LAGER: AN AUTOMATED LAYOUT GENERATING SYSTEM FOR DIGITAL SIGNAL PROCESSING CIRCUITS. USER MANUAL - VERSION 1.3

by

Jan Rabaey

Memorandum No. UCB/ERL M85/5

15 February 1985

(cover)

LAGER: AN AUTOMATED LAYOUT GENERATING SYSTEM FOR DIGITAL SIGNAL PROCESSING CIRCUITS. USER MANUAL - VERSION 1.3

by

Jan Rabaey

Memorandum No. UCB/ERL M85/5

15 February 1985

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

LAGER
An automated layout generating system for Digital Signal Processing Circuits
User Manual - Version 1.3

Jan Rabaey

Department of Electronic Engineering and Computer Science
University of California
Berkeley, CA 94720

December 1984

Table of contents :

This research was sponsored by DARPA under contract number N00039-84-C-0107

Part 1 : Philosophy and Architecture Description

1. Introduction

This text describes the hardware configuration of a programmable processor approach for digital signal processing. The particular characteristics of the described system are :

- Parallel processing to allow for an increased data-throughput.
- A limited instruction set to keep the arithmetic and control units small, allowing a number of parallel processors on the same chip.
- The basic speed of the processors is increased using fully pipelined arithmetic units. Multiplications are performed in a parallel-serial way . The elimination of a parallel array multiplier keeps the arithmetic units small.
- All arithmetic is performed in the two's complement number representation.
- The use of a well defined set of macrocells allows for full automatic layout generation.
- Debugging of the processors microcode is facilitated by a special purpose emulator. This simulator checks the input code, sets up the control sequencers, checks for timing conflicts. A full debugging mode is provided featuring traces, breakpoints and step procedures.
- A design file description language has been developed for this particular system. It contains a detailed description of the complete system and serves as a single input to the layout generator and the emulator.

The resulting system allows for a fast silicon implementation of custom signal processing applications. The system is particularly suited for speech-band applications, although higher frequency operations are possible, exploiting the parallel processing feature.

In this first part, we describe the architecture of the basic building blocks of the signal processing system.

2. General description

A signal processing machine, based on our concept, has the following basic architecture :

(1) A number of parallel processors. Each of these processors is built up using a combination of macrocells, being RAM-data-memory, a pipelined arithmetic unit, an I/O unit, an adress arithmetic unit, a control sequencer and a decision making finite state machine (optional).

(2) Interprocessor communication proceeds via a number of serial connections. The timing of these transfers is decoupled from the processor timing by data buffering.

(3) A parallel bus to communicate with the outside world. This bus is connected to only one processor and does not provide any data buffering. Strobe lines are provided by the processor control to time the data in- or output.

(4) A buffered host I/O-section, which allows for a low data rate communication with a host microprocessor. The data exchange proceeds at a rate, FRAME times slower than the sample rate, where FRAME is set by one of the processors. I/O data is buffered in a set of FIFO's .

The architecture is illustrated in Figure 1. A more detailed discussion of the different parts follows next.

3. The individual processor structure

Each processor is composed of a number of basic macrocells, as already described in the introduction. These macrocells are invoked with a set of application-dependent parameters as wordlength, number of words (RAM or ROM), programming (micro-code ROM), depth (FIFO's). The layout generator translates the design file description into a set of these parameters. A precise description of each of these macrocells and of their function in the system is given next.

3.1. The data memory

Variables, local to a processor, are stored in the local data memory. The present version provides for up to 128 RAM storage locations per processor. Constants are also stored in the data-memory, this with a minor change in the RAM-cell. This allows for a complete separation of data- and control paths, in contrast with most of the existing general purpose signal processors.
Two types of memory-addressing are provided : absolute addressing and indexed addressing through an ix- and an iy-register (cfr. address-arithmetic).

3.2. The arithmetic unit

The arithmetic unit is implemented as a pipelined structure with a variable wordlength dependent upon the application. All data is represented in two's complement notation.
Fig. 2 pictures the processor data path, which is basically a pipeline with four stages : the mor, the sor, the accumulator and the mir-registers. The path is composed of following building blocks:

- The mor-register is loaded with the contents of a Ram-word in a r(ead)-operation and with ~ mir (where ~ means bit inversion) in a w(rite) or mor := ~ mir operation (cfr. assembler-language part II).
- The barrelshifter allows for a right shift of 0 to 7 bits. It takes either the mor-register or the sor-register as input . The latter makes shifts of more than 7 bits feasible. When performing a parallel-serial multiply or divide operation, the combination shifter-sor is used to shift right repeatedly on successive cycles. This presents a sequence of partial products to the adder input.
- The complementer outputs either the true value, the inverse or the absolute value of the sor-register. The action performed is either controlled by the ROM-controllines or is

made programmable to allow for four quadrant variable-variable multiplications.

- The adder is saturating: if an overflow is detected, the output will be the maximum or minumum value that can be represented, for positive or negative overflow respectively. The output of the adder is loaded in the acccumulator . The adder has as input the so called abus and bbus. The abus equals either the output of the complementer or zero. Once again, the control of this multiplexer can be made programmable to allow for variable coefficients. (cfr. multiplication in part II). The bbus equals either the accumulator, the mor-register or zero.
- The mbus is either driven by the accumulator (default), the mor-register or from the i/o-unit (in case of input). The mbus drives the mir-register . This is a transparent latch, which may either load or hold data under program control and introduces an extra stage of possible storage in this way. All i/o-transfers are going through the mbus as well.

3.3. The i/o-unit

The i/o-unit consists of the connection to the parallel i/o-bus (for only one of the processors) and the serial interconnections with the other processors and the host i/o-unit. These serial i/o-units include serial-to-parallel (s/p) or parallel-to-serial (p/s) converters and also the buffering latches, which allow for an user-timing independent interprocessor communication.

3.4. The control sequencer

The controller consists mainly of two cycle-counters and the micro-code ROM. It is kept simple and small by a strict restriction of the looping and branching facilities, encountered in classical processors.

In fact, signal processing algorithms involve the unconditional execution of the same sequence of operations every sample interval. If functional multiplexing is used (wherein repeated sections of the algorithm are realized with the same hardware elements), the control sequencer must loop through a section of code several times. Still, the control flow is data-independent; the number of iterations is predetermined and does not vary from sample to sample. It is very desirable to maintain this data- independent control flow in pipelined systems.

Therefore, we came up with the following control sequence : each processor starts a new sample-interval with the execution of a so called "main-program". This is a piece of microcode, which is only executed once a sample. Next, the sequencer proceeds with the execution of a fixed number of instances of a "sub-program". The number of iterations is defined by the programmer and allows for hardware multiplexing. Note however that both main- and sub-program have a fixed length and do not contain any branching. How the iterations of a sub-program can deal with different data is described in the address-arithmetic unit.

It is however clear that some signal-processing algorithms require some decision-making and conditional operations. A restricted form of conditional testing is provided in this approach with the aid of an user-defined finite state machine.

3.5. The finite state machine

Decision-making can be implemented easily by the introduction of a conditional write instruction, wherein an assignment is made to a variable in data memory only if a condition-code bit is set.
This condition-code bit is one of the output variables of a user-defined finite state machine (pla). This output variable controls the write-enable of the data-memory . Following signals can be used as input to the finite state machine :

- The sign-bit of the accumulator, which allows for comparison operations.
- The individual bits or values of the ix- and the iy-index-registers.

User defined instructions, which change the state of the FSM are processed in parallel with data path operations.
The above organization is quite general and allows for any kind of decision making. It keeps the control flow (and at the same time the execution time) data-independent, which simplifies the design of real-time systems.

An additional task of the finite state machine is the setting of the eof (end-of-frame) and mof (middle-of-frame)-flags, which control the communication between signal- and host-processor (cfr. host-io unit).

3.6. The address-arithmetic unit

Two kinds of addressing are supported in the current architecture :

- direct addressing : the RAM-address is taken directly from the address field of the ROM-control-word without any modification. If a processor uses only this kind of addressing, no address-arithmetic unit is needed.
- indexed addressing : in some cases however, some kind of indexing is needed. This is e.g. the case when functional multiplexing (with subprograms) is needed, or for table look-up, interpolation, decimation ... Therefore, two index registers ix and iy with different functions have been provided . The contents of these registers is added to the ROM-address-field, when performing indexed addressing.
  The ix-register counts the iterations of the subprogram. This register is not user-accessible (with exception of the finite state machine, which can base decisions on the value of ix). The user only defines the modulus of the counter, when specifying the number of iterations of the subprogram. Ix-indexing is mainly used in functional multiplexing, where the data inputs and outputs and the state variables of the individual sections are accessed by x-indexing in an array.
  The iy-register is normally implemented as a counter with a user defined modulus. The counter is incremented at the start of every new sample period. This kind of indexing can be used for decimation. Another possibility is to implement the iy-register as a register, which can be loaded from the mbus via a parallel bus. This configuration allows for table lookup and memory precessing.

4. Interprocessor communication

As already described, interprocessor communication is performed over serial lines. The i/o-units of the different processors take care of the parallel-serial and serial-parallel conversion and also perform the necessary buffering . This scheme simplifies the job of the programmer, so that he does not have to worry about the synchronization of the timing of the different processors.

The emulator Demon (see part III) performs a basic role in this process : in fact, it checks if the programmer does not introduce timing conflicts, e.g. by sending a signal over a serial i/o-line which is still in use. It also warns the user if the additional delay over the serial lines introduces sample delays.
The first pass of the compiler Archer assigns the i/o-units to the different processors, determines on which side (transmitter or receiver) the buffering has to be done and provides the control signals, needed for the exact timing of the transfers.
This results in a communication proces, which is designed in a fully automated way and which is completely hidden to the user.

5. The parallel i/o-bus

The parallel i/o-bus allows input and output at sample-frequency rate. Only one processor can be connected to this bus and the data is not buffered in FIFO's. The timing of the data-flow is controlled by a number of strobe signals, which are generated by the control section of the processor. The layout generator takes care for the provision and the exact timing of

these strobe signals. Also important is that the hardware does not allow a parallel output and a parallel input in subsequent instruction-cycles.

6. The host-i/o section

A large number of speech applications require a low data rate communication with a host processor. Therefore, an i/o-unit with FIFO-buffers is provided. This unit communicates in a serial way with the different signal-processors (in an identical way as the interprocessor communication). The host-i/o unit itself consists mainly of S/P and P/S converters and the buffering FIFO's.

The exchange-proces is repeated over an interval of FRAME samples, while the communication with the host-processor is interrupt-controlled . These interrupt signals are generated by the finite state machine of one of the signal processors. A first interrupt signal (mof) is generated somewhere in the middle of the frame and starts the data transfer from host to i/o-block. The period between the mof-signal and the end of the frame has to be large enough to allow for a complete data transfer and has to be smaller than 3 milliseconds. A second interrupt (eof) denotes the end of the frame and initiates the data output to the host.

Once again, the timing and the architecture of the host-i/o-unit is completely generated by the compiler and hidden to the user.

7. Summary

The different hardware blocks have been descibed. In the next part, we describe a design file-language, which allows for a full description of a signal processing machine and which serves as a single input to both the emulator and the layout generator.

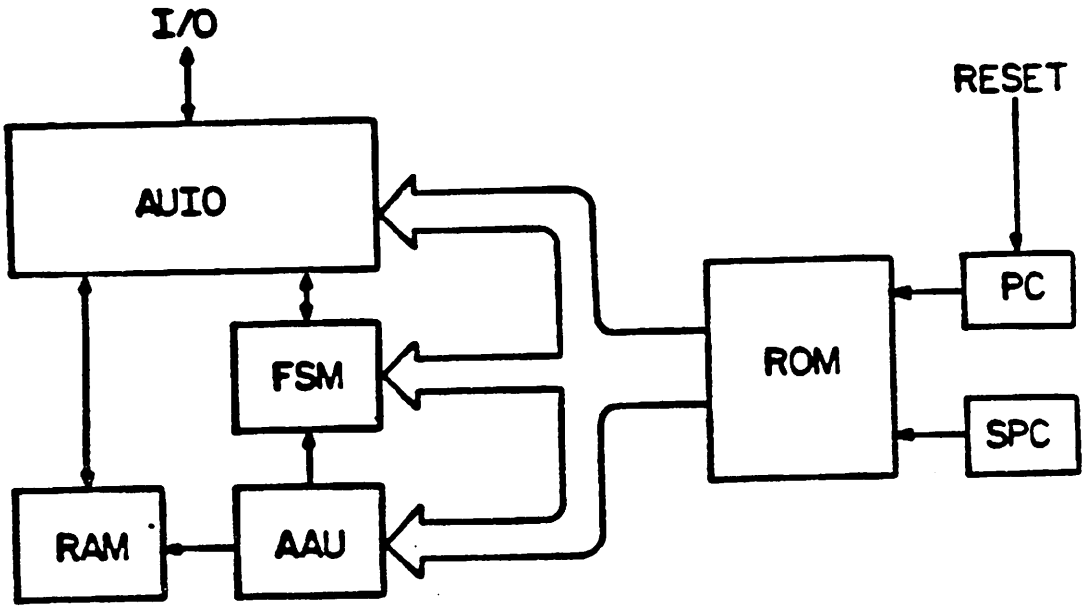


Fig. 1b Macrocells forming a single processor

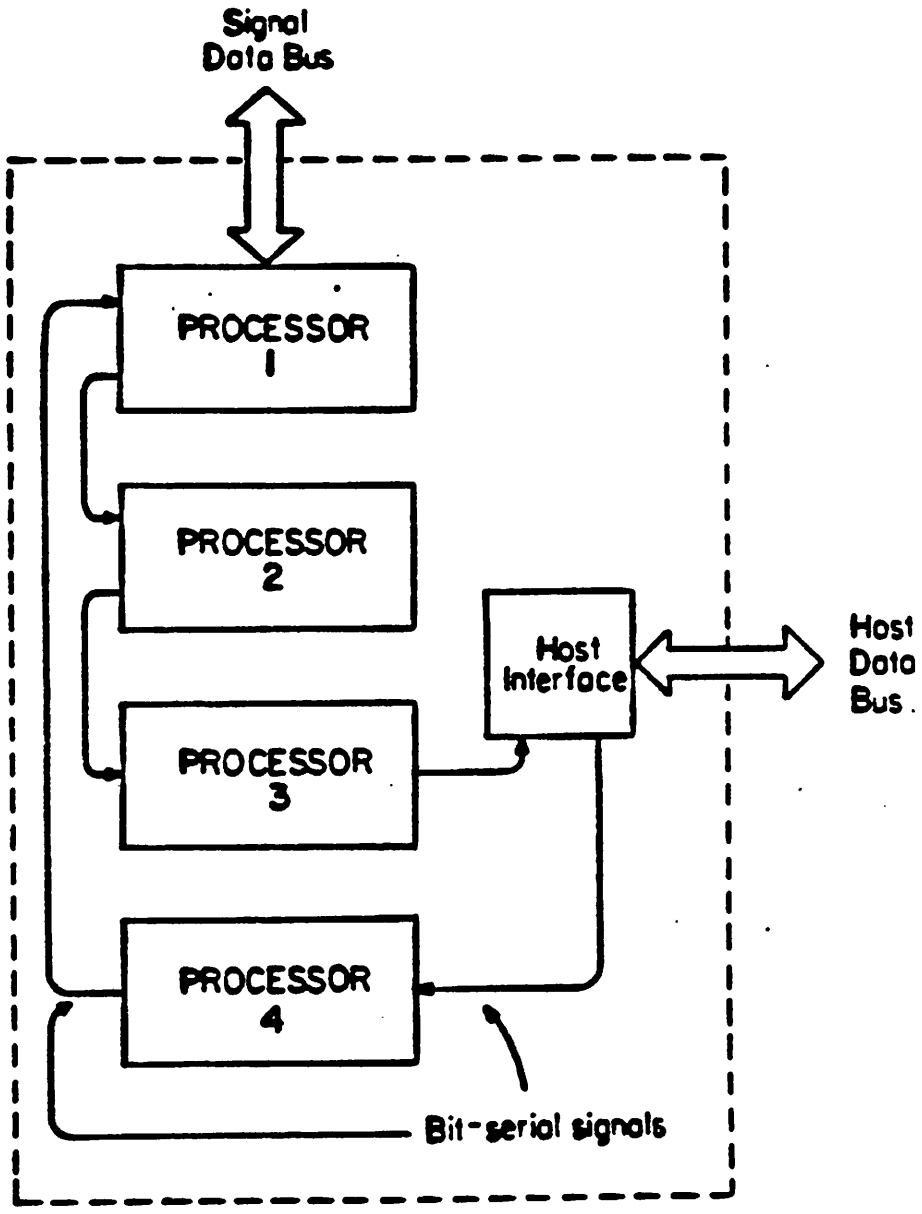


Fig.1 Organization of a four-processor IC

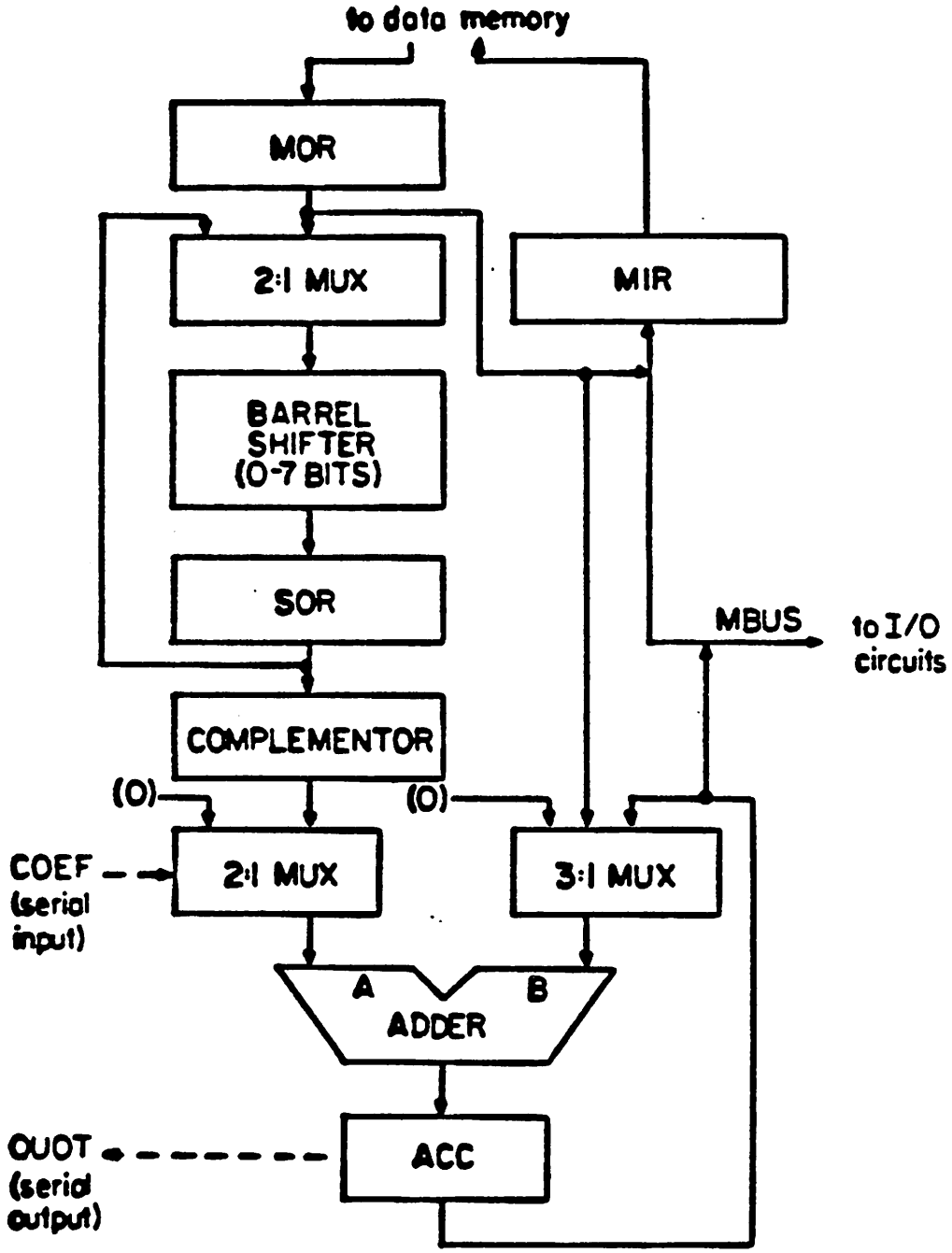


Fig.2 Processor data-path architecture

Part 2 : The design file

1. Introduction

The design file gives a full description of a signal processing IC and serves as a single input to the emulator and silicon compiler, so that both design tools are consistent . A special purpose language has been developed. This language, which describes the system at an intermediate level, can be kept rather simple, due to the restriction to a well defined processor and interprocessor architecture. (The design file will be generated by a higher level compiler in a future phase.)

For the formal definition of the syntax of the design file, a notation proposed by K. Jensen and N. Wirth [1] is used : semantic constructs are denoted by english words between the angular brackets < and >. These words are suggestive of the nature or meaning of the construct. Production rules use ::= to define a construct as an expression or a combination of constructs; curly brackets { } indicate repetition any number of times including zero; square brackets [ ] indicate optional factors (i.e. zero or one repetition); parentheses () are used for grouping. The vertical bar | is used for the or-ing of constructs.

The design file contains a detailed description of the basic blocks of the signal processing IC : the processors, the host I/O and signal I/O, the interprocessor communications. This determines the general format of the design file:

<design-file> ::= <globals> <I/O> (<proc> | <proc>) [<constraints>]

In the next sections, the syntax and the contents of each of these blocks will be discussed in detail. This is preceded by a discussion of the number representation used in our architecture and its implications on the arithmetic operations.

2. Number representation and arithmetic operations

The two's-complement number representation has proven to be the most flexible way to handle negative numbers in a binary number system and is therefore used for all arithmetic operations in our structure. In this section, details of this representation are given. The way in which two's-complement multiplications and divisions are performed is described.

In the two's-complement number notation, the representation of a positive number is identical with its representation in an unsigned binary format. If a number, x, is negative, the two's-complement of x is given by Eq. (1):

$$\textit{Two's-complement}(x) = 2^n - |x| \quad (x < 0) \quad (1)$$

One advantage of the two's-complement approach is that two's-complement addition is the same as the addition of two positive arguments.

2.1. Variable-variable multiplications

Suppose now that we want to multiply two two's-complement numbers x and y and that y is a fractional number ($-1 \leq y < 1$) with word-length n. It can be seen that the y can be expressed in terms of its two's-complement notation as shown in (2), where y_i is the i-th bit of the 2's-comp. representation.

$$val(y) = -y_0 + \sum_{i=1}^{n-1} \frac{y_i}{2^i} \quad (2)$$

This form allows a simple procedure for multiplication.

$$xy = -xy_0 + \sum_{i=1}^{n-1} \frac{x}{2^i} \cdot y_i \quad (3)$$

Equation (3) suggests a method of multiplying two two's-complement numbers in a serial-parallel fashion. Start with either 0 (when y is positive) or -x (y negative). Add x, shifted over i positions, if the i-th bit if the coefficient y is one. Note that the bits of the coefficient are needed serially on succesive cycles, MSB first, in order to control the addition of the shifted values. (Examples of how this procedure is microcoded are given in Section 6.4.2.9.)

Example 1 : multiplication of 011000 (3/4) with 0011 (3/8)

```
000000          /* sign bit of coeffient is zero */
0000000   /* first significant bit zero   */
00011000  /* bit 2 equals 1 : add x/4     */
000011000/* lsb equals 1   : add x/8      */

---------
001001000/* total = 9/32                  */
```

Example 2 : multiplication of 011000 (3/4) with 1101 (-3/8)

```
101000          /* sign bit of coeffient is one , invert x */
0011000   /* first significant bit is one , add x/2  */
00000000  /* bit 2 equals 0                          */
000011000/* lsb equals 1   : add x/8                 */

---------
110111000/* total = -9/32                 */
```

The result of multiplication has to be truncated to the wordlength of the processor, droppping the least significant bits (3 in the above examples). It is necessary to take this effect into account when determining the required number of bits in the data word for a given application and given performance criteria.

2.2. Multiplication with a constant

The serial-parallel multiplication procedure, described above, can be optimized when the coefficient is a high precision constant. The number of cycles needed can be reduced drastically by representing the coefficient in the canonical signed digit form. In this form, a constant is represented in the form (4), where the x_i have values of $\pm$ 1, and the exponents n_i are chosen so as to minimize j, the total number of digits. This representation typically has one third the numbers of the digits as does a binary representation, with no loss in precision.

$$csd(x) = \sum_{i=0}^{j} x_i 2^{n_i} \quad (4)$$

Example : the constant 0110111 (55/64) can be represented as :

$$2^0 - 2^{-3} - 2^{-6}$$

When multiplying by the above constant, only two additions are required. It should be kept in mind that is only very effective for constant coefficients, where the canonical form can be calculated before the processor's code is written.

2.3. Division of two variables

Divide operations are implemented using long division. To find the quotient N/D with |D| > |N|, one first has to determine the sign of quotient (for four quadrant divisions). The division operation itself is performed as follows : |N| is loaded in the accumulator. On succesive cycles |D/2, |D/4 ... is subtracted form the accumulator, the result being accumulated only if it is positive. If so, a one is added to the quotient-result, otherwise a 0 is added. In this way, a sign magnitude representation of the quotient is obtained in a bit-serial fashion. The result can be transformed to two's-complement notation using some additional hardware.

Example : Divide 001 (1/4) by 0101 (5/8) (the result is positive).

```
001000    |0101
111011    ---------     /* add - 000101 (bit 3 of quot = 1) */
------    0011_
000011
1111011                 /* add - 0000101 (bit 4 of quot = 1) */
-------
0000001
```

The 4-bit result equals 0011 (3/8) which approximates the exact result (2/5). Section 6.4.2.10 describes the procedure to implement this long division in microcode.

3. General syntax description

Each block has the same basic syntax :

<block> ::= <keyword [parameters] CR> begin <statement> end

with

<statement> ::= <dataline;> { <dataline;>}

Note that only the semicolon is considered as a statement separator. Blanks, tabs and carriage returns are ignored. The only exception is the .keyword line, which has to be terminated by a carriage return (CR). Certain keywords may be followed by parameters.

Comments can be included as follows :

<comment> ::= /* put comment here */

These comments can be inserted everywhere in the design file.

4. The global-block

A global is a variable which is shared between different processors or which is shared with the outside world (host- and signal-I/O). All these variables are processed in bit-serial form and are buffered in temporary latches. The only exceptions are signal-I/O variables, which are parallel and unbuffered. It must be noted that the bit-serial transfer causes a delay of a number of clock cycles, on the order of the word length of the global variable.

Global variables which are used for host I/O may be declared as arrays. Input or output of array variables is indexed automatically with the ix-register if the reference to the global is in a subprogram, and with the iy-register if the reference is in the main program. Global array variables are stored in FIFO structures in the Host Interface. Globals used for signal I/O or interprocessor communication are never arrays.

Globals may be indexed by the iy-register only in counter mode and not in pointer mode. These two modes of using the iy-register are descirbed below.

The syntax of a global block :

<global-block> ::= .global CR begin <block-statement> end
<block-statement> ::= <data_line;> { <data_line;>}
<data_line> ::= <variable_decl> {,<variable_decl>} : [<justification>]
<variable_decl> ::= <name [dimension] <word_length>>
<justification> ::= left_justified | right_justified

("name" is a user-supplied variable name.) Dimension declares the dimension of an array variable. If no dimension is specified, a scalar variable is assumed. Word_length defines the wordlength of the global. This denotes the number of bits which is transferred over the serial line. It defines the number of lines in the case of parallel transfer.

Justification denotes how the words are truncated if the wordlength of the processor and the global are different : left_justified preserves the most significant bits or pads the word with zero's at the right side (if the word has to be lengthened). Right_justified selects the least

significant bits (and drops the sign-bit). This is e.g. important when positive counter values have to be transferred. left_justified has been selected as the default value.

IMPORTANT REMARK : The definition of a global results in the generation of extra hardware at the transmitter and the receiver side. Excessive use of globals results in a untolerable growth of the processor-dimensions. The user should be careful and should try to keep the number of global definitions to a strict minimum in order to obtain an area-efficient design.

example :

```
.global /* interprocessor and I/O communications */
begin
   ka[10]<8>, ks[10]<8>; /* array variables with dimension 10 and
                            wordlength 8 - left_justified        */
   pitch_in<8>, pitch_out<8> : right_justified;
                         /* scalar variables with wordlength 8 -
                            right_justified                      */
end
```

5. The I/O-block

This block defines which of the global variables are selected as I/O-variables.

Syntax:

<io-block> ::= .io <<host_word_length>>CR begin <block-statement> end
<block-statement> ::= <data_line;> { <data_line;>}
<data_line> ::= <variable> {,<variable>} : <io_specification>
<io_specification> ::= host_in | host_out | signal_in | signal_out

The host_word_length is specified on the same line as the .io-keyword and determines the size of the parallel bus, connecting the host-I/O unit and the host-processor. This equals normallly the wordlength of the host processor itself (8 or 16 bits) and by default is set to 8.

The variable specification specifies which global variable (already defined in .global) is connected to a I/O-unit. The io_specification determines the type of I/O . signal-i/o is transferred through the parallel unbuffered bus, while host-I/O communicates with the host-processor through the FIFO-buffered host-I/O section.

example :

```
.io <16> /* 16 bit host interface */
begin
   host_in.d : ka,ks : host_in;
   host_out.d : pitch : host_out;
   speech_in.d : speech_in : signal_in;
end
```

6. The processor block

A processor can be considered as an assembly of different macrocells. Some of these cells have to be defined by the user (e.g. data-memory, finite state machine, arithmetic unit), others are partially or completely assembled by the compiler interpreting the user defined microcode (e.g. control section, I/O-units, address-arithmetic). This leads to following general format :

<proc> ::= .processor : <name <word_length>> CR begin <sub_blk> end
<sub_blk> ::= <locals> <constants> <fsm> <main_program> <sub_program>

In the .processor line, a name is given to the processor and the wordlength of the arithmetic unit and the data memory is determined. Note that this wordlength is sufficient to assemble the complete aritmetic unit.
The syntax and meaning of the different subblocks is demonstrated in the following sections. Note that each of these blocks is optional. However either a main_program or a sub_program should be provided.

6.1. The local-block

In the hardware description, it has already been mentioned that the data memory of each processor can be a mixture of RAM and ROM. The RAM-memory locations are denoted as locals, the ROM-words are defined as constants (see constant-block).

Syntax :

<locals> ::= .localCR begin <data_line;> {<data_line;>} end
<data_line> ::= <name[dimension]> {,<name[dimension]>}

If no dimension is specified, the variable is considered to be scalar, otherwise an array of length [dimension] is reserved.

6.2. The constant-block

For definition, see local-block.

Syntax :

<constants> ::= .constantCR begin <data_line;> {<data_line;>} end
<data_line> ::= <name[dimension] = value {,value}>

Each data-line contains the definition and the initialization of only one constant-type. Arrays of constants of length [dimension] can be defined. In that case, the number of values has to equal the dimension of the array.

example :

```
.processor : pitch <18>
/* implements the Gold pitchtracker algorithm */

begin
   .local
   begin
      thresh[6], ppc[6], pp[7], lpp[6], signal[6];
      ls, lp, lv, score, topscore, pitch, winner;
   end

   .constant
   begin
      TWO = 2;
      BLANK = 24;              /* definition of blanking interval */
      VOICED = 9;           /* speech if unvoiced if score < VOICED */
      WINDOW1 = 8;             /* windows to compare pitches        */
      WINDOW2 = -14;
   end

   _ definition of fsm and microcode
end
/* end of pitch_tracker definition */
```

The definitions in the above example make provisions for a data-memory of 43 words (38 RAM + 5 ROM).

6.3. The finite state machine

A limited form of decision-making is feasible with the definition of a finite state machine . This machine controls a conditional code-bit (cc), which in its turn governs the write operation. This results in a conditional write-instruction. The finite state machine operates in parallel with the processor.

The user defines the finite state machine completely. The state variables may be given arbitrary names, except for the following following reserved names : cc, mof and eof (cfr. host_io). Note that a conditional write operation is only possible when a cc-output has been defined and that only one processor may define the mof and eof bits.

Following signals can be used as input to the fsm : the states itself, the sign bit of the accumulator (TRUE if negative), the individual bits of the ix- and the iy-registers, and expressions of the form "ix=constant" or "iy=constant".

Syntax :

<fsm> ::= .fsm CR begin <command_def;> { <command_def;>} end
<command_def> ::= <cmd_name> : <equation> {,<equation>}
<equation> ::= <state> = <expression>
<expression> ::= combination of <booleans> and <operands>
<booleans> ::= <state>, sign , ix[c], iy[c], ix<>, iy<>
<operands> ::= &(AND), |(OR) and !(NOT)

The <cmd_name> is used to reference to a specified fsm-command in the microcode. An expression is evaluated from left to right with normal operator precedence : ! has the highest priority, and | the lowest. Parentheses can be used to change the precedence.
"ix<>" denotes the i-th bit of the ix-register, with ix<0> the least significant bit. "iy<>" has a similar interpretation. "ix[c]" denotes an equality comparison between the ix-register and a constant. This is used, for example, to execute an operation only during the final iteration of a subprogram. "iy[c]" is used similarly.

example :

```
.fsm /* finite state machine description */
begin
   SET : cc = sign;
      /* set condition code if acc >= 0 */
   AND_MINUS : cc = cc & !sign;
      /* set cc if acc <0 and cc = TRUE */
   APV : cc = cc & (!ix<0>&!slp&lsp | ix<0>&slp&!lsp);
   VPE : cc = iy[0];   /* set cc if iy = 0 */
   SIP : cc = !slp & lsp;
      /* set cc if peak */
   SIV : cc = slp & !lsp;
      /* set cc if valley */
   SSL : lsp = slp, slp = sign;
      /* set slp (slope) if acc >= 0,
        set lsp (last_slope) to slp */
end
```

6.4. The microcode-block : main program and subprogram

6.4.1. General block-syntax

The basic structure of the control-sequencer is quite strict. A processor starts a new sample interval with the execution of a main program, followed by a loop of x_mod subprograms. Note that all processors are synchronized in that they start a new sample at the same time. This means that a processor with a shorter program has to wait (execute nop's) until all other processors have finished their program. Both the main-program and the sub-program are optional.

The compiler infers the structure of the control sequencer from the microcode description. It counts the number of instructions in main- & sub-program and computes the number of cycles in a sample interval. This is done for all processors and the maximum value is taken as the modulus of the system's main program counter. The subprograms of two processors can be synchronized using the sync and the couple options (cfr. constaints).
Note that the microcode contains all the information needed for the generation and assignment of the I/O-units and the address arithmetic unit. The compiler scans the microcode to check for the occurence of indexed addressing and for different kinds of I/O.

Syntax :

<main_program> ::= .main_pr <y_mod>CR begin <microcode> end
<sub_program> ::= .sub_pr <x_mod>CR begin <microcode> end
<microcode> ::= <simultaneous_instr;> {<simultaneous_instr;>}
<simultaneous_instr> ::= <instruction> {,<instruction>}

y_mod and x_mod determine respectively the modulus of the iy- and ix-index counters. As already stated before, the ix-counter counts the number of iterations of the subprogram and is incremented at the begin of a new iteration (ix = -1 in the main program). The iy-counter (in counter mode addressing) is incremented at the start of a new sample-cycle and is used for e.g. decimation. These registers (or counters) are basically used in the indexed read- and write operations (cfr. microcode definition). The default values of x_mod and y_mod are both 1.

If the y_mod field of the .main_pr line is "*", pointer mode addressing is implied. In this event, the iy-register may be loaded from the processor mbus. The new value is available in the iy register on the second instruction following the instruction in which the assignment to iy is performed.

Each microcode line consists of a number of simultaneously executed pipeline instructions. The emulator checks the consistency of these instructions. The layout generator transforms the assembly level description into binary used for ROM programming.

6.4.2. The assembler syntax

The set of available microcode instructions is split into groups of similar instructions. Members of the same group are mutually exclusive, while instructions of different groups can be executed simultaneously. There are the instruction groups:

memory
sor
acc (accumulator)
mbus
mir (memory input register)
output
aip (accumulate if positive)
coef (coefficient)
quot (quotient)

perform simultaneously a mor, sor, acc, mbus, mir, output, aip, coef and quot- instruction.

6.4.2.1. The Memory Instructions

All these instructions affect the mor (memory input resister). "loc_add" denotes the address assigned the local variable "local". The immediate index "ind" is used to address the different elements of an array variable. "ind" can be omitted when pointing to the first element of an array or in the case of scalar variables.
The presence of a finite state machine defining 'cc' is assumed when invoking a conditional write-operation. In the case of indexed addressing, the contents of the ix- or iy-register (defined in Part I section 3.6) is added to the actual address.
Note that in the syntax definitions, ":=" denotes a storage action (assignment of a value to a storage location).

instruction	*action*
default	mor := -1
r(local[ind])	mor := mem(loc_add + ind)
rx(local[ind])	mor := mem(loc_add + ind + ix)
ry(local[ind])	mor := mem(loc_add + ind + iy)
w(local[ind])	{mem(loc_add + ind) := mir, mor := ~ mir}
wx(local[ind])	{mem(loc_add + ind + ix) := mir, mor := ~ mir}
wy(local[ind])	{mem(loc_add + ind + iy) := mir, mor := ~ mir}
wc(local[ind])	if (cc) { mem(loc_add + ind) := mir, mor := ~ mir}
wxc(local[ind])	if (cc) { mem(loc_add + ind + ix) := mir, mor := ~ mir}
wyc(local[ind])	if (cc) { mem(loc_add + ind + iy) := mir, mor := ~ mir}
mor := ~ mir	mor := ~ mir (no memory action)

6.4.2.2. The sor instructions (shift output register)

Instructions affecting the sor-register.

instruction	*action*
sor := mor	unshifted load of mor-register
sor := sor	unshifted load of sor-register
sor := mor>>n	arithm. right shift of mor over n bits (0 <= n <= 6)
sor := sor>>n	arithm. right shift of sor over n bits (0 <= n <= 6)
sor := mor<<n	arithm. left shift of mor over n bits (0 <= n <= 1)
sor := sor<<n	arithm. left shift of sor over n bits (0 <= n <= 1)

Note : the hardware doesn't support the left shift at present. A right shift of (n = 7) is supported instead. We expect the hardware adaptation to be done in the near future.

6.4.2.3. The Accumulator Instructions

The syntax of an accumulator instruction is somewhat more complicated. The adder has two input busses (called abus and bbus). Both of these can represent a whole set of different actions, so that a large number of combinations is possible. To shorten the description of the instructions, we use a simplified syntax; the accumulator instructions can take one of the following forms :

acc := 'abus'
acc := 'bbus'

acc := 'abus' + 'bbus'
acc := 'bbus' + 'abus'

where 'abus' and 'bbus' represent respectively entries from the 'abus'- and 'bbus'-tables.

'abus' table	
instruction	*action*
0	abus = 0
sor	abus = sor
~ sor	abus = ~ sor (bit-inversion of sor)
|sor|	abus = |sor| (absolute value of sor)
~ |sor|	abus = ~ |sor| (bit inverted form of absolute val.)
coef.sor	if (coef == 1) {abus = sor} else {abus = 0}
coef.~ sor	if (coef == 1) {abus = ~ sor} else {abus = 0}

The order of the arguments in the coef-instructions is not significant: e.g. sor.coef is equivalent to coef.sor .

'bbus' table	
instruction	*action*
0	bbus = 0
mor	bbus = mor
acc	bbus = acc
mor&acc (acc&mor)	bbus = mor&acc (bitwise AND-ing)

Most of these entries are clear from the above description. Some of them need however a more detailed specification.

- The output of the complementer is a 1's-complement (bitwise inversion) instead of a 2's-complement inversion. This results in an error of one least significant bit. E.g. the inversion of 0101 (5) yields 1010 (-6) instead of 1011 (-5). In digital filter implementations, this results in a small amount of additional noise. The effect must also be taken into account when using subtraction to perform a comparison.
- The output of the adder is saturating. (cfr. hardware-description)
- The coef-instruction (abus) is used for serial-parallel, variable-variable multiplications. One variable is loaded in the sor-register, while the second (coefficient) controls in a bit-serial way the output of the abus-multiplexer. In this way a shift-add multiplication is possible. More information can be found in the coef-instruction below.

6.4.2.4. The aip instruction

instruction	*action*
aip	accumulate if positive : load adder output in accumulator only if value is positive

This instruction makes the coding of a variable/variable division feasible (cfr. quotient-instruction). Besides the conditional accumulation, the aip-instruction adds a 1-bit to the

quotient-result when positive, otherwise a 0-bit is added.

Note : the sign bit of the accumulator (used in the finite state machine) is checked BEFORE the aip-hardware. This makes a negative sign possible, even when a aip-instruction is executed.

6.4.2.5. The mbus instructions

These instructions assign the value of a certain register to the mbus. It is however important to know that the mbus does not provide any storage and is not a stage in the pipeline. Storage is provided in the mir-register or an external global.

instruction	*action*
mbus = acc	(default)
mbus = mor	assigns mor-register to mbus
mbus = global	input command : loads external global

If the global (mbus = global) is defined as an array-variable, the iy- (ix-) register is used as the index-pointer to select the array-elements in the main- (sub-) program.

6.4.2.6. The latch-enable instruction (mir-register)

instruction	*action*
le	mir := mbus

The mir is a transparent latch. This means that the loaded value is immediately available and can be used in the same cycle for a write operation. Meanwhile the value is stored in the mir and remains there until the next "le" instruction. This in contrast with the other pipeline registers where the value is updated every cycle.

6.4.2.7. Output instructions

instruction	*action*
global := mbus	output of mbus to external global
iy := mbus	loads the mbus into the iy-index register

When the global (global := mbus) is an array global, the iy- (ix-) register is used as the index-pointer to select the array-elements in the main- (sub-) program.
The new value for iy ('iy := mbus'-instruction) is available in the iy-register on the second instruction following the instruction in which the assignment to iy is performed.

6.4.2.8. Finite state machine instructions

instruction	*action*
<FSM-instruction>	adapt fsm-status (execute user-defined instruction)

with <FSM-instruction> the user-defined keyword (called <cmd_name> in 6.3).

The use of the finite state machine is illustrated with the following example : suppose that we want to find the largest of two numbers, stored in the local variables a and b, and that we want to store that number in the local c. We define a finite state machine with only one instuction SET- :

SET- : cc = sign; /* set cc if acc <0 */

The following microprogram realizes the specified action :

```
r(a);                          /* ld a in mor */
r(b), sor := mor, mbus = mor, le; /* ld a in sor and mir, ld b in mor */
w(c), sor := mor, acc := sor;     /* ld a in c and acc, ld b in sor */
acc := acc + ~sor;                /* acc = a - b */
r(b), SET-;                    /* lb b in mor, set cc if b > a */
wc(c), mbus = mor, le;                 /* ld b in c if cc ( b > a ) */
                               /* else keep a in c */
```

6.4.2.9. The coefficient instruction — the multiplication operation

instruction	*action*
coef := global	feeds global into a P/S and starts shifting bits to the control of the abus-multiplexer

This instruction initiates a variable-variable multiplication. One of the variables, called the coefficient, is regarded as a fractional number (-1 <= coefficient < 1). The coefficient, which has to be defined as a global and is thus stored outside the processor's local memory, is parallel loaded into a P/S-converter and serially shifted into the control of the abus-multiplexer starting with the msb (sign-bit). When a 'one-bit' is presented, the contents of the sor-register is added to the accumulator. The contents of the accumulator is left unchanged in the case of a 'zero-bit'.

Basically, a variable-variable multiplication is performed in the following way. One variable is loaded in sor. On the next cycle, the coefficient is loaded into the P/S converter and the shifting is started. The msb (sign-bit) is presented to the multiplexer-control. The sor-value is shifted right repeatedly in the subsequent cycles, presenting a sequence of partial products to the adder input. The value of the coef-bit, present at that time, determines if this product is added to the accumulator value or not.

The following microcode fragment presents a typical multiply action. We want to multiply two variables a and b, stored as local variables in processor x. In a first step, b is transferred to c, defined as a left-justified global of wordlength 8 (the basic multiply action is going to take 8 cycles). Note that the tranfer of the n-bit variable b to the 8-bit global c results in a truncation : only the 8 most significant bits of c are retained.
Next, the serial shift-in of c in the data-path is started. The rest of the algorithm is identical to the two's-complement multiplication, defined in section 2.

```
/* multiplication - example */
r(b);
r(a), mbus = mor, c := mbus;      /* load b in the global c */
sor := mor;                       /* load a in sor */
sor := sor>1, acc := coef.~ sor, coef = c;
/* sign-bit of c (= b) shifted in coef :
   if (coef = 1) load acc with ~ sor, else acc=0 */
sor := sor>1, acc := acc + coef.sor; /* bit 2 of c */
sor := sor>1, acc := acc + coef.sor; /* bit 3 of c */
sor := sor>1, acc := acc + coef.sor; /* _      */
sor := sor>1, acc := acc + coef.sor;
sor := sor>1, acc := acc + coef.sor;
sor := sor>1, acc := acc + coef.sor;
acc := acc + coef.sor;            /* lsb of c   */
/* result of multiplication in accumulator */
```

Note that in a variable-variable multiply, the local data is always multiplied by a global coefficient. Thus, if the coefficient is stored locally it must first be assigned to a global as in the above example. Alternatively, the coefficient could originate from a different processor, where the global assignment is made.

A multiplication with a signed constant is however much simpler : Consider e.g. an 8-bit constant : 11100001 (represented in 2's complement representation). The following microcode fragment performs the multiplication of the mor-value with this constant.

```
/* multiplication with constant - example */
sor := mor;
sor := sor>1, acc := ~ sor; /* negative sign-bit, invert sor */
sor := sor>1, acc := acc + sor; /* 2nd 1-bit */
sor := sor>5, acc := acc + sor; /* 3rd 1-bit */
acc := acc + sor; /* 4th 1-bit = lsb */
/* result in accumulator */
```

The multiplication takes only four cycles here, the number of one-bits in the coefficient.

6.4.2.10. The quotient instruction – the division operation

instruction	*action*
global := quot	load result of division in global

Divide operations, by means of long division, are implemented using the "accumulate if positive" control option for the accumulator, combined with the global := quot instruction. This procedure results in a quotient, stored in a global variable (= fifo!) with a wordlength as defined in the global-definition. The bits of the quotient are obtained sequentially. The global := quot instruction closes the divide operation and stores the result in the global.
To find the quotient N/D with D >= 0 and |D| > |N|, the absolute value of N is loaded in the accumulator and -D/2 is loaded into the sor. The sign of N is automatically checked and routed into the quotient (= sign of the sor-register two cycles before the first aip). On succesive cycles D/2, D/4, _ is subtracted from the accumulator, the result being accumulated only if it is positive. A one (zero) bit is routed into the quotient-S/P if the result is positive (negative). This results in a sign-magnitude representation, which is converted automatically into a 2's comp. representation for the global.

example : (see section 2 for numerical example)

```
/* division - example , ka is defined as 8-bit global*/
r(N);
r(-D), sor := mor; /* N in sor - sign bit is tested */
sor := mor>1, acc = sor; /* load absol. val. of N in acc */
/* start division */
sor := sor>1, acc := sor + acc, aip; /*sign bit routed in quot*/
sor := sor>1, acc := sor + acc, aip; /*bit 1 in quot */
sor := sor>1, acc := sor + acc, aip;
sor := sor>1, acc := sor + acc, aip;
sor := sor>1, acc := sor + acc, aip;
sor := sor>1, acc := sor + acc, aip;
acc := sor + acc, aip;
ka := quot; /*division finished, quot in global ka */
```

Note : this example assumes that the negative value of the denominator D is available. This simplifies the code and also allows for a considerable noise reduction. Instead, we could have used the acc := ~sor + acc -command for the subtraction, but this would have resulted in a higher noise-level.

6.4.2.11. The nop instruction

instruction	*action*
nop	No Operations (executes the default commands)

Following instructions are executed by default (when not overwritten by another command) : mor := -1, sor := sor, acc := acc, mbus = acc, mir := mir. The major effect of these defaults is to refresh the values present in the pipeline registers.

7. The constraints

This block has been added to give the programmer a certain amount of control over the setup of the timing (and of the control sequencer) of the different processors. Other constraints than the ones listed can be considered on user's demand.

Syntax :

<constraints> ::= constraintsCR begin <data_block> end
<data_block> ::= <data_line;> {<data_line;>}
<data_line> ::= sync : <master> <slave> | <couple : <master> <slave> | external_sync

with <master> and <slave> processor-names.

The sync-option is used when the subroutines of two processors are communicating and thus have to be synchronised : this is the case when the subroutine of the master processor sends data (in the form of globals) to the subroutine of the slave. In order to avoid a sample-delay in this data-transfer, the sync-constraint has been implemented : the emulator (& compiler) checks the globals (and their delay), send from master to slave, and adjusts the timing of the slave so that the slave can access the data from the master in the same sample, and this without timing conflicts. It is clear that this constraint only makes sense when both processors have an equal number of subroutine iterations. Therefore, nops are added to the shortest subroutine.

The couple-option includes the sync-option, but puts some more constraints on the subroutine alignment : in this case, the slave-subroutine not only receives data from the master, but also wants to send data back. This data has to be present at the master-side before it is accessed in the NEXT iteration of master- subroutine. This asks for a somewhat more complicated alignment of the subroutines and the inclusion of extra nop-instructions.
The sync- and couple-constraints are illustrated in a pictorial way in Fig. 1.

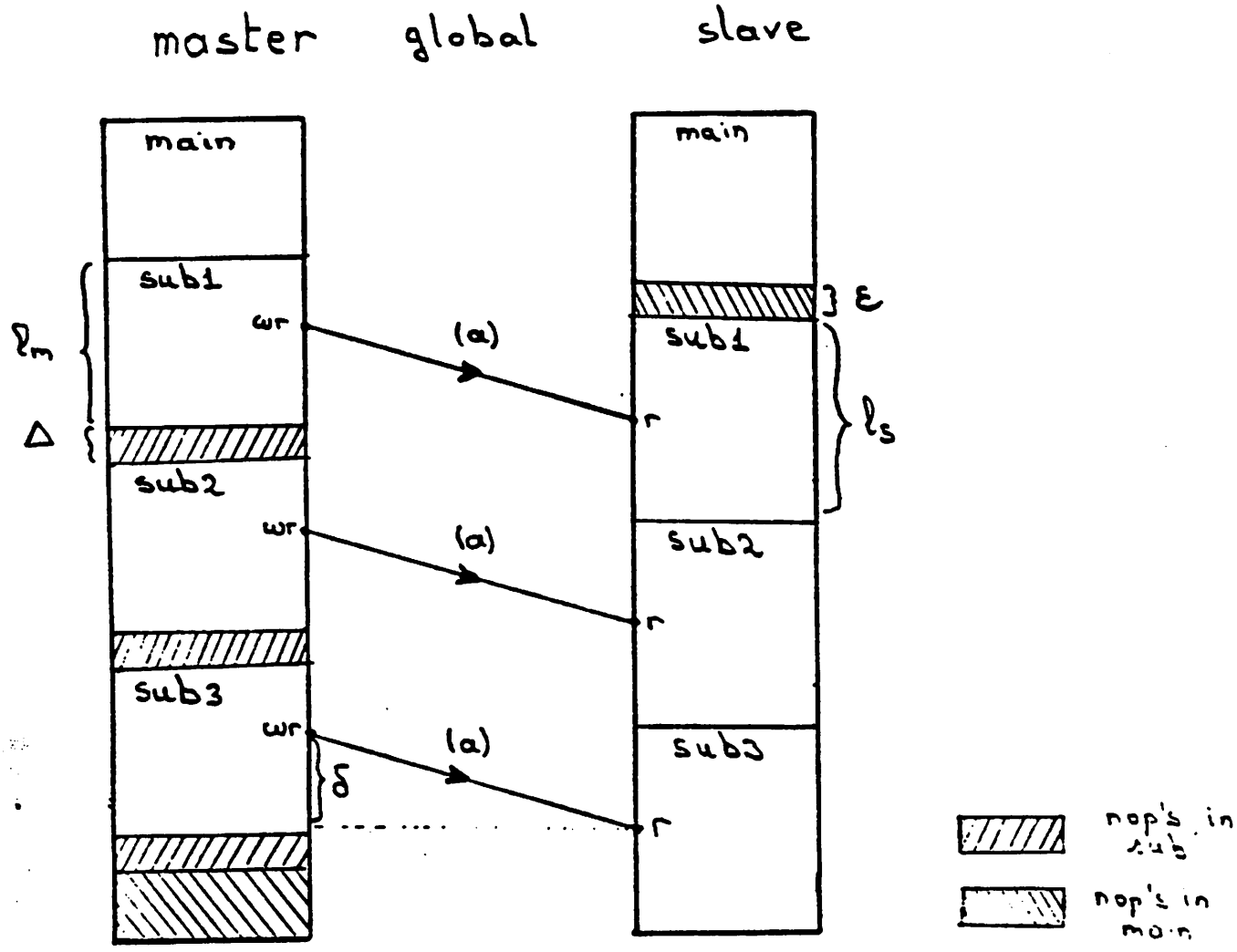
The external_sync-option allows for a synchronisation of the processors to an external clock. To achieve this, the compiler routes the reset-control lines of the processors to the outside world (the reset-line initiates a new sample-cycle and starts the execution of the first instruction of the instruction-rom).

8. Conclusions

The different aspects of the design file description of a concurrent signal processing system have been discussed. This description serves as the input to the emulator and compiler development tools.

References

[1] K. Jensen and N. Wirth , "Pascal, user manual and report", Springer-Verlag, New York 1974



a) The sync option

avoids sample delay in global communication from master to slave (for ALL globals)

- equalizes length of subroutines
  by adding Δ nop's to shortest routine
  with $\Delta = |l_m - l_s|$
- aligns subroutines of master & slave
  by adding ε nop's to main program of either {master o. slave
  so that $(t(r) \geq t(wr) + \delta)_{\text{each routine}}$
  with δ = transfer-delay

Fig. 1a

Part 3 : The emulator Demon

1. Introduction

In this part, the usage and basic functions of the emulator are described. The functions of this emulator are as follows :

(1) Remove syntax errors from the design-file description.

(2) Check the consistency of the design-file microcode.

(3) Set up the timing and the basic structure of the control sequencer.

(4) Set up the timing and the stucture of the interprocessor communications. Check the delays connected with the serial transfer.

(5) Debug the microcode and the algorithm.

(6) Perform full and realistic emulation on large data-files (e.g. speech).

The emulator performs function (1) to (4) during the read-in of the design file. If this description is found sane, an interactive stage is entered. In this stage, traces and breakpoints can be set to debug the microcode or a number-crunching run can be started . Input-data and output-data is taken from or dumped in files as defined in the design file.

2. Program evocation

SYNOPSIS

Demon -[wcdio] design_file

DESCRIPTION

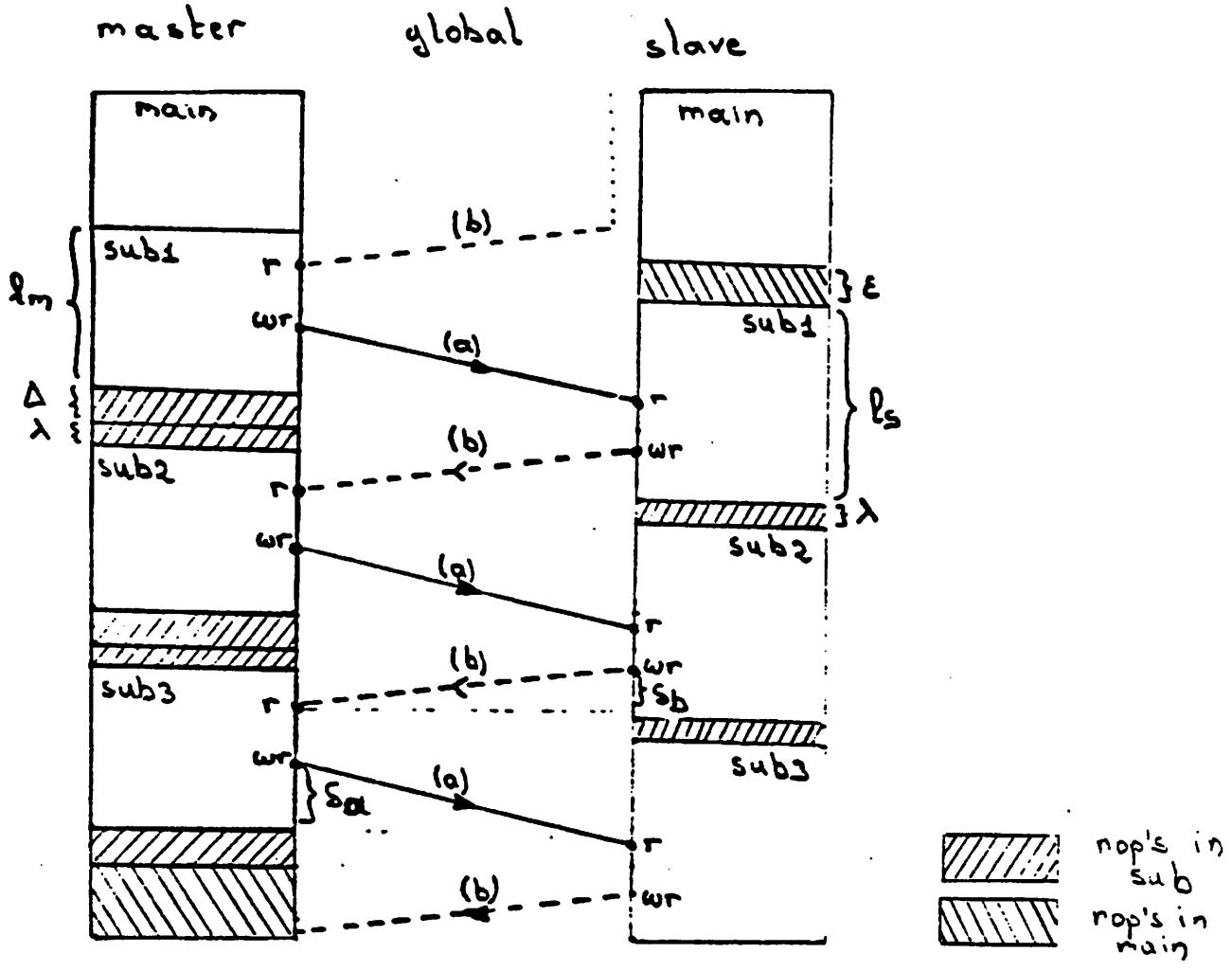
Demon reads in the design file and checks syntax and consistency. The data and control structures needed for emulation are set up. The exact timing of the interprocessor communications is evaluated and checked for consistency. The standard input (interactive commands) and output can be redirected in the normal way.

Warning messages will be issued only when the -w flag is set. Most of these warnings are concerned with possible i/o or interprocessor communication problems (e.g. when a global transfer causes a full sample delay).
A table, describing the timing of the different processors, can be obtained by setting the -c flag. Duration of main-program, subprogram, number of nop's, total sample execution time are displayed.
The -d flag generates a display of the rom-contents of the different processors, showing the status of the different control-lines for each micocommand.
A detailed overview of the interprocessor communications and the timing of the transfers can be obtained, setting the -i flag.
The -o flag suppresses the accumulator-overflow warnings.

3. The emulation and debug - mode

After the input and check of the design-file, the emulator requests for the datafiles (input and output). Demon remembers the datafile-names, used in the previous emulation.
After this, Demon enters the interactive mode, denoted with the prompt *Demon*> At that point, a set of commands are avialable for debugging and running:

- **run** *n (samples)* : execute n complete sample-cycles. Stop however when an intermediate breakpoint is encountered. When n is not defined, the emulation goes on until one of the input-data files is exhausted.
- **list** *start,stop* : list the input file from line *start* to line *stop*.
- **stop at** *line_nr* : installs a breakpoint at the specified design-file line. The specified line has to be an executable microcode-line.



b) The couple option

* avoids sample delay in global communication from master to slave (e.g. global (a)) == Sync-option
* makes also sure that global, written by slave in iteration (i) of the subroutine is read in iteration (i+1) of the master-subroutine (e.g. global b).

This option allows for a bidirectional communication between two processors in a synchronous way, using a minimum amount of hardware.

Realisation
- identical to sync
- Extra alignment is realized with λ extra nop's in subroutine of both master & slave

Example: See vocoder (example_manual)

Fig. 1b

- **cont** : resumes emulation after a breakpoint .
- **focus** *processor* : sets a pointer to the named processor. This processor is then considered as the default search processor for the *step, print and set* commands.
- **print** *variable* **of** *processor* : outputs the value of a global, local or register. The "of processor-name" is optional and is not needed for globals. For locals or registers however, the processor-name has to be defined if the processor to be examined is not the one which is in focus (see focus-command). Following registers (or flags) can be examined : mor, sor, acc, mbus, mir, ix, iy, pc, cc, eof, mof. The "print pipe" command gives an overview of the complete pipeline of the processor + the values of pc and index registers.
- **set** *variable* **of** *processor* : overwrites the value of a global, local or register. The "of processor-name" is optional and is not needed for globals. For locals or registers however, the processor-name has to be defined if the processor in question is not the one which is in focus (see focus-command). *Demon* displays the old value of the variable and asks for the new one.
- **trace** *register* **at** *line_nr* : trace the value of the named register, when the named micro-code line is executed. The defined line has to be executable.
- **step** *processor* : performs a single step on the microcode of the named processor. The processor-name can be omitted if the processor is in focus. The complete pipeline status is printed after completion of the step.
- **delete** *trace, break* : remove all traces (if *trace*-keyword), breakpoints (if *break*-keyword) or both (*no* keyword).
- **status** : displays a list of the currently set breakpoints and traces.
- **reset** : Resets the status of the processors to the initial values and rewinds the input-files. This allows the user to restart an emulation without a program-quit.
- **timing** : displays the timing tables of the processors (identical to the -c flag).
- **alias** *keyword expression* : performs an alias from expression to keyword. This allows the user to define his own keywords (which can be shorter than the standard set, provided by Demon). Alias without arguments shows the complete list of defined aliases. A permanent set of aliases can be defined in a *.emrc* file, located either in the current or the home directory of the user. These aliases are read in on the invocation of the program.

  Example : "alias p print pipe" generates a new keyword "p", which is equivalent to "print pipe". After this definition, "p of *processor-name* " is a perfectly legal construct.
- **shell** *command* : executes the defined cshell-command. If no command is defined, a new shell is fired up.
- **help** : gives an overview of the available commands.
- **quit**
- **end** : exits the emulator.

remark : the emulator assumes that all the input-data are in the right format and order (as defined in the global and io-specifications). Data is outputted in a way, conform to the global (word_length) and io (order)-specifications.

4. bugs

It is not yet possible to trace locals and globals. A format has to be defined to define this in a consistent way (conform with indexing).

Part 4 : Examples

1. Example 1 : 32 tap FIR lowpass filter

The first example contains the design file for a 32-tap fir lowpass filter (Figure 1). The main program handles in- and output of the data and some initialization. The sub-program implements a single tap (multiplication, addition and updating of the delay line). 32 iterations over this subprogram yield the complete fir-filter.
Note that the coefficients are stored as constants in the RAM-macrocell and that variable-variable multiplications are used in the tap-implementation. This resuts in a small micro-program, but is not very efficient in terms of execution time. Another possibility would be to hardwire the coefficients in the microcode using the canonical signed digit representation as done in Example 2.

```
/* design file for 32-tap FIR Lowpass Linear Phase Filter */
.global
  begin
  in<12>, out<12>, tap<12>;
  end

.io <12>
  begin
  in : signal_in;
  out : signal_out;
  end

.processor : one<16>
  begin
  .local
    begin
    dly[33], result;
    end

  .constant
    begin
    ./* tap weights scaled for 12 bit accuracy */
      weight[32] = 0,-1,2,5,-8,-12,18,25,-35,-47,62,84,-117,-174,301,919,
                   919,301,-174,-117,84,62,-47,-35,25,18,-12,-8,5,2,-1,0;
    end

.main_pr  /* initialization, in- and  output handling */
  begin
  r(result), mbus = in, le;                    /* input new sample */
  w(dly[32]), mbus = mor, out = mbus, acc := 0; /* output last result */
  w(result), le;                               /* clear result */
  end

.sub_pr <32>  /* implements one tap of the fir */
  begin
  rx(weight);
  rx(dly[1]), mbus = mor, tap := mbus;
  r(result), sor:=mor;    /* start multplication */
                    sor:=sor>1,acc:=mor + coef.sor,coef = tap;
                    sor:=sor>1,acc:=acc+coef.sor;
                    sor:=sor>1,acc:=acc+coef.sor;
```

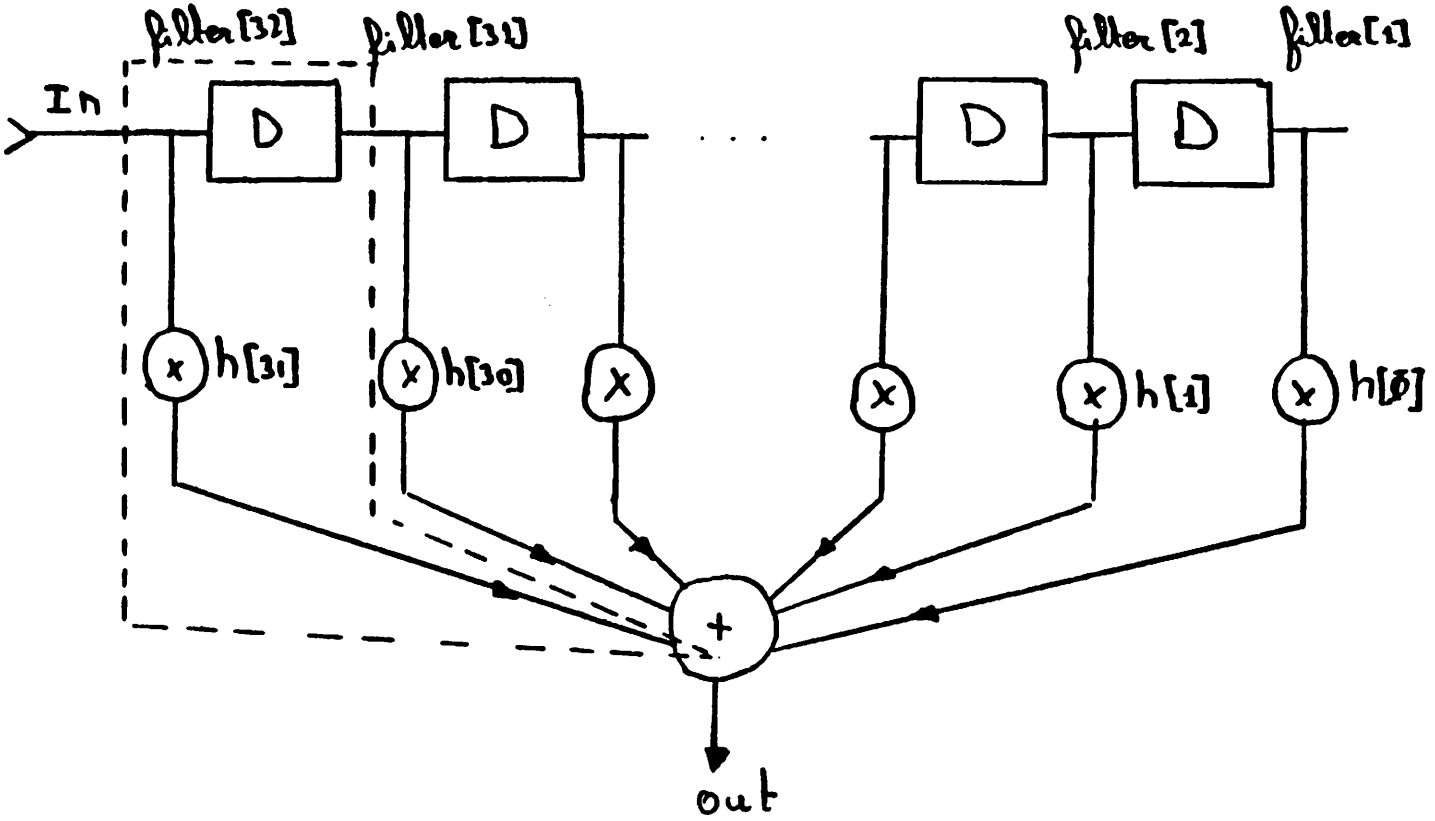


Figure 1 : 32 tap Fir lowpass

```
                    sor:=sor>1,acc:=acc+coef.sor;
                    sor:=sor>1,acc:=acc+coef.sor;
                    sor:=sor>1,acc:=acc+coef.sor;
                    sor:=sor>1,acc:=acc+coef.sor;
                    sor:=sor>1,acc:=acc+coef.sor;
                    sor:=sor>1,acc:=acc+coef.sor;
  rx(dly[1]),   sor:=sor>1,acc:=acc+coef.sor; /* update delay line */
  mbus = mor, le,          sor:=sor>1,acc:=acc+coef.sor;
  wx(dly),                 acc:=acc+coef.sor;
  w(result), le;
  end
end
```

2. Example 2 : Direct Form IIR bandpass filter

This example describes the implementation of the direct form implementation of a bandpass filter, using two second order biquad sections (Figure 2). The zeroes are implemented first, followed by the realization of the four poles. Note that in this example the coefficients are implemented in the canonical signed digit representation, which allows for a considerable reduction of the number of cycles/sample.

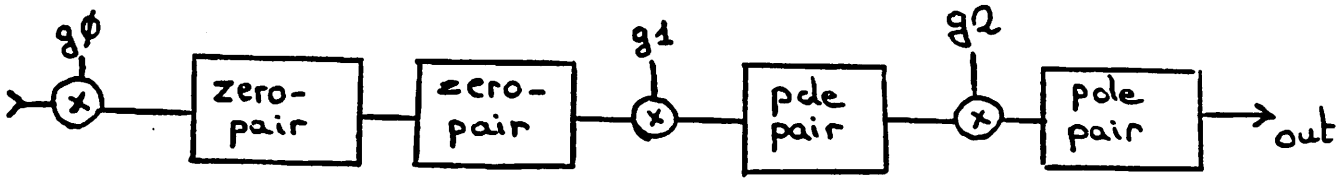
```
/* channel 7 of speech recognition filterbank - bandpassfilter */
.global
begin
   in <16>, out<16>;
end

.io
begin
   in : signal_in;
   out : signal_out;
end

.processor : bp <20>
/* implements first a set of four zeros, followed by two direct form sections */
begin
   .local
   begin
     zero1, zero2, zero3, zero4;
     bpf11, bpf12;
     bpf21, bpf22:
   end

   .main_pr
   begin
     /* implement 4-th order zero section (in form (1-z**-2)**2) *.
     /* also : scale input with g0 : 0.0- */
     r(zero1);
     mor := ~ mir, mbus = in, le, acc := mor;
     r(zero2), sor := mor >2, acc := acc;
     w(zero2), le, sor := mor, acc := sor;
     r(zero4), le, sor := mor, acc := acc + ~ sor;
     w(zero1), sor := mor, acc := acc;
     r(zero3), le, acc := acc + ~ sor;
     w(zero3), sor := mor, acc := acc;
     mor := ~ mir, le, sor := sor;

/*     1st bpf  g1= .0001
            a1= 1.011
            b1= -1.00-001
     2nd bpf  g2= .001
            a2= 1.1
            b2= -1.000—
*/
          r(bpf12), sor := mor >4, acc := sor;
          w(zero4), sor:= mor,  acc := sor, le;
                sor:= sor >3, acc:= acc + ~ sor;
          r(bpf11), sor:= sor >3, acc:= acc+ sor;
```

```
            sor:= mor, acc:= acc+ ~ sor, le, mbus= mor;
        w(bpf12), sor:= sor >2, acc:= acc+ sor;
        r(bpf22), sor:= sor >1, acc:= acc+ sor;
            sor:= mor, acc:= acc+ sor;
        w(bpf11), sor:= sor >4, acc:= ~ sor, le;
        mor:= ~ mir, sor:= sor >1, acc:= acc+ sor;
        r(bpf21), sor:= mor >3, acc:= acc+ sor;
        w(bpf22), sor:= mor, acc:= acc+ ~ sor, le, mbus= mor;
            sor:= sor >1, acc:= acc+ sor;
                acc:= acc+ sor;
        w(bpf21), le, out = mbus;
    end
end
```

Example 3 : Limiter Circuit

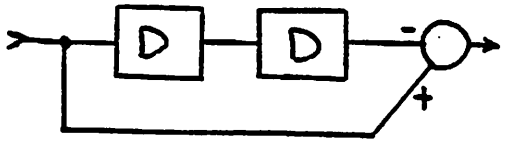
This example describes how the finite state machine can be used to perform decision making. The processor, described below, acts as a limiter circuit with the saturation characteristic of Figure 3. Two finite state machine commands are needed to implement this function, basically testing the sign bit of the accumulator after comparison of the signal with the upper and lower bounds.

```
.global
begin
   in<8>, out<8>;
end

.io
begin
   in : signal_in;
   out : signal_out;
end

.processor : limiter <8>
begin
   .local
   begin
     Result;
   end

   .constant
   begin
     Lower = -96;
     Upper = 64;
   end

   .fsm          /* finite state machine definition */
   begin
     SETIFLARGER : cc = sign; /* set cc if input > Upper */
     SETIFSMALLER : cc = !sign;      /* set cc if input < Lower */
   end

   .main_pr
   begin
     r(Result); /* output previous result */
     w(Result), mbus = in, acc := mor, le;
     r(Upper), sor := mor, out = mbus;
     r(Lower), sor := sor, acc := sor + mor; /* compare with Upper */
     r(Upper),          acc := sor + mor, SETIFLARGER;
                                /* compare with Lower */
     wc(Result), mbus = mor, le, SETIFSMALLER;
                          /* adjust if overflow */
     r (Lower);
     wc(Result), mbus = mor, le; /* adjust if underflow */
   end
end
```



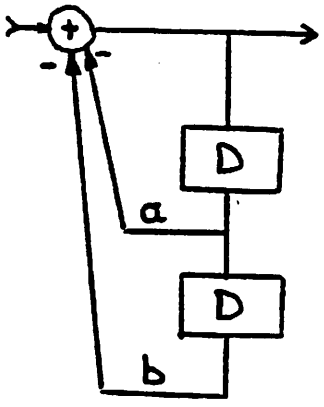
Block diagram

zero-pair:



$(1 - z^{-2})$

pole-pair



$$\frac{1}{1 + az^{-1} + bz^{-2}}$$

CSD (Canonical signed digit) notation:

$-1.00-001 = -1 + 1/8 - 1/64$

Fig. 2: Direct form bandpassFilter

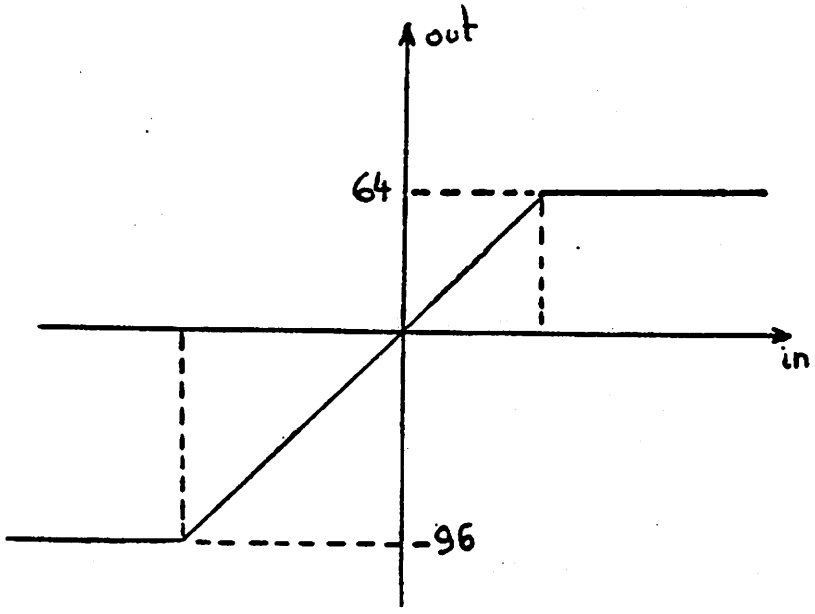


Fig. 3 : Limiter characteristic

3. Example 4 : LPC-10 vocoder

This example describes the implementation of an LPC-10 vocoder. (cfr. subsequent Figures). The speech input (to the analyser) and output (from the synthesizer) is performed over the parallel signal-bus, while the low rate speech parameters are transferred to a host proceesor via the host-interfacea (lattice coefficients, speech energy and pitch).

The algorithm has been divided over three processors in the following way :

processor 1 : filter

- Speech input + Preemphasis (before analysis filter)
- LPC10 analysis lattice filter
- LPC10 synthesis lattice filter + deemphasis

  Lowpass filtering of speech input before pitch detection

processor 2 : correlator

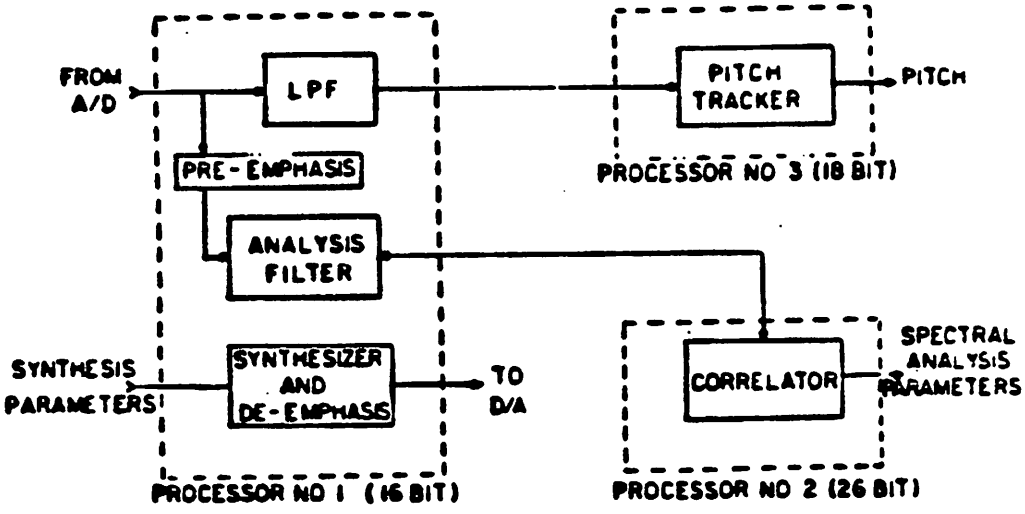
- implements the frame counter
- computes the excitation : using pitch and energy (from host-input). Generates unvoiced as well as voiced signal and selects one of them based on pitch value.
- computation of analysis filter coefficients (using Burg's method)

processor 3: pitch

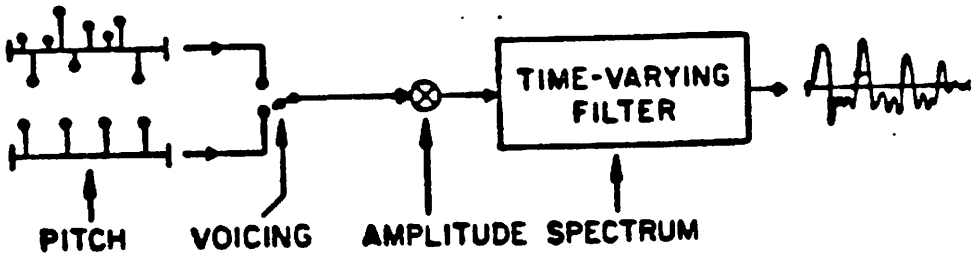
- Pitch computation using simplified Gold algorithm : six signals are computed and the pitch for each of them is updated every sample. Each sample one of the signals is selected and its pitch is compared with the six updated pitches. A score is computed. Every six samples, the pitch with the highest score is selected as the new pitch and compared with the voiced/unvoiced threshold.

  A full description of the vocoder implementation can be found in following reference :

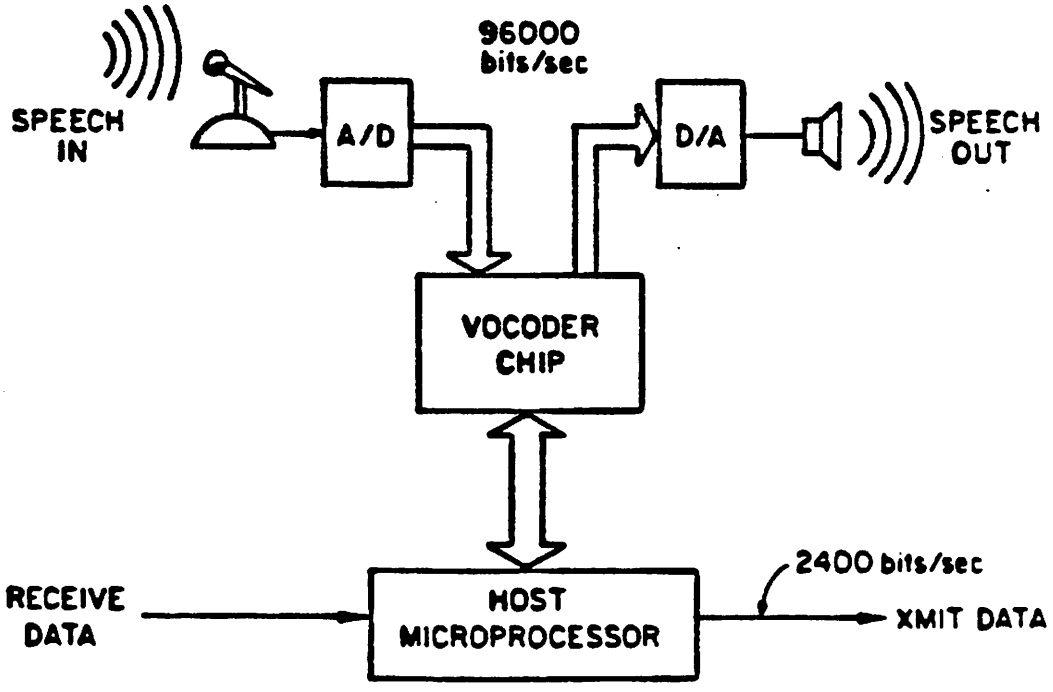
S. Pope, *Macrocell Design for Signal Processing, Chapter 1*, Ph.D. Dissertation, University of California, Berkeley, CA, December 1984



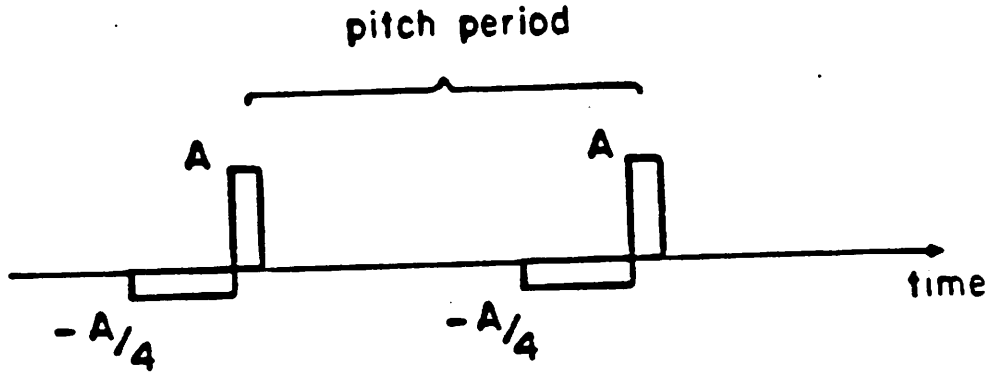
Multiprocessor organization



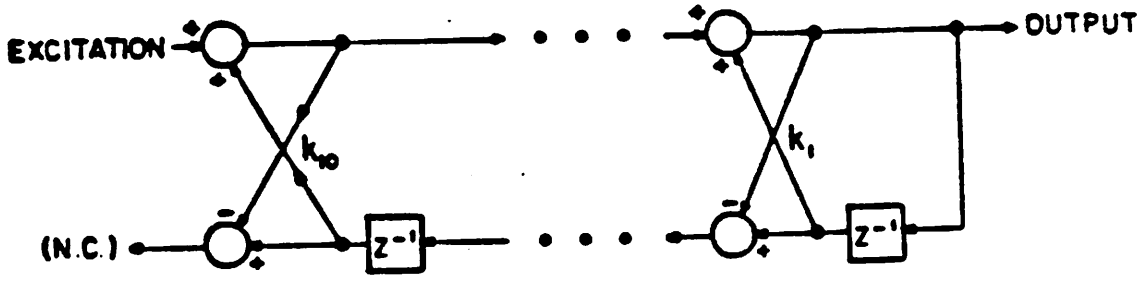
Vocoder speech model



Target system for the vocoder I.C.

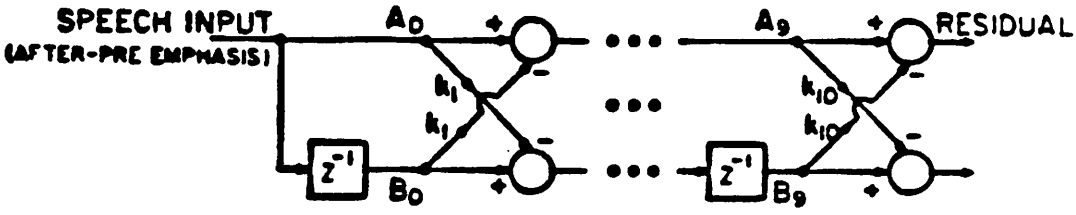


Voiced excitation waveform

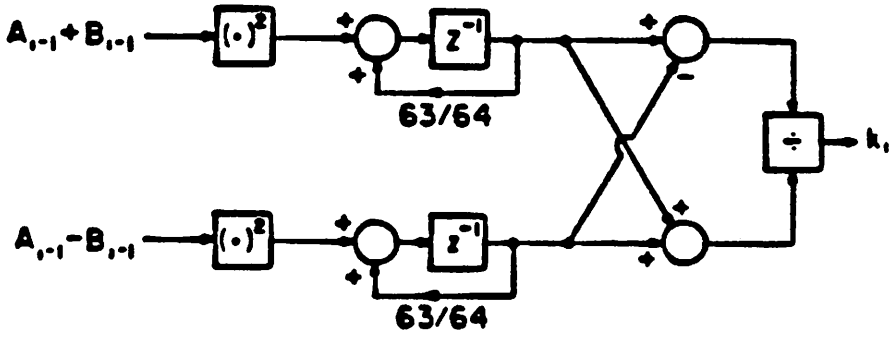


Lattice synthesis filter

Synthesis

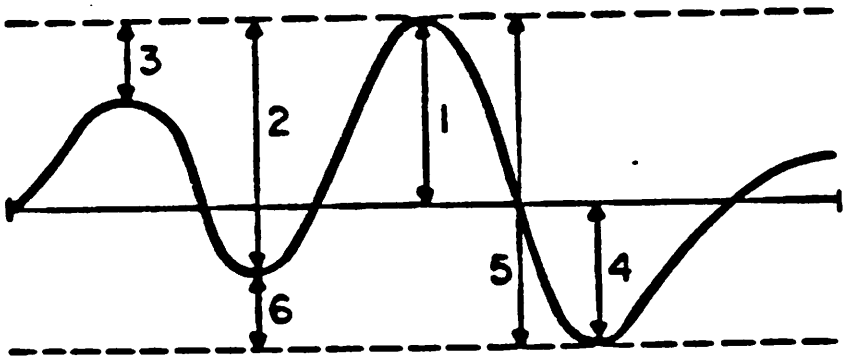


Analysis lattice filter

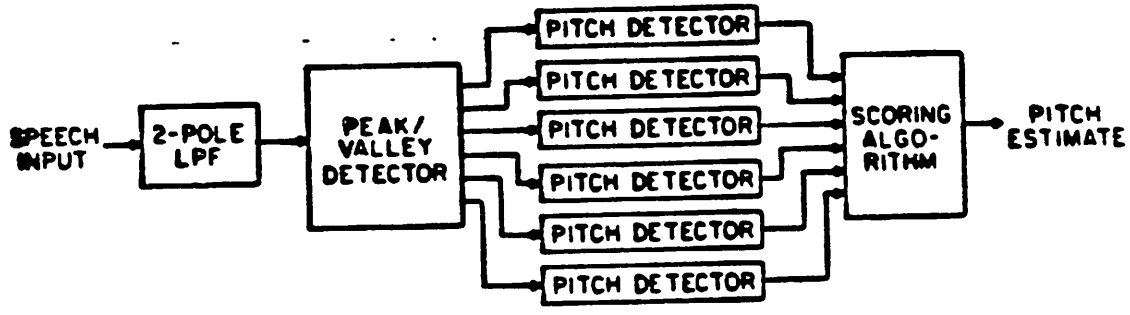


Correlator

Analysis



Six signals formed after peak-valley detection



Gold Pitch Tracker algorithm

Pitch Tracking

```
/* vocoder - full system */
/* the system is composed out of three processors :
 * filter, correlator and pitch_tracker
 */

.global /* interprocessor and i/o communications */
begin
   ka[10]<8>, ks[10]<8>;   /* only host i/o interconnections */
   ka_connect<8>, ka_shift<8>; /* ka-connection + multiplier */
   plus<16>, min<16>;
   speech_in<16>, speech_out<16>, residu<16>, excitation<16>;
   energy_out<24>;
   energy_in<16>;
   pitch_in<8>, pitch_out<8> : right_justified;
   low_pass<16>;
   square<13>;
end

.io <8> /* io-description */
begin
   energy_in, pitch_in, ks : host_in;
   energy_out, pitch_out, ka : host_out;
   speech_in : signal_in;
   speech_out : signal_out;
   residu : signal_out;
end

/*************************************************************************/
.processor : filter <16>  /* main_program : low_pass filtering and     */
                          /* emphasis + deemphasis of signals          */
                            /* subprogram : lattice filter : synthesis */
                          /* and analysis                              */
/*************************************************************************/
begin
   .local
   begin
      a, b[11], c, d[11], temp;
      e, f, g, h;
     ka_intern[11];
   end

   .main_pr <1>
   begin

     /* copy c to d[10], output residual, input speech (in e) */
     /* deemphasis filter */
     r(a);
     r(h), mbus = mor; /* residu = mbus; */
     r(c), sor := mor>3, acc := mor;
     w(d[10]), mbus = mor, sor := mor>1, acc := acc + ~ sor, le;
     r(e), acc := acc + sor;
     w(h), sor := mor, speech_out = mbus, le;

     /* preemphasis filter */
```

```
  sor := sor, acc := ~ sor;
  w(e), mbus = speech_in, sor := sor>2, acc := acc + ~ sor, le;
  sor := mor, acc := acc + sor;
  r(b[0]), sor := sor, acc := acc + ~ sor;
  w(temp), acc := acc + ~ sor, mbus = mor, le ;
  w(a), le;
  w(b[0]);

  /* Lowpass filters, first and second stages, input excitation */
  r(f);
  r(e), sor := mor>2, acc := mor;
  w(c), sor := mor>2, acc := acc + ~ sor, mbus = excitation, le,
     residu = mbus;
  r(g), acc := acc + sor;
  w(f), sor := mor>2, acc := mor, le;
  /* input last ka -coefficient and write in 10 */
  w(ka_intern[10]), mbus = ka_connect, sor := mor, acc := acc + ~ sor, le;
  acc := acc + sor;
  w(g), low_pass := mbus, le;
end

sub_pr <10>
begin
  /* implements analysis and synthesis lattice filter */

  /* compute ~ (a + b[i]) and ~ (a ~ b[i]) */
  r(temp);
  r(a), sor := mor;
  r(a), sor := sor, acc := sor + mor;
   mor := ~ mir, acc := ~ sor + mor, le;
  mor := ~ mir, sor := mor, le;
  rx(d[1]), sor := mor, acc := ~ sor;
  r(c), sor := mor, acc := ~ sor, plus := mbus;

  /* compute c = d(i+1)*ks + c */
  sor := sor>1, acc := mor + coef.~ sor, min := mbus, coef = ks;  /* sign
                                                                    bit */
  sor := sor>1, acc := acc + coef_sor;
  sor := sor>1, acc := acc + coef_sor;
  sor := sor>1, acc := acc + coef_sor;
  rx(ka_intern[1]), sor := sor>1, acc := acc + coef_sor;
  r(a), sor := sor>1, acc := acc + coef_sor, mbus = mor, ka_shift := mbus,
     ka := mbus; /* output ka to host_interface */
  r(temp), mbus = mor, sor := sor>1, acc := acc + coef_sor, le;
  mor := ~ mir, sor := mor, acc := acc + coef_sor;

  /* compute a = a - b(i)*ka */
  w(c), sor := sor>1, acc := mor + coef.~ sor, coef = ka_shift, le;
  sor := sor>1, acc := acc + coef_sor;
  sor := sor>1, acc := acc + coef_sor;
  sor := sor>1, acc := acc + coef_sor;
  sor := sor>1, acc := acc + coef_sor;
  r(temp), sor := sor>1, acc := acc + coef_sor;
  r(a), mbus = mor, sor := sor>1, acc := acc + coef_sor, le;
```

```
    mor := ~ mir, sor := mor, acc := acc + coef.sor;

    /* compute b(i+1) = b(i) - a*ka, update temp */
    mor := ~ mir, sor := sor>1, acc := mor + coef.~ sor, coef = ka_shift, le;
    w(a), mbus = mor, sor := sor>1, acc := acc + coef.sor,le;
    rx(b[1]), sor := sor>1, acc := acc + coef.sor;
    w(temp), mbus = mor, sor := sor>1, acc := acc + coef.sor, le;
    sor := sor>1, acc := acc + coef.sor;
    rx(d[1]), sor := sor>1, acc := acc + coef.sor;
    r(c), mbus = mor, sor := sor>1, acc := acc + coef.sor, le;
    mor := ~ mir, sor := mor, acc := acc + coef.sor;

    /* compute d(i) = d(i+1) - ks*c */
    mor := ~ mir, sor := sor>1, acc := mor + coef.~ sor, coef = ks, le;
    wx(b[1]), mbus = mor, sor := sor>1, acc := acc + coef.sor, le;
    sor := sor>1, acc := acc + coef.sor;
    sor := sor>1, acc := acc + coef.sor;
    sor := sor>1, acc := acc + coef.sor;
    sor := sor>1, acc := acc + coef.sor;
    sor := sor>1, acc := acc + coef.sor;
    /* input new ka-coefficient (from correl) for cycle i - 1 */
    acc := acc + coef.sor, mbus = ka_connect, wx(ka_intern), le;
    mor := ~ mir, le;
    wx(d), mbus = mor, le;
  end
end
```

```
/*****************************************************************/
.processor : correlator<26>  /* main_program : frame_counter and waveform -  */
                             /*                generator                     */
                             /* sub_program : correlator                     */
/*****************************************************************/
begin
  .local
  begin
    energ_temp;
    d[10], d[10];
    sample_counter;
     pitch_counter, pitch, voiced, unvoiced;
     random, gain;
  end

  .constant
  begin
    INCREMENT = 1;
    MASK = 32767;
    SIX = 6;
    FRAME_LENGTH = -90;
    MIDDLE_OF_FRAME = -70;
  end

  .fsm
  begin
    SET_IF_PLUS : cc = !sign;
    SET_IF_MINUS : cc = sign;
    MINUS_FIVE : cc = cc & !sign;
    CC_OR_MINUS : cc = cc | sign;
    EXOR : cc = !cc&!sign | cc&sign;
    EOF : eof = !sign, mof_flg = mof | (mof_flg & !eof);
    MOF : mof = !sign & !mof_flg;
  end

  .main_pr <1>
  begin
    r(sample_counter);
    r(INCREMENT), sor := mor;
    r(FRAME_LENGTH), sor := mor, acc := sor;
    r(MIDDLE_OF_FRAME), sor := mor, acc := acc + sor, SET_IF_MINUS;
    r(pitch_counter), sor := mor, acc := acc + sor, aip;
    w(sample_counter), sor := mor, le, acc := acc + sor, EOF;

    /* increment pitch_counter - compare with pitch */
    /* determine input-zone ( <5, =5, >5)
     * voiced = gain/4 if <5
     * voiced = ~ gain if 5
     * voiced = 0 if > 5
     */
    r(INCREMENT), sor := sor, MOF;
    w (pitch) , acc := sor + mor, mbus = pitch_in, le;
    w(pitch_counter), sor := mor, acc := 0, le;
    wc(sample_counter), sor := mor, acc := ~ sor + mor, le;
```

```
r(SIX), sor := sor, acc := 0, SET_IF_MINUS;
wc(pitch_counter), acc := sor + mor, le;
w(voiced), sor := mor, acc := acc, SET_IF_PLUS;
w(gain), mbus = energy_in, acc := sor + acc, le;
wc(voiced), sor := mor>2, mbus = mor, le, MINUS_FIVE;
r(random), acc := ~ sor;
wc(voiced), sor := mor, le;

/* generate random number ( for unvoiced case ) */
/* technique : exor lsb with lsb -1 and circular shift */
/* first, we have to check that the initial random num-
 * ber is different from zero : this tends to block the
 * generator .
 */
 r(INCREMENT), acc := sor;
 acc := mor&acc;
 r(random), sor := mor, acc := acc, le;  /* mor = -1 */
 w(unvoiced), sor := mor>1 , acc := acc + sor;
 r(INCREMENT), acc := sor, SET_IF_PLUS; /* check lsb */
 acc := mor & acc, le;
 mor := ~ mir, sor := mor, acc := acc;
 sor := mor, acc := acc + sor;
 r (MASK), acc := ~ sor, EXOR; /* exor lsb & lsb -1 */
 sor := mor, acc := acc & mor;
 w(random), sor := sor, acc := acc + ~ sor, le; /* left shift */

/* check if random number is zero */
/* if yes, transform to largest negative number */
r(unvoiced), acc := acc + sor, le;
sor := mor, CC_OR_MINUS;

/* adjust unvoiced to gain/8 or -gain/8 */
r(gain), acc := sor + mor;
wc(random), sor := mor>3, SET_IF_MINUS;
r(pitch), acc := sor;
sor := mor, le;
w (unvoiced), acc := sor + mor;

/* make voiced-unvoiced decision : if pitch = 0 :unvoiced */
/* output final waveform to excitation */
wc (unvoiced), mbus = mor, SET_IF_PLUS, le;
r (voiced);
wc (unvoiced), mbus = mor, le;
r (unvoiced);
excitation := mbus, mbus = mor, le;

/* send last energy_value to host_out */
r (energ_temp);
mbus = mor, energy_out := mbus;
end

sub_pr <10>
begin
 /* square plus-signal */
```

```
  mor := ~ mir, mbus = plus, le;
  sor := mor, mbus = mor, square := mbus;
  sor := sor>1, acc := coef.~ sor, coef = square;
  sor := sor>1, acc := acc + coef.sor;
  sor := sor>1, acc := acc + coef.sor;
  sor := sor>1, acc := acc + coef.sor;
  sor := sor>1, acc := acc + coef.sor;
  sor := sor>1, acc := acc + coef.sor;
  sor := sor>1, acc := acc + coef.sor;
  sor := sor>1, acc := acc + coef.sor;
  sor := sor>1, acc := acc + coef.sor;
  sor := sor>1, acc := acc + coef.sor;
  sor := sor>1, acc := acc + coef.sor;
  mor := ~ mir, sor := sor>1, acc := acc + coef.sor, mbus = min, le;
  sor := mor, acc := acc + coef.sor, mbus = mor, square := mbus;

  /* square minus-signal */
  sor := sor>1, acc := coef.~ sor, coef = square, le;
  sor := sor>1, acc := acc + coef.sor;
  sor := sor>1, acc := acc + coef.sor;
  sor := sor>1, acc := acc + coef.sor;
  sor := sor>1, acc := acc + coef.sor;
  sor := sor>1, acc := acc + coef.sor;
  sor := sor>1, acc := acc + coef.sor;
  sor := sor>1, acc := acc + coef.sor;
  sor := sor>1, acc := acc + coef.sor;
  sor := sor>1, acc := acc + coef.sor;
  sor := sor>1, acc := acc + coef.sor;
  mor := ~ mir, sor := sor>1, acc := acc + coef.sor;
  rx(c), sor := mor, acc := acc + coef.sor;

  /* start lowpass filtering of squared signals */
  mor := ~ mir, sor := mor>6, acc := mor + ~ sor, le;
  rx(d), sor := mor, acc := acc + ~ sor;
  wx(c), sor := mor>6, acc := mor + ~ sor, le;
  sor := mor, acc := acc + ~ sor;
  wx(d), sor := sor>1, acc := sor + acc, le;

  /* accumulator contains now D-C - calculate now (C-D)/2 & (C+D)/2 */
  sor := mor>1, acc := ~ sor, le;
  mor := ~ mir, acc := ~ sor + acc; /* (C+D)/2 in acc */
  w (energ_temp), sor := mor>1, le;
  sor := mor>1, acc := isort;

  /* start division */
  sor := sor>1, acc := sor + acc, aip; /*sign bit av. */
  sor := sor>1, acc := sor + acc, aip;
  sor := sor>1, acc := sor + acc, aip;
  sor := sor>1, acc := sor + acc, aip;
  sor := sor>1, acc := sor + acc, aip;
  sor := sor>1, acc := sor + acc, aip;
   acc := sor + acc, aip;
  ka_connect = quot; /*division finished */
end
```

end

```
.processor : pitch <18>
/* implements the GOLD -pitchtrack algorithm (can be improved) */

begin
   .local
   begin
     thresh[6], ppc[6], pp[7], lpp[6], signal[6];
     ls, lp, lv, score, topscore, pitch, winner;
   end

   .constant
   begin
     TWO = 2;
     BLANK = 24;
     VOICED = 9;
     WINDOW1 = 8;
     WINDOW2 = -14;
   end

   .fsm /* finite state machine description */
   begin
     SET : cc = !sign;
     AND_MINUS : cc = cc & sign;
      APV : cc = cc & (!ix<1>&!slp&lsp | ix<0>&slp&!lsp);
     VPE : cc = iy[0];
     SIP : cc = !slp & lsp;
     SIV : cc = slp & !lsp;
     SSL : lsp = slp, slp = !sign;
   end

   .main_pr <6>
   begin
     /* finish up calculation of previous time */
     r (score);
     r (topscore), sor := mor>1;
     sor := mor>1, acc := sor;
     r (score), acc := acc + ~ sor;
     ry (pp), acc := mor, SET;
     wc (topscore), acc := mor, le;
     wc (winner), le;

     /* if VPE (every six samples) compare topscore to constant VOICED,
        set pitch to either winner or zero and reset topscore        */
     r (winner);
     r (VOICED), acc := mor, VPE;
     wc (pitch), sor := mor, le;
     r (topscore), sor := sor, acc := 0;
     wc (topscore), acc := ~ sor + mor, le;
     r (signal), acc := 0, AND_MINUS;
     wc (pitch), acc := mor, le, SIP;
     wc (lp), le, SIV;
     wc (lv);
     w (ls);
```

```
 /* beginning of new sample . Compare new signal to last and set SSL,
    clear score */
 w (signal), sor := mor>1, mbus = low_pass, acc := 0, le;
 w (score), sor := mor>1, acc := sor, le;
 r (signal), acc := acc + ~ sor;
 mor := ~ mir, mbus = mor, SSL, le;

  /* send pitch to output port . Form signal[1] through signal[5] . */
 r (lv), sor := mor>1, mbus = mor, le;
 w (signal[1]), sor := mor>1, acc := sor;
 r (lp), sor :=mor>1, acc := sor + acc;
 w (signal[3]), sor := mor>1, acc := ~ sor, le;
 w (signal[2]), acc := sor + acc, mbus = mor, le;
 r (pitch), le;
 w (signal[5]), sor := mor>1;
 w (signal[4]), mbus = mor, acc := sor, le;
 pitch_out := mbus;
end

sub_pr <6>
begin
  /* increment pitch_counter by two and set condition code
     if greater than BLANK . Conditionally decay threshold .
     And condition code with peak-valley indicator .        */
  rx (thresh);
  rx (ppc), mbus = mor, le;
  r (TWO), sor := mor;
  r (BLANK), sor := mor, acc := sor;
  wx (thresh), sor := mor, acc := sor + acc;
  wx (ppc), sor := mor, acc := ~ sor + acc, le;
  r (lp), sor := sor>6, acc := ~ sor, SET;
  rx (signal), sor := sor>1, acc := acc + sor, mbus = mor,le;
  w (lp), sor := mor>1, acc := acc + sor;
  wxc (thresh), sor := sor, acc := acc + ~ sor, le, APV;

  /* And condition code with result of (signal > threshold)
     comparison, Conditionally update thresh, lpp, ppc, pp */
  rx(pp), acc := sor, AND_MINUS;
  wxc (thresh), acc := mor, le;
  wxc (lpp), le;
  rx (ppc), acc := 0;
  wxc (ppc), acc := mor, le;
  wxc (pp), le;
  r (lv);
  w (lv), mbus = mor, le;

   /* add contribution for this channel to score of current cand.
      do three window comparisons with current candidate and pp,
      lpp and pp + lpp . In each case, increment score with 2 if true */
  r (pp[0]);
  w (pp[6]), mbus = mor, le;
  ry (pp[1]);
  rx (pp), sor := mor>1;
  r (WINDOW1), sor := mor>1, acc := ~ sor;
```

```
      r (WINDOW2), sor := mor>1, acc := acc + sor;
      r (TWO), sor := mor>1, acc := acc + sor;
      r (score), sor := mor, acc := acc + sor, SET;
      ry (pp[1]), acc := mor + sor, AND_MINUS;
      wc (score), sor := mor>1, le;
      rx (lpp), acc := ~ sor;
      r (WINDOW1), sor := mor>1, acc := acc;
      r (WINDOW2), sor := mor>1, acc := acc + sor;
      r (TWO), sor := mor>1, acc := acc + sor;
      r (score), sor := mor, acc := acc + sor, SET;
      ry (pp[1]), acc := mor + sor, AND_MINUS;
      wc (score), sor := mor>1, le;
      rx (lpp), acc := ~ sor;
      rx (pp), sor := mor>1, acc := acc;
      r (WINDOW1), sor := mor>1, acc := acc + sor;
      r (WINDOW2), sor := mor>1, acc := acc + sor;
      r (TWO), sor := mor>1, acc := acc + sor;
      r (score), sor := mor, acc := acc + sor, SET;
      acc := mor + sor, AND_MINUS;
      wc (score), le;
    end
end

constraints
begin
  couple : filter, correlator;
end

/* end vocoder description */
```