A Comparison of Hardware Acceleration Interfaces in a Customizable Soft Core Processor

Gerald Hempel, Christian Hochberger
Chair for Embedded Systems
TU Dresden, Germany
Email: {gerald.hempel,christian.hochberger}@inf.tu-dresden.de

Andreas Koch
Embedded Systems and Applications Group
Technische Universität Darmstadt, Germany
Email: koch@esa.informatik.tu-darmstadt.de

***Abstract*—Due to the continuously decreasing cost of FPGAs, they have become a valid implementation platform for SOCs. Typically, a soft core processor implementation is used to execute the software parts of the SOC. As each system is individually designed for a particular application, the idea is natural to support compute intensive parts of the code through customized hardware acceleration. Two different architectural variants have been proposed for this purpose in SOCs: either as an instruction set extension with specialized pipeline implementation or as a peripheral component that is programmed through memory mapping. In this contribution we analyze the efficiency (speedup related to LUTs) of those two variants.**

I. Introduction

According to Moore's law the number of usable transistors doubles every 24 months. This is also true for FPGAs. Thus, we have seen a development of very large, yet low cost FPGA devices. Various vendors have developed such FPGAs, e.g. Xilinx with their Spartan family, Altera with their Cyclone II devices or Lattice with their ECP2 family. They all have reached the million system gate level, which offers an enormous capacity for logic. In the meantime these devices do not only contain logic cells, but also embedded memory blocks and even DSP blocks for arithmetic operations. The mix and amount of these elements are sufficient to implement small to medium size systems on a single FPGA, thus leading to a true system on a chip structure (SoC).

A primary component for such systems is the processor core. It is the heart of the system and determines its applicability to certain problems. Soft core processor implementations with a range of bit widths from 8 to 32 bit are used in FPGA based SoC designs. Small bit widths have the advantage of a small memory footprint for simple applications, but also imply a limited complexity of the application software. Wide instructions allow for much more complex applications, but will also require large amounts of memory even for small applications. In the classical microcontroller world 16 bit processors fill the gap between those extremes and are typically best suited for medium sized applications.

To close this gap, we have developed the SpartanMC SoC kit. It contains a soft core processor with 18 bit wide instructions and data path. Also, it delivers almost any type of peripheral interface that is widely used in these systems.

Building SoCs in FPGAs automatically enables us to add custom hardware components to speed up compute intensive parts of the application. Various options for architectural integration of these hardware accelerators have been proposed in the past. In SoCs mainly two variants need to be considered: either as instruction set extension or as peripheral component. It is not automatically clear which variant is better. We define an efficiency metric to evaluate the quality of the variants. Efficiency quantifies the gained speedup in relation to the amount of LUTs.

In this contribution we compare the efficiency of the two architectural variants of hardware accelerations. To this end we have implemented two applications from different application domains in both ways and purely in software.

II. Related Work

A number of design patterns recur in the wide spectrum of interfaces between hardware accelerators (HAs) and processors. They include whether the HAs are directly integrated into the processor pipeline (usually the EX stage), or accessed as memory-mapped peripheral devices located outside the pipeline. Also of interest is the way data is exchanged: Is it limited to registers, or can the HA access memory on its own (master-mode)?

While targeting ASICs, the Tensilica Xtensa line configurable 32b processors [1], allows the integration of HAs directly into the EX stage of the pipeline, with the regular register file being used for communication. Additionally, the HAs can be configured to have wide-word (up to 128b) master-mode memory access with arbitrarily calculated addresses through the processor's load-store units (LSU). This combines the availability of low-bandwidth CPU/HA communication with high-bandwidth memory accesses.

A similar example of tighter coupled HAs is the 32b Altera Nios II [2] reconfigurable soft-core processor, which also integrates HAs in the EX stage. However, since these HAs can have multi-cycle latencies (stalling the pipeline while in the meantime), they could perform master-mode memory accesses using a dedicated connection to the system bus (not using the processor LSU). With this freedom, the HA can realize arbitrarily complex algorithms (e.g., also containing loops).

The Stretch software-configurable processors (SCP) [3] also use a 32b Xtensa pipeline as core, but they run the HA as a separate (and possibly much longer) pipeline in parallel to the processor pipeline. The processor can write data into the dedicated 128b wide HA register file, but not retrieve results that way. Instead, special processor instructions can transfer wide HA register words from and to memory, but with all address calculations being performed in software. Recent versions of the SCP also embed 32 KB of small memory blocks directly into the HA, which are accessible in the processor memory-map. As with the Tensilica technology, Stretch HAs are limited to algorithms with fully unrollable loops. Variable-bound (or too-large) loops will be realized using software instructions to restart new iterations of the HA appropriately.

Xilinx configurable 32b MicroBlaze processors [4] allow the attachment of HAs outside of the pipeline, but accessible using dedicated (non-memory mapped) instructions transfering data between HA and the register file. The HA itself is only loosely coupled with the processor, and may implement algorithms of arbitrary complexity and using master-mode memory accesses.

The Triscend (now part of Xilinx) TE5xx [5] and A7S [6] reconfigurable SoCs (rSoC) have their HAs attached to the chip-internal busses as peripheral devices. The processor, an 8b 8051-derivative for the TE5x and an ARM7TDMI for the A7S, accesses the HA registers in a memory-mapped fashion. The HA is master-mode capable, but relies on regular address sequences computed by an external DMA controller. This loosely-coupled HA can execute loops independently.

As an example for a more complex HA, the hardware/software co-compiler COMRADE [7] generates peripheral-mode HAs directly attached to the rSoCs main memory controller. While memory-mapped communication with the 32b PowerPC processor is possible and used to exchange limited amounts of parameter data, this architecture is specialized for HAs executing independently of the processor using high-throughput master-mode to bulk process data in main memory, including generation and translation of virtual addresses [8].

III. The SpartanMC SoC-Kit

To use an FPGA as implementation platform for SoCs, we need to provide a number of tools and components to the system designer. Besides the processor core, a sufficiently large selection of standardized peripheral components needs to be available. Also, generation and programming of the system should be simplified as much as possible. Thus, tools like compiler and system builder are required.

A. Processor Core

8 and 32 bit processor soft cores are already available for FPGAs. Applications of medium complexity neither can be implemented with 8 bit cores nor do they require the computing power of the 32 bit cores. Thus, we have decided to provide an SoC-kit with a processor core corresponding to traditional 16 bit microcontrollers. In fact the SpartanMC core uses 18 bit wide instructions and data path as this has been established as the natural bit width in FPGAs for memory blocks and multipliers.

Fig. 1 shows the microarchitectural structure of the RISC processor core. The pipeline uses three stages. In the first stage instruction fetch, operand fetch and instruction decoding are carried out. In the second stage the ALU execution and memory access follow. The third stage writes back the results into the register memory. The core uses four different instruction formats all of 18 bits length. Fig. 2 shows the different instruction formats. All instructions provide at most two register addresses (immediate instructions only use one address and jump instructions use no address). Register addresses are 4 bit wide. The first operand is always used as destination operand. Register type instructions include an additional function code (bits 4 to 0) which specifies the actual ALU operation.

Using the dual ported internal memory blocks of modern FPGAs, the processor core can simultaneously access code and data without resource conflict (modified Harvard architecture).

In order to simplify the software design and to provide fast procedure and interrupt invocation, the SpartanMC core uses a sliding register window. Registers 0 to 3 are globals, registers 4 to 7 are the input window of the current procedure, registers 8 to 11 are the locals while registers 12 to 15 are the output window. Through procedure invocation, the current output window becomes the input window of the called procedure. Thus, each called procedure can access 8 new registers.

A more detailed description of the processor core together with a discussion of the architectural decisions can be found in [9].

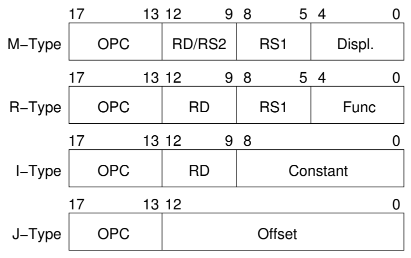

Fig. 2. The instruction formats of the SpartanMC

B. Peripherals

Peripherals are typically implemented in a memory mapped way. They can be implemented in two different ways: Simple peripherals provide registers that can be read or written by the processor core. Peripherals that work on larger volumes of data can use block rams as data interface to the processor. In this case the second port of the block ram is not connected to the instruction fetch stage of the core. Thus, the peripheral can work autonomously on the data in the memory block. This can be regarded as DMA style operation. It should be noted that we do not support true master mode DMA operations. This is caused by the simplified memory management in the SpartanMC which reserves exclusive access to the ports of the BlockRAMs to either the code fetch stage or the data access stage of the processor pipeline. In our experience, the missing

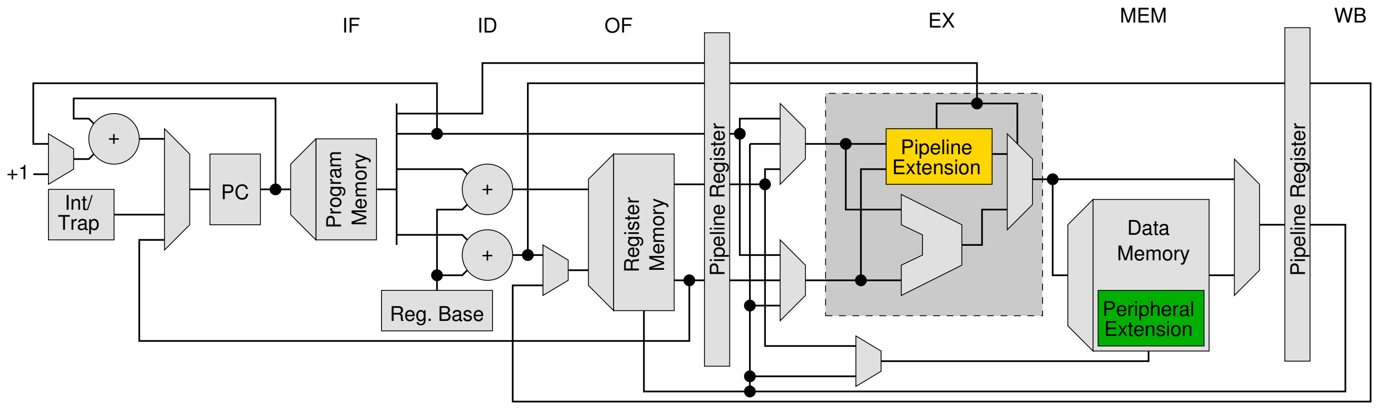

Fig. 1. Microarchitecture of the SpartanMC processor core

master mode DMA would only lead to copying overhead if data needs to be buffered between processing it with different peripherals. As most systems realized with the SpartanMC carry out dedicated functions, this does not occur.

Peripherals are connected to the data memory interface of the processor core. In order to avoid tri-state buffers, all incoming data is combined through a wide or-gate. Thus, all peripherals that are currently not addressed must provide a value of zero on their outputs.

A wide selection of standard peripherals is available: Digital IO, UART, SPI, I^2C, Timer with configurable capture and compare, IRQ controller. Furthermore, some more specialized interfaces are available: JTAG-interface, controller for segment based LCDs, intelligent stepper control unit and ultra sonic range measurement.

More complex peripherals using DMA style interfaces are available for the following protocols: USB 1.1 and USB 2.0, which requires an external physical layer circuit. CAN bus with configurable number of message filters, ProfiBUS master and slave which also provides additional capabilities for analyzing the traffic and to monitor the transmission lines (even for disrupted or ill formed transfers), LCD controller for pixel based displays.

C. Tools

The usability of the presented hardware components is very poorly if we do not supplement them with a number of software tools.

Appropriately configuring and connecting a selection of components may require non negligible knowledge of FPGAs and also may be very tedious. Thus, we have developed a system builder tool. It is a graphical application that allows in-experienced developers to define application specific systems. It generates all required Verilog sources and user constraints to synthesize the full system.

Programming of the processor should be done on the highest possible abstraction level to increase the programmers productivity. Currently, we provide a C compiler and linking assembler for this purpose.

For debugging and performance analysis purposes we provide a cycle accurate simulator. It allows setting of breakpoints and gathers statistics about the executed code. Also, it allows to integrate models of peripherals which can be cosimulated with the code.

The tools of the SoC-kit and the implementation flow are explained more thoroughly in [10].

IV. Options for Application Specific Hardware Acceleration

To accelerate a software application on the SpartanMC we have chosen two different approaches. Each of these approaches can be regarded as trade off between hardware usage and resulting speedup.

A. Peripheral Extension

One possibility to accelerate applications on the SpartanMC is the extension with a dedicated peripheral. As shown in figure 1 the new peripheral (green box) is mapped to the data memory using one of the concepts mentioned above. The development objective for such a peripheral unit is to integrate as much as possible of the target application in hardware. Basically, it would be possible to delegate processing steps back to software. As in this case an overhead of two clock cycles plus the time for the delegated operation would occur, it should only be taken into account if the amount of resources for this processing is extremely large. Thus, we may be forced to implement even parts of the algorithm in hardware which are not appropriate for a good hardware mapping.

A crucial point is the interface for such a peripheral unit. Typically, the processor is a bottleneck for data movement as we do not have master mode DMA. Even if the peripheral provides a large buffer for incoming and outgoing data the processor requires one clock cycle to load each data word to the peripheral buffer. This bottleneck can be neglected for streaming based algorithms which start their calculation with the first incoming data. Whereas it may be a serious drawback for algorithms which work on complete data blocks. An interesting feature of peripheral implementations is the possibility to work asynchronously to the host processor. This option implies the runtime overhead of interrupt handling and is only suitable for algorithms with large computing time.

A mentionable fact is the constant amount of hardware which is used for each peripheral interface, e.g. for address decoding or output scheduling. This hardware effort may be small especially for the SpartanMC but affects the efficiency gain if the peripheral unit and the algorithm are too simple.

B. Instruction Set Extension

Another possibility to accelerate applications on the SpartanMC is the usage of instruction set extensions. Typically, those extensions require hardware which is connected to the pipeline as additional execution unit. These afford a more selective hardware mapping which results in smaller hardware. At the SpartanMC we are able to use free opcodes for R-Type instructions to extend the pipeline. The existing ALU reads two 18 bit operands as input. It computes results for each R-Type instruction simultaneously independent from the opcode. The required output is selected afterwards with a multiplexer which is controlled by the opcode. To integrate a pipeline extension we have to expand this multiplexer and have to connect the input registers to our additional execution unit. In figure 1 the pipeline extension is shown as yellow box. As shown for the peripheral extension the pipeline extension also has a data transfer bottleneck. With a pipeline extension we are limited to two 18 bit operands. If we require more input data for calculation we have to serialize the data stream which affects the runtime. Compared to peripherals, pipeline extensions afford a much smaller scope for customized interfaces. Due to this fact, we demand an accurate application profiling to identify the profitable code section for pipeline extension.

C. Application Profiling

For our test application we are able to use the SpartanMC simulator to gather the required profiling data manually. The simulator allows us to step cycle accurate through an application and to observe the pipeline stages. Furthermore, we are able to set arbitrary break points on function and instruction level and generate code statistics. It provides the possibility to identify critical code sections for potential instruction set and pipeline extensions.

V. Evaluation

To test our approach we have chosen two compute intensive examples from different application domains. Firstly, we implemented an Advanced Encryption Standard (AES) [11] encryption which represents a typical cryptographic algorithm for embedded systems. Secondly, we chose Joint Picture Expert Group (JPEG) [12] decoding as a typically image processing algorithm. The first step was the implementation of each application in software which we used as baseline for our further evaluation.

A. JPEG Decoding

A JPEG image consists of the following elements: A Huffman coded stream which contains 64 byte blocks of amplitudes of the frequency coded source image. Each amplitude block contains the frequency domain representation for one color channel. The JPEG color space consists of one luminance component (Y), represents the brightness and two chrominance components (Cb and Cr) represents the color of the source image. For our JPEG decoding application we assume that one luminance block is assigned to one pair of chrominance blocks which results in no undersampling which is typically called 4:4:4 format. In reality most JPEG images assign one pair of chrominance blocks to 4 luminance blocks which is described by a proportion of 4:2:0. Additionally, a JPEG image is quantized and all amplitudes will be shifted by a constant value of 128.

Decoding a JPEG image starts with the transformation of the entropy encoded input stream into a 64 byte block of amplitudes. We assume that the data stream is stored in the processors internal memory. The processing of each block consists of the following steps: dequantisation, inverse zigzag reassignment of frequency values, inverse discrete cosine transformation (IDCT) which transforms the block in the spatial domain and finally inverse level shift which inverts the offset of 128. The resulting blocks are stored in a buffer for color space transformation which converts one Y one Cb and one Cr block to one block in RGB color space.

The runtime of the software JPEG decoder was determined with the SpartanMC simulator by setting break points to the relevant code sections. The values for entropy decoding, dequantisation, IDCT and inverse level shift are shown in table I. The measurements are corresponding to the operating time for one 64 byte block. For the following color space transformation we assumed two additional blocks. Thus, the results can not be simply extrapolated to complete image size.

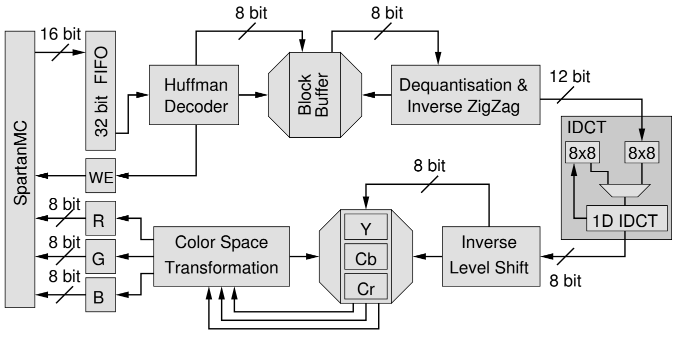

Fig. 3. Architecture of the JPEG peripheral module

To implement the JPEG peripheral extension we used the existing software as guideline. As shown in figure 3 we implemented a 16 bit data input as part of a 32 bit FIFO buffer. The input can be written if a write enable signal is set (WE register). To simplify matters, the write enable state is determined by the processor through polling. The 32 bit FIFO buffer holds the Huffman stream data. The decoding of the Huffman stream is implemented as table lookup. Thus, we are able to translate one codeword with each clock cycle. Therefore, we require 64 clock cycles to decode one block.

TABLE I
COMPARISON OF DIFFERENT IMPLEMENTATIONS FOR JPEG DECODING AND AES ENCRYPTION (EXECUTION TIME IN CLOCK CYCLES)

JPEG Decoding

Processing Step	Software Execution	Peripheral Extension	Pipeline Extension
Load/Store	–	64	–
Entropy Decoder	1460	64	1460
Dequantisation	4238	64	4238
IDCT	302424	130	416
Levelshift	1993	1	1993
Color Transform.	6659	64	384
$\sum$	316774	387	8491

AES Encryption

Processing Step	Software Execution	Peripheral Extension	Pipeline Extension
Load/Store	–	48	–
Key Expansion	9627	10	9627
Add Round Key	3384		3384
Shift Rows	528	12	528
Mix Columns	9670		240
Sub Bytes	2431		2431
$\sum$	25640	70	16210

If we assume each codeword with a length of 16 bit, which represents the worst case scenario, we need additional 64 cycles to reload the Huffman stream data to the 32 bit FIFO buffer. After the Huffman decoding stage the complete 64 byte block is stored in a buffer. The dequantisation stage is implemented as a multiplication of one codeword and a table value. This stage consumes 64 clock cycles. The inverse zigzag is implemented through controlled read operations of the block buffer which consumes no additional clock cycles. The common implementation of the IDCT algorithm uses two convolution operations to process one block. This two dimensional IDCT can be parted into two consecutive one dimensional IDCTs (1D IDCT). A schematic view of a 1D IDCT is shown in figure 4, it is based on synthesis results of Xilinx ISE. To compute one amplitude value with eight sets of coefficients we require one clock cycle. This results in 64 clock cycles for a 1D IDCT of one block. Since we need one cycle to load operands and use pipelining for our algorithm we are able to compute the complete 1D IDCT in 65 cycles. This results in a total of 130 cycles for the whole two dimensional IDCT. After the IDCT step all codewords are corrected to their final value through the inverse level shift. This can be performed combinational and consumes no clock cycles. Finally, we need to carry out the color space transformation which consumes only 64 clock cycles due to parallel processing of all three 8 bit color components. The resulting clock cycles for processing one block are shown in table I. As shown in table II we have synthesized the JPEG peripheral on a Xilinx FPGA (Spartan-3 XC3S400) which consumes 3441 lookup tables (LUT) and 13 multipliers. Of these, we use one multiplier in the dequantisation stage, four for the color space transformation and eight for the IDCT.

Furthermore, we implemented pipeline extensions for the IDCT and the color space transformation. As shown in table I the IDCT and color space transformation consume the largest amount of clock cycles in our algorithm which makes them good candidates for pipeline extensions. Furthermore, both processing steps mainly consist of complex arithmetic operations which makes them suitable for pipeline extensions. Also, an interesting candidate could be the dequantisation stage, which requires a large amount of clock cycles. But the dequantisation step performs only one multiplication which

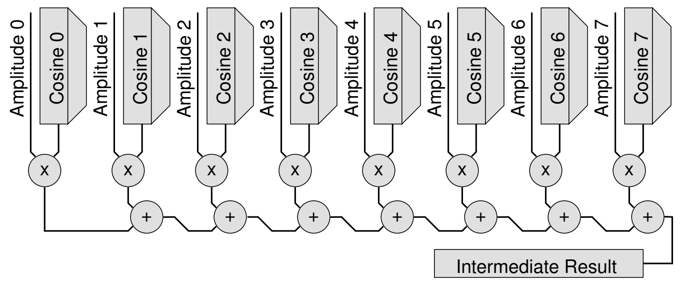

Fig. 4. 1D IDCT based on Xilinx ISE schematic view

will provide no speed up compared to the existing ALU.

To perform the IDCT in a pipeline extension we also implemented a 1D IDCT as shown in figure 4 and compute the whole IDCT in two iterations. As input for the 1D IDCT we read each amplitude from one operand register. Each amplitude is multiplied with a different cosine value which is taken from a table inside the extension. For a 1D IDCT we require eight clock cycles to load the input data, four clock cycles to write it to the operand registers of the pipeline extension and eight to compute the 1D IDCT results. Finally, we require eight additional clock cycles to store the data. Loading and processing of the data can be overlapped with a combined instruction. Thus, we can save two clock cycles. In total one 1D IDCT consumes 26 clock cycles for eight input values. The whole IDCT consumes 416 clock cycles for a complete block. A hardware implementation of the IDCT requires eight multipliers and 392 LUTs. The color space transformation requires two clock cycles to read all three input values from processor registers. The SpartanMC provides a 18 bit width data path which allows us to load two 8 bit values for the color space transformation at once. The calculation itself could be implemented in one clock cycle if we would use four multipliers. But since we use two clock cycles to write back all data, there is no need to complete the calculation within one cycle, because we are not able to write three results at once. As the calculation of the color components R and B both uses one multiplier and the G component uses two, we are able to split the calculation stage and share it with the write back stage which allows us to use two multipliers instead of four. The implemented extension needs six clock cycles for three input values and 384 clock cycles for a whole block. The implementation of the pipeline extension for color space transformation consumes 33 LUTs and two multipliers as shown in table II. The total amount of clock cycles for JPEG decoding is shown in table I. All processing steps without hardware acceleration are carried out in software and consume the runtime measured in the baseline.

B. AES Encryption

As a second benchmark application we implemented an AES encryption which is a special form of the Rijndeal-Cipher.

AES is a symmetric block cipher which uses a block size of 128 bit divided into four rows and four columns of bytes. The encryption is carried out in rounds. The number of rounds depends on the key length. For our test application we used a key length of 128 bit which results in ten rounds for a single block encryption. At the beginning of the encryption rounds the key expansion will be executed. This initial step expands the master key and produces an expanded key at the size of 44 byte ($\#columns * (rounds + 1)$). Due to this expansion each round can be calculated with another part of the expanded key. The key expansion is called only once for all blocks in the source data. The AES algorithm uses the following basic operations:

- Substitution. Replaces each byte through a constant table value. The table is called S-Box.
- Permutation. This step shifts the row to the left. Each row is shifted with different offset depending on its row number.
- Diffusion. This operation computes a matrix multiplication in GF(2^8).
- Add round key. Encrypts the data through adding the round key to the current block byte.

Given that key expansion already took place, the following sequence of operations forms a block encryption:

- Initially, the round key is added.
- 10 rounds of substitution, permutation, diffusion and round key addition are processed.
- Finally, substitution, permutation and round key addition finish the block encryption.

In the software implementation of the AES algorithm we measured the runtime for the encryption of a single 16 byte block. This includes one call of key expansion and one block encryption (12 round key additions, 11 substitutions, 11 permutations, 10 diffusions). The runtime measurements for the software were made through the SpartanMC simulator. Table I shows the total clock cycles for AES encryption.

For the AES peripheral we implemented one 16 bit input register as shown in figure 5. Thus, we require eight clock cycles to write one 16 byte block and eight additional cycles to write the key from the SpartanMC to the peripheral. The proper execution of each cipher stage is implemented in a control unit. Furthermore, this unit can be used to switch between

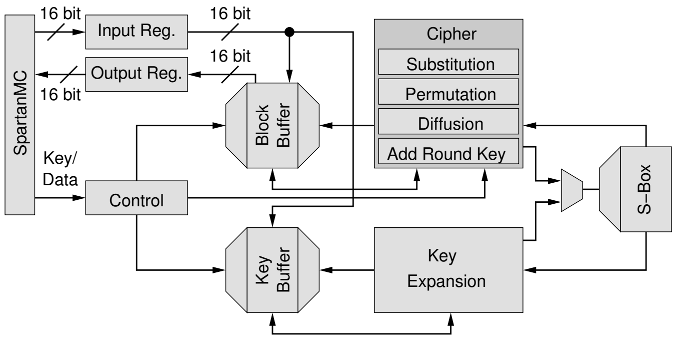

Fig. 5. Architecture of the AES peripheral module

key input or data input. The key and the current block data are stored in a dual ported memory which can be accessed from the cipher and key expansion hardware and from the input register. Firstly, we need to expand the master key using the S-Box. This is carried out sequentially and consumes ten clock cycles, although we could build faster implementations. As this part of the hardware is used only once during the encryption of many blocks, we chose to trade space for time and use the smaller implementation. All following steps are executed in each round. The substitution step also requires the S-Box table to substitute each block byte. We use eight dual ported BlockRAMs to store the S-Box which allows us to substitute 16 bytes at once. In figure 5 we have shown only one memory block to simplify the diagram. The permutation step is quite simple. It can be implemented through interchanging of output wires. The GF(2^8) matrix multiplication in the diffusion step is also carried out in parallel through combinational blocks. Finally, we have to encrypt the data within the add round key step which is implemented as xor operation. Since we can map all equations of AES within combinational logic, we are able to complete one cipher round within one clock cycle. As we call some processing steps 12 times we require 12 clock cycles to complete the whole block encryption. Eventually, we have to transport the data to the processor which is carried out through a 16 bit output register. This consumes another eight clock cycles for one block. Overall, we require 70 clock cycles, because we require one additional cycle for each load and store operation. Due to many combinational blocks the AES encryption hardware is quite large. As shown in table II we require 5650 LUTs and eight block rams on a Xilinx Spartan-3 XC3S400 FPGA.

As pipeline extension for AES we have chosen the diffusion step which consumes the largest amount of clock cycles. It requires high computational effort but only four bytes (one row) as operands. To execute the special operation we need to load four byte into two processor registers which consumes two clock cycles. Accordingly, we execute the first special diffusion operation and perform a write back for the first two values in one clock cycle. Afterwards, we need one clock cycle to store the first two values. In the fifth cycle we perform a write back for the second two values and in the sixth cycle we store the second values. One row of the diffusion step consumes six clock cycles. Hence, we need 24 clock cycles for the whole block. The total amount of clock cycles for AES encryption is shown in table I. The processing steps without hardware acceleration are carried out in software and consume the runtime measured in the baseline. The hardware effort of the pipeline extension is 255 LUTs as shown in table II.

TABLE II
COMPARISON OF RESOURCE UTILIZATION FOR JPEG DECODING AND AES ENCRYPTION

Implementation	**4-Input LUT**	**Multiplier**	**block ram**
JPEG Peripheral	3441	13	–
JPEG Pipeline Extension	425	10	–
AES Peripheral	5650	–	8
AES Pipeline Extension	255	–	–

TABLE III
COMPARISON OF SPEEDUP AND EFFICIENCY FOR DIFFERENT IMPLEMENTATION OF AES ENCRYPTION AND JPEG DECODING

Implementation	**Speedup**	**Speedup/LUT**
JPEG Peripheral	818.54	0.2379
JPEG Pipeline Extension	37.3	0.0878
AES Peripheral	366.2	0.0648
AES Pipeline Extension	1.58	0.0062

C. Comparison

We try to compare the efficiency of pipeline extensions and peripheral extensions. Assuming sufficient resources for our hardware, we implemented both variants for two different algorithms. To compare the results we calculated the speedup for each extension method relating to the software baseline. The speedups for the different algorithms are shown in table III. Evaluating our approaches we compare the efficiency which we define as speedup per LUT (speedup per area). We have taken the number of LUTs as measurement for additional hardware effort because we believe it is the best indicator for hardware usage on FPGAs. If we compare the efficiency of both algorithms we see that the efficiency of peripheral units is ten times higher for AES encryption and nearly three times higher for JPEG decoding. Sometimes, the availability of block rams could be regarded as bottleneck for FPGA soft cores. But even, if we use distributed RAM for our AES peripheral extension, which cost approx. 2k additional LUTs, we have a seven times higher efficiency compared to the AES pipeline extension. Furthermore, we made no special efforts to optimize our peripheral hardware. In regard to this evaluation we claim that it is evident that the peripheral implementation of the algorithms always represents the better solution for application acceleration in hardware.

VI. CONCLUSION

In this contribution we have analyzed different variants of customizing soft cores in FPGA based SoC implementations: Instruction set extensions and peripheral modules. We implemented applications from two different application domains in three variants: Pure software, software with instructions set extensions and as peripheral module. We have defined an efficiency metric to evaluate the quality of the different variants. Surprisingly, the efficiency of the peripheral implementation is always considerably higher than the efficiency of the instruction set extension. We believe that these findings also hold for other soft cores and for other application domains. Thus, our overall conclusion is the recommendation to implement soft core customizations always as peripheral modules.

We are currently working on a gcc based compiler that automatically extracts performance critical parts of the application and maps them to hardware components.

REFERENCES

[1] Tensilica Inc., "Xtensa customizable processors," http://www.tensilica.com/products/xtensa-customizable.htm, 2010.

[2] Altera Corp., "Nios ii custom instruction user guide," documentation, 2008.

[3] Stretch Inc., "Stretch technology," http://www.stretchinc.com/technology/, 2010.

[4] Xilinx Inc., "Microblaze processor reference guide," documentation EDK 11.4, 2009.

[5] Triscend Inc., "Triscend e5 configurable system-on-chip platform," data sheet 1.06, 2001.

[6] ——, "Triscend a7s configurable system-on-chip platform," data sheet 1.10, 2002.

[7] N. Kasprzyk and A. Koch, "High-level-language compilation for reconfigurable computers," in *ReCoSoC*, 2005.

[8] H. Lange and A. Koch, "Architectures and execution models for hardware/software compilation and their system-level realization," *IEEE Trans. on Computers*, vol. 99, no. PrePrints, 2009.

[9] G. Hempel and C. Hochberger, "A resource optimized processor core for FPGA based SoCs," in *DSD*, 2007, pp. 51–58.

[10] ——, "A resource optimized SoC kit for FPGAs," in *FPL*, 2007, pp. 761–764.

[11] J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.

[12] A. M. Leger, T. Omachi, and G. K. Wallace, "Jpeg still picture compression algorithm," *Optical Engineering*, vol. 30, pp. 947 – 954, 1991.