FPGA Implementation of Decimal Processors for Hardware Acceleration

Borup, Nicolas; Dindorp, Jonas; Nannarelli, Alberto

Published in:
Proceedings of NORCHIP 2011

Link to article, DOI:
10.1109/NORCHP.2011.6126729

Publication date:
2011

FPGA Implementation of Decimal Processors for Hardware Acceleration

Nicolas Borup, Jonas Dindorp and Alberto Nannarelli
Dept. Informatics and Mathematical Modelling
Technical University of Denmark
Kongens Lyngby, Denmark

***Abstract*—Applications in non-conventional number systems can benefit from accelerators implemented on reconfigurable platforms, such as Field Programmable Gate-Arrays (FPGAs).**
In this paper, we show that applications requiring decimal operations, such as the ones necessary in accounting or financial transactions, can be accelerated by Application Specific Processors (ASPs) implemented on FPGAs. For the case of a telephone billing application, we demonstrate that by accelerating the program execution on a FPGA board connected to the computer by a standard bus, we obtain a significant speed-up over its execution on the CPU of the hosting computer.

I. Introduction

A hardware accelerator is a co-processor, or an Application Specific Processor (ASP), that is connected to the computer's Central Processing Unit (CPU) via a standard bus [1].
Normally, the accelerator is a unit optimized for a set of numerical computations that can run efficiently all the applications requiring these computations. Graphics Processing Units (GPUs) are an example of such accelerators. Originally introduced to off-load CPUs from graphics, GPUs have evolved into general-purpose many-core systems [2].
The raw power of state-of-the-art GPUs is quite impressive [3]. However, processing "non-conventional" data, such as very long integers and modular arithmetic used in cryptography, or financial computation requiring the decimal number system, can benefit from Application Specific Processors. These ASPs can be implemented on Field Programmable Gate-Arrays (FPGAs). FPGA accelerators can be fine tuned to match exactly the algorithm, and FPGAs are easy to reconfigure according to the application.
Decimal arithmetic is usually implemented by software routines, as binary floating-point does not always round correctly [4]. However, software operations run 100–1000 times slower than the corresponding binary operations implemented in hardware. For these reasons, in the revised IEEE standard 754 [5] support for decimal representation was added, and some companies are already commercializing processors which include decimal units [6], [7].
In this work, we show that we can accelerate with a decimal processor implemented on FPGA the accounting typically done by telephone companies. As a case study, we consider a telephone billing application: the TELCO benchmark [8].
The results show that the execution of the benchmark on the FPGA based accelerator is about 10 times faster than the execution on the CPU.

II. The TELCO Benchmark

The TELCO benchmark [8] was developed by IBM to investigate the balance between input and output (I/O) time and calculation time in a telephone company billing application.
The benchmark, available in several programming languages, provides an example of IEEE standard 754 [5] compliant set of Decimal Floating-Point (DFP) operations: multiplication and addition.
The benchmark program reads an input file containing a list of telephone call durations. The calls are of two types (listed as L and D) and to each type of call a rate (price) is applied. Once the call price has been computed, one or two taxes (depending on the type of call) are applied.
The benchmark specifies the rounding modes to apply in the different parts of the accounting. The price of the call must be rounded to the nearest cent (round-to-even in case of a tie), while the tax is computed by truncating to the cent.
For example, for a D-type call of 329 seconds (Brate=0.00894, Btax=0.0675, Dtax=0.03410), we have:

$$\begin{aligned} \text{Price} &= 329 \times 0.00894 = 2.94126 \overset{ROUND}{\rightarrow} 2.94 \\ \text{Btax} &= 2.94 \times 0.0675 = 0.1984 \overset{TRUNC.}{\rightarrow} 0.19 \\ \text{Dtax} &= 2.94 \times 0.03410 = 0.10025 \overset{TRUNC.}{\rightarrow} 0.10 \end{aligned}$$

and the total cost of the call is

$$\text{Cost of call} = 2.94 + 0.19 + 0.10 = 3.23.$$

The pseudo-code of the accounting algorithm is listed in Fig. 1. Finally, the benchmark program computes the total for calls cost and applied taxes.

III. The Hardware Accelerator

A. The Hardware Platform

The hardware accelerator is implemented on the Xilinx Virtex-5 LX330T FPGA. This FPGA is embedded on the Alpha Data ADM-XRC-5T2 board and is connected to the host PC via the PCI Express bus (Fig. 2). The CPU of the host PC is the Intel Core2 Duo processor clocked at 3 GHz.
To transfer data between the board and the PC, Direct Memory Access (DMA) is used. This allows access to the system memory for reading and writing independently of the CPU. To further improve the performance, burst mode is used which gives the DMA controller exclusive access to the bus without interruption. The implementation of the DMA functions, along

```
if (calltype = L)
   P = duration * Brate;
else
   P = duration * Drate;
Pr = RoundtoNearestEven(P);
B = Pr * Btax;
C = Pr + Trunc(B);
if (calltype = D) {
   D = Pr * Dtax;
   C = C + Trunc(D);
  }
```

Fig. 1. Pseudo-code of TELCO benchmark.

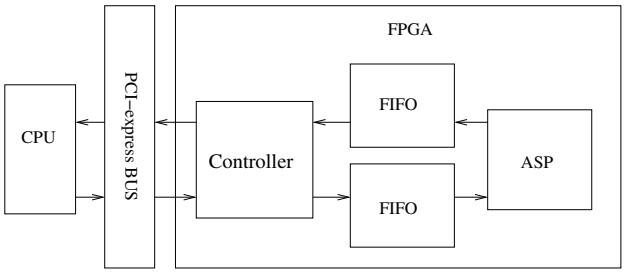

Fig. 2. Block diagram of the TELCO processor and the connection with the CPU.

with others, is included in a Software Development Kit (SDK) provided with the Alpha Data board. The SDK includes an application-programming interface (API), VHDL functions and examples.

B. Telco Processor

The TELCO processor (Fig. 2) includes two FIFO buffers which handles the input and output to the bus. A controller checks the bus to see when new data is transmitted and controls the signals to the FIFO buffers. Every time a new element (call duration) has been received, it is transmitted to the Application Specific Processor, or ASP, implementing the calculations of Fig. 1. Then, the result of the calculation is transmitted from the ASP to the output FIFO buffer. The controller is starting a DMA burst to the PC as soon as the output FIFO buffer holds enough bits to do so.

The data passed to the accelerator, and back to host the PC, are 32 bit vectors corresponding to 8 decimal digits when encoded in Binary Coded Decimal (BCD).

The structure of the ASP, detailed next, is shown in Fig. 3.

C. Telco Application Specific Processor

The ASP of Fig. 3 can be considered as divided into three main parts:

- Calculation of call price.
- Calculation (in parallel) of Btax and Dtax (if any).
- Calculation of the total cost.

1) Calculation of call price: Although we can transfer 8-digit BCD numbers, six decimal digits are sufficient (10^6 seconds $\simeq$ 11.5 days) to represent the duration of a call n. The call and tax rates are stored (hardwired) in the processor (FPGA's look-up tables). For the call rate r, selected by a multiplexer depending on the type of the call, a 5-digit fractional number is sufficient.

The product $p = n \times r$ is computed by a 6×5 BCD multiplier similar to the one described later in Section III-E. The product is a 11 BCD digit number with 5 fractional digit and it must be rounded to p_r, as explained in Section II, to the nearest cent.

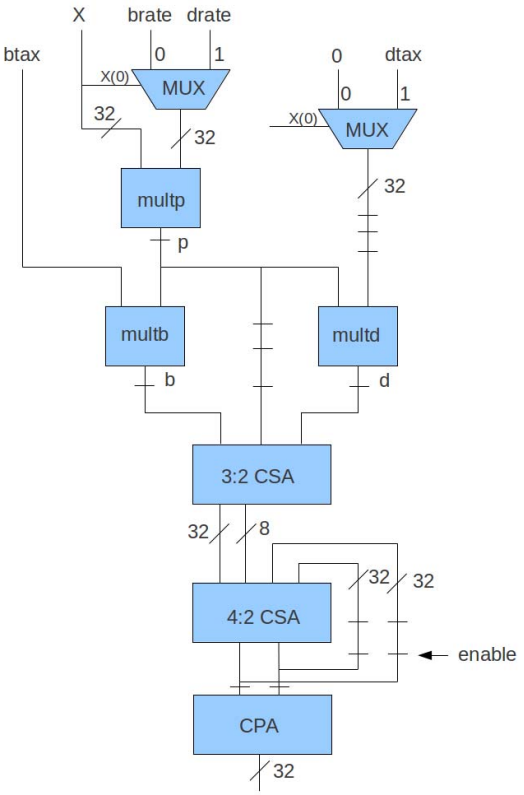

Fig. 3. Implementation of the ASP for the TELCO benchmark.

2) Calculation of Btax and Dtax: The cost of the two taxes (b and d) can be computed in parallel. Then, if the call is an L-type, the cost d can be ignored. The tax rates are represented by a 5-digit fractional number, as well. The application specifications state that the result of the multiplications, in this case, must be truncated to the cent (2nd fractional digit).

3) Calculation of the total cost: Once the call price and the taxes have been computed, the costs p_t, b (and d if any) are added to obtain the cost of the single call c_i. Then the cost of the call is added to the total cost.

To avoid the carry-propagation, some intermediate results are kept in carry-save representation. Detail on BCD carry-save

adders can be found in [9].
The output is a 8-digit BCD number (6 integer and 2 fractional digits). Six decimal digits for the integer part of the total cost (one million) should be adequate for the application. However, if the number of calls is huge (several millions) the number of digits can be easily extended in the accumulation stage.

D. BCD Addition

The addition of two BCD numbers

$$s = x + y$$

(previously aligned to the decimal point) is performed in three steps. By indicating with $x_i \in [0,9]$ and $y_i \in [0,9]$ the BCD digits of weight 10^i of the two significands x and y, with $c_i \in \{0,1\}$ the carry-in to digit of weight 10^i, and with $c_{i+1} \in \{0,1\}$ its carry-out, the three steps are:

1) A first stage computes from x_i and y_i the propagate p_i and generate g_i signals:

$$p_i = \begin{cases} 1 & \text{if } x_i + y_i = 9 \\ 0 & \text{otherwise} \end{cases} \qquad g_i = \begin{cases} 1 & \text{if } x_i + y_i \geq 10 \\ 0 & \text{otherwise} \end{cases}$$

2) A network to compute the carries from p_is and g_is:

$$c_{i+1} \;=\; g_i \text{ OR } (p_i \text{ AND } c_i)$$

This can be implemented with any binary prefix network.

3) A final stage to compute the final sum (modulus 10).

$$s_i = \langle x_i + y_i + c_i \rangle_{10}$$

Detail of the modulus 10 adder can be found in [9].

E. BCD Multiplier

The parallel multiplication shift-and-add algorithm is based on the identity

$$P \;=\; x \cdot y = \sum_{i=0}^{n-1} x y_i 10^i$$

where for decimal operands $y_i \in [0,9]$ and x is a n-digit BCD vector. To avoid complicated multiples of x, the multiplier y is normally recoded in a way that only the multiples x, $2x$ and $5x$ (and the respective negative multiples) are necessary [9]. That is, the multiplier digit is recoded $y_i = y_{Hi} + y_{Li}$ with $y_H \in \{0,5,10\}$ and $y_L \in \{-2,1,0,1,2\}$ as indicated in Table I, and the partial product is formed by adding the two multiples. For example, if $y_i = 8$, the corresponding partial product is obtained as

$$8x \cdot 10^i = (10x - 2x)10^i$$

y_i	0	1	2	3	4	5	6	7	8	9
y_H	0	0	0	5	5	5	5	5	10	10
y_L	0	1	2	-2	-1	0	1	2	-2	-1

TABLE I
RECODING OF y_i

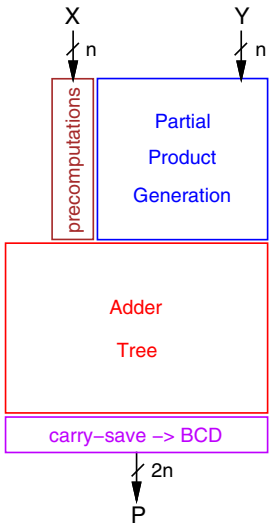

Fig. 4. BCD multiplier from [9]

A BCD multiplier scheme is shown in Fig. 4. It consists of four blocks:

1) **Precomputation** where the multiples of x are computed.
2) **Partial product generation** where each BCD digit y_i selects the corresponding multiples (partial product) according to Table I.
3) **Adder Tree** where all the partial products are accumulated by using an adder tree. There are several alternatives for the accumulations of partial products as reported in [9], [10], [11]. We opted for the scheme of [9].
4) **carry-save → BCD** is a carry-propagate adder similar to the one of Section III-D.

F. The Hardware Implementation

The accelerator of Fig. 2 is implemented on the FPGA of the Alpha Data board. The ASP of Fig. 3 is implemented with three 8x8 BCD digit multipliers and a final 8 BCD digit carry-propagate adder (CPA). The rates and taxes are chosen by setting multiplexers. The ASP is pipelined into 8 stages (3 stages are necessary for the multipliers), and it can sustain a maximum frequency of 127 MHz. However, the accelerator is clocked at 80 MHz which is the frequency of the DMA. Data are read from the input FIFO buffer into the accelerator and the total (partial) cost is queued in the output buffer and sent back to the CPU for logging, and to verify the functionality of the processor.
As the input FIFO buffer can be read every second clock cycle, the effective maximum frequency of operation is 40 MHz (25 ns per element) resulting in a processor latency of $8 \times 25 = 200$ ns.
Because the FPGA is quite large, only a small fraction of the logic is utilized. However, as the bottleneck is the data transfer from the host computer, the available space of the device cannot be utilized to further parallelize the algorithm.

IV. Experimental Results

The experiment consists in running the TELCO benchmark on the CPU of the host PC, and in processing the list of call durations on the TELCO processor.

The execution time when running the C program on the CPU is 1.5 seconds for a set of one million calls (elements). This value is obtained by averaging the readings of several runs as the experiment conditions (CPU tasks for operating system, etc.) vary at each run. The experimental results for the CPU execution of the benchmark are summarized in Table II. The execution time per element (e.g. $\frac{1.5\ s}{10^6} = 1.5\ \mu s$) is indicated between () in the table.

For large data sets (10^5–10^6 elements) the I/O time is between 0.3–$0.4\ \mu s$ per element. Consequently, we can assume that the decimal calculation time for large sets is about $1.2\ \mu s$ (or $1200\ ns$) per element. Because the I/O time is smaller than the computation time, the software execution of the benchmark on the CPU is *computation-bound*.

	I/O time		comp. time		exec. time	
# calls	$[ms]$	$[\mu s]$	$[ms]$	$[\mu s]$	$[ms]$	$[\mu s]$
$1 \cdot 10^5$	40	(0.4)	210	(2.1)	250	(2.5)
$5 \cdot 10^5$	180	(0.4)	660	(1.3)	840	(1.7)
$1 \cdot 10^6$	300	(0.3)	1200	(1.2)	1500	(1.5)

TABLE II
BENCHMARK EXECUTION TIME ON HOST PC.

Because the architecture of Fig. 2 was designed for minimal support to the decimal ASP, our accelerator can only process smaller sets of data to not incur in buffer overflow. For this reason, we measured the execution time to send the data to the ASP, and receive the computed results back in the CPU, for data sets of 10, 100, 200 and 300 elements. The execution times for the different sets are shown in Table III.

# calls (size)	execution time $[\mu s]$	time per element $[\mu s]$
10	196	19.6
100	162	1.62
200	185	0.93
300	198	0.66

TABLE III
EXECUTION TIME FOR PROCESSING THE TELCO BENCHMARK ON ASP.

Because the CPU timer only records start (data sent to ASP) and stop (result sent to CPU) times, we have to estimate the parts of the execution time which are spent in I/O and ASP computation. By considering two different entries in Table III, we can estimate the ASP computation time per element as

$$\frac{t_{size(b)} - t_{size(a)}}{size\,(b) - size\,(a)}$$

that for the set sizes 200 and 300 (maximum throughput) gives

$$\frac{198\ \mu s - 185\ \mu s}{300 - 200} = 0.13\ \mu s \quad \text{per element.}$$

From this estimate we can see that the decimal computation takes 130–180 ns per element. Because the pipelined decimal unit can sustain a maximum throughput of 40M elements per second (25 ns per element), the accelerator is slowed down by the I/O communication. In this case, the execution of the benchmark on the FPGA is *I/O-bound*.

Finally, by comparing the experiments on CPU and ASP, the speed-up for the computation time is

$$\text{speed-up}_{comp} = \frac{t_{comp-CPU}}{t_{comp-FPGA}} = \frac{1.20}{0.13} = 9.23$$

while the actual (computation and I/O time) speed-up is

$$\text{speed-up}_{actual} = \frac{t_{exec-CPU}}{t_{exec-FPGA}} = \frac{1.20}{0.66} = 1.81$$

V. Conclusions

In this work, we present the implementation of a hardware accelerator for decimal arithmetic implemented on an FPGA. FPGA implementations of accelerators are attractive for applications requiring non-binary number systems because, differently from CPUs and GPUs, the processor can feature special operators (e.g. BCD adders and multipliers).

As a case study, we chose an accounting application requiring the use of decimal arithmetic.

The results of the execution of the benchmark on the FPGA accelerator are compared with those of the benchmark execution on the CPU of the host PC. By considering the computation time (decimal part) the accelerator speed-up is about 10 times. The FPGA computation time per element is between 5–7 times the one achievable at the maximum throughput. Therefore, by redesigning the accelerator I/O interface to handle larger data sets, we should be able to increase the ASP throughput and further improve the speed-up over the CPU execution.

References

[1] S. Patel and W. mei W. Hwu, "Accelerator Architectures," *IEEE Micro*, vol. 28, pp. 4–12, July/Aug. 2008.

[2] D. Luebke and G. Humphreys, "How GPUs Work," *IEEE Computer magazine*, pp. 96–100, Feb. 2007.

[3] NVIDIA. "Fermi. NVIDIA's Next Generation CUDA Compute Architecture". Whitepaper. [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[4] M. F. Cowlishaw, "Decimal floating-point: algorism for computers," in *Proc. of 16th Symposium on Computer Arithmetic*, June 2003, pp. 104–111.

[5] *IEEE Standard for Floating-Point Arithmetic*, IEEE Computer Society Std. 754, 2008.

[6] L. Eisen *et al.*, "IBM POWER6 accelerators: VMX and DFU," *IBM Journal of Research and Development*, vol. 51, no. 6, pp. 663–684, 2007.

[7] S. Carlough, A. Collura, S. Mueller, and M. Kroener, "The IBM zEnterprise-196 Decimal Floating-Point Accelerator," in *Proc. of 20th IEEE Symposium on Computer Arithmetic*, July 2011, pp. 139–146.

[8] IBM Corporation. "The "telco" benchmark". [Online]. Available: http://speleotrove.com/decimal/telco.html

[9] T. Lang and A. Nannarelli, "A Radix-10 Combinational Multiplier," *Proc. of 40th Asilomar Conference on Signals, Systems, and Computers*, pp. 313–317, Nov. 2006.

[10] L. Dadda, "Multi Operand Parallel Decimal Adders: a mixed Binary and BCD Approach," *IEEE Transactions on Computers*, vol. 56, pp. 1320–1328, Oct. 2007.

[11] A. Vazquez, E. Antelo, and P. Montuschi, "A new family of high-performance parallel decimal multipliers," *Proc. of 18th Symposium on Computer Arithmetic*, pp. 195–204, June 2007.