An Infrastructural IP for Interactive MPEG-4 SoC Functional Verification

Trio Adiono[1], Hans G. Kerkhoff[2] & Hiroaki Kunieda[3]

[1]Institut Teknologi Bandung, Bandung, Indonesia
tadiono@paume.itb.ac.id
[2]MESA+ Institute for Nanotechnology, 7500AE Enschede, the Netherlands
h.g.kerkhoff@utwente.nl
[3]Tokyo Institute of Technology, Tokyo, Japan
kunieda@vlsi.ss.titech.jp

Abstract: This paper introduces a specific architecture including an infrastructural IP for functional verification and diagnostics, which is suitable for functional core-based testing of an MPEG4 SoC. Our advanced MPEG4 SoC results in a high complexity SoC with limited physical access to many different functional cores. The proposed test method provides direct monitoring and control for each core, which enables core verification at actual speed. It significantly decreases the verification time due to the large number of required test vectors in typical MPEG4 verification. Furthermore, it also makes the system scalable for functional core expansion due to upgrading of standards. The proposed infrastructural IP is also linked to PC-based interactive tools to simplify the verification of individual and integrated cores. It also provides detailed diagnostic data that enables simple system debugging. The debugging tools also feature test-pattern generation and simulation of expected values. Actual system implementation has shown full functionality of our proposed method.
Keywords: functional MPEG-4 verification; infrastructural IP; SoC testing.

1. Introduction

Advances in MPEG4 video standard development has resulted in complex SoCs which features a high logic density and a large number of pins. They consist of many dedicated hardware processing cores [1-3] such as a Discrete Cosine Transform (DCT), Inverse DCT (IDCT), Variable Length Coding (VLC), Quantizer (Q), inverse-Quantizer (IQ), Motion Compensator (MC) and Motion Estimator (ME) etc., which possess many different functions. Although large numbers of pins are required, the designer still has very limited access to each processing core inside the SoC.

In order to optimize chip-level verification, debug and program chip functionality, embedded core logic is incorporated in this design as an embedded infrastructural IP. Incorporating an embedded infrastructural IP provides the physical access to all processing cores inside the SoC, which surpasses the limitation of pin numbers, helps silicon debugging, improves test quality and increases the manufacturing yield. Furthermore, it also enables further functional development extensions to the device functionality, in order to cope with the latest advances in the video coding standard.

The utilization of a standard testing approach such as IEEE 1500 [4] has a limitation in providing a large number of test patterns. MPEG processing requires very specific test-pattern data, which usually consist of two- dimensional datas, which have continuity in the time domain. In addition, a significant amount of memory should also be provided to store the processing result. To preserve external compatibility with IEEE 1500 and 1149.1, a standard TAP controller is used as access mechanism for our approaching, in combination with a dedicated infrastructural IP.

Received: August 10, 2009. Accepted: September 15, 2009

This paper is organized in the following way. Section II will present the verification methodology proposed in this paper. In Section III, experiments are conducted to evaluate the effectiveness of the proposed verification methodology. Finally, a summary is provided in Section IV.

2. Verification Methodology

2.1 Architecture System

Hardware-software co-design has been employed to build an integrated system for functional core verification in an MPEG4 SoC as shown in Fig.1. Most of MPEG4 processing is performed with respect to a 16 x 16 pixels input image data (macro-block); the dedicated hardware processing core to perform this operation is called Macro-block Processing Unit (MPU). A typical MPU core is a DCT, IDCT, Q, IQ, VLC, IVLC, ME and MC. As seen in Fig. 1, the light box labelled "cores to be tested" is shown in the SoC as many MPUs those are connected to a single bus.

In the infrastructural IP part, the Core Test Interface (CTI) handles the verification data communication between the SoC and a host computer. Based on instructions and test patterns generated by the host computer, the Core Test Processor (CTP) inside the SoC controls the verification process for each functional core. Both the CTI and CTP are embedded in SoC as an infrastructural IP (I^2P).

The test patterns are generated by the diagnostic tools. The test pattern generation depends on the core that is going to be tested. The test pattern can be an original image sequence, bit-stream data, or a special test vector for a functional core that is generated from simulations. Therefore, the diagnostic tools have the capability to simulate all of the functionality of a core. They also have the same computational algorithm to ensure the same computational result between the simulation and hardware processing result.

The test pattern that is generated by diagnostic tools is downloaded into the Frame Store Memory (FSM) of the SoC. During actual operation of the SoC, the FSM is used for storing image frame data. Utilizing available system memory of the SoC reduces the additional hardware for test-pattern data storage.

However, to assign the MPEG SoC to process a test vector at a certain location, one needs to have a core test processor (CTP) as part of an infrastructural IP inside the SoC. This processor also has a programmability function in order to support all processing core testing. Therefore, before starting the testing, the diagnostic tools have to download data in the program memory of the CTP.

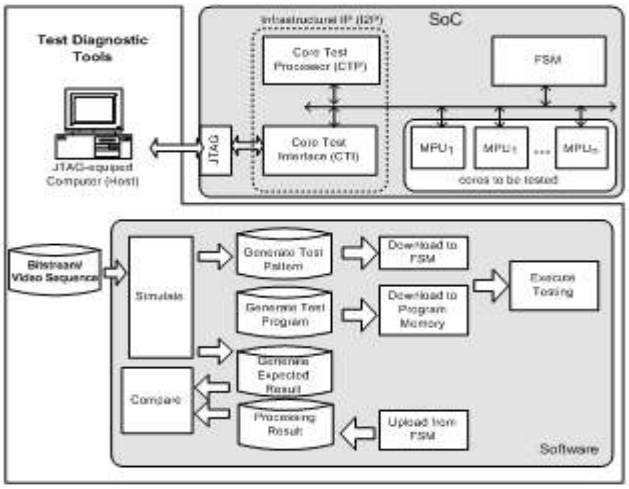
The CTP is also designed for being able to be controlled by the diagnostic tools. Communication between the diagnostic tool and the SoC is implemented via the IEEE TAP controller [4] circuit. This controller is used because a small number of pins are required (3) and relatively little additional logic in the SoC for interfacing. Moreover, it can also be used during structural testing of (wrapper-equipped) cores. Through this interface, the host processor can start, stop, and configure cores with special processing parameters.

Similar to the case of actual signal processing, the verification result data is stored inside the FSM. In the final stage, diagnostic tools can upload result data into the host computer, and carry out comparisons with simulated results. With the programmability features, one may configure the SoC for many different cases of verification, which result in diagnostic capabilities of the system.

2.2 The Core Test Interface (CTI)

The CTI handles the communication between CTP and host computer. Through this interface and the TAP controller, the host computer performs functional verification of the SoC by Read/Write (R/W) access to the registers of the CTP, Program Memory (PM) and FSM. By setting the register values of the CTP, the host computer can execute the verification program inside the PM. The test-bench data for verification is also written by the host computer

through this CTI. By accessing the PM, the host computer can replace the test program inside the PM for another verification procedure anytime. As consequence, the amount of testing is only limited by the PM size.



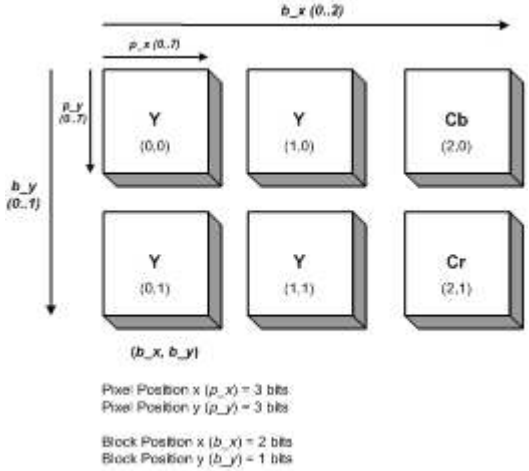
Functional verification system architecture.

The CTI consists of address decoding and data multiplexer circuits. They determine which address to be R/W by the host computer.

As the construction of a typical SoC system consists of many dedicated hardware MPUs, system control can be done through registers. Registers can be used for dedicated enable and reset of each core. In the other case, it can also be used for system configuration, such as image size, data scheduling etc.

2.3 The Core Test Processor (CTP)

The Core Test Processor is designed for applying test pattern data to each Macro-block Processing Unit (MPU). Therefore, the CTP must be able to access memory data inside the FSM according to each MPU data input-output scheduling. As a video-based MPU typically accesses the data in terms of pixels, in block or macro-block form, the CTP has been designed to be programmable for this type of data scheduling. As shown in Fig. 2, the CTP can access data in the scheme that also follows the MPEG input data format.



The memory data access scheme.

As defined in the MPEG4 standard [5], the input image data is presented in CIF format. In CIF format, image data is subdivided into a macro-block (16 by 16 pixels) and a block (8 by 8 pixels). To represent a 16 x 16 pixel image data, each macro-block data consist of four blocks of luminance data (Y) and two blocks of chrominance data (Cr and Cb).

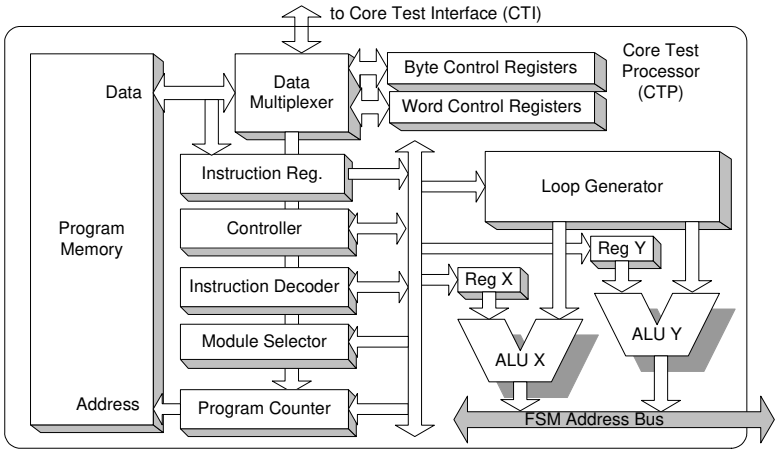
2.3.1 CTP Architecture

In order to generate the FSM address of an MPEG-4 core, a Core Test Processor (CTP), as part of an embedded infrastructural IP in MPEG4 SoC system, is designed as shown in Fig. 3. The CTP is designed as a programmable processor that generates an address by executing instructions in a Program Memory (PM). The program is designed to apply a test pattern and perform verification inside an MPEG-4 core. The executed instruction inside the PM is determined by the Program Counter register (PC). In order to generate the sequence of an address for memory R/W operation, the dedicated Loop Generator (LG) is designed. Using this unit, pixel, block or macro-block memory data R/W operations become possible. The LG generates the address sequence with respect to the zero position (Fig. 2). In order to randomly access a certain pixel, block or macro-block position, an offset address should be added with reference to the pixel, block or macro block position that is stored in the registers (REG_X and REG_Y). For this purpose, ALU_X and ALU_Y are used to add REG_X and REG_Y to the address value generated by LG. X and Y are separated, in order to easily access the data block in a two-dimensional position.

Using the Module Selector register, one can determine which core is going to be tested. Each functional core has a unique core number.

As can be seen in Fig. 3, the registers have been divided into two groups, which are byte- and word-oriented registers. With byte registers, one can have registers with much data addressing capability (2^{12}). On the other hand, a word register is designed for large amounts of data, with a small number of registers (maximum is 2^4).

Control signals inside the CTP are generated by the controller. It operates in 3 stages, Instruction Fetch (IF), Decoding (DEC) and Execution (EXE). In the IF stage, the instruction pointed by the program counter is transferred from the program memory into the instruction register. Start and stop of the CTP Controller unit is controlled by a RUN register value inside the Byte Control Registers.



The Core Test Processor architecture.

In the DEC stage, the data from the instruction operand value is extracted. This operand value is used for the CTP configuration which is dependent on the instruction type. For an instruction type related to memory access, which will be explained in next section, this operand

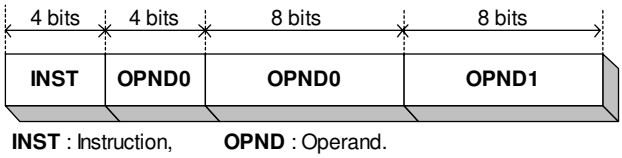
value is used to configure the maximum loop of the loop generator and Reg X and Reg Y. For an instruction related to program control, the operand value is used to determine the program counter value. For a function specific instruction, the operand value is used to set a specific register value, such as the module register or motion vector register.

In the EXE stage, the operation depends on the instruction. In a memory-access type instruction, such as IRD_LINE, IRD_COLUMN, IWR_LINE and IWR_COLUMN, the loop generator performs a looping operation to generate an address according to the maximum loop value defined in the instruction operand data. The amount of required clock cycles for this execution stage depends on the amount of generated addresses. Since in the rest of the instruction the operation is only to set the register value, the designated register value will be set in this stage within a clock cycle.

In order to inform the host computer with regard to the completion of the processing, an interrupt controller is used, where the host computer can detect its status by reading the interrupt controller value. Therefore, the host computer needs to observe this value after setting the run registers to a high value.

2.4 Instruction Sets

Instruction sets are designed in order to provide flexibility with regard to the application of test patterns to the MPU and perform verification. As consequence, three instruction types can be distinguished: *Register Access, Memory Access, and Program Control*. An instruction consists of three bytes of data as shown in Fig. 4.



The instruction format.

The complete list of instructions is shown in Table 1.

2.4.1 Memory Access Instructions

These types of instructions Read/Write the data from the frame store memory according to the instruction's operand value. Using the instructions IRD_LINE / IWR_LINE or IRD_COLUMN / IWR_COLUMN one can Read/Write the frame store memory data in horizontal (LINE) or vertical (COLUMN) direction; one can also perform pixel, block or macro-block data access as described in Fig. 2 by setting the **Δ** value of the instruction operand. The Δ*px* and Δ*py* operand values determine the size of the group of pixels to be read in *x* and *y* direction. As consequence, if one sets Δ*px* and Δ*py* to the maximum value (Δ*px=7,* Δ*py=7),* one will access those pixels as block of data. As another example, if one sets Δpx=0 and Δpy=7, it will read/write a line of 7 pixel data in y direction.

In the same way, Δ*bx* and Δ*by* operand values determine the amount of block data to be read/write. And also if one sets them to the maximum value (Δ*bx=2,* Δ*by=1),* it will access a macro block of data. The same case is also applied to the Δ*mbx,* Δ*mby* operand values. One can set them to access the data in macro-block format.

Different from the above read/write instructions, the ISET_ADD and IINC_ADD instructions are used to manipulate the reference address position for pixel, block or macro-block data read/write. The ISET_ADD instruction sets the reference address to its operand value which consist of pixel (px, py), block (bx, by) and macro-block position (mbx, mby). Therefore, one can set the reference position of data to be read inside a frame.

Instruction IINC_ADD increases the current address according to its operand value. Increment is done independently among px, bx, and mbx. Second complement representation

is used for operand representation. Therefore, using this instruction one can also decrease the address value. Different to ISET_ADD, this instruction is designed for relative address setting according to the current position. Usually it is useful for address setting inside a loop condition.

Instruction sets of the Core Test Processor. *fno* : frame number. Δ*mbx,* Δ*mb:* number of macro blocks to be R/W. Δ*bx,* Δ*by:* number of blocks to be R/W. Δ*px,* Δ*p*: number of pixels to be R/W.

Instruction name & Operand
Memory Access Type
IRD_LINE(Δ*mbx,* Δ*bx,* Δ*px,* Δ*mby,* Δ*by,* Δ*py*)
IWR_LINE(Δ*mbx,* Δ*bx,* Δ*px,* Δ*mby,* Δ*by,* Δ*py*)
IRD_COLUMN((Δ*mbx,* Δ*bx,* Δ*px,* Δ*mby,* Δ*by,* Δ*py*)
IWR_COLUMN((Δ*mbx,* Δ*bx,* Δ*px,* Δ*mby,* Δ*by,* Δ*py*)
ISET_ADD(*fno, mbx, bx, px, mby, by, py*
IINC_ADD((*fno, mbx, bx, px, mby, by, py*)
Program Control
IJUMP(*dest_add*)
ICALL(*dest_add***)**
IRTN
IWAIT(*n***)**
ISTOP
Function Specific
ISET_MOD(*module_no*)
ISET_MV(*mvx***,** *mvy***)**

2.4.2 Program Control Instructions

Several instructions are designed to control the program execution flow, such as IJUMP, ICALL, IRTN, IWAIT and ISTOP. The IJUMP instruction is used to execute the instruction in the destination address (dest_addr). By this instruction one can perform a loop operation or share several program routines for different functionality. Beside IJUMP, we also provide the ICALL instruction for execution of a subroutine program. This instruction helps to reduce the number of instruction lines by making many program routines for the subprogram that is frequently used. In order to return to the instruction after the ICALL instruction, the IRTN instruction is used.

The instruction ISTOP is used to terminate the program execution. Therefore, this instruction must exist at the end of every program. Finally, the instruction IWAIT(n) delays for n clocks before executing the next instruction. Usually this is used between read and write data from a module. The parameter n is bit [15...0] representing the number of clock cycles to wait before executing the next instruction.

2.4.3 Function Specific Instructions

This type of instruction is used for dedicated MPEG functionality instructions. Such as the ISET_MV instruction, which is used for setting the motion vector value during the testing of the Motion Compensation operation in MPEG-4 processing.

The ISET_MOD(*module_no*) instruction is used to select an active module. We have assigned a specific module number to each MPU. This instruction must be executed before reading data for each module.

3. System Simulation

In order to verify the embedded IP design via simulations, we have constructed the system using Synopsys design ware IP as shown in Figure 5. In this configuration, the host system is replaced with Synopsys Serial IP for verification. Using this IP one can perform data transmission and receive via the SoC JTAG interface. Therefore, this system will replace the host PC and diagnostic tools (fig. 1). Programming features provided by IP Verification, one can transfer generated test vectors directly, expect data and program memory data into this system. Therefore, the previous system can be simulated using this system.
In order to reduce data transfer time and to simplify the simulations, the test patterns are loaded into the FSM by using the memory model design ware from Synopsys. With these memory model features, one can load the test patterns in short time and perform comparisons with the processing result.

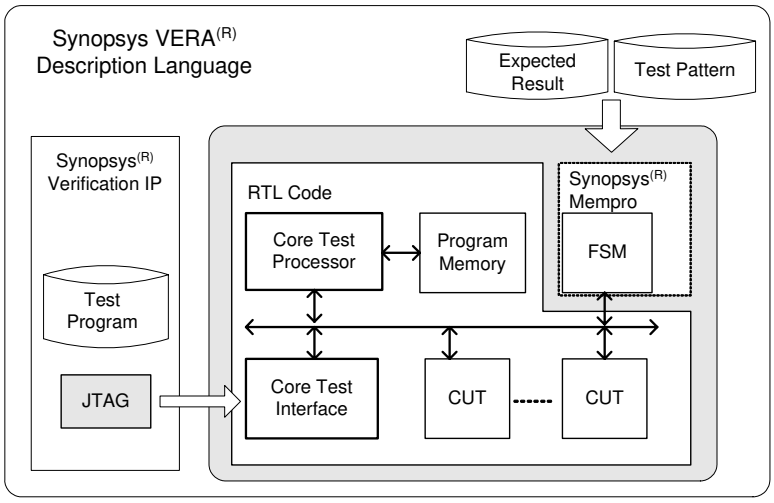
4. Diagnostic Tools

Diagnostic tools have several functions in the verification process, which are test pattern generation, SoC processing control, processing simulation and data comparison.

4.1 Test Pattern Generation

Depending on the core that is going to be tested, the diagnostic tool can provide the test pattern data in original image/bit-stream or a result of certain processing simulation. As example to verify the IDCT function, the diagnostic tool must perform the DCT function to generate DCT coefficient data as test pattern. Therefore, all processing inside the SoC system must be implemented in the debugging system. One must also consider the similarity of processing between hardware and software simulation. Especially in the case for processing such as the DCT that may be giving different results depend on the different ways of computation.

Beside the test vectors, the diagnostic tools also generate expected result data. This can be accomplished by simulating the input data with the simulation program inside the diagnostic tools.



System verification using Synopsys DesignWare.

4.2 SoC Processing Control

Since all testing mechanisms are controlled via register read and write operations, the management of the testing process can be implemented as interactive application by attaching a

register read/write function. As result, one can write or read all the available registers and memory inside the SoC via the diagnostic tools.

One can also combine several mechanisms as a specific program subroutine with a certain function. For example, writing memory data actually consists of generating data writing instruction and generating a block of data. Inside the diagnostic tools, we have generated this as special function; therefore, the user usually does not recognize each function step.

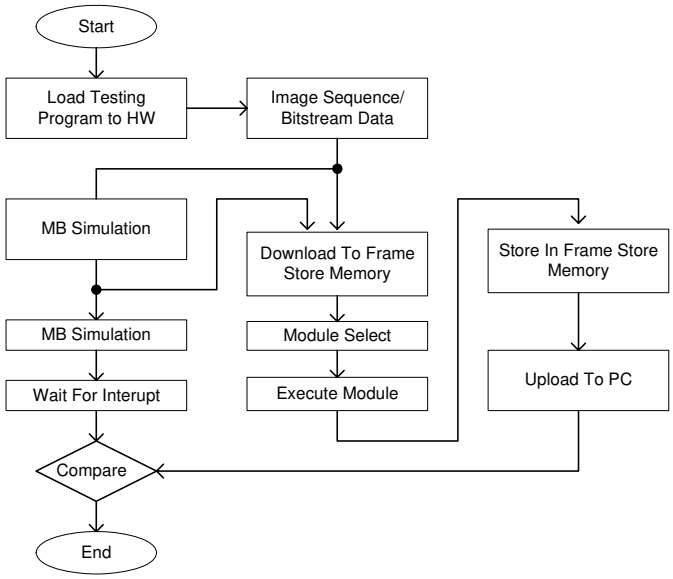
This program is also equipped with an interactive function, where the program automatically checks the interrupt register for determining the end of data processing.

The diagnostic tools can also use a standard frame video sequence data as input. Originating from this data, many other test vectors can be generated. With the instruction mechanism, one can easily transfer or read blocks of data. One can observe the results either from a text-based or graphical console (Figures 7 and 8).

4.3 The Software Verification Flow

Verification can be performed using the flow as shown in Fig. 6. At the beginning of verification, a dedicated testing program is downloaded from the PC to the CTP program memory. Afterwards, the input of the encoding and decoding process, video sequence or bit-stream data is downloaded into the diagnostic tools. Depending on the MPU to be tested, the video sequence or bit-stream data is directly loaded into the FSM or, if the MPU requires pre-processing, the processing is performed by the simulator inside the diagnostic tools.

In order to select the module to be verified, the module selector register is set according to the module being tested. After that, the register is set to execute the designated program inside the program memory. The processing result is stored inside the FSM for uploading to the PC. At the same time, the diagnostic tools simulate the same processing. The control unit inside the CTP issues an interrupt after each testing program has been completed. Both simulated and in-circuit processing results are compared in the PC. The result can also be visually compared and displayed by the diagnostic tools, as shown in Fig. 8.



The verification software flow.

If the bug is found, diagnostics can be performed by applying customized test patterns. Customization can be done in many ways depending on the tested MPU. For instance for MC diagnostics, one can input the same pixel value for checking the core functionality.

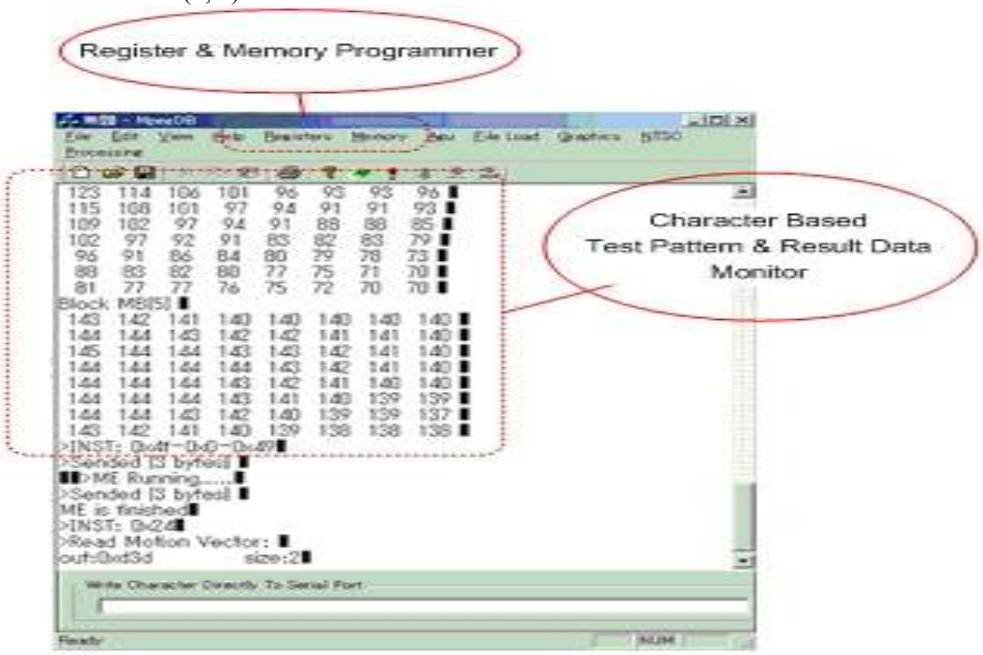
5. Experimental Results

In order to carry out experiments, we have developed an MPEG-4 SoC system, and performed verification by means of simulation as well as in-circuit. Simulation is carried out by using the system explained in section 2.3. Additionally, for in-circuit verification, we implemented an MPEG-4 SoC system using an FPGA as shown in Fig. 9. This system was built using an ALTERA FPGA chip and several chips for the frame store memory, data converters and other external interfaces.

The proposed system was developed for checking DCT, IDCT, Quantizer, Inv-Quantizer, ME, and MC cores. DCT testing is done by applying block image data into the FSM using the IWR_LINE instruction. The IWAIT instruction is used to wait for completion of the DCT processing. Afterwards, the processing result data is written into the FSM using the IWR_COLUMN instruction. The IWR_COLUMN instruction is used instead of IWR_LINE, due to the orientation change because of this processing. As a result, the DCT verification code is as follows:

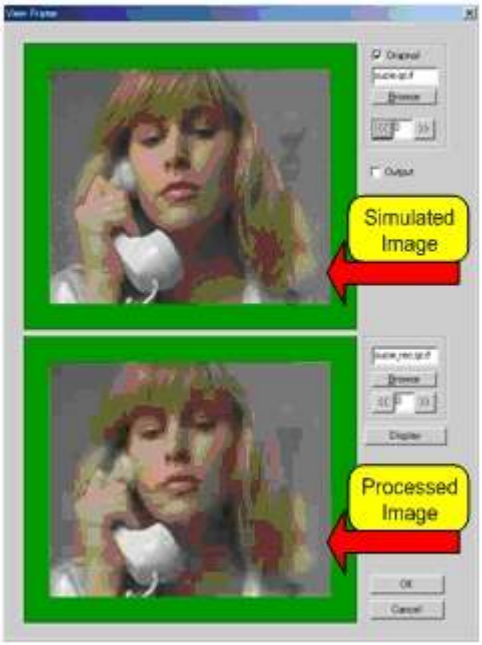
```
ISET_MODULE(01h)          // Select DCT module
ISET_ADD(0;5,1,0;4,2,0)   //Set Frame No=0, Macro-block=5,4 and Block
                          number(1,2)
IRD_LINE(0,0,7;0,0,7)     // Read a block of data from FSM
IWAIT(72)                 // Wait for 72 clocks
ISET_ADD(1;5,1,0;4,2,0)   //Set Frame No=1, Macro-block=5,4 and Block
                          number(1,2)
IWR_LINE(0;5,1,0;4,2,0)   // Write transformed data to FSM
ISET_MOD(00h)             // Release DCT module
ISTOP                     // Stop the execution
```

The above program performs the DCT operation to a block of data in frame 0, macro-block (5, 4) and block number (1, 2). The result of processing is stored in frame 1, macro-block (5, 4) and block number (1, 2).



Character-based visualization of the verification tool.

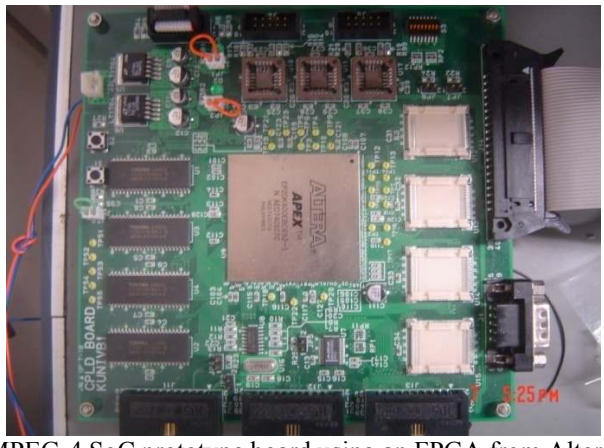
Almost in a similar way, one can perform the verification for the IDCT, Q and Inv-Quantization cores. As a result of DCT & Q processing, we can see the reconstructed image as illustrated in Fig. 10. It shows that the verification process can be done for individual DCT and IDCT functional verification.



Graphical-based visualization of the verification tool.

In the case of MC testing, one can use any frame image from a sequence. The data is read in the macro-block by using the same instruction as before, but with different operand for setting the image size into a macro-block. Afterwards we apply the complete range of MV values (typically between –15.5 to 15.5 in both x and y direction) using the ISET_MV instruction. As a result one can see that the macro-block data is shift according to the input motion vector value.

The ME unit usually has a local memory for internal system cache. For copying the test pattern into this cache data, the IRD_LINE instruction can also be used. After that, the ME unit can be started, and the result can be read using the Register Access instruction.



MPEG-4 SoC prototype board using an FPGA from Altera.

Experimental result for DCT/IDCT verification; the quantization value is five.

6. Conclusions

An infrastructural IP has been proposed to support cost-efficient debug, and board and system level functional test. Programmability of the system offers the flexibility of upgrading and fix bugs. Diagnostic tools on a PC are used to manage test patterns, and allow system reconfiguration for specific processing core tests. Verification can be done interactively and executed processing at actual system speed. It solves the problem of requiring large amounts of test patterns as required by MPEG-4 SoC system verification. The same system configuration can be easily verified in RTL description and chip level.

References

[1] M. Ohashi., et al. A 27MHz 11.1mW Video Decoder LSI for Mobile Application, *IEEE ISSCC Digest of Technical Papers*, February, pp. 366, 2002.

[2] H. Nakayama et al., An MPEG-4 Video LSI with Error-Resilient Codec Core based on a Fast Motion Estimation Algorithm, *IEEE ISSCC Digest of Technical Papers*, pp. 368, February, 2002.

[3] M. Takahashi, et al., A Scalable MPEG-4 Video Codec Architecture for IMT-2000 Multimedia Applications, *Proceedings of IEEE ISCAS 2000*, Geneva, Switzerland, pp. 188-191, 2000.

[4] http://standards.ieee.org/announcements/pr_ics.html

[5] International Standard, Information technology — *Coding of audio-visual objects*, ISO/IEC 14496-2.

[6] Y. Zorian., Guest Editor's Introduction: What is Infrastructure IP?, *IEEE Design & Test of Computers*, **19**(3), pp. 5-7, 2002.