MVA2005 IAPR Conference on Machine VIsion Applications, May 16-18, 2005 Tsukuba Science City, Japan
3-13

New Macroblock Engine Architecture for Video Processing

Trio ADIONO[1], Dani Fitriyanto[1], Akhmad Mulyanto[1], Sumek Wisayataksin[2], Kazumasa Takeichi[2], Dongju Li[2], Tati R. Mengko[1], Hiroaki KUNIEDA[2]

[1)] Department of Electrical Engineering, Bandung Institute of Technology, Indonesia
[2)] Department of Communications and Integrated Systems, Tokyo Institute of Technology, Japan
email : tadiono@paume.itb.ac.id

Abstract

A new engine for macro-block based video processing is introduced in this paper. This engine increases efficiency, flexibility and extensibility of data generation for macro-block based video processing system. In the proposed system, a new specific instruction sets that can access data in pixel, line, block, macro-block or frame within a clock cycle are introduced. Thus, efficiency of video processing system is increased. Additionally, the programmability of data access enables dynamic scheduling for various video processing applications. It extremely reduces processing time while reducing the control complexity as well. This architecture also has scalability for different size of image and has expandability for new macro-block based processor. Implementation to typical video compression application shows high performance result and easy system implementation.

1. Introduction

Most video processing algorithm involves processing a pixel with its neighbor pixels. In video compression algorithm, in order to exploit temporal and spatial redundancy, most of algorithms do processing to groups of pixels, called block and macro-block processing.

In conventional system, dedicated hardware based system implementation is usually designed for each macro-block processing unit, such as DCT, IDCT, ME, MC, IQ and Q. This approach requires large number of logic gates which increases design complexity. Moreover, dedicated design has very small flexibility for design scalability and expandability.

On the other hand, general purpose based video coding implementation requires computation overhead for macro-block based memory data accessed. Consequently, high clock frequency and high power consumption are required. These are not suitable for typical video processing application that needs low power features, and large computation times.

To overcome problems mentioned above, we introduce a new engine which is functioning to generate data for many types of image processing modules. This proposed engine provides easy access to memory data inside the frame store memory of video processing systems. Processor can access data in macro-block, block or pixel.

Due to its programmability features, we can also program processing element function according to required processing that may vary according to the content of video image data. Thus, we can employ dynamic scheduling for video coding system, based on video content that may increase computation efficiency.

2. Macroblock Engine System Architecture

In order to obtain high performance system architecture, the addressing of frame store memory method is firstly optimized to provide easy access of memory data. Then instruction sets are design to enable data read/write in various ways. Finally, considering both addressing method and instruction sets, the system architecture is designed.

2.1 Addressing method

To simplify and increase address generation speed, we arrange data inside memory in macro-block based system. We use pixel position inside block (*p_x and p_y)*, block number (*b_x* and *b_y)*, macro-block number (*mb_x* and *mb_y)* and frame number *(fr)* as address value, as illustrated in Figure 2.1 and Figure 2.2. As a result, we can directly map real macro-block address using its position parameters (*fr*, *mb_y*, *mb_x, b_y, b_x, p_y, p_x)* as shown in Figure 2.2.

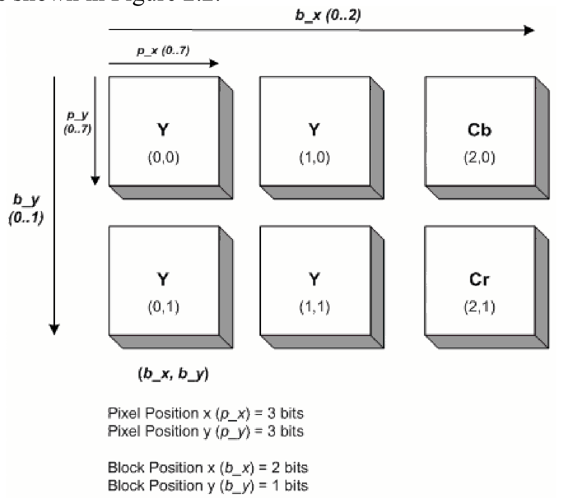
b_x (0..2)
p_x (0..7)
p_y (0..7)
b_y (0..1)
Y (0,0)
Y (1,0)
Cb (2,0)
Y (0,1)
Y (1,1)
Cr (2,1)
(b_x, b_y)
Pixel Position x (p_x) = 3 bits
Pixel Position y (p_y) = 3 bits
Block Position x (b_x) = 2 bits
Block Position y (b_y) = 1 bits

Figure 2.1 Pixel and Block position inside the macro-block addressing

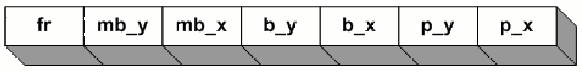

Figure 2.2 Memory address structure

2.2 Instruction sets

This address generator functionality is defined by sets of instructions which are specially designed for this processor. By utilizing this instruction, we can provide data for processing element according to its input-output data scheme.

Utilized Instructions can be categorized as memory access, address setting, jump, and control as shown in Table 2.1.

Table 2.1 Instruction Sets of Macroblock Engine.

Instruction name	Function	Description
IRD_LINE(*obj_operand*)	Access	Read Data in horizontal mode
IWR_LINE(*obj_operand*)	Access	Write Data in horizontal mode
IRD_COLUMN(*obj_operand*)	Access	Read Data in vertical mode
IWR_COLUMN(*obj_operand*)	Access	Write Data in vertical mode
IINC_ADD(*obj_operand*)	Address Setting	Increase add number
ISET_ADD(*obj_operand*)	Address Setting	Set Offset Address
ISET_MV(*mvx*, *mvy***)**	Address Setting	Set MV
IJUMP_FLAG(*Add*)	Jump	Jump to "Add" Address when Flag is '1'
IJUMP(*Add, JCount***)**	Jump	Jump to "Add" Address when Flag is '1' for "JCount"
IWAIT(n)	Control	
ISET_MODULE(*module_no*)	Control	Enable the indicated module
ISTOP	Control	Stop Running

2.3 Memory read/write instructions

This type of instruction read/write data from frame store memory according to operand value. The direction of data reading/writing within a block of data can be horizontal or vertical, depends on "LINE" or "COLUMN" type of instruction.

The value of operand, as shown in Table 2.2 shows △ value to be read. For example, to read a macro-block of data, we may write instruction as follows:

IRD_LINE(0; 0,2,7; 0,7);

Table 2.2 Operand Value

Obj_operand	*Fno, △ mbx, △ bx, △ px, △ mby, △ by, △ py*

2.4 Address setting instructions

This type of instruction manipulates address value that will be used as offset address by Memory Access Instruction. ISET_ADD will set current address to its operand value. Therefore we can set the reference position of data to be read inside a frame.

IINC_ADD increases current address according to its operand value. Increment is done independently among p_x, b_x, and mb_x. Second complement representation is used for operand representation. Therefore, using this instruction we can also decrease the address value. Unlike ISET_ADD, this instruction is designed for relative address set according to current position. Usually it is useful for address setting inside loop condition.

Unlike the other instructions, ISET_MV is only used for Motion Compensation operation that commonly used in video compression application. Using proposed architecture, mv process can be implemented by shifting address in x and y direction according to operand value of ISET_MV.

2.5 Control instructions

ISTOP Terminate the program execution.

ISET_MODULE (*module_no*) Select active module. We assigned specific module_no to each processing hardware. This instruction must be executed before read data for each module. In DCT/IDCT Function ISET_MODULE also include information inter/intra MB and quantization/inverse quantization value.

IWAIT(n) Delay for n clks before executing the next instruction. It is usually used between read and write data from module. n is bit [15..0]. Currently DCT and IDCT use this instruction.

2.6 Jump instructions

IJUMPNZ(N) This instruction will change program counter flag based on flagJMP register value. If the flagJMP=1 then PC=PC-N, otherwise PC=PC+1.

By this instruction, we increase the default reference address in **regAX** and **regAY** which were set by **ISET_ADD** instruction. The increment is done independently among p_x, b_x, and mb_x. Increment is also used to avoid overflow from each bus.

2.7 General architecture

In order to generate addressing according to macroblock based memory map, we design an AGU unit as described in Figure 2.3. In this system AGU generate addressing by executing instruction in Program Memory (ROM). Inside program memory, there are several instructions. The executed instruction will be determined by Program Counter (PC) which point out to the address of Program Memory. According to this instruction, AGU will perform loop operation in Loop Counter block. The address generated by loop counter will be added with offset value in register A of ALU_X and ALU_Y. Finally, both x and y address will be mixed again by multiplexer to perform real address.

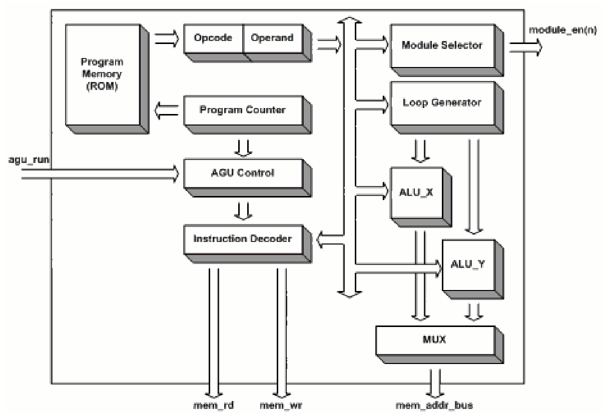

Figure 2.3 AGU Unit Design

2.7.1 AGU controller

AGU Controller has a function to fetch and execute the instruction. It works within 3 stages, Instruction Fetch, Decoding and Execute.

In Instruction Fetch stage, executed instruction is transferred from program memory into instruction register, according to address pointed by Program Counter.

In Decoding Stage, all AGU registers are set according to the instruction. For example, we program aguloop registers when IRD_LINE or IRD_COLUMN instruction. Except ISTOP instruction, the value of PGC is also in increase in this stage.

In Execution stage, the real address value is generated by aguloop unit. At the beginning of Execute stage, reset signal is generated.

2.7.2 Loop generator

According to our memory map structure and supported instruction sets, we construct cascade structure of counter. The counters are arranged from pixel counter to macroblock counter as shown in Figure 2.4. Only pixel counter the arrangement may be exchange between x and y. It depends on executed instruction, whether horizontally or vertically read. If horizontal (IRD_LINE), the pixel x counter comes first, otherwise pixel y will.

When execution stage starts, it starts counting from zero until delta value, which is assigned by operand 1 and operand 2 of instruction. Therefore we can generate address for pixel, line, block, macroblock or frame access.

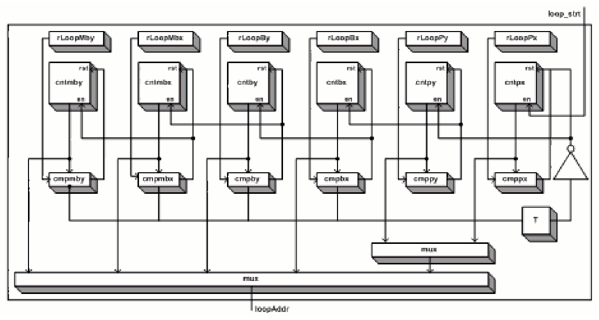

Figure 2.4 Aguloop Architecture

2.7.3 ALUX and ALUY

Since the generated address by aguloop module always refer to 0 position, we need randomly access in line, block, macroblock or frame. For that kind of purpose, we create an offset value which is store in regAX and regAY (differentiated between x and y). By adding these registers value and address generated by aguloop, we can access the data randomly.

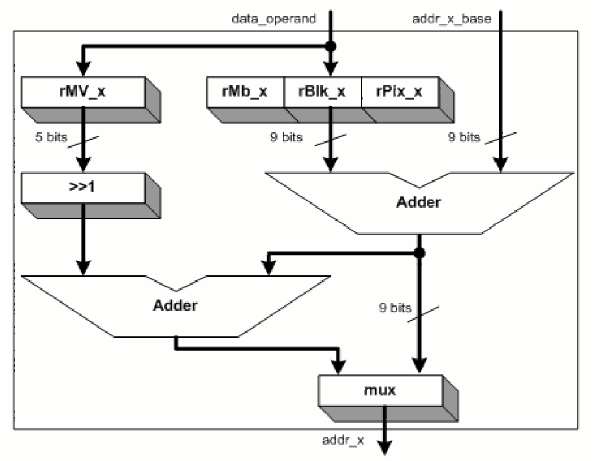

Figure 2.5 ALUX and ALUY Architecture.

3. Application in Video Coding

In order to verify system functionality, we develop macroblock-engine system using FPGA as shown in Figure 3.1. This system includes camera as data acquisition, and NTSC display.

Figure 3.1 Macroblock Engine System using FPGA

3.1 Application to DCT and IDCT

For verification of proposed architecture, we implement proposed system to DCT processing as shown in Figure 3.2. This processing is typically used in many video compression applications.

For this processing, following shows program structure for DCT processing. This program is implemented for a block processing. For frame processing, we must perform loop operation.

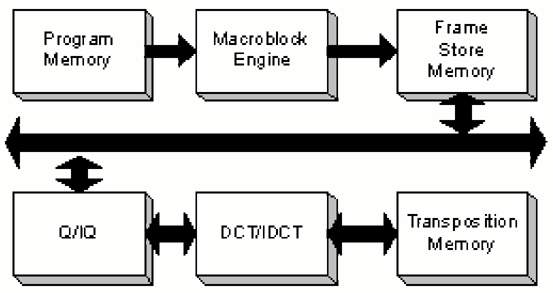

Figure 3.2 Application of Macroblock Engine for DCT/IDCT Transformer

For DCT processing we introduce parallel processing architecture for DCT coefficient computation. Eight processors are used to perform parallel computation. In order to enhance multiplication operation, LUT based multiplication is used to compute the multiplication of input data and coefficient of DCT.

3.2 Experimental Result

Implementation of DCT/IDCT processing shows that system can effectively perform processing with simple engine programming, as follows:

```
ISET_MODULE(0x01)
   100: IJUMP_FLAG(200,Flag)
        IRD_COLUMN(Py=7,Px=7)
        IWAIT(74)
        IWR_LINE_DCT(Py=3,Px=7)
  200:  ISHIFT_BLK
        IJNZ(100,6)
        ISET_MODULE(0x00)
        ISTOP
```

Figure 3.3 shows the processing of DCT/IDCT for quantize value with quantize value 2. We can learn that the reconstructed image is similar to input data. Applying larger quantize value, as shown in Figure 3.4, will produce larger image noise but small number of image coefficient which may result in small encoding bits.

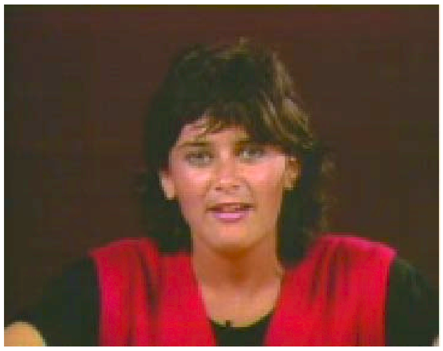

Figure 3.3 Reconstructed of video stream Miss America applying Quantize Value = 2

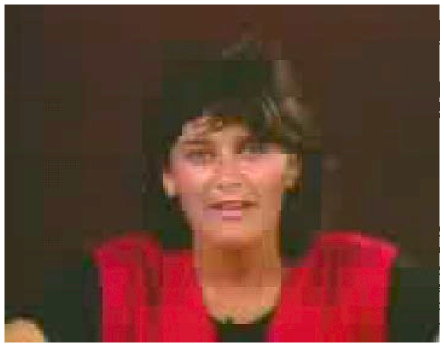

Figure 3.4 Reconstructed of video stream Miss America applying Quantize Value = 10

4. Conclusion

A new macro-block engine architecture which can randomly access data inside memory is introduced in this paper. Special instructions are design to provide data for massive parallel video processor. Addressing method and type of instructions used in this processor will decrease system complexity and increase system scalability that results in efficient VLSI implementation of the system. Programmability features also increase design reusability which may gain small design size and low design complexity. Shared control signal among processing element also result in easy of interfacing, reduce design time and easy for design verification and debugging.

References

[1] C. Honsawek, K. Ito, T. Ohtsuka, T. Isshiki, D. Li, T. Adiono, H. Kunieda, "System-MSPA design of H.263+ Video Encoder LSI for Face Focused Videotelephony," Proc. of IEEE APCCAS, No.00EX394, pp.152-155, China, Dec.4-6,2000.

[2] D. Li, T. Adiono, C. Honsawek, and H. Kunieda, "Multimedia LSI Design Based on Window-MSPA Architecture," ISPACS'99 Thailand, Pucket, Dec.8-10,1999 (Invited Paper).

[3] C. Honsawek, K. Ito, T. Ohtsuka, T. Isshiki, D. Li, T. Adiono, H. Kunieda, "System-MSPA Design of H.263+ Video Encoder/Decoder LSI for Videotelephony Application," IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Vol. E84-A Num. 11 pp.2614-2622 (2001.11)

[4] Dongju Li and Hiroaki Kunieda, "Memory Sharing Processor Array (MSPA) Architecture," IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Special Section on VLSI Design and CAD Algorithms Vol. E79-A, No. 12 Pp.2086-2096, Dec. 1996