ESKİŞEHİR TECHNICAL UNIVERSITY JOURNAL OF SCIENCE AND TECHNOLOGY
A- APPLIED SCIENCES AND ENGINEERING

2018, 19(3), pp. 685 - 697, DOI: 10.18038/aubtda.422250

A LOW AREA FULLY PIPELINED IMPLEMENTATION OF JPEG ON FPGA

Atakan DOĞAN [1, *], İsmail SAN [1]

[1] Department of Electrical and Electronics Engineering, Faculty of Engineering, Eskişehir Technical University, Eskişehir, Turkey

ABSTRACT

This paper presents a low-area and high-throughput design and implementation of JPEG encoder on FPGA. The design consists of three main components: (1) 2-D DCT module, employing the row-column decomposition technique, (2) Quantization in zigzag ordering, utilizing look-up tables, and (3) Entropy coder, transforming the quantized DCT coefficients into JPEG words. All components are fully pipelined and optimized for FPGA resource utilization. The proposed implementation of JPEG encoder is able to encode 143 and 71 SDTV frames per second with 720x480 gray scale and color pixels per frame, respectively, on Xilinx Spartan 6 FPGA. Moreover, the proposed architecture is capable of encoding at least 53 and 26 HD Ready TV frames per second with 1280x720 gray scale and color pixels per frame, respectively, on this FPGA chip. Thus, the proposed JPEG encoder architecture is well-suited to various image and video compressing applications where performance and area are significantly important.

Keywords: JPEG compression, Discrete cosine transform, Logic design, FPGA

1. INTRODUCTION

The Joint Photographic Experts Group introduced the JPEG compression standard for still photographic images [1], which has become the widely used lossy compression standard ever since. JPEG is heavily used in high-resolution image transmission applications including digital cameras, image scanners and so on. Such applications require both high-speed and low cost implementations of image encoding and decoding. In order to meet several needs of various applications, the JPEG standard specifies two classes of encoding and decoding processes: Discrete Cosine Transform (DCT) based processes for *lossy* compression and predictor based processes for *lossless* compression, each with a few modes of operation [1, 2]. Among the different operation modes of the DCT based lossy compression, the baseline mode is widely implemented in software and hardware for JPEG compression [3-10]. Consequently, the JPEG IP core architecture proposed in this study is based on the baseline mode.

In the literature, several studies are devoted to FPGA [3-7] and ASIC [8-10] implementations of the baseline JPEG compression. The common approach of these studies is to split the baseline JPEG compression process into four different modules handling 2-D DCT operation, quantization step, zig-zag ordering step, and entropy coding operations separately. Among these studies, only [4, 9, 10] provide detailed hardware design of each of these modules, while the others describe how each of them operates in general without elaborating on the hardware design. As a result, the hardware design of JPEG IP core proposed in this study is compared against [4, 9, 10]: (i) The proposed design implements 2-D DCT based on single 1-D DCT hardware, while they use two 1-D DCT modules. (ii) The proposed design performs the quantization in zig-zag ordering in a single module, whereas they do the quantization and zig-zag ordering in different modules. (iii) All designs are fully pipelined to obtain the highest throughput possible.

This study proposes a novel IP core for high throughput JPEG compression on low cost FPGAs. In order to achieve the best throughput possible, each module is designed to be fully pipelined and optimized for the

*Corresponding Author: atdogan@anadolu.edu.tr
Received: 09.05.2018 Accepted 09.07.2018

FPGA resource utilization, while meeting the requirements of the JPEG standard [1, 2]. All modules were captured in Verilog HDL. The proposed design is synthesized, simulated and verified with Xilinx ISE 14.7 tool. According to the results from ISE 14.7 tool, the JPEG IP core proposed is capable of compressing more than 143 SDTV frames per second with 720×480 gray scale pixels and 53 HD Ready TV frames per second with 1280×720 gray scale pixels per frame on a low cost Xilinx Spartan 6 FPGA chip.

The rest of the article is organized as follows: the design philosophy and hardware implementation details related to the proposed JPEG IP core architecture are presented in Section 2. The implementation results of the proposed JPEG core on different Xilinx FPGAs are given in Section 3. Furthermore, comparisons between the proposed design and other JPEG cores from the open literature are given in Section 3. Finally, the article is concluded in Section 4.

2. JPEG IP CORE ARCHITECTURE

Based on [1, 2], the proposed JPEG IP core is composed of three main components, which are 2-D DCT, quantization in zig-zag order, and entropy coder. FIFO based input and output interfaces provide low-complexity flow control between the modules, which provides write and read semantics similar to a FIFO buffer interface, and they are explained as follows:

- The input interface: A new data is received by the module on its *writeData* bus in the next rising edge of clock when *writeEn* signal is asserted and *full* signal is de-asserted during the current clock cycle. When *full* signal is asserted by the module, it cannot accept a new data word in the current clock cycle. The bit length and the direction of the input interface signals are given below.
  - *writeData* bus, input, 8-, 12-, or 96-bit
  - *writeEn* signal, input, 1-bit
  - *full* signal, output, 1-bit
- The output interface: A new data is ready on its *readData* bus in the next rising edge of clock when *readEn* signal is asserted and *empty* signals is de-asserted during the current clock cycle. When *empty* signal is not asserted by the module, there is always an available valid data word on its *readData* bus. Once the module cannot produce a new data word in the current clock cycle, the empty signal is asserted. The bit length and the direction of the output interface signals are given below.
  - *readData*, output, 12-, or 96-bit
  - *readEn*, input, 1-bit
  - *empty*, output, 1-bit

Following sections provide necessary details for each component in the proposed hardware design.

2.1. 2-D DCT Architecture

In this study, a low area implementation of the 2-D DCT is designed and implemented on FPGA. Its architecture is adopted and modified from a study proposed by [11] for ASIC implementation. In selecting the architecture proposed by [11] to be implemented on FPGA, there are three important reasons: (1) it is based on the row-column decomposition technique. Hence, a low area utilization is achieved since only single 1-D DCT component is used in a time-shared fashion, (2) a shift-register based transpose buffer is implemented in order to improve the utilization of Block RAM resources, and (3) simpler finite state machines are responsible to control the datapath since the control logic is distributed among the components.

Figure 1 (reprinted from [12]) illustrates the overview of the proposed 2-D DCT design for FPGA implementation. The main modules including ping and pong buffers, 1-D DCT component, transpose and output buffer are depicted in Figure 1. The main differences of 2-D DCT architectures between [11] and this study include a different operation of pong buffer, pipeline register inclusion and the logic for rounding method in 1-D DCT operation, output buffer, and a special pipeline that can be stoppable. In the following sections, each of five components are separately explained by specifying what each module is responsible to compute in 2-D DCT.

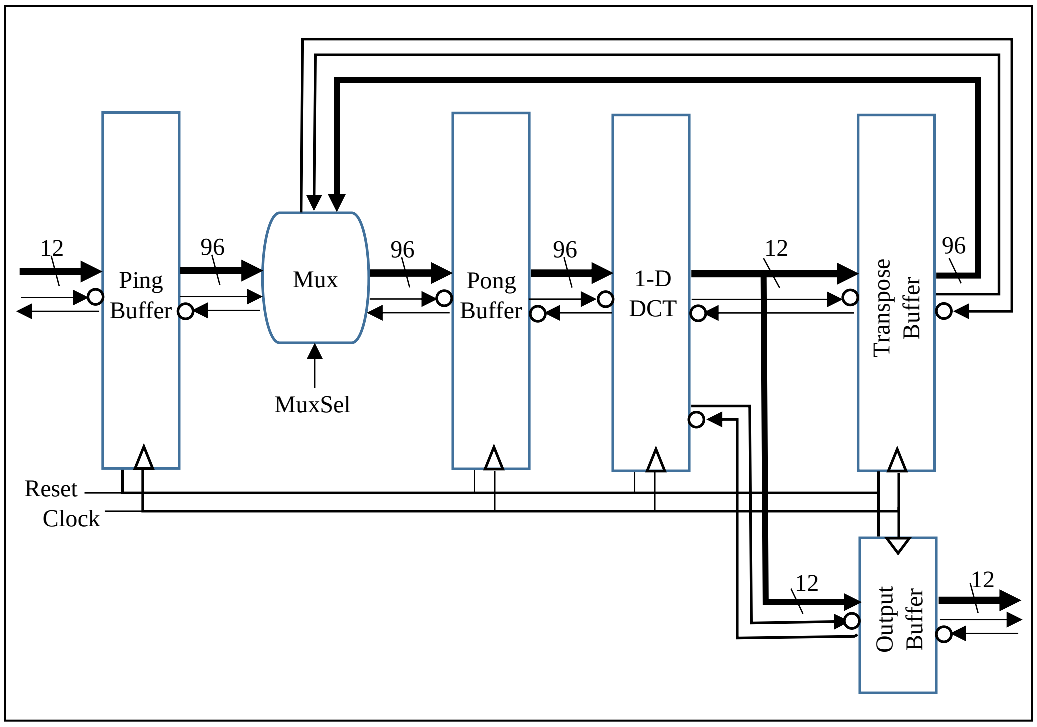

Figure 1. Harwdare architecture of 2-D DCT operation (reprinted from [12])

2.1.1. Ping-Pong buffers

Ping-pong buffering method contains two separate buffers: ping and pong buffers. While ping buffer is being loaded with input data, 1-D DCT component performs 1-D DCT operation using data stored in pong buffer.

The ping buffer is simply a 96-bit shift-register which is controlled by a finite state machine (FSM) with two states {*empty*, *full*}:

- *empty* state (serial-in): when *writeEn* signal is asserted, a new 12-bit word data is shifted into the ping buffer. Once the eighth 12-bit word is inserted into the ping buffer, finite state machine changes its state to the *full* state. Therefore, ping buffer takes at least eight clock cycles to become full. In this state, *empty* signal is asserted.
- *full* state (parallel-out): when *readEn* signal is asserted, 96-bit current state of ping buffer is read in single clock cycle and finite state machine changes its state to *empty* state. In *full* state, *full* signal is asserted.

According to the proposed ping buffer operation, loading all elements of 8x8 matrix into the ping buffer takes 64 cycles and transferring the rows of 8x8 matrix to pong buffer requires 8 cycles. Thus, totally it requires 64+8=72 clock cycles to be shifted in and out.

Pong buffer is simply a 96-bit register which is controlled by a finite state machine with four states {*oned_dct_empty*, *oned_dct_full*, *twod_dct_empty*, *twod_dct_full*}:

- *oned_dct_empty* state (parallel-in operation): If *writeEn* signal is asserted, a new row of eight pixels (96-bit word) is loaded into the pong buffer and finite state machine changes its state to *oned_dct_full* state. *MuxSel* signal is not asserted so as to load from the ping buffer. In this state, *empty* signal is asserted.
- *oned_dct_full* state (coefficient computation): in every clock cycle, a new 1-D DCT coefficient is computed. The machine stays in this state for eight cycles since there are eight pixels per row. During the computation of the last coefficient for a row, it goes to either *oned_dct_empty* state when this is not the last row, or *twod_dct_empty* state, otherwise. In this state, *full* signal is asserted.
- *twod_dct_empty* state (parallel-in operation): This state is same as *oned_dct_empty* state except that *MuxSel* signal is now asserted to receive a column of eight 1-D DCT coefficients from the transpose buffer.
- *twod_dct_full* state (coefficient computation): it is very similar *oned_dct_full* state except that it changes its state to either *oned_dct_empty* state when the last column is being processed, or *twod_dct_empty* state in the eighth clock cycle.

According to the proposed pong buffer operation, 64+8=72 clock cycles are needed to perform the computation of either 1-D or 2-D DCT coefficients. Therefore, a total of 144 clock cycles are required by pong buffer to complete the processing of 8x8 matrix of pixels. In a fully-pipelined operation, the ping buffer latency will be completely overlapped with the pong buffer latency, so they will together introduce a latency of 144 clock cycles.

2.1.2. 1-D DCT

Eight-point 1-D DCT operation is computed by using the row-column decomposition technique [13] and it is given as follows:

$$\begin{aligned}
z_0 &= d(x_0+x_7)+d(x_1+x_6)+d(x_2+x_5)+d(x_3+x_4)\\
z_2 &= b(x_0+x_7)+f(x_1+x_6)-f(x_2+x_5)-b(x_3+x_4)\\
z_4 &= d(x_0+x_7)-d(x_1+x_6)-d(x_2+x_5)+d(x_3+x_4)\\
z_6 &= f(x_0+x_7)-b(x_1+x_6)+b(x_2+x_5)-f(x_3+x_4)\\
z_1 &= a(x_0-x_7)+c(x_1-x_6)+e(x_2-x_5)+g(x_3-x_4)\\
z_3 &= c(x_0-x_7)-g(x_1-x_6)-a(x_2-x_5)-e(x_3-x_4)\\
z_5 &= e(x_0-x_7)-a(x_1-x_6)+g(x_2-x_5)+c(x_3-x_4)\\
z_7 &= g(x_0-x_7)-e(x_1-x_6)+c(x_2-x_5)-a(x_3-x_4)
\end{aligned}$$

where z_i denotes the transformed coefficient, x_i denotes the pixel data, $a=C_1$, $b=C_2$, $c=C_3$, $d=C_4$, $e=C_5$, $f=C_6$, $g=C_7$, $C_i=0{,}5\cos(k\pi/16)$, i=0,1,..7, and k=1,2,..7. Furthermore, let $X_0 = x_0+x_7$, $X_2 = x_1+x_6$, $X_4 = x_2+x_5$, $X_6 = x_3+x_4$, $X_1 = x_0-x_7$, $X_3 = x_1-x_6$, $X_5 = x_2-x_5$, and $X_7 = x_3-x_4$.

Figure 2 shows the proposed design for 1-D DCT operation which calculates the coefficients by means of these equations. The topmost Add/Sub component illustrated in Figure 2 computes either $X_0=(x_0+x_7)$ when *OddSel* is equal to 0 or $X_1=(x_0-x_7)$ when *OddSel* is equal to 1, and so on. Therefore, *OddSel* signal is asserted only when a coefficient with odd-index $\{z_1, z_3, z_5, z_7\}$ is computed, and Add/Sub components compute either $\{X_0, X_2, X_4, X_6\}$ or $\{X_1, X_3, X_5, X_7\}$ values in parallel.

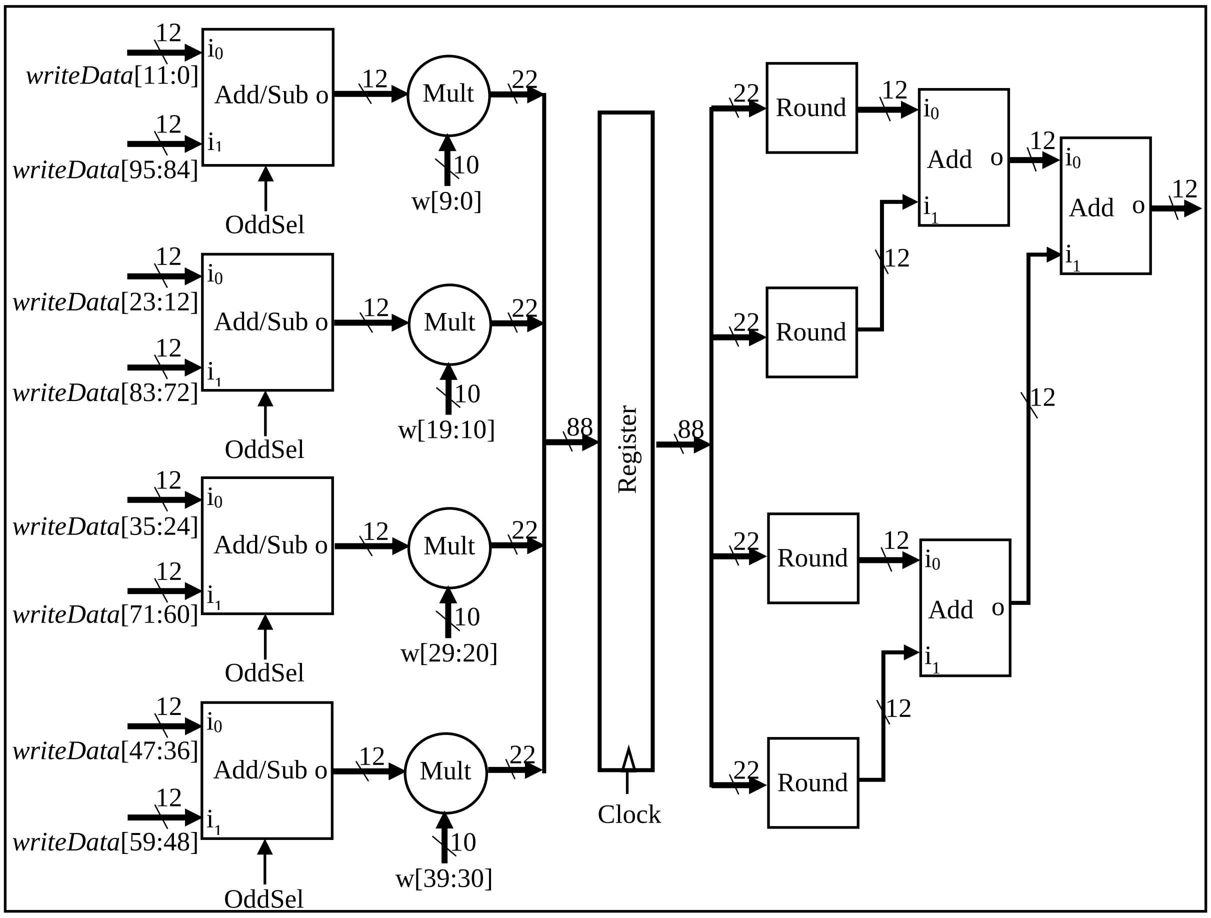

Figure 2. 1-D DCT architecture

The results of parallel Add/Sub components are fed into the multipliers. Four integer multiplications are performed in parallel for every coefficient. In Figure 2, *w*[39:0] represents the set of weights used during the multiplications. For instance, $z_2 = b^* \times X_0 + f^* \times X_2 - f^* \times X_4 - b^* \times X_6$ and $w[39:0] = \{b^*, f^*, -f^*, -b^*\}$, where * symbol is used to denote 10-bit 2's complement representations. There is single 8×40-bit look-up table to store the weight values and it is not explicitly shown in Figure 2. The address of this look-up table is 3-bit index value of the coefficient being computed. The table stores 40-bit rows for each address and each row stores four different weights (i.e., $\{b^*, f^*, -f^*, -b^*\}$ for z_2) in 2's complement format per coefficient.

After performing the multiplications, there is an 88-bit register which is controlled by a FSM with two states {*empty*, *full*} defined as follows:

- *empty* state: when *writeEn* signal is asserted, a new 88-bit word is received into the register and FSM makes a transition to the *full* state. In *empty* state, *empty_transpose* (*empty* signal for transpose buffer) or *empty_outbuff* (*empty* signal for output buffer) signal is asserted if 1-D DCT or 2-D DCT coefficients are being computed, respectively.
- *full* state: based on the results of four multiplications loaded in this register, a new 1-D DCT or 2-D DCT coefficient will be computed by means of the following rounding and adder tree circuits.

During the computation of 1-D DCT coefficients, as long as *writeEn* is asserted, a new 88-bit word will be stored into the register in every clock cycle and the machine stays in *full* state; otherwise, it goes to *empty* state. While computing 2-D DCT coefficients, on the other hand, a new 88-bit word will be stored into the register if both *writeEn* and *readEn_outbuff* are asserted so as to guarantee that the new 2-D coefficient is accepted by the output buffer. In this state, *full* signal is not asserted only if the output buffer becomes full during the computation of 2-D DC coefficients.

22-bit result $\{r_{21}, r_{20}, \ldots, r_0\}$ for each signed multiplication is rounded to 12-bit 2's complement value by a combinational logic circuit depending on the sign of the result and it is defined as follows:

- When the result is *positive*: There are three cases:
  - If $\{r_{21}, r_{20}, \ldots, r_{10}\}$ is the maximum 22-bit positive number, the rounded result is equal to $\{r_{21}, r_{20}, \ldots, r_{10}\}$.
  - If $\{r_{21}, r_{20}, \ldots, r_{10}\}$ is not the maximum 22-bit positive number and r_9 is equal to 0, the rounded result is equal to $\{r_{21}, r_{20}, \ldots, r_{10}\}$.
  - If $\{r_{21}, r_{20}, \ldots, r_{10}\}$ is not the maximum 22-bit positive number and r_9 is equal to 1, the rounded result is equal to $\{r_{21}, r_{20}, \ldots, r_{10}\} + 1$.
- When the result is *negative:* If r_9 is equal to 0, it is equal to $\{r_{21}, r_{20}, \ldots, r_{10}\}$; otherwise, $\{r_{21}, r_{20}, \ldots, r_{10}\} + 1$.

After rounding operation, 12-bit adder tree with four-input is used to compute either 1-D or 2-D coefficient values in 2's complement format.

2.1.3. Transpose and output

Transpose buffer is simply a shift register with 63×12=756-bit length and it is adopted from [11]. There are two scenarios to consider for this module:

- *serial-in*: If *writeEn* signal is asserted during 1-D DCT computation, a new 12-bit coefficient is serially shifted in this buffer.
- *parallel-out:* Consider the *shift_register*=$\{reg_{62}, reg_{62}, \ldots, reg_0\}$ that is composed of 63 12-bit registers. When the shift register becomes full, a set of eight registers *column*=$\{reg_{56}, reg_{48}, reg_{40}, reg_{32}, reg_{24}, reg_{16}, reg_8, reg_0\}$ store the first column of 1-D DCT coefficients. In the next clock cycle, the 64th 1-D DCT coefficient is shifted in while the first column is received into pong buffer. After the right-shift, *column* will store the second column of 1-D DCT coefficients.

The output buffer stores the coefficients of 2-D DCT operation and isolate the 2-D DCT hardware in Figure 1 from the following quantization component. Output buffer consists of two registers, namely reg_0 and reg_1, and they are controlled by a three-state {*empty*, *almost-full*, *full*} finite state machine whose states are defined as follows:

- *empty* state: it means that both registers are empty. When *writeEn* signal is asserted, a new 12-bit word is loaded into reg_0 and finite state machine changes its state to *almost-full* state. In this state, *empty* signal is asserted.
- *almost-full* state*:* when both *writeEn* and *readEn* signals are asserted or deasserted, FSM stays in here. Moreover, when they are asserted, a new word is received into reg_0 register. If *writeEn* signal is asserted, but *readEn* signal is de-asserted, a new word is received into reg_1 register and it goes to *full* state. If *writeEn* signal is de-asserted, but *readEn* signal is asserted, it changes its state to *empty* state since reg_0 register has been read. In this state, *empty* signal is asserted.

- *full*: If *readEn* signal is asserted, it makes a transition to *almost-full* state while old reg_1 register is loaded into reg_0. In *full* state, *full* signal is asserted.

In a fully-pipelined operation, the latency of the proposed 2-D DCT architecture is optimal in the sense that there are no wasted clock cycles. That is,

- It takes at most 76 clock cycles for the first 2-D coefficient to appear at the output buffer outputs. Within this 76 clock cycles, 64 clock cycles are spent for computing 64 1-D coefficient, 8 clock cycles for loading eight rows of pixel into the pong buffer, 1 clock cycle loading the first column of 1-D coefficients into the pong buffer, 1 clock cycle computing the first 2-D coefficient and 2 clock cycles of buffering latency due to 88-bit register and output buffer.
- It takes at most 144 clock cycles for the computation of all 2-D coefficients and 146 clock cycles for all 64 2-D coefficients to be sent out.

2.2. Quantization in Zigzag Order

The 2-D coefficients Z_{ij} of an 8×8 block should be uniformly quantized according to the quantizer step size $1 \leq Q_{ij} \leq 255$ from an 8×8 matrix called the *quantization table*. Specifically, quantization is defined as division of each 2-D DCT coefficient by its corresponding quantizer step size and it is followed by rounding the value to the nearest integer:

$$Z_{ij}^{*} = round\ (Z_{ij} \div Q_{ij}),\ 0 \leq i, j \leq 7$$

where Z_{ij}^{*} is the quantized 2-D DCT coefficient.

The quantization results in that most of the 2-D coefficients towards the lower right corner of 8×8 matrix, which are high-frequency coefficients, are zero. The zigzag ordering is used to rearrange the two dimensional 2-D coefficients in a one dimensional vector so that the low-frequency coefficients are placed before the high-frequency coefficients in the vector so as to maximize the compression during the entropy coding stage.

In the literature, it is typical that the quantization and zigzag ordering are handled in two different components. In the proposed design of JPEG encoder, however, these two components are combined into one architecture in Figure 3 by performing the quantization in the zigzag order.

The zigzag ordering is achieved by a two state {*write_ram*, *read_ram*} FSM together with a 64-entry RAM and 64-entry look-up table (Zigzag Table) as follows:

- *write_ram*: The DCT coefficients that are received in the column major order from the 2-D architecture are written into the RAM one by one. While the last coefficient is being written, the FSM goes to the other state.
- *read_ram*: The RAM is read in the zigzag order defined by [1] that is available from Zigzag Table while *read* signal is asserted. After the last coefficient is read from the RAM, the FSM goes to *write_ram*. As a result, the FSM stays in this state for 64 clock cycles.

Then, any DCT coefficient Z_{ij} that is read from the RAM is multiplied by an integer constant $Q_{ij}^{*}=2048/Q_{ij}$, where the 64 values of Q_{ij}^{*} are kept in a look-up table (Quantization Table in Figure 3) and Q_{ij} is the related quantizer step size from the quantization table in [1]. Note that there are luminance and chrominance quantization tables, one of which is chosen by *chrom* signal accordingly. Then, there is a register controlled by a FSM with two states {*empty*, *full*}, which is the same as the one in 1-D DCT. This register receives the 24-bit multiplication result in every clock cycle if its *writeEn* is asserted, and if either it is in *empty* state or its *readEn* is asserted. After that, the 24-bit multiplication result from the register is rounded into 11-bit integer number by a rounding logic similar to the one in Figure 2, and the quantized 2-D DCT coefficients are written into the output buffer one by one for the next stage.

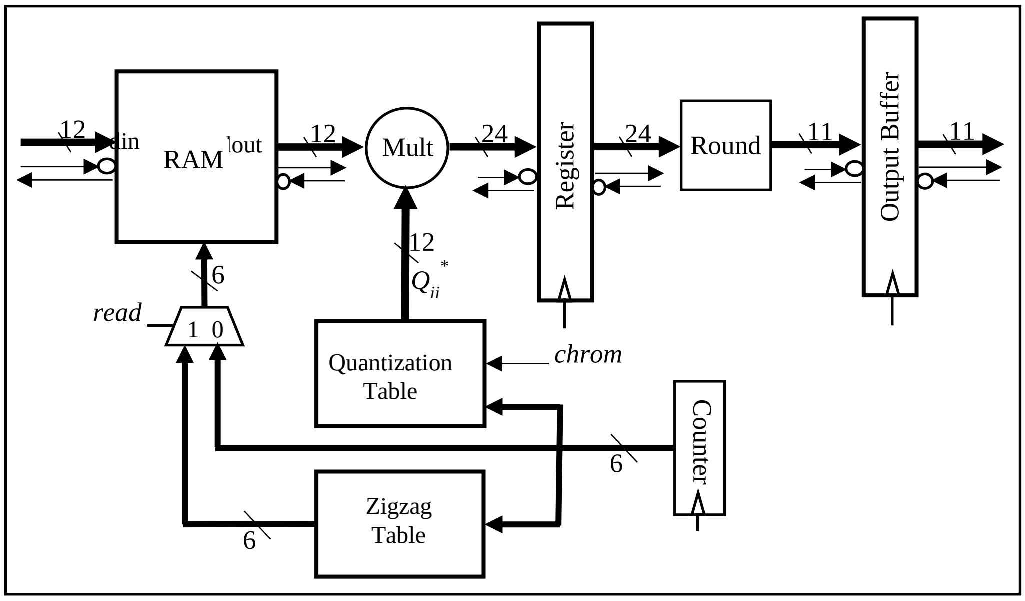

Figure 3. Quantization in zigzag order architecture

2.4. Entropy Coder Architecture

Entropy coding is the last step in JPEG compression, which loads the quantized 2-D DCT coefficients and provides to the output the compressed and assembled JPEG words. The entropy coding is achieved by three components, namely *run length coding*, *Huffmann coding,* and *assembler,* each of which is elaborated in detail below.

2.4.1. Run length coding

After the quantization in zigzag order, a vector of 64 coefficients will be ready for the entropy coding. Among these coefficients, the first element is called *DC component* and all other elements are known as *AC components*. According to the JPEG standard, the run length coding (RLC) must be applied to only AC components. In the proposed design of JPEG encoder, the RLC is simply achieved by a three state {*dc_coeff, ac_coeff, insert_zrl*} FSM followed by an output buffer as follows:

- *dc_coeff*: The FSM first receives the DC coefficient Z_{00}^* from the quantization architecture, inserts {*1'b1, 4'b0000*, Z_{00}^*} into its output buffer, and goes to *ac_coeff* state, where *1'b1* signals that this is the DC coefficient and *4'b0000* is the related run length of zeros.
- *ac_coeff*: The FSM receives the 63 AC coefficients Z_{ij}^* one by one, and there are several scenarios to handle:
  - If Z_{ij}^* is not the last coefficient:
    - If Z_{ij}^* is zero, *run_length* counter is incremented by one. If this is the sixteenth consecutive zero (*run_length*=15), *zrl_symbol* counter is also incremented by one.
    - If Z_{ij}^* is non-zero, there are two cases: If *zrl_symbol* counter is zero, {*1'b0, run_length*, Z_{ij}^*} is copied into the output buffer, and *run_length=0.* Otherwise, the FSM goes to *insert_zrl* state.
  - If Z_{ij}^* is the last coefficient:
    - If Z_{ij}^* is zero, {*1'b0, 4'b0000, 11'h000*} is put into the buffer, where {*0, 0*} is a special code known as *EOB* (end-of-block) symbol, and it goes to *dc_coeff* state.

  - If Z_{ij}^* is non-zero, there are two cases again: If *zrl_symbol* counter is zero, {*1'b0, run_length*, Z_{ij}^*} is copied into the output buffer, *run_length*=0, and it goes to *dc_coeff* state. Otherwise, the FSM goes to *insert_zrl* state.
- *insert_zrl*: While *zrl_symbol* counter is non-zero, {*1'b0, 4'b1111, 11'h000*} is inserted into the buffer, where {*15, 0*} is a special code known as *ZRL* (zero run length) symbol, and *zrl_symbol* is decremented by one. Once *zrl_symbol* is zero, {*1'b0, run_length*, Z_{ij}^*} is sent to the buffer, and the FSM goes to *ac_coeff* if Z_{ij}^* is not the last AC coefficient. Otherwise, it goes to *dc_coeff* for the next block.

According to this FSM, it takes 64 clock cycles at best or 70 clock cycles at worst to run length code 64 coefficients, which is sufficient enough to support a fully-pipelined operation without any stall clock cycles.

2.4.2. Huffmann coding

The output of RLC is a 16-bit word in the form of {1-bit *DC flag*, 4-bit *run length*, 11-bit *DC/AC coefficent*}. The Huffmann coding architecture proposed in Figure 4 receives such run length coded words and provides the Huffmann codes in a pipelined fashion as follows:

- *Stage-1 (Receive)*: When a new output word is available from RLC, the registers in the first stage of pipeline latches the new word as *zrl*= 4-bit run length, Z_{ij}^*=11-bit *DC/AC* coefficient, *dc*=1-bit DC flag. In addition to these received signals, 11-bit *diff* and 1-bit *lumin* signals are computed by the *differential coder* and the *chrominance block marker* architectures, respectively, and sent to the first stage.
  According to the JPEG specification, the DC components must be first differentially coded by a simple subtraction between the DC component (Z_{00}^*) of the current 8×8 block and the DC component (*prev*Z_{00}^*) of the previous block from the same source image component (Y, C_b, C_r). Even though it is not shown in Figure 4, the differential coder architecture consists of one adder and three registers in order to store the previous DC components of luminance and chrominance components. As a result, when the reception of a new DC component is signaled by asserting 1-bit *dc* signal, *diff* = Z_{00}^*- *prev*Z_{00}^* and Z_{00}^* is written into the related register.
  On the other hand, the luminance block marker is a simple counter that asserts its *lumin* output signal only if the current block being processed by the Huffmann coder is a luminance source image component.
- *Stage-*2 *(Category Selection):* The differentially coded DC coefficients and AC coefficients are passed through an encoder shown as the *cat (category selection) block* in Figure 4. That is, the cat block receives 11-bit coefficients and encodes them to 4-bit numbers according to the JPEG standard, where a 4-bit number indicates how many bits are significand in the current coefficient.
- *Stage-3 (Huffmann coding):* According to the JPEG standard, Z_{ij}^* (*x* in Figure 4) is decremented by one if Z_{ij}^* is negative (*signx*=0 in Figure 4); otherwise, it is kept the same. For any DC coefficient, its 4-bit category value is 15-bit Huffmann coded (11-bit Huffmann code and 4-bit Huffmann code length) by means of either the luminance DC Table or the chrominance DC table selected by *lumin2* signal. On the other hand, for each AC coefficient, its 4-bit run-length and 4-bit category value are first concatenated to form an 8-bit signal. Then, this 8-bit signal is 20-bit Huffmann coded (16-bit Huffmann code and 4-bit Huffmann code length) based on either the luminance AC Table or the chrominance AC table chosen by *lumin2* signal. Note that all DC and AC tables used are taken from the JPEG standard document [1].
- *Stage-4 (Send):* The output buffer is written by the following signals: 11-bit Z_{ij}^+ ($Z_{ij}^+ = Z_{ij}^*$ if $Z_{ij}^* \geq 0$; otherwise, $Z_{ij}^+ = Z_{ij}^*$-1), 4-bit category, 16-bit Huffmann code, and 5-bit Huffmann code length. Note that 4-bit Huffmann code length for each AC code word is kept as the decremented by one from its real code length in the AC tables to optimize the area usage. In order compensate this

decrementation, the 4-bit Huffmann code length for each AC code word is incremented by one before writing into the output buffer.

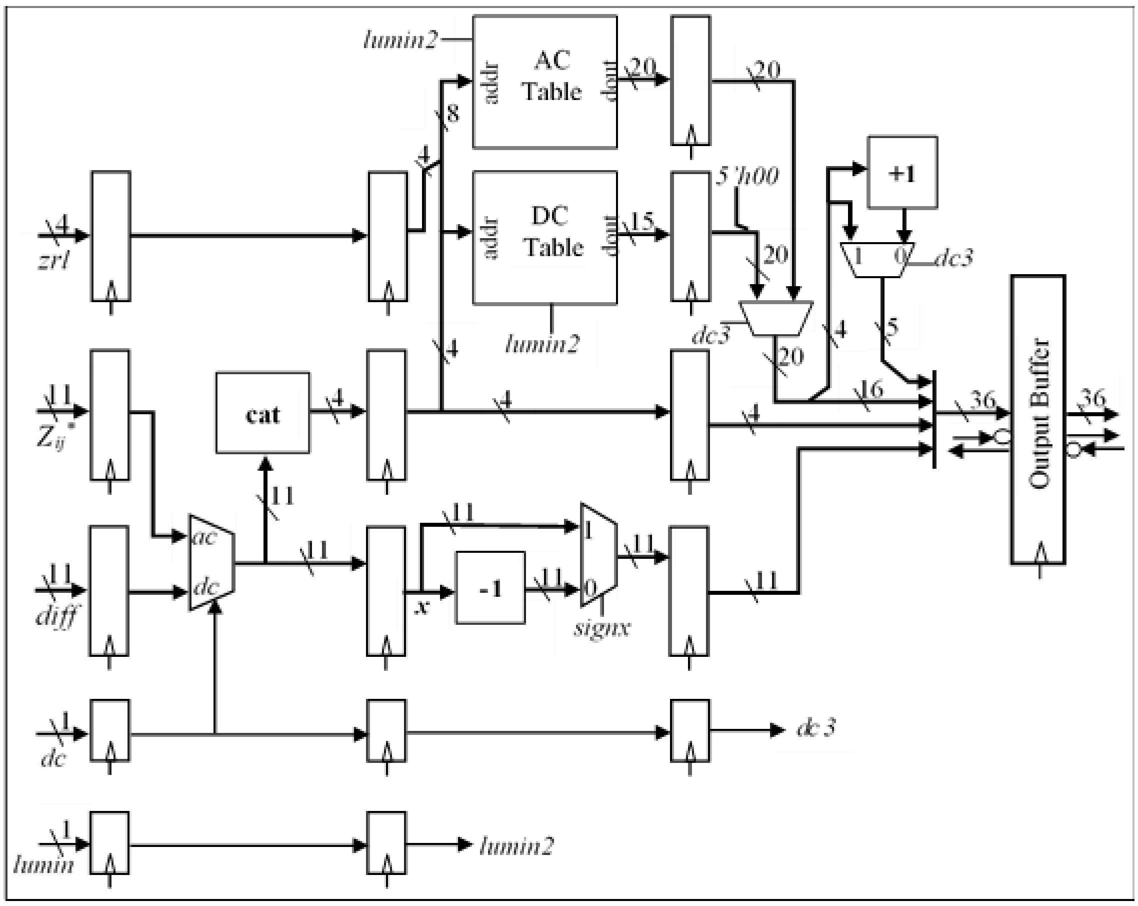

Figure 4. Huffmann coding in a pipelined fashion

2.4.3. Assembler

The assembler in Figure 5 is used to convert the variable length compressed data coming from the Huffmann coder into a stream of 32-bit compressed data.

The assembler works as follows. In the first stage of pipeline, an OR-mask circuit is used to set the insignificant bits of 11-bit coefficient (Z_{ij}^{+}) to zero according to 4-bit category (*cat*) of this coefficient. Meanwhile, 16-bit Huffman code (H_{ij}) is variably left shifted a number of times according to the same category value by a *Shifter* in order to bit align the Huffman code with the masked coefficient data. Finally, the left shifted Huffman code is ORed with the masked coefficient data, and the result is written in register *A*. On the other hand, the total length of the Huffmann code and coefficient, which is the sum of category (*cat*) and Huffmann code length (*len*), is placed into register *length-A*. Note that the result length cannot exceed 27 bits since the maximum length of the Huffman code is 16 bits and the coefficient is 11 bits.

In the second stage of pipeline, the content of register *B* is first variably left shifted a number of times according to *length-A* by another *Shifter* in order to bit align with the content of register *A*, which currently keeps {Huffman code, masked category}. Then, the left shifted register *B* is ORed with register *A*, and the result and its total length are stored into register *B* and register *length-B*, respectively. Note that the next state of register *length-B* can be *length-A+length-B* (output buffer cannot accept a new word), *length-B-32* (no new data is available from register *A*), or *length-A+length-B-32* (output buffer accepts a new word and register *A* has new data).

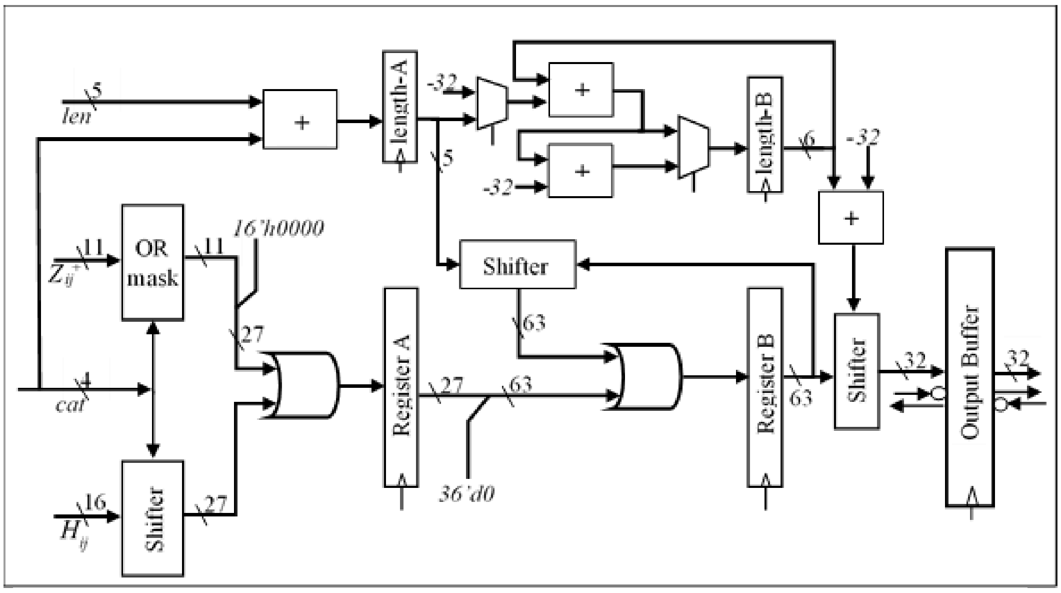

Figure 5. Assembler architecture in a pipelined fashion

In the third and final stage of the pipeline, a new 32-bit data word is written into the output port whenever the *length-B* is equal to or greater than 32 and the output buffer is not full. In order to find out the next 32-bit word to be sent, register *B* is variably right shifted a number of times according to (*length-B-32).*

3. IMPLEMENTATION RESULTS

The JPEG IP core architecture proposed in the article is captured with Verilog HDL with a device independent form, simulated and verified by a series of testbenches using Xilinx ISim. It is synthesized using Xilinx ISE 14.7 for several Xilinx FPGAs including Xilinx Spartan 3 (XC3S1000-5FG320), Spartan 3E (XC4VSX35-12FF668), and Spartan 6 (XC6SLX75T-3FFG676) FPGA devices.

Table 1 summarizes the synthesis results for the JPEG IP core and its three main modules for Spartan 3 and Spartan 6 FPGAs. According to Table 1, the entropy coder uses the most of FPGA’s LUT and FF resources, followed by 2-D DCT and quantizer modules. FFs utilizations are similar on both Spartan 3 and Spartan 6 FPGAs, while LUTs utilizations are lower on Spartan 6 FPGA. This is due to the fact that Spartan 6 equipped with 6-input LUTs (as compared to 4-input LUTs of Spartan 3) provides more efficient combinational logic implementation. Furthermore, the AC Huffmann tables are mapped to two BRAMs on Spartan 6 instead of a LUT-based implementation on Spartan 3. The 2-D DCT module has the lowest operation frequency among three modules and defines the maximum operation frequency (Fmax) of the JPEG IP core. Finally, since Spartan 6 is a newer technology than Spartan 3, the complete design and all individual modules achieve better maximum operation frequencies.

Table 1. Synthesis results of the JPEG IP core for color images

	Spartan 3					Spartan 6				
	LUTs	FFs	BRAMs	Multipliers	Fmax	LUTs	FFs	BRAMs	Multipliers	Fmax
2-D DCT	655	356	--	4	74.32	491	351	--	4	108.34
Quantizer	225	40	--	1	90.63	112	35	--	1	124.79
Entropy coder	1684	445	--	--	96.87	970	461	2	--	167.64
JPEG IP Core	2485	810	--	5	74.32	1439	812	2	5	111.92

Remember that it takes at most 144 clock cycles for the computation of all 2-D coefficients. Furthermore, the fully pipelined design of the JPEG IP core allows it to process an 8×8 block of pixels in 144 clock cycles in a pipelined fashion. Thus, the JPEG IP core, when mapped to a Spartan 6 FPGA, reaches a minimum period of 1.28 μs and processing rates up to 49.74 Msamples/s. This processing rate is sufficient for compressing more than 143 and 71 SDTV frames per second with 720×480 gray scale and color pixels per frame, respectively. Furthermore, the IP core proposed can compress more than 53 and 26 HD Ready TV frames per second with 1280×720 gray scale and color pixels per frame, respectively. These results indicate that the JPEG IP core proposed implemented on a low cost FPGA like Xilinx Spartan 6 can be deployed as an IP core of an M-JPEG video compressor directed to both SDTV and HD Ready TV applications.

The proposed JPEG IP core is compared against four other competitive designs in Table 2. According to Table 2, it is evident that the proposed IP core is clearly superior to [6] and [7] in terms of both resource utilization and maximum operation frequency. Note that comparing Stratix IV with Virtex 5 is slightly unfair for the proposed design since [14] reports that Stratix IV is 35% faster than Virtex 5 and packs 1.8X more logic element than that of Virtex 5. The JPEG compressors of [3] and [5] are better than the proposed one in terms of the LUTs utilization, which is due to implementing the tables on BRAMs, and Fmax. On other hand, they tend to use more FF resources and multipliers. Thus, there is no clear winner among [3,5] and the proposed design.

Table 2. A comparison of the JPEG IP core with other IP cores in the literature

	Technology	**LUTs**	**FFs**	**BRAMs**	**Multipliers**	**Fmax**
[3]	Spartan 3	1724	1275	2	11	80.00
[5]	Spartan 3E	1671	2450	4	8	101.35
[6]	Spartan 6	3020	2549	--	3	83.33
[7]	Stratix IV	1918	915	2^1	47	100.00
Proposed	Spartan 3	2486	810	--	5	74.32
	Spartan 3E	2459	816	--	5	90.32
	Spartan 6	1439	812	2	5	111.92
	Virtex 5	1567	705	--	5	184.15

^{1} Approximately.

4. CONCLUSIONS

This article proposes a fully pipelined and low area JPEG encoder architecture. The key element of the proposed methodology is to carefully design the pipeline stages in order to increase the resource sharing and decrease the total clock cycles to complete the overall JPEG operation. Furthermore, we compare the achieved implementation results of this study with four different competitive designs from the open literature. In order to minimize the FPGA resource utilization as low as possible, the proposed architecture employs the row-column decomposition technique for 2-D DCT transformation step in JPEG encoding, which yields a saving of 1-D DCT resource utilization. The proposed low area design achieves high processing rates that is well-suited to various image and video compressing applications.

ACKNOWLEDGEMENTS

This work has been supported by Anadolu University Scientific Research Projects under contract number 1705F200.

REFERENCES

[1] The International Telegraph and Telephone Consultative Committee (CCITT). Information Technology – Digital Compression and Coding of Continuous-Tone Still Images – Requirements and Guidelines. Rec. T.81, 1992.

[2] Wallace GK. The JPEG still picture compression standard. IEEE T Consum Electr: 1992; 38: 18-34.

[3] van Dyck W, Smodic R, Hufnagl H, Berndorfer T. High-speed JPEG coder implementation for a smart camera. J Real-Time Image Pr 2006; 1: 63-68.

[4] Agostini LV, Silva IS, Bampi S. Multiplierless and fully pipelined JPEG compression soft IP targeting FPGAs. Microprocess Microsy 2007; 31: 487-497.

[5] Pradeepthi T, Ramesh AP. Pipelined architecture of 2D-DCT, quantization and zigzag process for JPEG image compression using VHDL. Int J VLSI Com (VLSICS) 2011; 2: 99-110.

[6] Swarna KSV, Raju YDS. Implementation of soft processor based SOC for JPEG compression on FPGA. ICTACT J Microelectron 2015; 1: 1-7.

[7] Kishore B, Kumar BKS, and Patil CR. FPGA based Simple and Fast JPEG Encryptor. J Real-Time Image Pr 2015; 10: 551-559.

[8] Kaddachi ML, Soudani A, Lecuire V, Torki K, Makkaoui L, Moureaux J-M. Low power hardware-based image compression solution for wireless camera sensor networks. Comp Stand Inter 2012; 34: 14-23.

[9] Kovac M, Ranganathan N. JAGUAR: a fully pipelined VLSI architecture for JPEG image compression standard. Proc IEEE 1995; 83: 247-258.

[10] Sun S-H, Lee S-J. A JPEG chip for image compression and decompression. J VLSI Signal Proc 2003; 35. 43-60.

[11] Hsia S-C, Wang S-H. Shift-register-based data transposition for cost-effective discrete cosine transform. IEEE T VLSI Syst 2007; 15: 725-728.

[12] Doğan A. An efficient low area implementation of 2-D DCT on FPGA. 9th International Conference on Electrical and Electronics Engineering (ELECO), 771-775.

[13] Sanjeevannanavar S, Nagamani N. Efficient design and FPGA implementation of JPEG encoder using Verilog HDL. International Conference on Nanoscience, Engineering and Technology, 2011; 584-588.

[14] Altera White Paper. 40-nm FPGAs: Architecture and Performance Comparison. 2008.