A Multiprogrammed Parallel Architecture for Digital Signal Processing

Tao Li, Brent Nelson
Kelly Flanagan, Christopher Read

Department of Electrical Engineering
Brigham Young University
Provo, Utah 84602

ABSTRACT

A parallel architecture for DSP algorithms is presented along with a corresponding computation method known as *address-directed* computing. This approach overcomes the large amount of addressing overhead in conventional DSP programs, especially those coded in high level languages. The approach used is to decompose a computation into address and data streams. The address stream is computed using a set of n concurrently operating address processors. An architecture to implement this method is presented as well as an example of programming it.

INTRODUCTION AND MOTIVATION

Parallel architectures for digital signal processing are receiving significant attention in the research community and many general and special purpose architectures have been proposed [1,2,3,4,5,6,7,8]. Of these, many are parallel in nature including data flow machines and systolic processors. Most of the proposed parallel approaches decompose an algorithm into subtasks according to its natural data structure (subcomputations).

However, our examination of a number of DSP programs [9] on conventional processor architectures reveals that less than 20% of total instructions in a program were for data processing purposes while more than 80% were used for address processing. The following contains the results for a collection of C and FORTRAN programs compiled on a VAX.

Program	Data Instr.
FFT	21%
LPC Analysis	19%
2D Convolution	11%
Matrix Multiply	10%
Dyna. Time Warp	17%
Gauss Eliminate	13%

This work, then, is based on the following observations:

- Most parallel approaches to DSP computing only consider the data concurrency (an FFT may be broken into a number of butterflies which can be computed independently). The success of these approaches is very dependent on the algorithm chosen for implementation: many algorithms may have little or no inherent data concurrency.
- Many programs (especially those using multi-dimension data arrays) spend the vast majority of their time computing array offsets when written in high level languages. To achieve efficiency, software developers are still writing much code in assembler or as microcode routines for special purpose machines.
- A number of recently announced DSP chips do have separate address and data computation units. These are not programmed separately however but are controlled by a wide instruction word. Thus, the computation is still data-directed.
- The possibilities of *address-directed* computation have not yet been addressed in the research community. Specifically, we believe that the address generation flow has more inherent concurrency available (due to a lack of dependencies) than does the data computation flow.

This paper proposes a new approach for exploiting the parallelism found in signal processing algorithms. In it we propose a multiprocessor architecture (named PSP for Parallel Signal Processor) and a corresponding technique for programming it (parallel *address-directed* programming).

THE PSP ARCHITECTURAL OVERVIEW

The key point of our approach is to decompose the computation into address and data computation flows. The computation is then programmed around the address flow rather than the conventional data processing flow, hence the name *address-directed* computing. The architecture shown in Fig 1 implements this using a set of concurrently operating address processors and one or more **data driven** data processors.

A typical data operation can be expressed as a tuple: *[opcode, src1, src2, destination]*. The operation of the PSP system can be described as:

- one or more address processors compute the operand addresses and place these values into the appropriate queues.
- along with operand addresses, the address processors generate the opcode stream which is placed into the OP queue like any other value.
- when a complete tuple is present at the head of the queues, the data processor section fires by retrieving the operands, performing the indicated computation, and storing out the result. The actual PSP implementation may overlap the memory accesses and data computation to increase throughput.

The address processors and queues are linked by the interconnection network. As a result, during any clock cycle, any address processor can place an address into any one of the queues. The OP queue is reserved for opcodes which are generated by the address processors as well. These processors are capable of fixed point arithmetic such as addition, subtraction, as well as bit operations (rotation, bit reversal, and masking). They share a common memory unit for global and temporary variables. Synchronization between the address processors is realized through these shared memory locations.

The address processor architecture is shown in Fig 2. It contains a set of general purpose registers, a permutator, an adder, and a multiplier. It also includes a program store. The address processor is a RISC machine which executes simple data movement instructions, integer arithmetic operations, test and branch instructions, and data permutations. The destination for any of these instructions can be either a register or a operand queue.

An important point is that since the data processor and address processors operate on separate data, the data processor chosen for a particular configuration can be either a fixed or floating point unit. Also, since no data-dependent branching is allowed in the data processor (it directly executes the tuple stream generated by the address processors), conditional opcodes are provided so that tuples can be either ignored or executed based on a status flags register.

PROGRAMMING THE PSP

The PSP is currently programmed by hand. The next step in our research will be the development of a compiler to take advantage of the architecture. To date our programming of the PSP has entailed first coding the algorithm for a version of the architecture with a single address processor and then parallelizing this program for more than one address processor.

As an example of a program fragment for the PSP consider Figure 3. In this example that addresses and opcodes are placed into the appropriate queues (0-3) by specifying them as destinations of MOV or arithmetic instructions. In Figure 3 all the sub-operations for a single data instruction have been listed on a line. For a single address processor system these would be executed consecutively to fill the queues with the appropriate addresses and opcodes. If a four address processors were desired, the code could be compiled so that each processor would be dedicated to a single column in the program. If this were the case, each of the four processors would be required to have their own copy of R0 and perform the branching. This is because each address processor indeed executes its own independent program.

The program example is the inner loop for a four-tap FIR filter. Register R0 is the loop counter and it is initialized to 3. The input data is stored in memory locations 0-3 while the filter taps are stored in locations 4-7. The program uses memory locations 777 and 999 as temporary storage to perform the sum-of-products computation. The exact computation performed is:

$$memory_{999} = \sum_{i=0}^{3} memory_i \times memory_{i+4}$$

but notice that it is computed in backwards order since R0 is decremented from 3 to 0. Although the example was written to facilitate the use of one or four address processors, any number could conceivably be used in a PSP implementation.

Most of the algorithms programmed to date have used between one and eight processors. These include: FIR filter, DFT, FFT, adaptive filter, image neighborhood transform (edge detection, 2d image filtering, etc), dynamic time warp, and matrix multiplication. Processor utilization of the single floating point data processor ranged from 89% to 65% using four address processors. If the address processors, due to their simplicity, executed faster than the data processor then it would not be unreasonable to expect 100% data processor utilization for many sections of code.

CODE OPTIMIZATION

A number of code optimization techniques have been used in our programming practice. These include:

1. Variable distribution: variables such as loop counters, loop limits, and program constants are distributed to each address processor. This makes it unnecessary for them to communicate and synchronize using the shared address memory.

2. Instruction relocation: instructions in the original code which are not data dependent may be reordered and distributed between the address processors.

3. Serial loop unwrapping: a loop body is replicated n times. Instances of the loop variable in the loop body are replaced by constants. Each processor then executes a portion of each instruction in the replicated loop.

4. Parallel loop unwrapping: a loop body is replicated n times but each address processor executes one complete copy of the loop body.

The loop unwrapping techniques range from partial to complete unwrapping. That is, a loop of length $L = 16$ can be unwrapped to give 16 serial copies, removing any need for loop variables and branch instructions. On the other hand, it could be unwrapped into 4 copies (executed 4 times each), increasing the throughput.

SUMMARY, STATUS, AND FUTURE RESEARCH

An architecture was presented for parallel digital signal processing. Our work on programming this architecture has shown that it gives much higher data processor utilization than conventional architectures. The parallelism which this architecture exploits is in the address computation stream rather than the data computation stream.

An assembler and RTL simulator for the PSP architecture are currently being used for research on *address-directed* programming methods. In addition, we have begun the chip-level design of the PSP system.

We are now also investigating algorithms and transformations for parallelizing and optimizing PSP code. In addition, we are looking at methods for determining the optimal number of address processors for a particular task. Our next step is to develop an optimizing compiler for PSP which will take advantage of the multiple address processors.

REFERENCES

1. L. Hartima, K. Kronlof, O. Simula, and J. Skytha, "DFSP: A Data Flow Signal Processor", IEEE Transactions on Computers, Jan. 1986.

2. K. Hwang and P.S. Tseng, "An Efficient VLSI Multiprocessor for Signal/Image Processing", Proceedings of IEEE ICCD '85, Oct. 1985, p 172.

3. F. J. van Wijk, et al, "On the IC Architecture and Design of 2um CMOS 8 MIPS Digital Signal Processor with Parallel Processing Capability: The PCB 5010/5011", ICASSP 86 Proceedings, Apr. 1986, p 385.

4. T. Nishitani, et al, "Advanced Single-Chip Signal Processor", ICASSP 86 Proceedings, Apr. 1986, p 409.

5. J. R. Boddie, W. P. Hays, and J. Tow, "The Architecture, Instruction Set and Development Support for the WE DSP32 DIgital Signal Processor", ICASSP 86 Proceedings, Apr. 1986, p 421.

6. H. T. Kung, "Systolic Algorithms for the CMU Warp Processor", Proceedings of the Seventh International Conference on Pattern Recognition, International Association for Pattern Recognition, 1984, pp. 570-577.

7. S. Y. Kung, H. J. Whitehouse, and T. Kailath (editors), VLSI and Modern Signal Processing, Prentice-Hall, 1985.

8. D. Mundie and D. Fisher, "Parallel Processing in Ada", Computer, Aug. 1986, p 20.

9. "Programs for Digital Signal Processing", edited by the Digital Signal Processing Committee, IEEE Acoustics, Speech, and Signal Processing Society, IEEE Press, 1979.

10. D. Kuck, *The Structure of Computers and Computation*, John Wiley & Sons, New York, 1979.

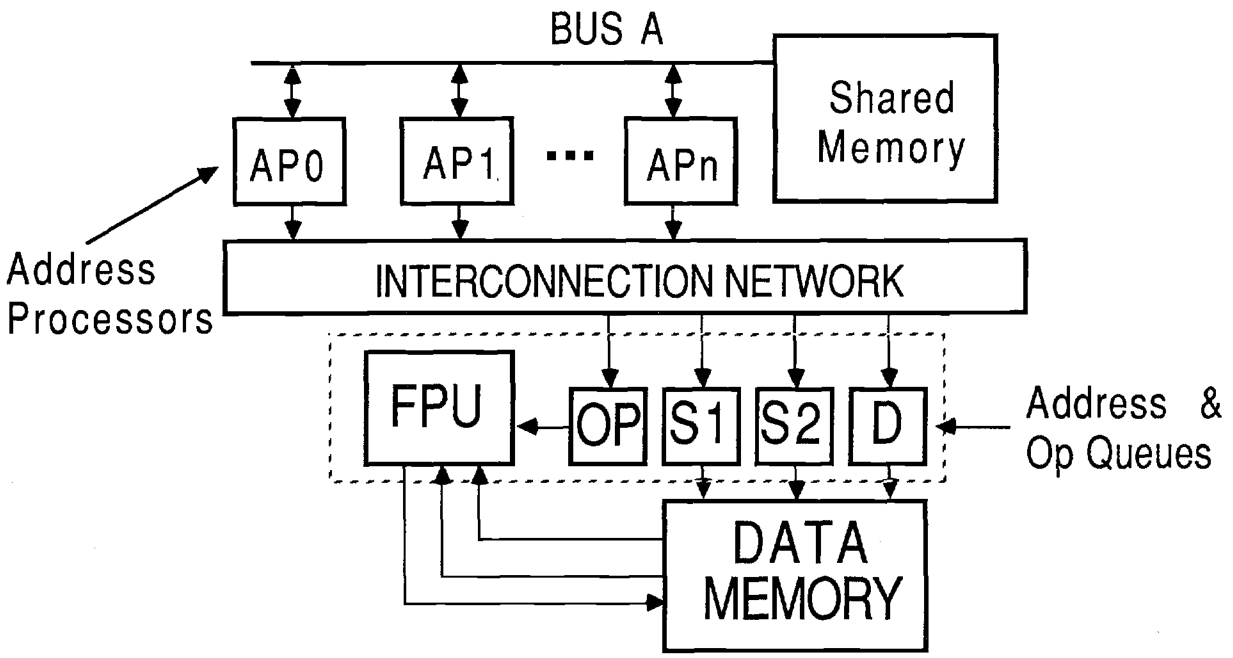

Figure 1: PSP Architecture

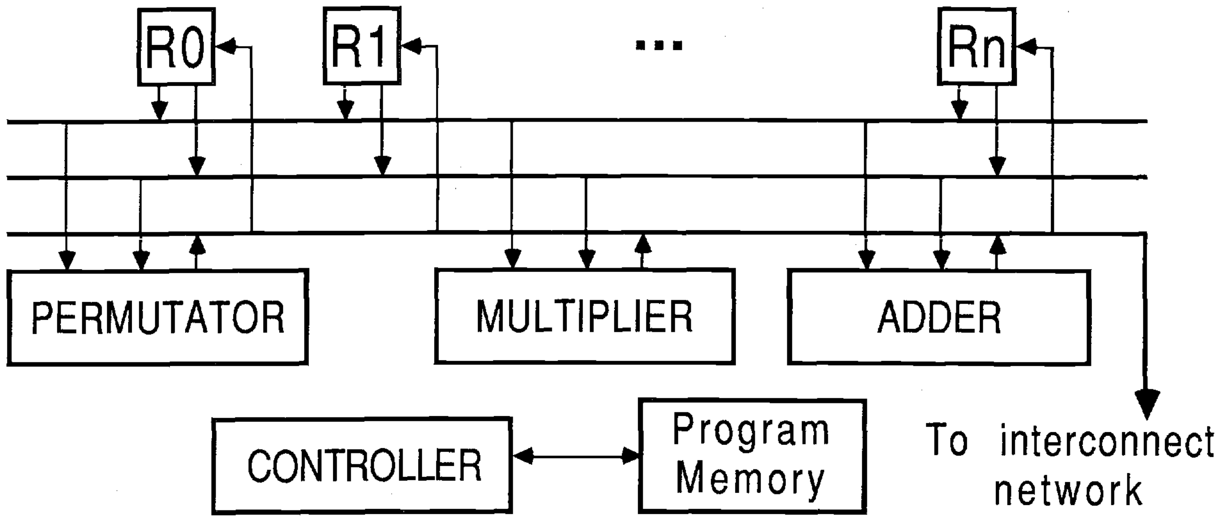

Figure 2: Address Processor Architecture

```
     MOV 3 R0        #Initialize loop counter
LABEL LOOP
     MOV FMUL Q0     MOV  R0 Q1     ADD   4 R0 Q2     MOV 777 Q3
     MOV FADD Q0     MOV 777 Q1     MOV 999    Q2     MOV 999 Q3

     DLN LOOP R0     #Decr R0, loop until negative
```

Figure 3: PSP Code Example