Customizing 16-bit floating point instructions on a NIOS II processor for FPGA image and media processing

Daniel Etiemble *, Samir Bouaziz ** and Lionel Lacassagne**
*LRI, ** IEF, University of Paris Sud
91405 Orsay, France
{de@lri.fr, Samir.Bouaziz@ief.u-psud.fr ; lionel.lacassagne@ief.u-psud.fr }

Abstract

We have implemented customized SIMD 16-bit floating point instructions on a NIOS II processor. On several image processing and media benchmarks for which the accuracy and dynamic range of this format is sufficient, a speed-up ranging from 1.5 to more than 2 is obtained versus the integer implementation. The hardware overhead remains limited and is compatible with the capacities of today FPGAs.

1. Introduction

Graphics and media applications have become the dominant ones for general purpose or embedded microprocessors. While some applications need the dynamic range and accuracy of 32-bit FP numbers, a general trend is to replace FP by integer computations for better performance in embedded applications for which hardware resources are limited. In this paper, we show that 16-bit FP computations can produce a significant performance advantage over integer ones for significant image processing benchmarks using FPGA with soft core processors while limiting the hardware overhead. By customizing SIMD 16-bit instructions, we significantly improve performance over integer computations without needing the hardware cost of 32-bit FP operators.

1.1 16-bit floating point formats

16-bit floating formats have been defined for some DSP processors, but rarely used. Recently, a 16-bit floating point format has been introduced in the OpenEXP format [1] and in the Cg language [2] defined by NVIDIA. This format, called "half", is presented in Figure 1.

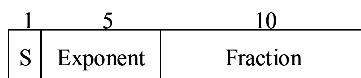

Figure 1: NVIDIA "half" format

A number is interpreted exactly as in the other IEEE FP formats. The exponent is biased with an excess value of 15. Value 0 is reserved for the representation of 0 (Fraction =0) and of the denormalized numbers (Fraction ≠ 0). Value 31 is reserved for representing infinite (Fraction = 0) and NaN (Fraction ≠ 0). For 0<E<31, the general equation for calculating the value in a floating point number is $(-1)^S$ x (1.fraction) x $2^{(\text{Exponent field-15})}$. The range of the format extends from $2^{-24} = 6 \times 10^{-8}$ and $(2^{16}-2^5) = 65504$. In the remaining part of this paper, the 16-bit floating point format will be called *half* or F16. We have considered 16-bit operations for general purpose processors in previous papers [3, 4]. For all applications for which this format is useful, we have shown that a simplified version of the half format gives similar results compared to the 16-bit version of the IEEE FP formats. By comparing the images resulting from computations with different FP formats (according to PSNR measures), we have shown that denormalized numbers are useless and that rounding towards 0 (truncating the low order bits of the final mantissa after adding or multiplying mantissas) gives similar results to other rounding modes. In other words, the simplest hardware solution is sufficient. In the rest of the paper, the 16-bit FP format will be called F16 and the usual "float" format will be called F32.

1.2 Data format for image and media processing

Image processing generally need both integer and FP formats. Convolution operations with byte inputs need 32-bit integer formats for the intermediary results. Geometric operations need floating point formats. In many cases, using the "half" format would be a good trade-off: the precision and dynamic range of 32-bit FP numbers is not always needed and 16-bit FP computations are compatible with byte storage if efficient byte to/from half format is available.

Research of points of interest within an image is a typical application: the objective is to reduce the image to a limited set of points considered as the most representative of the whole set to be used as an index for this image. Figure 2 shows the Achard and Harris algorithms. They share most computations and differ by the final step. They include a 3x3 Sobel gradient followed by 3x3 Gauss filters. The common part is typical of low level image processing. For integer computations, initial images with levels of gray have unsigned char format to code the pixels. Sobel gradient computations lead to short format to avoid overflow and the following multiplications lead to int. format. The 16-bit floating point numbers keep

the same format all along the computations without any overflow and a SIMD implementation is straightforward.

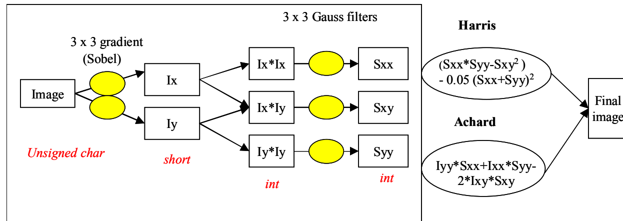

Figure 2: Achard and Harris algorithms for detection of Points of Interest (POI).

Non standard FP formats have been proposed for image and media processing. In [5], Fang et al propose lightweight floating point arithmetic to enable FP signal processing applications in low-power mobile applications. Using IDCT as benchmark, the authors show that FP numbers with 5-bit exponent and 8-bit mantissa are sufficient to get a Peak-Signal-to-Noise-Ratio similar to the PSNR with 32-bit FP numbers. These results illustrate another case for which the "half" format is adequate

1.3 FPGA with soft core processor.

FPGAs with soft core processors are now currently available from many suppliers. Customizing F16 instructions for the soft core is thus a simple approach to consider. Customized instruction-sets for embedded processors have been considered for a while [6]. Recently, transparent customization has been considered for ARM instruction set [7]. In this paper, we consider the customization of SIMD F16 instructions for the NIOS II processor provided by Altera for several boards. According to the different benchmarks, we consider the instructions to customize. Then, we measure the execution times of the different benchmarks with and without these supplementary instructions and we evaluate the corresponding hardware overhead.

2. Methodology

2.1. Description of benchmarks

For image processing, we first consider convolution operators: the horizontal-vertical versions of Deriche filters and Deriche gradient [3, 4]: these filters operate on 2D arrays of pixels (*unsigned char*), do some computation by using integers and deliver byte results. They are representative of spatial filters and have a relatively high computation to memory accesses ratio.

Then, we consider some algorithms that can be considered as intermediate level image processing. Achard and Harris algorithms for the detection of points of interests belong to this category. They have already been introduced in Figure 2. Optical flow algorithms belong to the same category. It is a function which is used to understand the difference between images caused by the motion. Points of interest and Optical flow are mainly used for image stabilization.

For media processing, we consider the FDCT functions of JPEG 6-a which are included in MediaBench [8]. There are three different versions of FDCT and IDCT functions called "integer", "fast integer" and "float". In [3, 4], we have shown that there is no significant difference between the original image and the final image (after coding and decoding) when using F16, integer or F32 formats. It is worthy to evaluate the execution time of the F16 format.

The code for all the benchmarks is provided in [9].

2.2. Hardware and software support

All the experiments have been done with the Altera NIOS development kit (Cyclone Edition) which includes the EP1C20F400C7 FPGA device. We used the NIOS II/f version of the processor, which main features are summarized in Table 1. All information on the device and processor features can be found in [10].

The NIOS II processor has a 50-MHz clock frequency when used with the Cyclone kit. As our benchmarks typically consist in loop nests for which branch outcomes are determined at compile time, the dynamic branch predictor is not useful. For integer computation, adding hardware multiplier and divider has a significant impact. A larger data cache size could also slightly improve performance. There is no hardware FP support: FP computations are done by software.

All the benchmarks have been compiled with the Altera Integrated Development Environment (IDE), which uses the GCC tool chain. –O3 option has been used in release mode. Execution times have been measured with the high_res_timer that provides the number of processor clock cycles for the execution time. Most of the results use the Cycle per Pixel metrics, which is the total number of clock cycles divided by the number of pixels. For each benchmark, the execution time has been measured at least 5 times and we have taken the averaged value.

Table 1: NIOS II/f processor features

Fixed features	Parameterized features
32-bit RISC processor	HW integer multiplication
Branch prediction	HW integer division
Dynamic branch predictor	4 KB instruction cache
Barrel shifter	2 KB data cache

2.3. 16-bit floating point operators

The 16-bit floating point operators have been designed from a VHDL library developed by P. Belanovic [11] for

embedded applications : it includes a 4-cycle pipelined version of an adder and a multiplier without all IEEE format specificities (no denormals, no NaN, etc). It has been written with behavioral VHDL code, which is the best way to profit from the Altera Quartus II compiler ability to exploit the FPGA features and the parameterized library of operators optimized for the FPGA devices. We first corrected some mistakes of the original design. After getting correct implementations of the 4-cycle adder and multiplier, we defined 2-cycle versions of the same operators to get the smaller latency compatible with the 50-MHz clock frequency. To compare performance, 32-bit FP add/sub (4-cycles) and multiplier (3-cycles) circuits have also been implemented.

The 16-bit divider has been implemented the non-pipelined divider provided by another VHDL library [12].

2.4 Customization of instructions

The technique to customize instructions for the NIOS processor is described in [13]. This technique is quite simple. Hardware operators defined with HDL language (VHDL or Verilog) can be introduced between input and output registers of the processor register file. Two types of operators can be defined. The combinational operators are used when the propagation delay is less than the processor cycle time. In that case, the defined interfaces are the 32-bit input data (dataa and datab) and the output data (result). When the propagation delay is greater than the clock cycle time, multi-cycle operations must be used. They have the same data interface than the combinational operators, plus clock and control signals: clk (processor clock), clk_enable, a global reset (reset), a start signal (active when the input data are valid) and a done signal (active when the result is available for the processor). As the processor need a 32-bit data interface, it is natural to define all our instructions as SIMD instructions: each one operates simultaneously on two 16-bit FP operands. This is a big advantage of using F16 operands as it doubles the throughput of operations.

Using the customized instructions in a C program is straightforward. Two types of "define" are used as the instructions can have one or two input operands:

- #define INST1(A) __builtin_custom_ini(Opcode_INSTR1, (A))
- #define INST2 (A, B) __builtin_custom_inii (Opcode_INSTR2, (A), (B))

3. The SIMD 16-bit FP instructions

SIMD F16 instructions include data computation and data manipulation. Load and store instructions use the 32-bit NIOS load and store instructions.

3.1 Definition of SIMD F16 instructions

An image generally consists of 8-bit data, coding levels of gray or each of the three basic colors. Data conversion instructions are thus needed, from/to bytes to/from F16 formats. As a 32-bit access load or store four bytes, two types of conversion instructions are needed, one for the low order bytes of a 32-bit word and the other for the high order bytes. Conversion instructions between 16-bit integer and F16 formats are also needed.

Low level image processing uses a lot of filters that compute a new pixel values according to the neighbor pixel values. The SIMD version of the code for these filters needs to correctly align the SIMD values before SIMD computation. Assuming that j = 0 mod 4, a 32-bit access loads the bytes T[i][j], T[i][j+1], T[i][j+2] and T[i][j+3] while the four neighbors are T[i][j+1], T[i][j+2], T[i][j+3] and T[i][j+4] or T[i][j-1], T[i][j], T[i][j+1] and T[i][j+2] In any case, a special treatment is needed as one byte belongs to a word and the three other ones to another word. The trick is to combine the conversion with the shift as shown in Figure 3. The shift is not for free as some shifts require two memory accesses.

Table 2 presents the different data conversion and manipulation instructions that are needed.

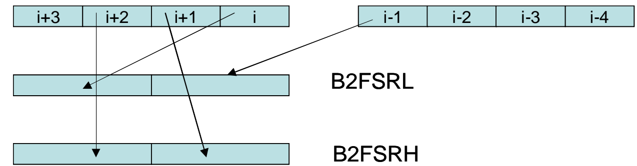

Figure 3: Byte to F16 and shift conversion instructions

Table 2: SIMD conversion, conversion and shift and shift only instructions

INST	Effect
B2F16L	Converts the two lower bytes of a 32-bit word into two F16
B2F16H	Converts the two higher bytes of a 32-bit word into two F16
F2BL	Converts two F16 into two unsigned bytes in the lower part of a 32-bit word
F2BH	Converts two F16 into two unsigned bytes in the higher part of a 32-bit word
S2F16	Converts two 16-bit integers into two F16
F2S	Converts two F16 into two 16-bit integers
B2FSRL	Converts the high order byte of a word and the low order byte of a word into two F16
B2FSRH	Converts the two middle bytes of a word into two F16
FSR (B,A)	Put the low order F16 of word B into high order F16 of results. Put the high order byte of word A into low order F16 of result.

Although the conversion and shift instructions have been defined as shift right instructions, it is easy to show that they can be used for shift left instructions. To shift left one position four bytes belonging to A (one byte) and B (3 bytes), the instruction B2FSRL (A, B) delivers the high part and B2FSRH (B) delivers the lower part. Supplementary mnemonics can be used for readability of right and left shifts without needing more hardware operators.

The arithmetic instructions are given in Table 3. Addition and Subtraction are implemented by a shared two-cycle operator. One control bit selects the operation. All the arithmetic instructions are multi-cycles instruction, except DP2 that divides by a power of 2 just by subtraction on the F16 exponents.

Table 3: SIMD arithmetic instructions

INST	Effect	Cycles
Notation	Word X consists of two F16 (XH and XL)	
ADDF	RL ←AL + BL ; RH ← AH + BH	2
SUBF	RL ←AL – BL ; RH ← AH - BH	2
MULF	RL ←AL * BL ; RH ← AH * BH	2
DIVF	RL ←AL / BL ; RH ← AH / BH	5
DP2	RL ←AL / 2^{BL} ; RH ← AH / 2^{HL}	1

3.2 Hardware cost of the SIMD F16 instructions

The number of logic elements can be used as a rough metrics to evaluate the hardware cost of each customized instructions. Other metrics could also be considered such as the number of connections, but they would basically complicate the comparison without giving more significantly precise information. At least, the "logic element" metrics gives a rough estimation of the chip area that is used. To evaluate the operator's complexity, we use two different percentage values.

The first one is the percentage increase of the number of logic elements compared to the reference version of the basic computing system including the CPU + the main system overhead (JTAG, I/O, timers). This figure corresponds to the overhead resulting from the use of customized instructions versus the reference system. There exists a "custom instruction" overhead that is needed when at least one custom instruction is added to the CPU ISA. It is also interesting to include the overhead corresponding to integer hardware multiplication and division.

The second one is the percentage of logic elements versus the overall number of logic elements available on the FPGA device (Cyclone kit in our experiments). This figure indicates the percentage of "logic element" resources that are lost for the rest of the applications.

The reference version of the NIOS II:f processor uses 2,409 logic elements and the system overhead uses 415 logic elements. The reference version is the one presented in Table 1 without hardware multiplication and division.

Table 4: Hardware cost of "customized instructions"

Operators	LE	Overhead	Use
HW Mul	563	20.6%	2.8%
HW Mul+Div	791	28.9%	3.9%
CI overhead	415	15.2%	2.1%
ADDF/SUBF	439	16.0%	2.2%
MULF	561	20.5%	2.8%
DIVF	962	35.1%	4.8%
DP2	17	0.6%	0.1%
B2FH	36	1.3%	0.2%
B2FL	29	1.1%	0.1%
B2FSRH	33	1.2%	0.2%
B2FSRL	7	0.3%	0.0%
F2BL	66	2.4%	0.3%
F32 ADD/SUB	528	19.3%	2.6%
F32 MUL	1094	40.0%	5.5%

Table 4 gives the hardware cost of the customized instructions. Overhead is the percentage of supplementary logic elements versus the reference computing system. Use is the percentage of logic elements used versus the overall number of logic elements available in the FPGA device.

As expected, the arithmetic operators use most of the extra resources with an overhead of respectively 16%, 21% and 35% for the addition, multiplication and division to add to the CI overhead (15%). The overhead for the other instructions is small: the total overhead for all conversion instructions sums to 6.3%, which is far less than any basic arithmetic instruction (except D2P). Using all the F16 instructions of table leads to 93.7% overhead versus the reference version. Considering the FPGA device use, all the F16 instructions correspond to 12.8% of the LE resources while the computing system is 14.5%. F32 scalar operations use far more hardware resources.

The supplementary F16 resources that are needed look quite reasonable. All the operations are not needed for all benchmarks, as shown in the next section.

3.3 F16 instructions used by the benchmarks

Table 5 shows the instructions that are used by our benchmarks. Not surprisingly, data conversions with and without shifts are present in all the image processing benchmarks, which also use the basic arithmetic instructions (ADDF, SUBF, MULF and DP2). Division is rare: it is only used by the optical flow benchmark. The DCT only use the three main arithmetic operations.

4. Measured results

4.1 Basic loops

We first give the execution time for some basic loops that help to evaluate the actual execution time of the main instructions (load, store, loop overhead, addition, multiplication, etc), either when using the usual integer instructions (with hardware multiplication and division) or when using SIMD F16 instructions. Table 6 gives the corresponding execution time (in clocks per iteration). In the SIMD case, the execution time corresponds to two F16 operations or two iterations of the inner loop when using F16 data. These figures will be useful to explain the measured execution times for the different next benchmarks. They both include the operation execution times and the data access time including the cache effects.

Table 5: F16 instructions used by the different benchmarks (1: Deriche HV; 2: Deriche gradient; 3: Achard; 4: Harris; 5: Optical flow, 6: DCT).

Instructions	1	2	3	4	5	6
ADDF	X	X	X	X	X	X
SUBF		X	X	X	X	X
MULF	X		X	X	X	X
DIVF					X	
DP2			X	X	X	
B2FH	X	X	X	X	X	
B2FL	X	X	X	X	X	
B2FSRH		X	X	X	X	
B2FSRL		X	X	X	X	
F2BL		X	X	X	X	
F2BH		X	X	X	X	

Table 6: Execution time of basic loops (Cycles per iteration (int or F32) or for two iterations (F16)

Loop	N=10	N=100	N=256
X[i] = A[i]	14.2	14.9	18.9
X[i] = i	6.7	6.6	6.25
X[i] = A[i] + B[i]	17.8	23.3	23.6
X[i] = A[i] + k	16	16.1	19.9
X[i] = ADDF16 (A[i] ,B[i])	21.6	26.9	27.6
X[i] = ADDF16 (A[i] ,k)	20.3	20.1	23.9
X[i] = A[i] * B[i]	31	35.1	35.7
X[i] = A[i] * k	27.2	28.1	31.9
X[i] = MULF16 (A[i] ,B[i])	24.4	27.1	27.7
X[i] = MULF16 (A[i] ,k)	18.2	20.1	23.9
X[i] = ADDF32 (A[i] ,B[i])	31.5	27.8	32.5
X[i] = MULF32 (A[i] B[i])	30.3	26.6	31.4

4.2. Deriche benchmarks

The Deriche benchmarks include the horizontal-vertical version of the Deriche filter and the Deriche gradient. They both use two arrays, which mean that the result array is different from the original one. The execution times for the filter are presented in Table 7 for the filter and in Table 8 for the gradient. For the filter, the F16 version is more than 2 times faster than the integer one: it comes from the SIMD instructions, a slightly faster multiplication while the addition is slightly slower and a better cache behavior. The int. version has more cache conflicts (When N=258, CPP = 102). For the gradient, the speed-up is limited to 1.3 for large enough images as SIMD advantage is counterbalanced by a lot of data manipulation and the only arithmetic operation is addition/subtraction which is slower with F16 than with integer operations.

Both for Deriche filter and gradient, the float version is slower than the integer version. First, the cache behavior is worse as the float arrays are four times greater than the byte arrays. Then, the F32 multiplication and addition instructions are scalar and respectively use 3 and 4 cycles. For the other benchmarks, we will not give the F32 results as they cannot compete with the F16 versions when the F16 accuracy and dynamical range are sufficient.

Table 7: Deriche filter execution time (CPP) on an NxN image according to N

N	32	64	128	256
F16	35.6	38.5	38.1	38.0
INT.	89	117	120	122
Speed-up	*2.4*	*3*	*3.1*	*3.2*
F32	105.9	105.3	105	NA

Table 8: Deriche gradient execution time (CPP)

N	32	64	128	256
F16	22.6	25.9	26.8	27.3
INT.	20.8	35	35.4	35.7
Speed-up	*0.9*	*1.3*	*1.3*	*1.3*
F32	70.5	72.4	73.4	NA

4.3 Achard and Harris benchmarks

Tables 10 and 11 give the results for Achard and Harris algorithms. As the algorithms have a significant common part, the results are close and significant of algorithms including a lot of low-level image processing. In both cases, the speed-up is greater than 1.5.

4.4 Optical flow benchmark

The optical flow benchmark, which corresponds to the original algorithm[14], includes a lot of computation. The speed-up is greater than 1.6. Table 11 also shows one advantage of F16 format. The amount of memory that is needed for intermediate is reduced. The 256x256 image can be computed with F16 and the Cyclone board while external memory is insufficient for integer format.

4.5 JPEG DCT

Table 12 gives the execution times of the "int." version implemented in JPEG 6-a. F16 version is similar to the "float" version implemented in JPEG 6-a, except that any float data has been replaced by F16 data. As there is no simple SIMD version of this FP code, we used scalar F16 operators by casting all the 32-bit results to 16-bit values. Even without using the SIMD feature, we have a 1.28 speed-up because the supplementary computations that are needed to control the data range in the integer version are not needed in the FP versions.

Table 9: Achard execution time (CPP)

N	32	64	128	256
F16	171	217	235	245
INT.	293	349	360	366
Speed-up	*1.71*	*1.60*	*1.53*	*1.49*

Table 10: Harris execution time (CPP)

N	32	64	128	256
F16	166	212	230	240
INT.	293	348	359	365
Speed-up	*1.76*	*1.64*	*1.56*	*1.52*

Table 11: Optical flow (CPP) on an NxN image

N	32	64	128	256
F16	137	183	198	207
INT.	293	312	323	NA
Short	318	339	350	356
Speed-up	*2.13*	*1.70*	*1.63*	*NA*

Table 12: JPEG DCT execution times (CPP)

Version	INT.	F16	*Speed-up*
CPP	59	46	*1.28*

7. Concluding remarks

Customizing 16-bit floating point SIMD instructions for the NIOS II processor leads to a significant speed-up for the image and media processing benchmarks for which the accuracy and data range of this format is sufficient. While the SIMD approach doubles the number of operations per iteration, the speed-up generally ranges from 1.5 to more than 2. Data manipulations that are needed for SIMD operations reduce the speed-up but the cache behavior is generally improved as the size of the arrays for the intermediate computations are divided by 2. F16 computations are generally simpler than the corresponding integer ones, as optimized integer code adds specific computations to extend the limited dynamic range of 32-bit integers. This is why the scalar F16 version of JPEG FDCT is faster than the integer version, even when it cannot benefit from the SIMD gain.

The overhead for the SIMD F16 operators remains limited and looks totally compatible with the hardware capabilities of to-day FPGA devices.

8. References

[1] OpenEXP, http://www.openexr.org/details.html

[2] NVIDIA, Cg User's manual, http://developer.nvidia.com/view.asp?IO=cg_toolkit

[3] L. Lacassagne and D. Etiemble, "16-bit floating point operations for low-end and high-end embedded processors", in Digests of ODES-3, March 2005, San Jose.. Available at http://www.ece.vill.edu/~deepu/odes/odes-3_digest.pdf

[4] L. Lacassagne, D. Etiemble, S.A. Ould Kablia, "16-bit floating point instructions for embedded multimedia applications", in Proc. CAMP'05, July 2005, Palermo

[5] F. Fang, Tsuhan Chen, Rob A. Rutenbar, "Lightweight Floating-Point Arithmetic: Case Study of Inverse Discrete Cosine Transform" in EURASIP Journal on Signal Processing, Special Issue on Applied Implementation of DSP and Communication Systems

[6] J.A. Fisher, "Customized Instruction-Sets For Embedded Processors", in Proc. 36th Design Automation Conference, New Orleans, June 1999.

[7] N. Clark, J. Blome, M. Chu, S. Mahke, S. Biles and K. Flautner, "An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors", in Proc. ISCA, Madison, June 2005.

[8] C. Lee, M. Potkonjak, W.H. Mongione-Smith, "Mediabench : A Tool for Evaluating and Synthesizing Multimedia and Communication Systems", Proceeding Micro-30 conference, Research Triangle Park, NC, December 1995.

[9] Benchmark code : http://www.lri.fr/~de/F16/code-NIOS

[10] Altera: www.altera.com

[11] P. Belanovic and M. Leeser, "A library of Parameterized Floating Point Modules and Their Use" in Field Programming Logic and Applications, FPL02, Montpellier, LNCS Vol. 2438/2002.

[12] J. Detrey and F. De Dinechin, "A VHDL Library of Parametrisable Floating Point and LSN Operators for FPGA", http//www.ens-lyon.fr/~jdetrey/FPLibrary

[13] Altera, "NIOS Custom Instructions, Tutorial", June 2002, http://www.altera.com/literature/tt/tt_nios_ci.pdf

[14] B. Horn and B. Schunck, "Determining optical flow", Artificial Intelligence, 17:185--203, 1981.