

SYSTEM-LEVEL POWER EVALUATION OF AN EMBEDDED SOFTWARE DATA BLOCK PROCESSING ALGORITHM

Kristian Hildingsson^{1,3} and Tughrul Arslan^{1,2}

¹The Institute for System Level Integration, Livingston, Scotland

²Department of Electronics & Electrical Engineering, The University of Edinburgh, Scotland

³Hitachi Micro Systems Europe Ltd., Maidenhead, England

ABSTRACT

Data block processing algorithms have demonstrated significant efficiency in terms of low power consumption when applied to mainly hardware implementation of digital signal processing algorithms. In this paper, a generic data block processing algorithm is applied to the implementation of an FIR filter on a System-on-Chip platform incorporating a micro controller and a programmable 32-bit DSP processor. The block processing algorithm is evaluated at the system-level including the performance metrics speed, energy, power and area. The data block processing technique achieves a reduction in energy consumption of 18% and memory accesses are reduced by 44%, for an 8-tap FIR filter. Our algorithm is targeted as a Macro block, which can be re-used in the design of more complex DSP systems on the SoC platform.

I. INTRODUCTION

There is an emerging interest for power conscious design techniques in Systems-on-Chip (SoC) at all levels of design abstraction. One way to efficiently incorporate low power design techniques in SoC design is to implement generic Macro blocks in both hardware and software. These Macro blocks are re-used across various designs, which has the benefit of reducing the design effort as well as shortening time-to-market. This paper presents a system-level evaluation of a low power generic data block processing technique, which is applied to the implementation of a digital finite impulse response (FIR) filter example. The FIR filter is a widely used algorithm in industrial applications such as mobile telephony and audio and image processing. The nature of the application in which the FIR filter is used will determine whether it will be a dedicated hardware solution or an implementation in software. A hardware implementation is often chosen when real-time response is required, or when a DSP processor is not available as part of the system. A software implementation is useful in a programmable system where the platform includes a DSP processor. In the latter case there is normally a DSP software library available for commonly used DSP algorithms. These library components

(or Macros) are often hand-written assembly code implemented for optimised execution.

Block processing is a generic algorithm that has been effectively applied to signal processing tasks such as FIR filters in order to enhance their performance in terms of power and speed. Work in the literature so far has focused on the implementation of block processing in both customised hardware and dedicated processors. In [1], data block processing is applied to an FIR filter implemented on a simulated single-multiplier DSP core. Comparisons are made between a conventional implementation and the block processing implementation for various block sizes. Significant reduction of switching activity in the multiplier unit is achieved when using block processing. Furthermore, the number of memory accesses (reads) is drastically reduced. A number of generic FIR cores are designed in [2] which exploits the data block processing technique as well as a low power coefficient segmentation technique. The two techniques are applied separately as well as in combination to achieve low power implementations of generic FIR filtering cores. The block processing technique proves to have the most prominent impact on the power savings.

In this paper, the data block processing technique is investigated at the system-level for the implementation of an FIR filter in embedded software on a commercial SoC platform. No work has previously been carried out in order to study system-level performance implications of mapping the algorithm on to the various components of the SoC platform, such as the micro controller, DSP, and peripherals. This study also includes overheads due to control tasks and communication among the various on-chip devices and peripherals.

The target system is a 32-bit RISC processor with a DSP extension with on-chip memory plus external memory. This system exemplifies a typical heterogeneous system, e.g a system comprising CPU, memory, and one or more Co-processors for digital signal processing. The RISC CPU handles control, I/O, and other device-level functions. The DSP unit processes bandwidth intensive functions and runs in parallel with the CPU. Algorithms such as the FIR filter shall therefore be considered as part of a larger application whereby parameterised calls are made to the FIR Macro, which executes efficiently on the DSP unit. The energy consumption of such Macro components is critical

since they are part of the overall system and therefore impact system-level energy efficiency.

The system-level power evaluation is facilitated by our Power Evaluation Tool (PET) for software evaluations, which generates execution statistics including information for performance metrics such as speed, energy and power consumption. The power modelling in PET is based on instruction level power models, which were developed in a similar way to what was done in [5][6].

The rest of the paper is organised as follows. Section II describes the target architecture, e.g. the 32-bit RISC+DSP processor. Section III describes the conventional implementation of the FIR filter on the DSP unit. Section IV describes the block processing FIR filter implementation. Section V describes the power evaluation environment. Section VI presents the results from evaluating the conventional and the block processing FIR filter implementations. Finally, Section VII concludes the main discussion and the results.

II. TARGET ARCHITECTURE

The processor used in this work has a 32-bit internal data bus, sixteen 32-bit general-purpose registers, eight 32-bit control registers and four 32-bit system registers. It has a RISC type instruction set with 16-bit fixed instruction word length. Basic instructions have an execution time of one cycle. The processor is of load/store architecture with a five-stage pipeline. It has on-chip cache memory of 16KB (mixed instruction and data), and there is XRAM/YRAM embedded memory with three independent read and write ports (see Fig. 1). The DSP unit is an extended Harvard architecture with a mixture of 16-bit and 32-bit instruction words. There are two data buses and one separate instruction bus. The extended DSP functionality includes a single-cycle 16*16 bit multiplier, barrel shifting, priority encoding, rounding, modulo addressing, and zero-overhead loop control. Four operations can execute in parallel, such as addition, multiplication and two load or store to XRAM/YRAM.

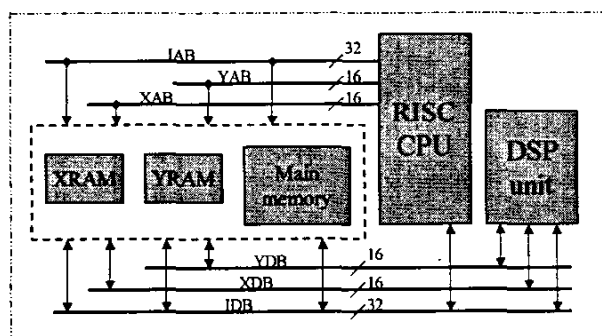


Fig. 1 Overview of the 32-bit RISC+DSP processor architecture

This system has been available to us in a 0.35μm technology single-chip solution on an evaluation board operating at 3.3V at 15MHz, with 1Mbyte of on-board (program) memory.

III. CONVENTIONAL FIR FILTER IMPLEMENTATION

A mathematical representation of an FIR filter algorithm is given in Eq. 1, where convolution of input data x and coefficient data h generates the output samples y . For a 4th order filter (i.e. $N=5$), Eq. 1 expands to the calculations shown in Fig. 2.

$$y[n] = \sum_{k=0}^{N-1} x[n-k] \times h[k] \quad (\text{Eq. 1})$$

In a conventional implementation the partial products are calculated in order as they appear in Fig. 2 for one output sample at a time. The calculation of one output sample requires N multiply-and-accumulate (MAC) operations and $N*2$ data values of x and h . The DSP unit can perform one addition, one multiplication and two memory reads, in parallel. The assembly code that implements these four parallel operations is given in Fig. 3, where instructions `MOVX.W @Ax0+,X0` and `MOVY.W @Ay0+,Y0` move data from XRAM and YRAM into registers X0 and Y0, and increment registers Ax0 and Ay0. The `PMULS` instruction multiplies the values in X0 and Y0 and stores the result in register M0. The `PADD` instruction adds the value of M0 to the value of A0 (accumulating). This parallel operation has a latency of three clock cycles, from that two new data values has been stored in X0 and Y0 till that they have been multiplied and the result added to the value of A0. The assembly code for generating one output sample for a 5-tap filter is given in Fig. 4. Since the operation is pipelined, some instructions can be excluded (this is indicated with a hyphen (-) in the code). The instructions in bold constitutes the inner loop, and will for larger filters be implemented using the zero-overhead loop control circuitry to limit code expansion. In addition to the code shown in Fig. 4 some extra loop overhead code is required to store each computed output sample (which is accumulated in register A0), (re)set memory pointers, and clear the accumulating register. The loop overhead code for a conventional implementation requires approximately 8 cycles, and is independent on the filter length.

In the conventional implementation the data streams on the inputs A and B to the multiplier will be as shown in Fig. 5 for the first two output samples, e.g. $x(0) \times h(0)$ followed by $x(-1) \times h(1)$, $x(-2) \times h(2)$, $x(-3) \times h(3)$ and so on until clock cycle $n+9$. The computation of the first two output samples hence requires 20 read accesses to XRAM and YRAM to obtain the data values of x and h . Fig. 5 shows that calculating the first partial products for $y(0)$ and $y(1)$ requires the three values of $x(0)$, $h(0)$ and $x(1)$.

In the conventional implementation the value of $h(0)$ is read twice. The challenge in data block processing is to reuse already obtained data values in order to reduce the number of memory accesses. This technique is described in the next section.

IV. BLOCK PROCESSING FIR FILTER IMPLEMENTATION

The block processing technique is reducing the number of memory accesses by computing two or more output samples interchangeably. It also reduces the amount of switching activity on the inputs to the multiplier unit. In theory, the block processing technique can be implemented with an arbitrary block size, where a block size of two means that two output samples are computed interchangeably. However, in practice there are limitations in the instruction set as to how many output samples can be computed interchangeably. Our investigation reveals that for this particular processor the largest possible block size is two. Fig. 2 showed the computation of partial products that are required in order to calculate the first four output samples. Fig. 6 suggests a grouping of these partial product calculations in blocks of two. It is observed that within each block the same coefficient h is being used for both partial products, and therefore only three different values needs to be fetched from memory for each block. The order in which the partial products are calculated within each block is determined by the indexes of x and h . As a start, consider the following order of calculation of partial products: $x(0) \times h(0)$ followed by $x(1) \times h(0)$, $x(-1) \times h(1)$,

$$\begin{aligned} y(0) &= x(0) \times h(0) + x(-1) \times h(1) + x(-2) \times h(2) + x(-3) \times h(3) + x(-4) \times h(4) \\ y(1) &= x(1) \times h(0) + x(0) \times h(1) + x(-1) \times h(2) + x(-2) \times h(3) + x(-3) \times h(4) \\ y(2) &= x(2) \times h(0) + x(1) \times h(1) + x(0) \times h(2) + x(-1) \times h(3) + x(-2) \times h(4) \\ y(3) &= x(3) \times h(0) + x(2) \times h(1) + x(1) \times h(2) + x(0) \times h(3) + x(-1) \times h(4) \end{aligned}$$

Fig. 2 Partial product calculations required to compute the first four output samples in a 5-tap FIR filter

```
PADD A0,M0,A0 PMULS X0,Y0,M0 MOVX.W @Ax0+,X0 MOVY.W @Ay0+,Y0
```

Fig. 3 Assembly code for the MAC-operation and two memory loads

```
- - MOVX.W @Ax0+,X0 MOVY.W @Ay0+,Y0
- PMULS X0,Y0,M0 MOVX.W @Ax0+,X0 MOVY.W @Ay0+,Y0
PADD A0,M0,A0 PMULS X0,Y0,M0 MOVX.W @Ax0+,X0 MOVY.W @Ay0+,Y0
PADD A0,M0,A0 PMULS X0,Y0,M0 MOVX.W @Ax0+,X0 MOVY.W @Ay0+,Y0
PADD A0,M0,A0 PMULS X0,Y0,M0 MOVX.W @Ax0+,X0 MOVY.W @Ay0+,Y0
PADD A0,M0,A0 - -
```

Fig. 4 Assembly code for conventional implementation, computing one output sample in 5-tap FIR filter

cycle(n+4)	cycle(n)
x(-4), x(-3), x(-2), x(-1), x(0) → Input A	
h(4), h(3), h(2), h(1), h(0) → Input B	
cycle(n+9)	cycle(n+5)
x(-3), x(-2), x(-1), x(0), x(1) → Input A	
h(4), h(3), h(2), h(1), h(0) → Input B	

Fig. 5 Data streams to multiplier for computation of the first two output samples in conventional 5-tap FIR filter

$x(0) \times h(1)$ and so on, moving in a zigzag fashion towards the last filter tap. The value of $h(k)$ is now kept for two consecutive multiply operations at a time, and the index k in $h(k)$ is hence incremented every second cycle. The index of x has a sequence of 0, 1, -1, 0 etc. which will cause problems in the implementation of address pointers. This problem is solved by re-ordering the calculation of partial products and instead start with $x(1) \times h(0)$ followed by $x(0) \times h(0)$, $x(0) \times h(1)$, $x(-1) \times h(1)$ etc. The index of x now sequences over 1, 0, 0, -1 etc., and is hence decremented every two cycles. The index sequence for h remains the same. The data streams to the multiplier inputs for the block processing implementation are shown in Fig. 7, which can be compared to the data streams of the conventional implementation in Fig. 5. It is observed that in every second cycle within both data streams (x and h) the value remains the same. Hence data values are being re-used in both data streams, which results in a reduction in the number of memory accesses by approximately 50%. In addition, the switching activity is reduced on the inputs to the multiplier since the values change only every second clock cycle. The corresponding assembly code for the block processing implementation is shown in Fig. 8, where NOPX and NOPY are idle data move operations. In the same way as for the conventional implementation the (-) indicates idle slots in the code, and the code in bold constitutes the inner loop of the filter. Using the block processing scheme for a block size of two means that two output samples are computed for each iteration of the inner loop, with accumulating registers A0 and A1.

$$\begin{aligned} y(0) &= x(0) \times h(0) + x(-1) \times h(1) + x(-2) \times h(2) + x(-3) \times h(3) + x(-4) \times h(4) \\ y(1) &= x(1) \times h(0) + x(0) \times h(1) + x(-1) \times h(2) + x(-2) \times h(3) + x(-3) \times h(4) \\ y(2) &= x(2) \times h(0) + x(1) \times h(1) + x(0) \times h(2) + x(-1) \times h(3) + x(-2) \times h(4) \\ y(3) &= x(3) \times h(0) + x(2) \times h(1) + x(1) \times h(2) + x(0) \times h(3) + x(-1) \times h(4) \end{aligned}$$

Fig. 6 Grouping of partial product calculations using data block processing algorithm with block size of two

```
cycle(n+9) cycle(n)
x(-4), x(-3), x(-3), x(-2), x(-2), x(-1), x(-1), x(0), x(0), x(1) → A
h(4), h(4), h(3), h(3), h(2), h(2), h(1), h(1), h(0), h(0) → B
```

Fig. 7 Data streams to multiplier for computation of first two output samples in block processing 5-tap FIR filter

```
- - MOVX.W @Ax0+,X0 MOVY.W @Ay0+,Y0
- PMULS X0,Y0,M0 MOVX.W @Ax0+,X0 NOPY
PADD A1,M0,A1 PMULS X0,Y0,M0 NOPX MOVY.W @Ay0+,Y0
PADD A0,M0,A0 PMULS X0,Y0,M0 MOVX.W @Ax0+,X0 NOPY
PADD A1,M0,A1 PMULS X0,Y0,M0 NOPX MOVY.W @Ay0+,Y0
PADD A0,M0,A0 PMULS X0,Y0,M0 MOVX.W @Ax0+,X0 NOPY
PADD A1,M0,A1 PMULS X0,Y0,M0 NOPX MOVY.W @Ay0+,Y0
PADD A0,M0,A0 PMULS X0,Y0,M0 MOVX.W @Ax0+,X0 NOPY
PADD A1,M0,A1 PMULS X0,Y0,M0 - -
PADD A0,M0,A0 - -
```

Fig. 8 Assembly code for block processing implementation, computing two output samples in 5-tap FIR filter

Therefore, the code size for the inner loop is larger compared to a conventional implementation. However, using the zero-overhead loop circuitry limits the code expansion. The loop overhead code for a block processing FIR filter implementation requires approximately 12 cycles, which is 4 cycles more compared to the conventional implementation. This increase in code size is due to the extra instructions required to store two output samples and update the additional memory pointers accordingly.

The main reason for why the maximum block size is two is because there are only two registers accessible by the PADD instruction. It was found in [2] that a block size of two yielded the best power savings.

V. POWER EVALUATION ENVIRONMENT

The evaluation of energy conscious design techniques (such as data block processing) is facilitated by methods to estimate and model power and energy consumption of a given system. Research in the area of estimation and modelling is dedicated to shortening simulation times and concurrently maintain a reasonable level of accuracy [3][4]. The challenge is to find techniques that link low-level behaviour to power models at higher levels of abstraction, in order to predict power at an earlier stage in the design flow. For the purpose of evaluating power conscious design techniques such as the data block processing technique, we have developed a Power Evaluation Tool (PET) for software power evaluations. The PET enables an arbitrary software execution to be characterised in terms of its hardware utilisation. This characterisation includes performance metrics such as speed, energy and power metrics. The PET is developed as an extension to the instruction set simulator of the target processor, and

Table 1: Speed and memory access characteristics generated by the Power Evaluation Tool (PET) for various filters

Filter type	Tot clock cycles	XRAM/YRAM	
		reads	writes
FIR8_conv	164	142	8
FIR8_block	134	76	8
FIR16_conv	463	542	16
FIR16_block	399	278	16
FIR28_conv	1147	1622	28
FIR28_block	1035	824	28

Table 2: Energy and power consumption estimates generated by PET for various filters (energy and power figures at system-level)

Filter type	Energy (uJ)		Power (mW)	
	min	max	min	max
FIR8_conv	2.37	2.41	217.0	220.5
FIR8_block	1.94	1.97	217.3	221.1
FIR16_conv	6.65	6.77	215.4	219.5
FIR16_block	5.73	5.84	215.4	219.5
FIR28_conv	16.4	16.7	214.4	218.9
FIR28_block	14.8	15.1	214.4	218.9

analyses the execution trace to generate a summary including the following information: decoded instructions by class, execution cycles per instruction class, energy consumption for decoded instructions, memory accesses (XRAM/YRAM and main memory), stall cycles, data dependent energy consumption, as well as total energy consumption and power consumption for the program subjected to evaluation.

To calculate power related metrics, the PET uses power models that are developed at the instruction level, using a similar methodology to that proposed by Tiwari et al. in [5][6]. In our work each power model has an upper and a lower bound for energy consumption, which reflects the dependency to data in instruction operands (for data transfer and arithmetic operations). The combination of instruction level power models and trace data from the instruction set simulator enables a fast characterisation of an arbitrary software execution, generating a performance profile at the system level.

We have used the PET in this work to evaluate different FIR filter implementations. The accuracy of the power/energy estimations in PET has been verified to within 5-10% compared with chip measurements.

VI. EXPERIMENTS AND RESULTS

Two generic FIR filters were implemented and tested – one for a conventional implementation and one utilising the block processing technique. Each of the two filters were implemented with 8, 16 and 28 taps and executed on the instruction set simulator for a single impulse response. The instruction set simulator generates an execution trace that is analysed by the PET, which in turn generates the performance profile of each execution.

Table 3: Chip measurements of power consumption of 8-tap filter

Filter type	Power in uP (mW)		Power in uP+mem (mW)	
	low switching	high switching	low switching	high switching
FIR8_conv	137.6	139.5	220.8	222.4
FIR8_block	137.0	139.2	220.5	222.1

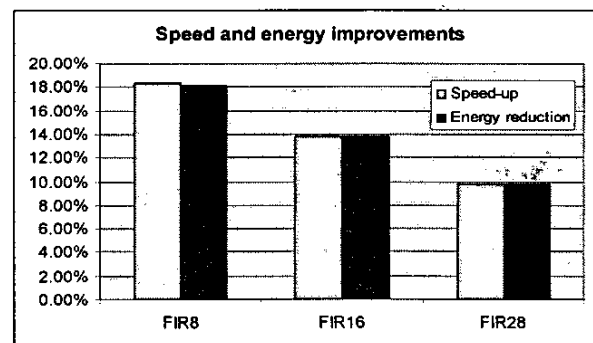


Fig. 9 Speed and energy improvements for different filter lengths when applying the data block processing algorithm

The 8-tap filter was also executed on the processor evaluation board in an infinite loop in order to measure the power consumption drawn by the system (RISC + DSP + on-chip peripherals + external memory). A summary of the PET evaluations of the various filters is given in *Tables 1-2*. Applying the data block processing technique achieves a speed-up in the range 10-18%. The memory read accesses are reduced by between 46-49%. The speed improvement is an effect of the loop unrolling that results from calculating two output samples interchangeably within one loop iteration. In other words, when applying the block processing technique the loop overhead code is shared over the computation of two output samples (rather than only one for a conventional implementation), which results in less cycles being expended compared to the same computation in a conventional implementation. The speed improvement becomes less prominent for a growing number of filter taps, which is due to the fixed number of clock cycles required for the loop overhead code. The reduction in energy consumption follows closely the improvement in speed. For example a 10% speed improvement leads to approximately 10% reduction in energy consumption for the 28-tap filter (see *Fig. 9*). This relationship between speed and energy means that a similar amount of power is dissipated during each clock cycle, regardless of which filter type is executed. It is observed that the difference in power consumption for the various filters is small (*Table 2*). *Table 3* shows the actual power consumption of the 8-tap filter being executed in an infinite loop on the processor evaluation board. Measurements were taken for the RISC + DSP (which also includes all on-chip peripherals) and the RISC + DSP + Ext. memory (i.e. the complete system). The switching activity was controlled through the input data, where low switching was created by applying zero input data, and high switching was created by applying input data with large Hamming distance between consecutive data values. Minor fluctuations were observed for the power consumption.

The code size (i.e. the area) is slightly larger for the block processing filter (as was mentioned in Section IV). With the zero-loop overhead circuitry the area can be kept constant for any length of filter, e.g. an implementation as a generic (parameterisable) Macro block. Although the block processing technique reduces the switching activity (hence power dissipation) in the multiplier unit, our experiments shows minor reductions in power consumption at the system-level. However, the reduction in number of clock cycles (i.e. speed-up) when applying the block processing technique potentially enables the filter to be executed at a lower clock frequency with maintained throughput characteristics. Executing the filter at a lower clock frequency will result in reduced power consumption. However, the same amount of switching is still required to complete the filter operation and therefore the energy consumption will remain the same.

VII. CONCLUSION

Our work proves that the generic data block processing technique can be applied to the software implementation of an FIR filter to reduce energy consumption at the system-level. Various length filters were implemented in a conventional way and by using the block processing technique. The various filter implementations were evaluated for speed, energy and power at the system-level. Results shows that the energy reduction is dependent on the filter size, and for an 8-tap FIR filter the energy consumption is reduced by approximately 18% when applying block processing. Memory accesses for the same filter are reduced by 44%. The reduction in energy consumption follows closely the improvement in speed. The difference in power consumption for the various filter implementations is small. However, the power consumption can be reduced by reducing the clock frequency and still maintain data throughput. The area (code size) is slightly increased when using block processing, but can be kept constant through the use of the zero-overhead loop circuitry of the DSP. Our work demonstrates that the block processing is a viable low power technique for the implementation of common digital filters on a programmable DSP processor. By incorporating the block processing technique in a Macro block, energy efficient implementations are easily re-used across various designs.

ACKNOWLEDGEMENTS

The authors would like to thank Hitachi Micro Systems Europe Ltd. (HMSE) and the EPSRC for their support under the Engineering Doctorate (EngD) programme (award number 99316566).

REFERENCES

- [1] Erdogan, A. T.; Arslan T., "Data Block Processing for Low Power Implementation of Direct Form FIR Filters on Single Multiplier CMOS Based DSPs", in Proc. IEEE Int. Symposium on Circuits and Systems, pp. D441-D444, June 1998.
- [2] Erdogan, A. T.; Hasan, M.; Arslan, T., "Algorithmic Low Power FIR Cores", IEE Proceedings circuits, devices, and systems, in print.
- [3] Givargis, T.D.; Vahid, F. Henkel, J., *Trace-driven system-level power evaluation of system-on-a-chip peripheral cores*, Proceedings of the Design Automation Conference - Asia and South Pacific (ASP-DAC), pp. 306-311, 2001
- [4] Mehta, H.; Owens, R.M.; Irwin, M.J., *A simulation methodology for software energy evaluation*, Proceedings of the Tenth International Conference on VLSI Design, pp. 509-510, 1997
- [5] Tiwari, V.; Malik, S.; Wolfe, A., *Power analysis of embedded software: a first step towards software power minimization*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Volume: 2 Issue: 4, pp. 437-445, Dec. 1994
- [6] Tiwari, V.; Tien-Chien Lee, M., *Power analysis of a 32-bit embedded microcontroller*, Proceedings of the Design Automation Conference - Asian and South Pacific (ASP-DAC), pp. 141-148, 1995