An 8×8 IDCT Implementation on an FPGA-Augmented TriMedia

Mihai Sima[†‡] Sorin Cotofana[†] Jos T.J. van Eijndhoven[‡]
Stamatis Vassiliadis[†] Kees Vissers[§]
[†]*Delft University of Technology, Delft, The Netherlands*
[‡]*Philips Research, Eindhoven, The Netherlands*
[§]*TriMedia Technologies, Inc., Milpitas, California, U.S.A.*
Phone: +31-(0)40-274-2593 Fax: +31-(0)40-274-4004
E-mail: M.Sima@et.tudelft.nl

Abstract

This paper presents an experiment which aims to assess the potential impact on performance yielded by augmenting a TriMedia/CPU64 processor with a reconfigurable core. We first propose the skeleton of an extension of the TriMedia/CPU64 architecture, which consists of a Reconfigurable Functional Unit (RFU) and the associated instructions. Then, we address the computation of the 8×8 IDCT on such extended TriMedia and propose a scheme to implement the 1-D IDCT operation on the RFU. When implemented on an ACEX EP1K100 FPGA from Altera, the proposed 1-D IDCT exhibits a latency of 16 and a recovery of 2 TriMedia (200 MHz) cycles, and occupies 42% of the device. By configuring the 1-D IDCT computing facility on the RFU at application load-time, a 2-D IDCT including all overheads can be computed with the throughput of 1/32 IDCT/cycle. This is an improvement of more than 40% over the standard TriMedia/CPU64.

1. Introduction

A common research question of today is the range of performance improvements that may be achieved by augmenting a general purpose processor with a reconfigurable core. The basic idea of such approach is to exploit both the general purpose processor capability to achieve medium performance for a large class of applications, and FPGA flexibility to implement application-specific computations. There have been various attempts to attach a reconfigurable core to a host processor in the last decade, most of them involving a simple general purpose processor [1, 2, 3, 4, 5, 6, 7, 8, 9]. This paper presents an experiment which aims to assess the potential impact on performance yielded by augmenting a TriMedia/CPU64 processor with a reconfigurable core.

We first propose the skeleton of an extension of TriMedia/CPU64 architecture, which encompasses a Reconfigurable Functional Unit (RFU) and the associated instructions. Computing facilities supporting operations of user-definable latency, recovery, and slot assignment can be configured on the RFU. In order to assess the potential of the proposed architectural extension, we chose an 8×8 Inverse Discrete Cosine Transform (IDCT) application as benchmark. An 8×8 IDCT coded with a modified 'Loeffler' algorithm [15] can be scheduled in the standard instruction set of TriMedia in 56 cycles [10]. Since the standard TriMedia provides good support for transposition and matrix storage, we decided to provide RFU-hardware support only for an 1-D IDCT computing facility. Mapped on an ACEX EP1K100 FPGA, the 1-D IDCT facility exhibits a latency of 16, a recovery of 2 TriMedia (200 MHz) cycles, and occupies 42% of the device. By configuring the 1-D IDCT computing facility on the RFU at application load-time, a 2-D IDCT including all overheads can be computed with a throughput of $1/32$ IDCT/cycle. This is an improvement of more than 40% over the standard TriMedia. Given the fact that TriMedia/CPU64 is a 5 issue-slot VLIW processor with 64-bit datapaths and a very rich multimedia instruction set [10], such an improvement within its target media processing domain [11] indicates that the TriMedia-FPGA hybrid is a promising approach.

The paper is organized as follows. For background purpose, we briefly present the most important issues related to IDCT theory and architecture of the reconfigurable core in Section 2. Section 3 describes the proposed architectural extension of TriMedia/CPU64. The current implementation of the 8×8 IDCT under the standard TriMedia, implementation issues of the 1-D IDCT computing resource on the FPGA, the execution scenario of the 2-D IDCT on the extended architecture, as well as experimental results are presented in Section 4. Section 5 completes the paper with some conclusions and closing remarks.

2. Background and preliminaries

In this section, we briefly present the theoretical background of the IDCT. We also review the architecture of the FPGA that we used as an experimental reconfigurable core.

2.1. IDCT theoretical background

The Inverse Discrete Cosine Transform is a highly computational intensive part of MPEG decoding, and is used in the JPEG (de-)compression of data as well. The transformation for an N point 1-D IDCT is defined by [13]:

$$x_i = \frac{2}{N} \sum_{u=0}^{N-1} K_u X_u \cos \frac{(2i+1)u\pi}{2N}$$

where X_u are the inputs, x_i are the outputs, and $K_u = \sqrt{1/2}$ for $u = 0$, otherwise is 1. For MPEG, a 2-D IDCT processes an 8×8 matrix X [14]:

$$x_{i,j} = \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} K_u K_v X_{u,v} \cos \frac{(2i+1)u\pi}{16} \cos \frac{(2j+1)v\pi}{16}$$

One strategy to compute the 2-D IDCT is the standard row-column separation. The 2-D transform is performed by applying the 1-D transform to each row (horizontal IDCTs) and subsequently to each column (vertical IDCTs) of the data matrix. This strategy can be combined with different 1-D IDCT algorithms to further reduce the computational complexity. One of the most efficient 1-D IDCT algorithm has been proposed by Loeffler [12]. A slightly different version of the Loeffler algorithm in which the $\sqrt{2}$ factors are moved around has been proposed by van Eijndhoven and Sijstermans [15]. In our experiment, we will use this modified algorithm (see Figure 1).

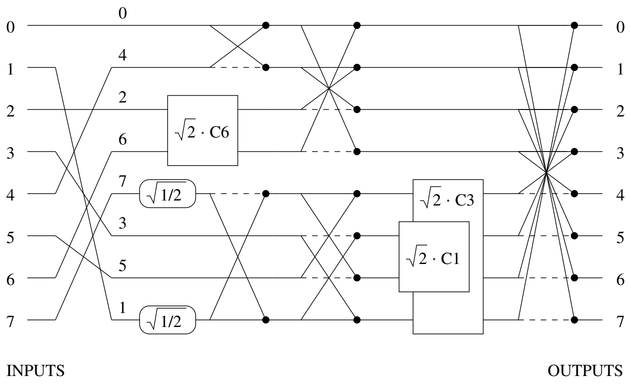
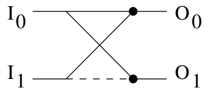


Figure 1. The modified 'Loeffler' algorithm.

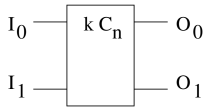
In the figure, the round block signifies a multiplication by $C_0' = \sqrt{1/2}$. The butterfly block and the associated equations are presented in Figure 2.



$$O_0 = I_0 + I_1$$
$$O_1 = I_0 - I_1$$

Figure 2. The butterfly – [12].

A square block depicts a rotation which transforms a pair $[I_0, I_1]$ into $[O_0, O_1]$. The symbol of a rotator and the associated equations are presented in Figure 3.



$$O_0 = I_0 k \cos \frac{n\pi}{16} - I_1 k \sin \frac{n\pi}{16} = C_n' I_0 - S_n' I_1$$
$$O_1 = I_0 k \sin \frac{n\pi}{16} + I_1 k \cos \frac{n\pi}{16} = S_n' I_0 + C_n' I_1$$

Figure 3. The rotator – [12].

Although an implementation of such a rotator with three multiplications and three additions is possible (Fig. 4 – a, b), we used the direct implementation of the rotator with four multiplications and two additions (Fig. 4 – c), because it shortens critical path and improves numerical accuracy. Indeed, there are three operations (two additions and a multiplication) on the critical path of the implementations with three multipliers, while the critical path of the implementation with four multipliers contains only two operations (a multiplication and an addition). Also, the initial addition involved by the three-multiplier implementations may lead to an overflow when fixed-point arithmetic is carried out.

2.2. The ACEX 1K FPGA architecture

The Altera ACEX 1K FPGA family [16] has been used as experimental platform for the reconfigurable core. This family has been chosen due to its favorable timing parameters. Also, this could allow future single-chip integration, as both ACEX 1K FPGAs and TriMedia are manufactured in the same TSMC technological process.

As the majority of Altera FPGAs, an ACEX 1K device contains an embedded array to implement memory and specialized logic functions, a logic array to implement general logic and interconnection network. The embedded array includes a number of Embedded Array Blocks. The logic array consists of a set of Logic Array Blocks, where each such block contains eight Logic Elements (LE) and a local interconnect network. A logic element is composed of a 4-input Look-Up Table (LUT), which can compute any function of four variables, a programmable flip-flop with a synchronous enable, a carry chain, and a cascade chain.

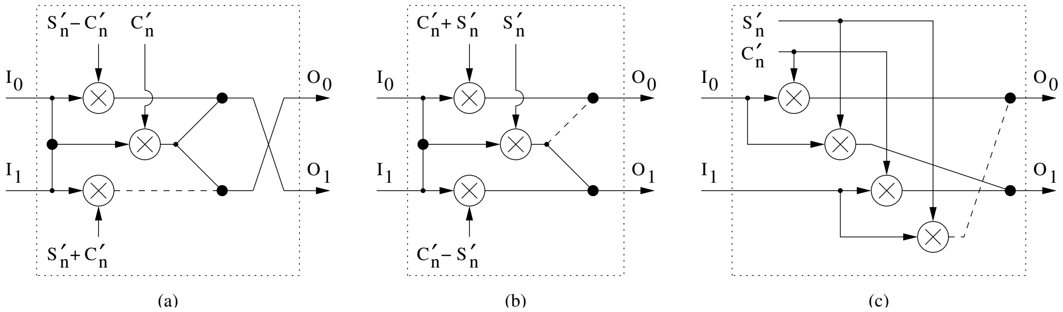


Figure 4. Three possible implementations of the rotator

Each LE drives both the local and a so called FastTrack interconnects. The simplified architecture of LE is depicted in Figure 5.

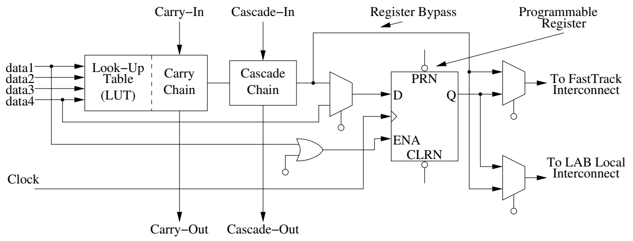


Figure 5. ACEX 1K Logic Element – [16]

The ACEX 1K architecture provides two types of dedicated high-speed data paths that connect adjacent LEs without using local interconnect paths: carry chains and cascade chains. The carry chain provides a very fast (as low as 0.2 ns) carry-forward function between LEs. The carry chain logic generates the carry-out signal, which is routed directly to the carry-in signal of the next-high-order bit. The carry-in signal feeds into both the LUT and the next portion of the carry chain. The cascade chain implements very-wide input functions with minimum delay. Adjacent LUTs can be used to compute portions of the function in parallel; the cascade chain serially connects the intermediate values, using an `AND` or an `OR` gate. Each additional LE provides four more inputs to the effective width of a function, with a delay as low as 0.7 ns per LE.

Signal interconnections within ACEX 1K devices and to and from device pins are provided by the FastTrack Interconnect, which is a series of fast, continuous row and columns channels that run the entire length and width of the device. Each I/O pin is fed by an I/O Element located at the end of each row and column of the FastTrack Interconnect. Each I/O Element contains a bidirectional I/O buffer and a flip-flop that can be used as either an output or input register to feed input, output, or bidirectional signals.

As a general view, we mention that the logic capacity of the ACEX 1K family ranges from 576 LEs for EP1K10 device to 4992 LEs for EP1K100 device. The maximum operating frequency for synchronous designs mapped on an ACEX 1K FPGA is 180 MHz.

The next section will introduce the architectural extension for the TriMedia/CPU64.

3. The skeleton of an architectural extension for TriMedia/CPU64

TriMedia/CPU64 is a 64-bit 5 issue-slot VLIW core, launching a long instruction every clock cycle [10]. It has a uniform 64-bit wordsize through all functional units, the register file, load/store units, on-chip highway and external memory. Each of the five operations in a single instruction can in principle read two register arguments and write one register result every clock cycle. In addition, each operation can be guarded with an optional (4^{th}) register for conditional execution without branch penalty. The architecture supports subword parallelism and is optimized with respect to media-processing. With the exception of floating point divide and square root, all functional units have a recovery[1] of 1, while their latency[2] varies from 1 to 4. The TriMedia/CPU64 VLIW core also supports double-slot operations, or super-operations. Such a super-operation occupies two neighboring slots in the VLIW instruction, and maps to a double-width functional unit. This way, operations with more than 2 arguments and/or more than one result are possible. The current organization of the TriMedia/CPU64 core is presented in Figure 6.

In the sequel, we will assume that the TriMedia/CPU64 processor is augmented with a Reconfigurable Functional Unit (RFU) which consists mainly of a reconfigurable array core. The reconfigurable functional unit is embedded into the TriMedia as any other hardwired functional unit, i.e., it receives instructions from the instruction decoder, reads its input arguments from and writes the computed values back

[1] Minimum number of clock cycles between the issue of successive operations.

[2] Clock cycles between the issue of an operation and availability of its results.

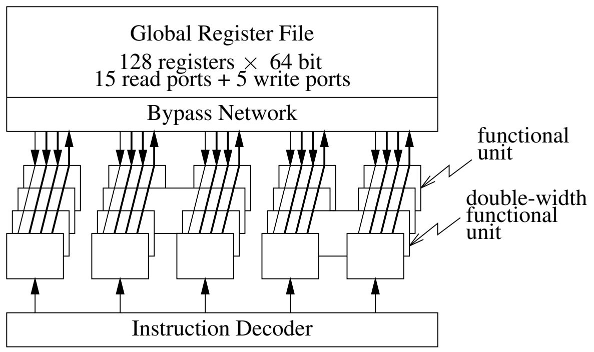


Figure 6. TriMedia/CPU64 organization – [10].

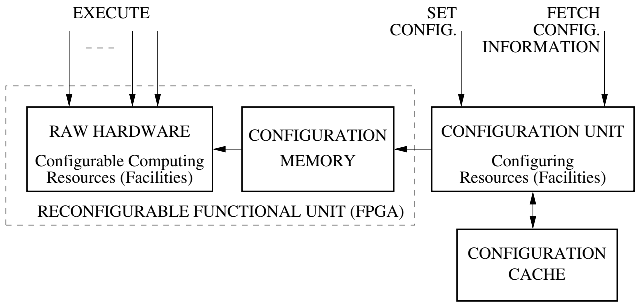


Figure 7. The proposed architectural extension for TriMedia/CPU64 VLIW core.

to the register file. In this way, only minimal modifications of the basic architecture, and also of the associated compiler and scheduler are required.

The RFU can be configured to be sensitive to an entire 5-slot VLIW instruction, i.e., to all five instructions issued simultaneously in the TriMedia/CPU64 slots. A number of computing facilities of different user-definable latency, recovery, and slot assignment (the issuing slot that the computing facility will be sensitive to) can be configured on RFU, and a set of single- or multi-slot operations can be performed by each facility. Therefore, the RFU can be configured to act as five independent functional units, each of them being controlled by an instruction in the VLIW, one 5-slot functional unit, or combinations of them. In all these situations, the RFU may use all 10 read and 5 write ports of the register file in a single cycle.

In order to use the RFU, a kernel of new instructions is needed. This kernel constitutes the extension of the TriMedia/CPU64 instruction set architecture we propose. A hardwired Configuration Unit which manages the reconfiguration of the raw hardware is associated to the RFU, as it is depicted in Figure 7. Generally speaking, the reconfiguration of the RFU is performed under the command of the `SET` instruction, while `EXECUTE` instructions launch the operations to be performed by the computing resources configured on the raw hardware [17]. In this way, the execution of an RFU-mapped operation requires two basic stages: a `SET` stage and an `EXECUTE` stage. A `FETCH` instruction could be used to load FPGA configuration information from Register File into a Configuration Cache.

Let us assume that multiple configurations can be stored in the configuration cache. By issuing a `SET` instruction, the configuration information mentioned by the first argument of `SET` is transferred from configuration cache into FPGA configuration memory[3] at a location specified by the second argument of `SET`. In this way, a flexible FPGA reconfiguration pattern is supported by the proposed architectural extension.

The user is given a set of `EXECUTE` instructions, which encompasses different operation patterns: single- or multi-slot operations, operations with an immediate argument, etc. It is the responsibility of the user to choose the appropriate `EXECUTE` instruction corresponding to the pattern of the operation to be executed. At the source code level, this may be done by setting up an *alias*, as it is described subsequently. Since the `EXECUTE` instructions are executed on the RFU with no checking of the configuration, it is still the responsibility of the user to perform the configuration management. More precisely, we provide no architectural support for such management; this task should be performed by the user in software. Since such management is outside the scope of the paper, we will not go into details.

For the semantics of an operation performed by a RFU-configured computing facility, its latency, recovery, and slot assignment are all user-definable, the source code of the application should contain information to augment the Machine Description File [18]. Such information will be used by the scheduler to schedule the newly defined operations. Assuming a user-defined `IDCT`, a way to specify such information is to annotate the source code, as follows:

```
.alias     IDCT  EXEC_3  ; specifies the IDCT alias
                         ; (EXECUTE_3 is a double-slot
                         ; operation with two inputs and
                         ; two outputs)
.latency   IDCT  7       ; specifies the IDCT latency
.recovery  IDCT  2       ; specifies the IDCT recovery
.slot      IDCT  1+2     ; specifies the slot assignment
                         ; of the IDCT instruction
```

In a similar way, the user can define as many RFU-related instructions as he/she wants. Since a single `EXECUTE` instruction may be used to specify different operations, multiple aliases for such instruction are possible.

[3] If the FPGA had been a multiple-context device, an `ACTIVATE` instruction would have swapped the current configuration.

In the subsequent experiment, we will define a single RFU-related operation which computes an 8-point 1-D IDCT. We will present implementation issues of the 1-D IDCT computing facility on the FPGA, and will evaluate the computing performance of an 8×8 IDCT.

4. Experimental results

In order to determine the potential impact on performance provided by the reconfigurable core, we will consider an 8×8 IDCT application as benchmark. The RFU is to be configured at application load-time, i.e., a burst of `FETCH` instructions followed by a `SET` instruction are scheduled on the top of the program code of the application. As we already mentioned, we will use an ACEX 1K FPGA from Altera as experimental platform for the reconfigurable core.

In the current implementation of the 2-D IDCT on the standard TriMedia/CPU64 architecture, all computations are done with 16-bit values, and make intense use of SIMD-style operations. The 8×8 matrix is stored in sixteen 64-bit words, each containing a half row of four 16-bit elements. Therefore, four 16-bit elements can be processed in parallel by a single word-wide operation. Next to that, being a 5-issue slot VLIW processor, TriMedia/CPU64 can execute 5 such operations per clock cycle.

This strategy is used for both the horizontal and vertical IDCTs. First, eight 1-D IDCTs (two SIMD 1-D IDCTs) are computed using the modified 'Loeffler' algorithm [15]. Then, the transpose of the 8×8 matrix is performed by a transpose unit which covers a double issue slot. Such a unit can generate the upper respectively lower two words of a transposed 4×4 matrix in one cycle. Therefore, the 8×8 matrix transpose is computed in eight basic operations. Finally, eight 1-D IDCTs (two SIMD 1-D IDCTs) are computed with the results generated by the transposition. Following the described procedure, a complete 2-D IDCT including all overheads (mostly composed of load and store operations) can be performed in 56 cycles [10].

Since the standard TriMedia provides a good support for transposition and matrix storage, we expect to get little benefit if we configure the entire 2-D IDCT into FPGA. Our goal is to balance the cost of storing the intermediate 2-D IDCT results into an FPGA-resident transpose matrix memory against obtaining free slots into TriMedia. Consequently, in our implementation on the extended TriMedia, we configure only an 1-D IDCT 2-slot computing resource on the RFU. By launching an 1-D IDCT super-operation having two 64-bit inputs and two 64-bit outputs, an 8-point 1-D IDCT is computed on eight 16-bit values. To calculate the 2-D IDCT, eight 1-D IDCT are firstly computed. Then a transpose is performed on the 8×8 data matrix using TriMedia native *byte shuffle* operations. Finally, eight 1-D IDCT are again computed. This execution scenario is presented in Figure 8.

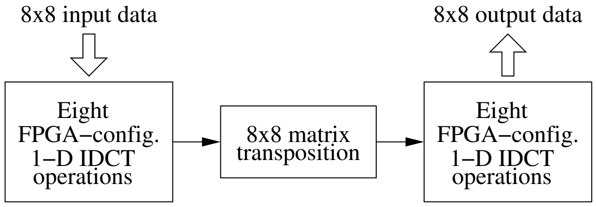


Figure 8. The computing scenario of 8×8 IDCT on extended TriMedia.

Let us assume a pipelined FPGA implementation of 1-D IDCT having a latency of 18 cycles[4], and a recovery of 1 which means that the FPGA clock frequency is equal with the TriMedia clock frequency. Unfortunately, such an FPGA clock cycle is prohibited for the FPGA we considered. The current TriMedia clock frequency is greater than 200 MHz, while the maximum allowable clock frequency for ACEX 1K is 180 MHz. Therefore, an 1-D IDCT hypothetical implementation having a recovery of 1 is not a realistic scenario. A recovery of 2 or more is mandatory. In the sequel, we will assume a recovery of 2 for 1-D IDCT and a 200 MHz TriMedia. This implies that the pipelined implementation of 1-D IDCT will work with a clock frequency of 100 MHz.

4.1. Implementation issues of the 1-D IDCT

Referring again to Section 2, and to Figures 1, 3, and 4, since the 1-D IDCT requires 14 multiplications, an efficient implementation of each multiplication is of crucial importance. For all multiplications, the multiplicand is a 16-bit signed integer represented in 2's complement notation, while the multiplier is a positive integer constant of 15 bits or less. A general multiplication scheme for which both multiplicand and multiplier are unknown at the implementation time exhibits the largest flexibility at the expenses of higher latency and larger area. If one of the operands is known at the implementation time, the flexibility of the general scheme becomes useless, and a customized implementation of the scheme will lead to an improved latency and area. A scheme which is optimized against one of the operands is referred to as *multiplication-by-constant*. Since such a scheme is more appropriate for our application, we will use it subsequently.

To implement the multiplication-by-constant scheme, we built a partial product matrix, where only the rows corresponding to a '1' in the multiplier are filled in. Then,

[4] As a fully combinatorial implementation of 1-D IDCT was found to have a delay which corresponds to 15-20 cycles at this clock rate, the assumption is realistic.

reduction schemes which fit into a pipeline stage running at 100 MHz are sought. It should be emphasized that a reduction algorithm which is optimum on a certain FPGA family may not be appropriate for a different FPGA family.

In connection with the partial product matrix, measured performances of several reduction modules for ACEX 1K are presented in Table 1. All the values in the table correspond to synchronous designs, i.e., both inputs and outputs are registered. The estimations have been obtained by compiling VHDL source codes with Leonardo Spectrum™, followed by a place and route procedure performed by MAX+PLUS II™. We would like to mention that although the figures typed in italics are generated by the software tools, they do not have real support, as the maximum operating frequency for the ACEX 1K is 180 MHz. The following settings of the software tools have been used: (1) Leonardo-Spectrum™: *Lock LCELLs*, *Map Cascades*, *Extended Optimization Effort*, *Optimize for Delay*, *Hierarchy: Flatten*, *Add I/O Pads*; (2) MaxPlus-II: *WYSIWYG*, *Optimize = 10 (Speed)*; (3) MaxPlus-II: *FAST*, *Optimize = 10 (Speed)*.

In order to implement an IDCT at 100 MHz, reduction modules which can run at 100 MHz or more should be considered. These modules are summarized below:

- Horizontal reductions of two, three, or four 16-bit lines to one line (Fig. 9 – a).
- Horizontal reduction of only two 30-bit lines to one line (Fig. 9 – b).
- Vertical reductions of three or four 7-bit columns to one line (Fig. 9 – c).
- Vertical reductions of six 5- or 6-bit columns to one line (Fig. 9 – d).

It should be mentioned here that Dadda population counters [19] of 3 or 4 bits can be implemented in only one logic level, i.e., with a delay of 0.6 ns [16] with two, respectively three LUTs. Also, Dadda counters of 5 or 6 bits can be implemented in two cascaded logic levels which exhibit a total delay of 1.2 ns, with seven LUTs. Although Dadda counters could theoretically be used as a reduction technique working at the same frequency with TriMedia, i.e., minimum 200 MHz, such an approach is limited by the maximum operating frequency of the ACEX 1K FPGAs: 180 MHz.

Reduction modules which can run at 100 MHz have been determined. Now we can start the implementation of each multiplication. We will present only the two most difficult examples: multiplications by $C'_0 = 5a82h$ which corresponds to the $\sqrt{1/2}$ block, and $C'_1 = 58c5h$, both of them belonging to the critical path of the modified 'Loeffler' algorithm. The number format is 2's complement fractional and the length of the word is 16 bit. In this way, only the most significant 16 bits of the product have to be stored.

Table 1. Performances of several reduction modules for ACEX 1K Speed Grade -1.

Reduction module		Performance $f$max – MHz		
		Leonardo-Spectrum(1)	MaxPlus-II WYSIWYG (2)	MaxPlus-II FAST (3)
Two-operand	16-bit adder	136	140	140
Three-operand		104	107	117
Four-operand		104	103	109
Five-operand		84	81	81
Six-operand		84	76	76
Two-operand	24-bit adder	112	114	114
Three-operand		89	94	94
Four-operand		89	86	90
Two-operand	28-bit adder	102	103	103
Three-operand		83	85	83
Four-operand		83	77	81
Two-operand	30-bit adder	98	102	102
Three-operand		88	93	91
Five-operand	3-bit adder	108	147	138
Six-operand		108	131	121
Seven-operand		108	128	116
Five-operand	4-bit adder	105	126	113
Six-operand		105	126	107
Seven-operand		105	111	114
Five-operand	6-bit adder	101	113	107
Six-operand		101	97	105
Seven-operand		101	94	97
Three inputs	Dadda population counter	*231*	*250*	*250*
Four inputs		*228*	*250*	*250*
Five inputs		155	175	169
Six inputs		155	*188*	*188*

The partial product matrix and the selected reduction modules and steps for multiplication by the constant C'_0 are presented in Figure 10. In the first step, the partial product matrix is built[5]. Then, reductions on the modules specified by the shaded areas are performed. The first stage generates four binary numbers of different lengths result, which are reduced to one row in the second stage. Therefore, a multiplication by the constant C'_0 is performed in two pipeline stages.

The partial product matrix and the selected reduction modules and steps for multiplication by the constant C'_1 are

[5] 's' represents the sign-bit.

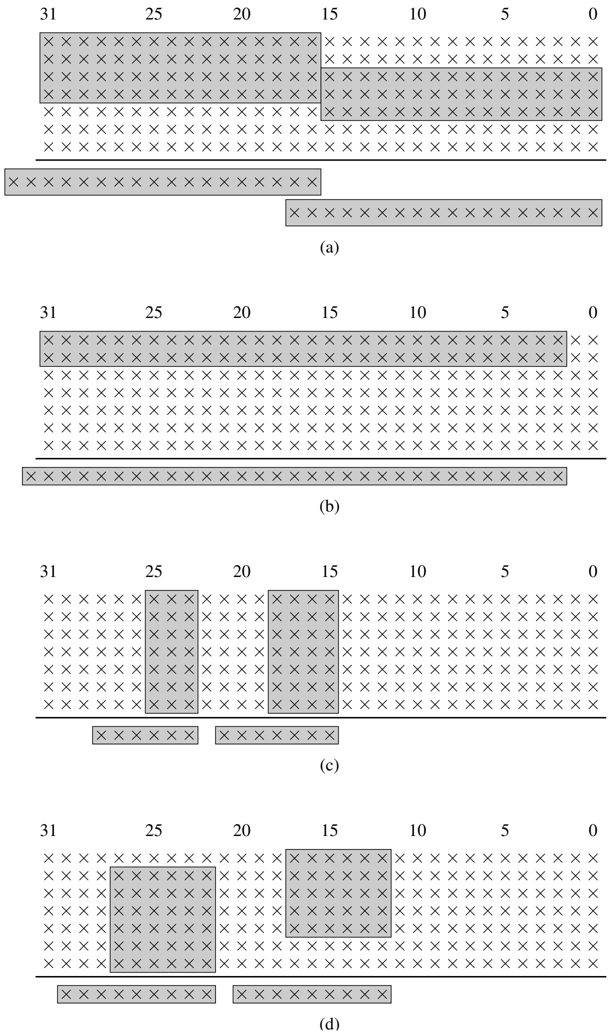


Figure 9. 100 MHz reduction modules on ACEX 1K.

presented in Figure 11. The reduction is performed in a horizontal way, two lines at a stage. Therefore, a multiplication by the constant C_1' is performed in three stages. The multiplication by the constant C_1' proved too difficult to be implemented in two stages only.

As a general rule, a horizontal reduction module consumes a lower area than a vertical reduction module of the same size. This situation occurs because a horizontal reduction module makes intensively use of the carry chain, as opposed to the vertical reduction module. A second observation is that the critical path of the 1-D IDCT is located on the odd part of the modified 'Loeffler' algorithm. Once the multiplication by constant C_1' is performed in three stages, there is no gain in performance to implement the other three multiplications, i.e., by constants S_1', C_3', S_3', in less than three stages. Therefore, the multiplications by the constants S_1',

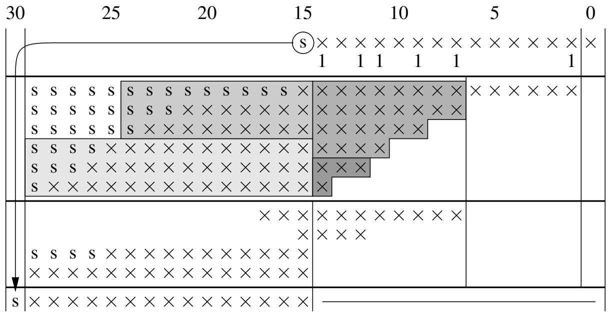


Figure 10. The partial product matrix and the selected reduction steps for multiplication by the constant C_0'

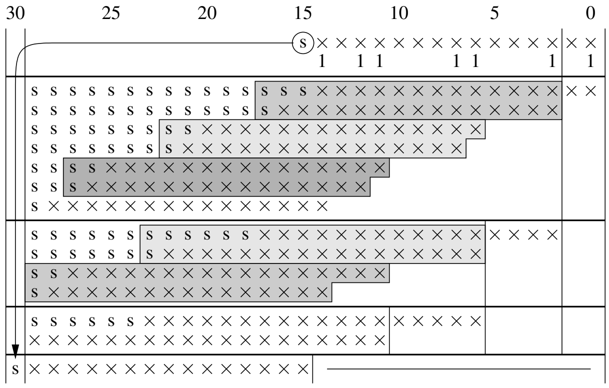


Figure 11. The partial product matrix and the selected reduction steps for multiplication by the constant C_1'

C_3', S_3' are implemented in three stages also, even though they may allow for an efficient (timing) implementation in two stages, too (however, at the expense of a slightly larger area). The same considerations apply for multiplications by the constants C_6' and S_6', as both of them are not located on the critical path. The sketch of the 1-D IDCT pipeline is depicted in Figure 12.

The latency of the 1-D IDCT is composed of:

- one TriMedia cycle for reading the input operands from the register file into the input flip-flops of the 1-D IDCT computing resource;
- two FPGA cycles for computing the multiplication by constant C_0';
- one FPGA cycle for computing the additions in the stage II of the algorithm;
- three FPGA cycles for computing the multiplication by constant C_1';

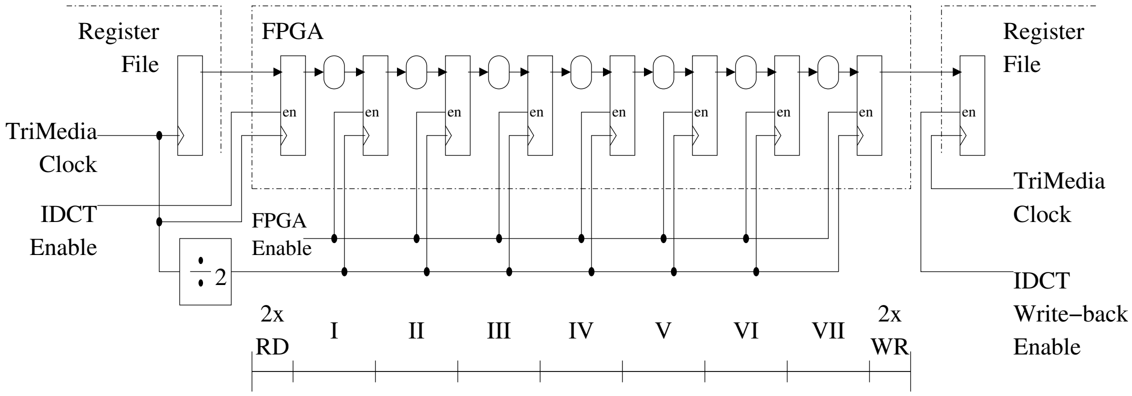


Figure 12. The 1-D IDCT pipeline

- one FPGA cycle for computing the additions in the last stage of the transform;
- one TriMedia cycle for writing back the results from the output flip-flops of the 1-D IDCT computing resource into the register file.

Therefore, the latency of the 8-point 1-D IDCT operation is $1 + (2 + 1 + 3 + 1) \times 2 + 1 = 16$ TriMedia cycles. We evaluated that 1-D IDCT uses 42% of an ACEX EP1K100 device and 257 I/O pins.

Finally, we would like to summarize the design tools:

- ModelSim™ SE/EE VHDL from Model Technology, version 54.b, revision 2000.06, has been used for simulating the VHDL source code.
- Leonardo-Spectrum™ from Exemplar, version v2000.1a2.75, has been used to generate the EDIF netlist file.
- MAX+PLUS II™ version 9.64 has been used to place and route the design available as an EDIF netlist file, and simulate the final mapped design.

4.2. 2-D IDCT implementation under the extended TriMedia/CPU64

As mentioned, an 1-D IDCT with a latency of 16 and a recovery of 2 is configured on the RFU at application load-time. We decided to assign the IDCT operation to the slot pair 1+2. After eight 1-D IDCT instructions, a burst of eight `TRANSPOSE` super-operations which computes the transpose of the 8×8 matrix are scheduled on the slot pairs 1+2, or 3+4. Then, eight 1-D IDCT instructions complete the 2-D IDCT. Before and after each 2-D IDCT, `LOAD` and `STORE` operations should be issued in order to fetch the input operands from main memory into register file, and store the results back into memory, respectively. The code was manually scheduled, and the result is presented in Figure 13.

In order to keep the pipeline full, back-to-back 1-D IDCT operation is needed. That is, a new 1-D IDCT instruction has to be issued every two cycles. Since true dependencies forbid issuing the last eight 1-D IDCTs of a 2-D IDCT so that to fulfill back-to-back requirement, the 2-D IDCTs are processed in chunks of two, in an interleaved fashion. A number of $2 \times 16 = 32$ registers are needed for this interleaved processing pattern. The computational performance of 2-D IDCT exhibits a throughput of $1/32$ IDCT/cycle and a latency of 42 cycles. It is worth mentioning that the machine is well balanced, none of the very-large instructions being fully occupied:

- `LOAD` or `STORE` operations are issued every other clock cycle, so the slots 4 and 5 are 50% used.
- `IDCT` super-operations are issued on slots $1 + 2$ every other clock cycle.
- The *transpose* super-operations are also issued on every other clock cycle, and the issuing slots can be either $1 + 2$ or $3 + 4$.

In this way, there are plenty of free slots which can be utilized for other purposes. Consequently, the announced figures represent the lower bound of the performance improvement which can be achieved on extended TriMedia.

In Table 2 we compare the performances of three 2-D IDCT implementations: on standard TriMedia [10], on FPGA-augmented TriMedia and on FPGA [20]. The 2-D IDCT implementation on standard TriMedia exhibits the lowest throughput (3.57M IDCT/sec), while the highest throughput (6.25M IDCT/sec) is achieved for the implementation under augmented TriMedia. That is, an improvement of 40% over standard TriMedia and 30% over FPGA has been obtained on extended TriMedia.

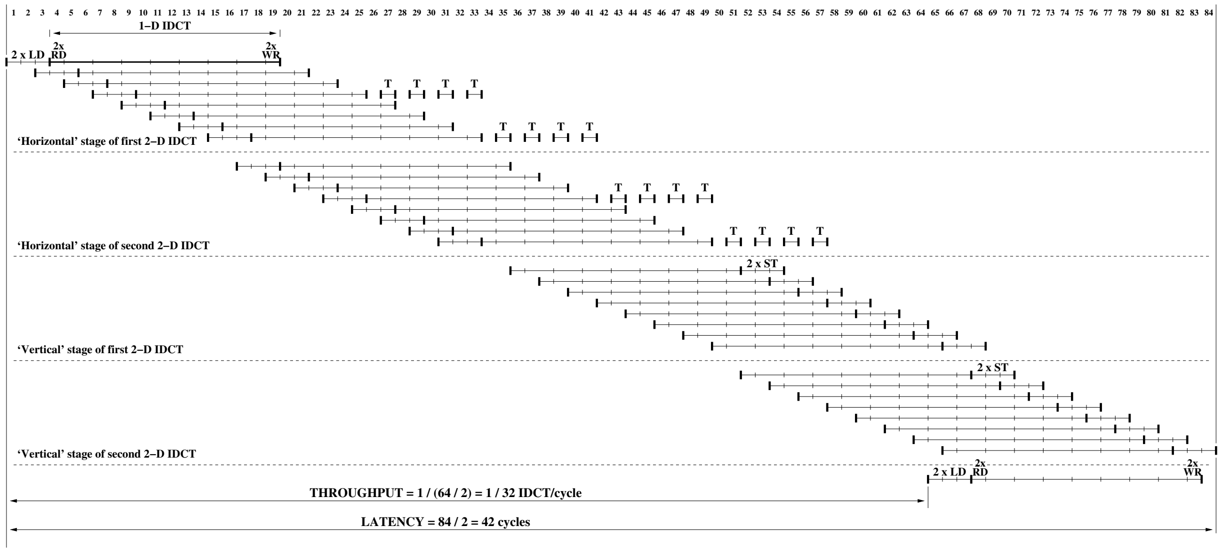


Figure 13. Schedule result for a 1-D IDCT having the latency of 16 and recovery of 2 (LD stands for `LOAD`, RD for read, WR for write, ST for `STORE`, and T for `TRANSPOSE`).

Table 2. 2-D IDCT performance figures for three 2-D IDCT implementations.

Implementation	FPGA family	Throughput		Latency		FPGA utilization
		IDCT/cycle	IDCT/sec	cycles	ns	
Standard TriMedia	n/a	1/56	3.57 M	56	280	n/a
FPGA-augmented TriMedia	EP1K100 (Altera)	1/32	6.25 M	42	210	42 %
FPGA alone	XCV600 (Xilinx)	n/a	4.09 ÷ 4.27 M	n/a	468 ÷ 489	88 ÷ 95 %

Finally, we would like to mention that in order to deal with the particularities of implementing computing resources with a different (lower) clock frequency than the TriMedia host, the scheduler should be changed to consider two additional requirements:

1. When the RFU pipeline is empty, scheduling a new RFU instruction can be done at every TriMedia cycle.
2. When the RFU pipeline is not empty, scheduling a new RFU instruction can be done only at any other TriMedia cycle.

5. Conclusions and future work

We have proposed an architectural extension for TriMedia/CPU64 which encompasses a reconfigurable functional unit and the associated instructions. On an FPGA-augmented TriMedia with three new `FETCH`, `SET`, and `EXECUTE` instructions, we obtained a performance improvement of 40% over standard TriMedia for an 8×8 IDCT application. 42% of an EP1K100 device has been used to implement the 1-D IDCT computing resource. As future work, we intend to define the complete architecture of the FPGA-extended TriMedia, consider more applications and operations for RFU, and evaluate the performance over the entire set of TriMedia dedicated multimedia-benchmarks using a cycle-accurate simulator.

Acknowledgements

The authors would like to thank Evert-Jan Pol with Philips Research for numerous helpful comments.

References

[1] M.J. Wirthlin, B.L. Hutchings, and K.L. Gilson, "The Nano Processor: A Low Resource Reconfigurable Processor," *Proceedings of 2nd IEEE Workshop on FPGAs for Custom Computing Machines* (FCCM '94), Napa Valley, California, Apr. 1994, pp. 23–30.

[2] R. Razdan and M.D. Smith, "A High Performance Microarchitecture with Hardware-Programmable Functional Units," *Proceedings of 27th Annual International Symposium on Microarchitecture* (MICRO-27), San Jose, California, Nov. 1994, pp. 172–180.

[3] M.J. Wirthlin and B.L. Hutchings, "A Dynamic Instruction Set Computer," *Proceedings of 3rd IEEE Symposium on FPGA-based Custom Computing Machines* (FCCM '95), Napa Valley, California, Apr. 1995, pp. 99–109.

[4] R.D. Wittig and P. Chow, "OneChip: An FPGA Processor With Reconfigurable Logic," *Proceedings of 4th IEEE Symposium on FPGA-based Custom Computing Machines* (FCCM '96), Napa Valley, California, Apr. 1996, pp. 126–135.

[5] J.R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," *Proceedings of 5th IEEE Symposium on FPGA-based Custom Computing Machines* (FCCM '97), Napa Valley, California, Apr. 1997, pp. 12–21.

[6] T. Miyamori and K. Olukotun, "A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications," *Proceedings of 6th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM '98), Napa Valley, California, Apr. 1998, pp. 2–11.

[7] S. Sawitzki, A. Gratz, and R.G. Spallek, "Increasing Microprocessor Performance with Tightly-Coupled Reconfigurable Logic Arrays," *Proceedings of 8th International Workshop on Field-Programmable Logic and Applications: From FPGAs to Computing Paradigm* (FPL '98), vol. 1482 of *Lecture Notes in Computer Science*, Tallin, Estonia, Sep. 1998, pp. 411–415.

[8] J.A. Jacob and P. Chow, "Memory Interfacing and Instruction Specification for Reconfigurable Processors," *Proceedings of 7th International Symposium on Field Programmable Gate Arrays* (FPGA '99), Monterey, California, Feb. 1999, pp. 145–154.

[9] B. Kastrup, A. Bink, and J. Hoogerbrugge, "ConCISe: A Compiler-Driven CPLD-Based Instruction Set Accelerator," *Proceedings of 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM '99), Napa Valley, California, Apr. 1999, pp. 92–100.

[10] J.T.J. van Eijndhoven, F.W. Sijstermans, K.A. Vissers, E.-J. Pol, M.J.A. Tromp, P. Struik, R.H.J. Bloks, P. van der Wolf, A.D. Pimentel, and H.P.E. Vranken, "TriMedia CPU64 Architecture," *Proceedings of International Conference on Computer Design* (ICCD '99), Austin, Texas, Oct. 1999, pp. 586–592.

[11] A.K. Riemens, K.A. Vissers, R.J. Schutten, F.W. Sijstermans, G.J. Hekstra, and G.D. La Hei, "TriMedia CPU64 Application Domain and Benchmark Suite," *Proceedings of International Conference on Computer Design* (ICCD '99), Austin, Texas, Oct. 1999, pp. 580–585.

[12] C. Loeffler, A. Ligtenberg, and G.S. Moschytz, "Practical Fast 1-D DCT Algorithms with 11 Multiplications," *Proceedings of International Conference on Acoustics, Speech, and Signal Processing* (ICASSP '89), Glasgow, Scotland, May 1989, pp. 988–991.

[13] K.R. Rao and P. Yip, *Discrete Cosine Transform. Algorithms, Advantages, Applications*, Academic Press, San Diego, California, 1990.

[14] J.L. Mitchell, W.B. Pennebaker, C.E. Fogg, and D.J. LeGall, *MPEG Video Compression Standard*, Chapman & Hall, New York, New York, 1996.

[15] J.T.J. van Eijndhoven and F.W. Sijstermans, "Data Processing Device and method of Computing the Cosine Transform of a Matrix," PCT Patent WO 9948025, to Koninklijke Philips Electronics, World Intellectual Property Organization, International Bureau, Sept. 1999.

[16] "ACEX 1K Programmable Logic Family," Datasheet, Altera Corporation, San Jose, California, Apr. 2000.

[17] M. Sima, S. Vassiliadis, S. Cotofana, J.T.J. van Eijndhoven, and K. Vissers, "A Taxonomy of Custom Computing Machines," *Proceedings of the First PROGRESS Workshop on Embedded Systems* (PROGRESS 2000), Utrecht, The Netherlands, Oct. 2000, pp. 87–93.

[18] E.-J. Pol, B.J.M. Aarts, J.T.J. van Eijndhoven, P. Struik, F.W. Sijstermans, M.J.A. Tromp, J.W. van de Waerdt, and P. van der Wolf, "TriMedia CPU64 Application Development Environment," *Proceedings of International Conference on Computer Design* (ICCD '99), Austin, Texas, Oct. 1999, pp. 593–598.

[19] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford University Press, New York, New York, 2000.

[20] K. Chaudhary, H. Verma, and S. Nag, "An Inverse Discrete Cosine Transform (IDCT) Implementation in Virtex for MPEG Video Application," Application Note 208, Xilinx Corporation, San Jose, California, Dec. 1999.