Improving DSP Performance with a Small Amount of Field Programmable Logic

John Oliver and Venkatesh Akella

Department of Electrical & Computer Engineering
University of California, Davis
{jyoliver, akella} @ece.ucdavis.edu

Abstract. We show a systematic methodology to create DSP + field-programmable logic hybrid architectures by viewing it as a hardware/software codesign problem. This enables an embedded processor architect to evaluate the trade-offs in the increase in die area due to the field programmable logic and the resultant improvement in performance or code size. We demonstrate our methodology with the implementation of a Viterbi decoder. A key result of the paper is that the addition of a field-programmable data alignment unit (FPDAU) between the register-file and the computational blocks provides 15%-22% improvement in the performance of a Viterbi decoder on the state-of-the-art TigerSHARC DSP. The area overhead of the FPDAU is small relative to the DSP die size and does not require any changes to the programming model or the instruction set architecture.

1 Introduction

Can we improve the performance and power or memory requirements of a state-of-the-art DSP with programmable logic? Many researchers have addressed this question in the past and many solutions have been proposed including customized instructions, loops [1], reconfigurable functional units, [2] and co-processor [3,4,5,6,7,8]. However most of the existing approaches do not factor the cost of the programmable logic in their evaluation - they tacitly assume that the die size penalty of adding programmable logic is not important. However, in embedded applications where DSPs are used, cost is a very critical factor. So, we would like to find a sweet spot for the programmable logic where a small addition to the die size in the form of programmable logic realizes maximum return in terms of improvements to performance (throughput), power, or memory requirements. For this we believe that the integration of programmable logic with a DSP should be viewed as a hardware/software co-design problem.

We illustrate our proposal using a Viterbi decoder as an example. First we analyze the optimized assembly code for Viterbi decoding on state-of-the-art DSPs and show that it is not the functional units that are the problem but the restrictions on the connection between the register file and the computational units that are the bottleneck which can be elegantly overcome by using a flexible interconnect network that can be realized using field-programmable logic. We call this new hardware block - FPDAU (Field Programmable Data Alignment Unit). This is situated between the register file of a processor and the computational units. This block dynamically re-configures the dataflow between the register file and the functional unit and hence eliminates a significant fraction of the instructions in the kernels of many important signal-processing algorithms. In order to determine the configuration of the FPDAU the implementation has to be approached as a hardware/software co-design problem. We will show the details of our implementation again using the Viterbi decoder on a TigerSHARC DSP as an example.

The techniques presented are general enough to be used with any other DSP that supports SIMD style processing such as the AltiVec and TI's TMS320c62xx. Also, we show that this approach is not just meant for Viterbi decoding, it can be used with other algorithms as well. In fact, a variety of DSP oriented computations like vector and matrix operations like transposing a matrix, finding the determinant of a matrix can benefit with the proposed architecture.

1.1 Organization of This Paper

First we will introduce Viterbi decoding and how it is efficiently implemented in assembly language on the TigerSharc that already has support for ACS computation. Then we will show what the bottleneck of the implementation is and its impact on the execution time and memory for K=5, 7, and 9 Viterbi decoding. We then propose a simple scheduler and programmable interconnect to rectify the problem. The design of the scheduler is described and its cost in terms of equivalent look-up-tables and die size is estimated. We show how the field programmable interconnect is used by the DSP programmer. The improvements in performance are then presented. We then describe other algorithms that can benefit by the proposed solution to demonstrate that this is not just for Viterbi Decoding. Finally, we compare our approach to related solutions in the area of DSP+PL hybrids and show why our approach is more promising.

2 Overview of Viterbi Decoding and Its DSP Implementation

Viterbi decoding is a critical application in embedded communication systems like 802.11-based wireless LAN; CDMA based cellular technologies and host of other applications that require data communication over noisy channels. It is part of the EEMBC benchmark suite. In spite of special support to execute Viterbi algorithm efficiently modern DSP are unable to meet the high data rate Viterbi decoding requirements imposed by standards such as the 3G and 802.11(a). So, we use Viterbi algorithm as an example in this paper to illustrate our technique.

First, to understand the computational requirements of a Viterbi decoder, it is useful to start with a convolutional encoder. Fig. 1 shows a ½ rate convolutional encoder for constraint length K=3. In this encoder, for every input bit, two output bits are transmitted. Each input is convolved through XOR operations with the previous two bits. The circuit in Fig. 1 can also be represented as a state-machine shown in Figure 2.

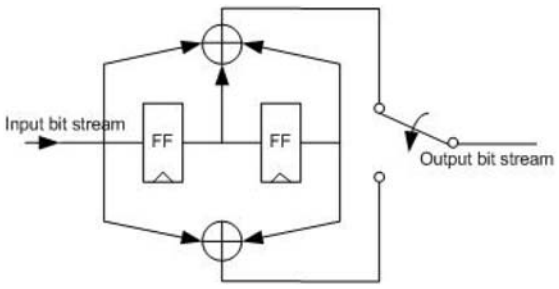


Fig. 1. K=3 Convolutional Encoder

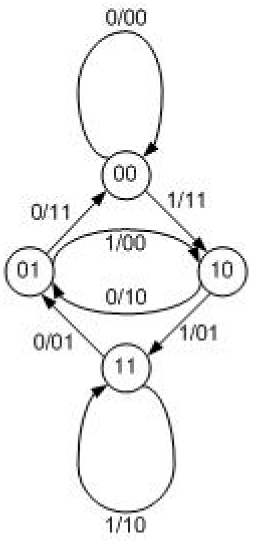


Fig. 2. State Diagram of K=3 Convolutional Encoder

The goal of a Viterbi decoder is to determine what the *most likely* inputs were, given an output data stream corrupted by a noisy transmission channel. A trellis is a map of all of the states from the encoder, drawn out to show each step in time. For the K=3 encoder shown in Figure 2, there would be four states in each time instance of the trellis. Fig. 3 shows a trellis used for Viterbi decoding for the K=3 convolutional encoder. Viterbi decoding consists of two tasks - the population of the trellis and the trace back through the trellis to find the path that yields the most likely sequence of states. Population of the trellis works as the follows. For each pair of input bits, the distance between the input bits and the expected output for each transition between states is calculated for each of the possible state transitions. In the

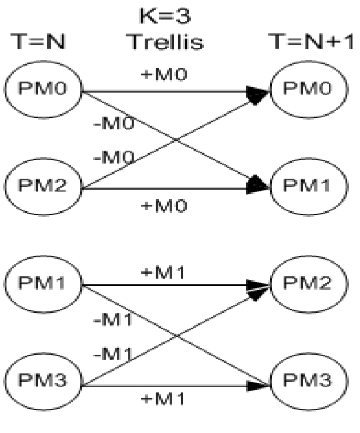


Fig. 3. K=3 Trellis Diagram

K=3 state machine shown in Figure 2, there are a total of 8 possible transitions, two to each state. This distance is represented by the +/- M0 and M1 in the trellis diagram in Fig. 3. The smallest distance to each state is chosen and saved for each state. For the next time instance, the same procedure is used except the chosen smallest distance to each state is added to the previous metric saved for that state. These accumulated distances are referred to as *path metrics*. This process of adding the input bits against the local path value, comparing the two local path values to find the smallest distance, and selection of the smallest path distance is often referred to as an Add-Compare-Select, or ACS. Many DSPs have custom ACS instructions to accelerate this process.

The traceback of a Viterbi decoder is simply the selection of the smallest accumulated state metric for each of states of the trellis. This computation is mostly serial, and relatively inexpensive in terms of instructions for Viterbi decoders of constraint lengths of 7 or more as a fraction of the total time.

2.1 AltiVec Implementation

The Altivec DSP co-processor[9] is a vector processor that operates on 128-bit vectors in 8, 16, or 32 bit SIMD mode. Assuming that path metrics are 32-bit values, we could store the four path-metrics PM0 to PM3 for the the ACS kernel for K=3 trellis (shown in figure 3) in one 128bit vector. Figure 4 shows the pseudo-assembly code for the implementation of the ACS kernel for K=3 where V0, V1, V2, V3, V4 and V5 are 128 bit vector registers. PM(x) denotes a 32-bit value that holds the accumulated path metric of state x. M0 and M1 represent the magnitude of the two different possible distances that may be generated for any input pair of bits. Figure 5 illustrates the flow of data between the registers and the result of the computation. For example, it shows that the least significant 32 bits of register V0 are obtained by adding PM(3) and M1 and so on. Now, in order to compute the new value PM(0) we need to find the minimum of PM(0)+M0 and PM(2)-M0 (please refer to Figure 3), but the vector-min instruction expects the two operands to be in adjacent locations in the vector register. This is an alignment restriction in SIMD processing and is results in simplification of the hardware.

So, the data needs to re-ordered so that the pairs of candidate path metrics are in adjacent sub-word locations in a vector register. This necessitates the need for the two *vec_merge* instructions shown in the pseudo code in Fig. 4.

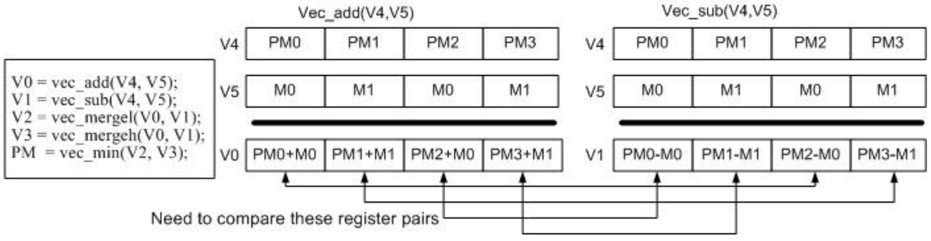

```
V0 = vec_add(V4, V5);
V1 = vec_sub(V4, V5);
V2 = vec_mergel(V0, V1);
V3 = vec_mergeh(V0, V1);
PM  = vec_min(V2, V3);
```

Fig. 4. Altivec Register Mapping of ACS and Pseudo Code

2.2 TigerSHARC Implementation

Is this restriction just a limitation of the AltiVec processor or is it more general? To investigate this we looked at other DSP architectures (with a completely different architecture style), namely, the TigerSHARC [**10**] from Analog Devices, which is a statically scheduled superscalar with various SIMD modes of computation and two independent functional units, that operate on 64-bit data. A block diagram of the TigerSHARC computational block is shown in Fig. 7. Each computational block is fed by a 32 entry, 32-bit register file with 4 read ports and 4 write ports. Within each computational block, there are 3 different SIMD modes, allowing for sub-word computations on 32-bit, 16-bit or 8-bit boundaries similar to the Altivec. There are restrictions on which registers may be used in a SIMD instruction. 32-bit SIMD calculations may be completed only on adjacent 32-bit registers. Similar restrictions are placed on 16-bit and 8-bit SIMD computations.

The K=3 trellis (presented in Fig. 3) can be mapped to one of the TigerSHARC computational blocks. Again the pseudo assembly code is shown in Figure 7 and the register dataflow is shown in Figure 8. PM(x) denotes a 32-bit value that holds the accumulated path metric of state x. M0 and M1 represent the magnitude of the two different possible distances that may be generated for any input pair of bits. PM3and PM2 are stored in register pair R5:4, PM1 and PM0 are stored in register pair R3:2 and M0 and M1 are assumed stored in register pair R1:0. This grouping of registers allows the TigherSHARC to use its 32-bit SIMD mode and represents an efficient implementation of Viterbi on TigerSHARC.

```
R15:12 = Add/Subtract(R5:4, R1:0);
R11:8  = Add/Subtract(R3:2, R1:0);
R15:12 = Merge(R15:14, R11:10);
R11:8  = Merge(R13:12, R9:8);
R15:12 = VectorMin(R15:14, R11:10);
R11:8  = VectorMin(R13:12, R9:8);
```

Fig. 5. TigerSHARC Viterbi ACS Pseudo-Code

On the right half of Fig. 6, both M0 and M1 as well as PM0 and PM1 can be fetched from the register file in a given cycle. Next, using a special instruction that allows addition and subtraction to operate on a pair of registers, the TigerSHARC can then produce half of all of the possible transitions for this stage of the trellis. The result of this computation is shown in the 128-bit result register R11:8. Likewise, the left half of Fig. 6 shows a similar computation for the other four possible transitions for the same stage of the trellis. Next we need to find the path with the minimum metric for each of the pairs of transitions to each state in the trellis, which can be done by the special *vector_min* instruction which also supports SIMD mode However, to utilize the SIMD mode, the vector minimum instruction expects the data to be compared in the same bit locations in both operands. The overlapping arrows in the

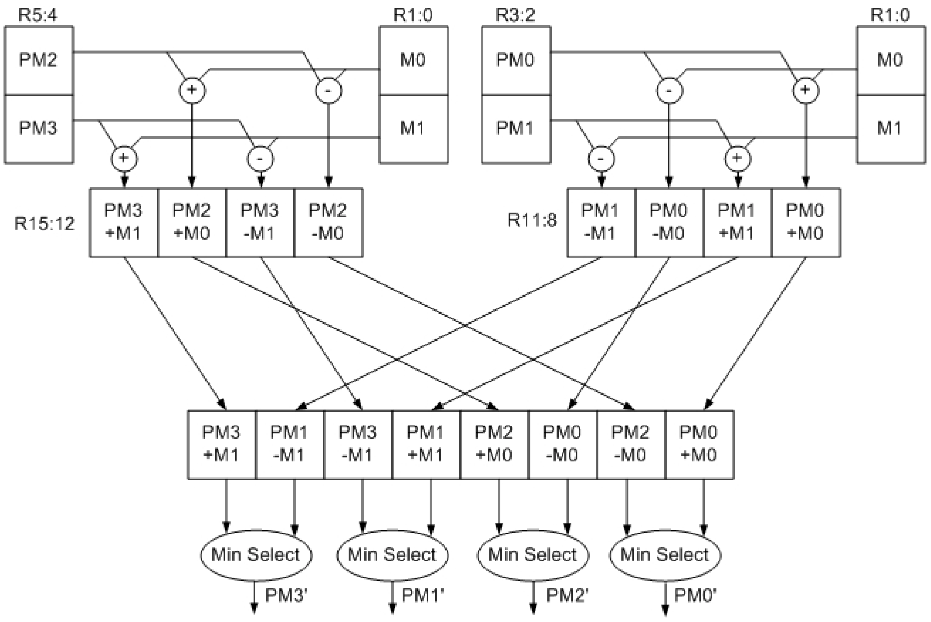


Fig. 6. Mapping of Viterbi ACS Dataflow to the TigerSHARC DSP

middle of Fig. 6 indicate the required data movement in order to utilize the SIMD vector minimum instruction. The overlapping arrows are realized by the *permutation* instructions that are similar in spirit to the *vec*_merge instructions in Altivec, i.e., they rearrange data in the register file. Finally, we analyzed the Texas Instrument'sC62xx DSP and found that a similar permute instructions are needed to overcome the SIMD restriction [11]. Table 1 shows the performance of the TigerSHARC on various different Viterbi decoders. The %ACS row indicates the fraction of the total cycles the TigerSHARC DSP spends on the trellis population and the %Permutes row indicates the fraction of the total cycles spent on permutations. These cycles are for the entire implementation of the GSM decoder. The fraction of the total cycles spent on ACS and permutations is similar for TI C6x DSP [12]and the AltiVec vector processor. From here on out, we will focus on the TigerSHARC architecture as we had access to the simulation tools for this platform.

Is there a more efficient way to address this problem? To investigate this we decided to profile the TigerSHARC implementation of a Viterbi decoder developed for the GSM wireless handset standard, which requires K=5, 16-bit data and 189 bit data frame.

Table 1. TigerSHARC Viterbi ACS Performance

For ½ Rate Viterbi Decoder, L=190 Bits	**K=5**	**K=7**	**K=9**
ACS Cycles	1960	4191	8459
Traceback Cycles	960	1245	1625
% Execution Cycles in ACS	67.1%	77.1%	83.9%
% of ACS Instructions which are Permutes	23.3%	25.0%	26.9%

3 HW/SW Co-design and the FPDAU

The data in Table 1 shows that a significant fraction of the computation cycles in the Viterbi decoder are spent in permute instruction, which are actually not doing anything useful in terms of the Viterbi algorithm. They are merely there to overcome the data flow restrictions to the function units in a typical DSP. So, the problem is not that the DSP do not have the appropriate instructions or the memory bandwidth (as shown in the previous section, most DSP do have special instructions to support Viterbi), but it is the data alignment restriction.

We propose a Field Programmable Data Alignment Unit (FPDAU) to circumvent the need for these permutation instructions. So, the data rearrangement will be done in hardware instead of software as it is being done now. This gives us two key benefits. It eliminates the instructions from the critical kernel of the computation and thereby provides improvements in performance and memory requirements and possibly reduces power and instruction cache pollution. It also gives us additional flexibility, because with a field-programmable hardware unit we can customize the dataflow to the specific algorithm being implemented.

Next we describe the details of the FPDAU and its integration with the DSP architecture and its programming model. We will illustrate this with the TigerSHARC DSP because we have access to their simulation tools. As noted before, a similar structure would work with other DSP as well; the programming model and the interface will differ.

The FP-DAU consists of two parts - a flexible interconnect that connects the register file to the ALU, Shifter and MAC units and a dynamically programmable state-machine to control the configuration of the flexible interconnect. The controller has configuration register that is mapped into the TigerSHARC's memory space. The placement of the FP-DAU is shown in Fig. 8.

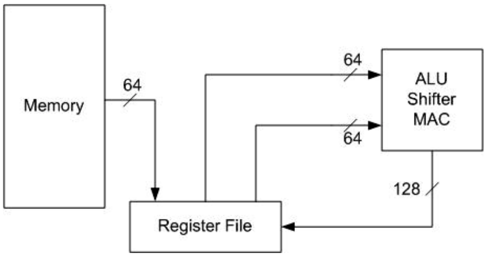


Fig. 7. Block Diagram of a TigerSHARC Computational Block

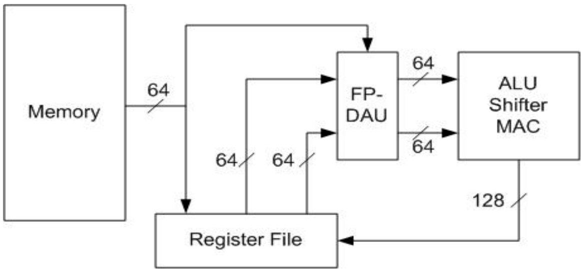


Fig. 8. Block Diagram of a TigerSHARC Computational Block with FPDAU

The detailed block diagram of the FPDAU is shown in Fig. 9. To support the data alignment required for Viterbi (the overlapping arrows in the middle of Fig. 6), we need an interconnect that is flexible only on word i.e. 32-bit boundaries. However, since the TigerSHARC does supports operations on bytes, we will design the FPDAU to support byte-wide granularity. The TigerSHARC register file has two 64-bit read ports, as shown in Fig. 8. The FPDAU needs to select one of 16 bytes from the register file and connect each of those bytes to a byte input of the computational unit. This interconnect can be built with 128 16-to-1 multiplexers. The dynamically programmable state machine inside the FPDAU controls the configurations of the multiplexers. As far as the impact of the FPDAU on the DSP critical path goes, there is a delay of an additional 16:1 multiplexer which does endanger the 300 MHz operating frequency of the TigerSHARC DSP. In the future as we move to finer geometries, we expect this to be less of an issue. We propose an identical FPDAU in both of the independent computational blocks of the TigerSHARC DSP.

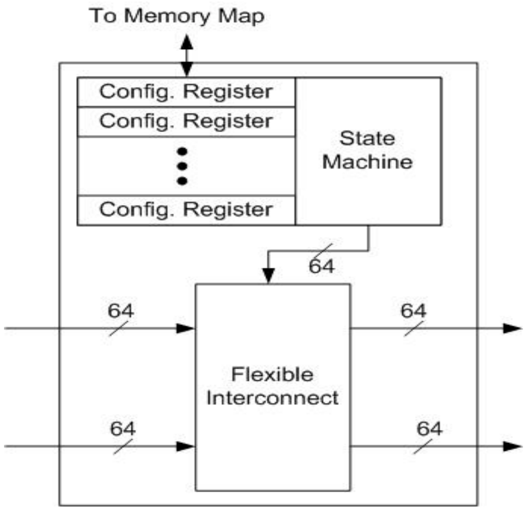


Fig. 9. FP-DAU Block Diagram

Next, the design of the dynamically programmable controller or the state machine shown at the top of Fig. 9 is described. The purpose of the controller is to define the configuration of the flexible interconnect of the FPDAU. This controller will be realized on traditional LUT-based fabric to give it maximum flexibility. This state machine will be clocked by the *read enable* signal of the TigerSHARC register file. In order to minimize the impact of the FPDAU on the instruction set architecture we require that the state machine does not have any additional inputs. Therefore, every time that the read enable is clocked and the FPDAU is active, the state machine will proceed to the next state. This has two consequences. First, we need as many states as there are register reads in the inner most loop of the algorithms that utilize the FPDAU. Secondly, it precludes us from using the FP-DAU in inner loops that have non-linear flow, such as branches or jumps.

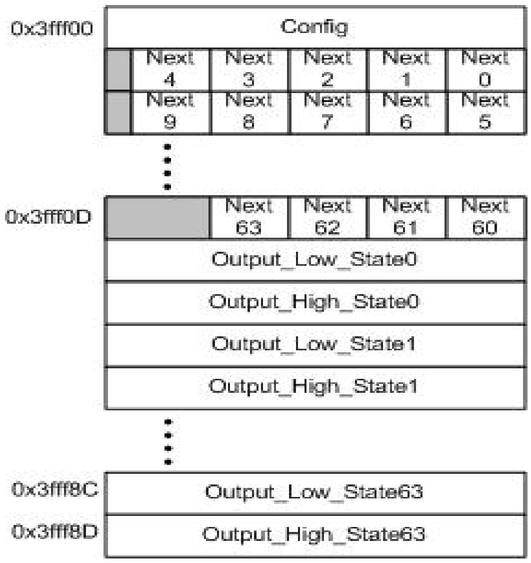


Fig. 10. FP-DAU memory map

However, with predicated execution and the tight loops in DSP kernels this is not much of a restriction. Note that this is a design decision to minimize the impact of the FPDAU on the instruction set architecture. If one has the ability to slightly modify the instruction set architecture (define new opcodes) more efficient and more general-purpose programmable state machines can be realized inside the FPDAU, without the restrictions listed above.

The configuration of the state machine has to be generated during the compilation of the application to the DSP processor. This will allow the data flow between the register file and the ALU to change (in customized way) every clock. The configuration space is memory mapped into the TigerSHARC's internal memory address space, as shown in Fig. 10. This allows the state of the programmable logic to be saved to memory, and also allows new states to be saved and restored by the TigerSHARC. In addition, the FPDAU needs a control register (one bit) that defines whether the FPDAU is active or not.

As noted in the beginning of the paper, the main objective of our work is to minimize the amount of programmable logic to achieve a certain level of performance improvement. That is why we did not advocate a new functional unit or a coprocessor to execution. So, how much area is required for the FPDAU? This requires the estimation of the area for the programmable state machine that is implemented in LUTs. For flexible interconnect structure (that gives us byte-wide data realignment), we need 128 16:1 multiplexers that results in 64 bits for each state of the state machine. Let us assume we have 64 states for the state machine, which should be sufficient to cover a wide range of applications (the inner loops for most applications have fewer than 64 instructions). Each state must have 6 bits to indicate which is the next state. Fig. 10 shows how the state machine of the FP-DAU is memory mapped into the TigerSHARC architecture. Table 2 summarizes the overhead of the FPDAU. The maximum initialization overhead should only be incurred if all 64 states of the

FPDAU's controller are used. The start overhead is the overhead of writing to the configuration register of the FPDAU's controller to start or stop the operation of the FPDAU. The hardware overhead cost is relatively minor when compared to a typical DSP die area of about 1 sq. cm^2. In the hardware overhead, we assumed that the FPDAU's controller is resident inside 4-LUTs. However, the controller could also use configuration SRAM, which would decrease the number of 4-LUTs needed dramatically. Finally, other functions could be included in the FPDAU, like zero/one insertion, bit reversal or any other simple operation. These added functions could marginally increase the needed hardware for the FPDAU, but would allow us to leverage the strengths of programmable logic on a DSP platform.

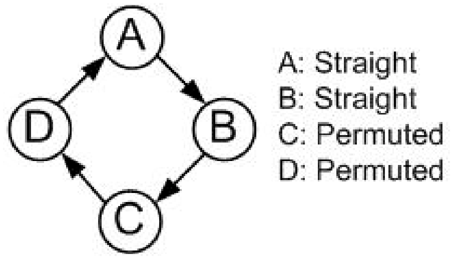
Table 2. FPDAU Overhead

Initialization Overhead	72 cycles, Maximum
Start Overhead	2 cycles per ACS Trellis Frame
Hardware Overhead	128 16:1 Muxes and 200 4-LUTs

3.1 Programming the FPDAU

Next we will describe how the programmer or the compiler uses the FPDAU, using the ACS computation of Viterbi as an example. The configuration for the FPDAU is generated from the register-transfer level assembly code. From the pseudo code shown in Fig. 5 (Viterbi decoder with K=3) we can see that if we omit the permute instructions; we only need four cycles to complete a single stage of the ACS trellis update. Since each of these four instructions accesses the register file, we will need a state-machine with four states to control the flexible interconnect. Note that typically a branch would be executed at the bottom of the loop, but it is omitted from the pseudo code in Fig. 5 for simplicity. Since the branch does not access the register file, it will not clock the state machine so we can ignore it from the perspective of configuring the FPDAU. Figure 11 shows the resultant state machine for Viterbi ACS derived from Figure 7. The register reads for the two add/subtract instructions are done in normal i.e. without any permutation. They are indicated by state A and state B in Figure 11. The two *vector_min* instructions are executed in states C and D of the state machine which requires the FPDAU to program the flexible interconnect to permute the data corresponding to the pattern shown at the bottom of Fig. 6. The new pseudo code required to complete a single stage of the ACS trellis update is also shown in Figure 11, as expected it eliminates the two permute instructions.

Finally, it is important to note that the FP-DAU should be disabled and the configuration of the FP-DAU is saved and restored upon entering interrupt routines. If the FP-DAU is to be utilized inside an interrupt service routine, the states of the FP-DAU must be saved and restored to the TigerSHARC's on-chip memory upon entering and exiting the interrupt, respectively. In most cases the entire configuration memory is not utilized, so the overhead of saving the FPDAU is typically a few cycles, especially given that the TigerSHARC has be ability to read/write 128-bits to memory in a given cycle. However, if the entire configuration space of the FP-DAU does indeed have to be saved, the maximum penalty is around 72 cycles to save the entire FP-DAU state.



```
R15:12 = Add/Subtract(R5:4, R1:0);     // A
R11:8  = Add/Subtract(R3:2, R1:0);     // B
R15:12 = VectorMin(R15:14, R11:10); // C
R11:8  = VectorMin(R13:12, R9:8);     // D
```

Fig. 11. Example FP-DAU Configuration for Viterbi ACS

4 Results from Viterbi Implementation

In this section we will summarize the results of the implementation of the Viterbi decoder on the TigerSHARC enhanced with the FPDAU. As noted before, the programmable logic is configured at compile time i.e. statically by analyzing the kernel of the computation, which in the case of this decoder has 20 instructions. Using the simple compilation scheme described above would translate into 20 states for the FPDAU programmable state machine. The one time overhead of writing to the FP-DAU configuration register and programming the states in the FP-DAU is 16 cycles. Two additional cycles are needed to turn on/off the FP-DAU when entering/exiting the ACS inner loop. Table 3 shows the performance improvements of the TigerSHARC DSP with the FP-DAU on Viterbi decoders of different lengths. Note that the improvement in terms of cycles saved is quite impressive (15 % to 23%) given that the TigerSHARC is already optimized to implement Viterbi efficiently. Also, note improvement also results in improvements to code density, which is quite useful in embedded applications. It may also result in power savings but the FPDAU itself will consume some power but we do not have access to the gate-level netlists of the TigerSHARC to evaluate exactly what the savings would be.

The additional area required for 64 16-to-1 Muxes is Y, incurring a total delay of Z in W process technology. The state machine in the FP-DAU requires the equivalent of X number of CLBs, at an area estimate of A um2 in W process technology.

Table 3. TigerSHARC Viterbi Performance with FP-DAU

L=190 Bits	K=5	K=7	K=9
ACS	1506	3146	6183
Traceback	960	1245	1625
Total Cycles	2466	4391	7808
% Speed Up	15.5%	19.2%	22.6%

5 Other Applications of FPDAU

Even though the focus of this paper was the implementation of a Viterbi decoder, it should be pointed out that the FPDAU concept is quite general and it has many applications. Basically, the FPDAU restores some flexibility of a Vector, VLIW, or SIMD mode processor by allowing the functional units to operate on any data in the register file. Without the FPDAU one has to waste valuable CPU cycles and power in rearranging the data so that a given instruction can execute properly. We have found applications for FPDAU in a variety of DSP applications especially those that involve matrix operations like Reed-Solomon decoding, finding the minimum or maximum in a vector, data interleaving and de-interleaving and matrix transpose. In each of these applications the FPDAU can be used, but exactly how it is used is determined by the hardware/software co-design of the application, as illustrated in this example. The interface and the programming model of the FPDAU will be the same but the configuration of the state machine will be different in each case and depending on the application the amount of improvement will also vary. For example, in an experiment with matrix transpose on the AltiVec we found that only half the merge instructions in the inner loop can be eliminated with the FPDAU. So, it is important to note that *not all permute (or data rearrangement) operations* can be eliminated with the FPDAU; this is the trade-off between the amount of configurable logic inside the FPDAU and its interface and the amount of flexibility. We deliberate restrict the inputs to the FPDAU to two and byte-level reconfigurability to minimize the area overhead of the FPDAU and its impact on the critical path of the processor.

6 Related Work and Conclusions

The idea of utilizing field programmable logic to accelerate computations is not new. Starting with the PRISC project in Harvard [13] and the work in BYU[3] on integration of DSP and reconfigurable logic and more recently the reconfigurable functional unit idea in the Chimera project in Northwestern University[2], there have been numerous efforts at integrating programmable logic with a processor. The key difference between those efforts and the proposed solution is in two areas (a) we treat DSP + programmable logic integration as a hardware/software co-design problem, hence what we propose is a methodology rather than a specific solution. So, it can be applied to any processor and any application (b) unlike the previous efforts we focus on the cost issue, which precludes us from using a co-processor or a new functional unit because that would add to the cost and change the instruction set architecture of the underlying processor – which poses problems in terms of adoption in embedded processors especially in the commercial arena. We believe that the solution proposed here finds a sweet spot in terms of return on investment in terms of the amount of programmable logic and the improvement in performance achieved. Also, it has minimal impact on the instruction set architecture and the programming model of a DSP, so it can be ignored without significant penalty if the application domain does not required it.

Also, if one has more chip area to spend on the programmable logic the FPDAU can be expanded to include other operations in the LUT area that is currently being used to only implement the programmable state machine. For example, one could have a bit-level operations support that could help in encryption algorithms like DES and AES. So, again the proposal here is a co-design methodology for the DSP + programmable logic platform, where the architect can choose how much chip area to spend on programmable logic and what operations to implement there with the FPDAU providing a general framework for programming and interface. If it is expanded further it will resemble the RFU idea in Chimera or the co-processor concept in the BYU project or Riverside project[1].

References

1 Stitt, G. and Vahid, F.: Energy Advantages of Microprocessor Platforms with On-chip Configurable Logic. IEEE Design and Test, 2002

2 Ye, Z., Moshovos, A., Hauck, S., Banerjee, P.: CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit, Computer Architecture News, (2000).

3 Graham, P., Nelson, B.: Reconfigurable Processors for High-Performance, Embedded Digital Signal Processing, Field Programmable Logic and Applications. (1999)

4 Fisher, J., Faraboschi, P., Desoli, G.: Custom-Fit Processors: Letting Applications Define Architectures. Hewlett-Packard Laboratories Cambridge, Cambridge, MA, (1996)

5 Compton, K., Hauck, S.: Reconfigurable Computing: A Survey of Systems and Software, http://www.ee.washington.edu/faculty/hauck/publications/ConfigCompute.pdf

6 Dehon, A.: The Density Advantage of Configurable Computing. IEEE Computer Magazine, (2000)

7 Tessier, R., Burleson, R.: Reconfigurable Computing for Digital Signal Processing: A Survey. Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology, (2001)

8 Hartenstein, R.: Reconfigurable Computing: A New Business Model – and it's Impact on SoC Design. Proceedings Euromicro Symposium on Digital Systems Design. IEEE Comput. Soc. (2001)

9 Ollmann, I.: Altivec. http://www.simdtech.org/apps/group_public/documents.php

10 Analog Devices: TigerSHARC DSP Hardware Specification. http://www.analog.com/Analog_Root/static/library/dspManuals/Tigersharc_hardware.html

11 Fridman, J.: Data Alignment for Sub-Word Parallelism in DSP. IEEE Signal Processing Magazine, IEEE, (2000). p.27-35.

12 Texas Instruments: TMS320C6000 CPU and Instruction Set Reference Guide. (2000), http://www-s.ti.com/sc/psheets/spru189f/spru189f.pdf

13 Razdan, R., Smith, M.: High-Performance Microarchitectures with Hardware-Programmable Functional Units, Proc. 27th Annual IEEE/ACM Intl. Symp. on Microarchitecture, pp. 172-180, November (1994)