International Journal of Computer Science and Applications, 2008, Vol. 5, No. 3a, pp 20 - 32

ALTIVEC VECTOR UNIT CUSTOMIZATION FOR EMBEDDED SYSTEMS

TARIK SAIDANI

Fundamental Electronics Institute, University of Paris South 11, 91405 , Orsay cedex , France
tarik.saidani@u-psud.fr

http://www.ief.u-psud.fr/~saidani

JOEL FALCOU

Fundamental Electronics Institute, University of Paris South 11, 91405 , Orsay cedex, France
joel.falcou@u-psud.fr

http://www.ief.u-psud.fr/~falcou

LIONEL LACASSAGNE

Fundamental Electronics Institute, University of Paris South 11, 91405 , Orsay cedex, France
lionel.lacassagne@u-psud.fr

http://www.ief.u-psud.fr/~lacas

SAMIR BOUAZIZ

Fundamental Electronics Institute, University of Paris South 11, 91405 , Orsay cedex, France

samir.bouaziz@u-psud.fr

Abstract

Vector extensions for general purpose processors are an efficient feature to address the growing performance demand of multimedia and computer vision applications. Embedded processors are the most widespread architectures for such applications. While providing sufficient computing power for these applications, they must take into account power, area and real-time constraints. In this paper, we propose two hardware optimization techniques to address those constraints: *RISCization* and instruction set customization. Experimental results show that those techniques both reduce time and power consumption by up to 50% when compared to the original ISA.

Keywords

SIMD instruction set; altivec ; vectorization; embedded systems;processor;customization; high performance image processing; power efficient architectures

1. Introduction

SIMD extensions were developed to face the growing demand of computing power of general purpose processors (GPP) coming from multimedia and gaming applications. They started appearing in 1994 in HP's MAX2 and Sun's VS extensions and can now be found in most of the GPPs. The vector instruction set exploits the data level parallelism (DLP) present in this kind of applications. Intel introduced MMX [Peleg (1996)], then SSE, SSE2 and SSE3 extensions for the Pentium processors, Freescale developed the Altivec [Diefendorff (2000)] unit on PowerPCs. These extensions contain 128-bit registers which provides 16-way 8-way and 4-way data level parallelism. The embedded processors are designed to perform a set of tasks under several constraints on the SoC (Sytem on Chip) area, execution time and power consumption. Since the workload in embedded applications become similar to PC applications those constraints become critical. Computer vision algorithms are a class of data intensive applications that we consider in this paper. They are characterized by regular operations on large sets of data, and composed of a combination of convolution kernels and other arithmetic operations. Image processing applications imply a large memory transaction per computation ratio and must satisfy real-time constraints in the context of video flow. Therefore, SIMD extensions are good candidates for improving the performances of those applications. However, when trying to extend the SIMD paradigm to embedded systems there are several hardware barriers, in particular for area and power consumption.

In this paper we apply some hardware optimization techniques to make the Altivec unit fits the embedded systems constraints, when making some assumptions about the application domain. The first one that we called *RISCization* consists in reducing the complexity of the instructions to increase the operating frequency and therefore to decrease the processing time. The second optimization takes benefit from the versatility of the Altivec ISA by restricting their functionalities. The remainder of the paper is organized as follows. Related work is discussed in Section 2. Section 3 provides the motivation of our work by demonstrating the efficiency of the SIMD units in processing computer vision typical operators. Section 4 introduces the *RISCization* concept with hardware implementation on a FPGA. The instruction customization technique is described in Section 5, and conclusions are drawn in Section 6.

2. Related Work

A lot of research work has been done in instruction set customization, a co-processor is automatically synthesized to accomplish a portion of a code where the program spend the major part of its execution time in [Athanas, P. M. (1993)]. In [Sun (2004)] an automatic methodology to select custom instructions to augment an extensible processors is described, in order to maximize its efficiency for a given application program. The author's methodology features cost functions to guide the custom instruction selection process, as well as static and dynamic pruning techniques to eliminate inferior parts of the design space from consideration. A Dynamic Instruction Set Computer (DISC) has

been developed in [Wirthlin (1995)], instructions occupy FPGA resources only when needed and FPGA resources can be reused to implement an arbitrary number of performance-enhancing application-specific instructions. The SIMD extension case was treated in [Chouliaras (2008)]. In most of the articles cited above the power consumption problem was addressed as a parameter for validating the approach, but tuning this power consumption was not actually considered.

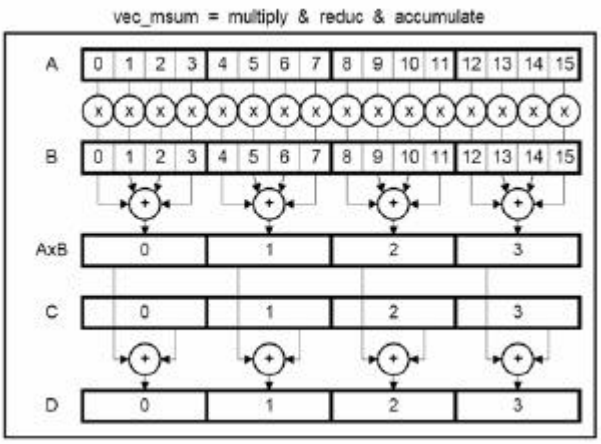

Fig.1. Altivec **`vec_msum`** instruction

In our paper we provide two optimization mechanisms which aims to reduce the power consumption in image processing operators.

3. Software SIMD Implementation

3.1. *Architecture Presentation*

AltiVec is an extension designed to enhance PowerPC processor performance on applications handling large amounts of data. The AltiVec architecture is based on a SIMD processing unit integrated with the PowerPC architecture. It introduces a new set of 128 bit wide registers distinct from the existing general purpose or floating-point registers. These registers are accessible through 160 new vector instructions that can be freely mixed with other instructions (there are no restriction on how vector instructions can be intermixed with branch, integer or floating-point instructions with no context switching nor overhead for doing so). Altivec handles data as 128-bit vectors that can contain sixteen 8-bit integers, eight 16-bit integers, four 32-bit integers or four 32-bit floating-point values. For example, any vector operation performed on a vector char is in fact performed on sixteen char simultaneously and is theoretically running sixteen times

faster as the scalar equivalent operation. AltiVec vector functions cover a large spectrum, extending from simple arithmetic functions (additions, subtractions) to boolean evaluation or lookup table solving.

The AltiVec ISA is more complete ISA than Intel SSE2 and SSE3, and provides some instructions like **`vec_msum`** which is very useful for dot product and FIR (Finite Impulse Response) computations. The synopsis of the instruction **`d=vec_msum(a; b; c)`** [Freescale (1999)] is given in Fig. 1. Eight bit multiplications are performed to provide

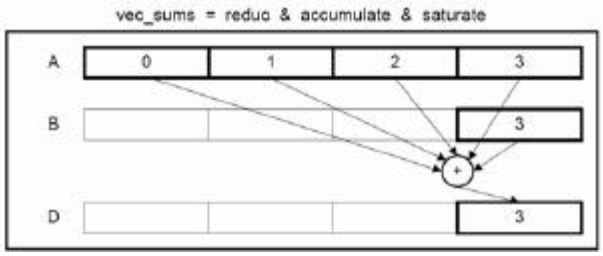

Fig. 2. 4-block reduction with **`vec_sums`** instruction

intermediate products that are accumulated through a 4-block reduction inside 32-bit blocks: **`P = reduc4(A:B)`**. The second stage of the instruction performs an accumulation of P with a third register

`D = reduc4(reduc4(A:B) + C).`

From a scalar point of view, the instruction performs sixteen 8-bit multiplications, eight 16-bit and four 32-bit sums, which gives a total of 36 scalar instructions inside a unique SIMD instruction. The 4-block reduction step makes this instruction well suited for filters whose size is a multiple of 4. One can note that for dot product, a second instruction **`vec_sums`** should be used to reduce the four 32-bit blocks D0;D1;D2;D3 inside one block (Fig. 2).

3.2. *Software Benchmark Results*

In order to demonstrate the benefits of vectorizing computer vision codes [Ollmann (2001)], we compared scalar and SIMD implementations of basic image processing operators: dot product and FIR. Therefore we performed software benchmarks on the scalar and SIMD versions of the code. The metric that we used in our experiments is the number of clock cycles per pixel *cpp*.

$$cpp = \frac{t.F}{N^2}. \qquad (1)$$

Where N is the width of a square input image, t the operator execution time in seconds and F the clock's frequency of the processor in Hertz (1Ghz on the PowerPC G4). Measuring the *cpp* to compare the different implementations is a fair comparison, since it

does not depend on the processor's frequency. Moreover, *cpp* is a relevant cache miss detector. In our benchmark we consider a data size varying from 128x128 to 1024x1024.

The first benchmarked operator is dot product, which operates on two vectors of size N2, and computes a scalar according to the equation:

$$a.b = \sum_{i=0}^{N^2-1} a_i.b_i \,. \tag{2}$$

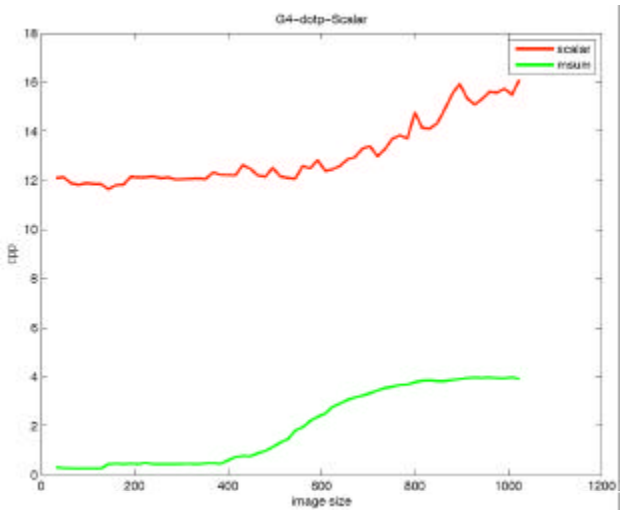

Fig. 3. *cpp* of dot product on PowerPC G4

Listing 1. dot product function using **`vec_msum`**

```
vector int dot_pr_msum(vector unsigned char *a , vector unsigned char *b, unsigned
int N)
{
        unsigned int i;
        vector unsigned char X, Y;
        vector int s, s_r;
        vector int zero_vector_32=(vector int)(0);
        s = zero_vector_32;
        for(i =0; i<N/16; i++){
          X=vec_ld (0, &a[i]);
          Y=vec_ld (0, &b[i]);
          s = (vector int) vec_msum(X,Y, (vector unsigned int)s);
        }
        s_r=vec_sums(s ,zero_vector_32);
        return s_r;


}
```

Listing 3 provides a comparison between the scalar and the SIMD version of dot product, using the dedicated instruction **`vec_msum`**. While the gap between the scalar and SIMD

implementation is almost the same (? ≅ 12), the speedup varies from x45 for small data sizes to x4.1 for large ones. We then conclude that the gain provided by vectorization is limited by the cache size.

The second basic image processing operator that we vectorized is a 4-tap FIR filter:

$$Y(i,j) = \sum_{k=0}^{3} f(k).X(i, j+k). \quad (3)$$

We consider a 4-tap FIR to fully exploit the DLP offered by the 16-byte registers. As we observed for the dot product, and for the same reasons, the gap between the scalar and vector implementations is constant (? ≅ 12). Moreover, the speedup varies from x14.9 for small images to x4.4 for large ones.

4. RISCization

The efficiency of vector units in processing computer vision algorithms being demonstrated, we can present the hardware optimization techniques that we performed to

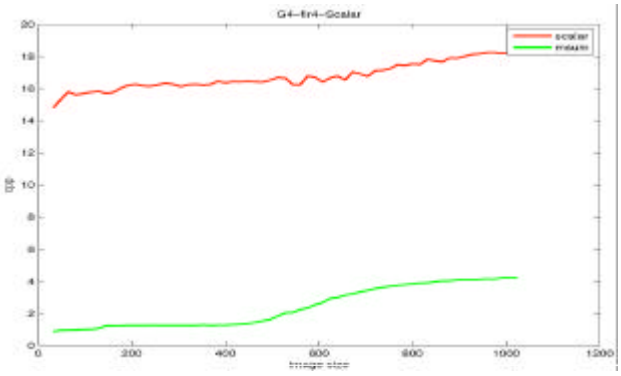

Fig. 4. *cpp* of FIR4 on PowerPC G4.

make the AltiVec instructions more convenient for embedded architectures. The first one that we called *RISCization* consists in replacing a complex SIMD instruction (*CISC*), with a set of simpler (*RISC*) instructions from the same ISA. First, this technique is implemented in software to see if it changes drastically the time performances of the algorithm. There are several ways to split the **`vec_msum`** instruction into a set of *RISC* instructions. For instance, by replacing 8-bit by 16-bit multiplications (16-bitx16-bit →16-bit) or by replacing 4-block reductions by 4 separate accumulators. The most simplified equivalent set of instructions is given in 5 and is described below:

(1) the input 8-bit operands A,B are converted into two 16-bit data (low part and high part).

(2) a 16-bit block wise multiplication is performed: neither overow nor truncation are necessary since input data are 8-bit wide.
(3) conversion to 32-bit blocks.
(4) 32-bit accumulation.
(5) reduction (using **`vec_sums`**) to sum the four 32-bit blocks.

Fig. 6 and Fig. 7 present the best *cpp* for three CISC and RISC versions. Several versions have been implemented (For instance with various schemes to replace reduction instructions.

Only three are presented, one for each multiplication instruction of AltiVec ISA: **`vec_msum`**, **`vec_mladd`** and **`vec_mule/vec_mulo`**.

Listing 2. FIR Filter vectorization

```
vector unsigned char **fir_vec(vector unsigned char** IM_OR, vector unsigned char**
IM_RES, int64 nrl, int64 nrh, int64 ncl, int64 nch, unsigned char*fir, int64 fsize)
{
        int64 i,j;
        int k;
        vector unsigned char im_or0,im_or1,im_or, im_res;
        vector unsigned char fir_vector;
        fir_vector.v=(vector unsigned char)(0);
        vector unsigned char dup_vector;
        dup_vector=(vector unsigned char)(0,1,2,3,0,1,2,3,0,1,2,3,0,1,2,3);
        vector unsigned int temp;
        vector unsigned int vector_zero32;
        vector_zero32=(vector unsigned int)(0);
        vector unsigned int acc00,acc01,acc10,acc11;
        vector unsigned short acc0,acc1;
        vector unsigned char res;
        vector unsigned short shifter;
        shifter=(vector unsigned short)(8);
        vector unsigned char permuter;
        permuter=(vector unsigned char)(0,4,8,12,1,5,9,13,2,6,10,14,3,7,11,15);
        for(k=0;k<fsize;k++){
                fir_vector.t[k]=fir[k];
                fir_vector.t[k]=fir[k];
                fir_vector.t[k]=fir[k];
                fir_vector.t[k]=fir[k];
        }
        fir_vector.v=vec_perm(fir_vector.v,fir_vector.v,dup_vector);
        for(i=nrl;i<=nrh;i++){
                for(j=ncl;j<nch;j++){
                        im_or0=vec_ld(0, &IM_OR[i][j]);
                        im_or1=vec_ld(15,&IM_OR[i][j]);
                        im_or=vec_sld(im_or0,im_or1,0);
                        acc00=vec_msum(im_or,fir_vector.v,vector_zero32);
                        im_or=vec_sld(im_or0,im_or1,1);
                        acc01=vec_msum(im_or,fir_vector.v,vector_zero32);
                        im_or=vec_sld(im_or0,im_or1,2);
                        acc10=vec_msum(im_or,fir_vector.v,vector_zero32);
                        im_or=vec_sld(im_or0,im_or1,3);
                        acc11=vec_msum(im_or,fir_vector.v,vector_zero32);
                        acc0=vec_sr(vec_pack(acc00,acc01),shifter);
                        acc1=vec_sr(vec_pack(acc10,acc11),shifter);
                        res=vec_pack(acc0,acc1);
                        IM_RES[i][j]=vec_perm(res,res,permuter);
                }
        }
   return IM_RES;
```

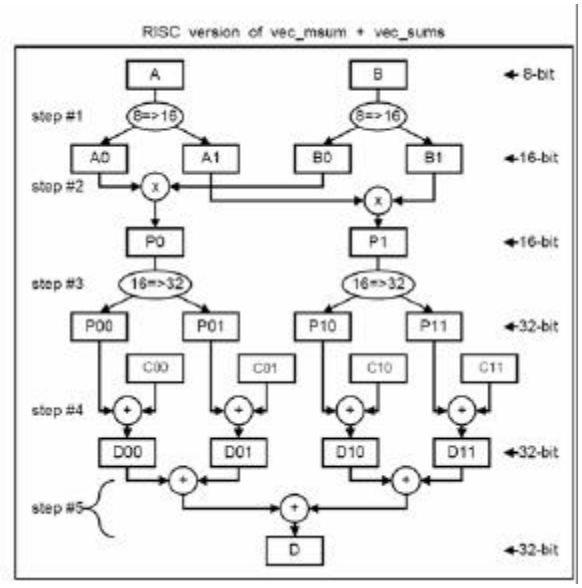

Fig. 5. **vec_msum** *RISCization*

- a CISC version with the **vec_msum** reduction instruction,
- a RISC first version, with the **vec_mladd** 16-bit multiplication-accumulation instruction: D = A xB + C,
- a second RISC second version , with only the vec-mule and vec-mulo 8-bit multiplication (8-bitx8-bit→16-bit) instructions.

For RISC versions, reductions are replaced by a set of additions (vec-add) and permutations (**vec_perm**). Obviously the *RISC* versions are slower than the *CISC* one since more complex computing is done at the same throughput (1 instruction/cycle). However, we can not assert that it is the case on hardware since operating frequency depends on the instruction complexity.

```
vector int dot_pr_risc(vector unsigned char *a, vector unsigned char *b,
        unsigned int N)
{
        unsigned int i;
        vector unsigned char X,Y;
        vector short A0,A1,B0,B1,zero_vector16;
        vector short P0,P1;
        vector int P00,P01,P10,P11,AC0,AC1,AC,sr;
        vector int zero_vector32=(vector int)(0);
        zero_vector16=(vector short)(0);
```

```
        AC=(vector int)(0);
        AC0=(vector int)(0);
        AC1=(vector int)(0);


        for(i=0;i<N/16;i++)
        {
                X=vec_ld(0, &a->v[i]);
                Y=vec_ld(0, &b->v[i]);

                A0=vec_unpackh((vector char)X);
                A1=vec_unpackl((vector char)X);
                B0=vec_unpackh((vector char)Y);
                B1=vec_unpackl((vector char)Y);

                P0=vec_mladd(A0,B0,zero_vector16);
                P1=vec_mladd(A1,B1,zero_vector16);

                P00=vec_unpackh((vector short)P0);
                P01=vec_unpackl((vector short)P0);
                P10=vec_unpackh((vector short)P1);
                P11=vec_unpackl((vector short)P1);

                AC0=vec_add(P00,P01);
                AC1=vec_add(P10,P11);

                AC=vec_add(AC,vec_add(AC0,AC1));
        }

        sr=vec_sums(AC,zero_vector32);
        return sr;


}
```

Listing 3. example code of dot product *RISCization*

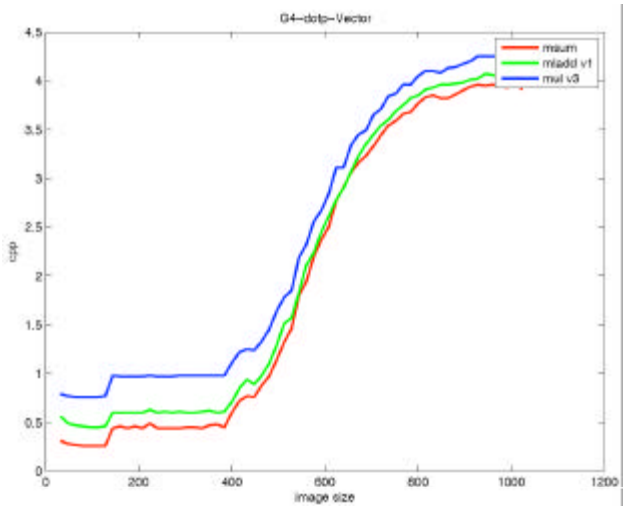

Fig. 6. *cpp* of dot product on PowerPC G4

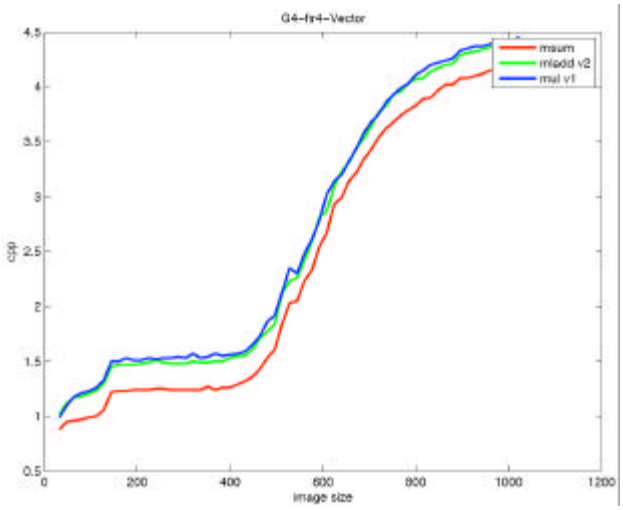

Fig. 7. *cpp* of FIR Filter on PowerPC G4

5. Hardware Implementation

In order to estimate the impact of *RISCization*, SIMD instructions have been synthesized on a Virtex4 [Xilinix (2007)] FPGA. *RISC* and *CISC* blocks combinations have been compared through orthogonal criteria like area utilization, running frequency *F*, power consumption *P* and execution time *t*. While *cpp* is meaningful for software benchmarks, *t* should be preferred in the case of hardware benchmarks. Since the running frequency depends on the implemented operators, the fastest implementation (with the smallest *t*) is not necessarily the one with the smallest *cpp*. Two assumptions have been made for the comparison: FPGA and the PowerPC G4 have the same AltiVec instructions latencies and the FPGA could be interfaced with a memory hierarchy whose specifications (cache size, associativity, and latencies) would be equivalent to those of PowerPC G4 ones. Using *Xilinx ISE* and *XPower Estimator* tools, power consumptions have been estimated for each standalone SIMD AltiVec instruction (these measures do not take into account the consumption of the I/O blocks). Tables 1 and 2 present the results for 256x256, 512x512 and 1024x1024 data size.

Table 1. dot product synthesis results.

Version	***cpp***	***Area %***	***F (MHz)***	***P (Watt)***	***t (ms)***	***E (µJ)***
256x256 data						
msum	0.44	22.5	151	0.202	0.191	39
mladd	0.61	24.2	256	0.165	0.156	26
mul	0.97	23.5	209	0.110	0.303	33
512x512 data						

msum	1.32	22.5	151	0.202	2.292	463
mladd	1.51	24.2	256	0.165	1.545	255
mul	1.78	23.5	209	0.110	2.227	245
1024x1024 data						
msum	3.91	22.5	151	0.202	27.154	5485
mladd	3.97	24.2	256	0.165	16.251	2681
mul	4.19	23.5	209	0.110	20.969	2307

Table 2. FIR synthesis results.

Version	***cpp***	***Area %***	***F (MHz)***	***P (Watt)***	***t (ms)***	***E (µJ)***
256x256 data						
msum	1.25	20.5	164	0.213	0.499	106
mladd	1.49	21.2	337	0.258	0.290	75
mul	1.53	21.5	337	0.264	0.298	79
512x512 data						
msum	1.84	20.5	164	0.213	2.940	626
mladd	2.12	21.2	337	0.258	1.650	426
mul	2.13	21.5	337	0.264	1.658	438
1024x1024 data						
msum	1.84	20.5	164	0.213	26.908	5731
mladd	2.12	21.2	337	0.258	13.700	3535
mul	2.13	21.5	337	0.264	13.856	3658

In order to estimate the efficiency of the embedded system, we measure the amount of energy E = t×P required to compute the algorithm. Once the processing speed is enforced (typically t < 33ms for 30 images/s), the total energy E is the constraint to optimize, not only the power P. One can note also, that synthesis tools provide the maximum running frequency. If the overall algorithm implementation is too fast, implying too high power consumption, the system can be down-clocked. First, if we look at the execution time, we can see that the CISC version is never the fastest one, but **`vec_mladd`** which is a complexity golden mean between **`vec_msum`** and **`vec_mule/vec_mulo`**. Gains vary from x1.2 up to x1.6 for dot product and from x1.7 up to x2.0 for FIR. These results are very important: from the execution time point of view, a direct implementation into an FPGA of the classic dedicated and "optimized-for" *CISC* instruction will lead to a non optimal implementation.

Second, if we focus on energy, the gains between *CISC* and *RISC* version range from x1.6 up to x2.0 for dot product and from x1.4 up to x1.6 for FIR. That validates our *RISCization* approach: usually the fastest implementation is also the most power hungry one, but it the case of mapping SIMD instruction from a general purpose processor to an FPGA, this is not the case: RISC implementations are quick and energy efficient!

6. Instructions Customization

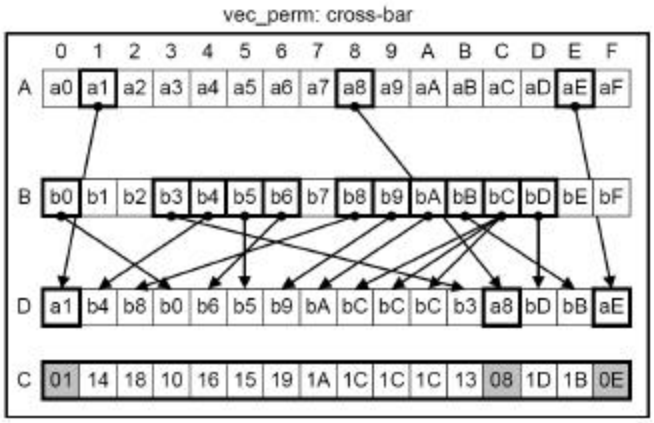

Fig. 8. AltiVec permutation instruction

The second hardware optimization that we performed on our kernels is instruction customization. This technique is complementary to *RISCization* as it reduces the size of those instructions used in the implemented operators. Thus it compensates the growth of instruction number caused by *RISCization*. Instruction customization reduces the useless capabilities of versatile instructions. The AltiVec instruction **`vec_perm`** (synopsis 8) is an interesting example. This instruction behaves like a complete crossbar and not only performs permutations with regular patterns (as SSE2+ instructions can do), but also with irregular patterns. For a given algorithm or application, only a few set of these capabilities are useful. For dot product and FIR, the only need is to construct unaligned vector registers. By specializing, such an instruction, area size and power consumption can be drastically reduced. Specialization has been applied to the multiply-add operation **`vec_mladd`**. The instruction is split into two instructions: **`vec_mul`** that does not exist in Altivec ISA in 16-bit version, and vec_add (vec_mladd(A,B,C)=vec_add(vec_mul(A,B),C)).

Table 3. gain provided by customization

Instruction	*Area Reduction*	*Power Reduction*
`vec_sums`	32 %	1 %
`vec_msum`	32 %	3 %
`vec_mladd`	22 %	3 %
`vec_sum4s`	30 %	3 %
`vec_perm(v1)`	56 %	5 %
`vec_perm(v2)`	35 %	5 %

Table 3 presents the area and power consumption reduction provided by instruction specialization. These results was obtained with a synthesis on a Virtex-4 FPGA. One can note that power gain is not important because the difference between versions resides in the logical blocks; these blocks are not great power consumers. However, customization can significantly reduce area occupation. As SIMD instructions are quite big they can prevent from completing a synthesis. Area reduction is the only solution to make all the required SIMD instruction fit into an FPGA.

7. Conclusion

We have presented the design of an AltiVec SIMD instruction unit for FPGAs. The major advantages of designing an AltiVec compatible unit is to be able to directly reuse PowerPC-aimed C code, without modifying it, into an FPGA, nor adding bugs, and thus reducing development time. We have used optimization techniques like instruction specialization and presented a new one called *RISCization* applying the CISC-to-RISC concept to hardware instruction implementation. Impacts of *RISCization* on basic signal processing algorithms are very interesting: the energy consumption has been reduced by a factor ranging from x1.4 up to x2.0 and area has been also reduced by a factor ranging from x1.3 up to x1.5 thanks to instruction customization. Current and future works are to develop high-level tools to perform an automatic design space exploration of the configurations to efficiently implement AltiVec coded applications into an FPGA.

References

[1] Athanas, P.M., Silverman, H.F. (1993): *Processor reconfiguration through instruction-set metamorphosis*, IEEE Computer 26(3) pp. 11–18.

[2] Chouliaras, V.A. *et al.* (2008): *Customization of an embedded risc cpu with simd extensions for video encoding: A case study*. Integr, VLSI J. 41(1) pp. 135–152.

[3] Diefendorff, K. *et al* (2000): *Altivec extension to powerpc accelerates media processing*. IEEE Micro 20(2) pp. 85–95.

[4] FreescaleSemiconductor (1999): *AltiVec Technology Programming Interface Manual*.

[5] Ollmann, I. (2001): Practical altivec strategies: *The why, how and when of optimization success and failure using altivec*. Technical report, Apple.

[6] Peleg, A., Weiser, U. (1996): *MMX technology extension to the intel architecture*. IEEE Micro 16(4) pp. 42–50.

[7] Sun, F. *et al.* (2004): *Custom-instruction synthesis for extensible processor platforms*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 23, pp. 216–228

[8] Wirthlin, M.J., Hutchings, B.L. (1995): *DISC: the dynamic instruction set computer. In Schewel, J., ed.: Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing*, Proc. SPIE 2607, Bellingham, WA, SPIE – The International Society for Optical Engineeringpp. 92–103.

[9] Xilinx (2007): *Virtex-4 family overview*, Technical report, Xilinx.