Mapping an FIR Filter to a 2-Dimensional Mesh of Processors

Howard CheHao Chang, Bevan M. Baas

Computer Engineering Research Laboratory
Department of Electrical and Computer Engineering
University of California, Davis

April 4, 2003

Abstract

This report presents and analyzes results of mapping an FIR filter algorithm onto a 2-Dimensional (2-D) Digital Signal Processing (DSP) processor array. The variation in throughput and hardware requirements over changing topologies is examined over a total of seven major topologies and eighty-five specific mappings. Performance results and mapping suggestions are also given.

1. Brief Introduction to the Programmable DSP Architecture

The proposed programmable DSP processor architecture [1] contains hundreds to thousands of simple but fast and energy efficient processor units per chip. Processors work asynchronously in different frequency domains and connect to neighboring processors by direct link through asynchronous First-In-First-Out (FIFO) buffers. In one approach, each processor unit has links to its four neighboring processors in the up, down, left and right directions. A processor unit has two input FIFOs and one output FIFO; therefore it can take inputs from two of the four directions and broadcast its results to neighbors along with a TAG. By this method, a processor can take inputs that are generated for itself only by comparing the TAG. Input and Output FIFOs have large enough buffers so that data can queue up without dropping samples. Each programmable DSP processor also has many features that exist in many modem DSP processors such as a built-in Multiply-Accumulate (MAC) unit, hardware memory address generators, and zero-overhead looping hardware support.

2. FIR Algorithm

The Finite Impulse Response (FIR) filter is a major function of digital signal processing [2]. An FIR filter is typically described by equation (3.1)

$$y(k) = \sum_{k=0}^{N-1} h(n)x(k-n) \tag{3.1}$$

where $y(k)$ is the FIR filter output, N is the number of taps or coefficients of the filter, $x(k\text{-}n)$ are the input samples, and $h(n)$ are the coefficients of the filter [3]. There are two kinds of classic FIR filter structures: Direct Form and Transpose Direct Form as shown in Figs. 3.1 and Fig. 3.2 respectively.

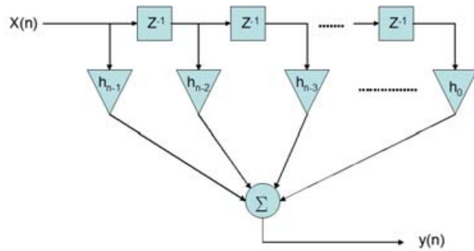


Fig. 3.1 Direct Form FIR filter

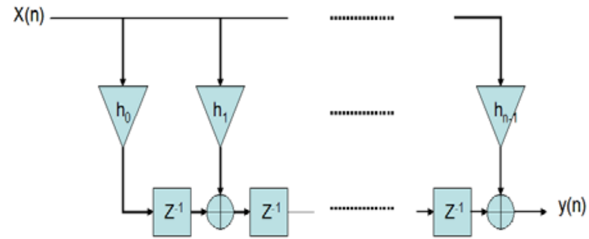


Fig. 3.2 Transpose Direct Form FIR filter

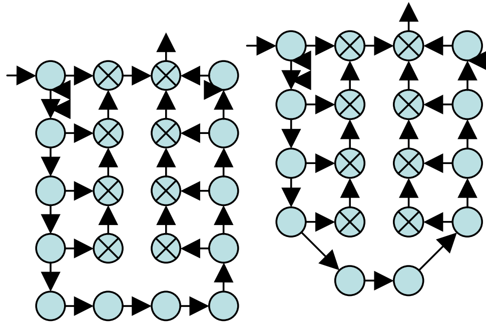
In the Direct Form FIR filter structure, a tapped delay line is located at the input and coefficient products are summed up by adders and then sent to the output. On the other hand, the Transpose Direct Form FIR structure broadcasts input data samples to all coefficient multipliers. Products from multipliers are sent to two-input adders and intermediate sums are delayed by delay registers between adders.

3. Mapping Methods and Results

The Transpose Direct Form FIR filter is commonly used to achieve high speed in ASIC designs because of its implicit pipeline feature. However, it is not cost efficient if we adapt this architecture in our DSP processor array because we don't have a common data bus in our design to perform the broadcasting of input samples to multipliers. To achieve this broadcasting, we need to allocate processor units to perform distribution. The overhead becomes more and more significant as the number of filter taps increases. For this reason, we focus on the Direct Form FIR filter structure in this report.

We examine seven different FIR topologies; three we call *U-type,* three *L-type* and one we call *I-type.* In both U-Type and L-Type topologies, we consider three different function assignment methods as shown in Figs. 4.1, 4.2, and 4.3. In method one, as shown in Fig. 4.1, multipliers and adders are combined. In method two, as shown in Fig. 4.2, multiplication units and input sample forwarding units are combined with each other. In method three, as shown in Fig. 4.3, the input sample forwarding units, multiplication units, and adder units are separated. If we use a single processor to perform input sample distribution, multiplication, and addition, the topology has an I-Type structure. Pseudo assembly code for 16-tap FIR filters was written to simulate and calculate the performance of each topologies and methods and is given in the Appendix of this report. In addition, the possible performance changes caused by changing processor interconnection network configurations from four neighboring connections to eight neighboring connections was investigated.

Figures 4.1, 4.2, 4.3, and 4.4 show that increasing the number of output ports from four to eight has only minor effects on performance and cost. The change in performance and cost is due to reducing the latency and hardware requirements by shortening the data-distribution delay line. Since the variation is not significant, this report focuses on the four output-port structure.



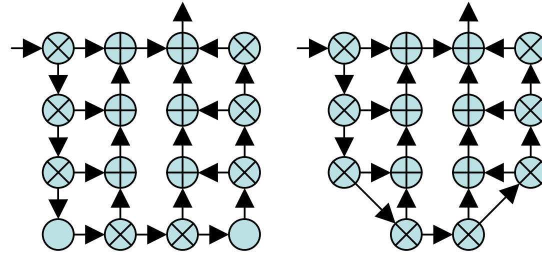
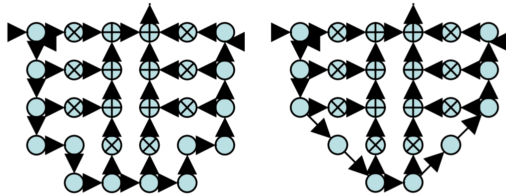


Fig 4.1 U-Type Method 1
4 output ports vs. 8 output ports

Fig. 4.2 U-Type Method 2
4 output ports vs. 8 output ports



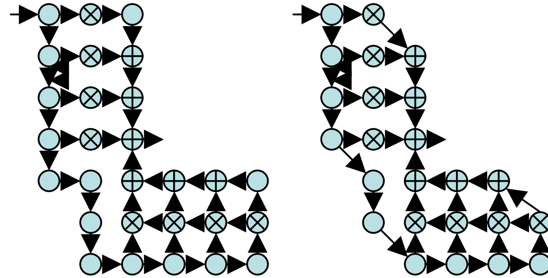


Fig. 4.3 U-Type Method 3
4 output ports vs. 8 output ports

Fig. 4.4 L-Type
4 output ports vs. 8 output ports

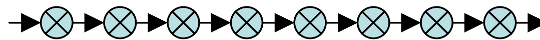


Fig 4.5 I-Type topology

Results of the mappings are summarized in Figs. 4.6 and Fig. 4.7. For the seven major topology categories, eighty-five specific mappings of the 16-tap FIR filter were designed and assembly code was written for each. In general, each mapping provides a different throughput and requires a different number of processors. The overall filter throughput is normalized to samples per clock cycle to make the analysis applicable for systems of any clock cycle time. For example, if the required throughput of an FIR filter is 150 Msamples/sec and the maximum clock rate of a given processing system is 1 GHz, the necessary throughput is then 0.15 FIR samples/CLK period. The goal is to develop a library of possible mappings which show the optimum mapping given a particular throughput requirement.

Figure 4.6 shows results of *throughput/number_of_processors* for the eighty-five mappings. As expected, mappings that utilize a greater number of processors generally provide a higher throughput, but with diminishing returns due to the non-linear speedup associated with adding additional processors. To achieve maximum throughput, which is 1 FIR sample per CLK period, the only two topologies available are U-Type method three and L-Type method three. Comparing these two topologies, we can see clearly that L-Type method three is a better placement method compared to U-Type method three because it requires fewer processor units to achieve the same throughput (54 processing elements versus 58). This holds true even when

the throughput is low. Data from Fig. 4.6 show that L type is the best choice when the desired throughput is between 0.25 samples/clk to 0.5 samples/clk. When the desired throughput is less than 0.25 samples/clk, I-type can always achieve the same throughput with the fewest processor units.

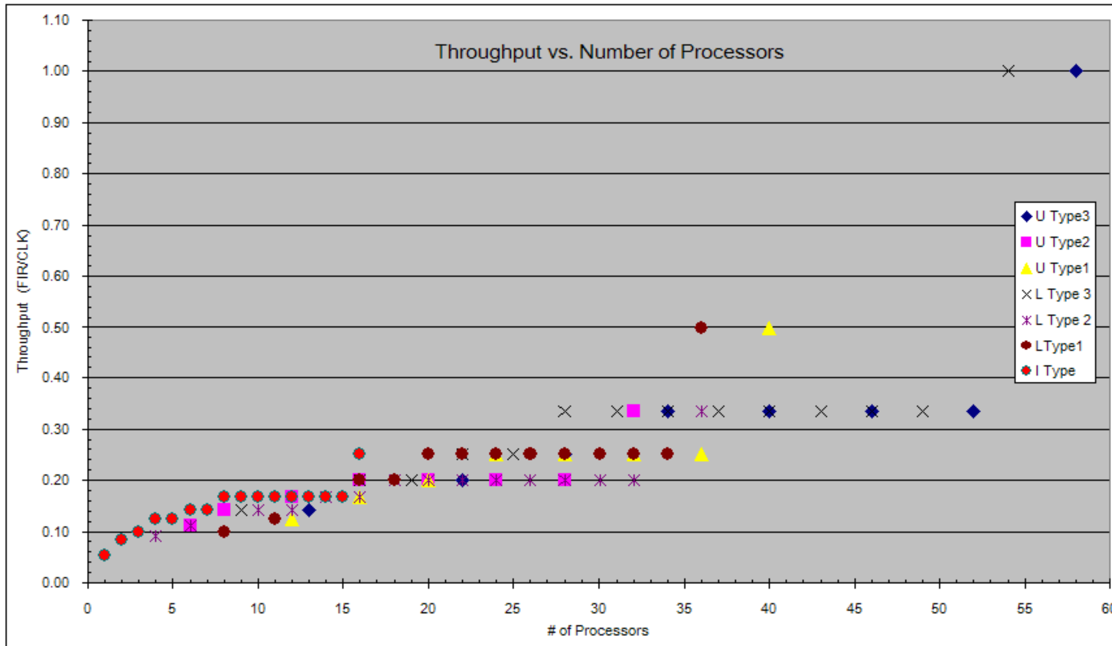


Fig. 4.6 Throughput vs. # of processors

Figure 4.7 shows the same data as Fig. 4.6 but normalizes the overall throughput by the number of processors required to perform the computation. This shows how efficiently processors are utilized by the particular mappings. I-type topologies are seen to have an outstanding *throughput/number_of_processors* ratio from Fig. 4.7. This outcome is very reasonable because these mappings execute add operations without an explicit ADD instruction by using the Multiply and Accumulate (MAC) instruction. Even though the slowest mapping takes 19 clock cycles per sample, which is 19 times slower than the fastest one by L-type type three, its hardware requirement is 54 times less than the fastest one. This result is an example of the rule of thumb that when a high throughput rate is not required, the I-Type structure not only consumes the least amount of processor units but also has the best *throughput/number_of_processors* ratio.

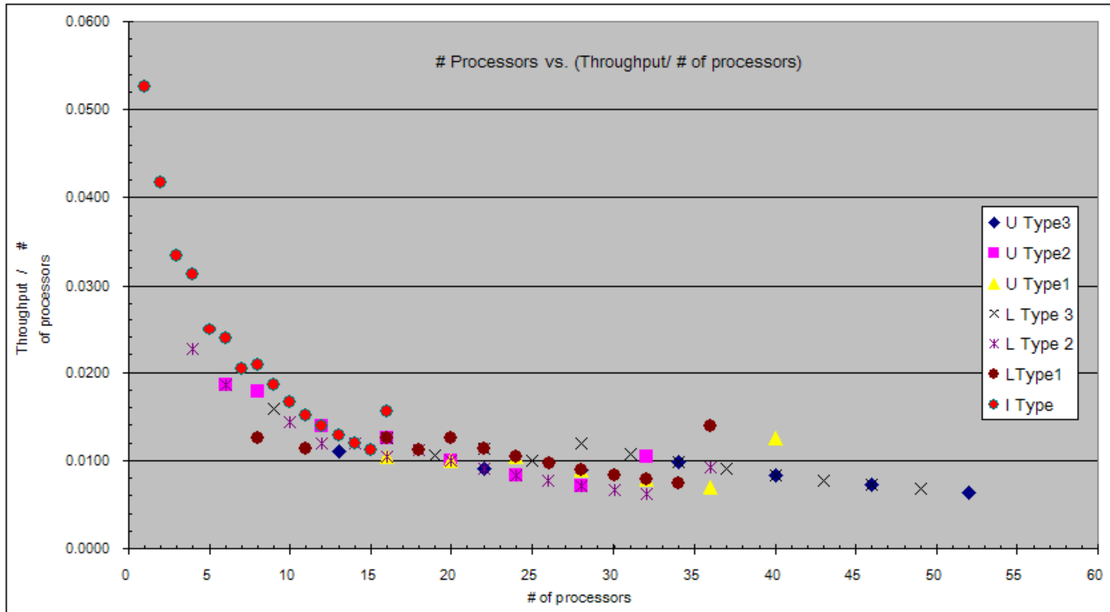


Fig 4.7 Number of processors vs. (Throughput/# of processors)

4. Conclusion and Future Work

We can achieve very high speed and power efficiency by having a processor array and keeping the processing units simple. From the results shown in Section 3, we can have plenty of flexibility by this direct mapping method. Among U-Type, L-Type and I-Type topologies, the latter two are very promising and have their own roles in different throughput ranges. We should further investigate other mapping methods because there is a significant gap between throughputs of 1 FIR sample/CLK period and 0.5 FIR samples/CLK period. The currently-proposed mappings require at least 54 processor units to achieve a throughput higher than 0.5 FIR samples/CLK period.

References

[1] Bevan M. Baas, "The Baseband Processor for an 802.11a Wireless LAN (54 Mbps, 5 GHz) and a Proposed High-Performance Programmable DSP Architecture." EEC290 Seminar, Roessler Hall, UC Davis, December 6, 2002.

[2] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[3] Chiung-Lung Chen, *FIR Architecture Synthesizer Based on CSD Code.* Master Thesis, Department of Electrical Engineering, National Central University. Taiwan, R.O.C.; June 1998.

Appendix

Assembly codes used to evaluate the performance and hardware requirements of each topology.

A. For I Type

```
# 16 Processors
========================================
L1:  MOVE WMem(0), IBuf0;
     MULT WMem(), WMem(0), SMem(0);
     MOVE OBuf, WMem(0);
     ADD OBuf, WMem(), IBuf0;
     BR L1;

# 2 ~ 15 Processor
========================================
L1:  MOVE WMem(0), IBuf0;
     MOVE OBuf, WMem(0);
     CLACC;
L2:  MAC ACC, WMem(), SMem();
     CBR L2;
     MAC WMem(), WMem(0), SMem(0);
     ADD OBuf, WMem(), IBuf0;
     BR L1;

# 1 Processor
========================================
L1:  MOVE WMem(0), IBuf0;
     CLACC;
L2:  MAC ACC, WMem(), SMem();
     CBR L2;
     MAC OBuf, WMem(0), SMem(0);
     BR L1;
```

B. For L Type1

```
# 16 Processors
========================================
# Distribute
L1:  MOVE OBuf, IBuf;
     BR L1;

# Multiply and ADD
L1:  MULT WMem(8), IBuf0, SMem(0);
     ADD  OBuf, WMem(0), IBuf1;
     BR L1:

# 2~15 Processors
========================================
# Distribute
L1:  MOVE WMem, IBuf1;
L2:  MOVE OBuf, WMem;
     CBR L2;
     BR L1;

# Multiply and ADD
L1:  CLACC;
L2:  MAC ACC, IBuf0, SMem();
     CBR L2;
     MAC WMem(), IBuf0, SMem(1);
     ADD OBuf, WMem(), IBuf1;
     BR L1;
```

C. For L Type2

```
# 16 Processors
========================================
# Distribute and Multiply
L1:  MOVE WMem(0), Ibuf0;
     MOVE Obuf, WMem(0);
     MULT OBuf, Wmem(0), Smem(0);
     BR L1;

# 4~15 processors
========================================
# Distribute and Multiply
L1:  MOVE WMem(0), Ibuf0;
     CLACC;
L2:  MAC ACC, WMem, SMem;
     CBR L2:
     MAC OBuf, Wmem, SMem;
     MOVE Obuf, WMem(0);
     BR L1;

# ADD(OUTPUT PU)
L1:  CLACC;
```

```
     ADD ACC, IBuf0, IBuf1;
     ADD OBuf, ACC, IBuf1;
     BR L1;
```

 # ADD

```
L1:  ADD Obuf, Ibuf0, Ibuf1;
     BR L1;
```

 # 3 processors

==

 # Distribute + Multiply

```
L1:  MOVE WMem, Ibuf1;
     CLACC;
L2:  MAC ACC, WMem, SMem;
     CBR L2;
     MAC Obuf, WMem, SMem;
     MOVE OBuf, WMem;
     BR L1;
```

 # ADD

```
L1:  CLACC;
     ADD ACC, Ibuf0, Ibuf1;
     ADD Obuf, Ibuf0, ACC;
     BR L1;
```

 # 2 processors

==

 # Distribute + Multiply

```
L1:  MOVE WMem, Ibuf1;
     CLACC;
L2:  MAC ACC, WMem, SMem;
     CBR L2;
     MAC Obuf, WMem, SMem;
     MOVE OBuf, WMem;
     BR L1;
```

 # ADD

```
L1:  ADD OBuf, Ibuf0, Ibuf1;
     BR L1;
```

D. For L Type3

16 Processors

==

 # Distribute

```
L1:  MOVE OBuf, Ibuf1;
     BR   L1;
```

 # Multiply

```
L1   MULT OBuf, Ibuf0, SMem;
     BR   L1;
```

 # Add

```
L1   ADD  OBuf, IBuf0, IBuf1;
     BR   L1;
```

 # 4 ~ 15 Processors (Worst case only)

==

 # Distribute

```
L1:  MOVE WMem, Ibuf1;
L2:  MOVE Obuf, WMem;
     CBR L2;
     BR   L1;
```

 # Multiply

```
L1:  CLACC;
L2:  MAC ACC, Ibuf0, SMem;
     CBR  L2;
     MAC OBuf, Ibuf0, SMem;
     BR   L1;
```

 # Add

```
L1:  ADD  OBuf, IBuf0, IBuf1;
     BR   L1;
```

3 Processors

==

 # Distribute

```
L1:  MOVE WMem, Ibuf1;
L2:  MOVE Obuf, WMem;
     CBR L2;
     BR   L1;
```

 # Multiply

```
L1:  CLACC;
L2:  MAC ACC, IBuf0, SMem;
     CBR L2;
     MAC OBuf, IBuf0, SMem;
     BR L1;
```

 # ADD (Suppose the coefficient is 5/5/6 distribution)

```
L1: CLACC;
    ADD ACC, IBuf0, IBuf1;
    ADD Obuf, IBuf1, ACC;
    BR L1;
```

E. For U Type1

```
# 16 Processors
========================================
# Distribution PUs
L1: MOVE OBuf, IBuf;
    BR L1;

# Multiply and ADD
L1: MULT WMem(8), IBuf0, SMem(0);
    ADD  OBuf, WMem(0), IBuf1;
    BR L1;

# 4~15 Processors
========================================
# Distribute
L1: MOVE WMem, IBuf1;
L2: MOVE OBuf, WMem;
    CBR L2;
    BR L1;

# Multiply and ADD
L1: CLACC;
L2: MAC ACC, IBuf0, SMem();
    CBR L2;
    MAC WMem(), IBuf0, SMem();
    ADD OBuf, WMem(), IBuf1;
    BR L1;
```

F. For U Type2

```
# 16 Processors
========================================
# Distribute and Multiply
L1: MOVE WMem(0), Ibuf0;
    MULT OBuf, Wmem(0), Smem(0);
    MOVE Obuf, WMem(0);
    BR L1;

# ADD(Normal PUs)
L1: ADD Obuf, Ibuf0, Ibuf1;
    BR L1;

# ADD(OUTPUT PU)
L1: CLACC;
    ADD ACC, IBuf0, IBuf1;
    ADD OBuf, IBuf0, ACC;
    BR L1;

# 4~15 processors
========================================
# Distribute and Multiply
L1: MOVE WMem(0), Ibuf0;
    CLACC;
    BR L1:
L2: MAC ACC, WMem, SMem;
    CBR L2:
    MAC OBuf, Wmem, SMem;
    MOVE Obuf, WMem(0);
    BR L1;

# ADD(Normal PUs)
L1: ADD Obuf, Ibuf0, Ibuf1;
    BR L1;

# ADD(OUTPUT PU)
L1: CLACC;
    ADD ACC, IBuf0, IBuf1;
    ADD OBuf, IBuf0, ACC;
    BR L1;

# 3 processors
========================================
# Distribute + Multiply
L1: MOVE WMem, Ibuf1;
    CLACC;
L2: MAC ACC, WMem, SMem;
    CBR L2;
    MAC Obuf, WMem, SMem;
    MOVE OBuf, WMem;
    BR L1;

# ADD
L1: CLACC;
    ADD ACC, Ibuf0, Ibuf1;
    ADD Obuf, Ibuf0, ACC;
    BR L1;
```

G. For U Type3

```
# 16 Processors
========================================
# Distribute
L1: MOVE OBuf, Ibuf1;
    BR   L1;

# Multiply
L1  MULT OBuf, Ibuf0, SMem;
    BR   L1;

# Add
L1  ADD  OBuf, IBuf0, IBuf1;
    BR   L1;

# 4 ~ 15 Processors (Worst case only)
========================================
# Distribute
L1: MOVE WMem, Ibuf1;
L2: MOVE Obuf, WMem;
    CBR L2;
    BR   L1;

# Multiply
L1: CLACC;
L2: MAC ACC, Ibuf0, SMem;
    CBR  L2;
    MAC OBuf, Ibuf0, SMem;
    BR   L1;

# Add
L1  ADD  OBuf, IBuf0, IBuf1;
    BR   L1;

3 Processors
========================================
# Distribute
L1: MOVE WMem, Ibuf1;
L2: MOVE Obuf, WMem;
    CBR L2;
    BR   L1;

# Multiply
L1: CLACC;
L2: MAC ACC, IBuf0, SMem;
    CBR L2;
    MAC OBuf, IBuf0, SMem;
    BR L1;

# ADD (Suppose the coefficient is 5/5/6
distribution)
L1: CLACC;
    ADD ACC, IBuf0, IBuf1;
    ADD Obuf, IBuf0, ACC;
    BR L1;
```

5. Update History

2003/08/04 Grammar and style changes.

2003/05/19 Example assembly code programs added.

2003/04/04 Written.