

ASIP Solution for Implementation of H.264 Multi Resolution Motion Estimation

Fethi Tlili, Akram Ghorbel

CITRA'COM Research Laboratory, Engineering School of Communications(SUP'COM), Tunis, Tunisia

E-mail: fethi.tlili@supcom.rnu.tn

Received March 19, 2010; revised April 20, 2010; accepted May 15, 2010

Abstract

Motion estimation is the most important module in H.264 video encoding algorithm since it offer the best compression ratio compared to intra prediction and entropy encoding. However, using the allowed features for inter prediction such as variable block size matching, multi-reference frames and fractional pel search needs a lot of computation cycles. For this purpose, we propose in this paper an Application Specific Instruction-set Processor (ASIP) solution for implementing inter prediction. An exhaustive full and fractional pel combined with variable block size matching search are used. The solution, implemented in FPGA, offers both performance and flexibility to the user to reconfigure the search algorithm.

Keywords: Motion Estimation, Half Pel, Quarter Pel, ASIP

1. Introduction

The fast growth of digital transmission services has created a great interest in digital transmission of image and video signals. These signals require very high bit rates in order to guarantee good video quality. Therefore, compression is used to reduce the amount of data needed for representing such signals. Compression is achieved by exploiting spatial and temporal redundancies in signals [1].

H.264 video coding standard currently allows an approximately 2:1 advantage in terms of bandwidth savings over MPEG-2, and it has the potential to allow further bandwidth savings of 3:1 and beyond. In other words, an H.264 coded stream needs roughly half of bit-rates to provide the same quality got by an MPEG-2 encoder. It also includes a video coding layer, which efficiently represents the video content independently of the targeted application. A network adaptation layer which formats the video data and provides header information in a manner appropriate to a particular transport layer is used. Finally, in order to decrease the decoder complexity, several application-targeted profiles and levels are defined which enable its successful use in different video applications and markets [2].

Despite the fact that it has kept the same coding aspect as previous standards based mainly on prediction, transform and entropy encoding, H.264 has introduced some key feature modules that have increased considerably the

coding efficiency as well as more flexibility in most of the coding process.

However, H.264 is also a substantially more complex standard than MPEG-2; and both the H.264 encoders and decoders are much more demanding in terms of computations and memory than their MPEG-2 counterparts [3]. This, coupled with the substantial amount of research needed to properly implement and optimize the entire relevant H.264 features, makes the development of high-quality H.264 encoders a daunting task.

In addition to the complexity added by H.264 standard, low power consumption, high performance and scalability are the major constraints imposed to designers in the development of video encoders and decoders [4]. In fact, with the diversity of configurations supported by this standard in terms of resolutions and applications, scalable architectures for video encoders are much appreciated by service providers. In this context, neither hardware implementation solutions are efficient since they lack flexibility, nor software solutions present good performance since processors are no longer satisfying the high computational processing tasks [5].

To meet all these constraints, processor characteristics can be customized to match the application profile. Customization of a processor for a specific application holds the system cost down, which is particularly important for embedded consumer products manufactured in high volume. Application Specific Instruction set Processors (ASIPs) are in between custom hardware architectures

offering good processing performance and commercial programmable DSP processors with high programmability possibilities. They offer good programmability and performance level but are targeted to a certain class of applications as to limit the amount of hardware area and power needed [6].

This paper is organized as follows: Section 2 presents a complexity analysis of the different encoder's modules followed by the description of motion estimation standardized by H.264. In Section 3, we will present the proposed algorithm for multi resolution motion estimation. Section 4 presents the proposed ASIP solution. In Section 5 we will present implementation results. Finally, we enclose the paper by Section 6 in which we will conclude this work.

2. H.264 Video Encoder Study

2.1. Main Innovations of H.264

To achieve the required performance, H.264 allows some key features that ensure good coding efficiency. The main innovations of this standard are:

- Intra prediction process.
- Tree structured motion estimation, weighted prediction, multiple resolution search.
- Spatial in loop deblocking filter.
- Integer DCT like Transform.
- Efficient Macro Block Field Frame coding
- CABAC which provides a reduction in bit-rate from 5% to 15% over CAVLC.

2.2. Complexity Analysis of H.264 Video Encoder

In order to analyze the complexity of the H.264 encoding procedure, some profiling tasks were done on the several modules of the encoder mentioned above. For this reason, some implementations were performed on single chip DSP using CIF resolution in baseline profile to get the most accurate results since we have to avoid inter-chip communication that can bother the profiling results. **Figure 1** presents the profiling results of UBVideo encoder implemented on DM642 DSP of Texas Instruments [7].

We can see that the most consuming video tasks are motion search which is using about 30% of the processing time while the intra prediction, motion compensation and encoding (including transform, quantization and entropy encoding) are using only 23% of the system resources. Motion search includes only the best matching search while all load and store tasks are included in data transfer task which is using about 32% of system resources. The remaining 15% of the resources are used by other tasks such as rate control, video effect detection and bitstream formatting.

Hence, we can see that motion estimation is a bottle-neck for video encoding algorithms which is taking most of system resources. However, motion estimation is the most important module in the compression procedure due to its efficiency. In this context, some video encoders are using FPGA solutions for implementing motion estimators as hardware accelerators since DSPs cannot handle the processing required by such tasks.

3. Proposed Motion Estimation Implementation

3.1. H.264 Motion Estimation

Luminance component of each macro-block (16×16 samples) may be split up in 4 ways: 16×16 , 16×8 , 8×16 or 8×8 as shown in **Figure 2**. Each of the sub-divided regions corresponds to a macro-block partition. If the 8×8 mode is chosen, each of the four 8×8 macro-block partitions within the macro-block may be split in a further 4 ways: 8×8 , 8×4 , 4×8 or 4×4 as presented in **Figure 3**. Partitions and sub-partitions give rise to a large number of possible combinations within each macro-block. This method of partitioning macro-blocks into motion compensated sub-blocks of varying size is known as tree structured motion compensation.

In addition to the variable block size matching, H.264 defines multi resolution search process in order to provide better quality especially for non translational motion and aliasing caused by camera noise. Experimental analysis shows that the half and quarter-sample-accuracy

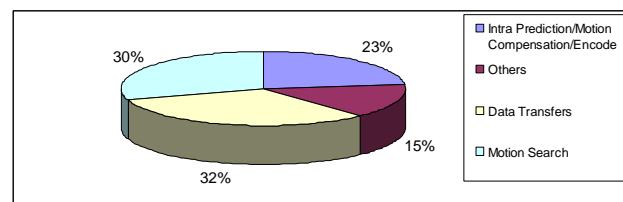


Figure 1. UbVideo encoder profile.

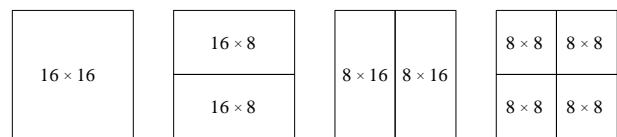


Figure 2. Macro-block partition.

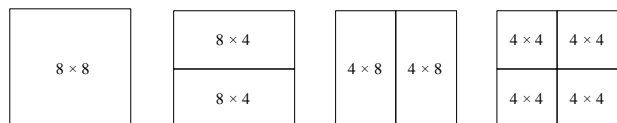


Figure 3. Macro-block sub partition.

motion search adopted by H.264/AVC provide a coding gain of 2 dB compared with MPEG-2 and H.263, which corresponds to a bit-rate savings of up to 30% [8]. Half pel search is performed on pixels interpolated using a 6 tap low pass filter. Furthermore, a quarter pel resolution search is established using a bi-linear filter applied on half pel interpolated pixels.

3.2. Proposed Motion Estimation Algorithm

The first step of the proposed ME algorithm consists in full pel resolution search. Current MB is searched in a predefined search area in the reference frame. In order to avoid unused computations and data load, the search is performed on 4×4 partitions base of the MB. For each 4×4 block, we search for the best matching position in the reference area. Every 4×4 block is independently parsed in all reference area. After that, a merging process is started in order to determine the best partition to be used for the current MB based on the best position which is stored relative to the top left pixel of the 4×4 block. The merging process is first used to determine if the current MB can be coded in partitions above than 4×4 . So, we compare the best positions of adjacent blocks for all 8×8 partitions: if all blocks have the same best position, current sub partition is 8×8 , otherwise, it could be 8×4 , 4×8 or 4×4 . If 8×8 mode is selected, a best position of the top left pixel is stored.

After that, we determine the MB prediction type that can be 16×16 , 16×8 , 8×16 or 8×8 . A merging process similar to the previous one is also used: if all 8×8 sub partitions have the same type and the same best position, MB prediction type is 16×16 ; otherwise it could be 16×8 , 8×16 or 8×8 . After fixing the MB prediction type, a motion vector is stored for each partition. Obviously, the more we use sub partitions, more data to be transferred increases. We note that at least 40% of inter prediction data is used to code motion vectors. For this reason, it is better to use bigger partitions when possible. So, a prediction cost can be added by making conditions for the merge process based on tolerance of one or two pixels in the best positions: for example, if two 8×8 blocks have the best positions displaced of 1 pixel, we can decide to merge them into one 16×8 partition.

After searching for the best matching and the best partition, we start fractional pel search. According to the best position, for each MB partition we interpolate the possible 8 half pixels positions around the selected partition as shown in **Figure 4**. The interpolation is equivalent to an up-sampling of the frame pixels using 6 tap low pass filter.

After that, a further search is performed in quarter pel accuracy using another interpolation process. Based on the best position obtained in half pel search, we generate pixels of all the 8 possible positions around the best loca-

tion. We note that motion vectors are multiplied by 4 in order to mention to the decoder if it has to interpolate pixels for motion compensation or not.

4. Proposed ASIP Solution

4.1. Analysis of the Proposed Motion Estimation Algorithm

In our work, we will adopt instruction selection methodology based on hardware architecture: first the hardware architecture is fixed containing selected functional units (FU) and then, instruction set architecture is determined according to the FUs. For this purpose, proposed algorithm is analyzed in order to pick up the most complex modules. These modules will be implemented in independent hardware blocks (dedicated FUs).

Proposed algorithm is composed mainly of 3 parts: full pel search, half pel interpolation and its associated search and finally quarter pel search with its final search.

In full pel search, the MB parses the whole reference area and 4×4 SADs are computed. In this step, the most complex process is the SAD computation since it includes difference computation, absolute value determination and accumulation. In [9], an analysis was performed on a motion estimation algorithm using SAD as a distortion measure; we found that SAD computation is using more than 97% of system resources.

In addition, sub pel motion estimation is also complex. In fact, the interpolation process for half pel is using 6-tap filter. Half samples are calculated through a 6-tap Wiener filter in both horizontal and vertical dimensions. The interpolation is processed as represented in **Figure 5**: dashed pixels correspond to full pixels in an 8×8 bloc. Non dashed pixels are half pixels that are calculated. For example, to interpolate half pixel 'b', we use E, F, G, H, I and J as full pixels. Calculation process is done as follows: $b = \text{Clip1}(((E - 5 \times F + 20 \times G + 20 \times H - 5 \times I + J) + 16) \gg 5)$; clip function is used to provide result in the interval $[0, 255]$: if result is less than 0 we affect 0 to 'b' and if it is more than 255 we affect 255 to 'b'. The same calculation process is done for vertical rows as 'h'.

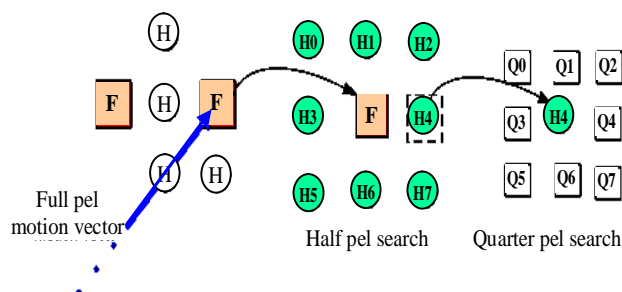


Figure 4. Fractional accuracy pixel search.

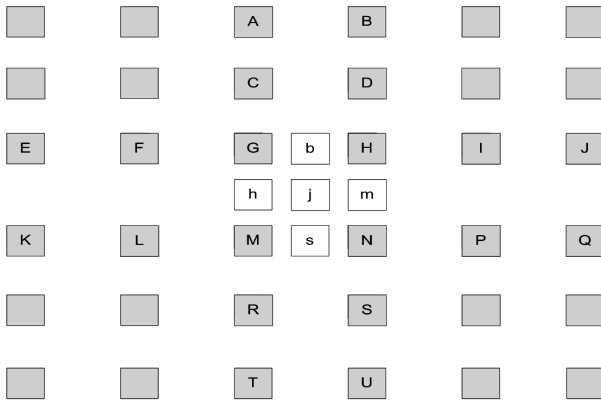


Figure 5. Half pel interpolation process.

Hence, half pel interpolation, as any filtering process is a very time consuming task and needs a lot of data load and store. Similarly, quarter pel interpolation is using bilinear filter to generate quarter pixels. Although the simplicity of the filter, this process also needs a lot of timing since it is applied to a large number of data.

In conclusion, the main complex modules in our proposed algorithm are the motion search, half pel interpolation and quarter pel interpolation. In our architecture, we will use hardware accelerators for these modules for better performance for our ASIP.

4.2. Functional Unit Selection

In our proposal, 3 hardware accelerators are used: SAD calculator, half pel interpolator and quarter pel interpolator.

The SAD calculator will be used to handle all SAD computation process including data load from internal memory and SAD calculation. The result is stored in a general purpose register.

Half pel interpolator module is used to interpolate half pixels according to the standardized filter. This module loads data from internal memory and interpolates pixels. Due to the complexity of interpolation, half pixels are stored in an internal memory to be used in further possessing tasks such as quarter pel interpolation or even half pixels. Finally, quarter pel interpolator loads data from internal memory and applies bilinear filter to generate quarter pixels. In order to avoid storing quarter pixels in memory, a SAD calculator is integrated in this module: reference pixels are loaded and quarter pel resolution SAD is computed. In motion compensation process, these pixels are re-computed since their computation is not as complex as half pixels.

In addition to the hardware accelerators for video processing, an Arithmetic and Logic Unit is used in the solution in order to accumulate SADs, generate pixel locations and memory addresses.

4.3. Instruction Set Selection

4.3.1. Video Instructions

- **SAD4Pix(DestReg,Curr_Pix_Addr,Ref_Pix_Addr,Pitch):** this instruction is used to compute SAD of 4 pixels based on current and reference pixel location and Pitch value. The choice of the 4 pixels size is based on the fact that the smallest partition allowed is 4×4 ; so to avoid using SAD instructions for all partitions, we call this instruction as much as the current partition contains 4 pixel lines. Since we adopt RISC (Reduced instruction Set Computer) architecture, current and reference pixel locations as well as Pitch value are stored in Special Purpose Registers (SPR). These registers are used only for video instructions since they need more than 2 input operands. Output of this instruction is stored in a General Purpose Register (GPR), DestReg in order to be accumulated to constitute the required SAD. The choice of the SAD computation size offers the flexibility to the user to choose block lines to be compared. In fact, we can compute only some specific lines in order to minimize the processing (for example odd lines or even lines).

- **Interp4HafPix(RefPixAddr,Pitch):** interpolates 4 half pixels and stores the result in internal memory. Input operands include the reference pixel address which refers to the first full pixel from which we start interpolation and a pitch value that is used for data load in case of vertical interpolation. This value is used to give the programmer the flexibility of modifying the search window size. These operands are loaded from SPRs while output interpolated pixels are stored in half pel memory since there is no need to store them in registers. In our motion estimation algorithm, after calling this instruction to interpolate half pixels of 1 MB, SAD4Pix instruction can be called in order to compute SAD in half pel resolution. For this reason, the pitch value is used in this instruction since the loading step in half pel memory is equal to 2. Hence, we avoid the use of 2 SAD instructions (one for full pel SAD and the other for half pel SAD).

- **Interp4QpixSAD(DestReg,Ref_pix,Curr_pix,Pitch):** used to interpolate 4 quarter pixels and compute quarter pel resolution SAD. We have chosen to separate half pel interpolation from quarter pel interpolation in order to give the user the flexibility to stop the search at any resolution according to the complexity of the algorithm. However, quarter pels are not stored and the corresponding SAD is immediately computed. In fact, quarter pels are no longer used by the system except the best match that is used for motion compensation where the best matching pixels are used. So, to avoid using huge memory size corresponding to store all interpolated pixels, we made the choice not to store them and to recompute the best matching pixels when required in motion compensation since their re-computation is easy as op-

posed to half pels. This instruction returns the SAD of the current position and the ALU decides for the best one to be used in motion compensation. Input operands to this instruction, reference and current pixels positions as well as pitch value are stored in SPRs. The output is stored in GPR, DestReg to be processed by the ALU for further decisions.

4.3.2. Memory Instructions

Memory instructions are used to transfer data between memory and registers or inter register transfer. Four instructions are used for this purpose:

- **MOVSG(Src, Dest)** is used to move data from specific to general purpose register. The operands of this instruction are formed by the addresses of registers to be manipulated.
- **MOVGS(Src, Dest)** is used to perform the inverse operation performed by MOVSG.
- **LOAD(SrcAddr, DestReg)** is used to load data from data memory to general purpose register. SrcAddr is the source address of data to be loaded while DestReg in the destination register ID.
- **STORE(SrcReg, DestAddr)** is used to store the content of a general purpose register in memory. The operands are SrcReg corresponding to the source register ID and DestAddr is the destination memory address.

4.3.3. Arithmetic and Logic Instructions

The main goal of these instructions is the accumulation of SAD values computed for each 4 pixels, computing pixel addresses, compare MB SADs and provide data for conditional jump. ALU instructions are processing only data from general purpose registers. We defined 3 arithmetic instructions:

- **ADD, SUB and MUL** are used respectively for addition, subtraction and multiplication operations. These instructions have 3 operands: the first one is the destination register ID containing the operation result while the 2 remaining operands are the IDs of registers containing source data to be processed.
- **SHIFT(SrcReg1, SrcReg2, SrcReg3)** is used for shifting data contained in SrcReg1 by the number of bits contained in SrcReg2. The shift direction is indicated by SrcReg3.

4.3.4. Control Instruction

The instruction JUMP introduces a change in the control flow of a program by updating the program counter with an immediate value that corresponds to an effective address. The instruction has 2 bits condition field (cc) that specifies the condition that must be verified for the jump: in if case the outcome of the last executed arithmetic is negative, positive or zero. Not only this instruction is important for algorithmic purposes, but also for improving code density, since it allows a minimization of the number of instructions required to implement a ME al-

gorithm and therefore a reduction of the required capacity of the program memory.

4.4. Architecture of the Proposed ASIP

4.4.1. Data Word Length

Data word length is a tradeoff between performance and complexity. In fact, the data word length corresponds to the instruction word length which is stored and manipulated by the processor. Hence, in case of longer instruction word length, we have the possibility of using more instructions and more registers which will accelerate the processing since memory access will be reduced. However, the instruction decoder will be more complex as well as the interconnection between components; therefore, the processor area will be larger.

In our proposal, we have only 12 instructions which can be coded on 4 bits. In order to simplify the hardware architecture, we have chosen to use 16 bits to code all instructions. So, 12 bits can be used to address the register file.

4.4.2. Register File Size

Since the instruction length is 16 bits and 4 bits are used to code instructions, the 12 remaining are used to code the different registers used. Since arithmetic instructions are using 3 GPPs, we will code each register on 4 bits, so 16 GPPs can be used in our architecture. On the other side, video instructions are using both GPPs and SPPs. So, 8 bits only can be used to code 3 registers in the instruction call: each register is addressed on 2 bits. So, 4 SPPs are used. At this stage, we can see the importance of the use of GPPs and SPPs: if we use only one register type, when calling video instruction, 12 bits are used to code 4 registers: 3 bits are used per register as a consequence. Therefore, only 8 registers are used in this case while in our design we are using 20 registers with the same instruction length. **Table 1** presents the different

Table 1. Instruction set architecture of the proposed ASIP.

Instruction	15	12	11	10	9	8	7	6	5	4	3	2	1	0
SAD4Pix	0000		RestReg		R1		R2		R3		-			
Interp4HafPix	0001	-				R1		R2		-				
Interp4QpixSAD	0010		DestReg		R1		R2		R3		-			
MOVSG	0010	-						Src				DestReg		
MOVGS	0011		SreReg										Dest	
LOAD	0100				#addr							DestReg		
STOR	0101		SreReg						#addr					
ADD	0110		DestReg			SreReg1					SreReg2			
SUB	0111		DestReg			SreReg1					SreReg2			
MUL	1000		DestReg			SreReg1					SreReg2			
SHIFT	1001		SreReg1			SreReg2					SreReg3			
JUMP	1010		CC						#addr					

instructions with the corresponding codes, operands with their corresponding size.

4.4.3. Micro Architecture

Figure 6 presents the micro architecture of the proposed ASIP.

The solution is composed of an instruction fetch module to load instructions from program memory, instruction decoder to enable the several functional units and a register file to store processed data. Video functional units are connected to the internal data memory and the ALU. Data load from external memory to internal memory is handled by a direct memory access controller.

5. Implementation Solution and Results

The proposed ASIP was implemented and synthesized on Virtex II Pro FPGA.

5.1. Memory Management

In our motion estimation algorithm, the search region area is fixed to 31×23 pixels. We note that we need to extend this search region by 16 pixels in both sides (right and bottom) since the last right-bottom position must be displaced of a (15, 12) vector from the centre. Furthermore, to interpolate boundary pixels, an extension of three pixels is needed for each side. **Figure 7** describes the search area with the several extensions.

Hence, the total search area has to be 53×45 ; so 2385 pixels have to be loaded from external to internal mem-

ory. Internal memory is designed to be 2×18 Kb block RAM integrated in Virtex II FPGA. We note also that a further 1×18 Kb block RAM is also needed to store the current MB. Internal memory is 8 bits width for implementation constraints: since we adopt exhaustive search, the whole reference area is parsed in order to search for the best matching MB; so, if we load more than one pixel from reference area, we will be faced to an alignment problem. To avoid such problems, we have chosen to load one pixel in each cycle assuming that this procedure is more consuming in time. Data load to internal memory is ensured by Direct Memory Access controllers which handles the transfer process while the CPU is running. When transfer is finished, an interrupt signal is mentioned.

Synthesis results of the DMA controller shown in **Table 2** presents that this module using roughly 10% of the available FPGA resources and can be run at 205 Mhz clock frequency.

5.2. SAD Engine

This engine is used to compute the SAD of 4 pixels. This module loads reference and current pixels from the internal memory and performs the SAD of 4 pixels in one call. The SAD module can be used in the SAD computation of the full pel or half pel search.

As described in **Figure 8**, the SAD engine is providing the output after 9 cycles from the start signal. The output is finally returned to the register file. We note that TMS320C64 DSP is providing SAD of 4×4 blocks

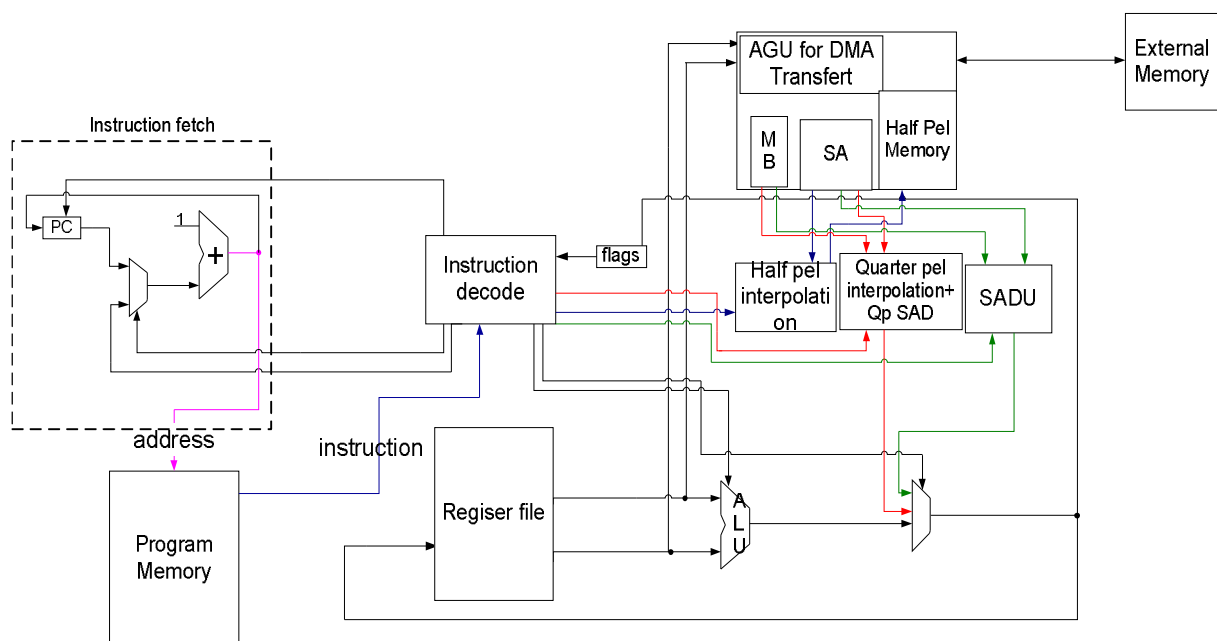


Figure 6. Architecture of the proposed ASIP.

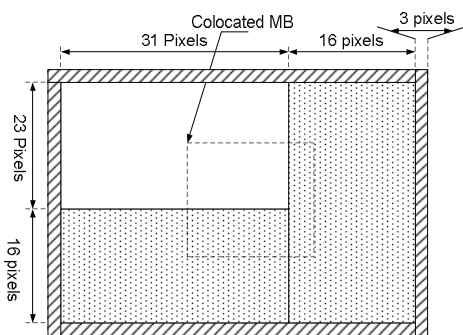


Figure 7. Search area organization.

Table 2. Synthesis results of DMA controller.

Device utilization summary	
Number of Slices	190 out of 1408 13%
Number of Slices Flip Flops	178 out of 2816 6%
Number of 4 input LUTs:	300 out of 2816 10%
Number of GCLKs	1 out of 16 6%
Timing Summary:	
Minimum period/Maximum Frequency	4.877 ns/205.025 MHz
Minimum input arrival time before clock	5.294 ns
Maximum output required time after clock	4.968 ns
Maximum combinational path delay	No path found

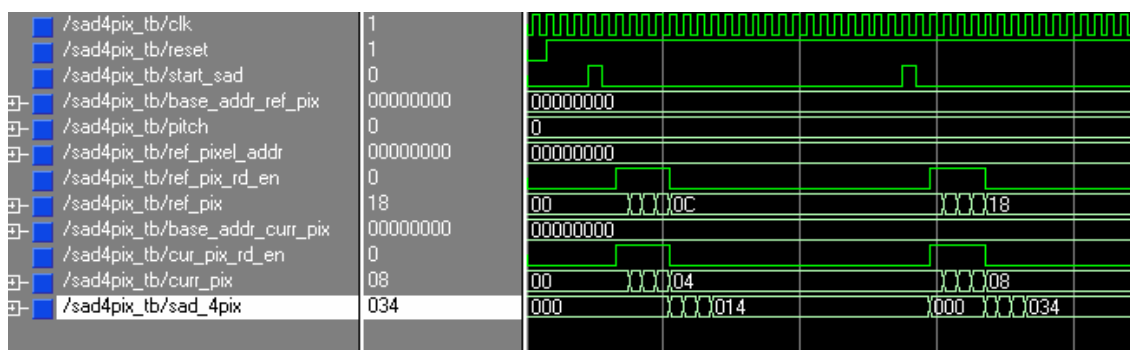


Figure 8. Timing diagram of SAD engine.

(split_sad8 × 8) in 200 cycles in the best case: when all data paths are fully used [10] while our system can provide the same result after 144 cycles without using pipeline.

5.3. Half Pel Interpolator

In our implementation, the proposed algorithm is derived by minimizing the number of memory access. The formulas to compute half-pixel interpolations are proposed by using the symmetry of the 6-tap FIR filter coefficients, resulting in significant reduction of the multiplications [11].

This engine is providing 4 interpolated pixels in each call. Input pixels are stored in 6 registers; the size of each one is 32 bits as described in Figure 9:

We note that pixels P3 to P6 form a line of a selected 4 × 4 block to be interpolated. The output pixels are H0 to H3. A Single Instruction Multiple Data scheme is adopted in our implementation. In this mode, adders and multipliers are applied simultaneously to the pixels of registers in order to get all interpolated pixels at the same time. All control signals are provided by an FSM.

We note that the interpolation takes 15 cycles including the load process from internal memory. Synthesis results are shown in Table 3.

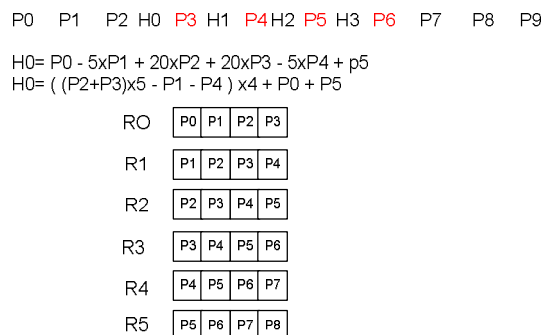


Figure 9. Input registers for halfpel interpolation.

Table 3. Synthesis results of half pel interpolator.

Device utilization summary	
Number of Slices	354 out of 1408 25%
Number of Slices Flip Flops	460 out of 2816 16%
Number of 4 input LUTs:	343 out of 2816 12%
Number of MULT18X18s	4 out of 12 33%
Number of GCLKs	1 out of 16 6%
Timing Summary:	
Minimum period/Maximum Frequency	5.504 ns/181.689 MHz
Minimum input arrival time before clock	4.679 ns
Maximum output required time after clock	3.638 ns
Maximum combinational path delay	3.802 ns

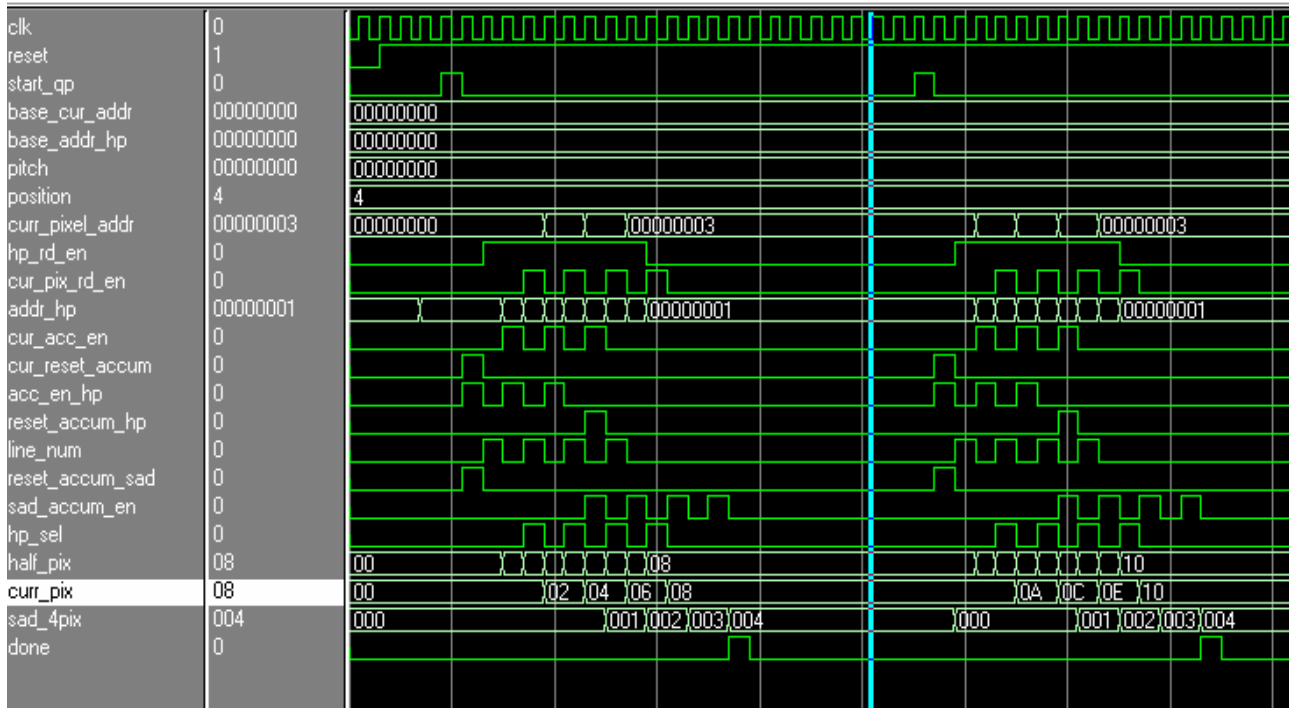


Figure 10. Timing diagram of Quarter pel interpolator.

5.4. Quarter Pel Interpolator

When receiving `Interp4QpixSAD(Ref_pix,Curr_pix,Pitch)` instruction, quarter pel interpolation and SAD computation are started. First, pixels loaded from half pel memory are fed into the interpolator module, then, the resulting quarter pixels are transmitted to the SAD module to be compared to the current pixels. We note that QP interpolator interpolates and generates the SAD of 4 pixels in each call.

Quarter pel SADs are returned after 14 cycles as shown in the timing diagram shown in **Figure 10**.

6. Conclusions

This paper has presented efficient instructions for implementing motion estimation process using most of the key features standardized in H.264. First, we analyzed the complexity of typical H.264 encoder. From this step, we concluded that ME is a bottle neck for the implementation. Then, we presented and analyzed an algorithm for ME. Based on the analysis, we proposed efficient accelerators for some modules which need most of the processing time. Based on the suggested hardware architecture, we fixed the instruction set architecture providing to users large coding flexibility ensuring scalability and multi-standard support. Proposed ASIP was implemented

on Virtex II pro FPGA with a total area use about 61% of the FPGA Slices and 43% of the total LUTs. The implemented modules can be run on 172 MHz clock.

7. References

- [1] Q. Y. Shi and H. F. Sun, "Image and Video Compression for Multimedia Engineering: Fundamentals, Algorithms, and Standards," 2nd Edition, CRC Press, Boca Raton, 2008.
- [2] Draft 3rd Edition of ISO/IEC 14496-10 (E), Redmond, WA, USA, July 2004.
- [3] F. Kossentini and A. Jerbi, "Exploring the Full Potential of H.264," *NAB*, 2007.
- [4] S. D. Kim, J. H. Lee, C. J. Hyun and M. H. Sunwoo, "ASIP Approach for Implementation of H.264/AVC," *Journal of Signal Processing Systems*, Vol. 50, No. 1, 2008, pp. 53-67.
- [5] P. Harm, *et al.*, "Application Specific Instruction-Set Processor Template for Motion Estimation in Video Applications," *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 15, No. 4, April 2005, pp. 508-527.
- [6] M. Kumar, M. Balakrishnan and A. Kumar, "ASIP Design Methodologies: Survey and Issues," *14th International Conference on VLSI Design*, Bangalore, 2001.
- [7] I. Werda and F. Kossentini, "Analysis and Optimization of UB Video's H.264 Baseline Encoder if Texas Instru-

- ment's TMS320DM642 DSP," *IEEE International Conference on Image Processing*, Atlanta, October 2006.
- [8] S. Yang, *et al.*, "A VLSI Architecture for Motion Compensation Interpolation in H.264/AVC," *6th International Conference on ASIC*, shanghai, 2005.
- [9] W. Geurts, *et al.*, "Design of Application-Specific Instruction-Set Processors for Multi-Media, Using a Retargetable Compilation Flow," *Proceedings of Global Signal Processing (GSPx) Conference, Target Compiler Technologies*, Santa Clara, 2005.
- [10] M. A. Benayed, A. Samet and N. Masmoudi, "SAD Implementation and Optimization for H.264/AVC Encoder on TMS320C64 DSP," *4th International Conference on Sciences of Electronic, Technologies of Information and Telecommunications (SETIT 2007)*, Tunisia, 25-29 March 2007.
- [11] C.-B. Sohn and H.-J. Cho, "An Efficient SIMD-based Quarter-Pixel Interpolation Method for H.264/AVC," *International Journal of Computer Science and Security*, Vol. 6, No. 11, November 2006, pp.85-89.