Index

EXECUTIVE SUMMARY

A flexible and reconfigurable processor for motion estimation has been developed by the Bristol Microelectronics group. The processor is designed to run fast-search block matching motion estimation algorithms for several hybrid video codecs like H.264. The processor core can be configured according to a set of constraints like required frame rate and logic resources available on the implementation platform. The core is also able to optimize its microarchitecture to the motion estimation algorithm selected to better meet these constraints. As the whole motion estimation process accounts for a very large portion of the encoding and decoding process time budget (up to 80%) this directly implies that most of the power consumption required to process video frames is wasted in this process. The processor is described using VHDL code.

The main task of the project was to reduce the overall power consumption of the motion estimation processor. This was done by exploring three different power consumption reduction features. The first one was the use of the design implementations tools to enhance the processor performance. The second was to execute various motion estimation search algorithms on the processor and evaluate its performance in terms of power and energy. Lastly, a shallow pipeline for the processor was created by making modifications to the processor architecture.

To evaluate the results of these techniques, the processor power consumption was benchmarked using of post-place & route timing simulations and data processing with a specialized power analysis tool. (Refer to section 3.1)

The final results of this project provide a solid insight and practical approach into various ways to reduce the power consumption of the processor. These results can be outlined as follows:

- A comprehensive study of FPGA power reduction techniques and a comparative evaluation an analysis into how these techniques could be implemented into the processor. (See Chapters 1 and 2)
- An optimal configuration to be used with the design implementation tools was obtained. This configuration improves both speed and power performance of the processor. (See sections 3.2 and 4.1)
- A dataset of results that provide a useful reference on how different motion estimation algorithms fare in terms of power, and how this can be used to reduce the processor energy consumption. (See sections 3.3 and 4.2)
- A shallow pipeline design that allows to reduce power consumption, processing time, and resources allocated for the processor implementation on an FPGA (See sections 3.4 and 4.3)

INTRODUCTION

This section provides a brief overview to the project, describes the aims and objectives of the present work and its general structure.

Presentation

Motion estimation is typically the most compute intensive part of a modern video coding algorithm such as H.264. It can represent between 50-80% of the total complexity. Motion estimation attempts to find the pixel area in the reference area most similar to the pixel area in the current frame and the approach used for this search has large implications on the overall performance of the algorithm and its complexity. Well known fast motion estimation algorithms include logarithmic search, three-step search, diamond search, hexagon search, etc. along with more advanced methods such as PMVFAST and UMH. The University of Bristol Microelectronics Group has developed a programmable motion estimation processor known as LiquidMotion. The processor is optimized for fast motion estimation search algorithms and it has been successfully tested in a FPGA board attached to a standard computer with a PCI interface. It is designed to execute user-defined block-matching motion estimation algorithms optimized for hybrid video codecs such as MPEG-2, MPEG-4, H.264 AVC and Microsoft VC-1. The core offers scalable performance dependent on the features of the chosen algorithm and the number and type of execution units implemented. The ability to program the search algorithm to be used, and to reconfigure the underlying hardware that it will execute on, combines to give an extremely flexible motion estimation processing platform. The processor is fully described using VHDL coding and the baseline configuration, consisting of a single 64-bit integer pipeline, is capable of processing HD video at 30 frames per second.

Power aware computing has become one of the main trends in computer science, strongly due to power limitations of mobile devices, and especially those aimed at the consumer market. Since the power consumption of the motion estimation processor represents such a large portion of the energy necessary for video coding, a special interest has arisen in trying to reduce the energy consumption of the core, while maintaining reliable operation and functionality.

Several approaches to achieve this goal are possible given the nature of the processor VHDL description combined with the inherent features of an FPGA-based implementation, and the flexibility of the core to run different motion estimation algorithms.

Architectural changes to the processor are possible through changes in the VHDL coding that describes its behavior. This in turn, allows for power reduction techniques, traditionally reserved for ASIC's to be applied to the FPGA-based implementation of the processor. Additionally, the implementation of a large and complex design like the LiquidMotion processor into an FPGA provides an opportunity to use the HDL synthesis and implementation tools available to affect the processor speed performance and power footprint, depending on the configuration of the tools. Finally, the processor capability of running different motion estimation algorithms and the availability of a processor specific toolset to create the source code necessary to execute them on the processor makes it possible to evaluate the

performance of different algorithms in terms of power consumption. All of these features are fully available to the end-user and represent a low-cost and accessible option to achieve the proposed goal.

The work presented here aims to explore these power reduction options, evaluate their effectiveness at meeting the goal and show a qualitative analysis of their advantages and drawbacks.

Aims

The project main aim is to reduce the power consumption of the motion estimation processor developed by the University of Bristol Microelectronics Group by means of using the tools and resources available to the end-user. As such, three main options to achieve this are explored in this work:

- The use of synthesis and implementation tools to modify the resources allocated for the implementation of the processor.
- The impact of using different motion estimation algorithms, on the processor average power consumption
- Modifications in the processor description, at the design and system level.

Objectives

- Literature review: A comprehensive study of prominent power saving techniques, commonly used for ASIC's, that can also be used for FPGA's, was made. The fundamental principles and mechanisms of these techniques were reviewed, with special focus to the details regarding the implementation for the case of FPGA's. Proposed ideas to implement each one of these techniques into the processor were drawn, followed by a critical analysis.
- Review of the processor structure and motion estimation algorithms: study of the documentation and literature regarding the structure and operation of the processor, as well as a brief overview about block-matching ME algorithms
- Estimate the power consumption of the original core: using post-place & route timing simulations and the Xilinx XPower Analyzer tool, an accurate estimation of the processor baseline configuration was obtained in order to use it as the gold standard for comparisons. The targeted platform to implement the design is a Virtex-5 FPGA and as such, the toolset used for implementation was the Xilinx ISE.
- Evaluation of the impact of synthesis and implementation tools: During the synthesis and implementation phase of the processor design, there are several options and constraints available that guide the toolset to produce the design implementation, and these directly affect the performance of the design in terms of area, speed, and power consumption. As part of an effort to explore optimizations available on newer FPGA's, the processor design was implemented on an alternative platform, an Spartan 6 FPGA. The processor was then implemented under different configurations and each one was evaluated as stated before, plus metrics regarding the utilization and allocation of the FPGA resources

- Evaluation of different ME algorithms in terms of power consumption: Using the processor toolkit (Sharpeye Studio), seven ME algorithms were described, compiled into processor instructions and then the memory netlists were regenerated to reflect these changes. The power consumption of each algorithm was then estimated using the methodology already established above.
- Modification to the processor description to create a shallow pipeline and evaluation: According to the critical analysis of the literature review, modifications to the original processor pipeline were carried out to reduce the number of registers inside the pipeline. Power estimation of the modified processor was then carried to establish the power reduction achieved

1.1.1 Pipelining technique to reduce glitches

Dynamic power consumption is defined as the power consumed by the circuit during the transition from one logic sate to another. This power is wasted in charging and discharging intrinsic and load capacitances of various elements in the circuit and also in the short circuit paths that appear briefly as the result of complementary logic transistors switching on and off at the same time[6]. As investigated in [10] glitches can account for an average of 26% of the dynamic power consumption of an FPGA, with figures that vary from 4% to 73%, depending on the data and control flow of the design [8]. Glitches occur because of inputs of logic gates arriving at different times during the same clock cycle. This causes the gates to temporarily switch to an undesired state. This behavior however, should settle before the next clock cycle in order for the circuit to operate correctly. One can quickly see that, for the very same reason they are appear in the first place, glitches can propagate further into several logic levels of a design, thus causing unnecessary transitions of the logic circuitry of a design and wasting power. In Figure 1.A an example of a spurious transition in an XOR gate feeding a sequential circuit is shown. Although the unwanted transition settles before the next rising clock edge, thus making it harmless for data flow, the glitch wastes power both, for the 1 to 0 and 0 to 1 transition.

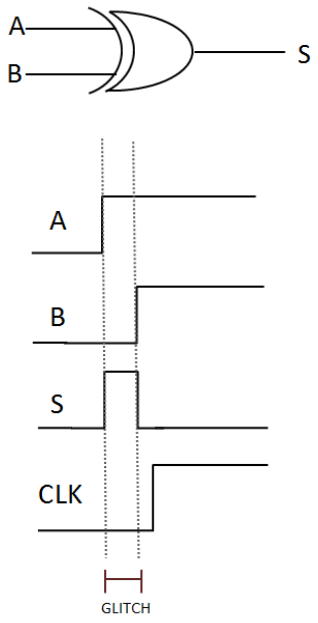

FIGURE 1.A SPURIOUS LOGIC STATE TRANSITION

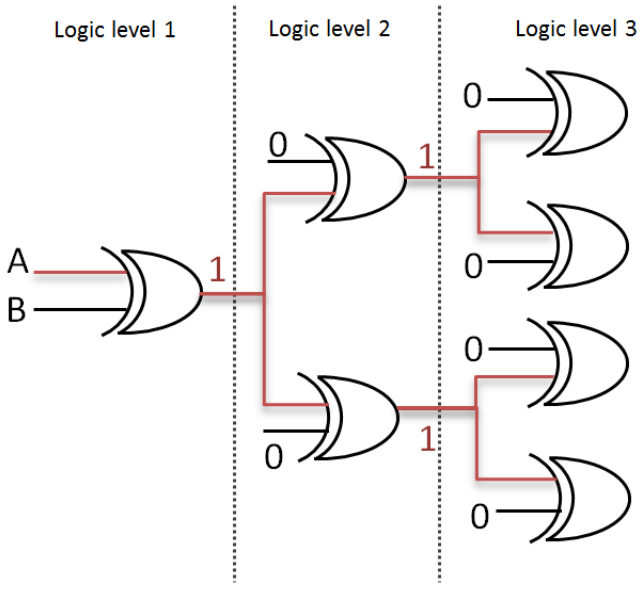

FIGURE 1.B GLITCH PROPAGATION THROUGH SEVERAL LOGIC LEVELS

In Figure 1.B the effects of glitch propagation are clearly shown in this XOR gate array. Assuming the combinational circuits are fast enough to switch several times before a clock period has completed, a glitch is able to propagate freely through many stages, with the

potential hazard of propagating a single glitch into several circuits, then greatly increasing the wasted power caused by a single glitch.

The concept of saving power by minimizing glitches in a circuit is not new and has been investigated in several papers already, both for ASIC's and FPGA's. Even though pipelining seems to be one of the easiest and most straight-forward options when dealing with glitch reductions, several other techniques have been proposed like the ones presented in [10] and [8]. In [8] a circuit level technique is presented: it that adds programmable delay elements to the circuit in an effort to synchronize the arrival times of various signals, thus effectively eliminating the source of glitches, while in [10] a synthesis technique that relies on logic function optimizations based on "don't -cares" to reduce glitches.

By pipelining a circuit one can decrease the occurrence of glitches. This is because between each pipeline stage a fewer number of logic levels are present and in turn the amount of levels that a glitch can propagate through is reduced [6],[7].This applies to both ASIC's and FPGA's but holds especially true for FPGA's since unwanted switching of logic elements also causes unnecessary switching of the routing logic [7]. In Figure 1.C it's shown how pipelining prevents glitch propagation by limiting the unwanted transition to the closest pipeline register, shown in blue.

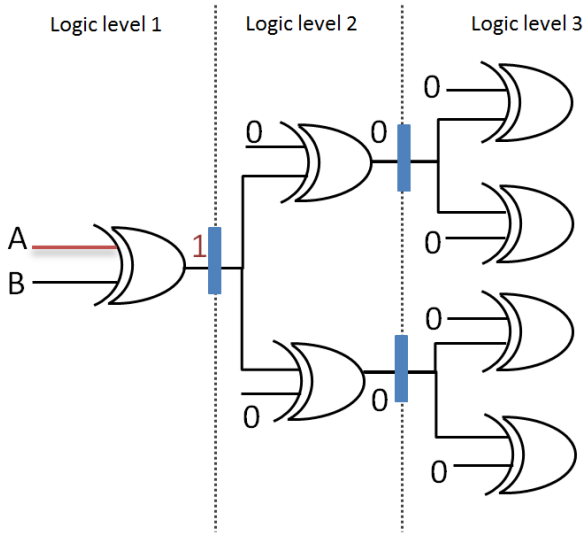

FIGURE 1.C PIPELINING OF LOGIC STAGES TO AVOID GLITCH PROPAGATION

As shown in [6] glitch related power consumption is an important concern when working with FPGA-based implementations. The inherent reconfigurable logic of FPGA's, like the Look Up Tables (LUT's), which have multiple inputs and whose output may be connected to several LUT's inputs further down in the data flow may trigger an avalanche-like effect when a glitch propagates through a very large chain of logic elements. Pipelining in this case limits the power consumption that each glitch can trigger. This is of special importance when dealing with logic blocks that implement arithmetic functions that process a large number of bits, like adders and multiply-accumulators.

Results derived from [6] conclude that, the more logic levels a design has, the more dynamic power consumption rises, and also that power savings can be achieved by simply adjusting the

way the internal logic of the design is connected together in order to, in turn, reduce the logic levels the design uses. This might be seen as an easy way to achieve power savings but one needs to take into consideration that alteration of the data and control flow of a design might not be possible on every design, and if possible, this may require significant time spent on by-hand optimization. By using this approach power savings of up to 34% are shown for a simple parity checking design in [6]

In [6] and [7] the effect of pipelining various designs with increasing numbers of stages is investigated. The results from both show incremental power savings for increasing pipeline stages introduced in the design. However it's also worth noting that there a limit on how much power savings can be obtained from pipelining a design beyond a certain point. It appears that beyond a certain number of pipeline stages the increment in savings is marginal at best and sometimes it's actually worse than the designs with fewer numbers of stages. The benefits of pipelining are also clear though, dynamic power savings range from 31% to 98% over the non-pipelined version, depending on the design and the number of pipeline stages used. Also, usually the area of the design doesn't increase, or increases only slightly, this is due to the added flip-flops needed to operate as the registers for each pipeline stage. The small area and resources overhead makes pipelining a remarkable option when area constraints are to be met or when resources, like the number of slices are limited, which would be the case for embedded platforms like the motion estimation processor.

1.1.2 Adaptive Pipelining

As it was stated before, pipelining of a design is a straightforward approach to achieve important dynamic power consumption savings. Traditionally pipelining has been used in scalar and superscalar processors to increase instruction throughput. However pipelining also introduces hazards and drawbacks such as bubbles due to misprediction penalties. The cost of a misprediction increases according to the depth of the pipeline, and so does the power wasted [3]. Adaptive pipeline is a technique by which the pipeline within a processor is shortened or extended depending on a given situation, like availability of resources.

Adaptive pipelining can also be used to save power resources when the processor is idle or when power is of higher concern than performance. This technique is sometimes complementary and an alternative to dynamic voltage scaling, as with the later, there is a voltage limit in which DVS can operate [4].

As explained in [5], [4] and [3] the mechanism to implement the adaptive pipeline consists in disabling the latches that compose the pipeline registers of some stages of the original, deep pipeline, thus effectively merging some stages and creating a shallow pipeline. Merging the pipeline stages together using this method allows for quick switching between the deep and shallow mode. This asynchronous method also allows for greater flexibility as any number of stages can be joined together just by bypassing (or making "transparent", also known as collapsing, this only means that the input data is set in the output lines as if it was simply a wire) the latches between those stages. In order for this approach to work, reconfigurable latch controllers have to be implemented in the pipeline design [3]. Despite the asynchronous nature of the "collapse" input of the latches, the merging of stages cannot just take place at

any time. The pipeline has to make the transition between full-depth and shallow just before new data arrives into the top stage and data in the lower stage has already been processed. This assures data integrity and allows for the stages to merge in time to process the new data being loaded. In a similar fashion, the mechanism for splitting stages is set when new data is loaded into the merged stage, the “collapse enable” signal is deasserted, making the latches operate again normally. because of this merging and splitting operation has to take place when new data is to be loaded into the latches, the “collapse” signal is latched with the rest of the data signals coming into the pipeline stage, then providing a signal that flows downstream into the pipeline and controls the “collapse” signal of the latches, while at the same time it’s synchronized with the data flow, allowing to preserve data integrity. This approach works if all latches are to be merged together, or in fixed blocks (as the collapse signal is the same for every set of latches) but if selective control is required to merge only certain blocks of stages then more than one “collapse” signal must be bundled into the downstream flow[3],[4].

The working mechanism to switch between deep and shallow mode is shown in Figure 1.D. In this example, stages are merged in pairs when switching to shallow mode, and the collapse enable signal is bundled into the dataflow stream. The signal is then synchronized with the data flow, ensuring that previous data has already processed by stage 2 and then allowing for the dotted register, between stage 1 and 2, to collapse.

Even though adaptive pipelining has been explored in a number of applications, few papers are dedicated to investigate the power savings related to this technique and nearly all of them are dedicated to its application in ASIC’s . The results of [4] show power savings that range from 23% to 40% in a 8-way superscalar processor, running different integer benchmarks. It is further explained in [4] that the power savings are mostly due to the reduced branch misprediction penalty and longer execution latency for dependent data in deep mode. In the case that was investigated, the shallow mode pipeline was half the size of the deep mode one.

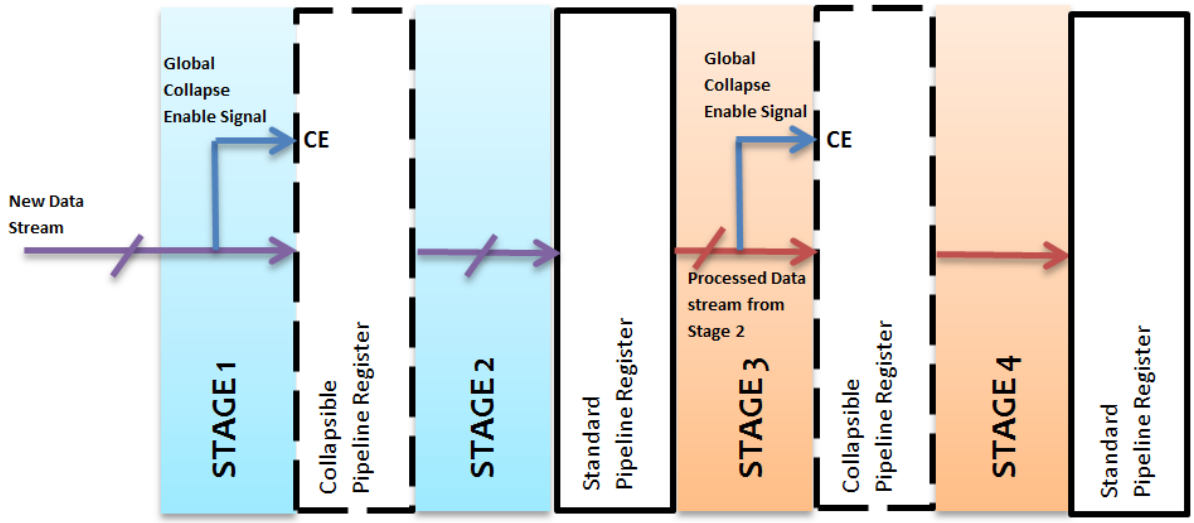

FIGURE 1.D SWITCH PROCESS INTO SHALLOW MODE FOR AND ADAPTIVE PIPELINE

However when the stages are merged, and as stated in [5], glitch propagation can happen through a larger number of logic levels, increasing dynamic power consumption to some extent. It's safe to conclude that there is a tradeoff between decreased power savings when merging pipeline stages due to increased possibilities of glitch propagation and increased savings because of a shallower pipeline that wastes less power due to misprediction penalties. Code benefits from the adaptive pipeline especially when it contains many branches [5], and it could be added further, when the branch prediction engine performance is poor, or when no branch prediction engine is present due to complexity and power issues.

1.2.1 Clock gating

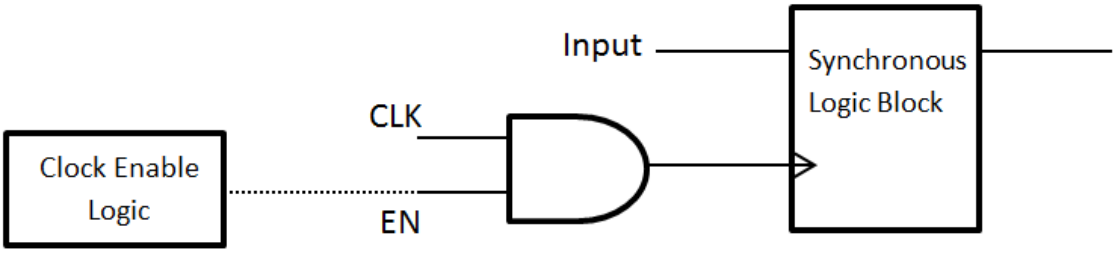

FIGURE 1.E SIMPLE CLOCK GATING IMPLEMENTATION

Clock gating is one well know technique that has been in use for more than a decade, as a power reduction technique for ASIC's. This technique is a method to reduce signal transitions in a part or a logic block inside a circuit, when the outputs of said part are not needed (or irrelevant) to the final output of the whole circuit. This is done by clocking removing the clock signal from said parts of the circuit, thus eliminating all unnecessary logic transitions in that block and in turn reducing any power wasted in those transitions [2], [9]. By gating the clock signal along with an enable input, the clock y selectively activated in a certain region of the circuit. In Figure 1.E the simplest approach to the clock gating technique is shown, where the clock is fed along with an enable signal into an AND gate to selectively activate or deactivate a whole logic block. The output of said block will only change when the clock signal is let through.

Presented in [2] is a comparative evaluation of the performance of the clock gating technique implemented in both ASIC and FPGA designs. Results show that inherently, when clock gating is implemented in FPGA's the power savings are less than those obtained when using ASIC's. In [2] it is elaborated that this reduction is due to the clock propagating through the clock trees in the FPGA's whereas in ASIC's the whole clock tree can be switched off to increase power savings even more. However the approach used in said work to implement the clock gating in FPGA's is significantly different than the one used in more recent papers like the one found in [9], as the former uses a feedback multiplexer to emulate the mechanism by which clock gating is implemented in ASIC's, and doesn't take into account recent FPGA's which include clock gating capabilities integrated on chip. Nevertheless it makes a point in observing that

because of the inherent differences between ASIC's and FPGA's, customization of the clock trees is possible to a lesser extent in FPGA's

1.2.2 Clock tree in FPGA's

The clock signal in FPGA's is distributed through clock trees that effectively feed the clock signal from one place to the rest of the circuit. For example a Xilinx Virtex-5 FPGA is divided into several regions, and such regions can be fed by a limited number of clock signals, depending on the structure of the clock tree. The tree is made up by rails or "spines" (vertical) that carry the clock signal through the chip and into the required region by branching into other (horizontal) spines. Routing switches are available at the intersections of the spines in order to pass the clock signal only to those regions containing logic that needs the signal, otherwise if the signal is let through without being needed by any logic, this would lead to a continuous waste of power in the chip [9].

It's clear that depending on the complexity and availability of resources like switches and branches, it's possible to divide each region further into smaller sub-regions, then allowing for more selective clock gating. It also holds true that clock gating, like many other power saving techniques, benefits the more when processing large arrays of bits, since normally one clock signal can drive several inputs, and each of those contributes some capacitance to the line. Also the length of the wire transporting the clock signal going all the way to the intended region adds even more capacitance to the line. Then, avoiding unnecessary switching on heavily loaded lines increases the power savings by using clock gating [9] Timing analysis is used in modern designs to optimize the clock synthesis. Also more recent FPGA's like the Xilinx Virtex-6 have clock gating capabilities integrated, as in the case of this particular FPGA, it is possible to enable or disable the clock signal for individual regions of the chip. According to Xilinx claims, clock gating can account for dynamic power savings of up to 80% [14]. It's also worth noting that clock gating in this particular FPGA, can also be applied to dedicated blocks like DSP slices and multipliers. This is a clear advantage when implementing designs that make heavy use of these elements like digital filters and, image and video processing applications. Shown in Figure 1.F is the diagram for a fine-grained clock distribution tree as can be found in modern FPGA's. Each of the primary spines (marked in red) is able to carry more than one clock signal. Eventually these clock signals are fed selectively into the lower branches (orange) by means of interconnection logic found on the intersections of the spines. This scheme allows to keep several sub-regions working at different clock speeds, and makes it easier to implement clock gating techniques.

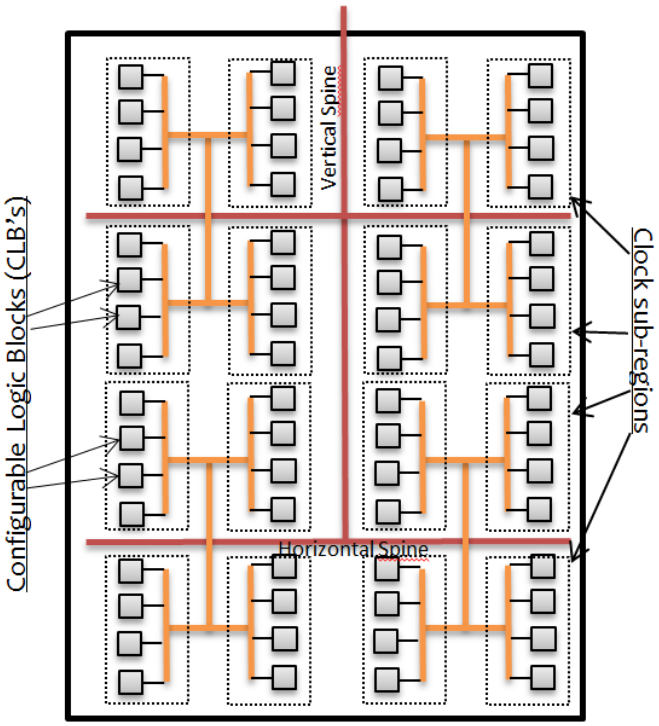

FIGURE 1.F FINE-GRAINED CLOCK TREE DISTRIBUTION ON A MODERN FPGA

From what has already been explained and from the work shown in [9] it's expected that the power savings achieved are directly related to the clock tree layout and the way it is fed into each region. The work in [9] relates specifically to this point, where the power savings are investigated according to the granularity with which the clock signal can be delivered to the regions and sub-regions, and also how is the clock delivered within those (this refers to whether vertical or horizontal spines are being used to fed the clock into the region, as each one provides different paths and different selectivity).

The results from [9] show that, in general, when the clock gating is more selective substantial power saving versus a non-clock gated design can be achieved. Also, two levels of implementation of the technique are investigated, one conservative and another one where it is implemented more substantially. According to this, the savings are about 12% when a limited amount of clock gating is used and more than 50% when the substantial gating is used. Noteworthy is an unexpected result in the finest range of granularity possible for the substantial implementation level. This situation is explained by the fact that deeper and larger clock trees add more capacitance to the line, as explained in the above paragraphs. It's possible to say then that a balance, or optimal point can be achieved, between more selectivity when clock gating logic blocks and added capacitance (which means more wasted power) because of using this longer clock trees.

It is estimated that clock power accounts for roughly 20% of the total power consumption within an FPGA [9]. Also, clock gating appears to yield the highest power savings when implemented in circuits in which the clock activity is low, meaning that some regions in the circuit can be held idle for longer times, which translates in larger power savings. This result means that the total average power savings due to clock gating are in the range of 6.2% to 7.7% for designs with low clock activity and marginal to negative savings when the clock

activity is high. Regarding implementation costs [9] concludes that actually there is a small performance penalty for any design implementing this technique on an FPGA. The penalty is about 0%-2% of the overall performance and it is due to the conflict of the synthesis tool when optimizing the placing of logic cells or slices with two goal targets, one of them would be reducing the clock power and the other one reduce path delay. Looking at these results it can be concluded that if power is a concern, like in embedded designs clock gating is a feasible option to obtain a small, but significant amount of power savings at the cost of increased logic complexity to implement the clock gating and a small performance penalty.

Presented in [1] is a combination of the pipelining and clock gating techniques to reduce power in FPGA-based implementations. Pipelining is implemented first and then the clock gating technique is applied. Results show that by doing this an extra 2% of average dynamic power saving can be achieved after pipelining. The work also points at the power overhead that comes when additional clock trees are used in an FPGA. It states that this overhead is due to additional clock management logic used for the additional trees and also because of the added tree capacitance. Drawing from these results it can be inferred that another optimization point for clock gating is found here, when balancing the increased power consumption due to both these effects and the potential savings of further clock gating a design. This means that there are power tradeoffs even when using a modern FPGA with many clock gating resources available due to the inherent programmable logic and routing.

One drawback from using this technique might be the need to modify the design to include the implementation of the clock gating control logic [2]. However, new tools are being released that automatically analyze the design and clock gate the design where possible. Xilinx has recently announced that his set of design tools are capable of automatically integrate clock gating when synthetizing designs, allowing for reductions in dynamic power of up to 30% [11].

1.3.1 Dynamic Voltage Scaling

Dynamic voltage scaling is a well-known technique used in ASIC's with the purpose of reducing power consumption as shown in works like [18] and [19]. DVS consists in dynamically adjusting the supply voltage of a circuit [21]. As dynamic power consumption is a quadratic function of voltage, significant power savings can be achieved by means of this technique. To achieve the maximum decrease in power consumption the circuit has to operate at the minimum voltage possible and at the same time guaranteeing proper circuit operation [21].

Both [18] and [19] show implementations of the technique in ASIC's and their results show a maximum of 50% power and 82% savings respectively, both using a 0.18µm fabrication technology. In both works DVS is implemented through a mechanism that acts either as feedback from the critical delay path, or as a delay synthesizer that emulates this same path. The voltage is then continuously scaled down until information from this mechanism informs the control logic that either a delay error is about to happen or it has just happened.

Delay errors are the result of operating the transistors at lower voltages. As the operation voltage decreases, switching in these elements becomes increasingly slower. And eventually the circuit is unable to meet the critical delay path because the added delay that the slower

switching causes. Ensuring that the circuit meets this timing constraint ensures proper operation of the circuit [21]

Since DVS provides significant power reductions in ASIC's, some research has already been conducted towards its implementation into FPGA's with papers showing successful results as in the likes of [20] and [21] which use similar approaches to address the critical delay path sensing.

As a matter of fact, DVS has a large advantage over other power saving techniques when implemented in FPGA's. This is because DVS reduces not only dynamic power consumption but static power consumption as well. This is a major motivation for implementing DVS into FPGA-based designs since, as stated in [2] static power consumption is inherently much larger for FPGA's than it is for ASIC's. This is because of the power that the programmable logic inside the chip, like LUT's (Look-up Tables), continuously consumes.

Another important characteristic of DVS is that, most of the time not only the voltage is adjusted, but alongside it, the frequency of the clock feeding the circuit is also adjusted [18], [20]. This approach allows to reduce power even more by dynamically adjusting the clock frequency if the voltage is lowered to a point where it causes delay errors. This trades increased latency in favor of reduced energy consumption [20]. Another advantage of DVS is that this technique is applied at system level. This means that no logic or area overhead comes from implementing it and also means that it can be co-implemented with other techniques that operate at design level, like pipelining and clock gating [21].

In [21], details of DVS implementation on a FPGA-based design are shown and the performance of this technique in terms of power savings is investigated. The approach used for [21] is a DVS mechanism that controls only the voltage. Several key points that must be observed in the case of FPGA's are highlighted in this work:

- The balance point between, lowest possible operating voltage and ensuring correct circuit operation, is not static. This is because path delay is also a function of die temperature and process variations. As such, two FPGA's of the same model, implementing the same design, might not be able to operate at the same voltage; and at any given time voltage in a circuit may have to be readjusted to a higher level to meet critical path delay (or if the DVS implementation allows for it, reduce the clock frequency) due to increasing temperature as a result of circuit operation. This last point can be further extended by stating that heavy loading of the output pins of a chip may lead to decreased performance of DVS, due to the chip having to source significant amounts of current, which in turn increases heat produced by the chip itself.

- Voltage supply of internal circuitry has to be separated from I/O supply. Only the internal circuitry can benefit from DVS, because I/O voltages have to be kept steady to standard logic levels in order to avoid compatibility issues when connecting outside logic to the FPGA. Thi, however, is not a problem, especially in modern FPGA's where the supply lines of the core and I/O blocks are separated.

- Aside from the delay errors that may occur as a consequence of lowering the voltage of the core, another type of errors known as “IO errors” can happen when the core voltage is so low that, even a high signal going from the core logic to the I/O blocks is recognized by the later as a logic low, making the output of the chip to be incorrect. As with delay errors, the critical path sensing mechanism has to implement a way to flag that this errors are about to occur or already have occurred in order for the voltage controller to be able to raise the core voltage again to acceptable levels .

The work in [21] uses a Logic Delay Measurement Circuit (LDMC) as means to track the critical delay path of the circuit and a voltage controller to reduce the voltage supply. A feedback signal going from the LDMC to the voltage controller allows to readjust the voltage when errors are about to occur in the circuit. The LDMC main component is a chain of inverters (128 inverters in total), with the output of each inverter as the input to a positive edge triggered flip-flop. The first inverter in the chain is driven by the clock signal of the circuit. The clock signal is then able to propagate through the chain and, on the rising edge of the clock, the flip-flops that sit at the output of each inverter, latch the values in the chain. At that moment, the clock signal will have propagated up to a certain point in the chain. As stated before, the speed at which the clock signal propagates through the chain is proportional to the current die temperature and the circuit operating voltage. The value read at the chain is then used to calculate the number of stages that the clock signal was able to propagate through. If the delay is above a predefined value, then the voltage is allowed to be decreased until a safety threshold is reached, at which point, the voltage decreasing is no longer allowed. If at any time the delay measured in the chain goes below this safety threshold the voltage is raised again. This compensates the effects that increased delays due to higher die temperatures may have on the circuit. Also pointed correctly in [21] is the fact that a large rate of change should be used to quickly achieve the lowest possible operating voltage. It’s safe to say that doing this would allow to increase the power savings reported by the DVS technique, and also to effectively compensate quick variations in die temperature (and also due to frequency variations, if the if the implementation mechanism allows for those) before they can cause errors. Shown in Figure 1.G is a simplified diagram showing the LDMC used in [21]

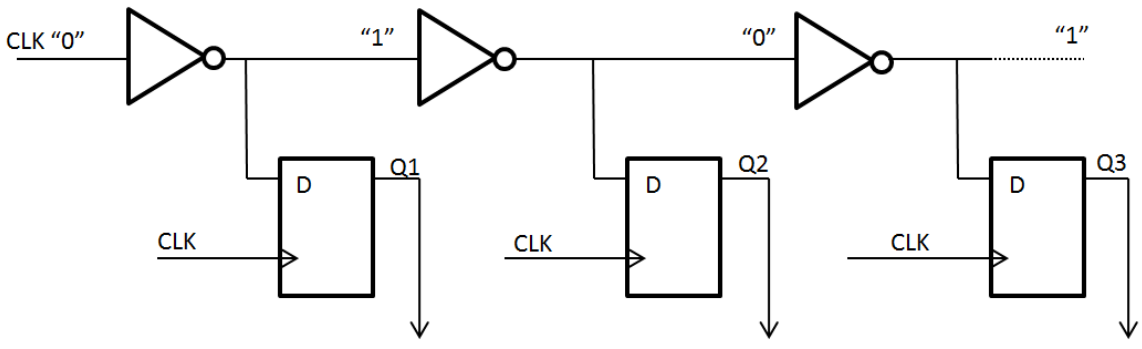

FIGURE 1.G SIMPLIFIED DIAGRAM OF A LOGIC DELAY MEASUREMENT CIRCUIT

The mechanism used in this work, while simple and efficient, has the important drawback that all threshold points have to be set up, and optimized manually. While this might seem like a safe approach it doesn’t account for process variations which implies that, for example, if a

design was to be implemented across several FPGA's, some of the chips would actually be running at a higher voltage than they could because the threshold points are static and the same for all chips. Another drawback is that since the frequency is kept constant, no tradeoff is allowed between design performance and power consumption, then neglecting important savings for non-critical applications.

The work presented in [20] addresses this issue by also implementing a mechanism to scale frequency where possible. A very similar approach to measure the critical delay path is used. In this work however, a second flip-flop in every node of the chain is added. This flip flop is actually connected to the output of the first flip-flop, with the purpose of reducing metaestabilty issues that arise from violating the setup time of the first flop. As part of a verification block a bit mask is set for this DVS implementation, in which the delay mask specifies the number of bits which the block has to check for correct transitions, before flagging a delay error (meaning that the chain pattern should always read alternating 0's and 1's for as many stages as specified in the bit mask).

This implementation of DVS is made using a Leon3 System-on-Chip implemented in a Virtex-4 FPGA. The Leon 3 soft processor implements itself a 7-stage integer pipeline, and for this case, it has been extended with a DVS control block and a Digital Clock Management unit connected as peripherals. The fact that the SoC itself implements a pipeline for data processing ads up to the complexity of the implementation, because it takes into account that data flow errors inside the processor pipeline present an important hazard. The DCM provides a reliable tool for frequency synthesizing and works together with the DVS block to dynamically find and optimal voltage and frequency point [20].

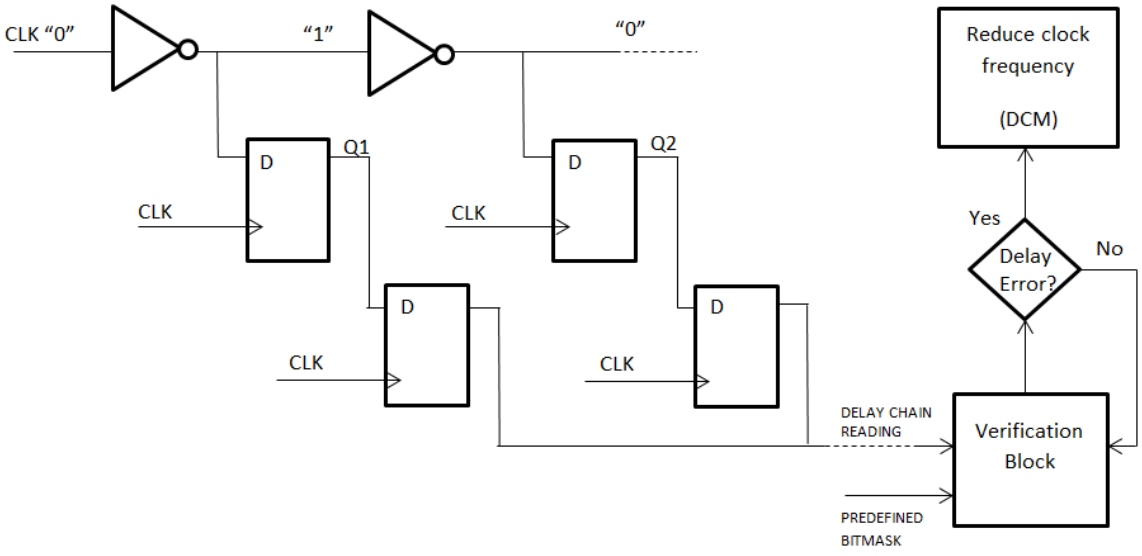

FIGURE 1.H SIMPLIFIED DIAGRAM THE FREQUENCY REDUCTION MECHANISM IN A DVS IMPLEMENTATION

The DVS working mechanism starts by reducing the core voltage according to processor activity. To avoid errors in the pipeline, the mechanism freezes the pipeline state before both the DVS and DCM blocks start to search for a new operating point. The algorithm used to balance voltage vs. frequency is not explained in full detail, but it is shown that the mechanism

is able to lower the frequency as a safety measure when the delay path line detects that the current operating point fails to meet the critical delay constraint. The supply voltage is allowed to vary between 0.9 and 1.2 volts. Shown in Figure 1.F is a simplified diagram of the delay path sensing mechanism and frecuency reduction criteria used in [20]. When delay errors are detected above the threshold established by the predefined bit mask, the clock signal frequency is reduced to prevent them.

Both [20] and [21] arrive at results that show an important decrease in power consumption when implementing DVS on FPGA-based designs. In [21] power savings for various designs are investigated and it was found that the average power savings are about 20%-30%, but minimum and maximum savings range from 4% to 54%. It is argued that potential savings when implementing DVS may have a relationship with the design being targeted, as different designs show different savings. Although, no conclusive explanation is given, logic delay and routing delay sensitivity to voltage variations is suspected as the main cause for this behavior.

In [20] notable power savings in static power are seen when decreasing voltage. At 0.9 volts of core voltage the static power dissipation was 31% of the total power and for 1.2 volts 42%. The work concludes that a more complex DVS mechanism implementing frequency variation allows for increased energy savings when performance can be traded in exchange for larger power savings.

Regarding the previous idea, it is suggested in [21] the idea that, it might be possible to compensate for the performance penalty resulting from reducing the clock frequency by adding additional execution units working in parallel. However the experiments conducted in [20] suggest that a critical point exists in which lowering the frequency of the circuit to avoid delay errors continues to reduce power dissipation but actually increase energy consumption. This is explained by the fact that latency increases when the operating frequency is decreased, in which case, despite the decrease in instant power dissipation, the design takes longer to perform the same task and the total energy consumed to complete the task is actually larger. This means that adding additional execution units may actually increase power consumption. Further investigation into both of these approaches is yet to be done to show any conclusive results.

The experimental results shown in [20] come from the fact that the delay mask was hand optimized for one case and compared against a mask generated with data from a timing analysis tool, showing again the need to find a mechanism to dynamically adjust the DVS thresholds, like in [21]. To this effect, it can be concluded that some by-hand optimization effort is required when implementing DVS using a delay chain on an FPGA, in order to maximize the power reduction.

The major advantages of implementing the DVS technique on FPGA based designs are that, the design itself doesn't need any modifications [21], as the critical delay path sensing mechanism can be implemented independently. While this is true on principle, in [20] it's shown that some level of integration may be required depending on the implementation of the mechanism to avoid data flow errors. Even so, the level of integration that might be needed, if any, is fairly low and it appears that doing so may present little effort to the designer [21]. This makes the

DVS technique a strong option when targeting to reduce power consumption of a design that is already fully developed [21].

1.3.2 Critic analysis of the power reduction techniques

Several conclusions may be drawn from the information presented in this review. First of all, the various power reduction techniques reviewed offer their own advantages and drawbacks and at the same time each one offers potential for future enhancements on their own. While these techniques have been widely used for ASIC's, the information regarding their implementation on FPGA's pales in comparison to the amount available for the former. Despite this, several successful FPGA implementations of these techniques have been investigated and proven to be considerably effective to achieve its goal. Also, judging by the content and, especially, the conclusions of many papers regarding this area, it's safe to say that further work needs to be done in order to optimize these techniques to its fullest extent.

It's also clear that while the basic concept of the implementation remains the same for both technologies, further considerations, regarding the intrinsic functionality and architecture of FPGA's, have to be taken into account.

Specific conclusions about each FPGA power reduction techniques are summarized below. The conclusions are directly biased towards the potential implementation on the Liquid Motion ME processor.

The pipelining technique offers notable dynamic power savings while at a relatively low implementation cost. The level of implementation though, increases according to design complexity and the number of logic levels a design contains, since more registers have to be created, then, more logic resources (CLB's) are needed to implement the technique. The design effort required is potentially low since the design data flow of the design is not altered and only extra logic is added between each logic stage. The adaptive pipelining variation of this technique offers further power savings to a design that is already pipelined. As shown in the review of this technique, if a pipelined design is constantly running into branch mispredictions or other issues that may force it to keep the pipeline only partially full (or stalled), then it can largely benefit from going into a shallow pipeline mode. This can reduce the misprediction penalty, then reducing the time and power wasted to fill the pipeline again. Even more, it could be possible for the shallow mode pipeline to actually, free up some resources used by the pipeline itself, decreasing power consumption even more and allowing for reallocation of those resources.

The clock gating technique offers a less than moderate amount of savings when implemented on FPGA's, as shown in the papers reviewed in this work. Even though modern FPGA's (and also vendor tools for design implementation) offer dedicated mechanisms to easily implement clock gating into a design, it would appear that the results obtained from it can vary greatly. It's safe to presume that designs containing a great amount of unused logic at any one time can benefit greatly from this technique, since those logic blocks are deactivated at said time, which directly translates into saved dynamic power. However, it's possible to argue that this is rarely the case for pipelined designs since a pipeline works, ideally, on the basis that each stage is processing data at the same time. To back up this statement it's worth noting that the

results from one of the cited papers show only a marginal decrease on power consumption when clock gating is implemented in an already pipelined design. Also the design effort that would be necessary to fully implement the technique on an existing design could be extensive, especially for a complex superscalar processor, because of the need to guarantee that clock gating doesn't interfere with the correct data flow of the design. Even though automated clock gating implementation tools may save some effort into this, it's very likely that a lot of by-hand modifications and tuning would be needed.

The DVS technique seems to be one to offer the largest power savings, since it also reduces static power consumption as well as dynamic power. It also has the advantage of being a system level technique, meaning that no direct modifications to the design would be needed. This doesn't mean though, that the implementation comes for free, since the implementation of the DVS control blocks, is already a challenge by itself. As shown in the review for this technique it's completely possible to integrate this technique into a pipelined processor while avoiding any potential data flow errors by integrating the feedback coming from the critical delay path sensor, into the design. A possible drawback to this technique is that, by-hand tuning of the critical delay path sensing mechanism is required in order for the DVS technique to get the largest possible amount of power savings. This presents a problem due to the highly reconfigurable nature of the processor which makes it impossible to carry the optimization process for each possible configuration without an integrated dynamic mechanism that takes care of it. Despite this, the implementation of DVS should yield significant power savings by setting the mechanism to an "average" value.

The implementation of any of the three techniques on the original processor will result in some level of decreased power consumption, however, according to this analysis, the adaptive pipelining and dynamic voltage scaling techniques appear to be the more suitable ones to be implemented, since both have a good cost-benefit relation, in terms of required effort, resources and complexity versus the potential power they could save.

Considering that the nature of the fast-search motion estimation algorithms is itself prone to create various conditional loops when executed on a processor, because of the iterative nature of the process itself and also, due to early termination and duplicated search-point avoidance conditions used to improve the speed of the process [15], the adaptive pipeline approach is thought to be able to achieve a considerable amount of power savings when implemented due to the reasons discussed previously. Additionally, the totally random nature of the data prevents the implementation of an effective branch prediction mechanism. This means that bubbles in the pipeline, created because of branch mispredictions or early terminated loops, are a common occurrence in the processor.

The review presented on this chapter has addressed the major implementation details that need to be taken into consideration to implement each one of the FPGA power reduction techniques into the processor description. Based on the review and the critical analysis, a decision was made to implement the adaptive pipelining technique. Specific implementation details of the technique are described in chapter X of this work.

CHAPTER 2: THE MOTION ESTIMATION PROCESSOR

2.1 Overview

The LiquidMotion processor is a fully synchronous, configurable and programmable Application Specific Instruction-Set Processor (APIS) for motion estimation, described using VHDL and targeted specifically at FPGA implementation. The processor is capable of processing HD video for various video codecs like H.264, MPEG-1 and MPEG-2. The processor is tuned to execute all fast block matching motion estimation algorithms. The flexibility of the core lies in its programmability that allows reconfiguration of its microarchitecture depending on the motion estimation algorithm selected, the codec, and the capability to choose the number and the type of execution units to be implemented [15],[22],[23],[24],[25].

Among the group of recently introduced video codecs like VS-1, AVS and H.264, the later has emerged as a widely favored option used to support HD video processing, as shown by the fact that was chosen as the preferred coding standard for Blu-Ray Discs and several high definition television broadcasting standards. H.264 achieves a considerable reduction in bitrates by relying on more complex mechanisms to perform decoding of the video frames and the inter-frame prediction, like integer 4x4 discrete cosine transform (DCT), deblocking filters, intra-prediction and more complex motion estimation [22],[25]. Standards like MPEG-2 used a simple motion estimation approach, while H.264 advances it by using variable block sizes, quarter pixel resolutions and multiple reference frames. This additions increase the complexity required for motion estimation process, which then requires up to 80% of the encoding time [22],[23]. Increasing complexity makes the use of a full-search algorithm an unacceptable choice when targeting area and resource limited platforms like embedded applications, because of the huge amount of processing resources it would take. This is because of the exhaustive approach the full search algorithm uses. A fast motion estimation algorithm like three-step search, diamond search and hexagon search among others, is best suited to reduce the amount of resources needed to implement motion estimation, as these algorithms reduce the complexity of inter-frame prediction [15],[23].

The LiquidMotion ME processor reconfigures and tunes the core microarchitecture according to the algorithm used and the mechanisms selected for motion estimation, then allowing for optimization between the complexity of the algorithm and the quality of the video while achieving a low resource footprint implementation. The processor configuration can be adapted, at compilation time, to meet power and area constraints based on the targeted implementation platform. Once these constraints have been met, an iterative optimization process can then be carried in order to meet the processor performance constraints and optimize the implementation according to the available hardware resources [15].

The processor toolset provides a compiler, assembler a cycle-accurate simulator and performance analysis tools. A simple programming model is used that serves as a high-level interface layer between the processor architecture and the programmer, then making possible to quickly build complex algorithms without the need to go into the processor architecture in-depth. A C-like language, called Estimo C is used for this purpose, an it's aimed specifically to

describe motion estimation algorithms with simple syntax and C control flow constructs (if, for, while, etc.) [15],[25]. The compiler is capable of extracting parallelism from the code to increase processor performance. Finally, by using the cycle-accurate simulator together with the and the analysis tools, it's possible to estimate the performance of a given processor configuration in terms of complexity (implemented LUT's), bitrate, frame throughput per second, peak signal to noise ratio (PSNR), and energy consumption. By doing this, several processor configurations can be compared to evaluate the various tradeoffs between overall performance and resources available [25].

2.2 The processor microarchitecture and top interface

The processor base configuration consists of a single 64 bit integer-pel, pipelined execution unit. Depending on the processor configuration more integer-pel execution units (IPEU's) , fractional-pel execution units (FPEU's) and interpolation execution units (IEU's) can be added [15],[22],[23].

Each of the IPEU's is processes 64 bit data and is deeply pipelined to achieve higher throughputs. 64 bit buses are used to access reference and macroblock memory, with the SAD also operating on 64 bit vectors in parallel. The memory is arranged in 64 bit words and accesses to it are unaligned since the macroblock being read can start at any position inside these words. Two 64 bit words are loaded from the reference memory and then the required 64 bits from those words are selected in the vector alignment unit. Each additional IPEU implemented is provided a copy of the point memory so that all IPEU's process 64 bit data in parallel. Each IPEU gets an incremented address to this point memory. By doing so, each unit can calculate the sum of absolute differences (SAD) vector for a different point for the same pattern [22],[23]. The processor microarchitecture when implementing a single IPEU only (base configuration) and its data flow is shown below in Figure 2.A

The engine is capable of processing motion estimation data for half-pel and quartel-pel values through the FPEU's and IEU. The IEU instantiations are limited to one but the number of FPEU's can be configured otherwise. The IEU interpolates the pixel area surrounding the macroblock of the winner integer motion vector. The interpolation process cycles three times to calculate the horizontal, vertical, and diagonal pixels respectively [15]. The half-pels are calculated using a six-tap filter. This process makes use of eight systolic 1-D interpolation processors, each with six processing units. With this approach each interpolator can produce a new half-pel sample per clock cycle with no idle cycles. Filling or emptying the pipeline takes hold of the ports of the reference memory ports, meaning that during this time the IPEU is stalled. Once interpolation is completed, the next macroblock is processed in the IPEU's [23]. Quarter-pel interpolation is processed on-demand by averaging the the half-pel and full-pel position data, thus requiring additional vector alignment units to process to load this data. Below, in Figure 2.B is a diagram isolating the FPEU microarchitecture and its data flow [15]. The COST Selector block at the bottom of the figure is used to optionally implement rate-distortion optimization to improve frame quality [23].

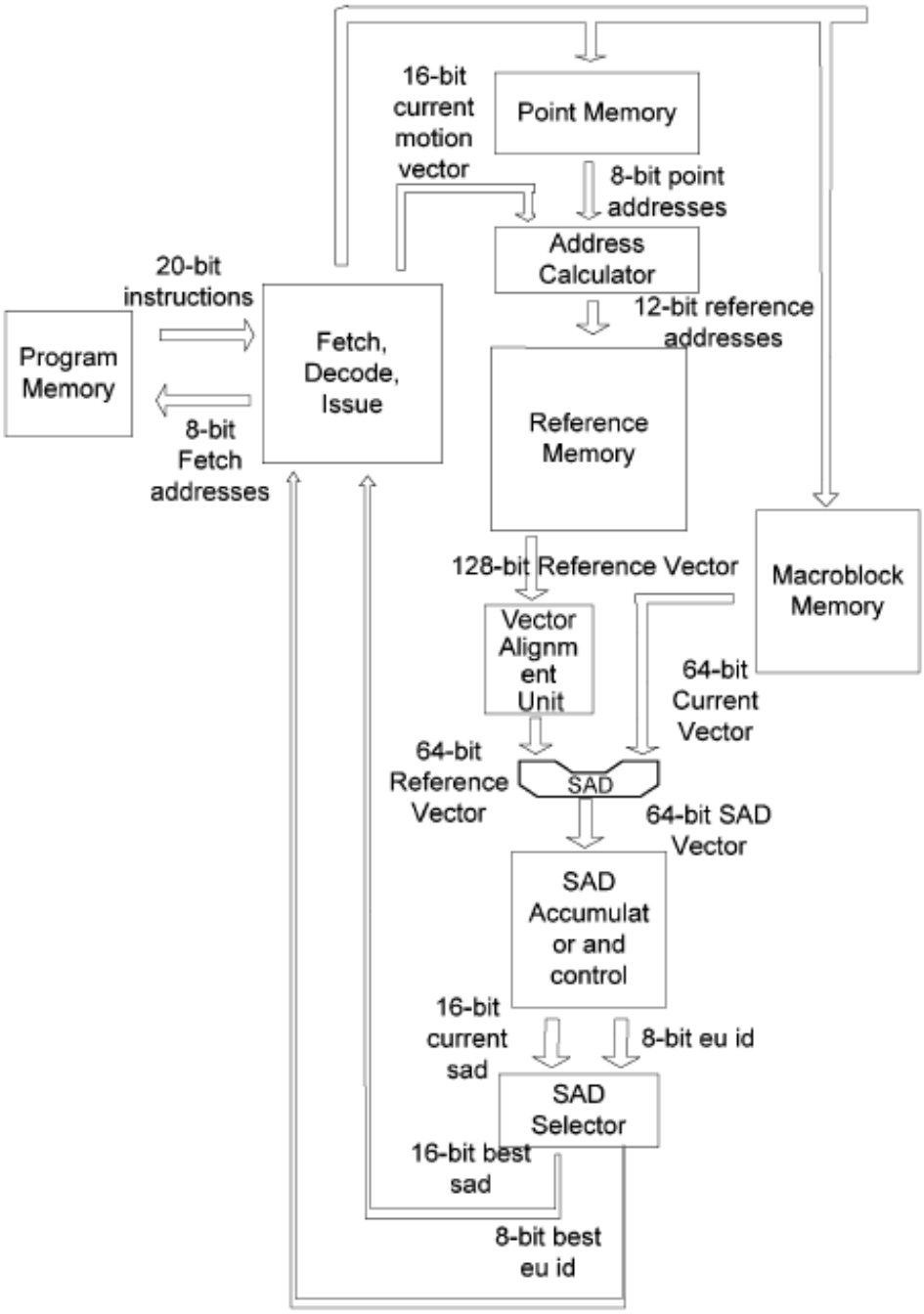

FIGURE 2.A PROCESSOR BASELINE MICROARCHITECTURE.
Taken from [15]

Reference memory is arranged in 128 bit rows but the horizontal search range is reduced to 112 bits in order to leave a 16 bit pixel area to load the column of next macroblock by using a sliding window technique. This allows loading of new data as the current one is being processed and at the same time prevents execution units from reading data that is being loaded. The memory is also interleaved in two double BRAM blocks. This guarantees that at least one memory port is free at all times, then preventing data flow bottlenecks [23],[15].

The core instruction set architecture (ISA) lacks any memory access instructions. A DMA engine sitting outside the processor core is in charge of feeding the frame and macroblock data to be processed in its internal memory [15].

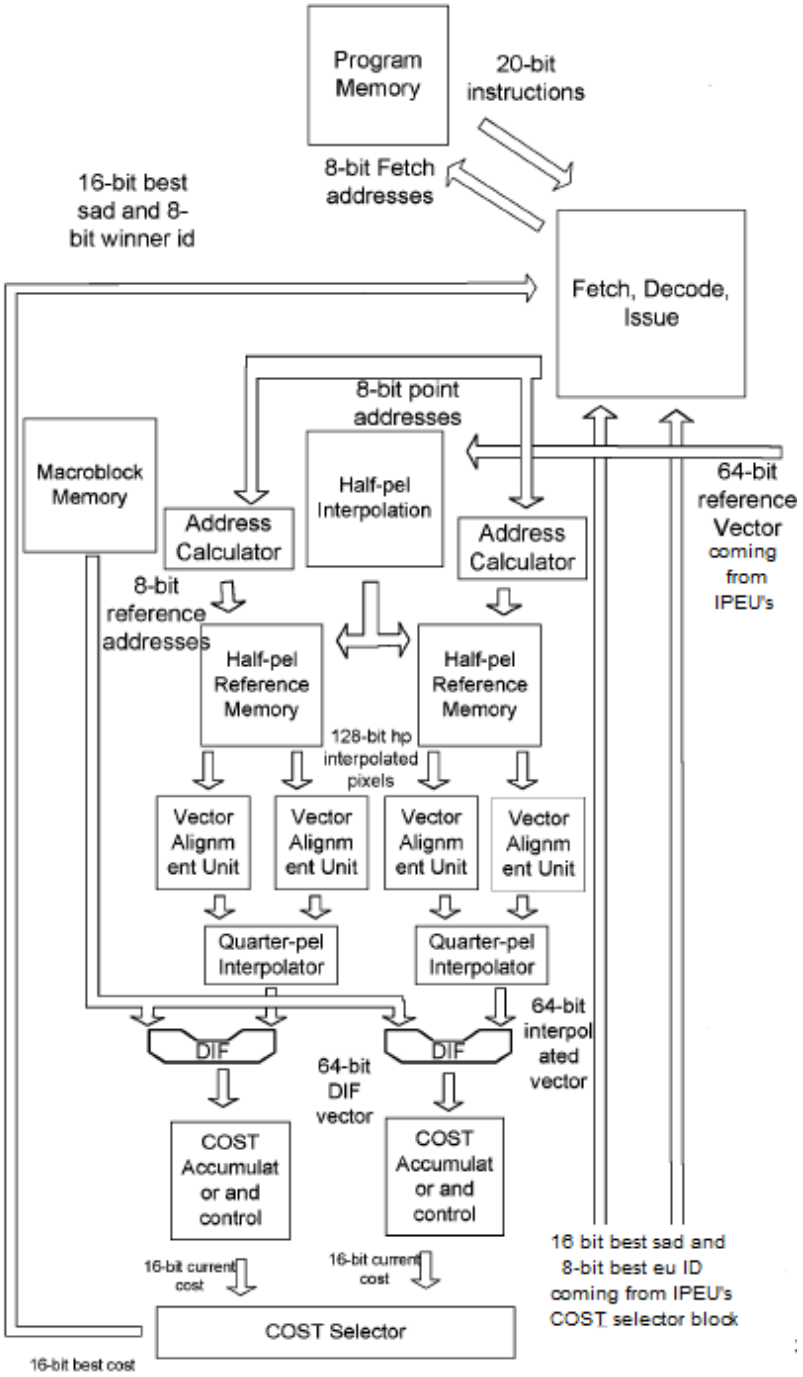

FIGURE 2.B MICROARCHITECTURE OF THE FRACTIONAL-PEL EXECUTION UNIT
Taken from [15]

The processor top interface can be seen below in Figure 2.C. The processor architecture offers 32 registers allocated for various functions like command, motion vector candidate, profile and results registers. These can be accessed through the marked register access ports. Access to both, point memory and program memory is done through the external DMA engine that interfaces with the processor using the DMA marked pins. The done_interrupt pin is asserted

when processing of a macroblock is completed. Also, the processor provides debug pins that allow access to the SAD and winner motion vector during processing [25].

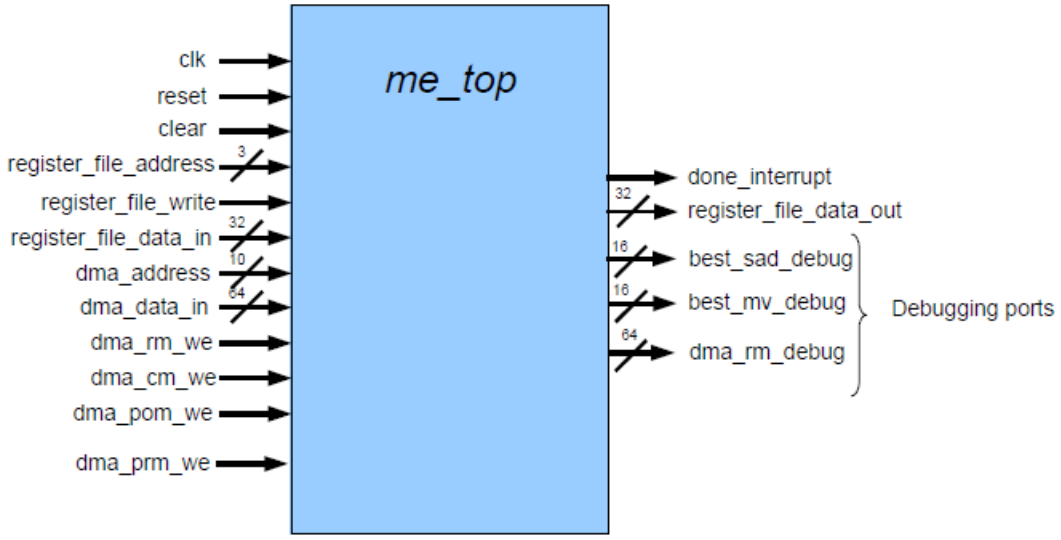

FIGURE 2.C LIQUID MOTION PROCESSOR TOP INTERFACE.
Taken from [25]

2.3 Block-matching motion estimation

Block matching motion estimation is a fundamental process for high definition video coding standards. Interframe prediction coding is a technique used in video processing in which, for a given group of frames, only the differences between said frames are encoded. The process takes advantage of the temporal and spatial redundancy found across successive frames of a video sequence. By doing this, the amount of total information encoded, the transmission bitrate, is significantly decreased, allowing to encoded large video sequences using a smaller amount of data [26],[27]. A more effective prediction means that fewer differences between frames are coded, maximizing data compression. For this reason, when encoding video sequences containing moving objects (meaning that the object is changing its position across the frame), then the motion of said object is need to be taken into consideration in order to reduce interframe prediction errors. To calculate the displacement of moving objects from frame to frame, the current frame is partitioned in several blocks called macroblocks. A search is then conducted over a search area in order to match one of the macroblocks from the current frame to another block in the reference frame [27].

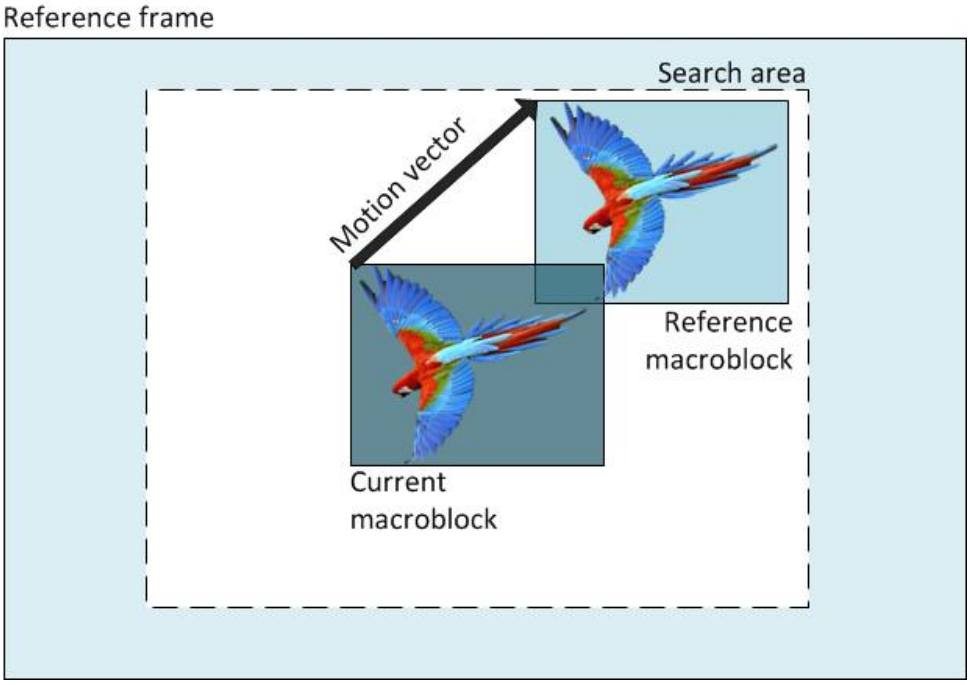

FIGURE 2.D BLOCK MATCHING IN MOTION ESTIMATION.

The search algorithm moves a block window around the search area in the reference frame until the best match is found. The search area defines the maximum displacement of the block that is acceptable for the interframe coding scheme, and the way in which the block window moves across the search area is determined by the algorithm used. The matching criteria is defined as that which minimizes the distortion between the blocks being matched, an several methods exist to evaluate this criteria, with the most common being the mean absolute difference (MAD) between blocks. Once the match is found, the displacement of the block can be expressed as a vector, called the motion vector, and the current block can be encoded with the vector information, plus the difference between the matched blocks [26],[27]. The block matching procedure can be observed above in Figure 2.D.

The determination of the displacement of the objects within the frame, using a per block basis scheme is called block matching motion estimation. It's worth noting that the match criteria used in the LIquidMotion processor is the sum of absolute differences (SAD) [15].

2.4 Motion estimation algorithms

When block-matching motion estimation is performed, the most straightforward approach to find the best matching block in the reference frame is to compute the mean absolute difference for all the points contained inside the search area. This approach, while accurate, comes at a high computational cost. The need to reduce computational complexity has led to the use of fast motion estimation algorithms that aim at reducing the number of search points calculated during block matching, at the cost of a slight increase in frame distortion [26]

2.4.1 Three step search (TSS)

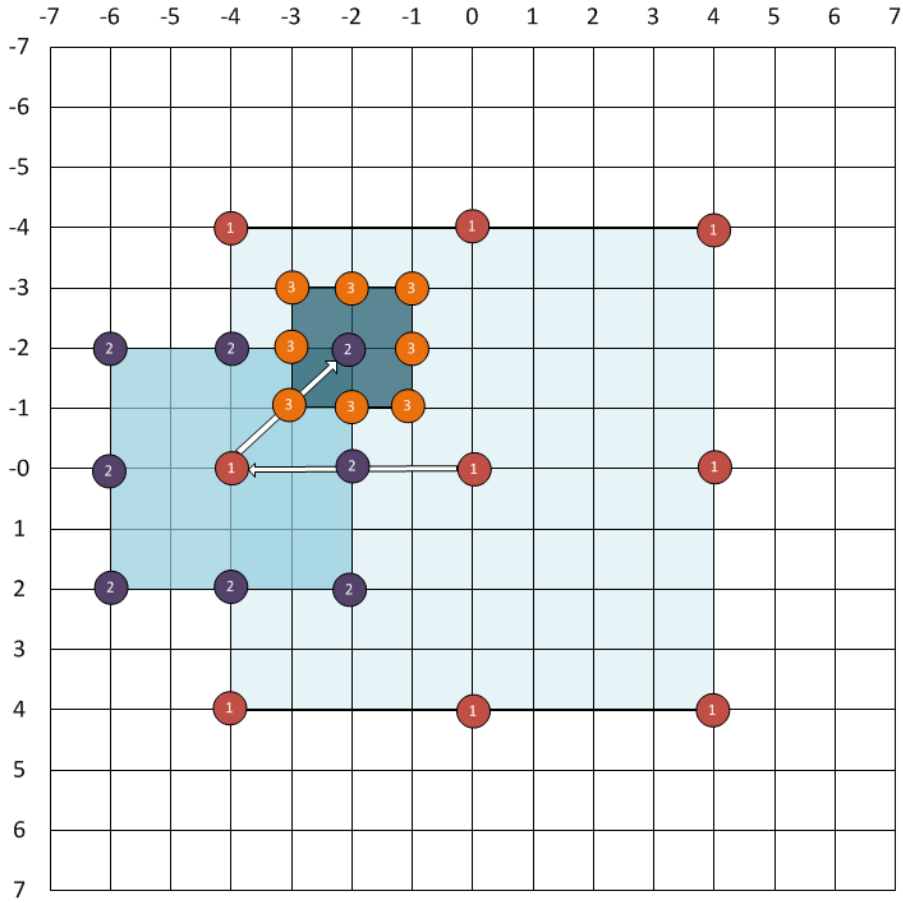

FIGURE 2.E THREE STEP SEARCH (TSS) ALGORITHM PROCEDURE

One of the earliest fast block-matching motion estimation algorithms to be proposed was the three step search. The algorithm begins the search at the center of the search window, setting a step size "S"of 4, which refers to the number of pixels/points that separate the search points that are to be calculated. In the first step the center point is checked, along with the points set at +/- S around it, in a square fashion. The process is then repeated for two additional iterations with the new center for the next step located at the point with the lowest cost in the previous step. The cost refers to the search point where the match criteria was the lowest of all points calculated in that step. The step size is halved with each successive iteration [28].

2.4.2 Four step search (FSS)

This algorithm operates in a similar way to the TSS, while providing an option for early termination in case the algorithm locates the best matching block in fewer steps than expected. As in TSS, the first step calculates the center point, plus the eight points around, but FSS uses a constant step size of 2 until the final step. If the point of lowest cost is located at the center of the square shape, then the algorithm jumps to the final step, in which the step size is reduced to 1. Otherwise, the algorithm performs a second iteration with a step size of 2, with the possibility of skipping to the final step if the lowest cost point is found at the center of the square [28]. The search procedure used in the FSS algorithm is represented graphically in Figure 2.F.

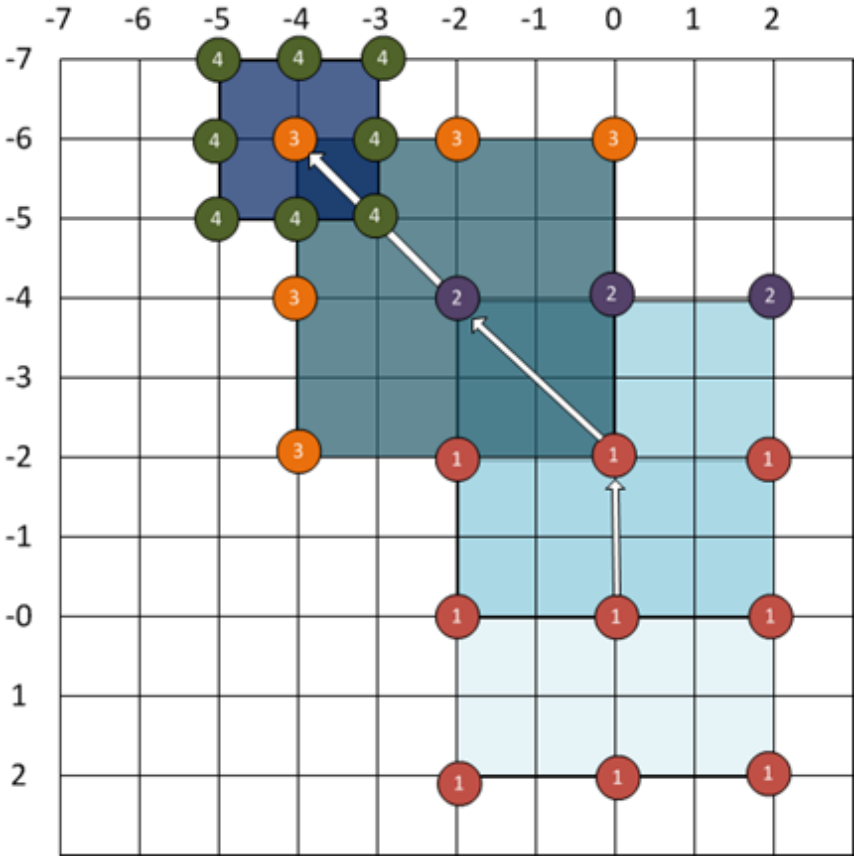

FIGURE 2.F FOUR STEP SEARCH (FSS) ALGORITHM PROCEDURE

2.4.3 Hexagon based search (HEXBS)

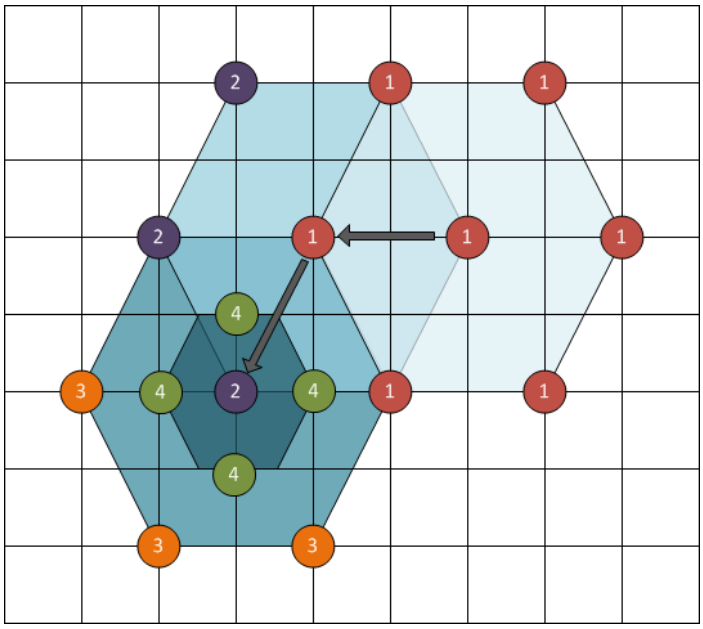

FIGURE 2.G HEXAGON BASED (HEXBS) ALGORITHM PROCEDURE, USING DIAMOND R

As the name implies, the HEXBS algorithm uses a hexagon shape composed of the center point and 6 surrounding points completing the shape. Since the hexagon more closely resembles a circle, it is proposed in [26] that this algorithm is capable of uniformly tracking motion in all directions, while at the same time providing redundancy between each step to reduce the total number of calculated search points. This can be appreciated graphically in Figures 2.G and 2.H.

The HEXBS algorithm works as follows:

- The algorithm checks all 7 points in the hexagon shape, the one in the center of the hexagon shape, two others located at a distance of 2 from the center and the other four points at a distance of $\sqrt{5}$ from the center
- Successive steps set the new center of the hexagon shape at the location of the lowest cost calculated in the previous step.
- If the lowest cost point is located at the center of the hexagon, then a final refinement step follows. Two variations of the algorithm are presented here:

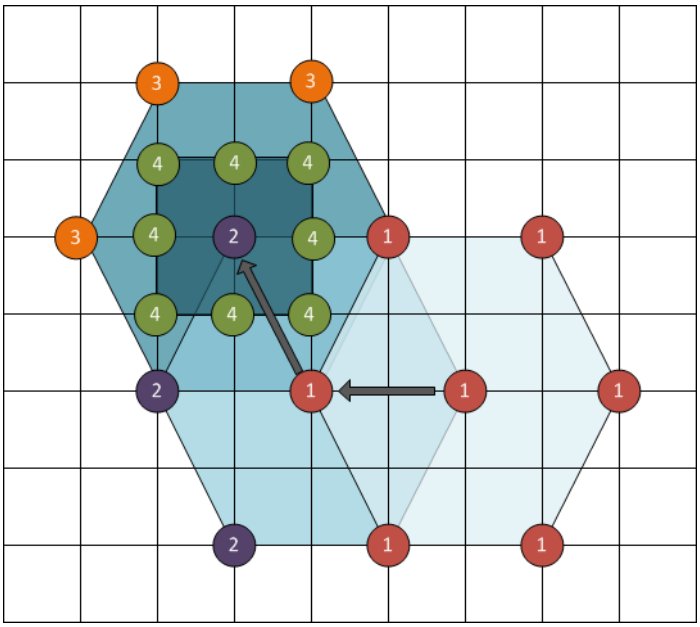

FIGURE 2.H HEXAGON BASED (HEXBS) ALGORITHM PROCEDURE, USING SQUARE REFINEMENT

A) The traditional HEXBS uses a diamond shape consisting of points located at a distance +/- 1 from the last lowest cost calculated point. This is detailed in Figure 2.G [26].
B) As used in the x264 library for encoding H.264, a square refinement, around the center point is used in the last step, at a distance +/- 1 from the center. This is appreciated in Figure 2.H [25].

- The maximum number of steps that area allowed depends on the search window size; this is because the resulting number of steps should be enough to cover to entire search window area in any direction.

2.4.4 Diamond search (DS)

The DS search algorithm makes use of a diamond search pattern and in principle operates in a very similar fashion as FSS, the main difference being the fact that the DS search doesn't have a maximum number of steps. The DS algorithm operates as follows

- It starts at the center of the search area, by calculating the cost of the 9 points forming the diamond, as shown in the points marked in red in figure 2.I.
- For the next iteration, the center of the new diamond is relocated to the point having the lowest cost in the previous iteration. Should this point be the center of the diamond, then the algorithm skips to the final step

- For the final step, the diamond is made smaller, consisting of only 4 points, not including the center point. The cost is calculated for this smaller pattern and the motion vector set to the lowest cost point

The characteristics of this algorithm allow it to get low distortion while greatly reducing the computational cost, when comparing it to the full search algorithm

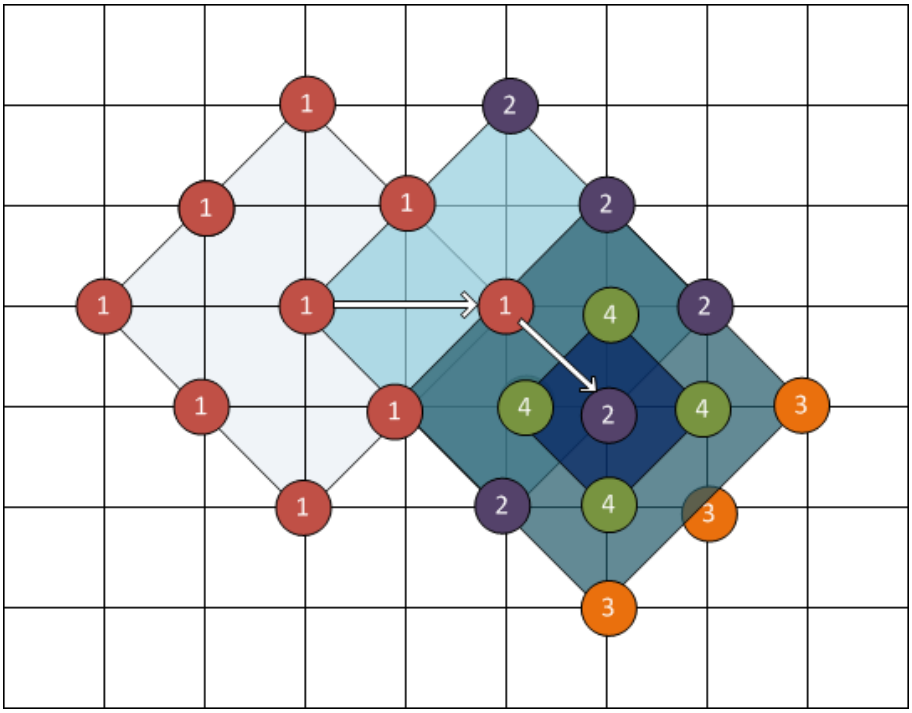

FIGURE 2.I DIAMOND SEARCH (DS) ALGORITHM PROCEDURE

2.4.5 Logarithmic search (LOG)

The LOG algorithm bears some similarities with both TSS and DS. The algorithm successively uses a diamond pattern, with the initial step size set at a distance of 4 from the center. The cost of all 5 points in the diamond is calculated, and then the next pattern location is shifted to the lowest cost point. If the lowest cost was found at the center point in the previous step, then the step size is halved and the cost is calculated for the new smaller sized pattern. The LOG search algorithm procedure can be appreciated graphically in Figure 2.J.

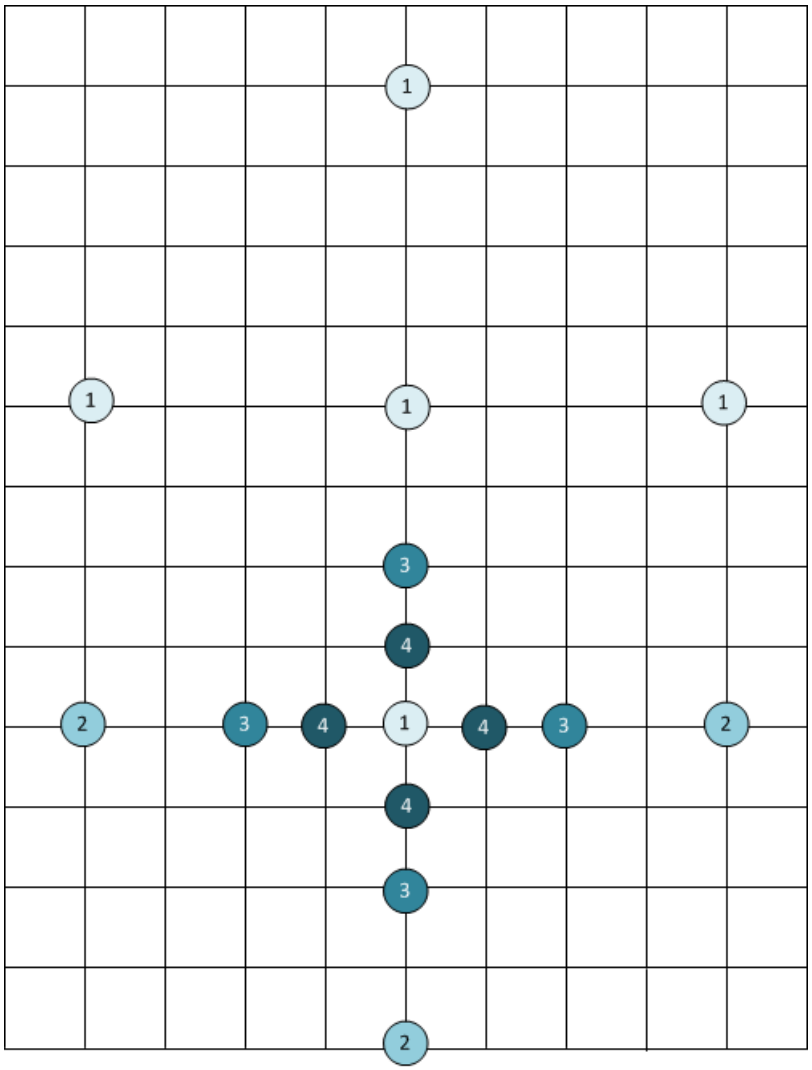

FIGURE 2.J LOGARITHMIC SEARCH (LOG) ALGORITHM PROCEDURE

CHAPTER 3: METHODOLOGY AND IMPLEMENTATION OF THE POWER REDUCTION FEATURES

3.1 Power consumption benchmarking methodology

Since the main objective of the current project was to reduce the power consumption of an FPGA-based design described using VHDL, a reliable and effective way had to be established in order to determine the power consumption of the original motion estimation processor, and to benchmark it again, after the power reduction enhancements have been completed. The Xilinx XPower Analyzer was the main tool used to generate power consumption data. Shown in Figure 3.A is the data flow that was fed into the power estimation tool in order to obtain these results. The processor netlist was generated from the VHDL source code, and later on, timing simulations using a testbench were carried out. The generated data was used to obtain a SAIF (Switching activity interchange file) stimulus file. This file was processed alongside data

generated after the synthesis, mapping and place & route process of the design had taken place. The resulting power estimation report was the means by which the processor power consumption was measured and compared. Also, Figure 3.A shows two additional steps to the ones described above (shown in dotted lines) that were added to account for the modifications made to the processor description (dotted, in blue) and the search algorithm that was executed on the processor (dotted, in red).

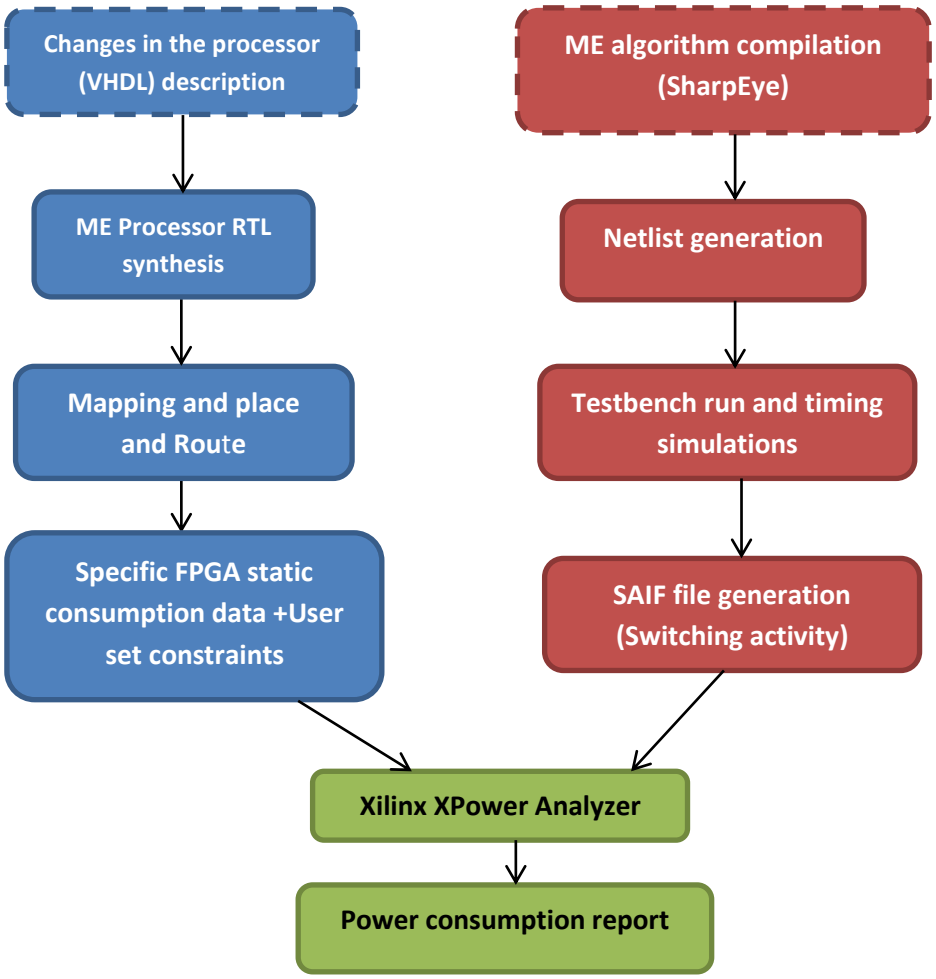

FIGURE 3.A POWER BENCHMARKING METHODOLOGY

3.1.1 The Processor Testbench

As part of the source code provided by the Bristol Microlectronics Group, a testbench was made available that allowed to test the functionality of the processor and its correct operation. The testbench was designed using VHDL code and the components instantiated inside it are the LiquidMotion processor, the reference and current memories, a logic description to simulate the behavior of a DMA scheme that feeds data into the processor, as well as a set of verification checkpoints used to test that the correct motion vectors are being calculated by the processor.

It's worth noting at this point that some of the processor elements, like the point and program memory cores, are not provided as source code, but rather as netlists generated using the

memory contents, and a configuration file containing the specifications of the memory functionality (size, write mode, etc.).

The testbench contains a series of macroblocks, corresponding to the “current” frame memory, that are matched to the data of a reference frame. The motion processor is originally set to use an HEXBS algorithm using a square refinement and a maximum of 8 search steps. The data concerning the search algorithm that the processor is set to execute is contained inside the point and program memory cores.

During the testbench execution the processor attempts to calculate the motion vector corresponding to the current macroblocks. The testbench then makes use of hard-coded values to compare the resulting motion estimation vector to the value that is expected as a result.

The testbench calculates the motion vector of each macroblock and then moves to match the next macroblock. Once all of the macroblocks have been processed the processor jumps back to process the first macroblock. Seven macroblocks are provided with the testbench and a complete iteration means that the processor has matched all of them.

3.1.2 Functionality of the Xpower Analyzer

The Xilinx Xpower is a power analysis tool that allows the user to visualize power consumption in FPGA-based designs [13], [15]. The tool is meant to be used late in the design implementation cycle, after the place and routing step has taken place. The tool is used by means of a graphical user interface (GUI) and by analyzing a design it’s capable of producing a detailed report of power consumption. It is able to list the power consumed by each of the resources in the FPGA, like clock trees, logic, I/O’s and dedicated blocks like DSP slices and RAM blocks. It also offers a thermal estimation based on the stimulus set for the design and operating conditions specified [13].

The Xilinx Power Analyzer calculates the dynamic power consumption of the design on the basis that the wasted power is due to switching of logic elements. Each of those elements drives an associated capacitance [16], which depends on the number of lines driven, the intrinsic capacitances of the process used to fabricate the chip, and the length of the routing. A stimulus file containing activity rates for the clock and inputs can be loaded into the tool or user specified values can be used instead. Activity rates are “the rate at which a net or logic element switches” (or toggles) [16]. For synchronous elements, toggle rates are assigned based on the clock signal that drives them, that is, switching of the element occurs at some percentage rate of the driving lock signal. Said percentage can be set manually by the user to analyze power consumption under various circuit conditions. As for the asynchronous elements in the design, the activity rate equals the rate at which those elements switch.

The stimulus and activity rates provided by the user are then processed along with chip specific data like capacitance and static power consumption for the particular FPGA I which the design is being implemented [16]. Power consumption for each element is then calculated as a function of: driven capacitance, voltage used to power the circuit or element, activity rate and clock frequency.

For activity rates not specified in the stimulus file or if the user needs to manually adjust them, the XPower Analyzer provides the means to individually set the rates of input signals and also clock frequencies for each clock tree. It's also possible to adjust other values like the core operating voltage and the global default rate, which is the default rate set for any signal with no specified rate. Moreover, the tool is also able to accept data regarding the loads that the FPGA will be driving at the output pins, like the current the load will be drawing from the pin (for DC loads) and the capacitance value for capacitive loads [15].

As stated before, thermal information is also generated when creating the power report for a design. It is possible to specify a set of variables that directly influence the thermal results, such as, ambient operating temperature within standard variations (-40°C to 125°C), air flow and chip package [15]. For the purpose of this project, all power estimations were made using standard temperature conditions (25 °C).

The XPower tool is capable to accept stimulus files in VCD (Value Change Dump) and SAIF format. These kinds of files contain simulation data generated by HDL simulators like Modelsim. For this purpose, it's possible to create and simulate a testbench in order to create reliable input information values for the design. The VCD/SAIF file is then generated and imported into the XPower Analyzer, which reads the data and calculates the design power consumption based on these values [15]. For a realistic analysis of power consumption, this is the preferred approach but manual setting of signal activity rates and other operation values give the designer additional information about the implementation, especially when investigating corner cases. The tool is able to report power consumption in terms of dynamic and quiescent power, a feature which is fundamental to evaluate the results of the current project.

The relatively simple, yet detailed, power estimation report makes it possible to compare data from different constraints on the inputs and design changes with relative ease. The Xilinx XPower Analyzer, overall, provides a reliable tool to benchmark the power consumption of FPGA-based designs as shown by its use in several papers researching this area like in [1] and [17]. For this project, it was used in conjunction with post-place & route timing simulations and a reliable testbench, to provide an accurate power estimation of the physical implementation of the LiquidMotion processor.

3.1.3 Power estimation

The standard procedure used across this project to obtain the estimation of the processor power consumption involved the use of post-place & route simulations to generate the stimulus files containing the switching activity rates.

For this project, post P&R simulations were carried out using the ISim simulator, and the stimulus activity file format chosen was the SAIF format. The ISim simulator was chosen due to its integration with the rest of the tools used for this project, when compared with more popular tools like Modelsim. Also, early tests reported a slightly higher level of confidence in the power estimation results when using the ISim simulator (this figure is determined by the number of nets within the design whose activity rates were obtained from simulation data with respect to the total number of nets in the design). The confidence level of the power

estimation was between 75% and 80%, depending on the processor implementation. The SAIF format was favored due to the reduced file size over VCD.

The power estimation dataflow is shown below in Figure 3.B and is explained fully in the following description

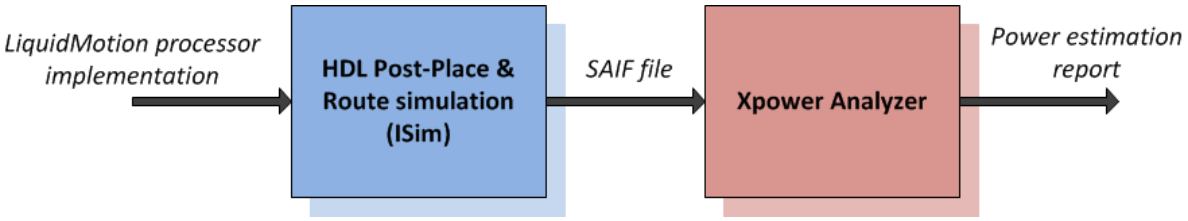

FIGURE 3.B POWER ESTIMATION DATAFLOW

After the processor design was fully implemented, the resulting netlists were exported into ISim, along with the target device configuration. Unless stated otherwise for specific cases in the following sections of this chapter, all timing simulations carried as part of this project involved a 1000 us simulation of the testbench described in section 3.1.1 of this work.

Once the post-P&R simulation was complete, the resulting SAIF file was processed using the XPower Analyzer. A power estimation report was then generated, and the results contained within it used to measure the resulting power consumption reduction for each implementation. The power estimation methodology described in this section was used for all the experiments described in the following sections of this chapter.

3.2 Use of the implementation toolset to change the processor performance

This section of the project involved the use of the various available options contained in the implementation toolkit, the Xilinx ISE software, to directly affect the processor implementation in terms of power, speed, and area. The ISE tool provides a number of predetermined configuration sets, aimed to impact different features of the design implementation. These configurations sets, referred as strategies, provide general optimizations that are meant to positively impact the design performance in the particular areas stated before. More specific options, which are not included in these predetermined strategies, are available as well. This opens the possibility to tune the options set in the implementation toolkit to maximize the performance obtained from the implemented design. Additionally, when one of the predetermined strategies is in use, it's also possible to make use of the SmartXplorer tool, which allows for automated implementation of the design using variations of the selected strategy, to a certain extent. This helps to further increase the impact of the existent strategies when used they are used to implement the design. Manual tuning of the implementation options is then carried to achieve an optimal strategy, specifically created for the LiquidMotion processor, based on the target device and on the nature of the design itself.

The processor implementation is also affected by the timing constraints set by the user. As explored in [15] the LiquidMotion processor is operated at 200 MHz as its standard frequency. The original source files of the processor include a timing constrain of 5 ns, set on the clock network, so the implementation tool is directed to guide the processor implementation to try

to meet this timing constrain, within the physical limits of the target device and the processor description. Typically, the closer the timing constraint is to the maximum achievable limit, the more constrained the implementation is and this results in the final implementation consuming a higher amount of power than a similar implementation clocked at a slightly reduced rate, due to implementation tradeoffs the tool makes in order to achieve the requested frequency. More about the findings presented in this work regarding this specific topic are found in the analysis and conclusions at the end of the present work.

Changes to the processor implementation were explored at three different stages of the implementation process:

- Synthesis
- Mapping
- Place & Route

The main device target for this part of the project was a Virtex-5 FPGA, a platform in which the processor was already tested successfully a number of times. Additionally, in order to further explore some implementation enhancements that are only available on newer FPGA devices, like automatic clock gating [11] an alternative device was targeted for implementation, a Spartan-6 FPGA. The details of both devices are briefly summarized below in Tables 3.1 and 3.2 below.

Device target	XC5VLX110T
Package	ff1738
Speed	-3
Synthesis tool	XST
Timing constraint on CLK network	10 [ns], 5 [ns], 4 [ns]

TABLE 3.1 VIRTEX-5 DEVICE IMPLEMENTATION CONFIGURATION DETAILS

Device target	XC6SLX100T
Package	fgg484
Speed	-3
Synthesis tool	XST
Timing constraint on CLK network	10 [ns], 5 [ns]

TABLE 3.2 SPARTAN-6 DEVICE IMPLEMENTATION CONFIGURATION DETAILS

3.2.1 Implementation variations using the Virtex-5 device

The following strategies are aimed at directly reducing power consumption, increase the design clock rate or reduce the allocated resources to implement the design and are available within the ISE tool for Virtex-5 devices as the implementation target:

- Power optimization with/without physical synthesis

- Timing performance with/without IOB packing
- Timing performance with physical synthesis
- Area reduction with/without physical synthesis

The custom implementation strategies were aimed at maximizing power reduction by increasing the level of computation effort the implementation tools take to process the design. The custom strategies created are shown below in Table 3.3.

Strategy	Custom implementation options
Power reduction Map + P&R	Map and P&R: Effort Level-> High, Extra effort-> Normal, Power reduction-> On
Power reduction Synthesis + Map + P&R	***Same as above, plus*** Synthesis: Optimization Goal-> Area, Optimization effort-> High, Power reduction-> On

TABLE 3.3 CUSTOM STRATEGIES IMPLEMENTED ON THE VIRTEX-5 DEVICE

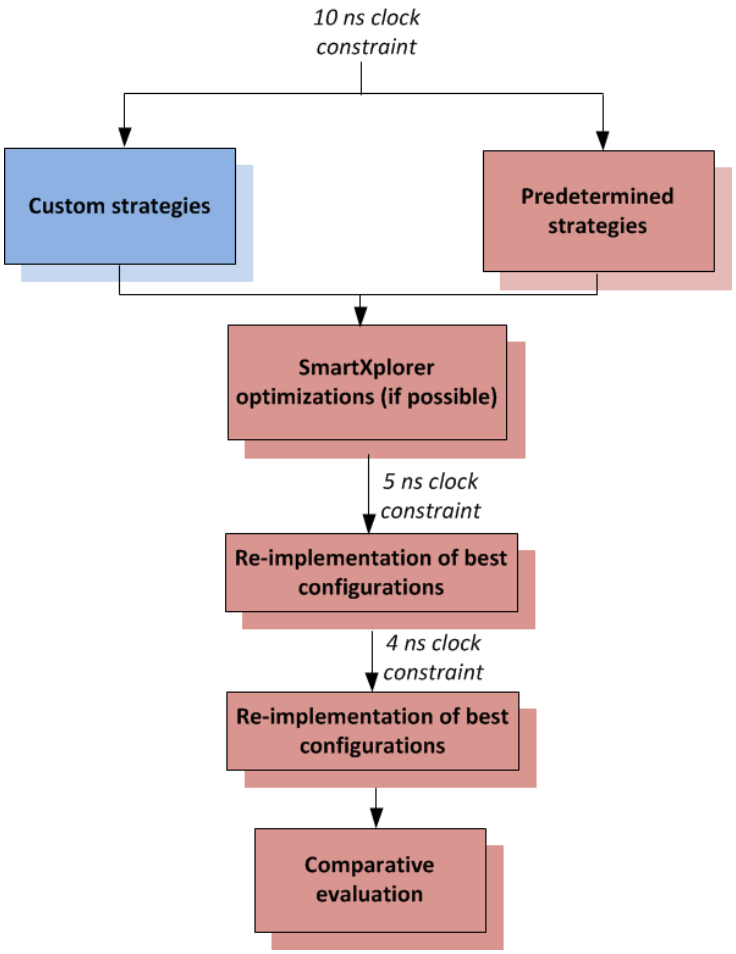

FIGURE 3.C METHODOLOGY DATAFLOW FOR THE IMPLEMENTATIONS TARGETNG THE VIRTEX-5 DEVICE

To explore the impact of these strategies, and other implementation options, on the design performance, the following, consecutive steps (shown as a dataflow diagram in Figure 3.C) were taken.

1.- Implementation of this strategies, using the settings shown in Table 3.1 and a timing constraint of 10 ns. Once each design was implemented, the standard procedure detailed in section 3.1 was carried out to estimate the power consumption of each strategy. The customized sets of strategies were then created in order to explore its impact on the implementation performance. A comparative analysis then followed to evaluate the effectiveness of each one to reduce power consumption, meet the timing constraint and reduce allocated resources for the implementation. The default implementation strategy was included in this analysis, for comparative purposes.

2.- Based on the results obtained from previous steps, the strategy judged as superior to increase the processor frequency, and the one best suited to reduce power consumption were then re-implemented using a timing constraint of 5 ns. If one of these strategies was a predetermined strategy, then the SmartXplorer tool was used to adjust said strategy, to increase its impact on the processor performance. If that was not the case, then the strategy was re-implemented without further adjustments. The power estimation for both strategies was then obtained using the established procedure and the total energy consumption at the maximum achievable frequency for each implementation was calculated.

The completion time at processor maximum frequency for each implementation was calculated as:

$$\text{Completion time @ max. freq} = \text{Completion time @ test freq.} \div \frac{\text{Max. achievable freq.}}{\text{Test frequency}}$$

Drawing from the equation in [12] for dynamic power consumption in FPGA's:

$$\text{Dynamic Power} = CV^2 f$$

Then the following approach is valid to obtain an accurate estimation on the processor average dynamic power consumption at different frequencies:

$$\text{Avg. dynamic power @ max. freq} = \frac{\text{Avg. dynamic power @ test freq.}}{\text{Test frequency}} \times \text{Max. achievable freq.}$$

The total energy consumption was then calculated as:

$$\text{Total energy consumption} = \left(\text{Completion time @ max. freq}\right) \times \left(\text{Average power consumption @ max. freq}\right)$$

A comparative analysis then followed according to these results, as presented in Section 4.1.1 of the present work

3.- A similar procedure to the one described in the previous step was carried out, this time using a timing constraint of 4 ns (corresponding to 250 MHz processor speed). As above, the total power consumption was calculated for both implementations and evaluated by means of a comparative analysis.

3.2.2 Implementation variations using the Spartan-6 device

Since the implementation toolkit had a wider range of custom optimization options available, this phase of the project was focused on customized strategies and the impact of specific implementation options.

The specific options considered for exploration were chosen based on the results obtained from section 3.2.1 of this chapter, as well as the architecture of the processor itself. These options were deemed to have the potential to decrease the power consumption of the processor and they were explored on a case-by-case basis, as every option was implemented separately and their impact on the processor performance was evaluated in terms of negative or positive impact on the power consumption. The methodology described in this Section is presented as a dataflow diagram in Figure 3.D.

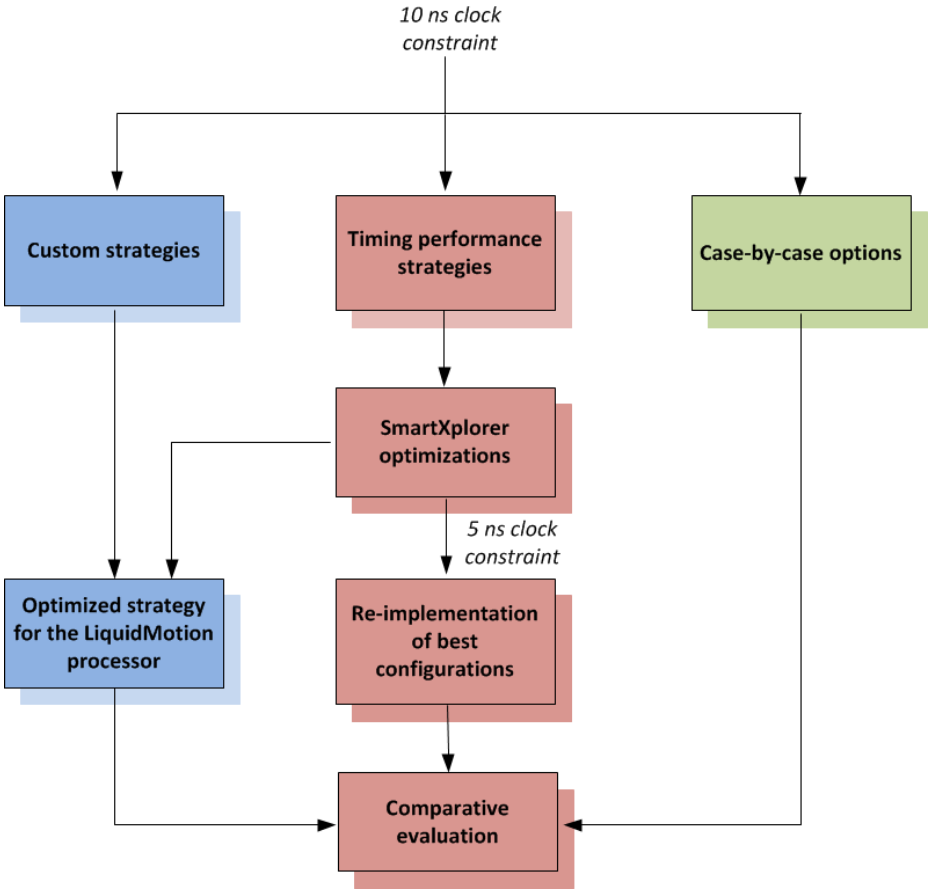

FIGURE 3.D METHODOLOGY DATAFLOW FOR THE IMPLEMENTATIONS TARGETNG THE SPARTAN-6 DEVICE

Again, based on the results from the previous section; from the available predefined strategies provided by the toolkit, only the ones aimed at increasing the processor clock rate were implemented. The rest of the strategies explored are completely customized and their details are shown in Table 3.4.

Other specific options considered for implementation on a case-by-case basis, and not listed inside any of the strategies:

- For Synthesis:
FSM encoding, RAM style, Auto BRAM packing.

- For Mapping:
Map slice logic.

Custom strategy	Synthesis	Map Options	P&R
Power reduction: Synthesis	*Optimization Goal:* Area *Optimization effort:* High *Power reduction:* On	Default	Default
Power reduction: Map + P&R	Default	*Effort Level:* High *Extra effort:* Normal *Power reduction:* On	*Effort Level:* High *Extra effort:* Normal *Power reduction:* On
Power reduction: Map + P&R + Clock gating	Default	*Effort Level:* High *Extra effort:* Normal *Power reduction:* High	*Effort Level:* High *Extra effort:* Normal *Power reduction:* High
Power reduction: enhanced Map+ P&R	Default	*Effort Level:* High *Extra effort:* Normal *Power reduction:* Extra effort *Combinatorial logic optimization:* On *LUT combining:* Area *Global Optimization:* Power	*Effort Level:* High *Extra effort:* Normal *Power reduction:* High
Power reduction: Area synthesis + enhanced Map + P&R	*Optimization goal:* Area *Optimization effort:* High *Power reduction:* On	*Effort Level:* High *Extra effort:* Normal *Power reduction:* Extra effort *Combinatorial logic optimization:* On *LUT combining:* Area *Global Optimization:* Power	*Effort Level:* High *Extra effort:* Normal *Power reduction:* High
Power reduction: Speed Synthesis + enhanced Map +P&R	*Optimization goal:* Speed *Optimization effort:* High *Power reduction:* On	*Effort Level:* High *Extra effort:* Normal *Power reduction:* Extra effort *Combinatorial logic optimization:* On *LUT combining:* Area *Global Optimization:* Power	*Effort Level:* High *Extra effort:* Normal *Power reduction:* High
Power reduction: Optimized	Default	*Effort Level:* High *Extra effort:* Normal *Power reduction:* Extra effort *Combinatorial logic optimization:* On *LUT combining:* Area *Global Optimization:* Power *Register Duplication:* On *Pack I/O registers into IOBs:* For Inputs and Outputs	*Effort Level:* High *Extra effort:* Normal *Power reduction:* High

TABLE 3.4 CUSTOM STRATEGIES IMPLEMENTED ON THE SPARTAN-6 DEVICE

Finally, taking advantage of the additional options available for the SmartXplorer tool when used with a Spartan-6 device, the predefined strategies provided with the tool that yielded the

fastest clock rate and the lowest power consumption were optimized using a 64 cycle iteration, using timing performance and power reduction algorithms respectively. The settings for the resulting implementations were combined with the already collected results and merged into an optimized strategy, listed in Table 3.4 as “Power reduction and Timing Performance: Optimized” Both this strategies were evaluated in terms of total energy consumption using a comparative analysis.

In a way similar to the procedure followed in section 3.2.1,the two configurations mentioned above were re-implemented using a clock signal constraint of 5 ns, corresponding to a clock rate of 200 Mhz. Likewise, the results obtained from these implementations were added into the comparative analysis.

Power estimation for all of the implementations explored in this section, were obtained using the procedure described in section 3.1 of the present work.

3.3 Implementation of various motion estimation algorithms

For this phase of the project, the main tool used was the SharpEye Studio v3.2.2. This toolset was specifically developed by the Bristol microelectronics group to use along with the motion estimation processor. It consists of assembler, compiler, a cycle accurate model, and analysis tools [25]. Both assembler and compiler use a C-like language that was created specifically to be able to develop motion estimation algorithms in a simple way, the EstimoC language. Apart from the standard C syntaxis, the language makes use of special structures aimed to express ME algorithms [17].

3.3.1 Methodology

Using the EstimoC language, the six fast-motion estimation algorithms described in Chapter 2.4 of the present work were implemented, plus the FS algorithm for comparative purposes. The source code was assembled and compiled using the toolkit and the default configuration file settings. As a result, MIF (memory initialization files) were obtained, one corresponding to the point memory and the other one to the program memory of the processor. The source code for the algorithms can be found in Appendix I of the present work. The source code for the FS and LOG algorithms was taken from the processor documentation with credit to their respective authors. The code for the HEXBS with square refinement is a modified version and properly credited as well. Original code was written for the rest of the algorithms.

3.3.2 Behavioural simulations

For the purpose of behavioral simulation, the MIF files were used directly to the speed of each algorithm and proper functionality. Behavioral simulations were carried in this way for each one of the ME algorithms. The testbench was then run for 1000 [us], with the exception of the FS algorithm which required a 6000 [us] run in order to go through all the macroblocks used in the testbench. Completion time was then recorded for each one of the algorithms at the time the motion vector was obtained for the 7^{th} macroblock

3.3.3 Design Implementation configuration and post-P&R simulations

In order to carry timing simulations and get the final power estimation results for each algorithm, the netlist files containing the point memory and program memory had to be regenerated using the Xilinx CORE Generator tool. Summarized in Table 3.5 below is the configuration used to regenerate both memory netlists. It's worth noting that the reason to use a Spartan-3 device in the configuration was, that the original memory netlists were created using this device as a target, and so, this configuration was kept in order to avoid possible implementation differences.

	Point Memory	Program Memory
Width [bits]	16	20
Depth [bits]	256	256
Port Configuration	Read & Write	
Write Mode	No read on write	
Primitive Selection	Optimize for Area	
Pin Polarity	Rising edge triggered/active high	
Target device	XC3S1500-5fg456	

TABLE 3.5 MEMORY NETLIST CONFIGURATION

The MIF files obtained for each algorithm were manually modified to convert them into COE files, which is a format acceptable for the CORE Generator tool. The COE files were then used together with the configuration stated above, to create and initialize the memory netlists.

Using the new netlists and the original processor source code, the LiquidMotion design was then implemented using the Xilinx ISE tool. The design implementation options are summarized in Table 3.6 below.

Device target	XC5VLX110T
Design Goal	Balanced
Strategy	Xilinx Default
Package	ff1738
Speed	-3
Synthesis tool	XST
Timing constraint on CLK network	10 ns

TABLE 3.6 IMPLEMENTATION CONFIGURATION DETAILS FOR THE VARIOUS MOTION ESTIMATION ALGORITHMS

Once the design was fully implemented, the standard procedure to estimate the power consumption, as described in section 3.1 of the present work, was carried out. As in the case for behavioral simulation, completion time for each algorithm was recorded at the time the motion vector was obtained for the 7th macroblock (the FS algorithm post-P&R simulation was extended to 6000 [us], as was the case for the behavioral simulation). Completion times for behavioral and post-P&R simulations were compared to verify that the values were equal.

3.4 Architectural changes into the processor description

Drawing from the information collected from the literature review about FPGA power reduction techniques, as presented in Chapter 1 of the present work, as well as the conclusions obtained from the critical analysis at the end of the review and the details about the processor functionality and architecture, a decision was taken to implement the adaptive pipeline technique into the processor description. Given the nature of FPGA based designs, the reconfigurability of the pipeline at runtime is an issue that could best be resolved at system level using dynamic partial reconfiguration as shown in [29]. This approach is preferable since it doesn't require changes into the processor description itself, which will add complexity to the processor and can be troublesome for large designs such as the LiquidMtotion processor. Because of this, it was deemed that the pipeline reconfiguration mechanism is an issue that could be better addressed in future works that can focus on the functionality of the processor. Then, the main focus of this section of the project was to create a shallow pipeline for the processor by reducing the number of registers contained in the stages. It's important to note, that the architectural changes explored here are aimed at the processor baseline configuration, meaning the processor uses only one integer pipeline.

3.4.1 Practical approach

Shown in Figure 3.E is a simplified diagram of the typical architecture found in most of the pipeline stages of the LiquidMotion processor. It contains several input ports that carry the data into the stage. The data is manipulated first using combinational logic to provide arithmetic processing, like addition, multiplication or offsetting functions. Data is then feed into registers just before the outputs and on the positive edge of the clock, data is transferred to the outputs of the registers, which in turn is connected to the output ports of the pipeline stage. The processed data is subsequently fed into the next pipeline stage. Feedback paths exist between different pipeline stages and also internally to each stage.

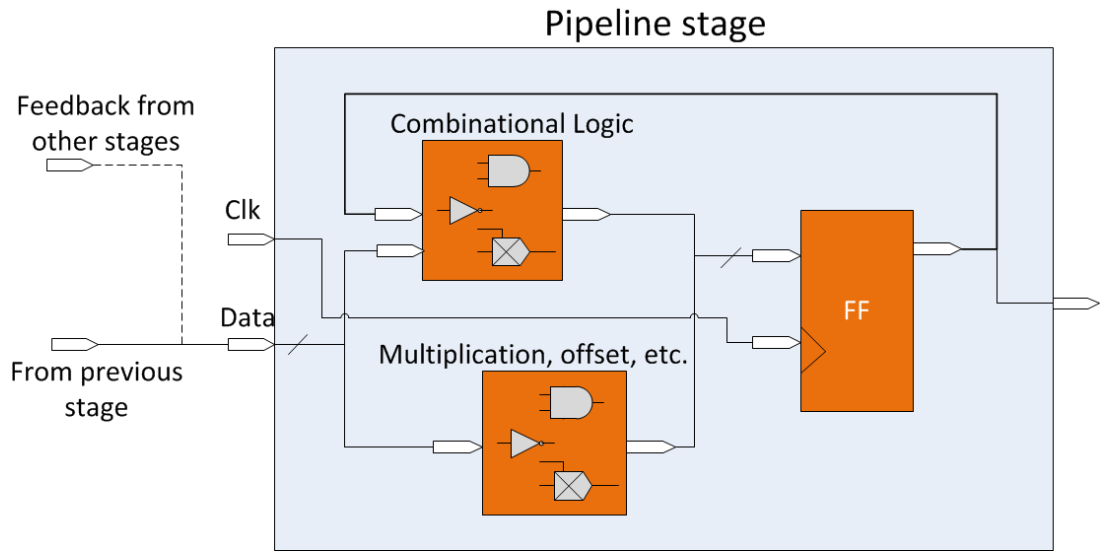

FIGURE 3.E TYPICAL ARCHITECTURE OF A PIPELINE STAGE INSIDE THE LIQUIDMOTION PROCESSOR

When implementing the processor in an FPGA platform, the registers are synthetized as flip-flops. These registers account for a high level of dynamic power consumption. As evident by the fully synchronous scheme used in the processor pipeline, every clock cycle, energy is used

to activate the flip-flop's. The approach used in this project to create the shallow pipeline, was to remove the registers in one of the pipeline stages, then allowing that stage to be merged with the stage below it. This idea is shown in Figure 3.F.

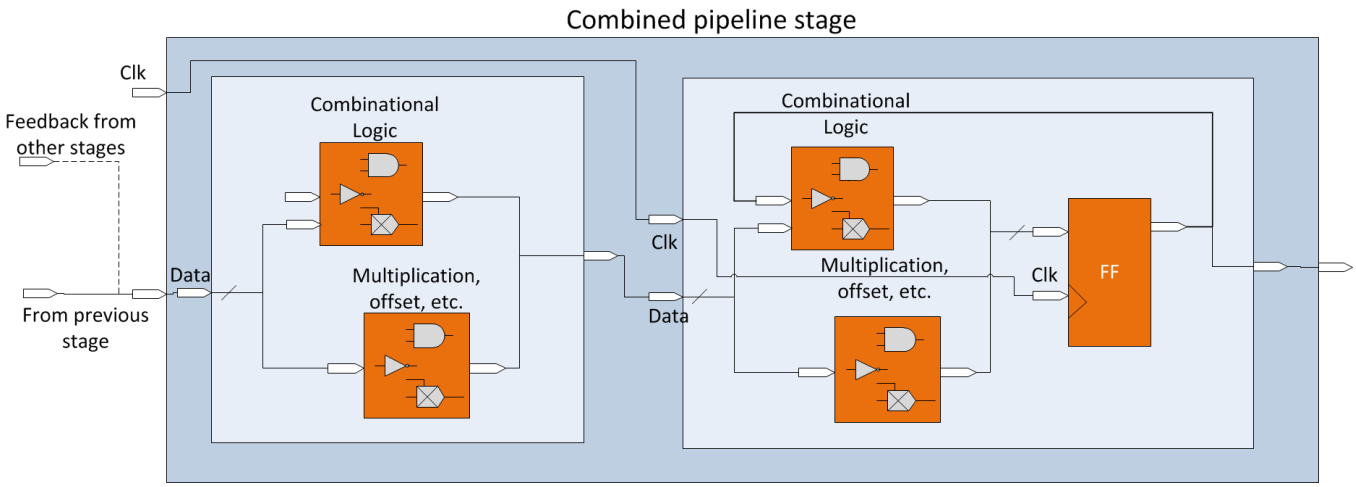

FIGURE 3.F PROPOSED MERGING OF THE PIPELINE STAGES INSIDE THE LIQUIDMOTION PROCESSOR

This approach required changes into the processor VHDL description in order to remove the registers. This was done by removing the synchronous description that was being synthesized as flip-flop's. The rest of the combinational logic was meant to remain as in the original source code, however, some minor changes had to be made in order to avoid synthetizing latches in place of the flip-flop's. Even though latches might appear to be a good replacement of the flip-flop's (due to the fact that latches consume less energy than FF's, when using an FPGA) whenever latches were synthetized in the implementation of the processor, the resulting power consumption increased significantly because of glitch propagation, an effect already anticipated when removing the registers inside the pipeline, and investigated in Chapter 1 of the current work. Other changes in the description included the removal of the input ports that controlled the synchronous scheme (clock, reset, clear) from the pipeline stages whose registers were removed, as this ports are no longer required by non-synchronous blocks.

Shown in Figure 3.G is a simplified diagram of the Liquid Motion processor integer pipeline, especial detail in the signal names is shown for the case of those stages deemed as potential candidates to have their registers removed. It's worth noting here that the processor architecture is largely nonlinear, and feedback paths coming and going to more than one stage are not uncommon. This is especially true for the Control Unit module of the pipeline. Several combinational logic blocks also exist inside these loops.

The main stages targeted to be merged were the Concatenate Unit and the Sad Selector Unit. The former was chosen due to the large size of the registers contained within the stage, which meant a large potential for power reduction, and the later for its strategic location at the end of the pipeline, which meant that it was less likely to be affected by complex feedback loops.

Essential parts of the modified source code can be found in Appendix II of the present work

3.4.2 Behavioral and Post-Place & Route simulations

Simulations were used in this phase of the project in order to verify that the changes made into the processor description were not altering the processor functionality. Behavioral simulations were used as a first approach to verify this point. For these simulations, only the source code relating to the pipeline was modified. No changes were made to the processor memories, meaning that the code was tested using the original HEXBS search algorithm contained in the memory cores. Functionality was verified by means of the MV/SAD checkpoints contained in the testbench code. 1000 us simulations were used for this purpose. However, successful execution of the testbench didn't guarantee feasibility of the design, as coding errors and erroneous descriptions can lead to either, generation of latches or to a non-synthesizable design.

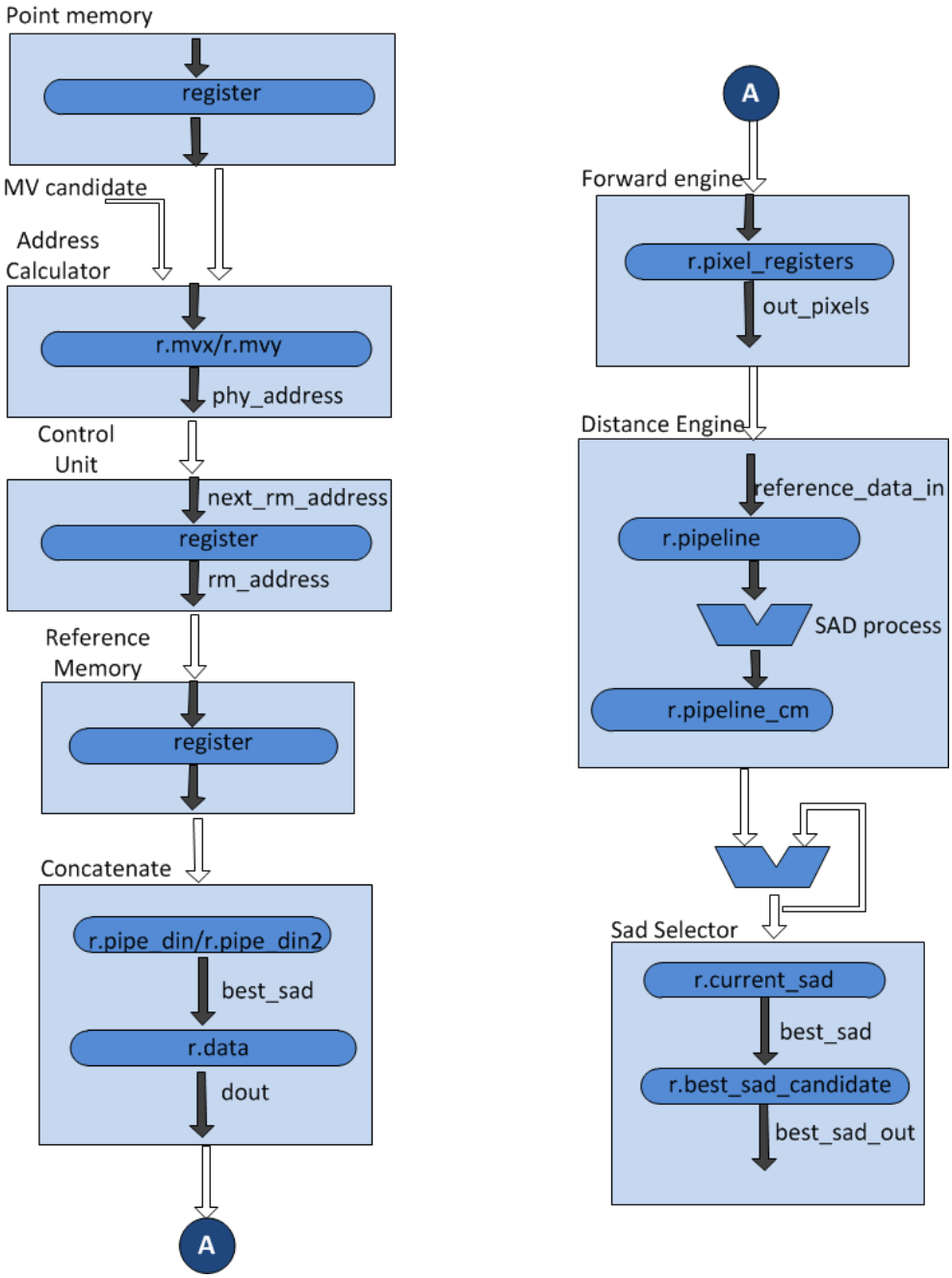

FIGURE 3.G SIMPLIFIED DIAGRAM OF THE LIQUIDMOTION PROCESSOR PIPELINE

The Post-P&R simulations were used as a second stage to verify processor functionality, but also to verify that the processor implementation was feasible and that it was successful in reducing the power consumption, compared to the original core. Feasibility of the design was judged in terms of the resulting implementation power consumption and by verifying that no latches were being generated. The later was further confirmed by looking at the Module Level Utilization report generated by the implementation tools. The total number of FF's allocated for the implementation was also recorded for comparative purposes. The configuration details regarding the processor implementation are the same as the ones listed before on Table 3.6.

As in the previous experiments, the methodology used to estimate the power consumption of the core is the one described in Section 3.1 of the present work.

CHAPTER 4: EXPERIMENTAL RESULTS AND ANALYSIS

In this chapter, the results from the experiments described on Chapter 3 are reported, evaluated, and analyzed. The discussion of the results includes a comparative and qualitative analysis of each separate experiment and also an overall discussion, presenting the advantages and drawbacks of each one of the power reduction features explored in this work.

4.1 Use and impact of the implementation tools to reduce the processor power consumption

The results obtained for the implementations listed on section 3.2 of the current work are reported in here. Shown in Table 4.A are the results regarding the processor implementation on the Virtex-5 and the results targeting the Spartan-6 are shown in Table 4.C. Reported in the tables is the total power consumption of each implementation, as well as the contribution of static and the dynamic into the total consumption. The maximum processor frequency is also reported. Significant numbers regarding the size and characteristics of each implementation are reported as well, namely the slice registers and slice LUT's which means the number of individual allocated resources for the implementation. The number of occupied slices means in how many slices are the previous mentioned resources allocated. Finally, the number of fully used LUT-FF pairs means how efficiently the LUT-FF pairs inside the slices are used, a higher number meaning more pairs were implemented. This last number also means, in an indirect way, how effectively slices are used, however, one must keep in mind that several LUT-FF are contained inside each slice (4 pairs for the Virtex-5 and, in the case of the Spartan-6 it's more complicated since one slice contains 8 LUT's and 4 FF's, meaning that 4 pairs of allocated LUT-FF's doesn't traduce into 100% slice utilization). In the case of Tables 4.B and 4.D, only the power and maximum frequency numbers are reported. The total energy consumption for these cases is reported. Since the implementations shown in these two tables work at different speeds, this was judged as the best way to compare their performance.

4.1.1 Results using the Virtex-5 device and discussion

Several aspects can be highlighted from the results presented in Table 4.A as follows:

High static power consumption: First of all, for the case of the Virtex-5 FPGA, it's obvious that there is a great difference between the dynamic and static power consumption, with the latter being more than 18 times the former. Regarding this point, it should be considered that the Virtex-5 platform, it's a high end platform, generally aimed more at timing performance issues than to power issues, and as such, a higher a tradeoff was made between lower power and higher design frequency. It's worth noting as well, that the particular FPGA used in the implementation is of a considerable high capacity, which translates into higher transistor counts. To exemplify this, the implementation utilization ratio for registers and LUT's was an average of 1% and 2% respectively, meaning there were still plenty of free resources. Considering that the main source of static power consumption is transistor leakage currents, this would explain the comparatively large results obtained for this case. In addition to this, it's important to note that the Virtex-5 platform is by now at the end of its life cycle, having been launched in 2006. These claims are further supported when one looks at the static power consumption of the alternative target platform used for this work, a Spartan-6 device, being a newer, low-cost and low-power consumption oriented FPGA.

Steady static power consumption: Directly related to the previous point is the fact that the static power consumption was at nearly constant value across all implementations. This means that, regardless of the amount of resources allocated for the implementation, the static power consumption remains steady, since this doesn't impact the source of static power wasted. It's possible to say that static power consumption is heavily dependent on the targeted FPGA device.

Dynamic power reduction: The implementation options can significantly impact the processor dynamic power consumption. For the best case, a reduction of 23.74% with respect to the original implementation was achieved on this area using the custom strategy "Power reduction Map + P&R". In this case it's possible to see from the table that the higher the number of fully used LUT-FF pair, the lower the power consumption was achieved.

About the predefined strategies: It's worth noting that in general, custom strategies fared better than the predefined counterparts both in terms of achieving lower power consumptions and also, while not the intended goal, custom strategies achieved higher implementation clock speeds than the original core. The higher speeds for the lower consumption strategies are thought to be the result of a more efficient and compact implementation, which means there are less switching components and shorter distances in the data paths.

Maximum achievable frequency: In general, all predefined strategies achieved noticeably lower clock speeds than the original, the exception being the "Timing performance w/o IOB packing" which fared only slightly lower than the original implementation and the custom "Power reduction Map + P&R" strategy. The later strategy achieved a 3.35% increase in clock frequency compared to the original core. The interpretation given to this was that the LiquidMotion processor may benefit more from a custom tailored strategy than any of the predefined ones provided by the implementation toolkit. This result motivated the exploration of a larger array of options and customs strategies in the experiment conducted using the Spartan-6 platform.

Options	Dynamic Power (avg.) [mW]	Static Power [mW]	Total Power [mW]	Max processor frecuency [Mhz]	Slice registers/ LUT's	Occupied slices	Fully used LUT-FF pairs
Design timing constraint goal: 100 Mhz							
Default	48.77	901.60	950.38	141.383	825/1,851	639	772
Power reduction Map + P&R	37.19	901.48	938.67	146.284	825/1,851	559	824
Power reduction Synthesis + Map + P&R	39.87	901.51	941.38	122.429	825/1,836	588	763
Power optimization w/physical synth.	39.34	901.51	940.85	122.986	822/1,710	480	776
Power optimization w/o physical synth.	39.34	901.51	940.85	122.986	822/1,710	480	776
Area reduction w/physical synthesis	41.35	901.53	942.87	124.378	827/1,574	451	752
Area reduction w/o physical synthesis	44.43	901.56	945.99	120.758	827/1,636	476	739
Timing performance w/IOB packing	48.94	901.60	950.54	127.307	847/1,898	667	777
Timing performance with physical synthesis	53.75	901.65	955.41	115.754	846/1,837	650	779
Timing performance w/o IOB packing	45.40	901.57	946.97	139.412	875/1,903	642	804

TABLE 4.A EXPERIMENTAL RESULTS USING DIFFERENT IMPLEMENTATION STRATEGIES FOR THE VIRTEX-5 DEVICE

Following the methodology described in section 3.2.1 of the current work, the strategies achieving the best result for lower power consumption and fastest clock speed were optimized using the SmartXplorer tool and re-implemented using higher timing constraints. The lowest power consumption was achieved using the "Power reduction: Synthesis + Map + P&R" strategy, and the fastest implementation was achieved using the "Timing performance w/o IOB packing" strategy (disregarding of course the higher speed of the former strategy, as well as the speed of the original implementation). The custom strategy couldn't be optimized due the fact that the SmartXplorer tool doesn't allow for this using the Virtex-5 platform. Because of this, only the "Timing performance w/o IOB packing" was optimized prior re-implementation. Results for this procedure are shown below in Table 4.B.

Again, judging from these results, several aspects can be pointed out:

Speed/Power tradeoff: As the frequency of the design is pushed to the limits, it becomes apparent that lower power consumption can only be sustained at the cost of a decreased design speed. In this case it was investigated if the higher clocked implementation could achieve lower energy consumption by taking advantage of the shorter time in which the motion estimation takes place. The result was that, in the end, both implementations achieved similar energy consumption levels using the 200 Mhz. constraint. One has to keep in mind that even though the energy consumption is nearly identical for both cases, the fastest implementation still runs almost 20 Mhz. faster than the lower power consumption core, which means a potential advantage when processing video in real time.

Virtex-5	Dynamic Power (avg.) [mW]	Static Power [mW]	Total Power [mW]	Max processor freq. [Mhz]	Total Energy consumption @ max. freq. [uJ]
Design timing constraint goal: 200 Mhz					
Default	50.47	901.62	952.09	187.477	68.393
Lowest Power	40.12	901.51	941.63	185.977	61.477
Fastest	44.30	901.56	945.86	204.040	61.401
Design timing constraint goal: 250 Mhz					
Lowest Power	38.47	901.50	939.97	153.870	67.349
Fastest	47.72	901.59	949.32	220.459	61.492

TABLE 4.B EXPERIMENTAL RESULTS OBTAINED USING DIFFERENT TIMING CONSTRAINTS ON THE CLOCK NETWORK FOR THE VIRTEX-5 DEVICE

Decrease in energy consumption: When the fastest implementation is compared to the original core, a reduction of 10.22% in the total energy consumption is observed. It's worth noting as well that the original core fails to meet the timing constraint of 200 Mhz. and that the fast implementation is 8.83% faster than the original core.

Timing constraint impact on the implementation performance: As depicted by the results using the 250 Mhz. constraints, it can be counter-productive to set the timing constraint above the maximum limits the implementation can achieve. For the lowest power implementation the dynamic power consumption drops, but the total energy consumption is significantly higher when compared to the low power implementation using the 200 Mhz. constraint. Correlating this result, directly with the speed/power tradeoff mentioned before, this effect can be explained by the fact that, as the timing constraint is set higher, the implementation tools make use of more aggressive algorithms to try to meet the design constraints, however, this might result in a significant decrease of speed or increase in power consumption, since the tight requirements leave little room for other implementation tradeoffs.

4.1.2 Results using the Spartan-6 device and discussion

Following the procedure outlined in Section 3.2.2, the following results are reported on Tables 4.C and 4.D for the case of the Spartan-6 implementations.

Options	Dynamic Power (avg.) [mW]	Static Power [mW]	Total Power [mW]	Max processor frecuency [Mhz]	Slice registers/ LUT's	Occupied slices	Fully used LUT-FF pairs
Default	30.39	81.48	111.87	111.869	840/1,845	647	809
Power reduction: Synthesis	31.42	81.52	112.94	98.610	840/1,816	652	799
Power reduction: Map + P&R	29.58	81.46	111.04	109.123	814/1,809	621	788
Power reduction: Map + P&R + Clock gating	29.67	81.46	111.13	116.212	814/1,809	621	768
Power reduction: enhanced Map+ P&R	29.09	81.44	110.54	118.441	818/1,403	505	546
Power reduction: Area synthesis + enhanced Map + P&R	29.66	81.46	111.12	101.719	818/1,439	501	546
Power reduction: Speed Synthesis + enhanced Map +P&R	28.87	81.44	110.30	103.638	818/1,406	534	552
Timing performance w/IOB packing	30.78	81.50	112.27	78.272	918/1,558	671	679
Timing performance with physical synthesis	30.78	81.50	112.27	78.272	918/1,558	671	679
Timing performance w/o IOB packing	29.75	81.46	111.21	123.793	923/1,552	601	716
Power reduction and timing performance: Optimized	29.09	81.44	110.54	118.441	818/1,403	505	546

TABLE 4.C EXPERIMENTAL RESULTS USING DIFFERENT IMPLEMENTATION STRATEGIES FOR THE SPARTAN-6 DEVICE

Summarizing from the results shown in the above table, it's possible to outline the following:

Static Power consumption: As predicted before, the power consumption for the Spartan-6 platform is much lower than for the Virtex-5. As the Spartan-6 is a platform focused into reducing overall power consumption and is a newer device fabricated using a smaller process, this result was expected. The static power consumption also stayed at nearly identical levels as in the Virtex-5. These arguments further support the idea that static power consumption is dependent on the targeted device, and that implementation variations don't affect static power. Noteworthy is the fact that, part of this power reduction comes also at the price of a notably reduced speed for all of the implementations, when compared to the ones using the Virtex-5 device.

Dynamic power reduction: The reduction was not as impressive as in the case of the Virtex-5, comparing the lowest power implementation, which was "Power reduction: Speed Synthesis + enhanced Map +P&R", to the original core, a reduction of 5% of dynamic power was achieved. The number of fully used LUT-FF pairs doesn't correlate as directly to the implementations with the lowest power, due the fact that the slices in the Spartan-6 device have a different architecture than those in the Virtex-5, as stated early at the beginning of section 4.1. Still, there is a marked correlation between lower power consumption and fewer occupied slices to implement the design. Like in the case of the Virtex-5 experiments, the power consumption is positively impacted when setting the toolkit implementation efforts to the highest level.

Implementation options explored in a case-by-case basis: The following options that adversely affected the design power performance

- FSM encoding
- RAM style
- Auto BRAM packing
- Map slice logic

On the other hand, the following options positively impacted the design power performance:

- Combinatorial logic optimization
- LUT combining: Area
- Automatic clock gating

Judging from this, it is apparent that the processor implementations receives no benefit from synthesis or BRAM optimizations, and is enhanced by optimizations that aim at reducing the allocated resources for the processor design at the mapping and P&R stages of the implementation.

For this case, the lowest power implementation was the "Power reduction: Speed Synthesis + enhanced Map +P&R" strategy and the fastest implementation was the "Timing performance w/o IOB packing" strategy. Both were optimized using the power reduction and timing performance algorithms respectively, using the SmartXplorer tool. Results are shown in Table 4.D.

Decrease in energy consumption: The total energy consumption was reduced slightly, with a 2.49% in the best case for the 100 Mhz timing constraint and 3.96% for the 200 Mhz constraint. It's important to note that, for the latter case, all of the strategies failed by a large

margin to meet the constraint. This could imply that slower devices don't benefit as much from implementation variations, as on principle they are not able to run the core at the rated 200 Mhz. speed, there is little room, in terms of speed, to tradeoff in favor of decreasing power consumption. Much more evident in the case of the Spartan-6 was the fact that the lowest power implementation and the fastest one, achieved similar levels of total energy consumption, with the fastest one still having the advantage of faster data processing.

Spartan-6	Dynamic Power (avg.) [mW]	Static Power [mW]	Total Power [mW]	Max processor freq. [Mhz]	Total Energy consumption @ max. freq. [uJ]
Design timing constraint goal: 100 Mhz					
Default	30.39	81.48	111.87	111.869	26.139
Lowest Power	28.87	81.44	110.30	103.638	25.486
Fastest	29.11	81.44	110.56	125.960	24.686
Design timing constraint goal: 200 Mhz					
Default	30.39	81.48	111.87	134.134	25.303
Lowest Power	28.55	81.43	109.98	117.481	24.620
Fastest	30.09	81.47	111.56	165.317	24.299

TABLE 4.D EXPERIMENTAL RESULTS OBTAINED USING DIFFERENT TIMING CONSTRAINTS ON THE CLOCK NETWORK FOR THE SPARTAN-6 DEVICE

SmartXplorer optimizations: Using this tool, it was possible to discover that the processor implementation also benefits from register optimization options. The options used were "Register duplication" and "Equivalent register removal". These options are aimed to reduce the fan out and to eliminate redundant registers respectively and both increase the maximum operating frequency of the processor. The option "*Pack I/O registers into IOBs:* For Inputs and Outputs" also increased the maximum clock rate. This last option works by merging FF's into the Input/Output Blocks of the FPGA, effectively reducing data path delays.

Specific implementation strategy for the LiquidMotion processor: By collecting the implementation results reported before, a specific implementation strategy was created for the motion estimation processor. This strategy is aimed at reducing power consumption while at the same time increasing the core speed in order to get the lowest total energy consumption possible.

4.2 Results and discussion on the impact of different motion estimation algorithms on the processor power consumption

The results obtained from the experiments outlined in section 3.3 of the present work are reported in Table 4.E. The table reports the total power consumption of each motion

estimation search algorithm, as well as the processing time that each one takes to process the same data and the total energy consumption.

From the results shown in this Section, the following remarks can be made:

Impact on dynamic power: Dynamic energy consumption is notably impacted by the motion estimation algorithm executed by the processor. Using the FS algorithm as the gold standard for comparation, up to 31.8% of power reduction in this area can be achieved. The difference in dynamic power can be explained by the way in which motion estimation algorithm operates. For example, comparing the FS to the TSS algorithm, the number of search points the processor calculates to match one macroblock is much smaller for the TSS algorithm, meaning that the processor will take longer to process the macroblocks using FS, and many signals will toggle at a different rate due to various operations taking place inside the processor more or less frequently to calculate the motion vector. This is generally due to having to process a larger amount of date, but other issues such as recovering from more branch mispredictions, depending on the algorithm, take place as well (please refer to Chapter 2 of the current work for more details on the motion estimation algorithms)

Algorithm	Dynamic Power (avg.) [mW]	Static Power [mW]	Total Power [mW]	Processing time [us]	Total energy consumption [uJ]
FS [15x15]	69.32	901.82	971.14	5516.920	5357.701
HEXBS (diamond)	47.27	901.59	948.86	601.520	570.758
HEXBS (square)	48.77	901.60	950.38	693.920	659.487
Diamond	49.46	901.61	951.07	743.720	707.329
Logarithmic	55.27	901.67	956.94	884.120	846.049
TSS	58.30	901.70	960.00	927.720	890.611
FSS	57.13	901.69	958.82	960.720	921.157

TABLE 4.E EXPERIMENTAL RESULTS OBTAINED USING DIFFERENT TIMING CONSTRAINTS ON THE CLOCK NETWORK FOR THE SPARTAN-6 DEVICE

Total energy consumption: The difference in total energy consumption when using two different fast searching ME algorithms can be up to 38.03%. This is a notable difference and means that the selection of the ME algorithm is not a trivial part when targeting low-power applications using the LiquidMotion processor.

Search algorithm tradeoffs: Even though this topic is outside the scope of the current project, when choosing a ME search algorithm to execute on the processor, one has to bear in mind that the choice will not only affect the power consumption, but also the quality of the video processing performed. This means that different algorithms have different distortion rates. In general, an algorithm that uses more search points has lower distortion rates, which translates

into better video quality. Distortion rate is calculated as the value of the matching difference between the matched macroblocks (i.e. the SAD in the case of the LiquidMotion processor). Further evaluation of the performance of motion estimation algorithms in terms of distortion and average search points can be found in [27] and related works.

4.3 Results and discussion on the impact of a shallow pipeline design to reduce the processor power consumption

The results of the implementation of the shallow pipeline in terms of power consumption are shown below in Table 4.F. Description of the methodology of the experiment can be found in section 3.4 of the current work.

Virtex-5	Dynamic Power (avg.) [mW]	Static Power [mW]	Total Power [mW]	Max processor freq. [Mhz]	FF's	Processing time [us]
Design timing constraint goal: 100 Mhz						
Original Core	48.77	901.60	950.38	141.383	825	693920
Shallow Pipeline	44.10	901.55	945.65	133.565	637	689320

TABLE 4.F EXPERIMENTAL RESULTS OBTAINED USING THE PROPOSED SHALLOW PIPELINE DESIGN FOR THE VIRTEX-5 DEVICE

Reduced dynamic power consumption: As a result of fewer registers contained in the pipeline, fewer FF's were allocated for the processor implementation, which translated into reduced waste of dynamic power. Power reduction accounted for 9.57% corresponding to 188 total registers that were eliminated.

Slightly reduced maximum frequency: As expected from the review presented in Chapter 1 of the present work, the processor maximum frequency was decreased. By decreasing the number of pipeline stages, the workload in the merged stages was increased, meaning that data has to go across a longer processing data path before reaching the next register, meaning in turn that the whole processor has to be clocked at a slower rate in order for it to have enough time to process the data from stage to stage. This is a well-known issue in computer architecture.

Decreased processing time: Also, as expected form the review of the processor architecture; processing time is decreased because of smaller penalties due to branch mispredictions. The shallow pipeline means that it takes less cycles for the processor to recover from these issues. Even though the change seems to be fairly small, it's important to point out that this also reduces total energy consumption at the same time and that the number of macroblocks used for the testbench is fairly small when compared to a full video processing application which can benefit even more from this decreased processing time

Steady static power: The creation of a shallow pipeline doesn't impact the static power consumption of the targeted FPGA. This behavior was anticipated in the review of the adaptive pipeline design in Chapter 1.

Increase of glitches in the design: Despite the fact that a significant amount of registers were removed from the design, the appearance of glitches was not enough to overshadow the reduction in power consumption achieved by taking away those registers in the first place. This means that, at least for the shallow pipeline created for this work, the effect of glitches in the design was low enough to be disregarded. It is possible that if more stages were to be merged, a point where no further power reduction could be made, despite the removal of a larger number of registers due to the propagation of glitches inside the processor architecture. Please refer to Chapter 1 of this work for further information on glitch propagation.

Implementation problems: Even though it was expected that some registers required a higher coding effort to be removed, due to the inherent complexity of the processor design, it also became apparent that some registers just couldn't be removed without affecting the processor functionality. The Concatenate unit was merged successfully, but in the process of removing the registers contained in the Sad Selector Unit it was discovered that, despite careful VHDL coding, latches where still being generated when implementing the design. While the processor description was being debugged to found the cause of this behavior, it was discovered that storage elements are, in fact, required for the functionality of this stage to be adequate. This was causing the implementation tools to synthesize latches (even when registers were removed) because an internal feedback loop existed inside the stage, and data storage elements were required in the loop, due to the simple fact that a wire just can't store any data. These inherent data dependencies exist because of the processor functionality and not because of its architecture. Concluding, it was apparent that some stages couldn't be merged, at least not without a very extensive change to the processor design. Using the implementation tools, it was found that an additional 117 registers could be taken away by merging the Sad Selector Unit, however, doing this also generated 40 latches on the design (neither the original core nor the shallow pipeline shown in Table 4.F contain latches). This led to increased power consumption in the shallow pipeline design, and the approach to merge the mentioned stage, was ultimately discarded.

4.4 Overall remarks and conclusions about the power reduction features presented in this work.

Concluding from the aspects outlined in the previous sections, it's possible to make some remarks based on the nature of each presented power reduction feature and the results achieved by each one.

Use of the implementation toolkit options: The results shown in the current work point out at the fact that a significant percentage (more than 10% for the case of original target device) of total energy consumption can be achieved by using a custom configuration for the implementation tools. As a result form this work, a custom configuration was created specifically to optimize the resulting implementation of the LiquidMotion processor, using the Xilinx ISE tools. The details of this configuration are shown in Table 3.4. Processor maximum

frequency is also increased (more than 8% for the Virtex-5 device) meaning an overall increase in processor performance. The apparent tradeoff of this benefit is the increased implementation runtime. Given the level of increased performance obtained, it's safe to say that, despite this small drawback, the investment of increased computation time in the implementation phase is well worth it. An additional benefit of using this feature is that the end-user is not required to know the details about the processor architecture or VHDL coding experience to make use of it. For this reason it is considered the most user-friendly power reduction feature explored in this work.

Use of different motion estimation algorithms: The choice between two different motion estimation algorithms can mean an effective decrease in total energy consumption of more than 38%. Other considerations, other than power consumption, should be taken into consideration when choosing an ME algorithm, especially when high video quality is required. Results from works aimed at exploring the performance of fast ME, like [26], and [27] point out at the fact that, most of the time, faster algorithms result in larger video distortion rates, but it also depends on the content of the video frames being decoded. Given this, the choice for ME algorithms with an acceptable distortion for high quality video might be reduced to one or two, but if the requirements are not as strict, then, switching to a less power hungry algorithm can save a notable amount of energy. It's also possible to conclude that the selection of the ME algorithm used, requires certain degree of knowledge by the end-user. As such, it's possible that the availability of this feature is restricted, at least to a certain degree, to users with said capabilities.

Impact of a shallow pipeline implementation on the LiquidMotion processor: The shallow pipeline stands as an effective way to reduce the processor dynamic power consumption, with the added value of reducing the macroblock processing time without running a different ME algorithm. On top of this, the total implementation size is reduced thanks to the fewer number of registers allocated for the processor, Given the nature of the processor, it could be possible to exploit this reduced size to implement additional pipelines at a lower resource cost. This technique, though clearly effective at achieving its goal, is considered the most restricted one due to the fact that it requires an adequate comprehension of the processor architecture and experience with designs described using VHDL code.

Concurrent use of the power saving features: Though not explored in this work, it's safe to argue that the concurrent use of the three techniques presented in here can achieve a large impact in the dynamic power consumption. It can be argued that one technique reinforces each other. For example, assume that the shallow pipeline has been implemented, as well as one of the lowest power ME algorithms. In this case the ME algorithm will benefit from the shorter pipeline, due to the shorter branch penalties, and will also benefit by the shorter processing time of the shallow pipeline, resulting in larger power reductions than the ones expected from adding up the reductions achieved by each technique.

CHAPTER 5: WORK EVALUATION AND FUTURE WORK TOWARDS REDUCING THE PROCESSOR POWER CONSUMPTION

After having analyzed the results of the experiments proposed in the current work, a critical evaluation of the current project is presented in this chapter, as well as outlining the aspects that could be further explored in future works.

5.1 Evaluation of the current work

Overall, the work presented in here offers a practical approach to tackle one of the main issues of power-aware computing and embedded applications. The LiquidMotion estimation processor, which was the target for all the power reduction features explored in here, represents a powerful and flexible platform for video processing, with its re-configurability being of its main advantages. Having said this, it's important to note that one of the main reasons why FPGA-based implementations haven't been widely adopted for commercial scale applications, is their higher power consumption when compared to ASIC's. In doing this project, a contribution was made towards and effective approach to increase the value of an already existing platform, by offering a practical way to reduce its power consumption.

The research review presented in Chapter 1, provides a solid and comprehensive study about FPGA power reduction techniques, a topic that, although it has been explored a number of times, needs to see much more development in the following years. Also the critical analysis of the power reduction techniques and the study of the processor architecture, on Chapter 1 and 2 respectively, provide a valuable starting point and insight for anyone aiming to implement these techniques or further enhance the work done in this project.

The power consumption estimation methodology used in this project is thought to be a reliable way to accurately estimate the power consumption resulting from the experiments, being second only, in terms of accuracy, to measuring the actual physical implementation on a FPGA board. This provides a high level of confidence to the results of the experiments carried out in this project.

The first practical contribution of the project is and optimized implementation strategy tailored for the LiquidMotion processor, which allows for decreased power consumption and increased clock frequency. The ISE implementation tool was chosen, because past works on this processor have targeted Xilinx FPGA's and these works have provided a reference of knowledge and data regarding the processor.

Another contribution is the practical dataset obtained as a result from testing the power consumption of different ME algorithms. This dataset provides a valuable reference when the processor is being implemented on a platform with limited power resources, like in embedded applications.

Finally, a contribution towards the direct development of the LiquidMotion motion processor was made in the form of the shallow pipeline design proposed here. The new design can be used standalone or interchangeably with the original pipeline design, and given the re-configurability features of the processor, it could be switched to from full to low power mode whenever the current video application is not demanding the totality of the processor resources by switching the pipeline design. The shallow pipeline opens the possibility to

explore further architecture changes that could potentially reduce the power consumption even further.

5.2 Future work

There are a number of aspects that could be further explored in subsequent or related works. First of all, for the case of the processor implementation variations using the toolset, a number of new FPGA devices, completely devoted to reduce power consumption as in the of the low power version of the Spartan-6 device, could be used as target for implementations, to see how these new devices can handle the processor energy consumption. Also, smaller devices (in terms of logic resources) could be explored. This approach could be beneficial to investigate a low-cost and low-power implementation of the processor for commercial applications.

The motion estimation algorithms explored in here were limited to non-adaptive ones. This was done because, at the time, there is work still ongoing on the processor, regarding the performance and functionality of the adaptive hardware to support this type of algorithms. Considering the positive results achieved by fast ME algorithms using fewer search points, it would be extremely interesting to explore the power consumption of adaptive ME algorithms on the LiquidMotion processor.

Regarding the architecture modifications in the processor description, future work could either focus on enhance the shallow pipeline design presented in the current work. Alternatively, the implementation of the DVS technique would be an interesting approach, since this technique not only reduces the dynamic, but also the static power consumption.

In conclusion, there is still a variety of possibilities that can be explored to further improve the performance of the LiquidMotion processor in terms of power consumption.

REFERENCES

[1] Kalaycioglu, C.; Ulusel, O.C.; Hamzaoglu, I.; , "Low power techniques for Motion Estimation hardware," *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on* , vol., no., pp.180-185, Aug. 31 2009-Sept. 2 2009

[2] Yan Zhang; Roivainen, J.; Mammela, A.; , "Clock-Gating in FPGAs: A Novel and Comparative Evaluation," *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on* , vol., no., pp.`584-590, 0-0 0

[3] Efthymiou, A.; Garside, J.D.; , "Adaptive pipeline structures for speculation control," *Asynchronous Circuits and Systems, 2003. Proceedings. Ninth International Symposium on* , vol., no., pp. 46- 55, 12-15 May 2003

[4] Koppanalil, J.; Ramrakhyani, P.; Desai, S.; Vaidyanathan, A.; Rotenberg, E.;, "A case for dynamic pipeline scaling", *Compilers, architecture, and synthesis for embedded systems* (CASES '02). *Proceedings of the 2002 international conference on,* ACM, New York, NY, USA, 1-8. 2002

[5] Efthymiou, A.; Garside, J.D.; , "Adaptive pipeline depth control for processor power-management," *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on* , vol., no., pp. 454- 457, 2002

[6] Bard, S.; Rafla, N.I.; , "Reducing power consumption in FPGAs by pipelining," *Circuits and Systems, 2008. MWSCAS 2008. 51st Midwest Symposium on* , vol., no., pp.173-176, 10-13 Aug. 2008

[7] Wilton, S.J.E.; Ang, S.; Luk, W.; "The Impact of Pipelining on Energy per Operation in Field-Programmable Gate Arrays," *Int. Conference on FPL,* pp. 719-728, August 2004.

[8] Lamoureux, J.; Lemieux, G.; Wilton, S.; , "GlitchLess: Dynamic Power Minimization in FPGAs Through Edge Alignment and Glitch Filtering," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* , vol.16, no.11, pp.1521-1534, Nov. 2008

[9] Huda, S.; Mallick, M.; Anderson, J.H.; , "Clock gating architectures for FPGA power reduction," *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on* , vol., no., pp.112-118, Aug. 31 2009-Sept. 2 2009

[10] Shum, W.; Anderson, J.H.; , "FPGA glitch power analysis and reduction," *Low Power Electronics and Design (ISLPED) 2011 International Symposium on* , vol., no., pp.27-32, 1-3 Aug. 2011

[11] What's New in Xilinx ISE Design Suite 12, 2010. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_2/whatsnew.htm [Accessed: 11 May. 2012].

[12] Xilinx, *Power Consumption at 40 and 45 nm*, White Paper: Spartan-6 and Virtex-6 Devices WP298 (v1.0), Apr. 2009. [Online]. Available:

http://www.xilinx.com/support/documentation/white_papers/wp298.pdf [Accessed: 11 May. 2012].

[13] XPower Analyzer. [Online]. Available: http://www.xilinx.com/products/design_tools/logic_design/verification/xpower_an.htm [Accessed: 11 May. 2012].

[14] Xilinx, *Xilinx Power Tools Tutorial*, Spartan-6 and Virtex-6 FPGA's UG733 (v1.0), Mar. 2010. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ug733.pdf

[15] Nunez-Yanez, J.L.; Nabina, A.; Hung, E.; Vafiadis, G.; , "Cogeneration of Fast Motion Estimation Processors and Algorithms for Advanced Video Coding," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* , vol.20, no.3, pp.437-448, March 2012

[16] Xilinx, *XPower Tutorial*, FPGA Design XPower (v1.3), Jul. 2002. [Online]. Available: ftp://ftp.xilinx.com/pub/documentation/tutorials/xpowerfpgatutorial.pdf

[17] Nunez-Yanez, J.L.; Nabina, A.; Hung, E.; Vafiadis, G.; , "Cogeneration of Fast Motion Estimation Processors and Algorithms for Advanced Video Coding," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* , vol.20, no.3, pp.437-448, March 2012

[18] Nakai, M.; Akui, S.; Seno, K.; Meguro, T.; Seki, T.; Kondo, T.; Hashiguchi, A.; Kawahara, H.; Kumano, K.; Shimura, M.; , "Dynamic voltage and frequency management for a low-power embedded microprocessor," *Solid-State Circuits, IEEE Journal of* , vol.40, no.1, pp. 28- 35, Jan. 2005

[19] Das, S.; Roberts, D.; Seokwoo Lee; Pant, S.; Blaauw, D.; Austin, T.; Flautner, K.; Mudge, T.; , "A self-tuning DVS processor using delay-error detection and correction," *Solid-State Circuits, IEEE Journal of* , vol.41, no.4, pp. 792- 804, April 2006

[20] Nunez-Yanez, J.L.; Chouliaras, V.; Gaisler, J.; , "Dynamic Voltage Scaling in a FPGA-Based System-on-Chip," *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on* , vol., no., pp.459-462, 27-29 Aug. 2007

[21] Chow, C.T.; Tsui, L.S.M.; Leong, P.H.W.; Luk, W.; Wilton, S.J.E.; , "Dynamic voltage scaling for commercial FPGAs," *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on* , vol., no., pp.173-180, 11-14 Dec. 2005

[22] Nunez-Yanez, J.L.; Hung, E.; Chouliaras, V.; , "A configurable and programmable motion estimation processor for the H.264 video codec," *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on* , vol., no., pp.149-154, 8-10 Sept. 2008

[23] Nunez-Yanez, J.L.; Spiteri, T.; Vafiadis, G.; , "Multi-standard reconfigurable motion estimation processor for hybrid video codecs," *Computers & Digital Techniques, IET* , vol.5, no.2, pp.73-85, March 2011

[24] OpenCores, *Motion Estimation Processor*, Overview. [Online]. Available: http://opencores.org/project,motion_estimation_processor [Accessed: 11 May. 2012].

[25] OpenCores, *The LiquidMotion Programmable/Configurable Motion Estimation Instruction Set Processor*, datasheet. [Online]. Available: http://opencores.org/download,motion_estimation_processor [Accessed: 11 May. 2012].

[26] Ce Zhu; Xiao Lin; Lap-Pui Chau; , "Hexagon-based search pattern for fast block motion estimation," *Circuits and Systems for Video Technology, IEEE Transactions on* , vol.12, no.5, pp.349-355, May 2002

[27] Jo Yew Tham; Ranganath, S.; Ranganath, M.; Kassim, A.A.; , "A novel unrestricted center-biased diamond search algorithm for block motion estimation," *Circuits and Systems for Video Technology, IEEE Transactions on* , vol.8, no.4, pp.369-377, Aug 1998

[28] Aroh Barjatya, "*Block Matching Algorithms For Motion Estimation*," Student Member, IEEE, DIP 6620 Spring 2004 Final Project Paper

[29] Becker, J.; Hubner, M.; Hettich, G.; Constapel, R.; Eisenmann, J.; Luka, J.; , "Dynamic and Partial FPGA Exploitation," *Proceedings of the IEEE* , vol.95, no.2, pp.438-452, Feb. 2007

Appendix I: Motion estimation algorithms source code

//Diamond search

```
//8 steps maximum
fpi = 8;
Pattern(largediamond)
{
check(2,0)
check(1,1)
check(0,2)
check(-1,1)
check(-2,0)
check(-1,-1)
check(0,-2)
check(1,-1)
}
Pattern(small)
{
check(0,1)
check(0,-1)
check(1,0)
check(-1,0)
}
check(0,0);
update;
//large d search
for(loop = 1 to fpi step 1)
{
check(largediamond);
#if( WINID == 0 )
#break;
}
//small diamond
check(small);
```

//HEXBS with square refinement

```
//Original code modified with permission from Jose Nunez-Yanez
//8 Steps maximum
fpi = 8;
Pattern(square)
{
check(0,1)
check(0,-1)
check(1,0)
check(-1,0)
check(-1,-1)
check(-1,1)
check(1,-1)
check(1,1)
}
Pattern(hexbs)
{
check(2,0)
check(-2,0)
check(1,2)
check(-1,2)
check(1,-2)
check(-1,-2)
}
check(0,0);
update;
//hexagon search
for(loop = 1 to fpi step 1)
{
check(hexbs);
#if( WINID == 0 )
#break;
```

/// 4 Step Search

```
S = 2; // Step size
for(i = 0 to 2 step 1)
{
check(0, 0);
check(S, 0);
check(S, S);
check(0, S);
check(-S, S);
check(-S, 0);
check(-S, -S);
check(0, -S);
check(S, -S);
update;
#if( WINID == 0 )
#break;
}
S = S / 2;
check(0, 0);
check(S, 0);
check(S, S);
check(0, S);
check(-S, S);
check(-S, 0);
check(-S, -S);
check(0, -S);
check(S, -S);
update;
```

/// Three Step Search

```
S = 4; // Step size
check(0, 0);
check(S, 0);
check(S, S);
check(0, S);
check(-S, S);
check(-S, 0);
check(-S, -S);
check(0, -S);
check(S, -S);
update;
do {
S = S / 2;
check(0, 0);
check(S, 0);
check(S, S);
check(0, S);
check(-S, S);
check(-S, 0);
check(-S, -S);
check(0, -S);
check(S, -S);
update;
} while( S > 1 );
```

```
//HEXBS search with diamond refinement
//8 steps maximum
fpi = 8;
Pattern(hexbs)
{
check(2,0)
check(-2,0)
check(1,2)
check(-1,2)
check(1,-2)
check(-1,-2)
}
Pattern(small)
{
check(0,1)
check(0,-1)
check(1,0)
check(-1,0)
}
check(0,0);
update;
//hexagon search
for(loop = 1 to fpi step 1)
{
check(hexbs);
#if( WINID == 0 )
#break;
}
//square refinement
check(small);

}
//square refinement
check(square);
```

Appendix II: VHDL essential source code of the shallow pipeline implementation

```
--ORIGINAL CODE
-----------------------------------------
--  entity      = concatenate
--  version     = 1.0
--  last update  = 1/08/06
--  author      = Jose Nunez
-----------------------------------------
--MODIFIED CODE
-----------------------------------------
--  NOTE
--  Part of the shallow pipeline
--  implementation
--  last update-> 15/09/12
--  author-> Alejandro Vaca
-----------------------------------------

-- sequential part replaced with simple signal assignment
-- registers: data, valid, pipe_din, pipe_din2,
-- pipe_addr, pipe_enable_hp_inter,pipe_enable
-- were removed
-- no latches are generated on implementation
-- code works fully at implementation level

-- FUNCTION
-- this unit makes sure that 8 valid pixels are assemble
depending on byte address

library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_unsigned."+";

entity concatenate64 is
            port(
            addr : in std_logic_vector(2 downto 0);
            clk : in std_logic;
            clear : in std_logic;
            reset : in std_logic;
            din : in std_logic_vector(63 downto 0);
            din2 : in std_logic_vector(63 downto 0);
            dout : out std_logic_vector(63 downto 0);
            enable : in std_logic;
            enable_hp_inter : in std_logic; -- working in
interpolation mode
    quick_valid : out std_logic; --as valid but one cycle earlier
            valid : out std_logic);  -- indicates when 64 valid
bits are in the output
end concatenate64;


architecture behav of concatenate64 is

type register_type is record
            data : std_logic_vector(63 downto 0);
    valid : std_logic; -- bytes are valid
    pipe_din : std_logic_vector(63 downto 0);
    pipe_din2 : std_logic_vector(63 downto 0);
    pipe_addr : std_logic_vector(2 downto 0);
    pipe_enable_hp_inter : std_logic;
    pipe_enable : std_logic;
end record;

signal r,r_in : register_type;
signal din_temp : std_logic_vector(63 downto 0);

begin
```

```
r_in.pipe_din <= din;
r_in.pipe_din2 <= din2;
r_in.pipe_addr <= addr;
r_in.pipe_enable_hp_inter <= enable_hp_inter;
r_in.pipe_enable <= enable;
r_in.valid <= r.pipe_enable;



valid <= '1' when r.valid = '1' else '0';
quick_valid <= '1' when r.pipe_enable= '1' else '0';


shift_data : process(r)


begin


  if (r.pipe_enable_hp_inter = '0') then -- when interpolating the good data is
at the beginning

            case r.pipe_addr is

              when "000" =>
                        din_temp <= r.pipe_din;
              when "001" =>
                        din_temp <= r.pipe_din(55 downto 0)&
r.pipe_din2(63 downto 56);
              when "010" =>
                        din_temp <= r.pipe_din(47 downto 0) &
r.pipe_din2(63 downto 48);
              when "011" =>
                        din_temp <= r.pipe_din(39 downto 0) &
r.pipe_din2(63 downto 40);
              when "100" =>
                        din_temp <= r.pipe_din(31 downto 0)&
r.pipe_din2(63 downto 32);
              when "101" =>
                        din_temp <= r.pipe_din(23 downto 0)&
r.pipe_din2(63 downto 24);
              when "110" =>
                        din_temp <= r.pipe_din(15 downto 0)&
r.pipe_din2(63 downto 16);
              when "111" =>
                        din_temp <= r.pipe_din(7 downto 0)& r.pipe_din2(63
downto 8);
    when others => null;
  end case;
  else
case r.pipe_addr is
              when "000" =>
                        din_temp <= r.pipe_din2;
              when "001" =>
                        din_temp <=  r.pipe_din2(55 downto 0) & r.pipe_din(63
downto 56);
              when "010" =>
                din_temp <= r.pipe_din2(47 downto 0) & r.pipe_din(63
downto 48);
              when "011" =>
                din_temp <= r.pipe_din2(39 downto 0) & r.pipe_din(63
downto 40);
              when "100" =>
                din_temp <= r.pipe_din2(31 downto 0) & r.pipe_din(63
downto 32);
```

```
                when "101" =>
                        din_temp <= r.pipe_din2(23 downto 0)
& r.pipe_din(63 downto 24);
                when "110" =>
                        din_temp <= r.pipe_din2(15 downto 0)
& r.pipe_din(63 downto 16);
                when "111" =>
                        din_temp <= r.pipe_din2(7 downto 0) &
r.pipe_din(63 downto 8);
   when others => null;
   end case;
   end if;


end process shift_data;

r_in.data <= din_temp;

dout <= r.data;


--removed sequential part and registers

                        r <= r_in;
--valid <= r.valid;

end behav; -- end of architecture
```