REMARC: Reconfigurable Multimedia Array Coprocessor

Takashi Miyamori* and Kunle Olukotun
Computer Systems Laboratory
Stanford University
takashi@ogun.stanford.edu, kunle@ogun.stanford.edu

Abstract

This paper describes a new reconfigurable processor architecture called REMARC (Reconfigurable Multimedia Array Coprocessor). REMARC is a reconfigurable coprocessor that is tightly coupled to a main RISC processor and consists of a global control unit and 64 programmable logic blocks called nano processors. REMARC is designed to accelerate multimedia applications, such as video compression, decompression, and image processing. These applications typically use 8-bit or 16-bit data therefore, each nano processor has a 16-bit datapath that is much wider than those of other reconfigurable coprocessors. We have developed a programming environment for REMARC and several realistic application programs, DES encryption, MPEG-2 decoding, and MPEG-2 encoding. REMARC achieves speedups ranging from a factor of 2.3 to 21.2 on these applications.

1 Introduction

As the demand for multimedia applications, such as video compression, decompression, and image processing, increasing the performance of these applications on general-purpose microprocessors is becoming more important. Recently, multimedia instructions have been added to most general-purpose microprocessor ISAs to increase performance of multimedia applications [1–3] . Most of these ISA additions work by segmenting a conventional 64-bit datapath into four 16-bit or eight 8-bit datapaths. The multimedia instructions exploit SIMD parallelism by operating on four 16-bit or eight 8-bit data values. However, a 64-bit datapath limits the speedups to a factor of four or eight even though many multimedia applications have much more parallelism to exploit.

Recently, computer architectures that connect a reconfigurable coprocessor to a general-purpose microprocessor have been proposed [4–10] . The advantage of this approach is that the coprocessor can be reconfigured to improve the performance of a particular application. All of these proposed architectures use field programmable gate arrays (FPGAs) for the reconfigurable hardware. The FPGA architecture, such as the small width of the programmable logic blocks and the programmable interconnection network, provides a great flexibility for the systems. On the other hand, FPGAs have following shortcomings:

- The small width of the programmable logic blocks results in large area and delay overheads to implement wider (8 – 16-bit) datapaths.
- FPGAs are slower than a custom integrated circuit and have lower logic density.

Array processors, such as general-purpose systolic array processors, wavefront array processors [11] , PADDI [12] , PADDI-2 [13] , and Matrix [14] , work very well on multimedia applications. These processors have 8-bit or 16-bit datapaths and each programmable logic block has an 8 to 32-entry instruction RAM that makes it easy to support multiple functions.

In this paper we describe a new reconfigurable processor architecture called REMARC (Reconfigurable Multimedia Array Coprocessor). REMARC is a reconfigurable coprocessor that is tightly coupled to a main RISC processor and consists of a global control unit and 64 16-bit programmable logic units called nano processor. We have developed a programming environment for REMARC and evaluated the performance of the architecture using several realistic application programs.

The rest of this paper is organized as follows. In Section 2, we describe the architecture of REMARC. Section 3 describes the programming environment for REMARC and an example of a REMARC program. In Section 4 we show the results of our performance evaluation. Finally, we conclude in Section 5.

*On leave from TOSHIBA corporation, System ULSI Engineering Laboratory, 580-1 Horikawa-cho Saiwai-ku Kawasaki, 210, JAPAN. E-MAIL: miyamori@sdel.toshiba.co.jp

2 REMARC Architecture

2.1 Architecture Overview

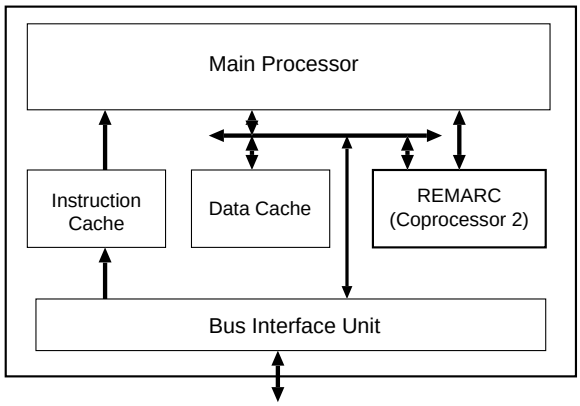

Figure 1: Block Diagram of a microprocessor with REMARC

Figure 1 shows a block diagram of a microprocessor which includes REMARC as a coprocessor. We used the MIPS-II ISA [15] as the base architecture of the main processor. The MIPS ISA can support up to four coprocessors. Coprocessor 0 is already used for memory management and exception handling, and coprocessor 1 is used for a floating point coprocessor. REMARC operates as coprocessor 2. The main processor issues instructions to REMARC which executes them in a manner similar to a floating point coprocessor. The difference between REMARC and a floating point coprocessor is that functions of REMARC instructions are programmable. With REMARC, users can define and configure their own instructions specialized for their applications.

Figure 2 shows a block diagram of REMARC. REMARC consists of 8x8 array of nano processors and a global control unit. Each nano processor has a 32-entry instruction RAM (nano instruction RAM), a 16-bit ALU, a 16-entry data RAM, and 13 16-bit data registers. Each nano processor can communicate to the four adjacent nano processors through the dedicated connections and to the processors in the same row and the same column through the 32-bit Horizontal Bus (HBUS) and the 32-bit Vertical Bus (VBUS). The nano processors do not have Program Counters (PCs) by themselves. Every cycle the nano processor receives the PC value, "nano PC", from the global control unit. All nano processors use the same nano PC and execute the instructions indexed by the nano PC in their nano instruction RAM. At this point, REMARC can be regarded as a VLIW processor in which each instruction consists of 64 operations. It makes REMARC much simpler than distributing execution control across the 64 nano processors.

The global control unit controls the nano processors and the transfer of data between the main processor and the nano processors. The global control unit includes a 1024-entry instruction RAM (global instruction RAM), data registers, and control registers. These registers can be accessed by the main processor directly using main processor instructions, move from/to coprocessor or load/store coprocessor. Eight 32-bit VBUSs are also used for communication between the global control unit and the nano processors. According to a global instruction the global control unit reads the data registers and outputs these values on the VBUSs. The nano processors can read these data from the VBUSs. Furthermore, the global control unit reads data from the VBUSs as the result of nano processor's calculation and stores them into the data registers.

2.2 Main Processor Architecture

The MIPS ISA is extended for REMARC using the instructions listed in Table 1.

rgcon	*src*
rncon	*src*
rex	*remarc_reg, offset(base)*
lduc2	*remarc_reg, offset(base)*
sduc2	*remarc_reg, offset(base)*
mtc2	*remarc_reg, src*
mfc2	*remarc_reg, dst*
ctc2	*remarc_reg, src*
cfc2	*remarc_reg, dst*

Table 1: New instructions for REMARC

The configuration instructions, *rgcon* and *rncon*, download programs from memory and store them in the global instruction RAM and the nano instruction RAMs respectively. The start address of the program is specified by the value of the source register (*src*).

Rex instruction starts execution of a REMARC program. The start address of the global instructions is specified by sum of the *offset* field and the *base* register.

Lduc2 and *sduc2* are load and store coprocessor instructions which transfer double word (64-bit) data between memory and the REMARC data registers. These instructions ignore the least significant 3 bits of their operand address. Therefore, all load and store addresses are on a double word boundary. *Lduc2* instruction saves the least significant 3 bits of its operand address into a control register for misaligned address data support.

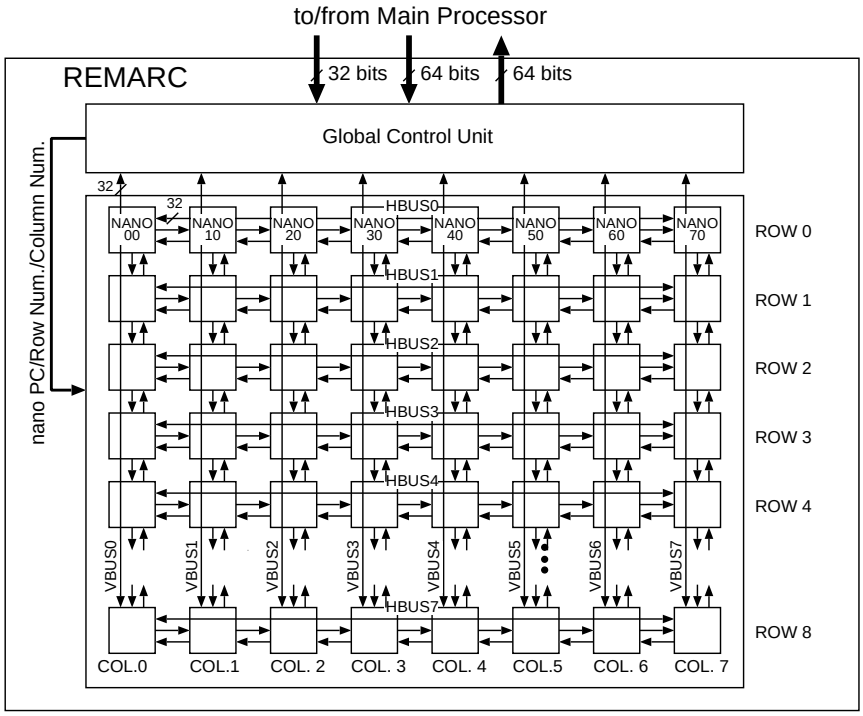

Figure 2: Block Diagram of REMARC

Mfc2 and *mtc2* are move coprocessor instructions which transfer word (32-bit) data between the general-purpose registers (integer registers) in the main processor and the REMARC data registers. *Cfc2* and *ctc2* instructions transfer data between the integer registers and the REMARC control registers.

2.3 Nano Processor

Figure 3 shows the architecture of the nano processor. The nano processor consists of a 32-entry instruction RAM (nano instruction RAM), a 16-bit ALU, a 16-entry data RAM, an instruction register (IR), eight 16-bit data registers (DR), four 16-bit data input registers (DIR), and a 16-bit data output register (DOR).

The nano processor can accept data from the DOR registers of the four adjacent nano processors (up, down, left, and right) through dedicated connections (DINU, DIND, DINL, and DINR). The DOR register data can be used for source data of ALU operations or data inputs of a DIR register. These local connections provide high bandwidth pathways within the processor array.

Other communication lines between nano processors are the 32-bit HBUSs and the VBUSs. Nano processors in the same row are connected by an HBUS, and a VBUS connects processors in the same column. The data in the DOR register can be sent to the VBUS or the HBUS. The HBUSs and the VBUSs allow data to be broadcast to the nano processors in the same row or column and to transfer of data between non-adjacent nano processors. The DIR registers accept inputs from the HBUS, the VBUS, the DOR, or the four adjacent nano processors. Because width of the HBUS and the VBUS is 32 bits, data on the HBUS or the VBUS are stored into the DIR register pair, the DIR0 and DIR1, or the DIR2 and DIR3.

The 16-bit ALU executes 30 instructions shown in Table 2. The ALU supports arithmetic, logical, and shift instructions similar to general purpose microprocessors. The load and store instructions access the local data RAM. The Minimum (MIN), Maximum (MAX), Average (AVE), and Absolute and ADD (ABSADD) instructions are effective for multimedia applications. Recently, some microprocessors also supported these kinds of instructions as multimedia extensions. The ALU does not include a hardware multiplier or multiply instructions. The Shift Right Arithmetic and Add (SRAADD) instruction provides a primitive operation for constant multiplications instead. The Shift and Logical instructions, SRLAND, SLLAND, SRLOR, SLLOR, are used for bit level data manipulations. The DR registers, the DIR registers, 8-bit or 16-bit immediate data, and the DOR registers in the four adjacent nano processors can be used as source data for the ALU operations. The result of the ALU operation is stored into the DOR register or one of the DR registers.

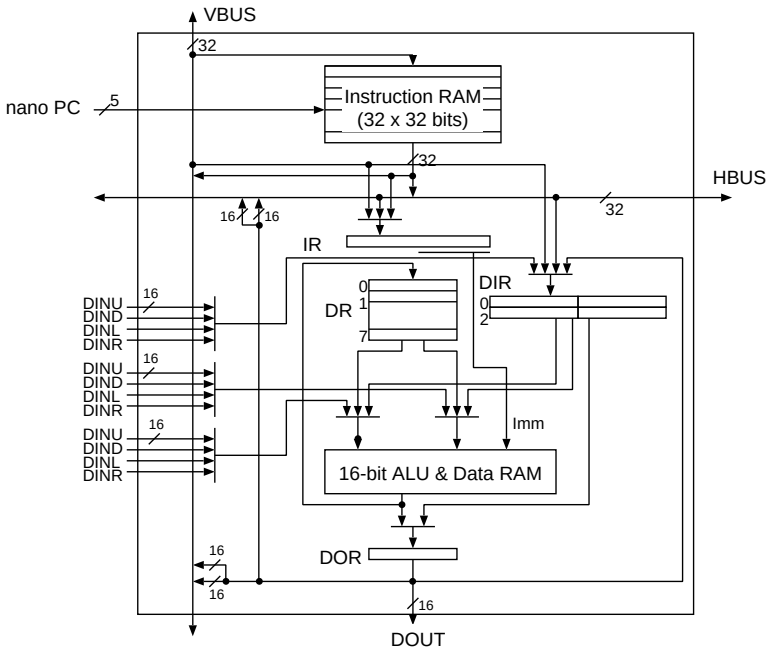

Figure 3: Nano Processor Architecture

ADD	*Add*
SUB	*Subtract*
SLTU	*Set Less Than Unsigned*
ADDI	*Add Immediate (8 bits)*
AND	*And*
OR	*Or*
XOR	*Exclusive Or*
NOT	*Not*
ANDI	*And Immediate (8 bits)*
MOV	*Move*
LDI	*Load Immediate (16 bits)*
SRA	*Shift Right Arithmetic*
SRL	*Shift Right Logical*
SLL	*Shift Left Logical*
SRAV	*Shift Right Arithmetic Variable*
SRLV	*Shift Right Logical Variable*
SLLV	*Shift Left Logical Variable*
LDA	*Load from Data RAM by Absolute Addressing*
LDR	*Load from Data RAM by Register Indirect Addressing*
STA	*Store to Data RAM by Absolute Addressing*
STR	*Store to Data RAM by Register Indirect Addressing*
MIN	*Minimum*
MAX	*Maximum*
AVE	*Average with Rounding*
ABSADD	*Absolute and Add*
SRAADD	*Shift Right Arithmetic and Add*
SRLAND	*Shift Right Logical and And*
SLLAND	*Shift Left Logical and And*
SRLOR	*Shift Right Logical and Or*
SLLOR	*Shift Left Logical and Or*

Table 2: ALU Operations of Nano Processor

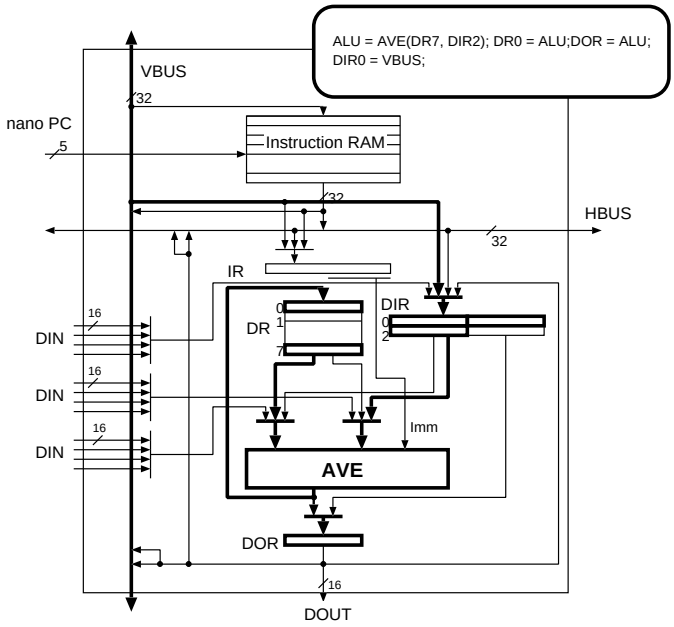

Figure 4: Nano Instruction Example

An example of a nano instruction is illustrated in Figure 4 . In this example, the DR7 and DIR2 registers are used for source data of *AVE* (Average) operation. This result is placed into the DOR register and the DR0 register. At the same time 32-bit data on the VBUS are stored into the DIR0 and DIR1 registers.

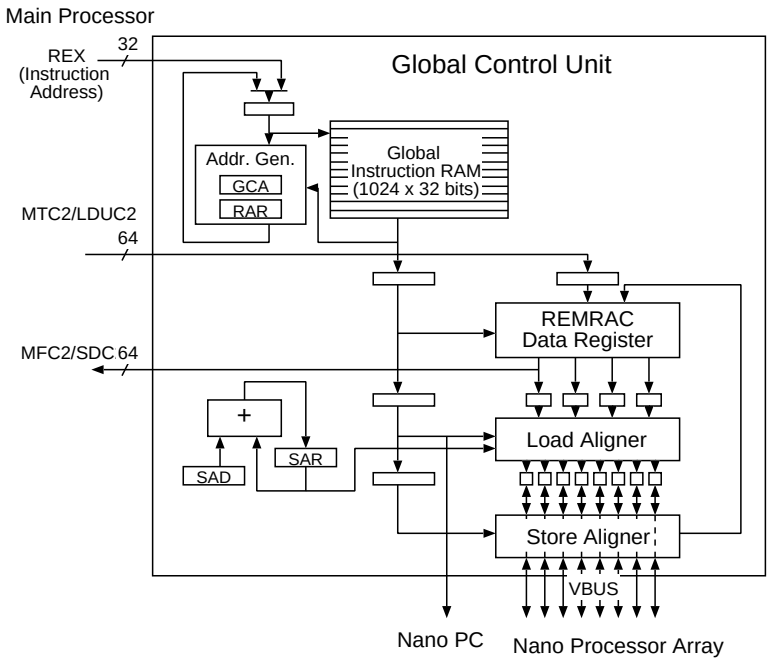

Figure 5: Global Control Unit Architecture

2.4 Global Control Unit

Figure 5 shows the architecture of the global control unit. The global control unit includes a 1024-entry instruction RAM (global instruction RAM), 64-bit data registers ($0 – $31), and four control registers. These are GCA(Global Configuration Address Register), SAR(Shift Amount Register), SAD(Shift Amount Displacement Register), and RAR(Return Address Register).

The two main functions of the global control unit are to control execution of nano processors and to transfer data between the main processor and the nano processors.

1) Control of Nano Processors

The nano processor does not have a Program Counter (PC). The global control unit generates the PC value ("nano PC") for all nano processors every cycle. All nano processors use the same nano PC and execute the instruction indexed by the nano PC. However, each nano processor has its own nano instruction RAM, so that different instructions can be stored at the same address of each nano instruction RAM. Therefore, each nano processor can operate differently according to the nano instructions stored in the local nano instruction RAM. This makes it possible to achieve a limited form of Multiple Instruction Stream, Multiple Data Stream (MIMD) operation in the processor array.

Multimedia applications have a significant amount of Single Instruction Stream, Multiple Data Stream (SIMD) type parallelism. During coding of application programs, we found that most of nano processors perform the same operation every cycle. Storing the same instructions in multiple nano instruction RAMs wastes nano instruction RAM resources. To make more efficient use of these resources, we defined two instruction types called HSIMD (Horizontal SIMD) and VSIMD (Vertical SIMD). In addition to the nano PC field, an HSIMD instruction has a column number field that indicates which column to use for the instruction. For instance, as shown in Figure 6 , if the column number field is 3, each nano processor in the column 3 outputs the nano instruction indexed by the nano PC to all nano processors in the same row through the HBUS. All the nano processors in the same row execute this nano instruction. Using HSIMD and VSIMD

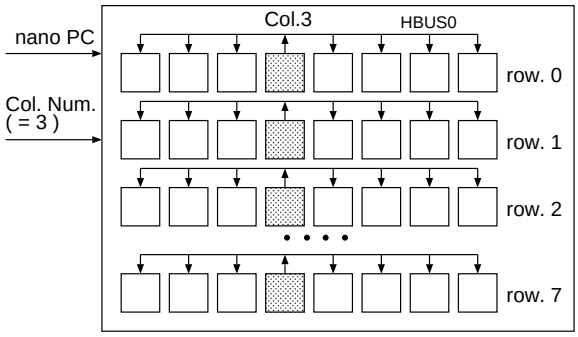

Figure 6: Example of HSIMD Instruction

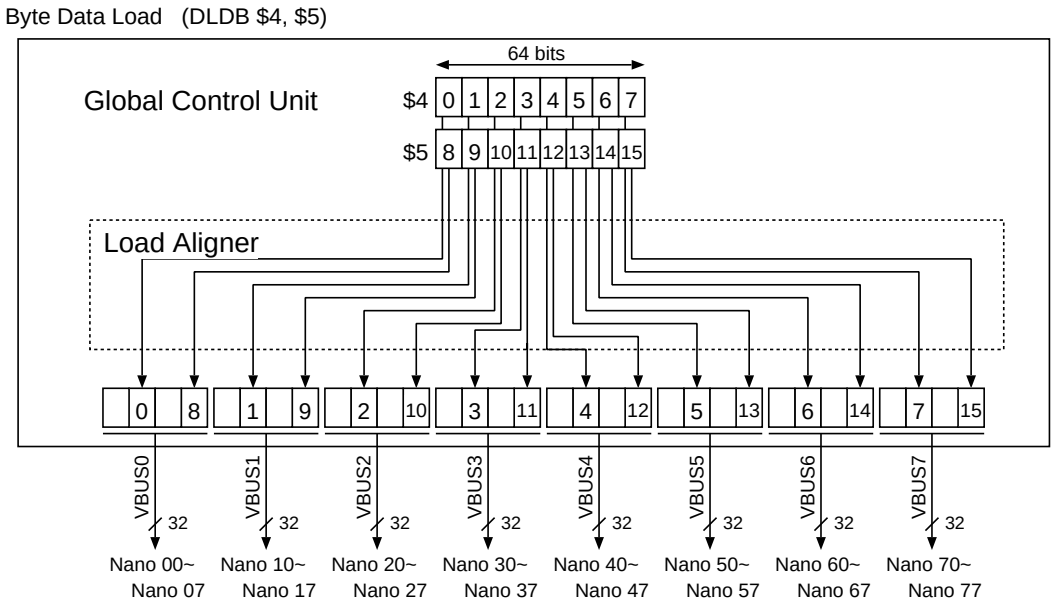

Figure 7: Example of Load Aligner Operation

instructions, nano instructions can be shared with 8 nano processors in the same column or row. This can reduce the number of nano instructions required by a factor of 8.

2) Transfer Data between Main Processor and Nano Processors
The data registers and the control registers in the global control unit can be accessed directly by the main processor instructions using lduc2, sduc2, mfc2, and mtc2 shown in Table 1. The global control unit also controls data transfer to and from nano processors through the load and store aligners according to the global instructions.

Figure 7 shows an example how the load aligner operates. The $4 and $5 REMARC data registers include 16 byte data. The load aligner outputs each byte data in the $4 and $5 to the corresponding VBUS. Therefore, the nano processors in each column can use each byte data. This operation is similar to unpack or expand instructions in the multimedia instruction sets [1–3] . The load aligner also supports halfword data load and word data load. Another feature of the load aligner is misaligned data load. The byte funnel shifter in the load aligner aligns data in the data registers according to the value in the SAR (Shift Amount Register) which stores the least significant 3 bits of the last *lduc2* instruction.

The store aligner is the reverse of the load aligner. It inputs data from VBUSs and packs them into the data registers.

3 Programming Environment

3.1 Programming Environment

We have developed the REMARC programming environment, as shown in Figure 8 , including the REMARC global instruction assembler and the nano instruction assembler. The global instruction assembler starts with a global assembly code and generates configuration data and label information. The configuration data are written in text. If the configuration data are stored in a file named *sample.gcfg*, the following C code can be used to initialize some array with the configuration data.

```
unsigned int remarc_gcfg[] =
#include "sample.gcfg"
;
```

The nano instruction assembler starts with nano assembly code and generates configuration data. The configuration data are loaded into an array along with the global configuration data.

The global assembler also generates a file named *remarc.h* which defines labels for the global assembly code. Using "*asm* " compiler directive, assembly instructions can be written directly into C code. The "*-S* " option of the *gcc* compiler generates intermediate assembly code which includes the new main processor instructions for REMARC. The *gcc* compiler does not support these new instructions, so the REMARC instruction assembler is used to translate them into binary code. Finally, the *gcc* compiler generates

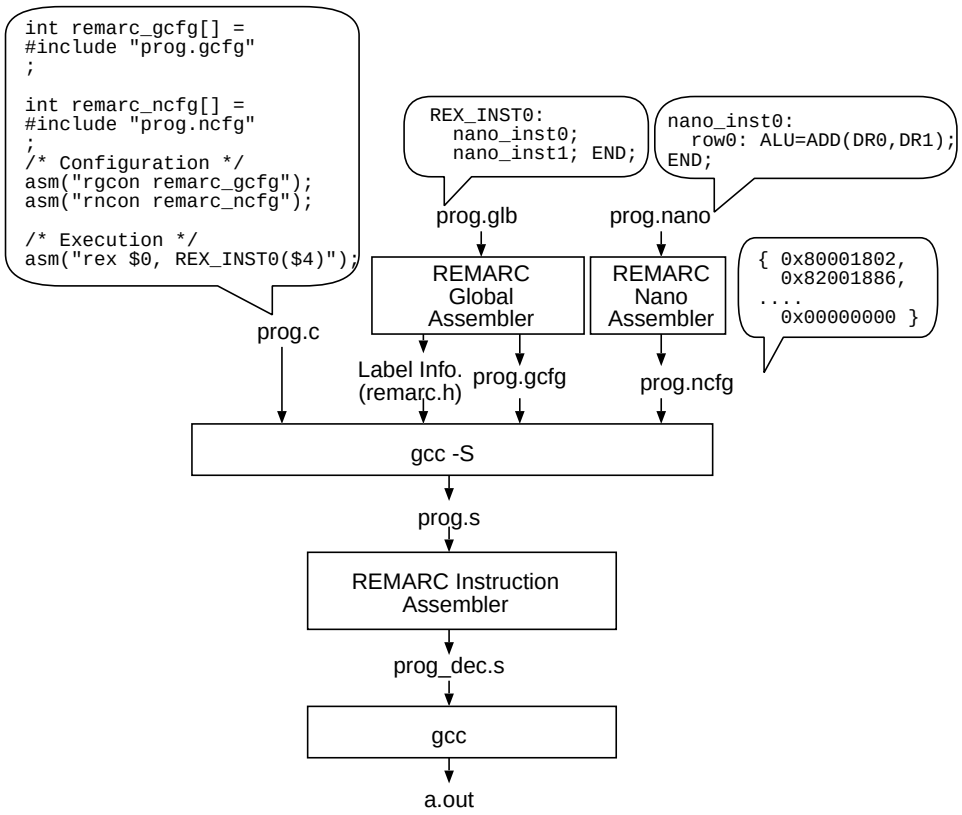

Figure 8: Programming Environment

executable code which includes the new main processor instructions and the global and nano configuration data.

3.2 Sample Program

Figure 9 is an example of a REMARC program. This example inputs two 8 halfword data from memory and makes average of each halfword data. The results of this operation are stored into memory. In lines 1–6 in *main.c*, the integer arrays *remarc_gcfg* and *remarc_ncfg* are initialized with configuration data stored in the files, *sample.gcfg* and *sample.ncfg*. In line 7, the file *remarc.h* which defines the labels of global assembly code is included. For instance, as shown in line 27, the label REX_PAVEH can be used as offset value which points to the global instruction.

In the main routine, configuration data are transferred into the global instruction RAM and the nano instruction RAMs by the *rncon* and *rgcon* instructions in lines 17 and 18. Then, four double word data are read from memory and stored into the $6, $7, $8, and $9 data registers by the *lduc2* instructions (in lines 23–26).

In line 27, the rex instruction starts execution of the REMARC global instruction whose start address is specified by its operand address. In this case, the operand address is sum of *remarc_gcfg* which is stored in the variable *rex_inst* and REX_PAVEH.

The global instructions shown in *sample.glb* are executed from the label REX_PAVEH. The first global instruction reads data from the $6, $7, $8, and $9 data registers into the VBUSs and starts execution of the nano instruction specified by the label VBUS_DIR0_ROW0. This nano instruction receives data from the VBUSs and stores them in the DIR0 and DIR1 register pair as shown in *sample.nano*.

The second global instruction starts execution of the nano instruction labeled by AVE_ROW0. This nano instruction calculates average of two halfwords stored in the DIR0 and DIR1 registers and places the result into the DOR register. The third global instruction starts execution of the nano instruction labeled by DOR_VBUSL_ROW0. This nano instruction places the DOR register data on the lower 16 bits of the VBUSs. Then, the global instruction reads the data on the VBUSs and stores them into the $10 and $11 data registers.

In lines 28 and 29 of *main.c*, the $10 and $11 data registers are stored into memory as the result of the parallel halfword average operation.

main.c

```
 1: unsigned int remarc_gcfg[] =
 2: #include "sample.gcfg"
 3: ;
 4: unsigned int remarc_ncfg[] =
 5: #include "sample.ncfg"
 6: ;

 7: #include "remarc.h"

 8: short int a[8] = {1,2,3,4,5,6,7,8};
 9: short int b[8] = {1,2,3,4,5,6,7,8};
10: short int c[8];

11: void main(int argc, char *argv[])
12: {
13:   unsigned int *pncfg, *pgcfg, *rex_inst;
14:   short int *pa, *pb, *pc;

15:   pncfg = remarc_ncfg;
16:   pgcfg = remarc_gcfg;
17:   asm("rncon (%0)" : : "r" (pncfg): "$1" );
18:   asm("rgcon (%0)" : : "r" (pgcfg): "$1" );

19:   rex_inst = remarc_gcfg;

20:   pa = a;
21:   pb = b;
22:   pc = c;

23:   asm("lduc2 $6, 0(%0)" : : "r" (pa) : "$1");
24:   asm("lduc2 $7, 8(%0)" : : "r" (pa) : "$1");
25:   asm("lduc2 $8, 0(%0)" : : "r" (pb) : "$1");
26:   asm("lduc2 $9, 8(%0)" : : "r" (pb) : "$1");

27:   asm("rex $10, REX_PAVEH(%0)" : : "r" (rex_inst) : "$1" );

28:   asm("sduc2 $10, 0(%0)" : : "r" (pc) : "$1");
29:   asm("sduc2 $11, 8(%0)" : : "r" (pc) : "$1");

30: }
```

sample.glb

```
# Example: Parallel Average Half-word

REX_PAVEH:
  VBUS_DIR0_ROW0; VBUS = DLDH($6, $8);
  AVE_ROW0;
  DOR_VBUSL_ROW0; $10 = STH(VBUS); END;
```

sample.nano

```
# Example: Parallel Average Half-word
VBUS_DIR0_ROW0:
 ROW0: DIR0 = VBUS;
END;

AVE_ROW0:
 ROW0: ALU = AVE(DIR0, DIR1); DOR = ALU;
END;

DOR_VBUSL_ROW0:
 ROW0: VBUSL = DOR;
END;
```

Figure 9: Program Example

3.3 Simulation Methodology

We developed REMARC simulator using the SimOS simulation environment [16] . SimOS models the CPUs, memory systems, and I/O devices in sufficient detail to boot and run a commercial operating system. As a base CPU simulation model, we used the "*MIPSY* " CPU model which models a simple single issue RISC processor similar to MIPS R3000. In addition to augmenting REMARC functions, we also added a mechanism to count stall cycles due to load, multiply, and divide latency. This modification makes the CPU model more accurate.

4 Performance Evaluation

In this section we present the performance evaluation results of REMARC using realistic application programs.

4.1 DES Encryption

The Data Encryption Standard (DES) is one of the most important encryption algorithms and has been a worldwide standard for over 20 years. It is widely used to provide secure communication over the Internet. DES is also a good application for reconfigurable processors because it includes a lot of parallelism and irregular data movements which make a software implementation on conventional microprocessors difficult and inefficient.

Figure 10 shows outline of the DES algorithm. DES inputs a 64-bit plain text. After the initial permutation, there are 16 rounds of identical operations including the expansion permutation, XOR with the key, S-box table lookup, the P-box permutation, and XOR with the result of the previous round. The final permutation is performed to the 64-bit result of the sixteenth round.

We decided to divide the algorithm between the main processor and REMARC. The initial permutation and the final permutation are executed by the main processor and the 16 rounds of operations are executed by REMARC.

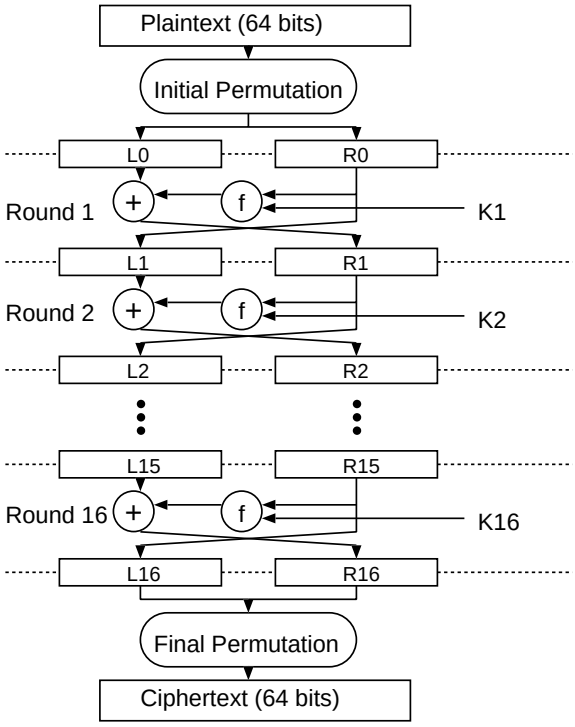

Figure 10: DES Encryption Algorithm

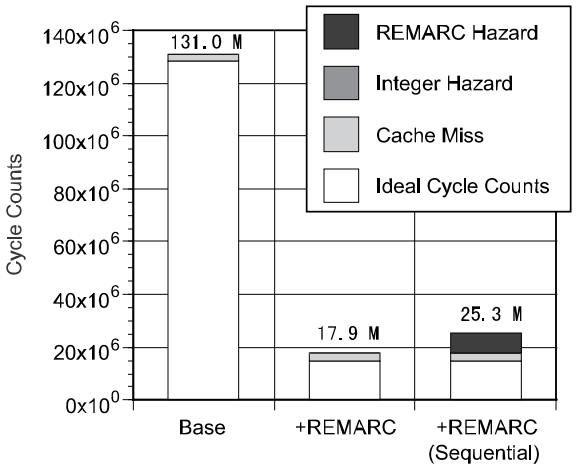

Figure 11: DES Encryption (1M Bytes) Results

Each row of nano processors executes two rounds. For instance, eight nano processors in the row 1 execute the first and second rounds and the row 2 execute the third and forth rounds. Therefore, the 16 rounds are pipelined into 8 stages. Although the nano processors in the same row perform different operations, the processors in the same column perform the same operations; therefore, VSIMD instructions can be used in the DES encryption program.

Figure 11 shows the result of the DES encryption of 1M bytes data. REMARC delivers a 7.3 times performance improvement. To optimize performance we introduced a software pipelining technique which can overlap operations in the main processor and REMARC. The sequential REMARC in Figure 11 disables software pipelining to stall the pipeline when the main processor reads the REMARC data registers during the executions of any *rex* instructions. This model decreases performance by 40%. The *rex* instruction for the DES encryption takes 51 cycles to execute. Software pipelining technique is required to hide the large latency of the *rex* instructions.

4.2 MPEG2 Decoding

MPEG-2 is one of the most popular video compression standards. The algorithm of MPEG-2 decoding consists of four stages, Variable Length Decoding (VLD), Inverse Quantization (IQ), Inverse DCT (IDCT), and Motion Compensation (MC).

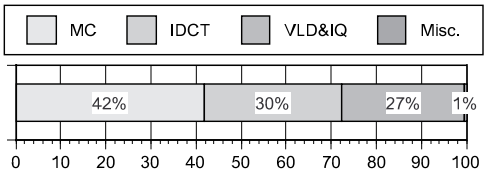

Figure 12: Execution Time Breakdown of MPEG2 Decoding

To determine the execution time breakdown we profiled the MPEG-2 decoding program by profiling tool "*Pixie*" [18] . Figure 12 shows the percentage of execution time spent in each stage while decoding 30 frames. The figure indicates that the execution time is spent in IDCT, MC, and VLD/IQ almost equally. This suggests that a reconfigurable logic needs to support multiple functions to accelerate the MPEG-2 decoding algorithm. For instance, if a reconfigurable logic only implements IDCT, maximum speedup is limited to a factor of 1.45. REMARC can support both IDCT and MC in the same configuration. This covers more than 70% of the execution time.

We used the *mpeg2decode*[20] program distributed by MPEG software simulation group as a base MPEG-2 decoding program. We defined and programmed REMARC global and nano instructions for IDCT and MC. The *mpeg2decode* program is modified using these REMARC instructions.

The REMARC instructions for MC are SIMD type instructions and very similar to multimedia instructions. They can operate 8 or 16 data at the same time using 8 or 16 nano processors.

The IDCT implementation is quite unique. We did not use the fast IDCT algorithms [21] because these algorithms have little regularity and are diffi-

cult to implement efficiently on architectures like REMARC. Instead, we used the IDCT algorithm which consists of two 4x4 matrix and vector multiplications. In this algorithm, each eight nano processors in the same row calculate the 1-D row IDCT. Then, each eight nano processors in the same column calculate the 1-D column IDCT. Because a 2-D IDCT is done in place, there is no need to execute matrix transposition. In this fashion an 8x8 nano processor array naturally implements a 2-D IDCT. The 2-D IDCT can be executed in 69 cycles by REMARC. Although a fast IDCT algorithm is not used, REMARC can realize a fast 2-D IDCT implementation which exploits the large amount of parallelism in the algorithm.

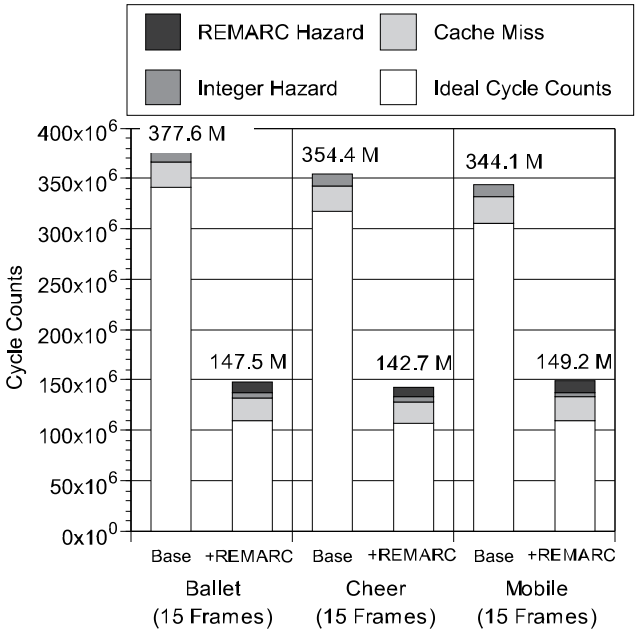

Figure 13: MPEG2 Decoding (15 Frames) Results

Figure 13 shows the execution time of the MPEG-2 decoding for 15 frames. Execution time is evaluated for the original single issue processor and the processor with REMARC. Three bit-streams *ballet*, *cheer*, and *mobile* are chosen as benchmarks. REMARC delivers speedups ranging from a factor of 2.31 to 2.56.

In Figure 13, cache miss stall cycles of the REMARC processor are almost the same as those of the original processor. Because the total execution cycles are reduce by half, the relative percentage of cache miss stall cycles becomes much larger in the REMARC processor. This suggests that memory systems including cache organizations are more important in the REMARC processor than the original processor. Some studies [17] show hardware prefetching schemes for data caches can reduce cache misses. Combination of the REMARC architecture and memory systems must be an interesting topic especially for stream based multimedia applications.

4.3 MPEG2 Encoding

We used the *mpeg2encode* [20] program also distributed by MPEG software simulation group as a base MPEG-2 encoding program. We began our effort by profiling the algorithm on a workstation as well as the MPEG-2 decoding.

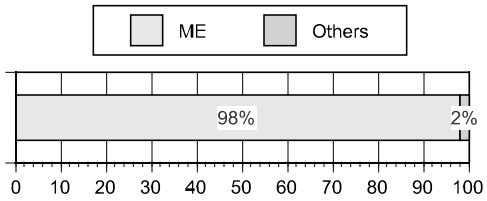

Figure 14: Execution Time Breakdown of MPEG2 Encoding

Figure 14 shows the execution time breakdown of the MPEG-2 encoding for 6 frames. For motion estimation we used a fullsearch algorithm and its search area is from -16 to +15 pixels for both X and Y directions. In this time we found 98% of the execution time is spent in motion estimation. Therefore we focused on accelerating motion estimation with REMARC.

We programmed four functions, full-pel frame search, full-pel field search, half-pel frame search, and half-pel field search, on REMARC. For the full-pel search we used a systolic implementation which is similar to some hardware implementations [22] . An 8x8 array of the nano processors is organized as an 1-dimensional systolic array. Macroblock size is 16x16 for the frame search and 16x8 for the field search. Because the size of the algorithm is larger than the number of the nano processors, we mapped four or two pixels to each nano processor.

For the first step of the full-pel search, current macroblock data are placed into each nano processor. Then, reference data are read from memory and broadcast to nano processors to calculate absolute differences with the current data. These absolute differences travel around nano processors and make sums of them in a systolic manner.

This systolic algorithm makes REMARC pipelined deeply, in 64 stages physically and 256 stages logically in case of the frame search. It takes 2048 (256 x 8) cycles to fill up the pipeline. Then, each sum of absolute differences is produced every 8 cycles. It can deliver more than a 100 times speedup for the kernel of the fullsearch routine.

Figure 15 shows the execution time of the MPEG-2 encoding for an I-picture (intra picture), a P-picture

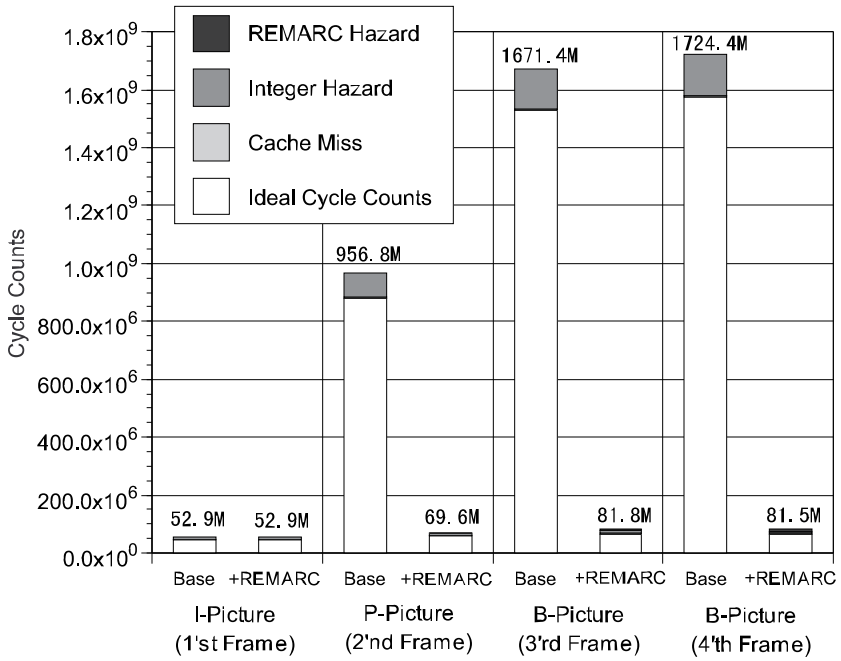

Figure 15: MPEG2 Encoding Results

(predicted picture), and two B-pictures (bidirectionally predicted pictures). We used the sample video "*ballet*" as a benchmark. Due to a limitation of the simulation time we used small size of pictures, 352 x 224 pixels. Execution time is evaluated for the original single issue processor and the processor with REMARC.

Because I-pictures are encoded without motion estimation, there is no performance improvement with REMARC. The REMARC processor achieves a 13.7 times speedup for the P-picture and 20.4 times and 21.2 times speedups for the B-pictures . As shown in Figure 15 REMARC also reduced integer hazard stall cycles mainly caused by load latency. In addition to maximizing its parallelism, the systolic algorithm can reduce the number of memory accesses dramatically using the local storages in the nano processors and the interconnections among them. These effects provide this significant performance improvement.

In this evaluation, to demonstrate REMARC's efficiency purely, we used almost same fullsearch algorithm shown in the original *mpeg2encode* program. In practice other motion estimation algorithms, such as hierarchical search algorithms [23], which reduce number of operations are going to be used. REMARC could also be used to accelerate these algorithms with similar performance improvements.

5 Conclusion

In this paper we have proposed a new reconfigurable coprocessor architecture REMARC that is suited for multimedia applications. Through the performance evaluations of the several realistic applications, we verified that REMARC could implement various algorithms which appeared in these multimedia applications. For instance, REMARC can implement SIMD type instructions similar to multimedia instructions like Intel's MMX for motion compensation of the MPEG-2 decoding. Furthermore, the highly pipelined algorithms, like systolic algorithms, which appear in motion estimation of the MPEG-2 encoding can also be implemented efficiently.

As the simulation results show, REMARC achieves speedups ranging from a factor of 2.3 to 21.2 for these applications.

Acknowledgment

We wish to thank Basem A. Nayfeh for his help with SimOS and Rachid Helaihel for his insightful comments. We also appreciate the help of Eiji Iwata with the programming of the MPEG2 decoding and encoding programs.

References

[1] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan, and Liang He, *"VIS Speeds New Media Processing"*, IEEE Micro, pp. 10–20, Aug. 1996.

[2] Alex Peleg and Uri Weiser, *"MMX Technology Extension to the Intel Architecture"*, IEEE Micro, pp. 42–50, Aug. 1996.

[3] Ruby B. Lee, *"Subword Parallelism with MAX-2"*, IEEE Micro, pp. 51–59, Aug. 1996.

[4] Kunle A. Olukotun, Rachid Helaihel, Jeremy Levitt, and Ricardo Ramirez, *"A Software-Hardware Cosynthesis Approach to Digital System Simulation"*, IEEE Micro, Vol. 14, pp. 48–58, Nov. 1994.

[5] Peter M. Athanas and Harvery F. Silverman, *"Processor reconfiguration through instruction-set metamorphosis"*, IEEE Computer, vol. 26, no. 3, pp. 11–18, Mar. 1994.

[6] André DeHon, *"DPGA-coupled microprocessor: Commodity ICs for the early 21st century"*, IEEE Workshop on FPGAs for Custom Computing Machines, 1994.

[7] Rahul Razdan, Karl Brace, and Michael D. Smith, *"PRISC Software Acceleration Techniques"*, IEEE International Conference on Computer Design, 1994.

[8] Ralph D. Witting and Paul Chow, *"OneChip: An FPGA Processor with Reconfigurable Logic"*, IEEE Symposium on FPGAs for Custom Computing Machines, 1996.

[9] John. R. Hauser and John Wawrzynek, *"Garp: A MIPS Processor with a Reconfigurable Coprocessor"*, IEEE Symposium on FPGAs for Custom Computing Machines, 1997.

[10] Scott Hauck, Thomas W. Fry, Matthew M. Hosler, and Jeffrey P. Kao, *"The Chimaera Reconfigurable Functional Unit"*, IEEE Symposium on FPGAs for Custom Computing Machines, 1997.

[11] Ulrich Schmidt and Sönke Mehrgardt, *"Wavefront Array Processor for Video Applications"*, International Conference on Computer Design, 1990.

[12] Dev C. Chen and Jan M. Rabaey, *"A Reconfigurable Multiprocessor IC for Rapid Prototyping of Algorithmic-Specific High-Speed DSP Data Paths"*, IEEE Journal of Solid-State Circuits, Vol. 27, No. 12, Dec. 1992.

[13] Alfred K. Yeung and Jan M. Rabaey, *"A 2.4GOPS Data-Driven Reconfigurable Multiprocessor IC for DSP"*, IEEE International Solid-State Circuits Conference Digest of Technical Papers, pp.108–109, 1995.

[14] Ethan Mirsky and André DeHon, *"MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources"*, IEEE Symposium on FPGAs for Custom Computing Machines, 1996.

[15] Gerry Kane, *"MIPS RISC Architecture"*, Prentice Hall, Englewood Cliffs, 1988.

[16] M. Rosenblum, S. Herrod, E. Withchel, and A. Gupta, *"The SimOS approach"*, IEEE Parallel and Distributed Technology, Vol. 4, No. 3, 1995.

[17] Daniel F. Zucker, Michael J. Flynn, and Ruby B. Lee, *"A Comparison of Hardware Prefetching Techniques for Multimedia Benchmarks"*, International Conference on Multimedia Computing and Systems, 1996.

[18] Michael D. Smith, *"Tracing with pixie"*, Technical Report No. CSL-TR-91-497, Computer Systems Laboratory, Stanford University, 1991.

[19] Bruce Schneier, *"Applied Cryptography: Protocols, Algorithms, and Source Code in C"*, 2nd edition, John Wiley and Sons, 1996.

[20] MPEG Software Simulation Group, *"mpeg2encode/mpeg2decode version 1.2"*, http://www.mpeg.org/MSSG/, July, 1996.

[21] W. H. Chen, C. H. Smith, and S. C. Fralick, *"A Fast Computational Algorithm for the Discrete Cosine Transform"*, IEEE Trans. on Communication, Vol. 25, pp. 1004-1009, Sept. 1977.

[22] Masayuki Mizuno et. al., *"A 1.5W Single-chip MPEG2 MP@ML Encoder with Low Power Motion Estimation and Clocking"*, Digest of Technical Papers, ISSCC'97, pp.256-257, 1997.

[23] V. Bhaskaran and K. Konstantinides, *"Image and Video Compression Standards"*, Kluwer Academic Publishers, 1995.