Iterative Design Method for Video Processors Based on an Architecture Design Language and its Application to ELA Deinterlacing

Gérard-A Bouyela Ngoyi, J.M. Pierre Langlois, and Yvon Savaria
Groupe de recherche en microélectronique et microsystèmes
École Polytechnique de Montréal, Canada
{gerard-armand.bouyela-ngoyi, pierre.langlois, yvon.savaria}@polymtl.ca

Abstract — This paper presents a design methodology for dedicated real-time video processors. The methodology begins with a basic processor that is progressively morphed into a specialized processor through five systematic steps. It differs from standard methodologies for ASIP design which place exclusive emphasis on the extension of the instruction set. The proposed methodology takes a global look at various processor and system considerations. The last step consists of removing unnecessary functionality from the instruction set. The required flexibility is attained by the use of an architectural description language. We use a basic deinterlacing algorithm to demonstrate the effectiveness of the methodology and present details of the various phases of the design process. Using ELA deinterlacing as a benchmark, the final processor uses 20% fewer logic elements, achieves a global acceleration by a factor of 11, and an improvement in area-delay product of 14, with respect to the basic processor.

I. INTRODUCTION

D eveloping real-time embedded video processors is necessary today with the ubiquitous deployment of multimedia systems. These systems require significant design efforts to meet stringent specifications in terms of area, data rate, power, energy consumption and flexibility [1].

In this paper, we describe a design methodology for custom video processors. We choose video deinterlacing with the ELA algorithm as benchmark. Deinterlacing restores missing lines from interlaced frames. Interlacing is a technique that reduces the video signal transmission rate by half. The video sequence is divided into odd and even frames containing only the odd or even lines of an image, respectively. This format is still used in broadcasting for compatibility reasons. However, a transformation is required in order to properly display the video on high definition televisions and digital video monitors [2][3]. The interlaced video signal must be converted to progressive scan video where frames contain all lines.

Simple deinterlacing approaches such as line doubling or vertical interpolation produce annoying artifacts, including a combing effect for moving objects and picture blurring. More advanced algorithms can restore picture quality at the expense of increased computational and memory costs.

This work was inspired by previous contributions from our research group [2][4]. The previous work targeted Xtensa reconfigurable processors, with an atomic central core to which specialized instructions can be added. Intrinsic core parameters such as the basic instruction set and the address and data bus width are fixed. These restrictions had become the bottleneck of the previous methodology. In order to remove them, the present paper proposes a design methodology exploiting the flexibility of an architecture description language (ADL) such as LISA 2.0 [5]. Furthermore, in [4], a more complex version of the deinterlacing algorithm was selected. Here, we consider a simpler, less parallelizable version of the algorithm to establish a lower bound on performance.

This paper is organized as follows. In section I, we briefly review relevant literature. Section II presents the proposed general methodology for the design of dedicated video processors. Section III demonstrates the application of the methodology with a specific application example. Sections V and VI present results and draw conclusions.

II. PREVIOUS WORK

There are generally three categories of video processors: ASICs, ASIPs, and general purpose. ASIC processors offer a designer the greatest flexibility, possibly by describing the processor down to the RTL level in a language like, for example, VHDL or Verilog. This specialization enables the designer to achieve the greatest levels of performance in terms of area and information rate at the expense of flexibility. At the other end of the spectrum, general-purpose processors allow a wide range of processing types, but at the expense of performance. ASIPs fall in between these two extremes, offering a compromise between flexibility and performance.

For example, Peters et al. presented a methodology based on the design flow of A|RT-Designer, which takes a description of an algorithm in C to generate a set of specialized instructions for the application [6]. The Xtensa configurable processors have also become very popular, letting designers configure and extend a basic processor, while being supported by a wide area of design tools [7]. However, a limitation of this approach is the limited set of atomic predefined options available to reduce processor complexity. A greater flexibility can be gained through the use of an architecture description language that gives a designer full flexibility. A popular development environment in this category is the Processor Designer from CoWare (previously known as Lisatek) [8].

Processor Designer is a design tool which allows reducing by several months the time required to design an integrated embedded-system solution. Included in the tool suite are templates of basic architectures for different types of processors (DSP, RISC, SIMD, and VLIW). Indeed, depending on the
nature of the application, the starting point may be any of these different types of processors. The proposed iterative design loop allows, regardless of the type of processors, on the one hand to scale the processor (i.e. adjusting the data path) and, on the other hand, to insert specialized functions.

These simple models ensure compatibility with an instruction set simulator, code profiler, debugger, and RTL synthesis and verification tools. They reduce the design effort needed to consider various levels of abstraction [9].

This development environment for dedicated processors has already been proven efficient in various projects. Hoffman et al. presented a state of the art ASIP design with Processor Designer [10]. Saponara et al. made a comparison between video processors designed with Lisatek and a DSP found on the market [8]. This comparison revealed a decrease in power consumption of approximately 90% (50 nJ/pixel instead of 580 nJ/pixel) for the Lisatek-based design. Schuster et al. have highlighted the flexibility of the design methodology with a VLIW processor [11].

An ADL-based methodology offers the possibility of fully customizing a processor to meet application specific constraints. This includes specifying the size of various buses and data-path component widths to the bit-level granularity.

III. GENERAL METHODOLOGY

The first step of the proposed methodology is to use a processor with a basic instruction set, described with an ADL in an environment like A|RT-Designer or Processor Designer. This processor can be selected from several architectures such as RISC, SIMD, VLIW or DSP. These models have a range of instructions similar to a general-purpose processor. We assume that a compiler exists that can target code for the processor. We consider a basic RISC processor with a four-stage pipeline, namely FD (fetch), DC (decode), EX (execute) and WB (write back), as shown in Fig.1.

The proposed design methodology takes, as input, a target C code and a basic architecture described in an ADL. The iterative process is shown in Fig. 2. The main loop should usually be performed several times. This loop corresponds loosely to traditional processor configuration design flows, but this methodology systematizes the different actions that can be taken. In general, the actions concern the parameterization and extension of the basic processor [12]. A fifth action taken in the last step removes unnecessary components from the processor. The listed actions are detailed as follows:

1) Decomposition of high-level code into specialized functions

The first step consists of separating sections of the high-level source code into smaller functions for which dedicated hardware is developed. This decomposition has the negative effect of increasing the number of cycles to execute the program due to context switching. Dedicated hardware is then defined for each function in a language such as LISA 2.0. Function calls are redirected to the specialized hardware.

2) Processor scaling

This step corresponds to a change in the widths of the local data and/or address buses. Depending on the processing involved, it is possible that the basic processor core is ill suited to the data. For example, one could imagine a 16-bit processor dealing with 24-bit data and with a 20-bit address space. In such a case, the data and address buses should be adapted to the application. Functional units must also be scaled accordingly. Not all architectures are configurable to this extent, but the use of an ADL, by definition, gives this flexibility.

3) Increase of the granularity of the specialized functions

If performance targets are still not met, the granularity of the specialized functions is increased. This implies combining larger blocks of high-level code and instantiating increasingly complex specialized functions. This action may increase the critical path of the processor. One way to alleviate this problem is to add another pipeline stage to the processor or to spread some instructions over several cycles. The operating frequency of the processor should not be reduced.

4) Addition of customized register files

A fourth action consists of defining customized register files. This effectively relocates data closer to the processor and reduces memory bandwidth requirements. Intermediate results can therefore be made accessible without incurring time-consuming memory access penalties.

5) Removal of unused instructions

This last action is normally taken only once, and only after performance targets have been met. It consists of enumerating all necessary instructions from the compiled C code and removing all others from the instruction set. This final step effectively reduces the area occupied by the processor. It also tends to shorten the critical path, mainly due to the simplification to instruction decoding.
IV. METHODOLOGY APPLIED TO VIDEO DEINTERLACING

We tested our methodology with the Processor Designer tool suite from CoWare. As a starting point for our exploration, we considered a 32-bit RISC processor implementing Edge-based Line Average (ELA) deinterlacing [2].

ELA is an intra-field deinterlacing method. It relies on spatial interpolation techniques to calculate the missing lines of a frame. It calculates each missing pixel based on the values of neighboring pixels in the same frame. To estimate a missing pixel, the algorithm uses six pixels: three pixels from the line above and three pixels from the line below the missing line. It first determines the interpolation direction according to the minimum of the following:

\[ a = \text{abs}(X(k-1,n-1) - X(k+1,n+1)) \text{ (diagonal left to right)} \]
\[ b = \text{abs}(X(k-1,n) - X(k+1,n)) \text{ (vertical)} \]
\[ c = \text{abs}(X(k-1,n+1) - X(k+1,n-1)) \text{ (diagonal right to left)} \]

Figure 3 Basic ELA

Once the minimum between a, b, and c is found, the missing pixel is calculated by performing an average of the pixels in the corresponding direction.

The five actions proposed in our methodology are detailed below for the implementation of this algorithm. For each action, we produced a different version of the processor, and the characteristics of each are given in Table 2. The starting architecture is called RISC_V1.

1) Decomposition of C code and creation of basic specialized instructions

In the first refinement step, we decomposed the initial high-level source code into three functions: \text{abs()}, \text{minimum}(), and \text{average}(). A specialized instruction was defined for each function. The first calculates the absolute value, while the second provides the direction of interpolation from three quantities. The third function calculates the missing pixel as the average of two pixels along the selected edge. This step represents the transition from RISC_V1 to RISC_V2 in Table 2. It results in an acceleration of 11%, but the number of LUTs increases by 51%.

2) Reduction of address bus widths from 32 to 16 bits

The selected basic RISC processor had a 32-bit instruction and data memory address space. However, for this application, 16-bit address buses are sufficient. Consequently, by halving the address buses, it is possible to encode effective addresses inside 32-bit instruction words. The effect on code size is dramatic; there is also a minor improvement in processor complexity. This type of transformation is not typically possible with other design methodologies that impose fixed bus widths. This step corresponds to the transition from RISC_V2 to RISC_V3 in Table 2. The number of cycles is reduced by 23% and the number of LUTs by 5%.

3) Increase of granularity

In this step, the three customized instructions are combined into one, \text{ela}(). This function takes as parameters three pixel values as from the upper line and three pixels from the lower line. It determines the direction of interpolation and returns the value of the missing pixel. The main loop of the code is shown in Figure 4. The number of cycles is reduced by more than 85% while the critical path remains largely unchanged. The penalty is an increase of 19% in the number of LUTs. This step corresponds to the transition from RISC_V3 to RISC_V4 in Table 2.

![Figure 4. Basic ELA C code modified](image)

This step corresponds to the transition from RISC_V4 to RISC_V5. We created three additional functions: \text{init_reg()}, \text{shiftwin()} and a version of \text{ela}() without parameters. The main processing loop is shown in Figure 5.

![Figure 5. Basic ELA C code modified](image)

Function \text{init_reg()} initializes the 8 specialized registers which contain data pixel values. Function \text{shiftwin()} allows the reuse of previously read pixels. Indeed, six pixels are involved in the calculation of each missing pixel (Fig. 6). However, four of these can be reused for the following calculation. Finally, function \text{elav2()} determines the direction of interpolation and it calculates the missing pixels.

![Figure 6. Sliding window](image)

Total program time increases and more logic resources are required. However, this version includes all customized instructions from previous versions. Furthermore, the code shown in Fig. 5 was purposely kept general, without the necessary manual loop unrolling shown in Fig. 4. Thus, a loop overhead penalty is incurred. That combining loop unrolling with the instructions of step 4 is possible but was not done.

This version has the advantage of requiring fewer total cycles. The increase in total program time, however, is an indication that it is time to move on to the last step of the methodology.

5) Removal of unused instructions

The removal of unused instructions was done manually by first examining the assembly code generated by the compiler.
All unnecessary instructions from the instruction set were removed from the processor description. In Table 2, the last version, V6, represents the removal of all basic unused functions and all specialized instructions added during the process of architectural exploration. From these, we kept only 18 basic instructions in addition to three specialized instructions. They are shown in Table 1.

### Table I


<table>
<thead>
<tr>
<th>Instruction</th>
<th>Add</th>
<th>Addiu</th>
<th>andi</th>
<th>b</th>
<th>ba</th>
<th>bne</th>
<th>cmplt</th>
<th>elav2</th>
<th>Init</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>Br</td>
<td>B</td>
<td>Br</td>
<td>CMP</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>Used (%)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

This step reduces the number of LUTs by 62% while increasing the maximum frequency by 7% from the previous step. Remarkably, the number of LUTs is reduced by 27% from the original, basic processor.

### V. Results and Discussion

The system was synthesized for a Virtex II Pro FPGA. Table 2 summarizes synthesis results for different steps of architectural exploration.

### Table II

**Synthesis results for all developed architectures**

<table>
<thead>
<tr>
<th>Architecture Description</th>
<th>Area in Virtex 2 Pro FPGA (2:Fizz967-6)</th>
<th>Number of Slices</th>
<th>Number of Flip Flops</th>
<th>Maximum Frequency (MHz)</th>
<th>Cycle (complex)</th>
<th>Number of Cycles</th>
<th>Time to calculate one 720 × 480 image (ms)</th>
<th>Normalized Area-Delay Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>RISC V1</td>
<td></td>
<td>4748 (34%)</td>
<td>1027 (3%)</td>
<td>9199 (33%)</td>
<td>83.8</td>
<td>80</td>
<td>260K</td>
<td>745</td>
</tr>
<tr>
<td>RISC V2</td>
<td></td>
<td>7277 (53%)</td>
<td>1638 (5%)</td>
<td>13918 (50%)</td>
<td>84.7</td>
<td>35</td>
<td>235K</td>
<td>667</td>
</tr>
<tr>
<td>RISC V3</td>
<td></td>
<td>6863 (50%)</td>
<td>1282 (4%)</td>
<td>13161 (48%)</td>
<td>83.4</td>
<td>35</td>
<td>180K</td>
<td>518</td>
</tr>
<tr>
<td>RISC V4</td>
<td></td>
<td>8257 (60%)</td>
<td>1371 (5%)</td>
<td>15628 (57%)</td>
<td>82.4</td>
<td>22</td>
<td>22.3K</td>
<td>65.1</td>
</tr>
<tr>
<td>RISC V5</td>
<td></td>
<td>9521 (69%)</td>
<td>1576 (5%)</td>
<td>17835 (65%)</td>
<td>75.2</td>
<td>24</td>
<td>22K</td>
<td>70.2</td>
</tr>
<tr>
<td>RISC V6</td>
<td></td>
<td>3678 (26%)</td>
<td>1447 (5%)</td>
<td>6681 (24%)</td>
<td>80.4</td>
<td>24</td>
<td>22K</td>
<td>65.7</td>
</tr>
</tbody>
</table>

Table 2 includes an area-delay metric to gain insight on these performance metrics of the various architectural choices. For the area term, we used the sum of the number of LUTs and flip-flops. Program time is measured for a full NTSC frame (720 × 480 pixels). We observe that there is a progressive reduction in execution time from one architecture to the next due to the inclusion of specialized functions (Fig.7). However, the critical path delay increases because the different specialized functions are becoming increasingly complex (increased granularity).

### VI. Conclusion

We proposed a method for the design of dedicated video processors based on a series of addition and/or withdrawal of specialized and basic instructions. While maintaining a high-level description, the code is accelerated by a factor of 11 and the area-delay product is improved by a factor of 14. The methodology was tested from a RISC32 processor and an algorithm for image processing (ELA). We propose a design methodology based on an iterative loop that allows specializing a basic processor core to increase its performance. We used an ADL to design our dedicated processor while remaining at a high level. We did not use standard processors, which offer a set of extensible instruction with fixed parameters and data paths, but we rather leveraged an ADL that offers more flexibility in describing dedicated processors.