Instruction Set Definition and Instruction Selection for ASIPs

Johan Van Praet Gert Goossens Dirk Lanneer Hugo De Man*

IMEC, Kapeldreef 75, B-3001 Leuven, Belgium
* Professor at K.U. Leuven, Belgium

Abstract

Application Specific Instruction set Processors (ASIPs) are field or mask programmable processors of which the architecture and instruction set are optimised to a specific application domain. ASIPs offer a high degree of flexibility and are therefore increasingly being used in competitive markets like telecommunications. However, adequate CAD techniques for the design and programming of ASIPs are missing hitherto. In this paper, an interactive approach for the definition of optimised microinstruction sets of ASIPs is presented. A second issue is a method for instruction selection when generating code for a predefined ASIP. A combined instruction set and data-path model is generated, onto which the application is mapped.

1 Introduction

Application Specific Instruction set Processors (ASIPs) are in between custom architectures and commercial programmable DSP processors. They allow field and mask programmability but are targeted to a certain class of applications as to limit the amount of hardware (area and power) needed. Consequently, ASIPs are often the best choice for embedded applications. To increase performance of such an ASIP, custom hardware accelerator data-path(s) can be added, which makes the ASIP a heterogeneous IC architecture [6].

The small number of algorithms to be mapped on an ASIP does not justify the effort of writing a compiler for each target architecture. In practice, assembly code therefore is often written manually, which is too high a cost. The solution is a *retargetable compiler*, with as additional advantage that it supports late changes on the instruction set.

This paper focusses on the instruction selection task in such a retargetable compiler. A second part of the paper shows how our instruction selection method can be used together with an application analysis tool for micro-instruction set definition. The techniques are implemented as a part of the synthesis and code generation system "CHESS".

2 Traditional instruction selection

An ASIP is usually specified by its instruction set and an abstract description of its data-path. The detailed description of the data-path with *all* connections is normally not available, nor is a description of the controller or micro-sequencing logic.

Traditional instruction selection techniques use tree pattern matching [1, chapter 9]. The set of *template patterns* is (manually) extracted from the instruction set and the graph representing the intermediate code of the application is covered by these patterns. For machines with a set of interchangeable general-purpose registers, tree pattern matching based on the dynamic programming method ensures optimal code. Several tools are available to perform this tree pattern matching, given an enumeration of the template patterns (e.g. [5]).

In contrast with early processors where each instruction resulted in one template pattern, horizontally microcoded processors and also recent (RISC) processors have more orthogonal instruction sets with parallelism in their instructions. In this case, using template patterns which each cover a complete instruction, has several drawbacks. The resulting patterns are rather large, which decreases the probability of matching; the number of possible patterns also grows too high, which slows the pattern matcher down. The traditional approach for microcoded processors therefore is to create a template pattern for each processor activity. Tree pattern matching then results in vertical microcode (without parallelism) and parallelism is introduced in the code by compaction (scheduling) algorithms, resulting in so-called horizontal microcode [7].

There are however some problems in directly applying these techniques to ASIPs :

- Most ASIPs are microcoded processors with parallelism in their instructions. But how can a processor activity, with the right granularity to allow efficient pattern matching (not too big patterns) and efficient compaction (not too small patterns), be determined? Deriving good patterns by hand is too much of an effort for a target architecture that will only be used to map a few algorithms on.

- The dynamic programming method (and some other traditional code generation methods) can only generate optimal code for an architecture which incorporates a general-purpose register-set. Moreover patterns used in pattern matching must be trees. For an ASIP this is often *not* the case, especially if the application domain is real time DSP. These ASIPs have few

0-8186-5785-5/94 $03.00

registers which are distributed over the architecture. Re-convergent paths (or even cycles incorporating a pipeline register, e.g. for multiply-accumulates) are no exception in ASIPs. This means that *graph* pattern matching is in fact needed rather than *tree* pattern matching. For effective code generation, the *connectivity* between registers and functional units has to be taken into account.

Recently, work has been done at e.g. BNR [11] to extend the pattern matching techniques in order to be applicable to ASIPs. We however propose another technique.

3 Instruction selection by bundling

We will use the following terminology (which is more extensively defined in [7]): *Directly coupled* micro-operations are primitive processor activities which pass data to each other through a *transitory* data storage resource, e.g. through a wire or a latch. A *bundle* is a *maximal* sequence of micro-operations in which each micro-operation is directly coupled to its neighbours. As a consequence (see section 3.1), a bundle must match a complete group of functional units (FUs) that is directly connected to addressable registers (via direct wires, buses, tristate drivers, or multiplexers). Examples illustrating the bundle concept will be given in section 3.2. Remark that a bundle can contain a pipeline register which is non-addressable and thus a transitory data storage resource.

We believe that a bundle has the right granularity to split the problem of instruction selection.

- A first subtask, called bundling, consists in grouping all *data flow operations* of a complete application in a minimal number of *bundles*. Each of the resulting bundles must be part of a processor instruction later on and thus fit in an instruction format. Data routing (register allocation) then changes the bundles into register transfers by annotating them with multiplexer settings, bus-driver settings and register addresses [10].

- The second subtask of instruction selection consists in putting the *control flow operations* and the *bundles* together into micro-instructions. This will be done during scheduling which means that the scheduler must know about the instruction formats and about some pipeline aspects which cause restrictions in the ordering of instructions.

3.1 A combined instruction set and data-path model

The approach followed in this paper is to use a combined instruction set and data-path model of the ASIP for instruction selection rather than a set of template patterns. This model does not necessarily correspond to the real ASIP data-path.

An example specification of an ASIP data-path is shown in figure 1 and its instruction set for arithmetic

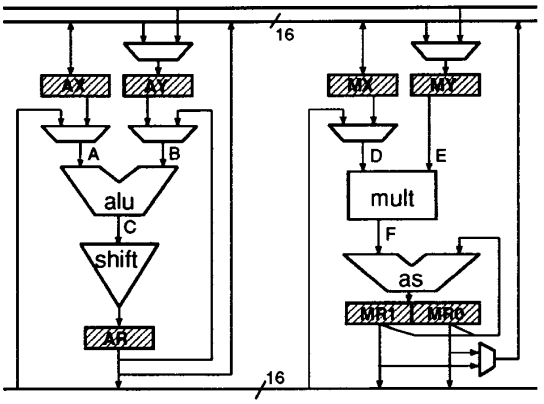

Figure 1: Data-path specification for the example.

format 1					
format		opcode		operand addresses	
0	0	00: + @alu		00: AX	0: AY
		01: − @alu		01: AR	1: AR
		10: and @alu		10: MR1	
		11: or @alu		11: MR0	
	1	0: + @alu	0: $\gg$1 @shift		
		1: − @alu	1: $\ll$1 @shift		

format 2			
format	opcode	operand addresses	
10	00: $*$ @mult, + @as	00: MX	0: MY
	01: $*$ @mult, − @as	01: AR	1: "1"
	10: $*$ @mult	10: MR1	
	11: + @as	11: MR0	

format 3		
format	opcode	shift-factor
11	00: AX	$\gg x$ @shift
	01: AR	with $x = [-4..3]$
	10: MR1	
	11: MR0	

Table 1: Instruction set specification for the example.

operations is given in table 1. A more complete instruction set would also contain (conditional) jump operations for implementation of the control flow and load/store operations for data storage in memory.

The data-path allows all combinations of operations on the ALU, the shifter, the multiplier and the adder-subtractor but the instruction set only encodes very few of them, using three instruction formats. The remaining combinations are not allowed because of *encoding restrictions*. Encoding restrictions are introduced to limit the number of bits in the instruction word, in this case to 7 bits, register addresses included. Allowing parallelism between the ALU and the multiply-accumulate structure would cost a lot of bits and would only rarely be used because of the interconnection (bus conflicts!). One or two load/store operations can usually occur in parallel with the arithmetic operations.

The instruction set is specified by the user in the nML language [4, 2]. In nML an instruction set is hierarchically described as an attributed grammar. The

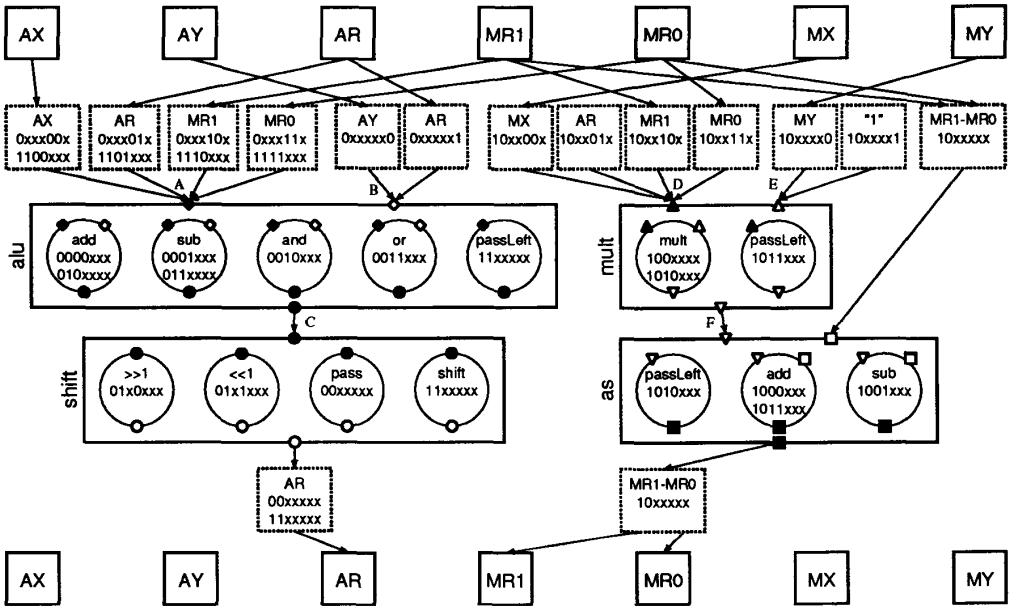

Figure 2: Combined instruction set and data-path model for the example of figure 1. Inside a rectangle different terminal symbols are used to represent connections : equal symbols are on the same net.

attributes belonging to a part of an instruction describe e.g. its associated actions (with in our case also the used hardware resources), its binary encoding, its assembly mnemonics,...

From the nML description the combined instruction set and data-path model is generated. For the example it is depicted in figure 2. It is generated as follows :

- A node is generated for each FU in the data-path. Two nodes are generated for each register; one for a read action and one for a write action. These nodes are the solid rectangles in figure 2.
- The direct interconnections between the FUs are added.
- The FU nodes are annotated with the operations they can perform and the associated instruction settings. This information is found in the instruction set description.
- Abstraction is made of the physical implementation of the connections between FUs and registers. These connections are modelled by simple point to point connections which pass through "register-select-blocks" (dashed rectangles). Such a block contains the instruction settings that enable the connection. A lot of register connections are missing in this example because the load/store instructions are left out.

A bundle is simply a path *between* register-select-blocks. A register transfer is a path from (a) register(s) on the top of the model to a register on the bottom, thus including register-select-blocks. However, while searching a path in the model, the compatibility of the instruction settings on the path must be checked (see example in section 3.2).

3.2 The bundling technique

Our bundling technique is an extension of the mapping technique described in [8, 9]. Before bundling the application is translated in a control data flow graph (CDFG) - see section 4. In the CDFG each operation node is annotated with all FUs from the combined instruction set and data-path model it can be executed on. If the model allows two nodes to be directly coupled (there is a path between them without conflicting instruction settings), the edge connecting these nodes is annotated with that combination. This is shown in figure 3 for a small data flow graph. The first add-operation can be performed on the ALU or on the adder-subtractor and for each there are two possible instruction settings. The $\gg$1-operation can be mapped onto the shifter, also with two possible instruction settings. In the combined instruction set and data-path model, it can be seen that an add-$\gg$1 bundle can be mapped on the ALU and shift FUs. Indeed, there is a connection between the two FUs and the intersection of their instruction settings needed for both operations of the bundle is not empty.

In a larger example some edges may have been annotated with several coupling alternatives. In selecting a coupling alternative, other alternatives on neighbouring edges can be deleted. So clever heuristics have then to be applied to select the right alterna-

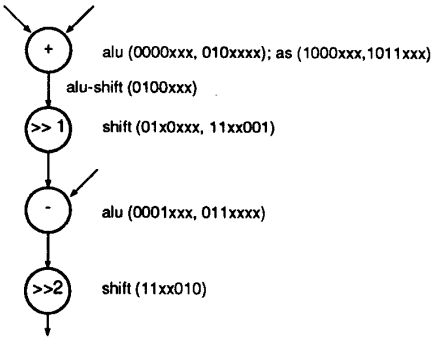

Figure 3: DFG annotated with mapping possibilities.

tive. These heuristics are an extension of the ones described in [8, 9]. In this way the algorithm *incrementally* combines operation nodes into sets of coupled micro-operations until the eventual bundles are obtained.

For our example the bundles are : [+, $\gg$1 : alu-shift(0100xxx)], [− : alu(0001xxx, 011xxxx)] and [$\gg$2 : shift(11xx010)]. These bundles are then extended by the data routing tool to complete register transfers. In our example the register transfers could be : [AR = (AX + AY) $\gg$1 : (0100000)], [AR = AR − AY : (0001010)] and [AR = AR $\gg$2 : (1101010)].

3.3 Advantages of our approach

Using the bundling technique with a combined instruction set and data-path model instead of using the more traditional pattern matching techniques has following advantages :

- The patterns must not be enumerated in advance, but are generated while performing the bundling.
- The matched patterns need not to be trees. All graph patterns are possible. See [8, 9].
- The bundling algorithm allows matched patterns to cross basic block boundaries (loop and condition boundaries) and performs the necessary operation duplications [8, 9].
- The model reflects both the instruction encoding and the connectivity between FUs and registers. Consequently it can be adopted by all tools in the retargetable code generator, not only the bundling tool but also the data-routing [10] and scheduling tool.

3.4 Similar approaches

The idea of creating a data-path model which incorporates all instruction restrictions was found by combining some ideas of the group working on the CBC compiler [4] and the group working on the MSSQ compiler [12, 13].

In the CBC approach nML is compiled into a combined instruction set and data-path model similar to the one explained above. As they do not use a data-path description as input, the compilation is more complex [2]. Their approach however differs from ours in the fact that they subsequently derive all possible patterns from that model and use the tree pattern matching tool described in [5]. Some heuristics are implemented to be able to handle graphs [3].

In the MSSQ compiler the combined instruction set and data-path model is derived from a detailed data-path description which also contains the controller and decoder description of the ASIP. The instruction annotations in the model are called I-nodes. Operations on I-nodes (and-ing and or-ing) are represented by I-trees [12]. An application is mapped on a statement by statement basis onto that model, each time directly generating complete transfers.

4 Instruction set definition

Because the tools in our retargetable compiler work on a combined instruction set and data-path model we can perform instruction set definition for a new ASIP at the data-path level. This is done in an interactive way, based on statistics obtained from an *analysis tool* and the *bundling tool*.

To define the instruction set of an ASIP intended for use in a certain application class, representative application parts are first selected. Then initial data-path parts are selected out of a library, based on the statistics obtained from the analysis tool. These parts are then iteratively updated — with as most important criteria area/performance trade-offs — and finally instruction encodings are defined.

This will be explained by a case study. Suppose that we want to define the instruction set for an ASIP in the application domain of speech recognition. One of the algorithms to be performed can then be the Pitch Extraction algorithm described in [14]. As example we will take the "sieve detection" function of that algorithm.

4.1 Initial data-path parts

We have developed a tool called *analyse* that extracts statistical design information from the CDFG description. *Analyse* can be called at multiple levels in the design trajectory. Initially *analyse* gives us an overview of all operations in the application, together with some frequently occurring operation sequences, which are good candidates to be directly coupled. This output is summarised in table 2. Based on this table we define the data-path parts of figure 4: The multiplication (by a constant) and the division will be expanded into add/sub/shift operations on a predefined data-path part. The sequence *"add-eq"*, which is part of the implementation of the loop counters and address calculation, has a very high occurrence so we decide to define a data-path part for it. Now all operations in the input algorithm can be expanded by our expansion tool in order to fit on the data-path parts. After expansion, *analyse* is called again to refine the statistics. For brevity, we only list the statistics on operation *sequences* in table 3.

At this point the *bundling* tool (see section 3) can be called. The data-path parts in figure 4 will not be

operation	word-length	occur-rence	candidate FUs
addition	6	2600	full adder
	12	640	
	16	320	
multConst	16	1920	multiplier adder_shifter
equality	6	2280	comparator, full adder
grtEq ($\geq$)	8	40	comparator, full adder
	16	960	
grt ($>$)	16	640	comparator, full adder
division	5	40	adder_shifter

sequences	occ.
add - eq	2280
div - grtEq	40
mult - grtEq	640
mult - grt	640
add - grtEq	320

Table 2: Output of *analyse* for input description.

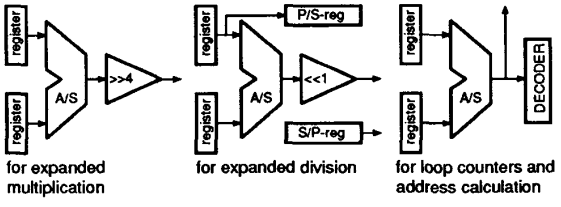

Figure 4: Initial data-path parts.

interconnected without an intermediate (addressable) register so no bundle will occupy more than one data-path part. No encoding restrictions are defined yet. The bundling tool gives the following statistics. Every pair of adjacent operations that is directly coupled, is printed in a list with its number of occurrences. Another list contains pairs of adjacent operations that are not (directly) coupled because of the data-path parts. The number of directly coupled edges, the number of non-coupled edges and their ratios to the total number

sequences	occ.	expansions of :
sequences of 2 operations		
add - eq	2280	
shift - add	1280	multiplication
shift - sub	640	
add - sub	960	mult./add - grt/grtEq
sub - sub	640	multiplication - grt/grtEq
sequences of 3 operations		
shift - add - sub	640	multiplication - grt/grtEq
shift - sub - sub	640	

Table 3: Operation sequences after expansion, for initial data-path parts.

directly coupled edges		
sequence	occurrence	
add - eq	2280	
non-coupled edges		
sequences	occurrence	expansions of :
shift - add	1280	multiplication
shift - sub	640	
add - sub	960	multConst/add - grt/grtEq
sub - sub	640	multConst - grt/grtEq

Table 4: Feedback of bundling for initial data-path parts.

of edges considered during bundling are also printed. A part of the statistics is summarised in table 4.

With these data-path parts 6040 edges (or 50.2 %) are directly coupled and 6000 edges (or 49.8 %) are not coupled. If a full cross bar interconnection scheme with register files of unlimited size is assumed between the data-path structures, the CDFG can be scheduled in 6241 cycles.

4.2 Improvement of the data-path parts

Table 4 shows that the operations resulting from the expansion of the multiplications are not coupled. If we want to achieve this, the shifter should be placed before the adder. With this done, the ratio of coupled edges increases to 66 % and the schedule length decreases to 4961 cycles.

Table 3 also contains some sequences of 3 operations. In these sequences the last subtraction comes from the relational operations $>$ and $\geq$ (compare with table 2). If we incorporate a comparator into the left-hand data-path part of figure 4, we obtain the data-path part of figure 5. We now have to perform a new expansion of the original application algorithm on the new data-path parts. The ratio of coupled edges then becomes 91 % and the schedule length 3961 cycles.

4.3 Combining the data-path parts into an instruction set

When *analyse* is given a scheduled graph as input, it provides occupation ratios and a table with the occupation patterns for each FU. Figure 6 shows the occupation patterns for the adder-subtractor in each of the data-path parts. When examining occupation patterns for all FUs we see that there is only a small overlap between the occupation patterns of the division data-path part and the other ones. In fact, to save area, the division and multiplication data-path parts can be combined into the one depicted (on the left) in figure 7. The operations from the multiplication and

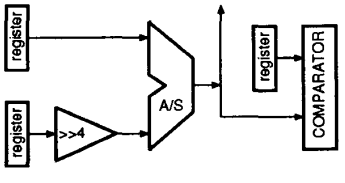

Figure 5: Shift-Add-Comp data-path part.

```
loop/add-subtractor       [xxxxxx.x.........x...x....]
multiplic./add-subtractor [.xx.xx.x..............x.x]
division/add-subtractor   [........x.xxxxxxxxxxx....]
("x" : resource is occupied ; "." : resource is free)
```

Figure 6: Occupation patterns for each add-subtractor in data-path parts.

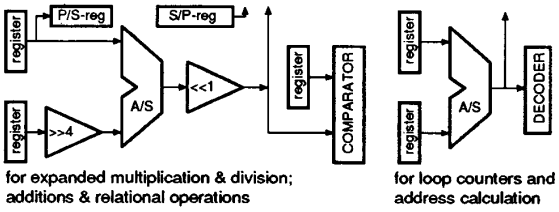

Figure 7: Final proposal for data-path parts.

the division expansion will thus be coded in different micro-instructions. Performing bundling on the parts of figure 7 gives the same results as previously. The two remaining data-path parts should not be merged any further because they are heavily used in parallel.

Out of these observations we can derive two instruction formats needed to control the data-path parts of figure 7 for this CDFG. The first format contains two orthogonal parts, controlling simple shifts/additions/subtractions (e.g. expanded multiplication) and the address/counter operations respectively. The second format controls the expansion of the division. At this point we have made a specification as in figure 1 and table 1.

The last additional encoding restrictions did not affect the total schedule length, which is still 3961 cycles. Performing the changes of this section on the parts without the comparator (less area) is also possible (schedule length : 4921 cycles).

5 Conclusions and future work

We have shown how instruction selection for ASIPs can be done by generating a combined instruction set and data-path model from the instruction set and an abstract data-path and performing operation bundling on that model.

We also demonstrated a method to iteratively define a data-path and an instruction set for an ASIP. Each run of the analysis and bundling tools gives statistical information for that purpose.

First versions of the analysis, bundling, data routing and scheduling tools are currently available. Some future work will consist of further implementing these ideas and experimenting with some preprocessing of the model. This preprocessing would allow faster path searching in the model. The heuristics in the bundling algorithm also have to be refined and tested.

Acknowledgements

The authors gratefully acknowledge contributions from Koen Van Nieuwenhove (EDC-Mentor Graphics), who developed the first version of the analysis tool. The research was sponsored by the EU under the ESPRIT-2260 (SPRITE) and ESPRIT-9138 (CHIPS) projects.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers (Principles, Techniques, and Tools).* Addison-Wesley Publishing Company, 1988.

[2] A. Fauth, M. Freericks, and A. Knoll. Generation of hardware machine models from instruction set descriptions. In *VLSI Signal Processing VI*, Oct. 1993.

[3] A. Fauth, G. Hommel, C. Muller, and A. Knoll. Global code selection for directed acyclic graphs. *Proceedings Compiler Construction*, 1994.

[4] A. Fauth and A. Knoll. Automated generation of DSP program development tools using a machine description formalism. In *Proc. IEEE of ICASSP 93*, Minneapolis, 1993.

[5] C. W. Fraser and R. R. Henry. BURG — fast optimal instruction selection and tree parsing. *ACM SIGPLAN Notices*, 27(4):68–76, Apr. 1992.

[6] G. Goossens, F. Catthoor, D. Lanneer, and H. De Man. Integration of signal processing systems on heterogeneous IC architectures. *6th ACM/IEEE HLSW.*, 1992.

[7] D. Landskov, S. Davidson, B. Shriver, and P. Mallett. Local microcode compaction techniques. *ACM Computing Surveys*, 12(3):261–294, Sept. 1980.

[8] D. Lanneer. *Design Models and Data-Path Mapping for Signal Processing Architectures.* PhD thesis, K.U. Leuven-IMEC, 1993.

[9] D. Lanneer, F. Catthoor, G. Goossens, M. Pauwels, J. Van Meerbergen, and H. De Man. Open-ended system for high-level synthesis of flexible signal processors. In *EDAC-90*, 1990.

[10] D. Lanneer, M. Cornero, G. Goossens, and H. De Man. Data routing : A paradigm for efficient data-path synthesis and code generation. *7th ACM/IEEE HLSS*, 1994.

[11] C. Liem, T. May, and P. Paulin. Instruction-set matching and selection. *Proc. of the European Design and Test Conference (EDAC)*, 1994.

[12] L. Nowak. Graph based retargetable microcode compilation in the MIMOLA design systems. In *Proceedings of the 20th ACM workshop on MicroProgramming (MICRO-20)*, 1987.

[13] L. Nowak and P. Marwedel. Verification of hardware descriptions by retargetable code generation. In *26th ACM/IEEE Design Automation Conference*, 1989.

[14] R. Sluyter, H. Kotmans, and A. Leeuwaarden. A novel method for pitch extraction from speech and a hardware model applicable to vocoder systems. *Proc. IEEE ICASSP 80*, pages 45–48, Apr. 1980.