IEEE Int. Conference Neural Networks & Signal Processing
Zhenjiang, China, June 8~10, 2008

AVS VIDEO STANDARD IMPLEMENTATION FOR SOC DESIGN

Xin Jin[*1,2], *Songnan Li*[1] *and King Ngi Ngan*[1]
[1]Dept. of Electronic Engineering, The Chinese University of Hong Kong, Hong Kong, China
[2]Dept. of Electronics and Information, Huazhong Univ. of Sci. and Technol., Wuhan, China 430074
{xjin, snli, knngan}@ee.cuhk.edu.hk

ABSTRACT

AVS1-P2 is the newest video standard of Audio Video coding Standard (AVS) workgroup of China, which provides close performance to H.264/AVC main profile with lower complexity. In this paper, a platform independent software package is developed for AVS1-P2 decoder to facilitate embedded video codec development. In order to minimize the on-chip memory and save the time consumed in on-chip/off-chip data transfer, an MB-based architecture is developed by modifying the data flow, decoding hierarchy, and the buffer definition and management for low-level decoding kernels. Such system architecture provides over 80% reduction in on-chip memory compared to the frame-based architecture when decoding 720p (1280×720) sequences. By modularizing the decoding kernels and data transfer modules, the proposed MB-based decoder facilitates the AVS video decoder development on the target platform.

Key Words —— AVS video standard, MB-based architecture, video coding

1. INTRODUCTION

AVS1-P2 is the newest video coding standard of Audio Video coding Standard (AVS) workgroup of china, which was initiated by the government of China in 2003 and was approved to be the national standard GB/T 200090.2-2006 in year 2006 [1]. AVS1-P2 is based on the hybrid coding framework including 8×8 block-based prediction, integer transform and entropy coding techniques. It aims for applications as digital broadcast, high-density laser-digital storage media, and so on. Currently, it is being applied in IPTV and China Mobile Multimedia Broadcasting (CMMB). AVS promises improved performance close to H.264/AVC main profile with lower complexity [2].

AVS1-P2 is designed based on hierarchical bitstream structure from sequence header, picture header, down to slice data and macroblock (MB) data. The diagram of AVS1-P2 is depicted in Fig. 1. As shown in the figure, each input video frame is first spilt into 16×16 MBs. Then, 8×8 block-based Intra prediction and Inter prediction are

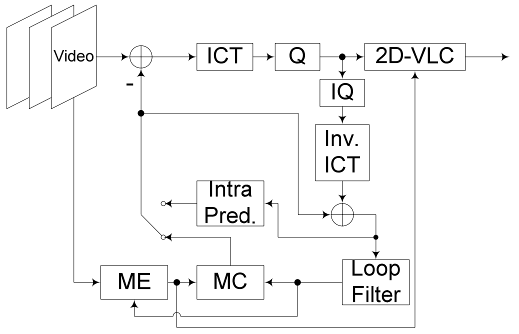

Fig. 1. AVS1-P2 encoding diagram.

performed. For Intra prediction, five 8×8 directional modes are defined for luma and four modes are defined for chroma. For Inter prediction, four partition modes listed as 16×16, 16×8, 8×16 and 8×8 are enabled for motion estimation (ME) and motion compensation (MC) together with the maximum two reference frames and quarter-pixel interpolation accuracy. The residuals between the original image and the predicted image are coded using an 8×8 integer transform and quantization. Adaptive two-dimensional variable length code (2D-VLC) is used to generate the coded bit-stream for 8×8 level-run pair. The blocking artifacts caused by 8×8 block-based transform and quantization is filtered by in-loop de-blocking filter to improve the subjective quality of the output and the objective quality of ME and MC.

Being a national video standard, AVS1-P2 has already drawn great attention from the industry. Generally, there are two categories of video decoder implementations: one is the decoder on PC platform for realtime video playback; the other is the system-on-chip (SoC) design for embedded video codec used in consumer electronic devices. For the first category of implementation, decoder with frame-based architecture, similar to that used in the reference decoder [3], is commonly used to fully exploit the large memory resources on PC to pursue fast decoding speed. All of the reference information is saved for the whole frame; Intra prediction and deblocking filter are directly performed in the reconstruction frame; MC is performed directly in the reference frames. While, for the second category of implementation, the frame-based architecture can not be used because that the size of on-chip memory on the chip is too limited to save those information, and data transfer between the on-chip memory and the off-chip memory is time consuming [4]. However, the decoder implementation

[*] This work has been done while the author is with the Chinese University of Hong Kong as a postdoctoral fellow.

for SoC design is very important for AVS, since it can save a large amount of royalty cost for Chinese multimedia industry. Consequently, in this paper, a platform independent software package of AVS1-P2 decoder is developed for SoC design. Different from the frame-based architecture designed for implementations on PC platform, an MB-based decoding architecture is implemented to minimize the requirement of on-chip memory, to modularize on-chip and off-chip processing, and to facilitate embedded video codec implementation on target platform. Compared to the frame-based architecture implementations, the MB-based decoder reduces over 80% on-chip memory requirement as decoding 720p (1280×720) sequence.

The rest of this paper is organized as follows. The data flow in our MB-based decoder together with its architecture is first presented in Section 2. The detailed implementation methods including the interface definition and update processing are secondly presented in Section 3 for four major kernels, namely, buffer update, Intra prediction, motion compensation, and deblocking filter. The storage complexity is analyzed in Section 4 with the comparison to the frame-based implementation. Section 5 concludes the paper.

2. DECODER DATA FLOW AND SYSTEM ARCHITECTURE

2.1. Decoder data flow

Fig. 2 depicts the data flow used in the MB-based decoder. Since this decoder is developed for SoC design, the data flow of the software package simulates the data communication in the target system. The compressed bit-stream file is stored on the hard disk. After the file is loaded to the off-chip memory (as SDRAM on PC), the decoder loads the bit-stream piece by piece from the off-chip memory to the on-chip bit-stream buffer via the dedicated on-chip/off-chip data transfer routines. After decoding, the reconstructed MBs are written from on-chip to off-chip to form the reconstructed frame. Via the file input/output interface, the reconstructed frame is written to hard disk to save as decoded sequence.

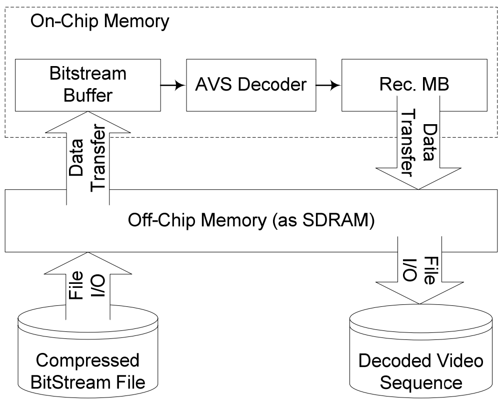

Fig. 2. MB-based AVS decoder data flow.

2.2. MB-based architecture

Based on the above data flow, the architecture of the MB-based decoder is depicted in Fig. 3.

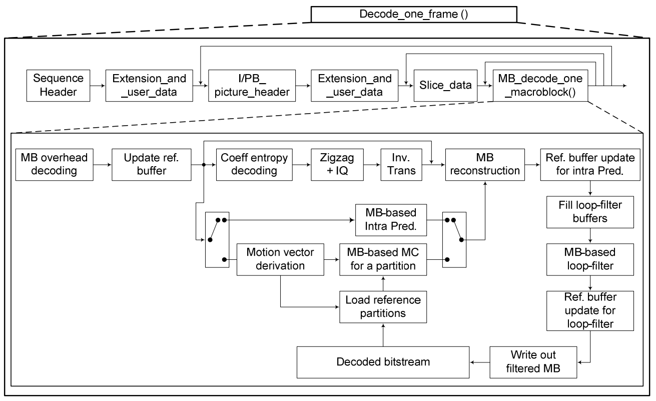

Fig. 3. MB-based AVS decoder diagram and code partition.

As shown in the figure, the decoder is built on two hierarchies. In order to decode a frame, the decoder first calls the functions to decode the high level syntax in the bitstream like sequence header (if it is the first frame), picture header, slice data and etc. As it starts to decode slice data, the decoder triggers the function MB_decode_one _macroblock() repeatedly until the entire slice of data is decoded. The MB_decode_one_macroblock() is the major decoding function for an MB which consists of several low-level kernels like macroblock overhead decoding, MB-based buffer updating, entropy decoding, Intra prediction, loop-filter, MC and etc.. All of these kernels are constructed for processing a macroblock of data at a time. There are dedicated routines for data transfer between on-chip and off-chip (e.g. loading the bitstream and reference partitions, and writing out the reconstructed MBs).

In the following section, the implementation of four major MB-based modules denoted as MB-based buffer update, Intra prediction, MC and deblocking filter will be described detailedly with the emphasis on buffer definition, data management and update process, while the decoding algorithm of Intra prediction, MC and deblocking filter will not be described since they are the same with that defined in the standard [1].

3. KEY KERNELS IMPLEMENTATION

3.1. MB-based buffer update

In order to minimize the on-chip memory, most of the low-level decoding kernels are processed in the MB-based buffers which save the information only needed for decoding the current MB. Before decoding the next MB, these buffers need to be updated by saving the just decoded MB's information for following decoding and by replacing the neighboring information for the next MB. This buffer update process is denoted as "Update ref. Buffer" in Fig. 3. Here, four sets of decoding information buffers defined for motion vector (MV), reference index (RI), quantization

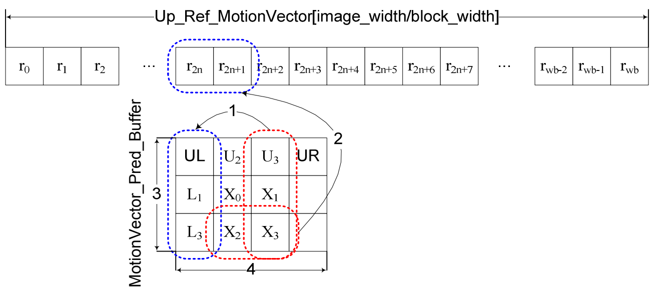

Fig. 4. Buffer definition and update for motion vectors.

parameter (QP) and luma Intra prediction mode (LIPM) are introduced.

Using the buffers for MV as an instance, Fig. 4 illustrates the buffer definition and update method. *Up_Ref_MotionVector* is the reference buffer saving all of the decoded MVs of the whole block line above the current MB. *MotionVector_Pred_Buffer* is the working buffer for MV-related decision, of which X_0 to X_3 save MV of each block of the current MB, others save prior decoded MVs of spatially neighboring blocks. If the neighboring MB has partition type other than 8×8, the MV of larger partition will be duplicated to each 8×8 block for saving. Before decoding the next MB, MB numbered as n+1, these two buffers need to be updated to save the just decoded MVs of MB n, and to refresh the neighboring MVs for MB n+1. The update consists of three steps: first, the MVs in U_3, X_1 and X_3 are copied to the left of the current MB as indicated by arrow 1 to update the left reference MV; then, MVs in r_{2n} and r_{2n+1} in *Up_Ref_MotionVector* are replaced by values in X_2 and X_3 for decoding the MVs of the vertically adjacent MB; thirdly, MVs in r_{2n+2}, r_{2n+3} and r_{2n+4} in *Up_Ref_Motion Vector* are copied to U_2, U_3 and *UR* above the current blocks in *MotionVector_Pred_Buffer*. Using such buffer update, the MVs saved in the working buffer are ready for decoding MB n+1, and the MVs in *Up_Ref_MotionVector* are automatically updated for decoding the next MB line.

Similar to motion vector, the reference index uses the same size of reference buffer and working buffer. The working buffer for QP is only 2×2, since each MB has only one QP value and the QP of the MB above-right of the current MB is not needed. The size of reference QP buffer is image width divided by the width of an MB. For the buffers used to save LIPM, 3×3 working buffer and image_width/ block_width+2 reference buffer are used. Values saved in the first and the last elements of reference buffer are -1, which represent the image boundary. The updating methods for all of these buffers are similar to that of MV.

3.2. MB-based Intra prediction

Different from the frame-based implementation that performs Intra prediction directly in the reconstructed frame, the MB-based luma Intra prediction is implemented in the three buffers as shown in Fig.5. The 16×16 buffer with bias is a common buffer for decoding the current MB. It saves the predicted samples during Intra prediction or Inter

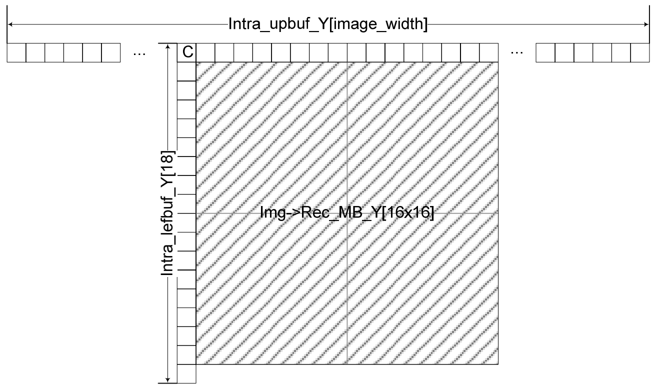

Fig. 5. Luma Intra prediction buffer definition.

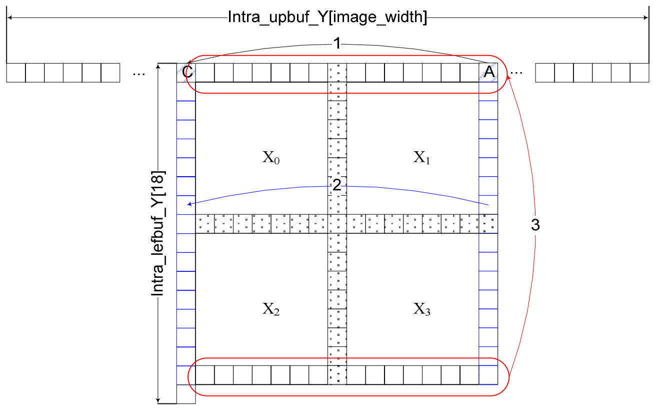

Fig. 6. Reference pixel allocation and update for luma Intra prediction.

prediction and updates them to be reconstructed samples by adding the prediction value with the reconstructed residue after inverse quantization and inverse transform. *Intra_upbuf_Y* saves the whole line of decoded samples above the current MB. *Intra_lefbuf_Y* saves the decoded samples left to the current MB, of which the first element, denoted as *C*, is the sample spatially above-left to the current MB, and the 17th element *Intra_lefbuf_Y*[17] equals to the 16th.

The buffer access during luma Intra prediction and buffer update method are interpreted in Fig.6. As depicted in the figure, current intra-coded MB is divided into four 8×8 blocks X_0 to X_3, which will be decoded sequentially. Since each intra-coded 8×8 block needs to use prior decoded samples in adjacent blocks to generate prediction block, different blocks uses samples in different buffers as its reference. For X_0, samples in *Intra_upbuf_Y* and *Intra_ lefbuf_Y* are its reference samples; for X_1, the right column of samples in X_0 (8 dotted samples) and *Intra_upbuf_Y* are its reference; for X_2, the samples in *Intra_lefbuf_Y* and the bottom row in X_0 (8 dotted samples) are its reference; for X_3, the dotted right column in X_2 and the dotted bottom row in X_1 are its reference.

After finishing decoding the whole MB, no matter the MB is intra-coded or inter-coded, the pixel buffers will be updated as: copy pixel *A* in *Intra_upbuf_Y* to position *C* in *Intra_lefbuf_Y* following arrow 1; copy right column of samples of current reconstructed MB (16 samples in blue) to the corresponding position in *Intra_lefbuf_Y* as indicated by arrow 2; thirdly, copy the bottom row of samples (16 samples in red circle) from the current MB buffer to the corresponding position in *Intra_upbuf_Y* following arrow 3. Using such mechanism, *Intra_lefbuf_Y* is ready for decoding the next MB, and *Intra_upbuf_Y* is automatically ready for decoding the next MB line after the current MB line is decoded.

The buffer update for chroma (4:2:0) Intra prediction is almost the same with luma Intra prediction. The only difference is the buffer size defined for it. The size of the buffer saving the samples above the current MB is half of the image width. The size of the buffer saving the samples left to the current MB is 10.

3.3. MB-based motion compensation

Motion compensation in the decoder is used to retrieve inter predicted partition according to the motion information. For the frame-based implementation, interpolation is directly performed in the reference frame. However, the size of an entire reference frame is too large for an embedded system to load into the on-chip memory. Consequently, the MB-based MC is working as depicted in Fig. 7.

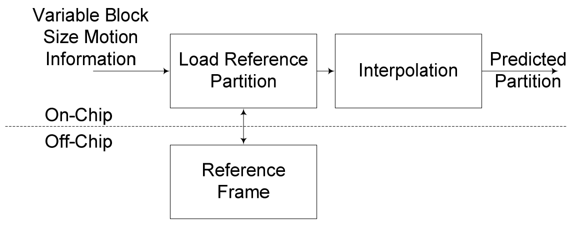

Fig. 7. MB-based motion compensation flow.

As shown in the figure, MC first retrieves the reference partition with integer-pixel accuracy from the reference frame according to the motion information and loads it to the on-chip memory via data transfer routines; then the interpolation is performed in on-chip memory to generate predicted partition with higher accuracy. Since the interpolation algorithm is the same with that described in the standard, only the reference partition loading method will be introduced in the following.

Still using luma as an instance, how many integer samples need to be loaded from the reference frame is determined by the partition size and where the motion vector points to. The size of partition can be one of four partition sizes defined as 16×16, 16×8, 8×16 and 8×8. While, the pointed position of motion vector is more complex. In case the motion vector points to an integer-sample position (denoted as *A* in Fig. 8), the same number of integer samples as the partition size needs to be loaded into the on-chip memory; otherwise, a larger number of integer samples needs to be loaded to generate those noninteger positions. Table 1 lists the number of integer samples that needs to be loaded into the on-chip memory according to the MV positions depicted in Fig. 8.

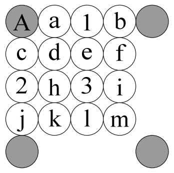

Fig. 8. Interpolation positions: gray samples are samples at integer-position; samples numbering 1 to 3 are samples at half-sample position; others are samples at quarter-sample position.

TABLE 1 The number of integer reference samples according to different MV positions.

MV position	The number of integer samples (W×H)	
	W (Width)	H (Height)
A	partition_width	partition_height
1	partition_width + 3	partition_height
2	partition_width	partition_height + 3
3	partition_width + 3	partition_height + 3
a, b	partition_width + 4	partition_height
c, j	partition_width	partition_height + 4
h, i	partition_width + 4	partition_height + 3
e, l	partition_width + 3	partition_height + 4
d, f, k, m	partition_width + 3	partition_height + 3

As shown in the table, in case the motion vector points to half-sample position as “1”, partition with (*partition_ width*+3) × *partition_height* integer samples needs to be loaded into on-chip memory. Based on such loading method, the maximum on-chip buffer size used to save the reference partition is 20 by 20, which is much smaller than the size of an entire reference frame, and the on-chip/off-chip data transfer is well modularized for target platform.

The reference partition loading flow is much easier for chroma (4:2:0), since its interpolation method is much simpler than luma [1]. Only (*partition_width*+1) × (*partition _height*+1) integer samples need to be loaded into the on-chip memory if MV does not point to an integer-sample position.

3.4. MB-based deblocking filter

Different from those frame-based implementations that filter MBs after the whole image is reconstructed, MB-based deblocking filter aims to filter each MB immediately after it is reconstructed so that to save the on-chip memory and memory bandwidth. The modules related to the MB-based filtering are shown in Fig.9.

As shown in the figure, the strength decision and edge filter are the standard defined filtering processes, which will not be introduced in this paper. While, the processes before and after those two modules are designed specially for the MB-based data allocation and buffer update, which will be described detailedly in the following using luma as an

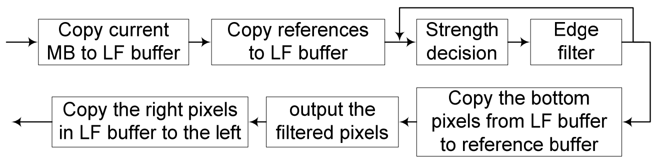
Copy current MB to LF buffer
Copy references to LF buffer
Strength decision
Edge filter
Copy the bottom pixels from LF buffer to reference buffer
output the filtered pixels
Copy the right pixels in LF buffer to the left

Fig. 9. MB-based deblocking filter flow.

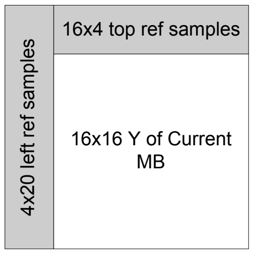

Fig. 10. Loop filter working buffer for luma.

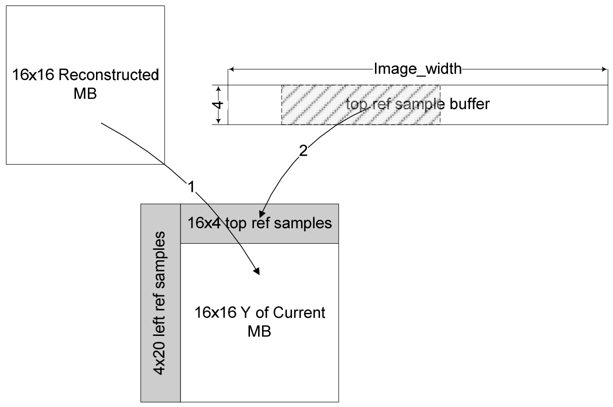

Fig. 11. Memory management before filtering.

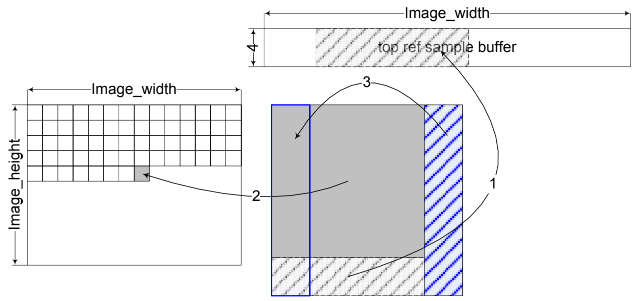

Fig. 12. Memory management after filtering.

instance. There are two MB-based buffers defined for luma deblocking filter: one is the major filter working buffer, saving the reconstructed MB and the reference samples left to and above the current MB; the other is reference buffer saving prior filtered lines of samples above the current MB. The size of the working buffer is 20×20 as shown Fig.10. The size of reference buffer is image width by 4.

Based on the above two buffers, the buffer management flow before filtering is illustrated in Fig.11. As shown in the figure, the reconstructed MB is firstly copied to the bottom-right corner of the loop-filter buffer as indicated by arrow 1. Then, the reference samples spatially above the current MB are copied from the reference buffer into the working buffer as indicated by arrow 2.

After filtering four edges of the current MB, the data copy and buffer update scheme is shown in Fig.12. The bottom-left 16×4 samples (area with gray bias) are first copied to the reference buffer to replace those samples that copied in step2 in Fig. 11 for filtering vertically adjacent MB. The top-left 16×16 final filtered pixels (gray area) are secondly written out to the decoded frame via on-chip/off-chip data transfer routines. Finally, the 4×20 right samples (area with blue bias) are copied to the left as indicated by arrow 3 to be the reference samples for the next adjacent MB's filtering, during which the value of the samples will be further changed.

The buffer management and update manner of chroma (4:2:0) is almost the same with that of luma except the buffer size. The size of working buffer of any chroma component is 12×12, while the reference buffer size is a half of image width by 4.

4. COMPLEXITY ANALYSIS

Based on the above implementation and modularization, it is very easy to put our MB-based AVS decoder into a target platform. The only modification needs to be done is to reorder some of the modules for parallelism and use platform-depended instructions for optimization. Since the computational complexity of the decoder is platform-depended, only the storage complexity is analyzed in this section by comparing with the frame-based implementation. Table 2 presents a comparison of the storage requirements between the MB-based AVS decoder and the frame-based AVS decoder. As shown in the table, *w* and *h* represent the image width and height, respectively; *n* is the number of reference frame with the maximum value of 2. Since the frame-based decoder performs MC directly in the reference frame, the buffer used to load the reference partition is not available (denoted as N.A. in Table 2). Deblocking filter in the frame-based decoder is performed directly in the reconstructed frame after all of the MBs of the image have been decoded. If designing a chip using the MB-based decoder, the first four buffers in Table 2 will be put into off-chip memory, while others will be put into on-chip memory to omit on-chip/off-chip data transfer. While, if using the frame-based decoder to design the chip, reconstruction frame and reference frames are required to be put into

TABLE 2 Storage requirements, in bytes, for the MB-based AVS decoder and the frame-based AVS decoder.

Buffer name	**Formula in the MB-based decoder**	**Formula in the frame-based decoder**
Reference indices of previous frame	$w/8\times h/8$	$w/8\times h/8$
Motion vectors of previous frame	$w/8\times h/8\times 2\times 2$	$w/8\times h/8\times 2\times 2$
Reference frame	$n\times w\times h\times 1.5$	$n\times w\times h\times 1.5$
Reconstruction frame	$w\times h\times 1.5$	$w\times h\times 1.5$
Reference indices	$(w/8+2)+4\times 3\times 2$	$w/8\times h/8\times 2$
Motion vectors	$((w/8+2)\times 4+4\times 3\times 4)\times 2$	$w/8\times h/8\times 2\times 2\times 2$
Reconstruction MB	16×16×1.5	16×16×1.5
Reference partition	20×20+9×9×2	N.A.
QP values	$w/16+4$	$w/16\times h/16$
Intra prediction mode	$(w/8+2)+9$	$(w/8+2)\times(h/8+2)$
Intra prediction	$(w+18)+(w/2+10)\times 2$	16×16+40×3
Deblocking filter	$(w\times 4+20\times 20)+(w/2\times 4+12\times 12)\times 2$	In reconstruction frame
MB temp data	~1565	~1566
Static tables	3961	3961

on-chip memory, which is extremely too expensive. Table 3 compares the buffer size for input video with different resolutions as *n* equal to 1. The first two buffers listed in Table 2 are ignored in Table 3, since those two buffers must be put in the off-chip memory.

TABLE 3 Storage requirements, in bytes, for different image resolutions.

Buffer type		CIF *w*=352, *h*=288	4CIF *w*=704, *h*=576	720p *w*=1280, *h*=720
Reference frame		152064	608256	1382400
Reconstruction frame		152064	608256	1382400
On-chip buffer	Frame-based decoder	24271	77891	168791
	MB-based decoder	11445	15537	30073
	Frame-based vs. MB-based	2.12:1	5.01:1	5.61:1

As shown in the table, "On-chip buffer" refers to the latter 10 buffers listed in Table 2. For the frame-based decoder, the calculation regards the size of "Reference partition" and "Deblocking filter" as zero. Even though the size of reference frame and reconstruction frame are not taken into account, the MB-based decoder still saves significant amount of on-chip memory. The saving ratio is increased with the increment in image resolution.

Roughly comparing the decoding speed of the MB-based decoder and the frame-based decoder, if the on-chip memory of the chip is as large as the size of RAM used in current PC, the frame-based decoder must work faster than the MB-based decoder because of the time saving in buffer update and management. Unfortunately, current generic on-chip memory is less than 1M bytes [5], where the MB-based decoder will work faster than the frame-based decoder because of the time saving in on-chip/off-chip data transfer.

5. CONCLUSIONS

In this paper, a platform independent AVS video decoder is developed for SoC design. Simulating the data access and transfer routines between the on-chip memory and off-chip memory, an MB-based architecture built on two hierarchies is developed for the decoder with adequate buffer definition, data allocation and buffer update. All of the low-level decoding kernels, as entropy decoding, inverse quantization and transform, Intra prediction, MC, deblocking filter and etc., are constructed using the MB-based structure to process a macroblock at a time. Compared to the frame-based architecture, the MB-based architecture can significantly save the on-chip memory. The modularization of function and data flow facilitates the AVS video decoder development on SoC.

ACKNOWLEDGEMENT

The work was supported in part by the Hong Kong Applied Science and Technology Research Institute (ASTRI) and the Chinese University of Hong Kong Focused Investment Scheme (Project 1903003).

REFERENCES

[1] AVS Video Expert Group, Information Technology-Advanced Audio Video Coding Standard Part 2: Video, in Audio Video Coding Standard Group of China (AVS), Doc.AVS-N1063, December, 2003.

[2] Lu Yu, Feng Li, et al, "Overview of AVS-Video: tools, performance and complexity" Proc. of SPIE International Conference on VCIP. Beijing, China, pp. 679–690, July 2005.

[3] AVS reference software, "rm52j_r1.zip", FTP://159.226.42.57/incoming/dropbox/video_software/P2_software.

[4] Nikil D. Dutt and Alexandru Nicolau, "On-Chip vs. Off-Chip Memory: The Data Partitioning Problem in Embedded Processor-Based Systems", ACM Transactions on Design Automation of Electronic Systems, vol. 5, pp. 682-704, July, 2000.

[5] DaVinci^{TM} Digital Media Processors, http://focus.ti.com/paramsearch/docs/parametricsearch.tsp?family=dsp&sectionId=2&tabId=1852&familyId=1300.