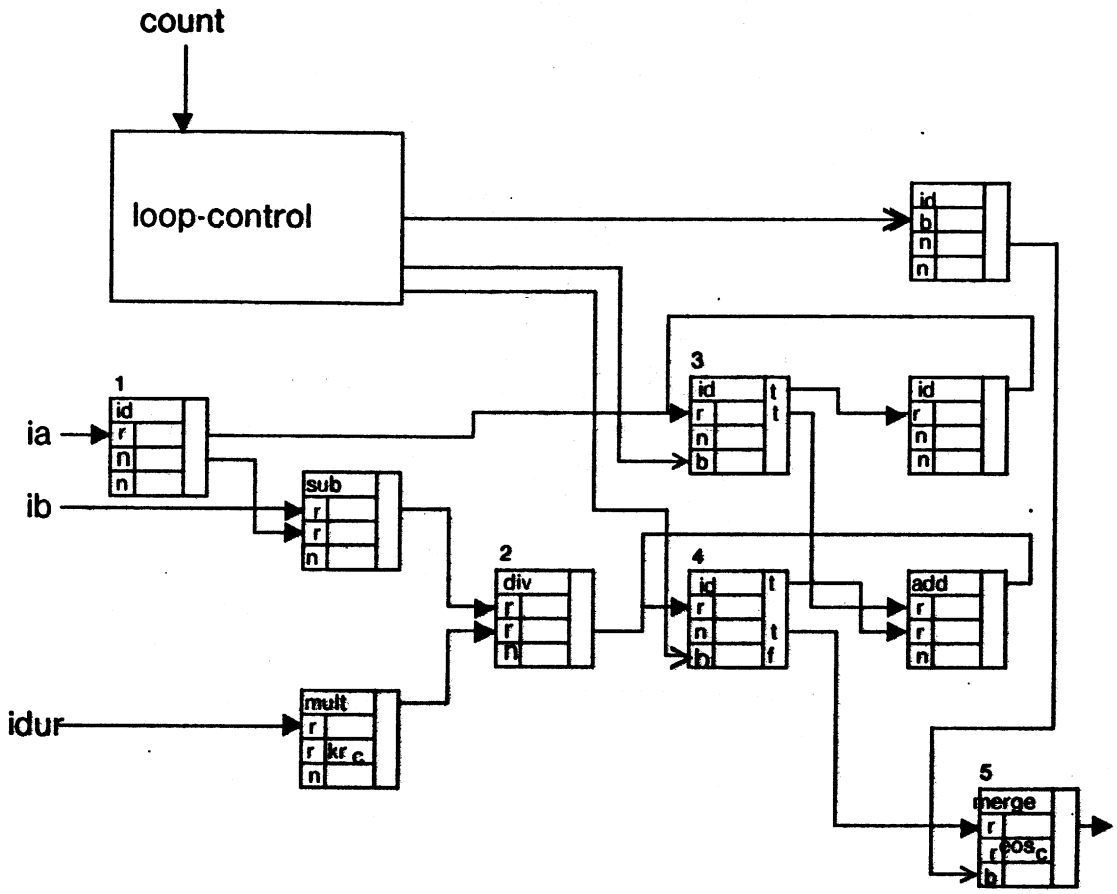
Fig. 5.11



DATA FLOW GRAPH OF THE FUNCTION SEGMENT

(b)

Fig. 5.11



INSTRUCTION CELL TRANSLATION OF SEGMENT

Fig. 5.12

acknowledges cell 2 when its third operand is *false*. It cannot fire until it has received an acknowledgement from instruction cell 5. Both cells 3 and 4 acknowledge the output cell of loop-control. Note that cells 1, 2, and the sub and mult instruction cells do not need to be optimally pipelined since they fire only once per note.

5.4.2 Oscillators

The oscillators make up the second class of signal generators. They output periodic waveforms and are essential to music synthesis. Without oscillators, there can be no sense of pitch to a sound. The oscillators in Music-11 are **table, phsor, oscil** and **foscil**. The function **table** is not really an oscillator, but is included in this group because when paired with **phsor**, the two form an oscillator. **Table, phsor** and **oscil** are discussed in the following paragraphs.

5.4.2.1 The table functions

A function statement in Music-11 using **table** looks like any of the following three statements,

in **table** *indx, ifn, ixoff*
kn **table** *kndx, ifn, ixoff*
an **table** *xndx, ifn, ixoff*

The **table** functions simply access the function table *ifn* using a real number equal to the sum of the *ndx* and *ixoff* inputs to linearly interpolate between two entries.

In Music-11, **table** can output a note rate signal (the only signal function to do so), a control rate signal or an audio rate signal. A separate function for each rate is written in VAL. Functions **itable** and **ktable** are shown in Fig. 5.13.

The output of **itable** is note rate so it is of type **stream[real]**. This output stream contains one member for each note in the score file. Likewise, the output stream of **ktable** contains one member for each note in the score file, but its members are control rate streams. Both **itable** and **ktable** use the function interp (Fig. 5.14(a)) to access the function table. The *index* input of interp is a real

```
function itable(indx, ixoff :stream[real]; ifn : stream[array[real]] returns stream[real]);

for
        indcx : stream[real] := ndx;
        xoffset : stream[real] := ixoff;
        ftable : stream[array[real]] := ifn;
        oustream : stream[real] := [ ];
do
        if index = [ ] then outstream
        else iter
        oustream := consl(outstream,interp(first(ftable),first(ndx)+first(offset));
        xoffset := rest(offset);
        ftable := rest(ftable);
        index := rest(index);
        enditer
        endif
endfor
endfun

function ktable(ndx : stream[stream[real]]; ifn : stream[array[real]]; ixoff : stream[real]
                                                                    returns stream[stream[real]]);

for
        index : stream[stream[real]] := ndx;
        ftable : stream[stream[real]] := ifn;
        xoffset : stream[real] := xoff;
        outstreams : stream[stream[real]] := [ ];
do
        if index = [ ] then outstreams
        else iter
        oustreams := consl(outstreams,for
                        index2 : stream[real] := first(index);
                        addstream : stream[real] := [ ];
                do
                        if index2 = [ ] then addstream
                        else iter
                        addstream :=
                            consl(addstream,interp(first(ftable),first(index2)+first(xoffset));
                        index2 := rest(index2);
                        enditer
                        endif
                endfor
        xoffset := rest(xoffset);
        ftable := rest(ftable);
        index := rest(index);
        enditer
endfor
endfun
```

VAL FUNCTIONS ITABLE AND KTABLE
Fig. 5.13

number *int.frac*. Interp uses *frac* to linearly interpolate between entries *int* and *int+1*. The function **atable** is not depicted but its VAL code and hence its flow graph and cell translations are exactly the same as **ktable**. However **atable** expects its input stream *ndx* to consist of audio rate streams whereas the *ndx* input to **ktable** must be made up of control rate streams. Even though the VAL code for both is the same the user must specify his wishes by writing either **ktable** or **atable**.

The data flow graph for **ktable** (or **atable**) is shown in Fig. 5.15. Since interp has a depth of six actors (Fig. 5.14(b), the dotted lines beside the interp box must each contain six identity operators to insure a maximally pipelined graph.

5.4.2.2 The phsor functions

In Music-11 the phsor functions take the form,

kn **phsor** *kcps, iphs*
an **phsor** *xcps, iphs*

The phsor functions output a moving value *phase* where $0 <= phase < 1.0$. This moving value accumulates an increment dependent on the *cps* input. The input *iphs* is a note rate stream and determines the initial phase value for each note. If *iphs=0* then the initial phase equals 0, otherwise the initial phase is the value that was last calculated for the previous note. As an example, if the following statement is included in an orchestra file,

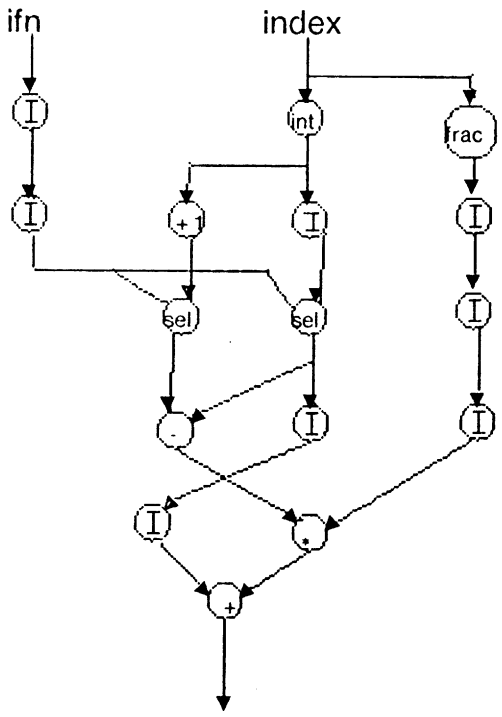
```
;              kcps    iphs
k1     phsor   100,     0
```

the output signal k1 will cycle from 0 to 1 with a frequency of 100 cps. Its initial phase will always be 0. Since k1 is a control rate signal the accumulated increment will be *100/kr*. In general the increment is equal to *kcps/kr* (*xcps/sr* for an audio rate output).

Fig. 5.16 shows the VAL function **kphsor**. Like all other functions that produce control rate signals, its output is of type **stream[stream[real]]**. The function **rmod**(*x,y*) is assumed to be a built in operation on the data flow machine. It carries out a real (as opposed to integer) mod operation. It

```
function interp(ftable : array[real]; index : real returns real);

let
        whole : integer := int(index);
        fraction : real := frac(index);
in
        ftable[whole]+(ftable[whole+1]-ftable[whole])*fraction
endlet

endfun;
```

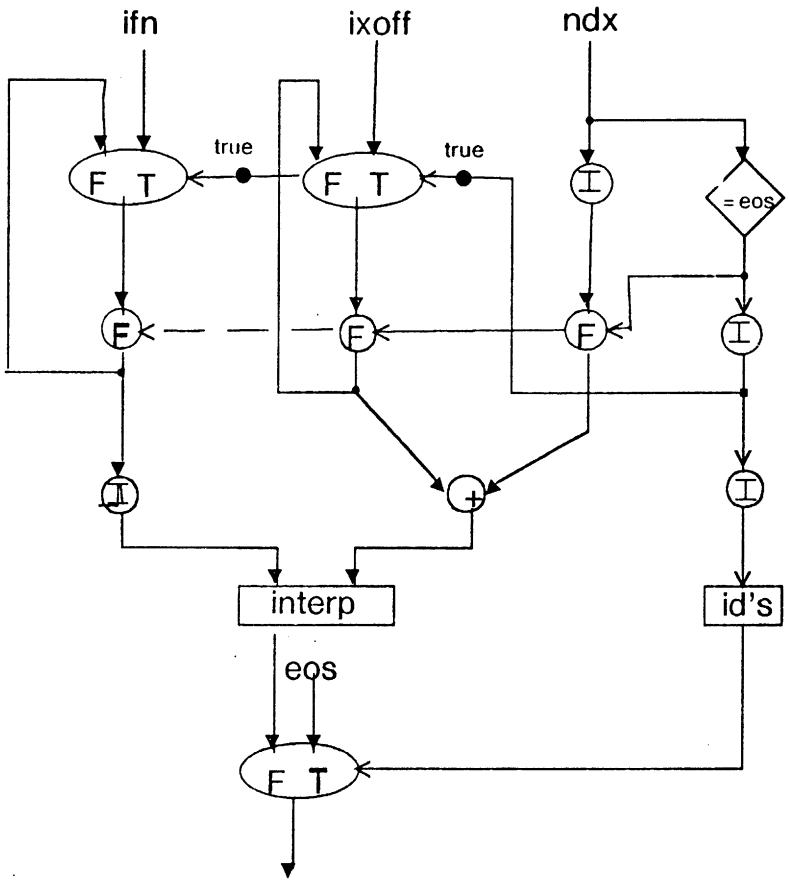
VAL FUNCTION INTERP
(a)



DATA FLOW GRAPH FOR THE FUNCTION INTERP

(b)

Fig. 5.14



DATA FLOW GRAPH FOR THE FUNCTION KTABLE

Fig. 5.15

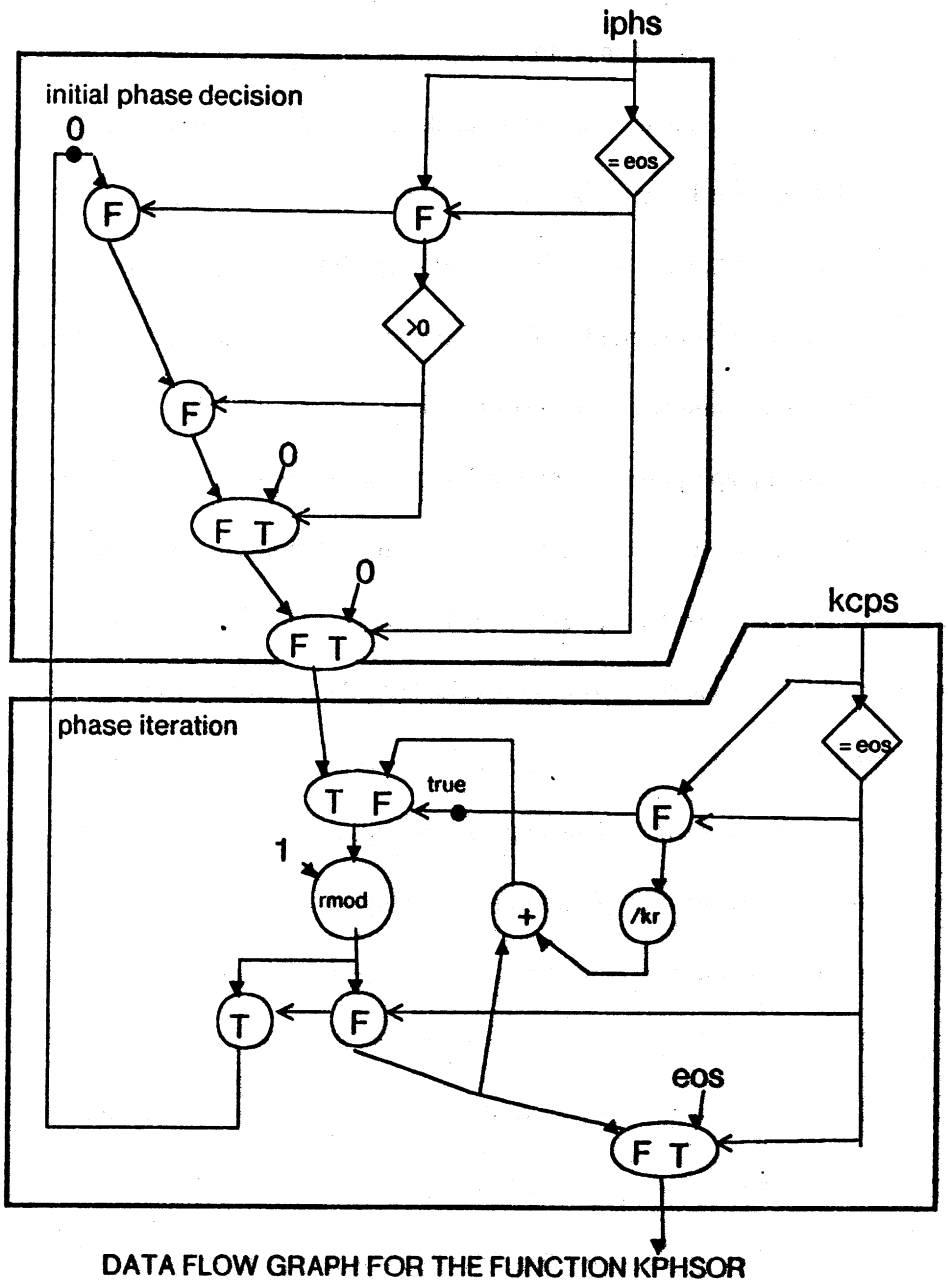
```
function kphsor(kcps : stream[stream[real]] ; iphs : stream[real]
                                                    returns stream[stream[real]]);

for
        cps : stream[stream[real]] := kcps;
        phase : stream[real] := iphs;
        outstreams : stream[stream[real]] = [ ];
        phs : real := 0.0;
        addstream : stream[real] := [ ];
do
        if amp = [ ] then outstreams
        else iter
        addstream,phs :=
                for
                        cps2 : stream[real] := rmod(first(cps));
                        phs2: real := if first(iphs) >= 0.0
                                                    then first(iphs)
                                                    else phs
                                            endif;
                        addstream2 : stream[real] := [ ];
                do
                        if cps2 = [ ] then addstream2,phs2
                        else iter
                        addstream := consl(addstream,phs);
                        phs := rmod(phs + first(cps2)/kr,1.0);
                        cps2: = rest(cps2);
                        enditer
                        endif
                endfor
        outstreams := consl(outstreams,addstream);
        cps := rest(cps);
        phs := rest(phs);
        enditer
        endif
endfor

endfun;
```

VAL FUNCTION KPHSOR
Fig. 5.16

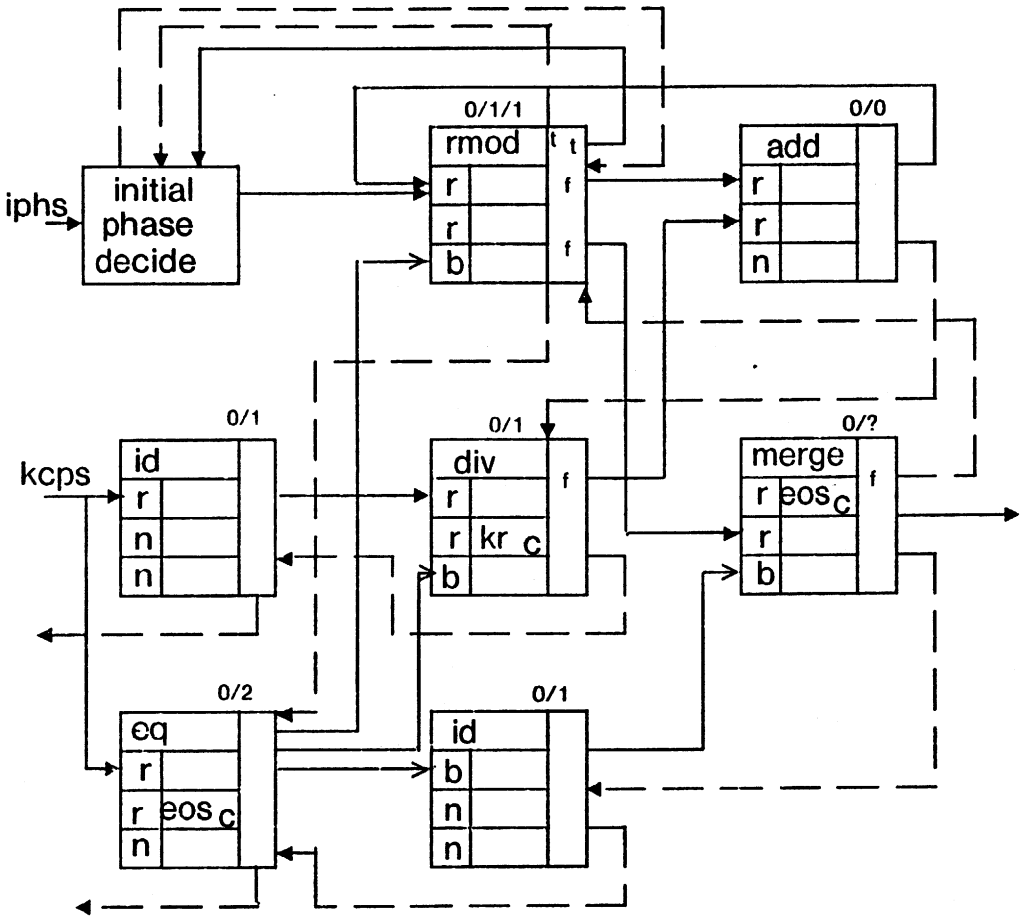
handles negative values of x in the following manner. **rmod**(*-.3,1.0*)=*.7* not *.3*.

The flow graph for **kphsor** is depicted in Fig. 5.17. The initial phase value of zero is present on the top left F-gate. For each stream in *kcps*, the phase iteration section of the graph produces a stream of equal size. At the end of each stream $kcps_n$, an *eos* token will cause the $=eos$ condition to be true. The last computed phase value does not drop through the bottom F-gate into the merge actor but passes through the T-gate to be saved for the next iteration. If $iphs_{n+1} < 0$ then the saved phase values will be the first phase value of the iteration involving $kcps_{n+1}$. An *eos* token will follow the last *iphs* member through the graph. When it becomes the input to the "$=eos$" actor, the actor will consume the *eos* token and place a value of *true* on the top T-gate. This will allow a zero to drop into the phase iteration graph portion. The **rmod** actor will fire with the zero token as an input and output a zero token on the input arcs of the F- and T- gates. A final *eos* token from the *kcps* stream of streams will cause a *true* token to appear on the bottom F- and T- gates. Thus the zero token will pass through the T-gate and be correctly positioned to leave the graph in in its initial state.

The **kphsor** flow graph is not maximally pipelined but it can be translated into a cell graph that is, almost (Fig. 5.18). The box labelled initial phase decide is a direct actor to cell translation from the flow graph. The phase iteration graph portion can easily be translated into a cell graph that can output tokens at the optimum rate However at the end of every iteration cycle when the last phase value is saved (the last phase value is the output of the rmod instruction cell when its third input is *true*), seven cells must fire in sequence (five of them in the initial phase decide subgraph) before the phase iteration graph can produce an output token. But this firing sequence occurs only once for every note in the score file. With a sampling rate of 50,000 a note would have to be less than .001 seconds before the output rate would be seriously impaired.



DATA FLOW GRAPH FOR THE FUNCTION KPHSOR
Fig. 5.17



INSTRUCTION CELL TRANSLATION OF KPHSOR

Fig. 5.18

5.4.2.3 The oscil functions

In Music-11, the oscil function takes the form

kn **oscil** *kamp, kcps, ifn, iphs*

when outputting a control rate signal and,

an **oscil** *xamp, xcps, ifn, iphs*

when it produces an audio rate signal. As explained in chapter one, **oscil** accesses samples of a periodic wave stored in a function table *ifn* at a frequency of *kcps* (or *xcps*). These samples are multiplied by the *amp* input to produce the output. An example will aid the reader in understanding the algorithm used in obtaining the **oscil** output.

Consider the statement,

```
;                    kamp   kcps   ifn    iphs
k1          koscil   10000, 440,   1,     -1
```

If one cycle of a sine wave is stored in function table 1, then **oscil** becomes a sinusoidal oscillator with frequency of 440 and amplitude of 10000. As explained before, **oscil** steps through the function table. But how is it determined which entry to access in function table 1? It is known that all function tables are of type **array[real]** and have limits ifn[0 : 511]. If the frequency of the signal is 440 cycles/sec, the period is 1/440 sec/cycle. k1 is a control rate signal so there are *kr* samples/sec and *kr*/440 samples/cycle. The table contains 512 entries/cycle, so the sampling increment should be (512*440)/*kr* entries/sample or in general (512*cps)/*kr* entries/sample. In order that an illegal array access is not made, the accumulated increment *x* should not be used to read the function table but **rmod**(*x*,512).

The VAL function **koscil** calculates its array index using a method similar to the one outlined in the previous paragraph. The accumulated increment *x* is not **rmod**(512*cps/*kr*,512) but **rmod**(cps/*kr*,1.0). The function table index is obtained by multiplying *x* by 512. The calculated index is a real number and as in the **table** functions, is used to linearly interpolate between two

successive entries in the function table. The *iphs* input determines the initial entry to be accessed in *ifn* for each note. If *iphs*<=0 then the initial entry is zero. Otherwise the initial entry is 512**iphs*. The function **koscil** can be defined as follows:

```
function koscil(kamp,kcps:stream[stream[real]]; ifn:stream[array[real]];
                                    iphs:stream[real]] returns stream[stream[real]]);
        let
                phasesig:stream[stream[real]]:=kphsor(kcps,iphs);
                oscilsig:stream[stream[real]]:=ktable(phasesig*512,ifn);
        in
                kamp*oscilsig
        endlet
endfun
```

The function **kphsor** is used to calculate x and **ktable** uses $512*x$ to access *ifn*. Since **koscil** is composed of two functions that have already been defined, its graph will not be detailed.

5.4.3 Signal modifiers

The third class of signal functions is the signal modifiers. The majority of these functions is a digital filter of some sort. The simplest of these filters is **tone.**

The Music-11 format of **tone is,**

an **tone** *asig, khp, istor*

The function **tone** performs a low-pass filter operation on the input audio rate signal *an.* The half-power frequency of the filter is *khp.* The algorithm used is the filter equation, y[n]=b0*x[n]+a1*y[n-1], where x[n] is the input corresponding to *asig* and b0 and a1 are filter coefficients calculated from the half-power frequency by the function tone_coeff (Fig. 5.19(b)). The input *istor* determines the initial value of y[n-1] for each note. If *istor* = 0 the inital value of y[n-1] at the start of the next note is zero. If *istor*$\neq$ 0 the inital value of y[n-1] is the last value of y[n] from the previous note.

Fig. 5.19(c) shows the VAL function **tone. Tone** calls the function convert_coeff (Fig. 5.19(a))

```
function convert_coeff(khp : stream[stream[real]] returns stream[stream[real]]);

for
        hp : stream[stream[real]] := khp;
        bo : stream[stream[real]] := [ ];
        al : stream[stream[real]] := [ ];
        sbo: stream[real] := [ ];
        sal: stream[real] := [ ];
do
        if hp = [ ] then b0,al
        else iter
        sb0,sal := for
                        sb0 : stream[real] := [ ];
                        sal : stream[real] := [ ];
                        hp2 : real := first(hp);
                        mb0 : real;
                        ma1 : real;
                do
                        if hp = [ ] then sb0,sal
                        else iter
                        mb0,ma1 := tone_coeff(first(hp2));
                        sb0 := consl(sb0,mb0);
                        sal := consl(sal,ma1);
                        hp2 := rest(hp2);
                        enditer
                        endif
                endfor
        b0 := consl(b0,sb0);
        al := consl(a2,sa1);
        hp := rest(hp);
        enditer
        endif
endfor

endfun
```

VAL FUNCTION CONVERT_COEFF

(a)

```
function tone_coeff(hp : real returns real);

let
        b : real := 2.0-cos(2*pi*first(hp)/sr);
        al : real := if hp > 0.0 then b-sqrt(b*b-1) else -b+sqrt(b*b-1) endif;
in
        al,1-al
endlet

endfun
```

VAL FUNCTION TONE_COEFF

(b)

Fig. 5.19

```
function tone(asig,khp : stream[stream[real]] ; istor : stream[real]
                                                    returns stream[stream[real]);

for
        x,y : stream[stream[real]] := convert_coeff(khp);
        b0: stream[stream[real]] :=control_to_audio(x);
        a1: stream[stream[real]] :=control_to_audio(y);
        sig : stream[stream[real]] := asig;
        stor : stream[real] := istor;
        result : real := 0.0;
        outstreams : stream[stream[real]] := [ ];
        addoutstream : stream[real] := [ ];
do
        if sig = [ ] then outstreams
        else iter
        addoutstream,result := for
                        b0 : stream[real] := first(b0);
                        a1 : stream[real] := first(a1);
                        sig : stream[real] :=first(sig);
                        outstream : stream[real] := [ ];
                        result : real := if first(istor) = 0.0
                                                then 0.0
                                                else result
                                        endif;
                do
                        if sig = [ ] then outstream
                        else iter
                        result := first(b0)*first(sig)+first(a1)*result;
                        outstream := consl(outstream,result);
                        b0:=rest(b0);
                        a1:=rest(a1);
                        sig:=rest(sig);
                        enditer
                                endif
                endfor
        outstream := consl(oustreams,addoutstream);
        b0:=rest(b0);
        a1:=rest(a1);
        sig:=rest(sig);
        stor:=rest(stor);
        enditer
        endif
endfor

endfun;
```

VAL FUNCTION TONE
(c)

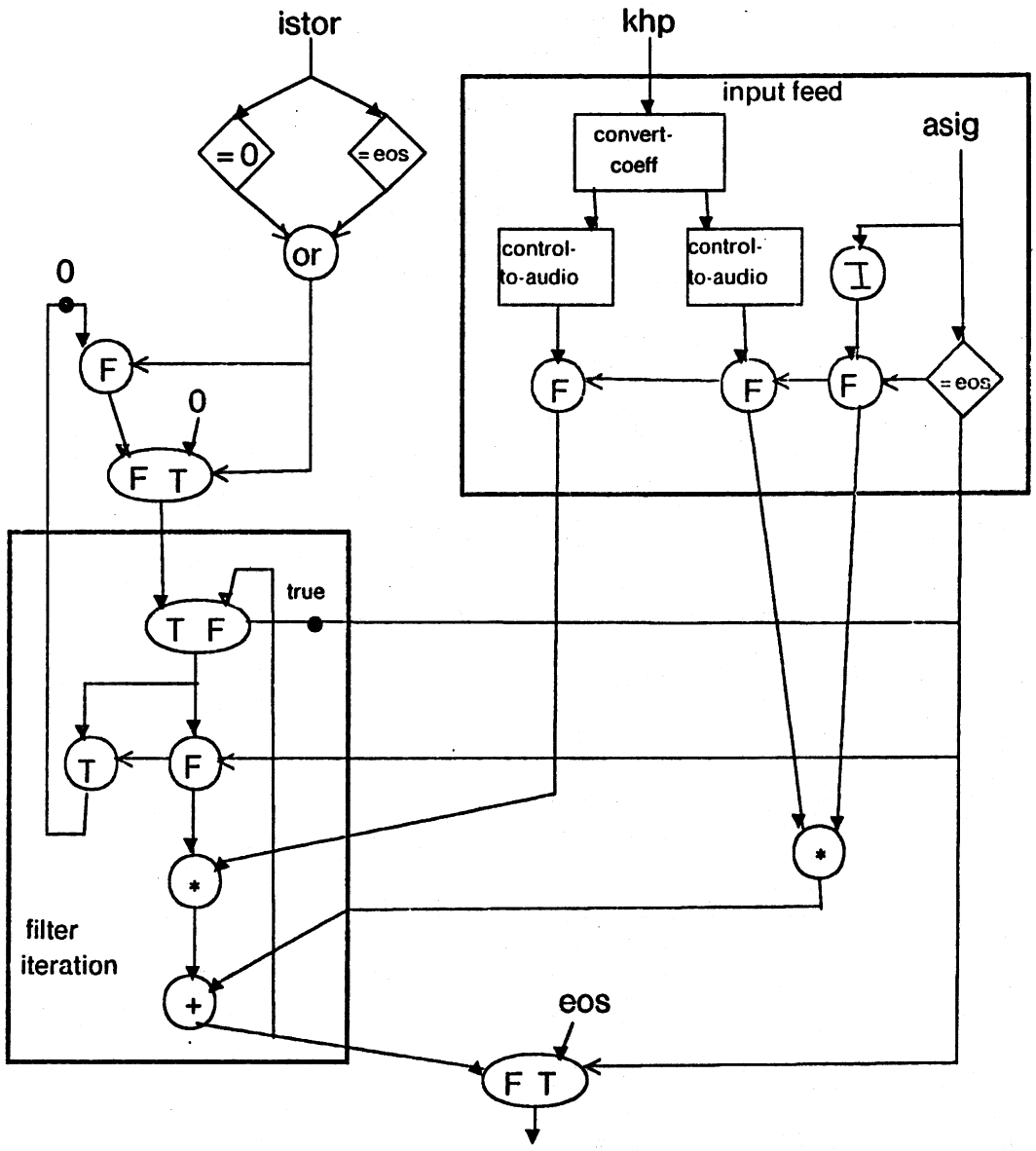
Fig. 5.19

to convert the control rate stream of khp to two control rate streams of filter coefficients. The two streams are converted to audio rate streams by the function **control_to_audio**.

The flow graph for **tone** is shown in Fig. 5.20. In order to give the reader a full understanding, a brief example is presented. Let $asig_0 = x$, $b0_0 = c$, $a1_0 = d$ and $istor_0 = 1$. x, c and d are all streams and correspond to the first note in the score. Furthermore, let m be the length of streams x, c and d. Because $istor_0 = 1$, the initial zero token drops through the F-gate and merge actor into the filter iteration subgraph. The initial value of *true* on the input of the merge actor allows the value zero to rest on the input arc of the F- and T-gates. The condition $x_0 = eos$ is false so the filter subgraph outputs $y[0] = x_0 * c_0$. The value of $y[0]$ is fed back through the merge actor in the filter iteration subgraph where it drops through to the F- and T-gates. If $m>1$, then $y[0]$ becomes the input to the multiply operator and the filter iteration subgraph outputs $y[1] = x_1 * c_1 + y[0] * d_1$. When $y[m-1]$ is calculated, it becomes the output of the filter iteration subgraph and is fed back to the merge actor where drops through to the two gates. This time, the $=eos$ condition for x is true. Thus $y[m-1]$ is swallowed by the F-gate and falls through the T-gate. The value of $istor_1$ determines whether $y[m-1]$ will remain the same or be assigned the value zero when computing $y[m]$.

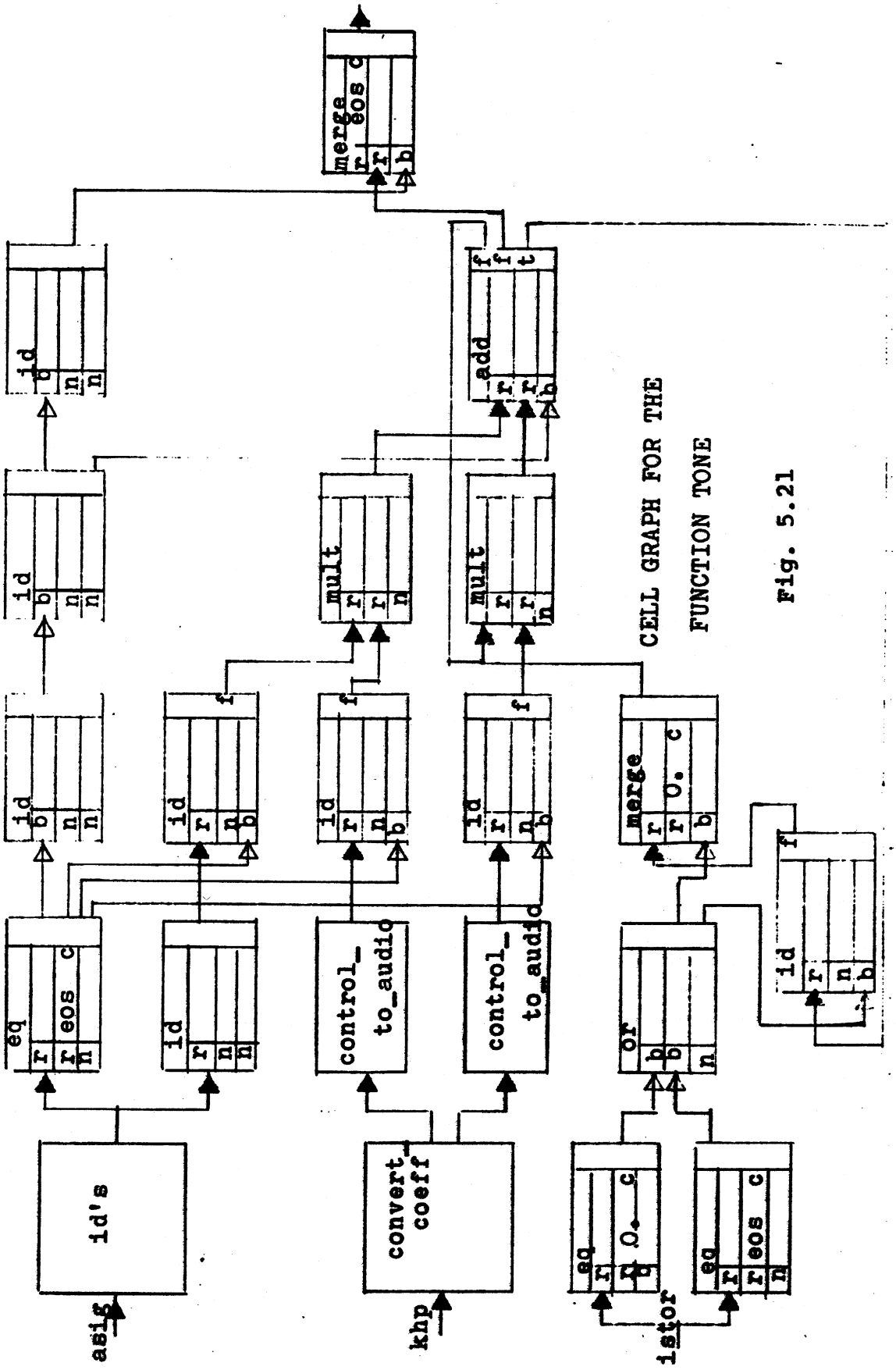
When the last note has been played, the *eos* token of the *istor* stream causes the last calculated output to be consumed by the F-gate and a zero value to drop through the merge actor of the filter iteration subgraph. The final *eos* token of the *asig* stream of streams produces a *true* token on the F- and T-gates. Therefore the zero token passes through the T-gate and the graph is restored to its original state.

The cell graph for **tone** is shown in Fig. 5.21. The filter iteration subgraph can be translated into an instruction cell graph that will output values at an optimum rate between consecutive *false* inputs. When the boolean input is *true* (when the $=eos$ condition for asig is true), a sequence of seven cells must fire before the next output is produced. However, this occurs only at the end of each note and does not seriously impede the output rate. The box labelled *id cells*, contains a sequence of



DATA FLOW GRAPH FOR THE FUNCTION TONE

Fig. 5.20



CELL GRAPH FOR THE FUNCTION TONE

Fig. 5.21

m identity instruction cells where *m* is large enough to insure that the graph is pipelined.

5.5 Function Tables

In Music-11 the user can create arrays and use them to store evenly spaced samples of waveforms. This is done by including a function statement in the score file. (The reader is referred to chapter one for a detailed description of the function statement.) In Music-df, the same capability is offered. However, instead of assigning the function generators a number that becomes *p4* in the function statement, a meaningful name is given to each different waveform option. For instance function generator # 10 in Music-11 becomes **sinetable** in Music-df.

All function tables are of type **array[real]** and contain 513 entries. This size is considered large enough, particularly since all the signal functions that access function tables perform an interpolation. The last entry is a copy of the first entry so that an illegal array access cannot be made when a signal function interpolates. The array limits are **[0 : 512]**.

A representative function generator, **sinetable** is detailed in this section. The algorithm for **sinetable** is expressed first in a VAL function and then a data flow graph. Since the instruction cell coding is a direct translation of the flow graph, it is not included.

5.6 Sinusoidal Sum Function Table

The Music-11 function generator # 10 was discussed in chapter one. It computes a sum of sinusoids and stores them in a function table. In Music-11, a statement containing function generator # 10 looks like,

f *p1 p2 p3 10 str1, str2, str3,...*

where

str1,str2,str3... - the strength of the first, second, third etc.

partials

Function generator #10 in Music-11 becomes the function **sinetable** in Music-df (Fig. 5.22(a)). It expects its input in a different format than that of Music-11. The first input *strength* is a stream of the strengths of the partials. The function **sinetable** calls sinsesum (Fig. 5.22(b)) to construct an unscaled array and give the maximum value contained by the array. The function **sinetable** then uses the scale input to determine whether or not to scale the sinesum's output array to a peak value of 1. A Music-df function statement using **sinetable** has the form,

functiontable sinetable *[str1,str2,str3...],scale*

The flow graphs for **sinetable** and sinesum are shown in Fig. 5.23. The function sinesum calculates the 512 points of the waveform using the equation,

function_table[i]: =

str1*sin(2π/512/1*real(i))+

str2*sin(2π/512/2*real(i))+...

strn*sin(2π/512/n*real(i))

Within the calculation of the contents of each table entry the VAL function sinesum computes the contribution of each partial in sequence. It is hoped that a compiler would be smart enough to see the potential for parallelism that is not expressed in the VAL function. Consider the case where sinesum is called with only one partial. Fig. 5.23(b) depicts the flow graph for this case. If the number of partials equals *n* then the parallelized flow graph would be as shown in Fig. 5.23(c) where the number of addition operators needed would be *n-1*.

```
function sinetable(strength : stream[real] ; scale : real returns array[real]);

let
        functiontable : array[real], scalefactor : real := sinesum(strength);
in
        if scale > 0
        then functiontable
        else
                for
                        functiontable : array[rea] := functiontable
                        index : integer := 0;
                do
                        if index > 512 then  functiontable
                        else iter
                        functiontable[index] := functiontable[index]/scalefactor;
                        index := index + 1;
                        enditer
                        endif
                endfor
        endif
endlet

endfun
```

VAL FUNCTION SINETABLE

(a)

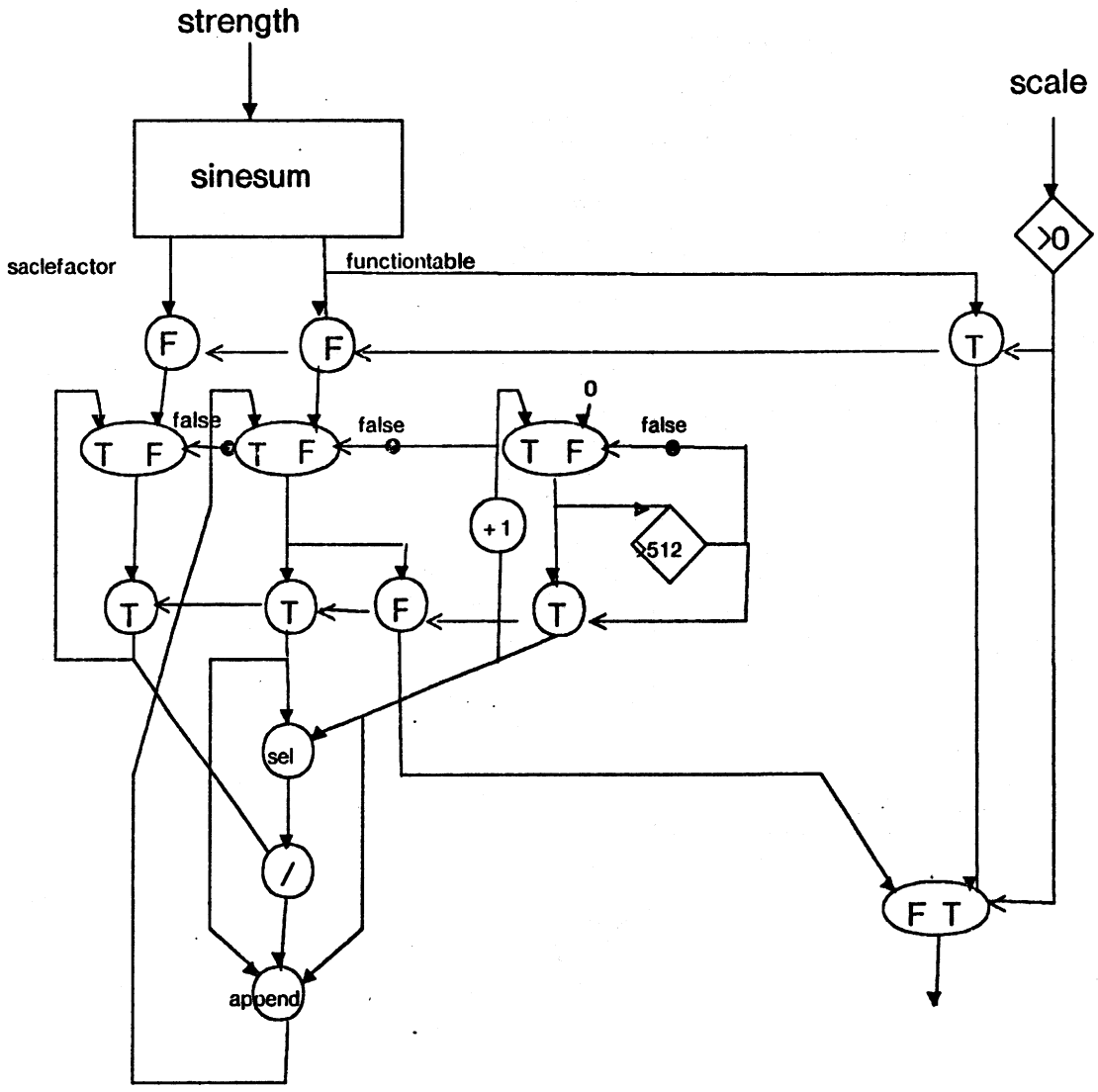
Fig. 5.22

```
function sinesum(strength :stream[real] returns array[real]);

let
functiontable : array[real] , max : real :=
        for
                addentry :real;
                scalefactor : real := 0;
                count : integer := 0;
                functiontable : array[real] := array_fill(0,-1,0);
        do
                if count = 512 then functiontable, scalefactor
                else iter
                addentry :=
                        for
                                strength : stream[real] := strength;
                                x : real :=0;
                                partial :real := 1;
                        do
                                if strength = [ ] then x
                                else iter
                                x := x+ first(strength)*sin(2*pi/512/partial*real(count));
                                strength := rest(strength);
                                partial := partial + 1;
                                enditer
                                endif
                        endfor
                count := count + 1
                functiontable: = array_add_hi(functiontable,addentry)
                scalefactor := max(scalefactor,addentry);
                enditer
                endif
        endfor
in
        array_add_hi(functiontable,functiontable[0]),scalefactor

endlet
```

VAL FUNCTION SINESUM

(b)

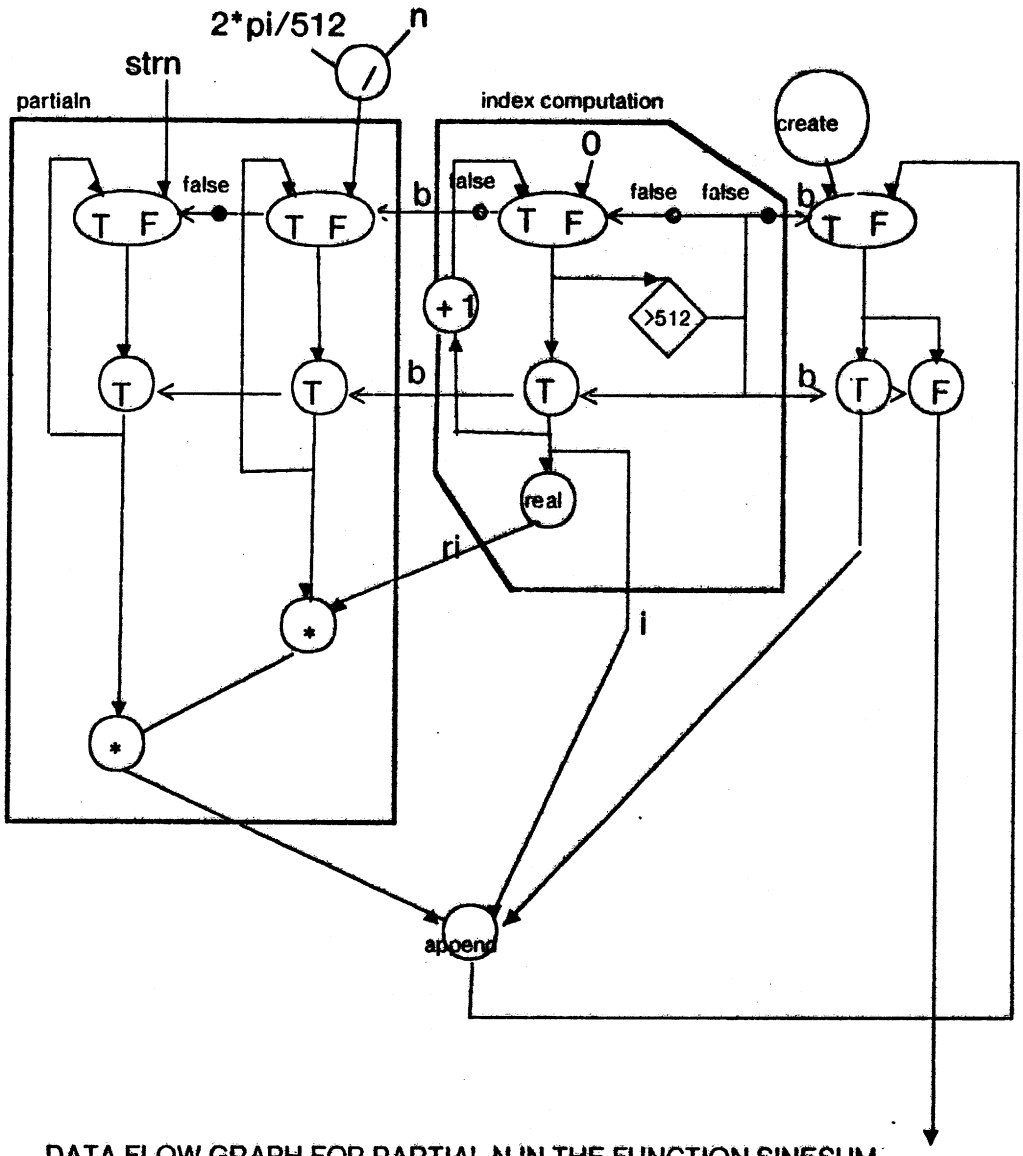
Fig. 5.22



DATA FLOW GRAPH FOR THE FUNCTION SINETABLE

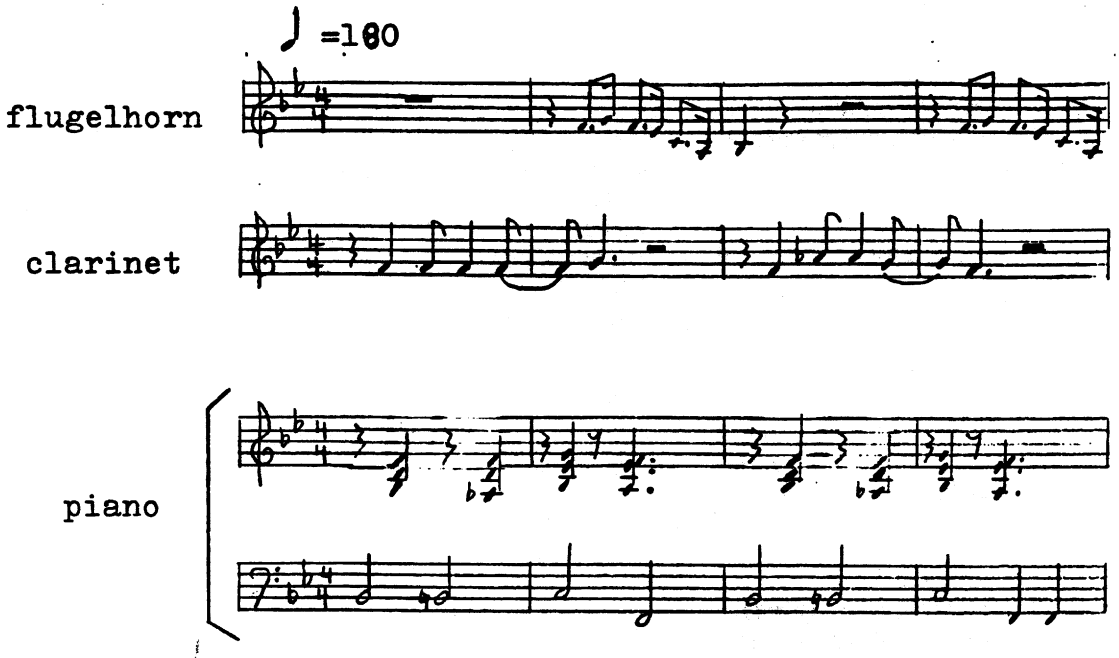
(a)

Fig. 5.23



DATA FLOW GRAPH FOR PARTIAL N IN THE FUNCTION SINESUM
(b)

Fig. 5.23



CONVENTIONAL SCORE

Fig. 6.1

```
beginorchestra
sr = 40000
kr = 5000
ksmps = 8
nchnls = 1

flugelhorn =
    instr flugelhorn
    notefreq = cpspch(p4)
;                                   kamp                  irise                 idur   idec
    modenvelope     klinen          notefreq*p7,          .06,                  p3,    .02
;                                   xamp                  xcps                  ifn    iphs
    modulator       aoscil          modenvelope,          notefreq*.995,        1,     -1
;                                   kamp                  irise                 idur   idec
    ampenvelope     klinen          1000*p5,              p6,                   p3,    .03
;                                   kamp                  kcps                  ifn    iphs
    tremelo         koscil          p9,                   p8,                   1,     -1
;                                   xamp                  xcps                  ifn    iphs
    flugsound       aoscil          ampenvelope*tremelo,  notefreq+modulator,   1,     -1
                    output          flugsound
    endin

clarinet=
    instr clarinet
    notefreq=cpspch(p4)
    carrierfreq=3*notefreq
    modfreq=2*notefreq
;                                   xamp                  irise                 idur   idec
    modenvelope     alinen          10000,                .2,                   p3,    0
;                                   xamp                  xcps                  ifn    iphs
    modulator       aoscil          modenvelope           modfreq,              1,     0
;                                   xamp                  irise                 idur   idec
    carrierenvelope  alinen         4*carrierfreq         .2                    p3,    .15
;                                   xamp                  xcps                  ifn    iphs
    clarsound       aoscil          carrierenvelope,      carrierfreq+modulator,  1,    0
                    output          clarsound
    endin
```

ORCHESTRA FILE

Fig. 6.2
(1 of 3 pages)

```
piano1=
    instr piano1
    nfreq1=cpspch(p4)                          ;unstreteed pitch
    nfreq2 =nfreq1-(10/nfreq1)                 ;stretched pitch for high notes
    nfreq3=nfreq1+(nfreq1/200)                 ;stretched pitch for low notes
    if nfreq1<196
        then  nfreq=nfreq2
        else  if  nfreq1<784
            then  nfreq=nfreq1
            else  nfreq=nfreq3
        endif
    endif
    mod1amp=(nfreq*(8-log(nfreq)))/(log(nfreq*(log(nfreq))
    mod2amp=(20*(8-log(nfreq)))/nfreq
    mod1freq=nfreq+(nfreq/200)
    mod2freq=(nfreq*4)+(nfreq/200)
;                       ia  idur1  ib  idur2  ic  idur3  id  idur4  ie  idur5  if
    ampenv    klinseg   1,  p3*.05  .6,  p3*.05,  .2,  p3*.15,  .15,  p3*.25,  .07,  p3*.5,  0
;                       xamp                    xcps                    ifn  iphs
    mod1      aoscil    mod1amp*mod1freq,       mod1freq,               1,   0
    mod2      aoscil    mod2amp*mod2freq,       mod2freq,               1,   0
    string1   aoscil    ampenv,                 nfreq+mod1+mod2,        1,   0
    string2   aoscil    ampenv,                 nfreq+mod1+mod2+.007,   1,   0
    string3   aoscil    ampenv,                 nfreq+mod1+mod2-.007    1,   0
              output    (string1+string2+string3)*7000
    endin

piano2=
    instr piano2
    nfreq1=cpspch(p4)                          ;unstretched pitch
    nfreq2 =nfreq1-(10/nfreq1)                 ;stretched pitch for high notes
    nfreq3=nfreq1+(nfreq1/200)                 ;stretched pitch for low notes
    if nfreq1<196
        then  nfreq=nfreq2
        else  if  nfreq1<784
            then  nfreq=nfreq1
            else  nfreq=nfreq3
        endif
    endif
    mod1amp=(nfreq*(8-log(nfreq)))/(log(nfreq)*(log(nfreq))
    mod2amp=(20*(8-log(nfreq)))/nfreq
    mod1freq=nfreq+(nfreq/200)
    mod2freq=(nfreq*4)+(nfreq/200)
;                       ia  idur1  ib  idur2  ic  idur3  id  idur4  ie  idur5  if
    ampenv    klinseg   1,  p3*.05  .6,  p3*.05,  .2,  p3*.15,  .15,  p3*.25,  .07,  p3*.5,  0
;                       xamp                    xcps                    ifn  iphs
    mod1      aoscil    mod1amp*mod1freq,       mod1freq,               1,   0
    mod2      aoscil    mod2amp*mod2freq,       mod2freq,               1,   0
    string1   aoscil    ampenv,                 nfreq+mod1+mod2,        1,   0
    string2   aoscil    ampenv,                 nfreq+mod1+mod2+.007,   1,   0
    string3   aoscil    ampenv,                 nfreq+mod1+mod2-.007    1,   0
              output    (string1+string2+string3)*7000
    endin
```

ORCHESTRA FILE
Fig. 6.2
(2 of 3 pages)

```
piano3=
    instr piano3
    pitch1=cpspch(p4)
    pitch2 =pitch1-(10/pitch1)
    pitch3=pitch1+(pitch1/200)
    if pitch1<196
        then  pitch=pitch2
        else  if  pitch1<784
            then   pitch=pitch1
            else   pitch=pitch3
        endif
    endif
    mod1amp=(pitch*(8-log(pitch)))/(log(pitch)*(log(pitch))
    mod2amp=(20*(8-log(pitch)))/pitch
    mod1freq=pitch+(pitch/200)
    mod2freq=(pitch*4)+(pitch/200)
;                       ia  idur1   ib  idur2   ic  idur3   id  idur4   ie  idur5   if
    ampenv      klinseg     1,  p3*.05  .6,  p3*.05,  .2,  p3*.15,  .15,  p3*.25,  .07,  p3*.5,  0
;                       xamp                xcps                    ifn  iphs
    mod1        aoscil      mod1amp*mod1freq,   mod1freq,           1,   0
    mod2        aoscil      mod2amp*mod2freq,   mod2freq,           1,   0
    string1     aoscil      ampenv,             pitch+mod1+mod2,    1,   0
    string2     aoscil      ampenv,             pitch+mod1+mod2+.007,  1,  0
    string3     aoscil      ampenv,             pitch+mod1+mod2-.007  1,  0
                output      (string1+string2+string3)*700
    endin
pianobass=
    instr pianobass
    pitch1=cpspch(p4)
    pitch2 =pitch1-(10/pitch1)
    pitch3=pitch1+(pitch1/200)
    if pitch1<196
        then  pitch=pitch2
        else  if  pitch1<784
            then   pitch=pitch1
            else   pitch=pitch3
        endif
    endif
    mod1amp=(pitch*(8-log(pitch)))/(log(pitch)*(log(pitch))
    mod2amp=(20*(8-log(pitch)))/pitch
    mod1freq=pitch+(pitch/200)
    mod2freq=(pitch*4)+(pitch/200)
;                       ia  idur1   ib  idur2   ic  idur3   id  idur4   ie  idur5   if
    ampenv      klinseg     1,  p3*.05  .6,  p3*.05,  .2,  p3*.15,  .15,  p3*.25,  .07,  p3*.5,  0
;                       xamp                xcps                    ifn  iphs
    mod1        aoscil      mod1amp*mod1freq,   mod1freq,           1,   0
    mod2        aoscil      mod2amp*mod2freq,   mod2freq,           1,   0
    string1     aoscil      ampenv,             pitch+mod1+mod2,    1,   0
    string2     aoscil      ampenv,             pitch+mod1+mod2+.007,  1,  0
    string3     aoscil      ampenv,             pitch+mod1+mod2-.007  1,  0
                output      (string1+string2+string3)*500
    endin
out flugelhorn+clarinet+piano1+piano2+piano3+pianobass
endorchestra
```

ORCHESTRA FILE
Fig. 6.2
(3 of 3 pages)

above middle C. *int*=8 for any note that lies between high C and middle C. For all other notes, .01 is multiplied by the number of half steps it is away from the nearest C below it and added to the octave pitch value for that C. Therefore the octave pitch value for D above middle C is 8.02. The **cpspch** function is particularly useful when writing score files. No composer wishes to have to look up the cps value that corresponds to every note in his score.

Two other functions that the reader may not be familiar with are **klinen** and **klinseg**. As can be seen from the flugelhorn instrument block, the inputs for **klinen** are *kamp, irise, idur* and *idec*. **Klinen** first computes samples of a curve that rises linearly from 0 to 1 in *irise* seconds. It remains at the value 1 until *idur-idec* seconds into the note, at which time it decays linearly back to zero in *idur* seconds. This curve is then multiplied by *kamp* to produce **klinen**'s output signal.

The function **klinseg** is used in the piano instrument blocks and takes the general form:

outsig **klinseg** *ia, idur1, ib, idur2, ic, idur3...*

Klinseg construct a series of linear segments, the first having endpoints *ia, ib* and lasting for *idur1* seconds, the second having endpoints *ib, ic* and lasting for *idur2* seconds, the third having endpoints *ic, id* and lasting for *idur3* seconds etc. The output of the **klinseg** function in the piano instrument blocks contains five line segments.

All three instrument blocks use the FM synthesis technique to produce their output sound. The flugelhorn contains a tremelo of which the depth and speed are controlled in the score file. The clarinet is a simple FM instrument. The piano uses a doubly modulated signal for its output sound. It also models the fact that three strings are struck whenever a note is played and that one or more of these strings could be out of tune. The string1, string2 and string3 variables represent these three strings. String2 and string3 are slightly off pitch.

With an orchestra file in hand, the appropriate score file that will play the instrument can be written. Fig. 6.3 shows the score file that will play the instruments in the orchestra file according to the score in Fig. 6.1. It is important to note that whereas one usually considers a musical piece to start

```
tempo  0        180
functiontable        sinetable     [1], 1
c flugelhorn score
c    p5 = note amplitude
c    p6 = note amplitude envelope rise time
c    p7 = modulation index
c    p8 = tremelo frequency
c    p9 = tremelo amplitude
c        instr          start   dur     pitch     p5      p6      p7      p8      p9
note     flugelhorn     5       .75     8.05      10      .2      3       6       50
note     flugelhorn     5.75    .25     8.07      9.25    .08     .5      4       20
note     flugelhorn     6       .75     8.05      9       .2      3       6       50
note     flugelhorn     6.75    .25     8.03      8.25    .08     .5      4       20
note     flugelhorn     7       .75     8.00      8       .2      3       6       50
note     flugelhorn     7.75    .25     7.09      7.25    .08     .5      4       20
note     flugelhorn     8       1       7.11      7       .15     3       5       20
note     flugelhorn     13      .75     8.05      10      .2      3       6       50
note     flugelhorn     13.75   .25     8.07      9.25    .08     .5      4       20
note     flugelhorn     14      .75     8.05      9       .2      3       6       50
note     flugelhorn     14.75   .25     8.03      8.25    .08     .5      4       20
note     flugelhorn     15      .75     8.00      8       .15     3       5       20
note     flugelhorn     15.75   .25     7.09      7.25    .08     .5      4       20
note     flugelhorn     16      1       7.11      7       .15     3       5       20

c clarinet score
c        instr          start   dur     pitch
note     clarinet       1       1       8.05
note     clarinet       2       .5      8.05
note     clarinet       2.5     1       8.05
note     clarinet       3.5     1       8.05
note     clarinet       4.5     1.5     8.07
note     clarinet       9       1       8.05
note     clarinet       10      .5      8.08
note     clarinet       10.5    1       8.08
note     clarinet       11.5    1       8.07
note     clarinet       12.5    1.5     8.05
```

SCORE FILE
Fig. 6.3

(1 of 2 pages)

```
c piano scores
c       instr       start   dur     pitch
c right hand
note    piano1      1       1       8.05
note    piano2      1       1       8.02
note    piano3      1       1       7.10
note    piano1      3       1       8.05
note    piano2      3       1       8.02
note    piano3      3       1       7.08
note    piano1      5       1       8.07
note    piano2      5       1       8.03
note    piano3      5       1       7.10
note    piano1      6.5     1.5     8.05
note    piano2      6.5     1.5     8.03
note    piano3      6.5     1.5     7.09
note    piano1      9       1       8.05
note    piano2      9       1       8.02
note    piano3      9       1       7.10
note    piano1      11      1       8.05
note    piano2      11      1       8.02
note    piano3      11      1       7.08
note    piano1      13      1       8.07
note    piano2      13      1       8.03
note    piano3      13      1       7.10
note    piano1      14.5    1.5     8.05
note    piano2      14.5    1.5     8.03
note    piano3      14.5    1.5     7.09
c left hand
note    pianobass   0       2       6.10
note    pianobass   2       2       6.11
note    pianobass   4       2       7.00
note    pianobass   6       2       5.10
note    pianobass   8       2       6.10
note    pianobass   10      2       6.11
note    pianobass   12      2       7.00
note    pianobass   14      1       5.10
note    pianobass   15      1       5.10
end
```

SCORE FILE

Fig. 6.3
(2 of 2 pages)

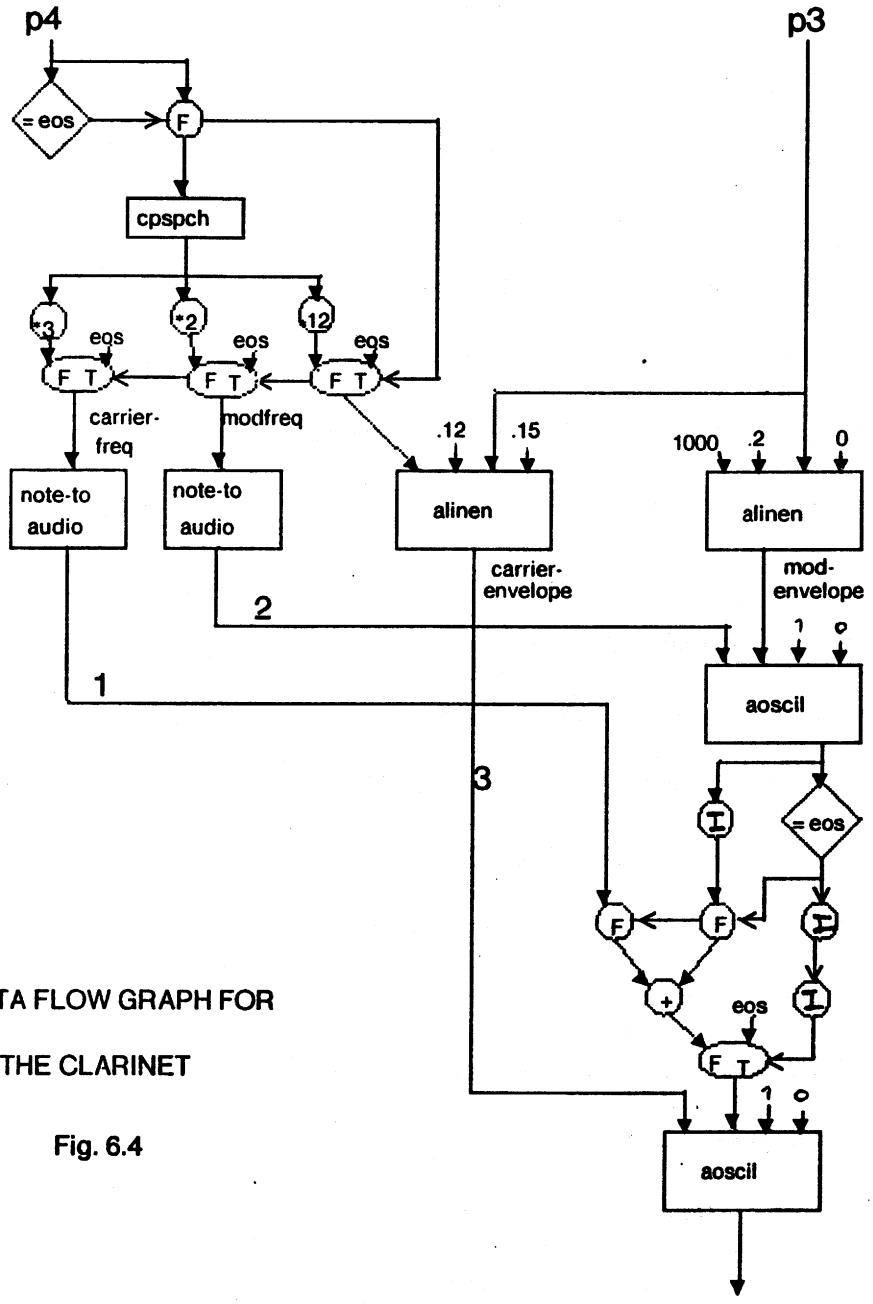
on the first beat, in the notation of a Music-df score a piece starts on the zeroth beat. Thus the clarinet plays its first note on beat 1 in the Music-df score, not beat two.

6.1.2 Performance

With the orchestra and score files written to his satisfaction the musician would invoke a performance program on the host computer of the Music-df system. This program would first send the **functiontable** statement of the score file to the data flow machine where one cycle of a sine wave would be stored in function table no.1. The performance program would then sort all the note statements in the score files in ascending order of *p2*. Then the *p2* and *p3* values would be converted from beat values to second values. The tempo statement in Fig. 6.3 specifies a constant tempo of 180 beats/min. Thus all *p2* and *p3* values would be divided by 3. After the score file has been attended to, the performance program would compile the orchestra file into instruction cells and send them to the data flow machine. The parameters from then the note statements would then be transmitted to the data flow machine to play the orchestra.

6.1.3 Data flow representation of an instrument block

Chapter five detailed how the orchestra file could be compiled. However it did not give an example of how the flow graph for an instrument description, (the box labelled <name> in Fig. 5.5(d)) might be generated. Fig. 6.4 depicts the flow graph for the clarinet in the orchestra file of Fig. 6.2. Note that the paths through which note rate signals (ie. *p4*) flow need not be maximally pipelined. However the paths of the graph through which audio rate signals flow must be maximally pipelined. Identity operators would have to be inserted in Fig. 6.4 on the lines labelled 1, 2 and 3 to ensure that all audio rate paths in the flow graph contain the same number of actors.



DATA FLOW GRAPH FOR

THE CLARINET

Fig. 6.4

6.1.4 System Specifications

When the orchestra file of Fig. 6.2 is compiled, it consists of 3500 instruction cells. Thus the instruction cell memory must contain 112,000 free bytes in which the orchestra file can reside. Of the 3500 instruction cells fifty are structure operations. Thus it is thought that two processors are adequate in handling the required computation rate. One of them would of course be a structure processor. For a sampling rate of 40 Khz the structure processor would have to process structure operations at a rate of 2 Mhz. With a typical memory access time of 500 nsec this appears achievable. Of the remaining 3450 instruction cells containing scalar operations, 1400 take note rate signals as their operands. Therefore only 2050 scalar operations need to be processed every 25 microsec. A scalar processor with a processing rate of 85 Mhz could easily handle these operations at the desired speed. It would be necessary for the data flow machine to have two additional processors. One would be dedicated to the handling of the note statement parameter I/O. The other is required to manage the output of the samples to the DACs. The arbitration and distribution networks would have to be built so that the difference between the time from which an instruction is enabled to the time at which the result arrives at its destination cell is no longer than 25 microsec.

A look at Fig. 6.1 helps to determine the demands that are set on the transmission rate of the note statement parameters. The worst case in terms of parameters/sec that have to be sent from the host to the data flow machine occurs in several places, one of them being in the second measure when the flugelhorn plays G natural and F natural in succesion. The section of the sorted score file that corresponds to these notes is:

```
note        flugelhorn    5.75    .25...
note        flugelhorn    6       .75...
note        pianobass     6       2...
```

It is reasonable to assume that note parameters, pn: $n>1$, will be sent in packets each packet containing the note parameter and the instrument for which it is intended. A note parameter packet

might possibly contain six bytes, four for the parameter value and two for the parameter number and the instrument identifier. The flugelhorn instrument takes nine parameters and the pianobass instrument takes four parameters. Thus a total of nineteen note parameter packets must be sent in .25 beats or .0833 seconds. The requirement of the interface between the host computer and the data flow machine is that 228 note parameter packets (possibly 1368 bytes) per second be transmitted.

6.2 Conclusion

This thesis set out to investigate the feasibility of real time performance of a musical composition on a computer synthesis system. The Music-11 synthesis system and synthesis language were used as models in the design of a proposed Music-df system. The Music-df system uses a data flow machine to exploit the parallelism that exists in the performance of a composition. The previous section presented an orchestra and score file of medium complexity and illustrated that real-time performance of these files could occur if certain design specifications of the data flow machine and its interface to the host computer are met.

The advantages of the Music-df system are apparent. Is behavior more closely parallels that of a real orchestra. Once the orchestra has been loaded into the data flow machine different score files can be used to play it without having to recompile the orchestra. The instruction cell translation of the orchestra file returns to its initial state after performance. The savings in storage is large. On conventional computer synthesis systems, the samples of the output voltage waveform must be saved. For one minute of sound and a sampling rate of 40 Khz, 2.4 million real values have to be stored. Finally, the greatest advantage of the Music-df system from a composer's point of view is the relatively small turnaround time. A real time system spares the composer the annoyance of having to wait long periods of time to hear his composition. When he fine tunes a parameter, a composer would like to be able to hear a sound while the previously produced sound is still fresh in his mind.

Whether or not the requirements of the data flow machine outlined in the last section can be

met has not yet been determined due to the fact that the first data flow machine whose architecture is that of Fig. 4.3 is presently under construction. This machine could well forecast the future of a system similar to the Music-df system.

6.3 Suggestions For Future Research

It is easy enough to determine the required specifications for the Music-df system so that a given orchestra and score file can be performed in real time. However it is impractical to build a Music-df system for every orchestra and score file to be performed. It is much more expedient to construct an all purpose system that would support many different orchestra and score files. In order to determine what the design specifications of such a system would be, more analyses of existing orchestra and score files needs to be undertaken. As an example, for the two files presented in this chapter, a requirement of the data flow machine was that the structure processor be able to process structure operations at a rate of 2 Mhz. With a typical memory access time of 500 nsec, a 2 Mhz rate is an upper limit of what the structure processor can handle. The structure controller of Fig. 4.4 can handle instructions only as fast as the structure memory will allow it. It is not unusual for an orchestra file to have more oscillators than the orchestra file of Fig. 6.2, requiring a higher structure processing rate. It is clear that one structure memory cannot meet this requirement. The only solution is to have multiple instruction memories. A structure processor consisting of two structure memories accessed by on structure controller could deliver a 4 Mhz instruction rate provided the operations were evenly distributed between the two memories. The number of structure processors, the number of memories in each structure processor are two of several system parameters that would have to be studied more carefully in the design of a more general real time synthesis system.

REFERENCES

[1] Ackerman, W. B. and J. B. Dennis. *VAL - A value-oriented algorithmic language preliminary reference manual.* Technical Report LCS/TR-218, Laboratory for Computer Science, Mass. Institute of Technology, Cambridge, Mass., June 1979.

[2] Acherman, W. B. *A Structure Memory For Data Flow Computers.* Technical Report LCS/TR-186, Laboratory for Computer Science, Mass. Institute of Technology, Cambridge, Mass., August 1977.

[3] Arvind, and K. P. Gostelow. A computer capable of exchanging processors for time. *Information Processing 1977,* North Holland, New York 1977, 849-854.

[4] Arvind, K. P. Gostelow, and W. Plouffe. *An Asynchronous Programming Language and Computing Machine.* Techinal Report 114A, Dept. of Information and Computer Science, University of California, Irvine, December 1978.

[5] Brock, J. D. and L. B. Montz. *Translation and Optimization of Data Flow Programs.* Computation Structures Group, Laboratory for Computer Science, Mass. Institute of Technology, Cambridge, Mass. July 1979.

[6] Chowning, J. M. The Synthesis of Complex Audio Spectra by Means of Frequency Modulation. *Journal of the Audio Engineering Society,* Volume 21, No. 7, September 1973.

[7] Dennis J. B. and D. P. Misunas. *A Preliminary Architecture for a Basic Data-Flow Processor.* Computation Structures Group, Laboratory for Computer Science, Mass. Institute of Technology, Cambridge, Mass, August 1974.

[8] Dennis J. B. and K. -S. Weng. *Application of Data Flow to the Weather Problem.* Computation Structures Group, Laboratory for Computer Science, Mass. Institute of Technology, Cambridge, Mass. May 1977.

[9] Dennis, J. B., C. K. Leung and D. P. Misunas. *A Highly Parallel Processor Using a Data Flow Machine Language.* Computation Structures Group, Laboratory for Computer Science, Mass. Institute of Technology, Cambridge, Mass. June 1979

[10] Dennis J. B. and K. -S. Weng. An abstract implementation for concurrent computation with streams. *Proceedings of the 1979 International Conference on Parallel Processing,* IEEE, August 1979.

[11] Dennis J. B., *The Varieties of Data Flow Computers.* Computation Structures Group, Laboratory for Computer Science, Mass. Institute of Technology, Cambridge, Mass., August 1979.

[12] Dennis J. B., G. A. Boughton and C. K. C. Leung. *Building Blocks for Data Flow Prototypes.* Computation Structures Group, Laboratory for Computer Science, Mass. Institute of Technology, Cambridge, Mass., February 1980.

[13] Howe, H. S. *Electronic Music Synthesis.* W. W. Norton & Company Inc., New York, 1975.

[14] Mathews, M. V. *The Technology of Computer Music.* The MIT Press, Cambridge, MA, 1969.

[15] Moorer, J. A. Signal Processing Aspects of Computer Music - A Survey. *Computer Music Journal,* February 1977.

[16] Oppenheim, A. V. and R. W. Schafer. *Digital Signal Processing.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1975.

[17] Stanek, J. A *Exploration of Concurrent Digital Sound Synthesis on a Prototype Data-Driven Machine.* Dept. of Computer Science, The University of Utah, December, 1979.

[18] Stoy, J. E. *Functions in the Form 1 Data Flow Machine.* Unpublished Communication.

[19] Todd, K. *An Interpreter for Instruction Cells.* Unpublished Communication.

[20] Vercoe, B. *Reference Manual for the Music-11 Sound Synthesis Language.* Experimental Music Studio, Mass. Institute of Technology, Cambridge, MA, 1980.

[21] Von Foerster, H. and J. W. Beauchamp. *Music By Computers.* John Wiley & Sons, Inc., New York, 1969.

CS-TR Scanning Project
Document Control Form

Date : 8/24/95

Report # LCS-TR-257

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 130 (135 images)
Not to include DOD forms, printer instructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☒ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number): UNN'ED BLANK FOLLOWS PAGES #3, TITLE PAGE, ABSTRACT

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP:	(1-6)
UNN'ED TITLE BLANK, ABSTRACT, BLANK, PAGE #3, UNN'ED BLANK	
	(7-130) PAGES #'ED 4-113, NO PAGES 114-115, 116-129
	(131-135) SCAN CONTROL, PRINTER'S NOTES, TRGT'S (3)

Scanning Agent Signoff:
Date Received: 8/24/95 Date Scanned: 9/22/95 Date Returned: 9/28/95

Scanning Agent Signature: Michael W. Cook

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 9/29/1995

M.I.T. Libraries
Document Services