Design of a Processor Optimized for Syntax Parsing in Video Decoders

Nicolas Siret, Jean François Nezan, Aimad Rhatay

To cite this version:

Nicolas Siret, Jean François Nezan, Aimad Rhatay. Design of a Processor Optimized for Syntax Parsing in Video Decoders. Conference on Design and Architectures for Signal and Image Processing (DASIP), Nov 2011, Tampere, Finland. pp.CD. hal-00661330

HAL Id: hal-00661330
https://hal.science/hal-00661330

Submitted on 19 Jan 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DESIGN OF A PROCESSOR OPTIMIZED FOR SYNTAX PARSING IN VIDEO DECODERS

Nicolas Siret[1] *and Jean-François Nezan*[1] *and Aimad Rhatay*[2]

[1] European university of Brittany, France
INSA, IETR, UMR 6164, F-35708 RENNES
[2] Lead Tech Design
F-35700 Rennes, France

ABSTRACT

Heterogeneous platforms aim to offer both performance and flexibility by providing designers processors and programmable logical units on a single platform. Processors implemented on these platforms are usually soft-cores (e.g. Altera NIOS) or ASIC (e.g. ARM Cortex-A8). However, these processors still face limitations in terms of performance compared to full hardware designs in particular for real-time video decoding applications. We present in this paper an innovative approach to improve performance using both a processor optimized for the syntax parsing (an Application-Specific Instruction-set Processor) and a FPGA. The case study has been synthesized on a Xilinx FPGA at a frequency of 100 MHz and we estimate the performance that could be obtained with an ASIC.

Index Terms— Application-Specific Instruction-set Processor (ASIP), embedded processor, processor architecture, Instruction Set Architecture (ISA), bit manipulation operations, video decoding, circuit design, multimedia.

1. INTRODUCTION

Ongoing research in the field of video decoding has shown how to implement efficiently video decoders on recent architectures. Usually these architectures are based on DSPs [1], ASICs [2], or high performance processors. However, DSPs and embedded-processors still face limitations in terms of computing power for applications that require high performance. On the other hand, full hardware solutions on ASICs offer the highest performance and a minimal electrical power consumption but are not flexible. In this paper, we present an advanced hardware-software codesign architecture composed of an Application-Specific Instruction-set Processor (ASIP) and a programmable logical unit (FPGA). Such an architecture is particularly attractive to improve performance while keeping flexibility because it allows the execution of an application to be split into hardware processing for parts that require performance (e.g. IDCT) and software processing for those that require flexibility (e.g. syntax parsing).

Using ASIPs on heterogeneous platforms is an innovative method to improve performance compared to usual embedded processors. Designers can use dedicated instructions that reduce the number of cycles required to execute a program. These dedicated instructions are added to the processor core and to the Instruction Set Architecture (ISA). This offers a good trade-off between flexibility, computing power, and power consumption. This is also a way to decrease the power consumption of a design.

The flexibility offered by embedded-processors on heterogeneous platforms is particularly attractive in the field of multimedia video decoding because it allows the implementation of the application to be updated when new video standards are published. This is also true for the syntax parsing components which extract video data from a compressed bitstream. In fact, it allows complex algorithms (e.g. CABAC) to be added, if necessary, in a second step of development. The syntax parsing is the part of the decoding process which is the least studied in the literature because it is almost sequential and non-optimizable. In a previous paper, we presented a method and results to implement a video decoder on a codesign architecture made up of a general purpose soft-core and FPGA. Performance was restricted by the hardware soft-core generalist architecture and the software syntax parser which was not optimized to be processed on a soft-core [3]. This paper presents our work on the codesign architecture, especially on the processor architecture so as to improve performance while keeping flexibility. It makes the following contributions:

- We show the MPEG-4 video decoder implementation on the heterogeneous platform in particular the flexible parser (section 3). The parser takes advantage of a dedicated function called *ShowNbits*. This function efficiently extracts an arbitrary number of bits from a bitstream no matter the processor architecture.
- We show how to modify the source code of a syntax parser to take advantage of a specific instruction called **showbits** (section 4). This specific instruction extracts an arbitrary number of bits from a word in a single cycle.

- We present the ASIP hardware architecture and describe how the specific **showbits** instruction is implemented into the hardware core of the soft-core (section 5).

The paper introduces results obtained with our method for an MPEG-4 video decoder synthesized on a Xilinx Virtex-4 platform at a frequency of 100 MHz. The benefit is about 30% more FPS compared to general purpose soft-core processors.

2. BACKGROUND

This section presents related work on video decoder implementations, on advanced bit manipulation operations and our approach compared to others.

2.1. Application-Specific Instruction-set Processors

ASIPs are general purpose processors with specific instructions added to their Instruction Set Architectures (ISA) that aim to improve performance and to simplify the development of applications. ASIPs offer a good tradeoff between general purpose processors (RISC or CISC architecture) and full hardware solutions (on ASICs) by providing software scalability and better performance than general purpose processors. This results in a reduced time to market for IPs providers and a lower software complexity than fully optimized software solutions.

The hardware implementation of an ASIP consists of a general processor with the addition of specific instructions within the ALU, or functional units within the core [4, 5]. In short, the software implementation consists in updating the cross compilation tools, usually GCC/GDB. Thus, the development of an ASIP is carried out in four main steps:

1. updating the core of the processor by adding the hardware code required to process a specific instruction,
2. updating the ISA and the software compiler (e.g. GCC),
3. adding the specific instruction to the debugger (e.g. GDB) to allow designers to debug their applications,
4. testing and validating the operation of the added instruction on the processor.

In our field of research (i.e. video decoding), adding bit manipulation operations to the core of an ASIP offers significant profits [6]. In fact, these bit manipulation operations are always required to process the syntax parsing while they are not well supported by common general-purpose processor architectures. Indeed, general-purpose processors efficiently process 8, 16, 32-bits size arithmetic operations (e.g. addition, subtraction, multiplication, etc.) but not bit-size operations which are required to extract an element arbitrary. Ongoing research presents various methods [4, 7] to implement these bit manipulation operations. The hardware implementation is usually realized using functional units (i.e. hardware components which process a specific algorithm in parallel with the ALU) added to the core of the processor We chose to add the dedicated instruction within the ALU rather than in independent functional units to avoid modifying the core architecture and processing.

2.2. Related Work

Real-time video decoding is difficult to achieve on embedded processors because it requires an efficient implementation and a high-performance processor or DSP [5, 8]. Usually, one or more coprocessors are implemented in addition with the processors to process algorithms that required high performance, like the Inverse Discrete Cosine Transform or the bitstream parsing (i.e. CABAC and CAVLC) [9]. However using a coprocessor requires modification of the software code, restricting the code flexibility and portability. On the contrary, ASIPs can be used to keep the software code portability.

In the field of video and audio decoding, syntax parsing extracts bits from a bitstream and transfers these bits to the other video or audio decoding entities [9]. To read an arbitrary number of bits from a bitstream, the usual methodology (which is described in the standard) consists of coding and using a *ShowNbits* function based on bit manipulation operations [5]. However, these operations are not well supported by common general-purpose processor architectures.

Researchers have presented methods to efficiently manage bit manipulation operations with results obtained on general purpose processor architectures [8], DSPs [1, 5] or applications without real-time constraints [4]. This leads to a lack of exploration of real-time applications implemented on codesign architectures even though they are more and more used. In fact, they offer more performance with a lower power consumption compared to common processor architectures. We complete ongoing research by presenting a complete implementation of a real-time MPEG-4 video decoder on an architecture made up of an ASIP and hardware IPs implemented on a FPGA.

3. VIDEO DECODING AND SYNTAX PARSING

This section describes the MPEG-4 video decoder architecture and the parsing algorithm that uses the *ShowNbits* function.

3.1. Architecture of a video decoder

The video decoder presented in this paper is one of the applications provided by the Reconfigurable Video Coding [10, 11] (RVC) framework. It is composed of three main parts: the Parser that extracts compressed video data from the bitstream,

the AC/DC prediction (sometimes referred as the "(intra, inter) Prediction") that predicts video data from blocks in the same image and the Inverse Discrete Cosine Transform. The MPEG-4 video decoder architecture is introduced in Fig.1.

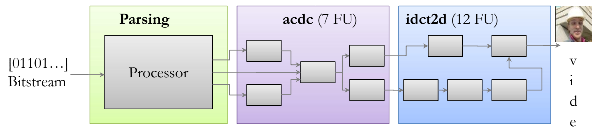


Fig. 1. MPEG-4 video decoder architecture.

The video decoder is a dataflow program described as a network composed of the three subnetworks Parser, AC/DC and IDCT2D. Unidirectional optimized FIFOs are implemented between each actor of the networks to ensure the correct transmission of data. The AC/DC and IDCT2D subnetworks are suitable to hardware processing because they are made up of blocks, also called Functional units (FUs), which process almost parallel algorithms (thus data are pipelined). On the other hand, the Parser performs almost sequential algorithms (e.g. reading input bit, storing input bit, looking for start code ...) and thus, is suitable to software processing. The AC/DC and IDCT2D FUs compute video data (4:2:0 configuration) using the decoded data: QP, $last$, run, $value$, provided by the Parser.

3.2. Syntax parsing algorithm

The scalability of processors is valuable to developing and updating parsing algorithms when new standards are published. All the more so, independently of multimedia applications, these algorithms may be updated and are almost sequential. We implemented the parser on an embedded processor and updated its ISA to evaluate the performance. The MPEG-4 video parser provided by RVC is still limited on embedded processors because it has not been optimized for software processing [3]. Thus, we replaced it with a hand-coded parser which has been developed according to the MPEG-4 standard. This parser has been optimized to supply real time video on DSPs [12]. Moreover, this method is compliant with the RVC framework which encourages the replacement of critical performance FUs by software or hardware IPs.

A detailed description of the MPEG-4 parser is beyond the scope of this paper (but presented in the MPEG standard [13]). However, an overview of the hand-coded parser is presented in Fig. 2. Ten steps are necessary to extract video data from the compressed video bitstream. In steps (1), (2), (3), (4) (within $VideoObjectLayer$), the Parser waits for a start code. Depending on the "Video Object" information provided by the start code, it updates the "Video Object Layer"(VOL) or creates a new one. The VOL contains the video object definition and the layer parameters (e.g. image resolution, scalability, etc.) extracted from the bitstream. Then, in steps

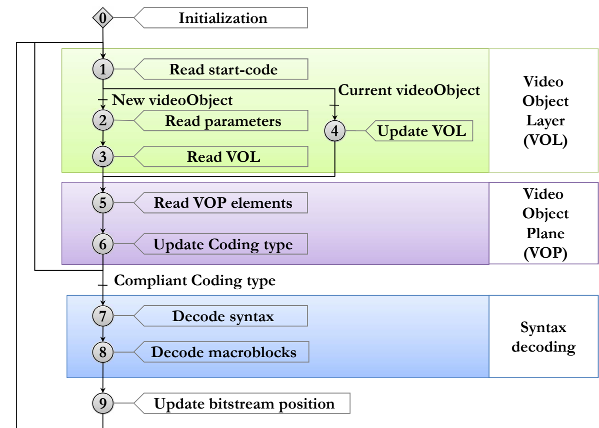


Fig. 2. The parsing processes in ten steps.

(5), (6) (within $VideoObjectPlane$), the parser extracts the frame or plane parameters (e.g. quantification, macroblock type, etc.) and builds the "Video Object Plane" (VOP). Finally, in step (7), (8) (within $DecodeSyntax$), the parser computes the video elements (QP, $last$, run, $value$) using the VOL and VOP data. In step (9), it updates the pointer of the bitstream position. The video elements are sent to the IOs of the processor using an aRDAC specific instruction.

3.3. Software implementation of the syntax parsing

The syntax parser component reads bits from the bitstream, realizes tests and sends compressed video data to the other components of the video decoder. To read an arbitrary number of bits from a bitstream, an efficient method consists of coding an optimized *ShowNbits* function that will be called when necessary. The locations and numbers of bits (which is set for each feature to extract) is defined in the standard. Within the MPEG-4 video parsing software, the *ShowNbits* function is called more than a thousand times [14] per sequence. Thus, performance of the parsing process depends on the speed of this bit-extraction operation.

The hand-coded parser also uses a *ShowNbits* function to extract N bits from an M-bit size word. The *ShowNbits* function requires three arguments: the bitstream, the location of the first bit to read and the number of bits to read. Depending on the processor design, two different algorithms are usually processed. In big-endian architectures, see in Fig.3, the algorithm first loads part of the bitstream into a register, next left shift the word in register to remove unnecessary left bits, then right shift the word to replace the word in the register (and remove unnecessary right bits) and finally return the result. In little endian architectures, bit mask operations are also necessary.

Our algorithm, presented in Fig.4, ensures the portability of both architectures. It has the following behavior:

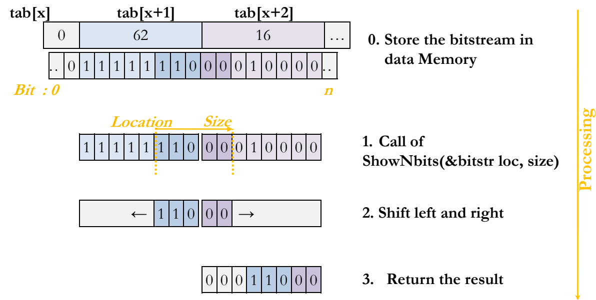


Fig. 3. Execution of the *ShowNbits* function on a big-endian architecture.

1. the position of the bit (pos_bit) and the position in the word (pos_char) are computed,
2. the data to extract are stored in a register so as to keep only the required bits. The four-steps loading ensure the portability between big-endian and little-endian architecture,
3. unnecessary bits are masked (using binary and operations),
4. the N bits word is right shifted (depending on the location and the size of the bits to extract),
5. the N bits word is returned.

```
static ulong showNbits (const unsigned char *const
                        RESTRICT tab,
                        const long position,
                        const int n) {
  const int       pos_bit  = position & 0x7 ;
  const long      pos_char = position >> 3;
  unsigned long   d;

  d = *(tab + pos_char) << 24 ;
  d = d | *(tab + pos_char + 1) << 16;
  d = d | *(tab + pos_char + 2) << 8;
  d = d | *(tab + pos_char + 3);
  d = d & 0xffffffff >> pos_bit;
  d = d >> (32 - pos_bit - n) ;
  return (d);
}
```

Fig. 4. Algorithm of the *ShowNbits* function using 32-bit big-endian architecture.

Although, the *ShowNbits* function is quite simple, it requires several cycles to build the M bits size word, to shift and to mask (if necessary) the bits. Hence, a specific instruction which processes these operations allows the performance to be improved [4, 7].

4. SOFTWARE IMPLEMENTATION OF THE SHOWBITS INSTRUCTION

To improve the performance, we added a **showbits** instruction into the core of the processor. This instruction loads the data from the bitstream and processes the shifts operations in a single cycle. We present in this section the modifications realized on the ISA to add this specific instruction and the optimizations realized on the software code of the *ShowNbits* function. These optimizations aim to reduce the number of cycles required to process the *ShowNbits* function.

4.1. Updating the aRDAC Instruction Set Architecture

We updated the the cross compilation tools (GCC and GDB) to add the **showbits** instruction into the ISA. In this way, first we patched the GCC-GDB definition files and then we rebuilt the cross compilation tools. The ISA architecture requires the use of two data registers and an operation code. The operation code defines the instruction for both the processor and the compiler. It is composed of a code operation (i.e. a binary label which identifies an instruction) and two arguments (i.e. two input registers). We define the code operation as **showbits**, the first argument is the register, which contains the bitstream, and the second argument is the register that contains the location and the size of the N bits word to extract.

4.2. Using the showbits instruction within the *ShowNbits* function

The *ShowNbits* function can be used, either directly in the C program, or in an assembler part of the program. We chose to hand-write it in assembler to avoid the creation of useless register initialization operations by GCC. As shown on Fig.5, the updated *ShowNbits* function has the following behavior:

1. the required variables and registers are initialized (not presented in the figure),
2. the default value of the fixed variables is loaded within registers (e.g. $7, 8, etc.) (not presented in the figure),
3. the current location in the bitstream is computed (not presented in the figure),
4. the location and the size are concatenated in a single word which is stored in the first input register (the location is 4-bits left shifted) (lines 13 to 24 and 29),
5. the required part of the stream is loaded from the data memory and stored in the second input register (line *25*),
6. the **showbits** dedicated instruction is called (line *36*),
7. finally, the result of the **showbits** instruction is return (line *40*).

```
static __inline ulong showNbits (const unsigned
                                  char *const RESTRICT tab,
                                  const long position,
                                  const int n ) {
  unsigned long    pos_char;
  unsigned long    pos_size;

  //
  // The initialization, the bitstream loading, and the
  // location and the size computing are not presented
  // in the figure.
  //

  asm  volatile(        "and␣%0,%1"
                        : "=r" (pos_size)
                        : "r" (position), "0" (pos_size)
                        );
  asm  volatile(        "add␣%0,%1"
                        : "=r" (pos_char)
                        : "r"  (tab), "0" (pos_char)
                        );
  asm  volatile(        "lsli␣%0,#0x8"
                        : "=r" (pos_size)
                        : "0"  (pos_size)
                        );
  asm  volatile(        "ld␣%0,(%1)"
                        : "=r" (pos_char)
                        : "r"  (pos_char)
                        );
  asm  volatile(        "add␣%0,%1"
                        : "=r" (pos_size)
                        : "r"  (n), "0"(pos_size)
                        );
//
// done ! pos_size  = n + ((position & 0x7) << 8);
//
  asm  volatile(        "showbits␣%0,␣%1"
                        : "=r" (pos_char)
                        : "r"  (pos_size) , "0" (pos_char)
                        );
  return (pos_char);
```

Fig. 5. Part of the optimized *ShowNbits* function which use the **showbits** instruction.

In addition to these optimizations, the code of the *ShowNbits* function is inlined by the compiler so as to avoid the loss of cycles due to the function call overhead. This allows the generated assembler code of the parser to be smaller and more efficient.

5. HARDWARE IMPLEMENTATION OF THE SHOWBITS INSTRUCTION

We present in this section two main points, first the hardware implementation of the **showbits** instruction within the core of the processor, then the optimizations realized on the processor (in particular on the memory architecture) to improve the performance.

5.1. Overview of the "'aRDAC" processor

The Parser is compiled to be processed on an embedded processor. We chose a processor called "aRDAC" because it is a soft-core optimized for embedded targets and has a core that can be modified. It is based on a three stage pipeline Harvard architecture with a CISC type instruction mode, but which operates as a RISC-type processor. Precisely, we worked with the 32-bits version of the aRDAC and we used the aRDAC plug-in (GNU toolchain) for Eclipse IDE which uses GCC/GDB. The aRDAC is sold by *Lead Tech Design*[1] company.

The aRDAC processor is composed of eight main components shown in Fig. 6. The data and program memories, the memory controller, the ALU and the various registers are usual components of processors. The IO block manages the inputs and outputs of the aRDAC and the transfer block ensures the transmission of the data between the registers and the data memory.

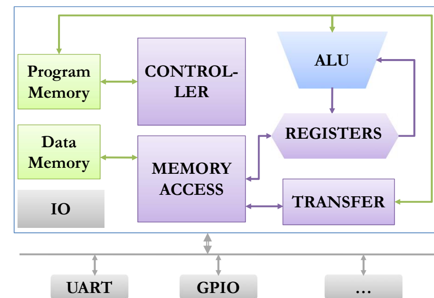


Fig. 6. The aRDAC processor.

5.2. Hardware implementation of the showbits instruction

We added the **showbits** instruction into the core of the aRDAC. It is processed in a single cycle to preserve the RISC architecture and the processing performance of one instruction per cycle (i.e. 1MIPS/MHz). This instruction divides by six the number of cycles required to execute the *ShowNbits* function (see section 6). The **showbits** hardware processing is introduced in Fig.7. The behavior of the instruction is presented below:

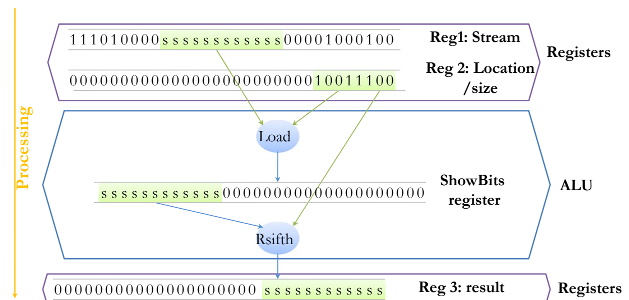


Fig. 7. Execution of the **showbits** within the ALU.

[1] Lead Tech Design : http://www.leadtechdesign.com/

1. the **showbits** instruction is decoded,

2. if necessary the registers used by the **showbits** instruction are initialized,

3. the location and the size of the N bits to extract from the M-bits word are read from the register,

4. Part of the M-bits word is transfered from a global register into a **showbits** register. The transfer is done according to the location to keep only the required bits (others are consequently removed),

5. the required N bits are right shifted,

6. the required N bits are transfered from the **showbits** register into a result register.

As shown in Fig. 8, we modified the ALU to add the **showbits** register and the **showbits** instruction operator. The load from the data memory to the **showbits** register and the use of a barrel-shifter allow this instruction to be executed in a single cycle, while preserving performance of the processor. The additional hardware cost is nearly null because we used the existing barrel-shifter to process the shift operation, and only add a new single internal register.

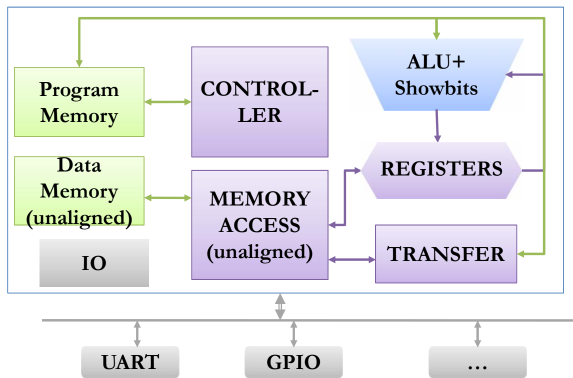


Fig. 8. The aRDAC-ASIP processor.

5.3. Hardware optimizations on the hardware aRDAC architecture

We also updated the memory architecture to optimize the hardware processing, especially when it is necessary to read bits which are not aligned in the memory, in others words, when the M-bits word is split between two consecutive memory addresses (e.g. address n and address $n + 1$) in the data memory. In such a case, most of the processors need at least two cycles to load a datum and to store it into a register. During the first cycle, the first part of the M-bits word is read at the address n in the memory and stored into a register A. Then, during the second cycle, the second part of the M-bits word is read at the address $n + 1$ in the memory and stored into the same A register with the correct location. The memory architecture of these processors is called ”aligned”.

As introduced in Fig. 9, there are two kinds of memory architectures: aligned and non-aligned. A memory built on

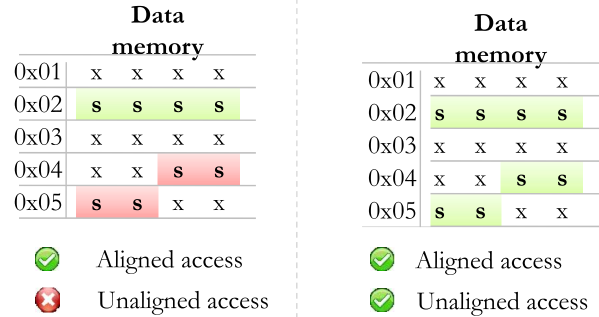


Fig. 9. The two kind of memory architectures: ”aligned” and ”non-aligned”.

an aligned architecture allows the access to be aligned only. A memory access is aligned provided that data read from the memory are 8*k bits long (e.g. 8 bits, 16 bits, 32 bits, etc.) and data addresses are l-byte aligned. If a non aligned access is needed on an aligned architecture (e.g. the data starts at the 3nd byte instead of the 4th byte), then the memory manager may read two 4-bytes word and store the required part of them in a return register, or it may generate an alignment fault. On the other hand, a memory built on a non-aligned architecture allows the access to be aligned and non aligned.

Currently, most designs use aligned memory address for their simplicity. It is then up to designers to optimize their software code to avoid non-aligned memory access. On the other hand, non-aligned memories offer flexibility and performance but require a more complex hardware logic. The complexity of the memory optimization explains that the use of non-aligned memory architectures in processors are largely patented [15,16]. We modified the memory architecture of the aRDAC to perform non-aligned access. In this way, we updated both the memory manager and the data memory so as to have the following behavior (for load operations):

1. the address is decoded by the memory access block,

2. if an unaligned access is detected, the address is split into a MSB and a LSB address,

3. these two addresses are sent to the memory,

4. the MSB and LSB data are read according to the MSB and LSB address,

5. these data are masked and concatenated depending on the address,

6. the result word is transfered to the result register.

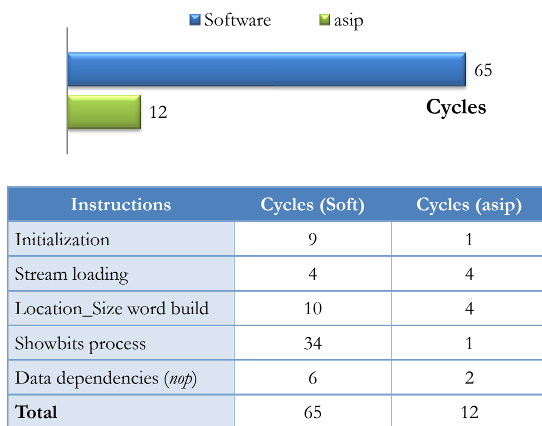
The store operation has similar behavior except that it processes a store operation instead of a load operation.

6. RESULTS

In this paper we focused on an MPEG-4 video decoder case study but our method is available for other applications. The video decoder was implemented in two parts: the Parser compiled to be executed on the optimized processor and the AC/DC and IDCT2D blocks were synthesized as hardware IPs. Results presented are obtained using Modelsim simulator and Xilinx synthesizer (on a Virtex4 - ML402 evaluation platform).

6.1. Results on the *ShowNbits* function

We have first tested the Parser with two MPEG-4 videos (a CIF and a QCIF resolution) on both the aRDAC and the ASIP. The ASIP is the version of the aRDAC with the core modifications presented in this paper. Measurements which are introduced in Fig.10 show that the number of cycles to process the *ShowNbits* function is divided by almost six using the ASIP. In fact, results show that 65 cycles are required for the pure software solution and only 12 cycles for the optimized software solution. The first call of the *ShowNbits* function imposes the initialization of dedicated locals registers which consume four additional cycles.



Instructions	Cycles (Soft)	Cycles (asip)
Initialization	9	1
Stream loading	4	4
Location_Size word build	10	4
Showbits process	34	1
Data dependencies (*nop*)	6	2
Total	65	12

Fig. 10. Number of cycles required to process the *ShowNbits* function, with and without the **showbits** specific instruction.

The pure software solution requires several shifting operations in addition to data processing (e.g. tfr, add, or, etc.) which is why 65 cycles are needed to process the algorithm. By contrast, the use of the **showbits** specific instruction dramatically reduces the number of shifting instructions (i.e. two sift operations to build the pos_size word) and the number of load instructions (i.e. a single instruction loads the bitstream into a register).

6.2. Performance of the MPEG-4 codesign decoder

We then tested the MPEG-4 decoder with the parser compiled for the ASIP and the AC/DC, IDCT2D implemented as IP on the FPGA. The data transfers between the ASIP and the IPs were made through FIFOs. Results presented on Fig. 11 show an increase of 25% of the number of frames per second (FPS).

Design	Operating frequency	FPS (CIF) SAN000	FPS (QCIF) Foreman
aRDAC	100MHz	13	50
aRDAC-ASIP	100MHz	17	63
aRDAC-ASIP	50MHz	8	30

Fig. 11. Performance of the codesign MPEG-4 video decoder.

The maximum operating frequency of the ASIP on this FPGA is about 125MHz which allows about 20 FPS to be displayed. Moreover, both synthesis and simulation results point out that the number of FPS increases linearly with the operating frequency which means that newest FPGA (e.g. Virtex-6) allow displaying a real-time CIF video. Nowadays, in the field on embedded processing, real time full HD videos (i.e. 1080p, 30 FPS) are still rarely supported because only full hardware designs on ASICs are able to display such resolutions. In fact, in order to display this resolution it is necessary to compute about 62,208,000 pixels per second which is equivalent to 16 pixels computed every nano-second. On the other hand, real time HDTV video (i.e. 720p 30 FPS) is more and more supported thanks to co-design architectures. On the design presented in this paper and without optimizations, an operative frequency of nearly 1.6GHz for the ASIP is necessary to process a real-time video decoding (2,2GHz with the general purpose ISA).

Finally, we use tools of ASIC creation to get the final performance. Results show that the maximum operative frequency for the aRDAC-ASIP is about 1GHz. This result is interesting because our previous experiences with co-design architectures [3] show that several optimizations allow performance of the software to be improved. In this way, we estimate that an operative frequency of nearly 1.2 GHz is necessary to decode HDTV video with an optimized parsing algorithm allowing *quasi* real-time video decoding. Moreover, with a processor optimized for image processing (the aRDAC is a general purpose processor) it may be possible to decode HDTV video at a frequency under 800 MHz.

7. CONCLUSION

This paper has presented a method to implement an MPEG-4 video decoder on a heterogeneous platform composed of an ASIP and a FPGA. We have shown the interest of using a specific instruction for the parsing process and how we updated

the core (and the ISA) of the processor. We have also presented the modifications made on the memory architecture to improve the performance of the design. Experimental results show a 25% increase of FPS and a decrease of the logical use (nearly 10%) for an additional hardware cost of the processor as almost null. The linear increase of the FPS number with the clock frequency estimate shows that video can be decoded in real time with newest FPGAs (up to a CIF resolution) or ASICs (HD resolution). Therefore, benefits in term of performance compared to usual processor architectures and in term of flexibility compared to full hardware architectures are significant. What is more, processors are almost always used in designs, if only to configure hardware IPs or to process algorithms which do not require performance.

An interesting area for future research involves generating both the hardware and the software code with dedicated instructions using the RVC framework. Another possible area is the development and evaluation of a processor and a software code fully customized for video processing to evaluate the pros (area, performance, power efficiency) and cons (lack of flexibility and portability, programming difficulties) of such a core.

8. REFERENCES

[1] F. Pescador, M.J. Garrido, C. Sanz, E. Juarez, M.C. Rodriguez, and D. Samper, "A real-time H.264 MP decoder based on a DM642 DSP," in *Electronics, Circuits and Systems, 2007. ICECS 2007. 14th IEEE International Conference on*, 2007.

[2] R.C. Kordasiewicz and S. Shirani, "ASIC and FPGA implementations of H.264 DCT and quantization blocks," in *IEEE International Conference on Image Processing*, 2006, pp. III – 1020–3.

[3] Nicolas Siret, Ismaïl Sabry, Jean-François Nezan, and Mickaël Raulet, "A codesign synthesis from an MPEG-4 decoder dataflow description," in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1995–1998.

[4] Yedidya Hilewitz and Ruby B. Lee, "Fast bit gather, bit scatter and bit permutation instructions for commodity microprocessors," *J. Signal Process. Syst.*, vol. 53, pp. 145–169, November 2008.

[5] Mladen Berekovic, Hans-Joachim Stolberg, Mark B. Kulaczewski, Peter Pirsch, Henning Möller, Holger Runge, Johannes Kneip, and Benno Stabernack, "Instruction Set Extensions for MPEG-4 Video," *The Journal of VLSI Signal Processing*, vol. 23, pp. 27–49, 1999, 10.1023/A:1008188618930.

[6] Hyo-Jin Kim, "Bit stream parsing apparatus for audio decoder using normalizing and denormalizing barrel shifters," in *Patent US 5986588*, 1999.

[7] Y. Hilewitz and R.B. Lee, "A new basis for shifters in general-purpose processors for existing and advanced bit manipulations," *Computers, IEEE Transactions on*, vol. 58, no. 8, pp. 1035 –1048, Aug. 2009.

[8] J. Kneip, S. Bauer, J. Vollmer, B. Schmale, P. Kuhn, and M. Reissmann, "The MPEG-4 video coding standard-a VLSI point of view," in *Signal Processing Systems, 1998. SIPS 98. 1998 IEEE Workshop on*, 1998, pp. 43 –52.

[9] Yung-Chi Chang, Rlao-Chieh Chang, and Liang-Gee Chen, "Design and implementation of a bitstream parsing coprocessor for MPEG-4 video system-on-chip solution," in *VLSI Technology, Systems, and Applications, 2001. Proceedings of Technical Papers. 2001 International Symposium on*, 2001, pp. 188 –191.

[10] Marco Mattavelli, Ihab Amer, and Mickaël Raulet, "The Reconfigurable Video Coding Standard [Standards in a Nutshell]," *Signal Processing Magazine, IEEE*, vol. 27, no. 3, pp. 159 –167, May 2010.

[11] Matthieu Wipliez, Ghislain Roquier, and Jean-François Nezan, "Software Code Generation for the RVC-CAL Language," *Springer journal of Signal Processing Systems*, 2009.

[12] Jean-François Nezan, Mickaël Raulet, and Olivier Déforges, "Integration of MPEG-4 Video Tools onto Multi-DSP Architectures using AVSynDEx fast Prototyping Methodology," in *IEEE Workshop on Signal Processing Systems. SIPS*, 2002, pp. 207–212.

[13] ISO/IEC 14496-2: 2004, "Information technology - Coding of audio-visual objects - Part 2: Visual," 2004.

[14] Mladen Berekovic, Gerhard Meyer, Yong Guo, and Peter Pirsch, "Multimedia RISC core for efficient bitstream parsing and VLD," in *Multimedia Hardware Architectures 1998*, Sethuraman Panchanathan, Frans Sijstermans, and Subramania I. Sudharsanan, Eds. 1998, vol. 3311, pp. 131–141, SPIE.

[15] Timothy D. Anderson, Hoyle David, Donald E. Steiss, and Steven D. Krueger, "Microprocessor with non-aligned memory access," in *Patent US 6539467 B1*, 2003.

[16] Hoyle David, Joseph R. Zbiciak, and Jeremiah E. Golston, "Microprocessor with non-aligned scaled and unscaled addressing," in *Patent US 6539467 B1*, 2003.