Optimized ASIP Architecture for Compressed BWT-Indexed Search in Bioinformatics Applications

Nuno Sebastio, Paulo Flores and Nuno Roma
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
Rua Alves Redol, 9, 1000-029 Lisboa - Portugal
Email: {nuno.sebastiao, paulo.flores, nuno.roma}@inesc-id.pt

***Abstract*—Compressed indexes are adopted by a vast set of bioinformatics applications that deal with extremely large datasets, mainly due to the inherently high memory requirements of uncompressed alternatives. However, the additional computational overhead that is imposed by the usage of such indexes makes them harder to implement in embedded computational platforms, such as biochips, with strict processing and power restrictions. Furthermore, compressed indexes are often characterized by a significant usage of bit-level operations, some of which are not commonly available on General Purpose Processors (GPPs). To circumvent this limitation, an Application-Specific Instruction-set Processor (ASIP) architecture is proposed to accelerate the processing of biological sequences (e.g., alignment, mapping, etc.) using compressed full-text indexes based on the Burrows-Wheeler Transform (BWT). The proposed processor was built over a RISC micro-architecture and extends the Xilinx MicroBlaze ISA with additional bit-level operations, especially tailored for compressed indexes. When used to perform search operations over the considered compressed index, the proposed architecture provides a reduction of the number of required instructions by about one half. Furthermore, when prototyped on a Xilinx Virtex-7 FPGA, the ASIP proved to offer an overall speedup between 3.1x and 4.5x for the execution of a single threaded operation. To ensure a further processing scalability, the proposed ASIP was designed in order to be easily used as the basic processing unit of multi-core systems, especially tuned for the parallel processing of massive datasets of biological reads.**

***Keywords*—Application-Specific Instruction-Set Processor, Compressed text indexes, DNA alignment, Heuristic algorithms**

I. Introduction

In bioinformatics, most high performance sequence alignment tools make use of heuristic algorithms that rely on fast exact string matching operations [1], [2]. These tools typically implement the matching operation using full-text indexes that present extremely high space requirements, which may even be prohibitive if not adequately designed (20x the size of the original text [3]). Such space amount not only poses difficult challenges for accommodating it in persistent memory but it also cannot be completely loaded into the main memory of a typical embedded system, thus imposing a high penalty on the index access time.

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under project HELIX (Ref. PTDC/EEA-ELC/113999/2009), project THREadS (Ref. PTDC/EEA-ELC/117329/2010) and project PEst-OE/EEI/LA0021/2013.

To overcome these space limitations, the usage of compression techniques to store the index is an important requirement when dealing with very large reference *texts*. One of such compressed indexes, with particular interest in bioinformatics [1], [2], [4], [5], is the FM-index [6], based on the Burrows-Wheeler Transform (BWT), and which can make use of several compression techniques, such as move-to-front transform, run-length encoding, variable-length prefix coding and wavelet trees. However, the added complexity that is imposed by the adopted compression techniques often requires some additional processing time to access the data in the compressed format.

Most of the research efforts that have been devised to optimize index based applications have focused on increasing the performance of the *compression* phase, since it is usually the most critical and time consuming phase in non-bioinformatics applications. In this scope, [7] and [8] describe completely custom hardware solutions to accelerate the widely known BWT-based *bzip* compression utility. However, none of these solutions can be used during the decompression, which becomes critical when the number of searches is very large and must be performed with a small execution time.

On the contrary, the *decompression* phase is widely used in the scope of a vast number of bioinformatics applications. In this particular domain, the very high computational workload that is involved is mainly due to the massive datasets that are commonly dealt in this field (e.g. when mapping an entire genome, the amount of characters to be searched for in a single mapping operation can be over 10 times the amount of characters in the index). Therefore, efficient decompression and search over the index structure should also be attained, as the amount of search operations is extremely high.

In [9], a hardware implementation of the search operation over the FM-index was proposed. Nevertheless, despite being an extremely efficient implementation, it uses the FM-index in an uncompressed format, thus not being targeted to applications that require the more space efficient versions of the index. In fact, due to the extra processing cost to decompress the data, many software tools implemented in common General Purpose Processors (GPPs) [1], [2], [4], [5] are unable to take advantage of a compressed index, commonly resorting to sampling of the index data, in order to reduce the space cost. Nevertheless, this technique also leads to an increase of the processing time as it becomes necessary to reconstitute the index data at runtime.

Accessing information in compressed indexes typically makes use of bit-level operations, for which common GPPs are not adequately tuned, thus presenting a high processing time. To circumvent this limitation, the usage of specialized structures that increase the execution efficiency of these operations is highly recommended. Furthermore, the advent of self-contained, portable and battery powered biological analysis systems, with strict power and energy constraints, requires the design of novel processing systems with both reduced power consumption and a high energy efficiency. These constraints may be met by using efficient structures that range from completely custom and non-programmable dedicated solutions with reduced flexibility [9], to specialized but still highly programmable solutions, based on Application Specific Instruction-set Processors (ASIPs).

In this context, an innovative ASIP architecture that is capable of reducing the processing time of massive string search operations based on compressed index structures is proposed. This ASIP was specifically evaluated in the bioinformatics application domain, which commonly deals with very large datasets. To attain this objective, the proposed ASIP makes use of a new instruction-set extension that includes dedicated instructions for bit-level operations. Furthermore, the proposed ASIP architecture is based on an efficient RISC micro-architecture that includes a comprehensive set of highly optimized functional units that were specifically developed to implement the new instructions. Moreover, to ensure a high degree of processing scalability, the design of the proposed ASIP considered the possibility to use it as the basic processing unit in a multi-core system, especially tuned for the parallel processing of massive datasets of biological short reads.

II. Compressed Indexes

Among the several index structures that have been proposed, a specific and prominent case-study will be particularly considered in this paper: the FM-index [6]. This index has been recognized as one of the most efficient in the bioinformatics domain and it has been used in several publicly available software tools. Nevertheless, the proposed ASIP architecture can equally be applied to accelerate the search based on other commonly used structures that benefit from similar compression methods, with entirely equivalent results.

The FM-index is based on the BWT [10] of the original text and on a specific directory information ($Occ(c, i)$) that allows searching a given *pattern* over the *text* in linear time. The BWT alone can be used to reconstruct the original text and to perform the search operations. However, this requires a large computational effort, due to the last-to-first mapping method [10] used to navigate through the BWT. To reduce this computational complexity, the Occ table (part of the directory) contains information that allows to easily jump to the *next* position in the BWT. For each position i of the BWT, this table holds integer values that represent the number of occurrences of each alphabet symbol c in the substring $\mathrm{BWT}(S)[0..i]$ (additional details can be found in [6]).

An example of the index data structure is presented in Fig. 1, for the reference *text* string $\mathrm{BWT}(S) = \text{“ACGGTTAT}''$ (the terminator '\$' is concatenated to the end of the *text*). Fig. 1(a) represents the BWT of this text ($\mathrm{BWT}(S)$) while Fig. 1(b) illustrates the $Occ(c, i)$ table. Considering a Deoxyribonucleic Acid (DNA) sequence with 4 alphabet symbols, if the Occ table is stored in an uncompressed form, it may require as much as 20 times more space than the BWT itself (assuming that a character occupies 8 bits and an integer 32 bits).

To reduce the space occupied by the index data, especially the Occ table, several methods can be employed. A tradeoff between time and space complexity must always be considered when choosing a compression method. Among the various possibilities, a particularly interesting method that allows for a medium level of compression uses bit vectors to directly represent the data in the Occ table.

Accordingly, the bit-vectors of $Occ(c, i)$ ($BV\_Occ(c, 0..i)$) represent the positions in the BWT$[0..i]$ prefix where the corresponding symbol c occurs. As an example, if the $BV\_Occ(c, j)$ bit is set, then symbol c occurs at BWT$[j]$. Hence, $Occ(c, j) = Occ(c, j - 1) + 1$. More generally, $Occ(c, j)$ is calculated by counting the number of set bits in the bit-vector from that symbol up to position j. In terms of computational requirements, the use of this compression strategy requires the efficient calculation of the number of set bits in a given vector.

Hence, if one considers the existence of an auxiliary function, BitCount(b), that returns the number of bits that are set in the input bit-vector b, it is possible to calculate the occurrences of c in BWT[0..j] by simply computing $Occ(c, j) = \text{BitCount}(BV\_Occ(c, 0..j))$. However, the application of this BitCount function to compute $BV\_Occ(c, 0..j)$ may require accessing a large number of memory locations (those covering the range from position 0 to position j of the bit-vector). To circumvent this limitation, the BitCount function should operate at a granularity corresponding to the size of the processor word (e.g. 32 bits) and the used index should also sample the values of $Occ(c, i)$ at regular intervals

BWT(T) = T\$TACGATG

(a) BWT of the reference text S.

BWT(T)	T	\$	T	A	C	G	A	T	G
$Occ(c, i)$	0	1	2	3	4	5	6	7	8
\$	0	1	1	1	1	1	1	1	1
A	0	0	0	1	1	1	2	2	2
C	0	0	0	0	1	1	1	1	1
G	0	0	0	0	0	1	1	1	2
T	1	1	2	2	2	2	2	3	3

(b) Table $Occ(c, i)$.

Figure 1. Example of a FM-index data structure (BWT and directory) for the reference text S=“ACGGTTAT”.

$BV\_Occ(c,i)$	T	$	T	A	C	G	A	T	G
$	0	1	0	0	0	0	0	0	0
A	0	0	0	1	0	0	1	0	0
C	0	0	0	0	1	0	0	0	0
G	0	0	0	0	0	1	0	0	1
T	1	0	1	0	0	0	0	1	0

$Occ(c,k*l)$									
$	0				1				1
A	0				1				2
C	0				1				1
G	0				0				2
T	1				2				3

Figure 2. Example of the used FM-index compression (BWT and directory) for the text S="ACGGTTAT" using a subsampling interval of 4 positions for the Occ (note that BV_Occ is represented at the bit level).

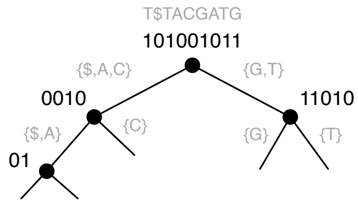


Figure 3. Wavelet tree representation of the BWT(T)=T$TACGATG.

of size k. Therefore, the desired computation simply becomes $Occ(c,j) = Occ(c,j/k) + \text{BitCount}(BV\_Occ(c,j/k+1..j))$. Fig. 2 illustrates this data structure on a compressed index using the described strategy.

Another efficient compression method makes use of wavelet trees to represent the data of the BWT. A wavelet tree is a specialized data structure that to represent strings, by recursively partitioning the alphabet and storing, at each node, the locations where each character of the alphabet subset occurs. Just like the previously described Occ directory, bit vectors are also widely used in this data structure. An example of this tree is given in Fig. 3 for the BWT string "T$TACGATG". At each node of this tree, the alphabet is divided into the right and left subsets and the bitvector at the corresponding node represents which characters belong to the right or left subsets. The information that is stored only pertains the bitvectors (the remaining data that is shown in Fig. 3 is only for illustrative purposes). The data in this structure presents some similarities with the data stored in the Occ table, thus allowing to calculate the Occ values. In this case, determining $Occ(c,i)$ requires to navigate the tree to the lowest node that represents the character c and count the number of bits in such bitvector that correspond to character c (0's or 1's). However, to reach such node it is also necessary to previously count the number of bits in the intermediate levels of the tree. Therefore, to obtain any $Occ(c,i)$ value it is necessary to perform $log\sigma$ bit count operations, with σ representing the alphabet size. With this data structure, not only is the data of the Occ table compressed, but also the BWT text itself, resulting in a much higher compression ratio. To extract the information from this representation, it is also required to make an extensive use of bit counting operations (such as the BitCount(b) function) to navigate the tree and obtain the data, whether it is the BWT string itself or the $Occ(c,i)$ data.

Symbol encoding table	
$	11110
A	110
C	1110
G	10
T	0

BWT(T) = $TACGATG

Encoded BWT(T)
0111100110111010110010

Figure 4. Example of a variable-length prefix encoding of the BWT(T) string.

Besides the previously described methods, it is also possible to use variable-length prefix codes to store the data of an index in a compressed form. This type of code allows to represent symbols with a reduced and variable bit-length without requiring a special marker that signals the separation between consecutive symbols in the bit-stream. This is achieved by guaranteeing that any given symbol code is not a prefix of any other symbol. An example of a variable-length prefix encoded string is given in Fig. 4, for the example BWT sequence.

To decode a variable-length prefix encoded sequence, it is usually sufficient to count the number of consecutive bits (e.g. recursively count the number of 1's in the encoded bit-stream). With a set of logical shift and bit counting operations it is possible to easily decode such sequences.

Hence, independently of the adopted compression strategies, the implementation of compressed index search procedures make an extensive use of bit-level operations. Therefore, it is widely advantageous to provide the targeted processor architecture with a native support for this type of instructions.

III. Proposed instruction-set extension

Due to their inherently higher execution times, the higher space efficiency that is offered by compressed indexes is not always exploited in bioinformatics. This major constraint results not only from the higher algorithmic complexity and irregularity but also from the consequent reduced efficiency of the corresponding execution unit.

As previously referred, when searching for a given sequence pattern on compressed index data structures, a high number of bit-level operations is required, as the information within the index is coded at the bit-level. As a result, the offer of a broad set of new instructions that more efficiently deal with bit-level operations, as opposed to the usual word-level operations of common GPPs, is regarded as a potential means to improve the efficiency of these search procedures. Furthermore, when coupled with the implementation of rather efficient execution units, it is possible to envisage a processing platform with particularly high processing throughputs for this class of prominent, but rather demanding, data structures.

Besides the common *or/and* bitwise logical operations (mainly used for bit masking) and the *shift right/left* operations (used for bit-level manipulations), the decode and search procedures over a compressed FM-index may be significantly accelerated by offering the programmer with a comprehensive set of operations to accelerate bit-level counting functions.

TABLE I
PROPOSED INSTRUCTION-SET EXTENSION.

Mnemonic	Operands			Semantics
popcnth	Rd	Ra	Rb	Rd = #bits_1 (Ra&((0x1 ≪ Rb) -1))
popcntl	Rd	Ra	Rb	Rd = #bits_0 (Ra&((0x1 ≪ Rb) -1))
msbh	Rd	Ra	Rb	Rd = orderMSb1 (Ra&((0x1 ≪ Rb) -1))+1
msbl	Rd	Ra	Rb	Rd = orderMSb0 (Ra&((0x1 ≪ Rb) -1))+1
leadh	Rd	Ra	Rb	Rd = #leading_1s (Ra&((0x1 ≪ Rb) -1))
leadl	Rd	Ra	Rb	Rd = #leading_0s (Ra&((0x1 ≪ Rb) -1))

lbt	Rd	Ra	Rb	Addr = Ra+Rb
				Rd=(*Addr=TT.TAG[0..3] ? TT.DATA[] : *Addr)

lbti	Rd	Ra	IMM	Addr = Ra+Imm
				Rd=(*Addr=TT.TAG[0..3] ? TT.DATA[] : *Addr)

swt	Rd			TT.TAG[0]=Rd[15..8]; TT.DATA[0]=Rd[7..0]

These specific operations are not usually available on common GPPs, leading to a significant processing overhead in these devices. Furthermore, other forms of compression (like variable length prefix coding) also benefit from the availability of other specialized bit-level instructions (e.g. the order of the most significant bit and the number of leading bits).

Another important aspect concerning the usage of compressed text indexes is observed when searching for a given symbol in the index: the value of the character representing such symbol usually cannot be directly used to address the index structure (e.g. the ASCII value of 'A' cannot be typically used to directly address the directory data structure). As a result, when the index is consulted, the character being searched for must be translated to its corresponding numerical representation in the index (e.g. 'A' = 0, 'C' = 1, etc.), which requires accessing a translation table normally stored in the main memory.

Accordingly, to efficiently address these important limitations in the execution of search operations over a compressed index, it is advantageous to complement the commonly available bit-level logical and shift instructions, usually present in most GPPs, with a set of specialized instructions that *i)* count the number of bits in a word; *ii)* determine the order of most the significant bit; *iii)* count the number of leading bits; and *iv*) transparently translate the value of a given character to the corresponding index addressing value. Such instruction-set extension potentially provides a significant reduction of the number of instructions that are executed along the search procedure, with a consequent improvement of the resulting performance.

Table I presents the complete set of instructions that comprises the proposed instruction-set extension. The new bit level instructions implement the count of the number of set (1) or unset (0) bits in a given word (*popcnth* and *popcntl*), the count of the number of leading high or low bits (*leadh* and *leadl*) and the order of the most significant high or low bit in a word (*msbh* and *msbl*). To implement automatic and transparent symbol translations, the *lbt* and *lbti* instructions perform a memory load of a byte followed by an automatic translation of its value. Finally, the *swt* instruction stores the translation values in the translation table.

With exception of the *swt* initialization instruction, the proposed instructions follow a 3 operand format, consisting of a destination register (Rd) and two source values (Ra and Rb or an Immediate value). For the bit level instructions, the Rb register is also used to mask the value of the Ra register, thus allowing for a greater flexibility. For the translation, two load instructions are provided: one using only register-based addressing; and another that uses an immediate value to determine the address. The translation store instruction (*swt*) has one single operand, corresponding to the value that should be stored in the translation table.

The proposed set of instructions were implemented as an extension to the Xilinx MicroBlaze instruction-set. Besides being a GPP commonly used in embedded applications, this base Instruction Set Architecture (ISA) was chosen due to its low complexity, to the availability of free hardware descriptions of the underlying micro-architecture and to the existing support in the GNU GCC compiler collection.

A. Compiler support

To provide effective support to the programmer, a dedicated back-end for the well known GCC compiler suite was designed, thus allowing for an easier programability of the proposed ASIP. This backend is based on the already available backend for the MicroBlaze processor and supports the complete set of new instructions.

Furthermore, it is worth noting that the GCC compiler suite provides a set of high level functions to address some of the operations (e.g. *__builtin_popcount*). These functions are either mapped to a single instruction available on the target architecture or calculated using an optimized library that achieves the same result using the available instructions (typical solution for common GPPs, such as Intel, AMD and ARM). Some of the now proposed instructions can be directly mapped to the corresponding GCC built-in function, as is the case of the *popcnth* instruction, which maps directly to the *__builtin_popcount* function and the *leadl* instruction, which maps to the *__builtin_clz* function, thus providing an even higher integration of the proposed ASIP with a full-featured programming toolchain.

IV. IMPLEMENTED ASIP ARCHITECTURE

The proposed ASIP was developed by using the MB-Lite [11] RISC micro-architecture as its base processing structure, to implement the proposed instruction-set extension. This decision was mainly supported by the lightweight and freely available nature of this processor core, completely described in VHDL and implementing the basic ISA of the Xilinx MicroBlaze GPP. Furthermore, the modular design of this core makes it fully customizable and easily adaptable to the requirements of the now proposed instruction-set extension. Its a 32-bit Harvard Reduced Instruction Set Computer (RISC) micro-architecture is composed by a 5-stage pipeline, implementing an in-order processing scheme.

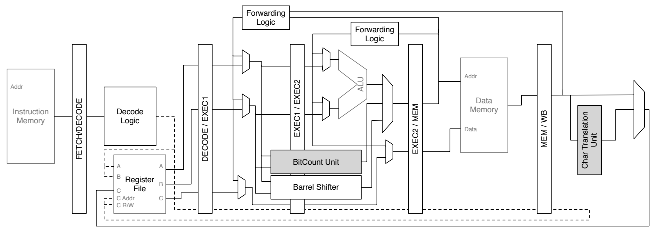


Figure 5. Processor datapath: the shaded blocks represent the newly introduced units in the base micro-architecture.

A. Processor Datapath

The developed ASIP adopts a simple and efficient datapath to implement the proposed instruction-set extension. However, contrasting to the original 5-stage MB-Lite processing structure, the pipeline of the proposed ASIP is composed by 6 stages, as a result of splitting the execution stage into two distinct stages, to improve the performance of the processor and allow for the inclusion of pipelined functional units. In this particular arrangement, the ALU was moved to the second execution stage, while a large amount of the data multiplexing and forwarding was maintained on the first stage. Additionally, convenient forwarding logic was also added to the second stage, although with a reduced complexity due to the fact that it only considers forwarding from the next pipeline stage. This strategy allows to maintain the processor operating frequency, despite the inclusion of more complex functional units (e.g. the barrel shifter).

To address the execution of the two new groups of instructions of the proposed instruction-set extension, two new functional units (see shaded blocks in Fig. 5) were included: the *Bit Count* unit, which implements the bit-level instructions, and the *Translation* unit, which implements the translation instructions. The *Bit Count* unit is implemented as a two-stage pipeline and was split in the two execution stages, while the *Translation Unit* was introduced in the Write-Back stage. The location of the translation unit provides the means for an immediate translation of the loaded value from memory, without requiring the issue of an additional instruction.

Besides these specialized functional units, the computational requirements of the search operation over a compressed index also demands for the inclusion of a barrel shifter (for efficient masking and array indexing operations). Although the base MB-Lite micro-architecture already includes a barrel-shifter, it is integrated in the Execution stage of the 5-stage processor pipeline, which significantly limits the processor's maximum operating frequency due to the higher delay that this unit imposes. To overcome this issue, the adopted 6-stage pipelined datapath includes a two-stage pipelined barrel-shifter as presented in Fig. 5. This configuration enables the processor to more efficiently execute the barrel-shift operations without limiting the overall maximum operating frequency and thus improving the overall performance.

The several changes to the datapath design required the control unit of the processor to be conveniently adapted, in order to detect the additional data hazards that arise from this new construction. Furthermore, the processor forwarding lines, which help in the reduction of the number of data hazards, were also adapted to support the 2-stage pipelined barrel-shifter.

B. Specialized Functional Units

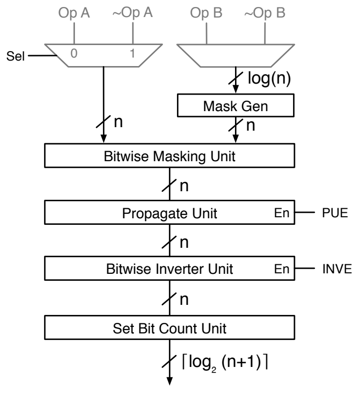
A special attention was considered in the design of the two functional units that were included in the processor datapath, in order to avoid any significant constraint in the maximum operating frequency. Furthermore, these units ensure the execution of all the proposed instructions with a latency of one single clock cycle, in accordance with the remaining data processing instructions of the base micro-architecture.

1) Bit Count Unit

The *Bit Count* functional unit, whose block diagram is presented in Fig. 6, is responsible for executing all the bit level instructions of the proposed instruction-set extension. It is composed of four stages: *i)* an initial stage that performs a bitwise *masking* function; *ii)* a *propagate unit*, used in the evaluation of the order of the most significant bit, as well as the number of leading bits in a word; *iii)* a controlled *bitwise inverter*, to define whether set or unset bits will be evaluated in the following stage; and *iv)* the *set bit count* unit, capable of counting the number of set bits in a word. Both the A and B input operands are 32-bit words.

The set of operations that are implemented by this unit is controlled by using several control signals, which are generated by the decoding logic of the processor. The table presented in Fig. 6 shows the several combinations that control which operation is executed by the *Bit Count* functional unit.

The first stage of this functional unit provides the hardware support for a bitwise *masking* operation coupled with the mask's generation. This preliminary stage is offered as an optional pre-processing step in all the newly proposed operations, providing an extra reduction of the amount of instructions that otherwise would be required to obtain the masked value,



	Sel	PUE	INVE
popcnth	0	0	0
popcntl	1	0	0
msbh	0	1	0
msbl	1	1	0
leadh	1	1	1
leadl	0	1	1

Figure 6. Bit Count Functional Unit.

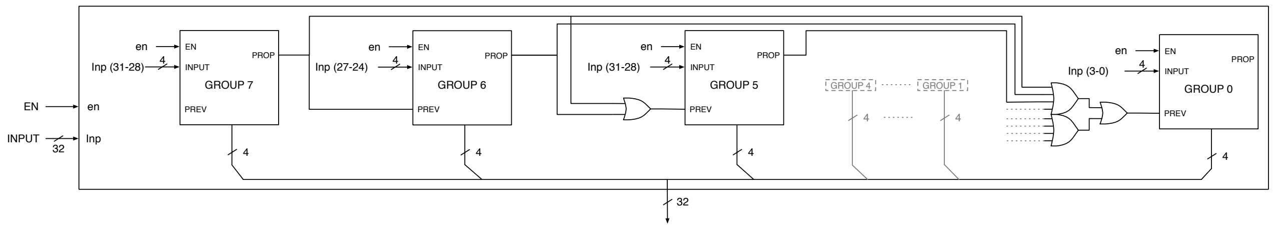


Figure 7. Block diagram of the set bit propagate unit.

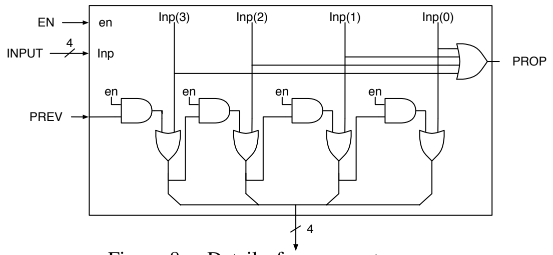


Figure 8. Detail of a propagate group.

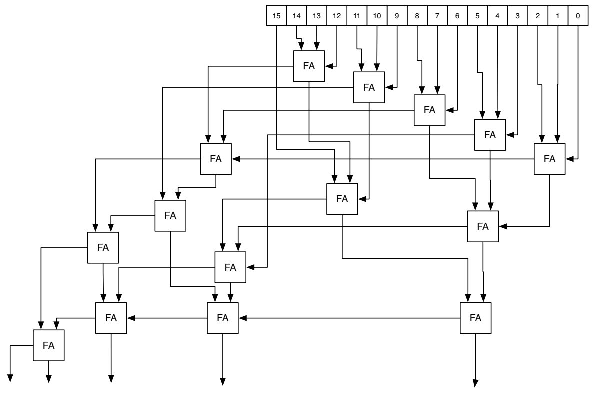


Figure 9. Adder tree for a 16-bit set bit count unit.

thus contributing to the reduction of the whole application execution time. The generation of the mask is performed by setting to high the $n + 1$ least significant bits of the word, where n is the value of the B input operand.

To support the execution of the *msbh/l* and the *leadh/l* instructions, a propagate unit was implemented and included in the second stage of this functional unit. This unit, whose architecture is presented in Fig.s 7 and 8, propagates the most significant set bit of the input operand to all of the lower order bits of the output. The operation of this unit is controlled by its enable signal: when deasserted, the input operand is forwarded directly to the output. When used in combination with the other stages, this unit gives support to the implementation of instructions that determine the order of a given bit (e.g the most significant bit).

In order to reduce the critical path, this propagate unit is composed of several groups that perform a look-ahead of the output value. Within each propagate group, a given output bit can be seen as the result of the logical *or* of the corresponding input bit with all of the leading (higher order) input bit values (see Fig. 8).

At this respect, it is worth noting that the developed hardware description of this propagate unit allows for a customizable architecture with a varying number of groups and, consequently, a shorter or longer critical path. For example, when considering a propagate group of 4 bits to process a 32-bits word, the length of critical path can be reduced from the original 64 levels to only 13 logic levels.

The third stage of the proposed functional unit is a controlled bitwise inverter, implemented as an array of XOR gates. Finally, the forth and last stage of this functional unit is composed of a *set bit counter*, responsible for determining the number of bits that are set to high within a processor word. This bit count unit was implemented with an adder tree of full adders. In Fig. 9 it is illustrated an example configuration that processes a 16 bit word. Naturally, the implemented adder tree corresponds to a straightforward extension for a full 32-bit word in order to comply with the MB-Lite word size. This *set bit counter* unit provides the basis for the implementation of the several variations of bit-level instructions that rely on an efficient counting operation of asserted bits in the input word.

2) Translation Unit

To assist the index search operations over the compressed index, a translation table was included in the ASIP micro-architecture, in order to reduce the number of memory accesses and clock cycles that are required to translate a given symbol to the representation used within the index directory information. Furthermore, this translation unit can also be used to assist in the search of the two strands of the considered DNA sequence, in which the complementary sequence has to be determined. This particular search operation is usually performed in two steps: *i)* determination of the complementary sequence; and *ii)* search for this sequence in the index (requiring a translation step to access the index symbols). With this unit, it is possible to simultaneously perform these two operations with a single instruction, by coding it with the adequate values.

The developed translation unit, shown in Fig. 10, is responsible for translating a given 8-bit value (the usual size of an ASCII character representation) to another value, based on a smaller based on a smaller representation (commonly less than 8 bits). This unit operates similarly to a fully associative cache memory, by outputting the stored *DATA* value whenever the input value matches any of the *TAG*s. Naturally, considering the number of binary comparators that are required to implement this unit (one for each *TAG*), the number of

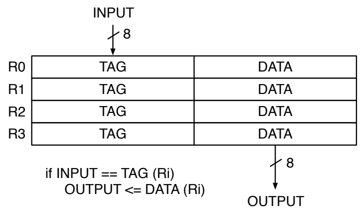


Figure 10. Translation unit.

positions in this unit should be kept to a minimum, although the developed structure allows for an arbitrary number of positions. At this respect, it should be noted that the number of required positions is typically small for the considered usage scenario of DNA sequence assembly and mapping, in which only four distinct symbols (A,C,G and T) are used to represent the DNA sequence, thus requiring only four positions. If protein sequences are considered, 22 positions will have to be implemented in order to accommodate the representation of each amino-acid.

This translation memory is pre-initialized by means of a dedicated instruction that stores a 16-bit word to this table: the most significant 8-bits represent the TAG, while the least-significant 8-bits represent the DATA to be returned (translated value). The write operation to this unit functions as a FIFO: the oldest tag-data pair is discarded when a new pair is written.

V. Experimental Results

To evaluate the proposed ASIP architecture, a thorough analysis focusing both the hardware resources and the obtained performance was conducted. In this section, the additional hardware resources that are required by the introduced functional units and by the changes to the micro-architecture are analyzed, as well as the resulting impact on the maximum operating frequency. Afterwards, the proposed instruction-set extension is firstly evaluated in terms of its efficiency to reduce the number of required instructions to execute the three compression methods presented in Section II. Finally, a thorough performance assessment of the developed ASIP is presented, highlighting the benefits of the proposed instruction-set extension and of the optimized micro-architecture in the execution of the search operation over the compressed indexes by using both the base GPP and the proposed ASIP. The developed ASIP architecture was prototyped using a Xilinx Virtex7 FPGA (XC7VX485T), embedded in the VC707 Evaluation Kit by using the Xilinx ISE 13.1 and the Modelsim SE 10.0b development tools.

In Table II, the proposed ASIP is compared with the original MB-Lite in terms of the occupied hardware resources. The presented resource values of the MB-Lite architecture correspond to a configuration that includes the barrel-shifter. As it can be observed, these results demonstrate that the new specialized functional units increase the used amount of hardware resources (LUTs) by 34% when compared to the base architecture. In terms of the operating frequency, it is possible to observe that the proposed ASIP micro-architecture, with its

TABLE II
Resource usage of the proposed ASIP.

	LUTs	Registers	RAMB18	Max. Freq.
MB-Lite	878	333	3	137 MHz
Proposed ASIP	1177	498	3	247 MHz

custom 6-stage pipeline, is capable of achieving a significantly higher maximum operating frequency (247 MHz vs 137 MHz), which allows an increased processing throughput.

To evaluate the proposed instruction-set extension, an indexed search procedure was programmed and executed both on the proposed ASIP and on the original MB-Lite micro-architecture, for comparison purposes. The used dataset includes fragments from the *E.coli* genome (U00096.3), with approximately 10×10^3 characters, as the reference sequence (*text*) and a set of 100 short read sequences, with 35 characters each, originated from the same genome.

When compared in terms of the required number of instructions, the original MicroBlaze instruction-set requires about 18 instructions to perform the same operation as the *popcnth/l* instruction, an average of 14 instructions to perform the *msbh/l* operation and an average of 20 instructions to perform the *leadh/l* instruction. From this comparison, it can be observed that the number of instructions can be reduced by a factor as high as 20x, as is the case of the *leadh/l* operations.

The chart presented in Fig. 11 depicts the obtained speedup considering the required number of clock cycles to calculate the values of the $Occ(c, i)$ matrix, for the FM-Index and Wavelet tree, and to decompress a string, for the variable-length prefix code. For each of the considered compression strategies, two different evaluations of the speedup are shown: *i)* for the procedure code segment that directly deals with the bit-vectors, with values ranging from 4 to 6.8, and *ii)* for the complete procedure that implements the $Occ(c, i)$ computation (FM-Index and Wavelet trees) and the decoding (variable-length prefix code). These results take into consideration the attained reduction in terms of executed instructions and the few stalls that are introduced during processing (due to data hazards arisen by the new pipeline arrangement). As expected, the speedup values of the code segments that directly manip-

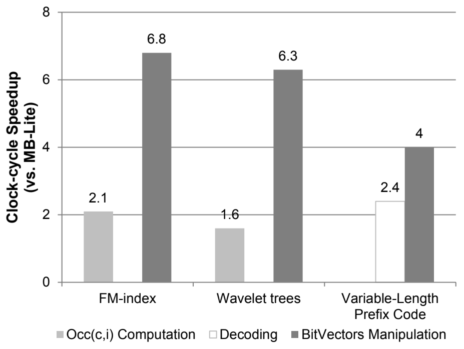


Figure 11. Performance comparison, in clock cycles, of the proposed ASIP executing the operations on the three compressed data structures (FM-index, Wavelet trees and Variable-Length Prefix Code).

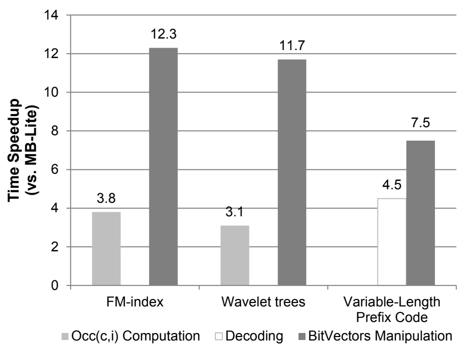


Figure 12. Performance comparison, in execution time, of the proposed ASIP executing the operations on the three compressed data structures

ulate the bit-vectors are higher than those that relate to the complete procedure, in which other instructions are executed (Ahmdahl's law). Nevertheless, even when considering the complete procedure, the proposed instruction-set extension allows for a significant speedup, which ranges from 1.6 to 2.4, according to the used data structure.

Regarding the actual execution throughput, which takes into consideration the actual operating frequency of the ASIP and the attained reduction in terms of clock cycles to execute the operations, it is possible to observe from Fig. 12 that the speedup of the proposed ASIP may be as high as 12.3, when considering the bit-manipulation segment of the procedure's code. When considering the complete procedure, the speedup ranges from 3.1 to 4.5.

Besides the execution throughput, the conducted analysis also considered the energy consumption of the proposed ASIP, by using the Xilinx power estimation tool considering the maximum clock frequency for each architecture. Fig. 13 depicts the obtained energy values of the proposed ASIP as well as those of the MB-Lite. From these results, it is possible to observe that the proposed ASIP requires a significantly smaller amount of energy to execute the complete procedure, mainly due to the smaller run time it needs to execute the operations. This reduction is even more significant if only considering the bit-manipulation segment of the code. Hence, the presented results demonstrated that the proposed ASIP provides advantages both in terms of the throughput and the energy efficiency, making it an adequate core to be included in a multi-core platform targeted at the high-performance processing of biological sequences. Such multi-core structure would adopt a shared-memory model (to allow a transparent access to the shared index data) and a processing model based on data-level parallelism, in which each core processes a subset of the typically vast query sequence set.

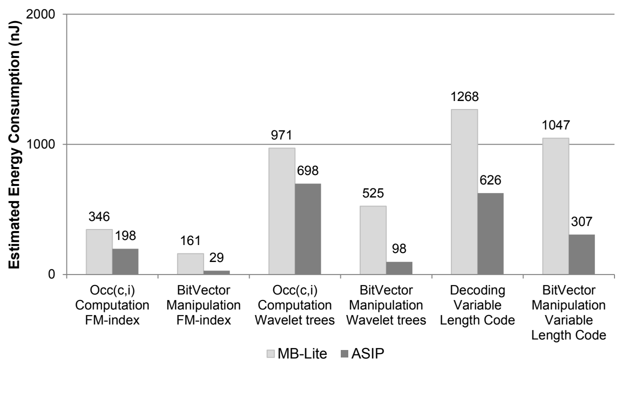


Figure 13. Comparison of the consumed energy.

VI. Conclusions

This paper proposed a new ASIP architecture especially designed to accelerate the processing of biological sequences using compressed full-text indexes based on the BWT. The proposed processor was built over a RISC micro-architecture and extends the Xilinx MicroBlaze ISA with additional bit-level operations, especially tailored for compressed indexes. The obtained results show that the proposed instruction-set extension clearly reduces the number of required clock cycles to about one half when executing several operations on compressed data structures. In terms of execution time, the especially designed datapath of the underlying micro-architecture allows to reach speedups that range from 3.1 to 4.5. When evaluated in terms of energy, the proposed ASIP is capable of performing the same operations using a substantially lower energy, thus improving both the performance and the energy efficiency.

References

[1] F. Fernandes, P. da Fonseca, L. Russo, A. Oliveira, and A. Freitas, "Efficient alignment of pyrosequencing reads for re-sequencing applications," *BMC Bioinformatics*, vol. 12, no. 1, p. 163, 2011.

[2] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," *Genome Biol.*, vol. 10, no. 3, p. R25, 2009.

[3] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, vol. 14, no. 3, pp. 249–260, Sep. 1995.

[4] T. W. Lam, W. K. Sung, S. L. Tam, C. K. Wong, and S. M. Yiu, "Compressed indexing and local alignment of DNA," *Bioinformatics*, vol. 24, no. 6, pp. 791–797, 2008.

[5] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows-Wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, July 2009.

[6] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proc. of the 41st Annual Symp. on Foundations of Computer Science*, 2000, pp. 390 –398.

[7] P. Szecowka and T. Mandrysz, "Towards hardware implementation of bzip2 data compression algorithm," in *Mixed Design of Integrated Circuits Systems, 2009. MIXDES '09. MIXDES-16th Int. Conf.*, June 2009, pp. 337–340.

[8] S. Arming, R. Fenkhuber, and T. Handl, "Data compression in hardware - the burrows-wheeler approach," in *Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2010 IEEE 13th Int. Symp. on*, Apr. 2010, pp. 60–65.

[9] E. Fernandez, W. Najjar, and S. Lonardi, "String Matching in Hardware Using the FM-Index," in *IEEE 19th Annual Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, may 2011, pp. 218 –225.

[10] M. Burrows and D. J. Wheeler, "A Block-sorting Lossless Data Compression Algorithm," Digital Equipment Corporation, Tech. Rep. 124, May 1994.

[11] T. Kranenburg and R. van Leuken, "MB-LITE: A robust, light-weight soft-core implementation of the MicroBlaze architecture," *Design, Automation and Test in Europe Conf. and Exhibition (DATE)*, pp. 997–1000, Mar. 2010.