ESL flow for a hardware H.264/AVC decoder using TLM2.0 and high level synthesis: a quantitative study

Article *in* Proceedings of SPIE - The International Society for Optical Engineering · May 2009

DOI: 10.1117/12.821647

CITATIONS
2

READS
82

5 authors, including:

Pedro P. Carballo
Universidad de Las Palmas de Gran Canaria
46 PUBLICATIONS **99** CITATIONS
SEE PROFILE

Pedro Hernández Fernández
Universidad de Las Palmas de Gran Canaria
7 PUBLICATIONS **7** CITATIONS
SEE PROFILE

Gustavo Marrero Callico
Universidad de Las Palmas de Gran Canaria
161 PUBLICATIONS **1,031** CITATIONS
SEE PROFILE

Antonio Nunez
Universidad de Las Palmas de Gran Canaria
141 PUBLICATIONS **468** CITATIONS
SEE PROFILE

ESL flow for a hardware H.264/AVC decoder using TLM-2.0 and high level synthesis: a quantitative study

M. Thadani*[a], P. P. Carballo [a], P. Hernández [a], G. Marrero [a], A. Núñez [a]
[a]Institute for Applied Microelectronics (IUMA), University of Las Palmas of Gran Canaria, 35017 Las Palmas de Gran Canaria, Spain

ABSTRACT

The present paper describes an Electronic System Level (ESL) design methodology which was established and employed in the creation of a H.264/AVC baseline decoder. The methodology involves the synthesis of the algorithmic description of the functional blocks that comprise the decoder, using a high level synthesis tool. Optimization and design space exploration is carried out at the algorithmic level before performing logic synthesis. Final, post-place and route implementation results show that the decoder can operate at the target frequency of 100 MHz and meet real time requirements for QCIF frames.

Keywords: ESL, SystemC, TLM-2.0, high-level synthesis, logic synthesis, FPGA

1. INTRODUCTION

ESL design methodologies based on the SystemC language, high level synthesis and TLM (transaction level modeling) have become increasingly important in the electronic design industry due to increasing design complexity. High level synthesis coupled with the SystemC language allows hardware description at a higher level of abstraction than RTL (Register Transfer Level) languages such as VHDL, allowing designers to be more productive. In an ESL flow, the high level synthesis tool handles the conversion of the algorithmic description of the hardware to the RTL description, taking care of implementation details[1]. Algorithmic and transaction level modeling provide a system-level view that allows fast and early architectural exploration and design, and design optimization.

In the work hereby reported, an ESL design methodology is established and applied to create a synthesizable SystemC model of a baseline H.264/AVC decoder. The decoder is aimed at low-end mobile and handheld devices using the QCIF frame format, and the synthesizable model has to meet specific area and speed requirements, for a hardware implementation on a Virtex-4 FGPA. The established ESL flow includes transformation, modeling and refinement steps as front, middle and back-end steps respectively. The starting point for the research is a C++ model of the decoder, divided and organized into independently verified functional blocks (hardware-oriented), which were obtained through appropriate transformations from the reference C++ application software.

As a first step, a transaction level model of the decoder is created, using the TLM-2.0 standard. This is a high level functional model in which inter-block communication involves the exchange of transaction objects using TLM-2.0 interfaces. The TLM-2.0 model is verified using the same test cases applied to the C++ model and is used as the reference model for the subsequent micro-architecture exploration and RTL implementation.

Each of the decoder functional blocks is refined, implemented and verified independently. The refinement methodology employed involves a modeling phase, high level synthesis, optimization and logic synthesis.

In the modeling phase, the code is modified for an initial high level synthesis in the SystemC editor. Among the changes made is the replacement of the TLM-2.0 socket interfaces with SystemC signal channels, creating a pin-level interface, and the addition of timing information using wait() statements.

High level synthesis is performed next using Agility Compiler, in order to obtain an area and delay estimation summary. In order to determine if the results are satisfactory at this stage, initial reference values are set for the target operation frequency and for the FPGA area consumption, depending on functional block complexity and other factors.

*mthadani@iuma.ulpgc.es; phone +34 928 451229; fax +34 928 451083

VLSI Circuits and Systems IV, edited by Teresa Riesgo, Eduardo de la Torre, Leandro Soares Indrusiak,
Proc. of SPIE Vol. 7363, 73630K · © 2009 SPIE · CCC code: 0277-786X/09/$18 · doi: 10.1117/12.821647

If the area and delay results obtained do not meet the requirements, an iterative process involving the optimization of the SystemC code in the editor and the execution of Agility Compiler to obtain area and delay results is performed. Once the synthesis results obtained with Agility Compiler are satisfactory, logic synthesis is performed next using Synplify Pro, leading to a further improvement in area and delay results.

After obtaining logic synthesis results, a final implementation using Xilinx tools is carried out to produce the FPGA programming information, obtaining final area and delay results. Full decoder performance is estimated using these results, and the decoder would be capable of processing more than 30 frames per second for a QCIF frame size at a maximum operating frequency of 100 MHz.

The paper is organized as follows. Section 2 describes the algorithmic C++ model of the decoder, while section 3 describes how a TLM-2.0 model of the decoder is obtained from the C++ model. Section 4 describes the methodological steps followed to implement each of the functional blocks, and section 5 presents results obtained in the application of the design methodology. Finally, section 6 concludes the paper.

2. ALGORITHMIC C++ MODEL OF THE DECODER

As discussed in the previous section, the starting point for the research was a C++ model of the H.264/AVC decoder, divided and organized into independently verified functional blocks, obtained from the reference C++ application software (JM version 12.4)[2].The code extracted from this software has been modified and optimized, and any additional code necessary added, as required. The C++ model of the decoder adheres to the H.264/AVC Baseline profile, however, only one reference frame is used (the previous frame) and flexible macroblock ordering, ASO (Arbitrary Slice Ordering) and slice groups are not supported.

Figure 1 shows the H.264/AVC decoder. The H_CONTROL block performs the hardware controller function in the decoder. It activates the CAVLD block in order to read the syntax elements pertaining to the current macroblock from the bitstream. The I_QUANT (Inverse Quantization) and I_TRANS (Inverse Transform) blocks are activated next in order to obtain the dequantized coefficients and the residual macroblock, respectively. The INTER_P (Inter Predictor) or INTRA_P (Intra Predictor) block will be activated next depending on whether the samples of the current macroblock were obtained using temporal (inter macroblock) or spatial prediction (intra macroblock). Finally, the D_FILTER (Deblocking Filter) block will be activated if filtering is enabled for the current macroblock. A decoded frame (F'n) is obtained once all the macroblocks belonging to a frame have been filtered[3].

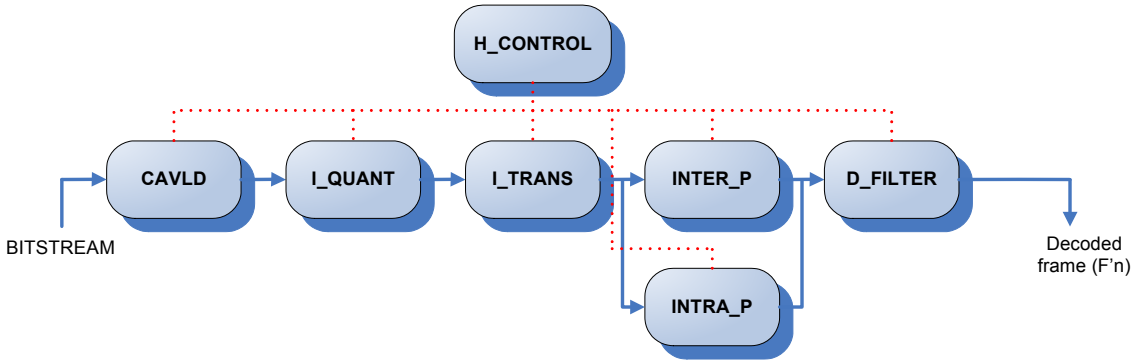

Fig. 1. H.264/AVC decoder

The C++ model features the functional blocks which appear in figure 1: H_CONTROL, CAVLD, I_QUANT, I_TRANS, INTER_P, INTRA_P and D_FILTER. In this model of the decoder, given the high level of abstraction, the H_CONTROL block performs control tasks by invoking the C++ functions associated to the other functional blocks sequentially. The decoder functional blocks work at the macroblock level, and therefore these functions are invoked once per macroblock.

In order to decode a macroblock, data needs to be transferred between functional blocks. In the C++ model of the decoder, a global data structure is used for data transfer. This is a C++ custom data structure which is shared by all the functional blocks that comprise the C++ model of the H.264/AVC decoder, and is used for data transfer between functional blocks (Fig 2).

In order to access the global data structure, each of the functional blocks that comprise the decoder (except the H_CONTROL block) have functions to load data (load results) from and store data (store results) in the global data structure. The load results function is used to load the subset of the input fields of the global data structure required by the functional block in question into an input data structure. The store results function is used to store the output data generated by the block, contained in an output structure, in the corresponding output fields of the global data structure. The fields of the input and output data structure are of type integer. Apart from the load and store functions, the functional blocks feature a function which performs the block's main function, reading input data from the input data structure and storing output data in the output structure. This function will invoke any other additional functions required by the functional block to perform it's tasks.

Therefore, in order to invoke a decoder functional block, the H_CONTROL block first invokes the load results function to load the input data from the global data structure. Then it invokes the corresponding block function which will read input data from the input structure and store output data in the output structure. Finally, the H_CONTROL block will invoke the store results function to store output data in the global data structure. This makes the output data available to other functional blocks. For example, once the H_CONTROL block has invoked the CAVLD block, the dequantized coefficients will be available to the I_QUANT block in the global data structure.

Macroblock neighbour data is required by certain functional blocks and such data is managed by the H_CONTROL block and is stored in the global data structure before the block is invoked, but no load or store functions are used for this purpose. For example, for a given macroblock, the INTER_P block requires motion vectors belonging to the neighbour to the left, neighbour above, neighbour above and to the left and neighbour above and to the right. The H_CONTROL block must make this data available to the INTER_P block in the global data structure before this functional block is invoked.

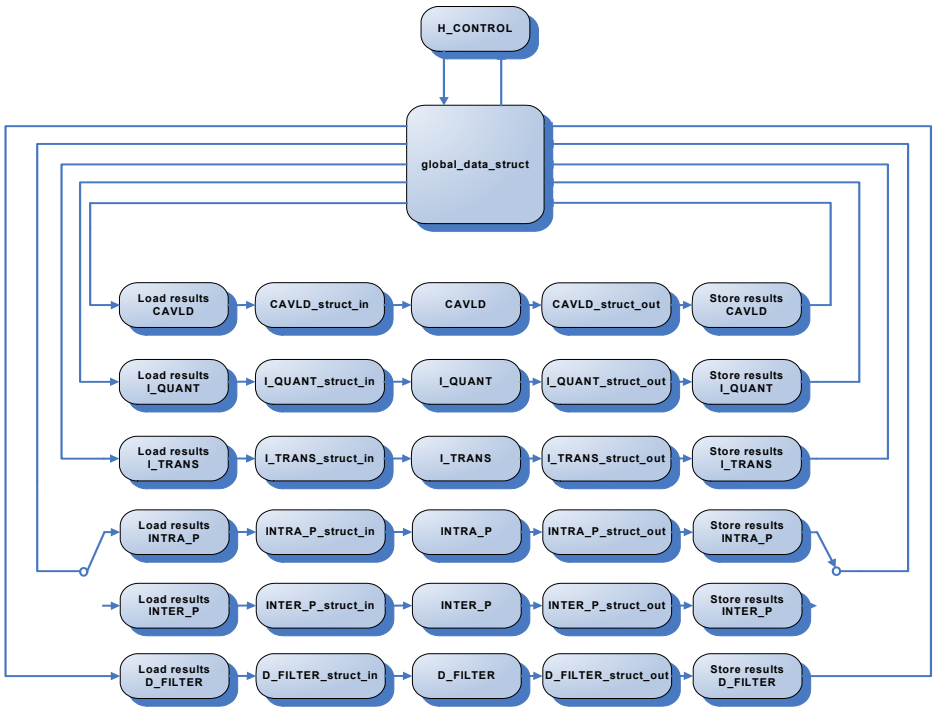

Fig. 2. Data transfer scheme used in the C++ model of the H.264/AVC decoder

Additionally, sample data is required by the INTRA_P, INTER_P and D_FILTER blocks. Memories are implemented in the C++ model of the decoder as 2-dimensional arrays (height and width), which are included in the global data

structure. The memories used in the C++ model are the Picture Buffer (PB), Decoded Picture Buffer (DPB) and the Temporal Picture Buffer (TPB). These memories are accessed using read buffer and write buffer functions, within the load results and store results functions.

The DPB (which stores the reference frame) is accessed by the INTER_P block to perform the inter prediction process, and after that the INTER_P block stores the unfiltered samples corresponding to the current macroblock in the TPB. In order to perform the intra prediction process, the INTRA_P block accesses the TPB to read unfiltered samples corresponding to neighbour macroblocks, and stores the unfiltered samples corresponding to the current macroblock in the TPB. The D_FILTER block reads unfiltered samples stored in the TPB, filters these samples and stores them in the PB. The reason why the TPB memory is included is that the INTRA_P block requires unfiltered neighbour samples and therefore cannot read these samples from the PB, as they are being filtered and stored in this memory by the D_FILTER block. This memory access scheme is shown in Fig. 3.

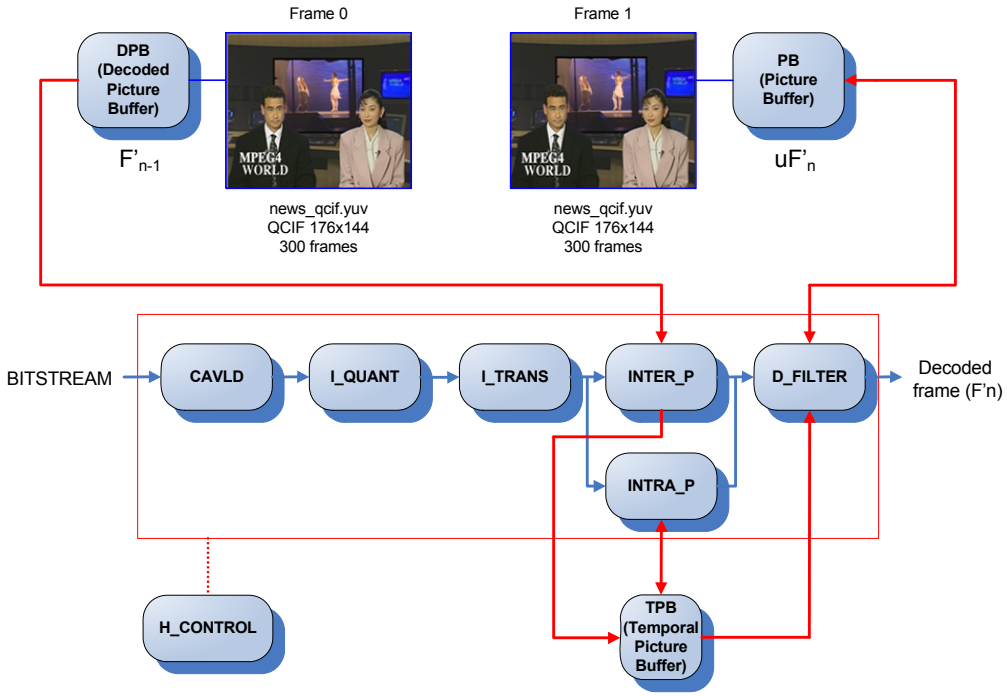

Fig. 3. Memory access scheme used in the C++ model of the H.264/AVC decoder

As an example, the DPB in the figure contains frame 0 of the news test video sequence in QCIF format while the PB contains frame 1 of this same test video sequence.

Each of the functional blocks that comprise the C++ model of the H.264/AVC decoder, and the full decoder, were verified using a series of test cases. The test cases comprise one or more input files, and a reference output file. The video sequences used for the test cases were: deadline, carphone, foreman, mobile and news. The frame format used was QCIF (Quarter Common Intermediate Format) with a resolution of 176 x 144 pixels and colour space 4:2:0. These test cases were run with a QP value of 28, a constant bitrate of 1 KB/s and a constant bitrate of 600 KB/s (constant bitrates mean the QP will be variable). The intra picture period value used was 4. Additionally, in order to test IPCM macroblock compatibility, the following test case was run: news test sequence, 300 frames, 1 slice, QP 0 and period of intra pictures 4. In order to test CIF (Common Intermediate Format) frame format (with a resolution of 352 x 288 pixels) compatibility, the following test case was run: foreman test sequence, 300 frames, 44 slices, QP 28 and period of intra pictures 4.

3. TLM-2.0 MODEL OF THE DECODER

In order to create a TLM-2.0 model of the H.264/AVC decoder from the C++ model described in the previous section, the basic features of the TLM-2.0 standard are used: initiators, targets, transaction objects and sockets[4]. Initiator modules initiate new transactions, while target modules respond to transactions initiated by other modules. A transaction object is a data structure passed between initiators and targets. Sockets are used to pass transactions objects between initiators and targets. Transaction objects are sent through initiator sockets in initiator modules and received through target sockets in target modules. Transaction objects are of type generic payload, which is a general purpose transaction type implemented in the TLM-2.0 standard, and feature a series of attributes. For the purpose of creating the decoder model the following standard attributes were used: command, address, data pointer, data length, streaming width and response status.

The socket interface through which the transaction objects are sent can be blocking or non-blocking. Our model uses the blocking interface, which is associated with the loosely-timed coding style, which focuses on functional execution with minimal or no timing detail. In this case, the model is untimed (as no clock is used) since the goal is the creation of a model whose functionality can be verified with minimal simulation overhead and which serves as the reference model for the subsequent RTL implementation.

Figure 4 shows the TLM-2.0 model of the H.264/AVC decoder. In order to obtain this model from the C++ model, the global data structure and load and store functions are removed. In the model, data is transferred between the decoder functional blocks (CAVLD, I_QUANT, I_TRANS, INTRA_P, INTER_P, D_FILTER and H_CONTROL) using TLM-2.0 socket interfaces. These are point to point communications between independent SystemC modules, as opposed to having a global data structure accessed using load and store functions. The model features, like the C++ model, three sample memories: the PB, DPB and TPB, which were discussed in the previous section and are also accessed using sockets. These memories are arrays declared in independent SystemC modules instead of arrays included in a global data structure, as in the C++ model. Since the address attribute of generic payload only allows the specification of one address value, memory arrays are 1-dimensional in the TLM-2.0 model instead of 2-dimensional as in the C++ model, as this would have required the specification of two address values (height and width).

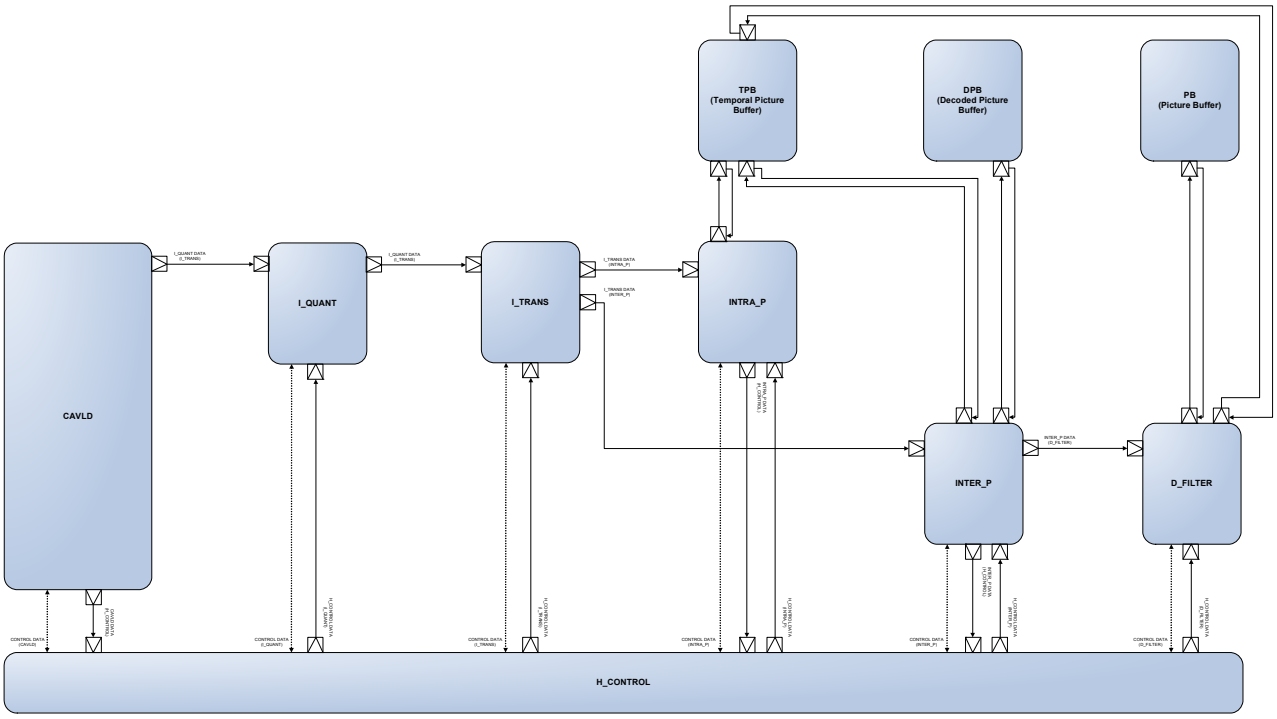

Fig. 4. TLM-2.0 model of H.264/AVC decoder

In the C++ model, given the high level of abstraction, the H_CONTROL block performs control tasks by invoking the load results, main block function and store results functions, as described in section 1. In the TLM-2.0 model, the H_CONTROL block activates each of the functional blocks that comprise the decoder sequentially, using SystemC control signals. The H_CONTROL block asserts a start signal to activate a functional block, and once the functional block has finished processing a macroblock, it asserts a finish signal. Each of the functional blocks that comprise the TLM-2.0 model have an associated start and finish signal, represented by a bidirectional dotted line from the H_CONTROL block to the corresponding functional block in Fig. 4.

The functional blocks that comprise the TLM-2.0 model of the H.264/AVC decoder are independent SystemC modules that feature one thread process and one or more blocking transport methods. As discussed earlier, the blocking interface is used for data transfer, and the TLM-2.0 standard requires that the target module implement one blocking transport method for each target socket. An exception is the CAVLD block that only acts as an initiator, transferring data to the I_QUANT and H_CONTROL blocks, and does not include any blocking transport methods. The memories featured in the model only act as targets and therefore don't include a thread process, only the blocking transport methods.

The blocking transport method is implemented in the target module but invoked by the initiator module in order to transfer the data. For example, the I_TRANS block would implement a blocking transport method to receive data from the I_QUANT block, and this method would be invoked by the I_QUANT block to transfer the data. In order to transfer data between decoder functional blocks, only the write command is used, and this is shown in Fig. 4 as the socket pair does not include the return path. For memory accesses, both the read and write commands are implemented and the return path is shown. The write command and the read command (for memories) are implemented in the blocking transport method in either case.

Upon receiving a transaction object, the blocking transport method checks certain generic payload attributes to determine whether the values are acceptable. As stated earlier, for the purpose of creating the decoder model the following standard attributes are used: command, address, data pointer, data length, streaming width and response status. The address attribute is used by the INTRA_P, INTER_P and D_FILTER blocks for memory access, and it is checked to determine whether it is not greater than the highest address value, taking the size of the memory arrays into account. The streaming width attribute is also used for memory access and allows the transfer of multiple samples located in contiguous memory addresses with only one blocking transport call. However, data transfer between functional blocks does not use this feature and therefore the streaming width attribute is set to match the data length attribute, and the data length attribute itself is set to the size of the transaction object, both for data transfer between functional blocks and for memory access. The blocking transport method checks whether the attributes have acceptable values and therefore the conditions above are met, otherwise an error is generated.

If the generic payload attributes associated to the transaction object are acceptable, then the read or write command is executed in the blocking transport method and the response status attribute is set to indicate the successful completion of the transaction. Once this is done, the blocking transport method notifies the thread process that data has been received using a SystemC event object.

The thread process is implemented as an infinite loop, and within this process the functional block first waits for data from other functional blocks (an exception is the CAVLD block, which only acts as an initiator). For example, the I_TRANS block receives data from the H_CONTROL and I_QUANT blocks, which act as initiators. Therefore, the I_TRANS block features two blocking transport methods that will notify data reception to the thread process using each one event object. Once data has been received, the functional block will wait for the H_CONTROL module to assert the start signal. When the rising edge of the start signal is detected, data received from other functional blocks will be copied to the input data structure of the functional block. The input data structure, as described in the previous section, is a custom data structure whose fields represent the input data of a functional block. In the case of the INTRA_P block, for example, a subset of these fields represent data that comes from I_TRANS block, while the rest represent data that comes from the H_CONTROL block. If the functional block needs to access the sample memories, this step will be done next. The INTRA_P, INTER_P and D_FILTER blocks act as initiators for memory access, copying the required samples to their input structures.

Once the input data structure contains all the data required to process the current macroblock, the functional block's main function will be invoked, and output data corresponding to the current macroblock will be copied to the functional block's output data structure. The input data structure, functional block main function, other functions required by the functional block to process a macroblock, and the output structure are obtained from the C++ model.

The macroblock output data contained in the output data structure will be transferred to other functional blocks and/or written to memory using the corresponding initiator sockets. In the case of the INTRA_P block, for example, a subset of the fields of the output data structure contain the macroblock 4x4 intra modes that will be transferred to the H_CONTROL block using a transaction object through the corresponding socket interface. The remaining fields of the output data structure contain the unfiltered samples of the current macroblock, which will be written to the TPB memory using the corresponding socket interface.

After data has been transferred to other functional blocks and/or written to memory, the functional block will assert a finish signal, notifying the H_CONTROL block that the current macroblock has been processed and that data transfer is complete.

4. ESL REFINEMENT METHODOLOGY

As discussed in the previous section, the functional blocks that comprise the TLM-2.0 model of the H.264/AVC decoder are SystemC modules with socket interfaces. These socket interfaces are used for data transfer between functional blocks. Within the SystemC modules, the original C++ model code is used to perform the block's function.

In order to create a synthesizable SystemC model of the decoder, a refinement methodology that involves a modeling phase, high level synthesis, optimization and logic synthesis is used. This methodology is applied to each of the functional blocks, which were refined, implemented and verified independently. The following subsections will discuss each of these steps.

4.1 Modeling

In the modeling phase, the goal is the creation of a synthesizable SystemC model of a functional block and a corresponding testbench, in order to verify the functional block independently. The tool used in this phase is Microsoft Visual studio 2005 with version 2.2 of the OSCI SystemC library[5]. A series of steps are performed in order to obtain initial high level synthesis results, and these steps are discussed in the following paragraphs.

The first step that is performed is the addition of input ports for the clock and reset signals. In order to implement the reset signal, a function of the high level synthesis tool, Agility Compiler version 1.2, is used. This function allows the definition of a global asynchronous reset for the whole design, and it is invoked in the constructor of the top level module of a functional block. For simulation with Microsoft Visual Studio, this reset signal is not used. In the functional block modules, internal and output signals are initialized outside the thread process loop, while variables are initialized in the module constructor.

After adding the clock signal, the thread processes within the modules are made sensitive to the rising edge of this clock signal.

The next step of the modeling phase is the replacement of the TLM-2.0 socket interface with SystemC signal channels. An interface with a request, validate and data signal is used. In this interface, the module that transfers data is a source module, and the module that receives data is a sink module. The sink module asserts the request signal, and waits until the source module is ready to transfer the data. When the source module is ready, it asserts the validate signal, meaning that the data is valid. When the sink module detects that the validate signal has been asserted, it de-asserts the request signal and reads the data signal. Likewise, when the source module detects that the request signal has been de-asserted, it de-asserts the validate signal. The validate signal can remain asserted for more than one clock cycle if multiple data words need to be transferred. In this case the source module won't de-assert the validate signal when the request signal is de-asserted, only when the transfer of all the data words is complete.

Each TLM-2.0 socket interface of the functional block is replaced with this request, validate and data signal interface. If the module acts as a source module (it has an output socket) the request signal will be an input signal, while the validate and data signals will be output signals. If the module acts as a sink module (it has an input socket) the request signal will be an output signal, while the validate and data signal will be input signals. For example, the I_QUANT block acts as a sink to receive the quantized coefficients and other data required to process a macroblock from the CAVLD block, and acts as a source to transfer dequantized coefficients to the I_TRANS block.

When a functional block acts as a sink to receive data, the data word or words received will be stored in an input memory. Each input interface has an associated memory, as data reception happens in parallel using point to point interfaces, and the memories used only have one write port. For example, the I_TRANS block receives one data word from the H_CONTROL block, which is stored in an input memory, and 384 dequantized coefficients (256 for luminance samples + 128 for chrominance samples) from the I_QUANT block, which are stored in another input memory. Likewise, when a functional block acts as a source to send data, data words are read from output memories and sent through the corresponding interfaces.

The memories used in a functional block are synchronous and feature one read and one write port. Each memory used in the design is an independent SystemC module instantiated in the top module of the functional block. For simulation, the memories are implemented as arrays of the appropriate depth and width. In the high level synthesis stage, additional code is added to the memory modules, so that Agility Compiler will implement these memories using FPGA memory resources.

The memory architecture used in the decoder allows one sample to be read from memory or written to memory in each clock cycle. This means that the input and output interfaces will receive or transfer one sample per cycle, and that the core module will read from or write one sample to memory per clock cycle for processing. This also applies to the quantized coefficients that the CAVLD block transfers to the I_QUANT block, to the dequantized coefficients that the I_QUANT block transfers to the I_TRANS block, and to the residual elements that the I_TRANS block transfers to the INTRA_P and INTER_P blocks. This memory access scheme is used to simplify the decoder architecture and provides sufficient bandwidth to decode QCIF frames at a frame rate of 30 frames per second, as will be seen in the results section.

It was decided at this stage that the synthesizable SystemC model of a functional block would be divided into at least two modules, featuring an architecture comprising an interface module and a core module. The interface module manages the input and output interface or interfaces, as described above, and the core module performs the actual macroblock processing. The interface and core modules communicate using memories: the core module reads the input data stored by the interface module in memory and stores output data in memory, which will be read by the interface module. This partition is done to simplify the following optimization phase and improve both high level synthesis times and results, and presents a logical separation of the functional block's interface and functionality. The core module can be divided into additional sub-modules to further improve synthesis results, as will be discussed later. The interface module also performs control tasks, as it activates the core module using a start signal and is notified by the core module once the macroblock has been processed using a finish signal.

Once activated by the interface module, the core module reads input data from the input memory or memories, and stores it in the input data structure of the functional block. The original C++ code included in the SystemC module is invoked to process the macroblock, and output data is stored in the output data structure. This data is written to the output memory or memories, so that the interface module can transfer this data through the output interface or interfaces.

The input and output data structures are custom data structures used in the C++ model in which variables and arrays are of type integer, as described in section 2. Although these structures are synthesizable, they are removed in order to reduce the area requirements of a functional block.

At this stage, the functional block is verified using the same testbench used to verify the C++ model. The testbench is modified to use the same request, data and validate signal interface used by the functional block. Verification using this testbench is done every time changes are made in the modeling phase.

The next step performed in order to prepare the SystemC module of the functional block for the initial high level synthesis, is the addition of timing information to the core module. Although synthesis would be possible without doing this, the entire core module is implemented as a combinational circuit in such a case and area requirements exceed FPGA resources, even for the less complex functional blocks such as I_QUANT and I_TRANS.

Adding timing information involves adding SystemC wait() statements to the code. The interface module that replaces the TLM-2.0 socket interfaces is already cycle-accurate as it includes the necessary wait() statements so that the input and output interfaces can work as described earlier.

Several factors must be taken into account when adding wait() statements to the code. Every wait() statement added to the code means that the functional block will require an additional clock cycle to process a macroblock, as the processes are sensitive to the rising edge of the clock and a wait() statement means the processes will wait for the next clock cycle

and therefore data will be registered. Therefore, the number of cycles required to process a macroblock can be controlled directly, with the addition or removal of wait() statements. Also, since the operation frequency is determined by the critical path of the design, code lines with many arithmetic or logic operations done serially will be split by adding wait() statements and creating intermediate registers, so that the number of arithmetic and logic operations done before a wait() is limited to 3 for the initial high level synthesis.

After the addition of wait() statements, the core module code is simplified by turning multidimensional arrays into single-dimensional arrays. This is done because operating with arrays of more than one dimension requires nested for loops to generate indices for each dimension, and more registers for the index values, complicating the hardware required and leading to worse synthesis results. Additionally, it is of particular interest to convert multidimensional arrays that hold constant values to single-dimensional arrays as this would allow the specification of these arrays as ROM memories in the high level synthesis stage, to reduce area requirements.

The next step is the replacement of the integer data type, used in the C++ model, with the SystemC sc_uint and sc_int data types, which are found to yield better high level synthesis results, as they allow the specification of the exact register widths in bits. Other changes made to the code are related to the replacement of the input and output structures of each functional block with input and output memories, as described earlier. Sections of code that read from or write to these structures will instead access the corresponding memory using functions that have been created for this purpose. The number of variables and arrays used for temporal results is reduced whenever possible, writing to memories directly in order to reduce area requirements.

4.2 High level synthesis and optimization

Once the functional block code has been modified in the modeling phase, Agility Compiler is run next in order to perform high level synthesis. The tool is configured to use EDIF as the output format with default optimizations enabled, and to use the target FPGA, a Xilinx Virtex 4 (part number xc4vfx60ff1152-11). Once high level synthesis is performed, Agility Compiler generates an area and delay estimation summary. This report provides a breakdown of FPGA resources used for each file that comprises a functional block, and the longest combinational paths in the design. We use the number of LUTs required to estimate the area requirements for the functional block, and the maximum logic and routing delay from flip flop to flip flop (critical path) to estimate the frequency of operation.

The area requirement is compared to a reference value that depends on block complexity. Less complex functional blocks such as the I_TRANS and I_QUANT blocks are allowed a maximum area requirement of 5000 LUTs, while the INTER_P, D_FILTER and CAVLD blocks can have an area requirement of no more than 15000 LUTs. Other blocks must have an area requirement below 10000 LUTs. The initial critical path reference value is 10 ns, but was allowed to reach 15 ns, seeing that the logic synthesis tool is able to optimize the critical path to reach the target frequency of 100 MHz, for values up to 15 ns.

If either the area requirement or critical path are above the reference values for the functional block, an iterative process involving the optimization of the SystemC code in the editor and the execution of Agility Compiler to obtain area and delay results is performed, in order to bring the values obtained in the report as close as possible to the reference values. In this optimization phase, the goal is not only to adjust the frequency but the number of cycles as well, so that the performance goal of achieving over 30 frames per second for QCIF video sequences is met, while at the same time satisfying area requirements.

In order to reduce area requirements in the optimization stage, additional changes were made to the code of each functional block to reduce fanout problems. In the original C++ code, shared variables and arrays are read in many different branches, meaning that the resultant hardware registers will have high fanout. This applies particularly to the more complex blocks such as INTRA_P, INTER_P and D_FILTER, which process the luminance and chrominance components of the macroblock serially reutilizing the same variables in many cases. This leads to an increase in area requirements and makes it harder to optimize the critical path. To alleviate these problems, the core module of the INTRA_P, INTER_P and D_FILTER blocks is divided into a luminance core and a chrominance core, which process the macroblock components in parallel. This improves synthesis results and times as it is easier for Agility Compiler to optimize smaller modules. For example, the core module of the INTRA_P block has an area requirement of 20544 LUTs, and a critical path value of 13,88 ns. After dividing the module, the resultant luminance module has an area requirement of 12738 LUTs and a critical path value of 12,92 ns, while the chrominance module has an area requirement of 3595 LUTs and a critical path of 13,78 ns. In this case the improvement in the area requirement is over 4000 LUTs, and it can be seen that the critical path is in the chrominance module and that the luminance module requires over three

times as many FPGA LUTs as the chrominance module. The high area requirement of the luminance module is again due to fanout issues. The arrays used to store the macroblock neighbour samples for 16x16 intra prediction are read from many different branches in the code, leading to high fanout and high area requirements as a result. The solution in this case is the removal of these arrays, reading samples directly from memory when required, and relying on one or two registers for operations. The functions that had these arrays as parameters will have one or more memory base addresses as parameters, from which to read the required samples. After performing these changes to the luminance module code, the area requirement is reduced by more than 10000 LUTs.

Once area has been optimized to bring it as close as possible to the reference value, the critical path is optimized next. To optimize the critical path, the Agility Compiler report is used, as it provides a list of the code lines in the longest combinational path, the FPGA resources in the path and the delays between them. It must be taken into account that optimizing one value might lead to the worsening of another, as reducing the area requirement and obtaining a high operation frequency are frequently mutually exclusive goals. For example, as stated earlier, the luminance module of the INTRA_P block was optimized to reduce the area requirement by 10000 LUTs, however, the changes made to the code also lead to an increase in the critical path to 16 ns. In order to reduce this critical path, the Agility Compiler report is studied to determine which code lines contribute to the critical path. The lines involved feature a calculation involving three consecutive arithmetic operations which are done combinationally. The combinational path is broken by registering intermediate results, as shown below.

Table 1. Critical path optimization example.

```
// Original code
ih_acu = (3 * (sample_add_0 - sample_add_1)) + ih_acu;
wait();

// Modified code
partial_result_sub = sample_add_0 - sample_add_1;
wait();
partial_result_mult = partial_result_sub * 3;
wait();
ih_acu = partial_result_mult + ih_acu;
wait();
```

After optimizing both area and critical path, the number of cycles required to process a macroblock is optimized next. In order to determine the total number of cycles that a functional block requires to process a macroblock, code is added to the functional block to subtract the start and finish SystemC simulation times and divide by the clock period, obtaining the number of cycles. Reducing the number of cycles involves removing wait() statements that are not in the critical path. Wait() statements that represent an unnecessary wait for the next clock cycle, can be removed. In these cases, Agility Compiler has inferred the hardware elements to perform the associated operations in parallel, but the hardware elements are used serially due to the addition of wait() statements.

4.3 Logic synthesis and implementation

Once the functional blocks have been optimized and area and delay estimation values are satisfactory, Agility Compiler is run again, this time using VHDL as the output format with default optimizations enabled. An additional option specified is flatten hierarchy, so that only one VHDL file is generated for the functional block, instead of one for each of the modules that comprise the functional block.

The VHDL code obtained is used to perform logic synthesis with Synplify Pro version 9.6.1. This tool is configured to use the target FPGA. For synthesis, the resource sharing option is disabled, retiming is enabled and no timing constraints are set, so that the tool will attempt to obtain the highest frequency possible. These settings were found to give better results for the project discussed in this article.

After obtaining logic synthesis results, the EDIF output file generated by Synplify Pro is used to obtain final area and delay results using Xilinx ISE version 10.1. The tool is configured to use the target FPGA and other options are set to default values.

5. RESULTS

Table 2 shows results for each functional block that comprises the H.264/AVC decoder, and the full decoder. The table shows synthesis results (area, critical path and frequency values) obtained with Agility Compiler and Synplify Pro.

Table 2. Area, critical path and frequency results obtained with Agility Compiler and Synplify Pro

	Area (LUTs)		Critical path (ns)		Frequency (MHz)	
	Agility Compiler	Synplify Pro	Agility Compiler	Synplify Pro	Agility Compiler	Synplify Pro
CAVLD	8671	7352	13,59	5,73	73,58	174,40
I_QUANT	3799	2753	13,73	5,73	72,83	174,40
I_TRANS	2884	2819	10,65	5,73	93,90	174,40
INTRA_P	7059	6185	13,69	5,73	73,05	174,40
INTER_P	14831	12240	14,54	8,22	68,78	121,70
D_FILTER	9826	6978	13,03	5,73	76,75	174,40
H_CONTROL	7521	6088	13,75	5,73	72,73	174,40
DECODER	57822	46871	15,56	8,44	64,27	118,48

It can be seen that Synplify Pro is able to optimize the area requirement obtained with Agility Compiler significantly in some cases. The area requirement obtained for the D_FILTER and INTER_P blocks in Agility Compiler is reduced by more than 2500 LUTs in Synplify Pro. Critical path values obtained with Agility Compiler are likewise improved significantly. Synplify Pro is able to obtain a maximum frequency of 174,40 MHz for all functional blocks except the INTER_P block, which achieves a critical path value of 8,22 and a corresponding frequency of 121,70 MHz. This is due to the complexity of the ¼ pixel interpolation algorithm required by the H.264/AVC standard, which must be performed in the inter predictor functional block. The full decoder area requirement obtained in Synplify Pro is below the total number of LUTs available in the Virtex-4 FPGA used. The maximum operating frequency for the full decoder, 118,48 MHz, is below the INTER_P block frequency, as expected.

Table 3 shows area, critical path and frequency values obtained with Xilinx ISE, for each functional block and the full decoder, after performing place and route. Xilinx ISE is able to improve the Synplify Pro area requirement modestly, and the critical path value is improved for the I_TRANS block, resulting in a maximum operating frequency of 197,04. This might be due to the simplicity of the inverse transform functional block and the fact that ISE is a tool by Xilinx and can target the Virtex-4 FPGA hardware better. The maximum operating frequency for the full decoder, 118,48 MHz, is above the required frequency value of 100 MHz.

For this operating frequency, the decoder can process QCIF images with a frame rate above 30 frames per second. In order to determine this, the maximum number of cycles required to process a macroblock is obtained using the test cases discussed in section 1, for each functional block and for the full decoder. The CAVLD, I_QUANT, I_TRANS and INTRA_P blocks require no more than 5000 cycles to process a macroblock in the worst case. The D_FILTER block takes at most 15000 cycles to process a macroblock, while the maximum number of cycles that the INTER_P block requires to process a macroblock is below 25000 cycles. The H_CONTROL block and full decoder require no more than 30000 cycles to process a macroblock in the worst case. Assuming a worst-case situation in which all macroblocks are processed in 30000 cycles, the decoder would still yield a frame rate above 33 frames per second for an operating frequency of 100 MHz.

Table 3. Area, critical path and frequency results obtained with Xilinx ISE

	Area (LUTs)	Critical path (ns)	Frequency (MHz)
CAVLD	8521	7,31	136,80
I_QUANT	2716	6,74	148,46
I_TRANS	2804	5,08	197,04
INTRA_P	6102	6,44	155,21
INTER_P	12061	8,81	113,47
D_FILTER	6881	7,70	129,82
H_CONTROL	7428	6,87	145,56
DECODER	46513	9,13	109,53

6. CONCLUSIONS

In this paper, an ESL design methodology was established and was put to use in the creation of a synthesizable SystemC model of a H.264/AVC decoder. This decoder adheres to the baseline profile of the H.264/AVC standard and is aimed at low-end mobile applications and handheld devices. Results obtained in the application of the methodological steps show that the SystemC model of the decoder meets expected area and frequency requirements. For an operating frequency of 100 MHz, frame rates above 30 would be achievable for a QCIF frame size.

The synthesizable SystemC model of the decoder will be used for future research and development, being of particular interest the deployment on a prototyping platform, in order to validate the results obtained with the software tools employed in the design methodology.

ACKNOWLEDGEMENTS

The work described in this paper is in part supported by the ARTEMI+ project TEC2006-13599-C02-02/MIC (Architectures for portable multi-standard multi-network multimedia terminals) from the Spanish Ministry for Education and Science, National Plan for Research.

REFERENCES

[1] Coussy, P., Morawiec, A., [High-Level Synthesis], Springer Science + Business media B.V., 1-12 (2008).
[2] JM: H.264/AVC reference decoder version 12.4, http://iphome.hhi.de/suehring/tml/
[3] Richardson, I., [H.264 and MPEG-4 Video Compression], John Wiley & Sons Inc., 160-162 (2003).
[4] Getting Started with TLM-2.0, http://www.doulos.com/knowhow/systemc/tlm2/
[5] OSCI standards – Open SystemC Initiative, http://www.systemc.org/downloads/standards/