Low-Complexity Two Instructions Set Computer for Suffix Sort in Burrow Wheeler Transform

J. H, Kong
The University of Nottingham,
Jalan Broga, 43500 Semenyih,
Selangor Darul Ehsan, Malaysia.
Email:
keyx9kjh@nottingham.edu.my.

L.-M. Ang
School of Engineering,
Edith Cowan University,
Joondalup, WA 6027, Australia.
Email: li-minn.ang@ecu.edu.au.

K. P. Seng
School of Computer Technology,
Sunway University Malaysia,
No. 5, Jalan Universiti,
Bandar Sunway, 46150 Petaling
Jaya, Selangor, Malaysia.
Email: jasmines@sunway.edu.my.

***Abstract*—The Burrow Wheelers Transform (also called the block sorting compression) was referred to as the jewel lossless compression due its effectiveness and is used as the core algorithm in bzip2 compressor. With much attention given, hardware realization of the BWT algorithm has been limited due to the complexity of suffix sorting computation. Given the small hardware-footprint design trend, we propose the use of a reconfigurable FPGA platform and unified computer architecture with minimal hardware components. In this paper, we are presenting a low-complexity two instructions set computer architecture (TISC) for the lexicographical sorting in Burrow Wheelers Transform. The proposed architecture has been implemented and tested using the DK Design Suite software environment, which provides a Handel-C Hardware Descriptive language to ease the design process. A Celoxica RC10 board which houses the Spartan 3 XCS1500L-4 FPGA is used.**

Keywords-Burrow Wheelers Transform; BWT; BWCA; URISC; Computer Architecture; FPGA; Suffix Sort

I. INTRODUCTION

Practical application for image processing algorithms till today is a great challenge due to the requirements for stringent resource constrained design environments. In the sub-domain of the field of computer vision, image, multimedia and data compression has always been in the center of the researcher's attention to discover alternatives for an efficient hardware realization of lossless and lossy-compression algorithms. The drive for this motivation is a result from the explosive amount of multimedia digital data emerged from continuous advancements in digital communication systems. The rate of development in mobile communications and systems demand higher data bandwidth in transmission. With this, data compression solution offers a suitable approach to reduce the data redundancy.

Among all the lossless compression algorithms known, the Burrow Wheelers Transform (BWT) has gained a certain amount of attention due to its effectiveness in lossless compression. The BWT is capable of obtaining compression ratios close to the prediction by partial-matching (PPM) based algorithms [1]. However, the originally proposed BWT is resource demanding, both in hardware and software requirements. Specifically, the BWT endorses the lexicographic sort (also known as the suffix sort) which is memory-demanding. There are a few variants and improvements proposed. We have found out that in [2], the author has explained that in literature, there are three different approaches can be found, which will be explained in three different areas: 1. reducing the complexity of the task altogether, 2. using parallel sorting to speed up the algorithm, such as the weave-sorter and, 3. using a sufficiently simple suffix-sorting algorithm.

Over the years of developments, VLSI technologies have been a great platform for hardware implementation. For prototyping purposes, FPGAs allows hardware structure configurability and algorithmic synthesis. The very nature of the BWT algorithm requires a large amount of memory, software and hardware. A matrix is required for storing the suffixes for operations such as shifts and relocation and even lexicographical sorting. This means that the very foundation of the algorithm is memory consuming thus, our aim is to reduce hardware resource consumption through efficient hardware implementation techniques. The work from Martinez [1] is a good example of hardware implementation. The author in [1] uses a Virtex XCV300-4 BG352 FPGA, running at a clock speed of 45MHz, with parallel Weave-sorter architecture. On the other hand, the work from Mukherjee [3], which is actually the original proposal of weave-sorter architecture, also uses the Virtex XCV-300 FPGA. The most recent work that we found is by Arming [2], using an Altera Cyclone II EP2C35F484C6N which consists of 32k LEs (logic elements) with 483kbit of internal memory and a Altera Stratix II EP2S180F1020C3 which consists of 179k LEs with 9Mbit of internal memory. The author in [2] had also based his design on the weave-sorter. With these examples, we can observe that the popular approach is the hardware solution, not the software or programming solution. Thus, we aim to provide the simplest hardware platform with custom-written programs running the BWT program.

In this paper, we are presenting an ultimate reduced instruction set computer architecture (URISC) for the lexicographical sort (will be referred as the suffix sort

978-0-7695-4959-0/13 $25.00 © 2013 IEEE
DOI 10.1109/ACSAT.2012.61

throughout the rest of this paper), namely the two instructions set computer (TISC). The proposed architecture has been implemented and tested using the DK Design Suite software environment, which provides a Handel-C Hardware Descriptive language to aid the design process. A Celoxica RC10 board which houses the Spartan 3 XCS1500L-4 FPGA is used.

The structure of this paper is organized as: section 1 is the introduction. Section 2 is the background and literature review of related topics. Section 3 presents our proposed methodology (the TISC sorter) is more detail. Section 4 discusses our findings and results and lastly, section 5 is the conclusion.

II. BACKGROUND AND LITERATURE REVIEW

A. Burrow Wheelers Transform

The original BWT algorithm begins by creating a two dimensional matrix of the original data stream, with each row of the matrix filled with the predecessor row, line shifted cyclically to the left by one place. As an example, if a given symbol string consists of *n* number of symbols; the expanded dimensional size of the symbol matrix is *n x n*. If the string S = [a, b, c, d], the next row of the matrix will be filled with the left-shifted string S = [b, c, d, a]. The author in [4] has given a set of good examples of how the BWT works. To illustrate, let input string S = [a, b, r, a, c, a], with the number of symbols *n* = 6 and the character set in the strings are X = [a, b, c, r]. In the table 1 below, we can observe that the initial string [a, b, r, a, c, a] and the original index I, [0, 1, 2, 3, 4, 5] is moved to the row position with the index of number 1, I = 1. The result of the BWT execution yields a string of symbols which is chosen from the last column of the matrix and an index number to identify the where the initial string is in positioned and recovers the original string while BWT decode and the selection of the last column give the string of [c, a, r, a, a, b] as the output of the BWT.

B. Suffix Sort (Weave Sorter)

Data sorting is one of the fundamental parts of data processing algorithms. For hardware implementations, there are a few known sorters namely: the Systolic, Bitonic, Weave and Insertion [5]. One of the sorters that caught our attention is the Weave-sorter. In [1], the author has shown a comparator block for the his weave-sorter implementation. From the original BWT proposal, it is required that a matrix generation would hold all the cyclic versions of the symbols. At the end of the day, a comparator must be present in order to execute any form of sorting. The parallel sorting architecture employs a more complex hardware platform with an enhanced sorting strategy as compared to the original Weave-sorter in [3, 6].

TABLE I. THE OUTPUT OF THE BWT IS THE LAST COLUMN (S5) OF THE SORTED MATRIX.

Row	S0	S1	S2	S3	S4	S5
0	a	a	b	r	a	c
1	a	b	r	a	c	a
2	a	c	a	a	b	r
3	b	r	a	c	a	a
4	c	a	a	b	r	a
5	r	a	c	a	a	b

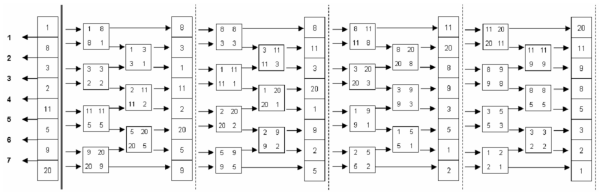

Figure 1. The two levels of parallel sorting (figure taken from [1]).

To make a fair comparison, the author has compared the original Weave-sorter and the parallel sorter in terms of the total number of steps to sort an array. In [3], the author has mentioned that a Weave-sorter has a complexity of O(n) and only requires 4n steps to sort a string. With the parallel sorting strategy, the author has proved that the number of sorts is improved to n/2 (where n = the number of data). For example, the worst case of number of sorts that will occur using the parallel sorting strategy for 8 data is 4. In this paper, we will adapt to the parallel sorting strategy for minimum number of sorts in worst case. The parallel sorting strategy is shown in figure 1.

C. One Instruction Set Computer (OISC)

The one instruction set computer (abbreviated as OISC sometimes also referred as the URISC or the ultimate reduced instruction set computer, we refer) is a single

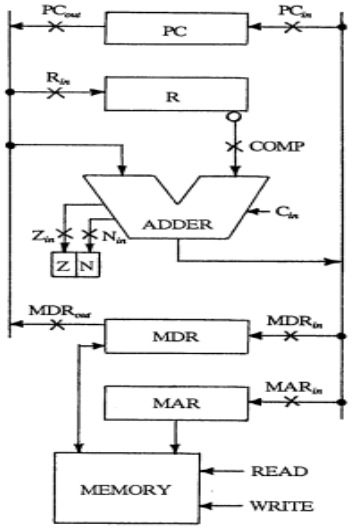

Figure 2. The illustration an URISC architecture (figure taken from [8]).

instruction set computer and also the penultimate reduced instruction set computer [7]. The idea of an OISC is the opposite of a CISC (complex instruction set computer), which incorporates many complex instructions as micro-programs within the processor. At a higher level, an OISC does project CISC-like property as the synthesis of low-level instructions became more and more.

An OISC is more like a RISC (reduced instruction set computer) due to the fact that it has been called the 'ultimate RISC'. The OISC is very simple, by examining its features and properties; we can gain greater insight than of RISC and CISC because such complex and varied instruction sets has a more fundamental and simplest features hidden. One particular type of OISC that uses only one instruction called the SBN instruction (Subtract and Branch If Negative).

By using only the SBN instruction, the OISC is able to perform data addition and subtraction. Logical operations can be performed to execute data movement from one location to another. The detailed operation of the URISC can be found in [8] and figure 2 shows the schematic of the URISC architecture presented in [8] and it is observed that the URISC consists of an Adder circuit as its sole ALU.

III. The Proposed Two Instruction Set Computer for Suffix Sorting

In this section, we are presenting the architecture for the TISC sorter. We are also covering the idea that inspired the following TISC proposal and the core idea of the definition of sorting behavior. In the figure 6, we have illustrated an abstraction of the computer architecture that is built based on the original URISC model, namely the two instructions set computer (TISC). The TISC architecture provides the platform for configuration of different models to cope for specific applications. The ALU core is a space for programmer to put in application-specific processing units for various logical or arithmetic functions. Thus, new instruction sets are born with new ALUs installed. Like URISC, the TISC employs the SBN instruction and hence, with just one additional ALU that we propose (explained in section III.A), we propose an architecture capable of running the parallel sorting functions.

A. The AS-ALU (Application Specific Arithmetic Logic Unit)

To sort in lexical order, a comparator is essential. Inspired by the parallel sorter proposed in [1], we adapted the author's approach of having a comparator block or a comparator cell for the sorting purpose. But during the development process, we have realized that in order to completely sort an array of data, the basic instruction set for the architecture has to be defined. The TISC architecture allows programmers to define an AS-ALU (Application Specific Arithmetic Logic Unit), in other words, a design space for programmer to put in ALUs. In our application, we would need to define a comparator block hence; a new instruction set to the TISC architecture has to be initialized. In this paper, we are presenting two instructions: the SBN (subtract and branch if negative) and the CBL (compare and branch is A is larger). The SBN instruction is crucial to the whole architecture. It defines the fundamental character of the computer. The advantage of the SBN architecture is that, the SBN instruction itself is capable of four different logical operations: ADD / SUBTRACT, MOVE / COPY, JUMP and CLEAR. In computer programming, such a universal instruction set is deemed powerful as there is no additional hardware or additional instructions set required. With the URISC SBN architecture, a fundamental processing engine is realized. A more detailed description of the extended SBN instructions can be found in table 2.

For sorting, we introduce the CBL (Compare and Branch if A is larger) instruction. The function of the CBL instruction is fairly simple. In figure 6, we can see that the illustrated architecture has 2 input parameters into the TISC AS-ALU: Input_A and Input_B. Since the architecture is run by an FSM, the data movement and processing are fixed within 9 clock cycles. By taking advantage of this, together with the event of when a CBL instruction is called, the negative flag will be forced if the value of Input_A is larger than Input_B. Due to the nature of the architecture, when a negative flag is triggered, the subsequent data movement would involve the over-writing of the PC value hence, a JUMP operation is executed. In other words, the CBL instruction is very similar to the conditional SBN JUMP instruction, with the difference of have a logical condition (SBN is triggered by the output of an arithmetic operation on to 2 input parameters while the CBL is triggered by the comparison outcome of 2 input parameters). By designing the architecture this way, it would greatly decrease the need for additional hardware in the ALU core since the SBN and CBL is sharing the same negative flag.

TABLE II. The Synthsis of the SBN Instruction Sets for a Higher-Level Functionality

Instruction Sets	Description	3-tuple instruction format
SBN ADD / SUBTRACT	2 SBN instructions would result to an addition operation. Hence B – (-A) = A+B and the value of (A+B) is stored in the B's address.	1) SBN $A, $C, 0; (C = -A), where address C contains the value of zero and A is the A input. 2) SBN $C, $B, 0; B –(-A) = B+A, where address B contains the value of B input.
SBN MOVE / COPY	2 SBN instructions would result to a MOVE operation. Hence D = A, and the value of A is stored in the D's address.	1) SBN $A, $C, 0; (C = -A), where address C contains the value of zero and A is the A input. 2) SBN $C, $D, 0; (D = +A), where address D contains the value of zero.
SBN JUMP	1 SBN instruction would result to a JUMP operation. The	1) SBN $X, $Y, $J; (J = jump address)

Instruction Sets	Description	3-tuple instruction format
	condition would be the resultant output of X-Y = a negative value.	where X-Y = -ve (value)
SBN CLEAR	1 SBN instruction would result to a CLR memory operation.	1) SBN $A, $A, 0; (A-A = 0), where address A contains the value of A input.

TABLE III. THE TISC INSTRUCTION SETS

Operation	Function Code (1-bit MSB)	Instruction Format
SBN (Subtract and Branch if Negative)	0	(0 @ address A), address B, Target
CBL (Compare and Branch if Negative)	1	(1 @ address A), address B, Target

In table 3, we can show that both instruction sets: the SBN and CBL is differentiated via a 1-bit MSB. The 1-bit MSB is appended at the value of first address of the memory. Note that a complete instruction set takes 3 values (in the form of a 3-tuple instruction).

B. The Program Memory and Instruction Set Synthesis

In process of developing the TISC sorter, we are aiming to have a more complex program rather than a complex hardware. To achieve this, we have foreseen that in order to describe the behavior of a complex algorithm such as data sorting, we would need to understand the algorithm in advance before attempt to write a code to operate the computer architecture.

The main idea that inspired our work is from the parallel sorting scheme in [1]. The parallel sorting scheme uses 7 'compare and swap' blocks and a total of 4 levels are used. Based on the worst case of number of sorts that will occur, using the parallel sorting strategy for 8 data requires 4 rounds of even and odd adjacent comparators. To perform the same operations, we used the synthesized instruction sets to create a higher-level of functions, to mimic the parallel sorting strategy's hardware behavior, in the software environment. The figure 3 and 4 shows the pseudo-code of the program.

The notion of 'compare and swap' is divided into two separate actions: 'compare' and 'swap'. With the first condition met, only then a 'swap' would occur. The pseudo-code in figure 3 represents the 7 comparisons made within the parallel sorting strategy (even and odd adjacency comparison). The CBL instructions are used to point to the respective memory locations for data comparisons. Firstly, the CBL instruction is called to compare the first and second data (out of the 8). If data A is larger than data B, a branch will occur hence, the comparison operation is completed. The second step would be the data swapping. After JUMP operation is done, the new PC value will be starting point of the architecture thus the data swapping operation begins. The program written covers all 7 comparisons. Once all the comparisons are made, a loop is injected to fulfill the N = 4 worst case iteration.

```
149 //swap_sort block1
150 0x010, 0x010, 0x000, //211 - 213 //clear mirror hub temp
151 0x011, 0x011, 0x000, //214 - 216 //clear mirror hub temp
152 0x019, 0x010, 0x000, //217 - 219 //swap 1 to 2 (-ve to +ve)
153 0x018, 0x011, 0x000, //220 - 222 //swap 2 to 1 (-ve to +ve)
154 0x018, 0x018, 0x000, //223 - 225 //clear swap -ve temp
155 0x019, 0x019, 0x000, //226 - 228 //clear swap -ve temp
156 0x010, 0x018, 0x0E7, //229 - 231 //write back to -ve swap temp
157 0x011, 0x019, 0x0EA, //232 - 234 //write back to -ve swap temp
158 0x021, 0x023, 0x0BA, //235 - 237 //jump to back to weave-sroter
```

Figure 3. The program codes written to execute the seven 'compare and swap' operation.

```
130 //all 7 comparators
131 0x110, 0x011, 0x0D2, //compare jump to swapblock1
132 0x112, 0x013, 0x0ED, //compare jump to swapblock2
133 0x114, 0x015, 0x108, //compare jump to swapblock3
134 0x116, 0x017, 0x123, //compare jump to swapblock4
135 0x111, 0x012, 0x13E, //compare jump to swapblock5
136 0x113, 0x014, 0x159, //compare jump to swapblock6
137 0x115, 0x016, 0x174, //compare jump to swapblock7
138 0x022, 0x027, 0x0B7, //check loop (loop = 4) (+1)
139 0x021, 0x023, 0x18F, //jump to force-end (+1) // 2
```

Figure 4. The program code performs the data swapping from one memory to another in the event of branching.

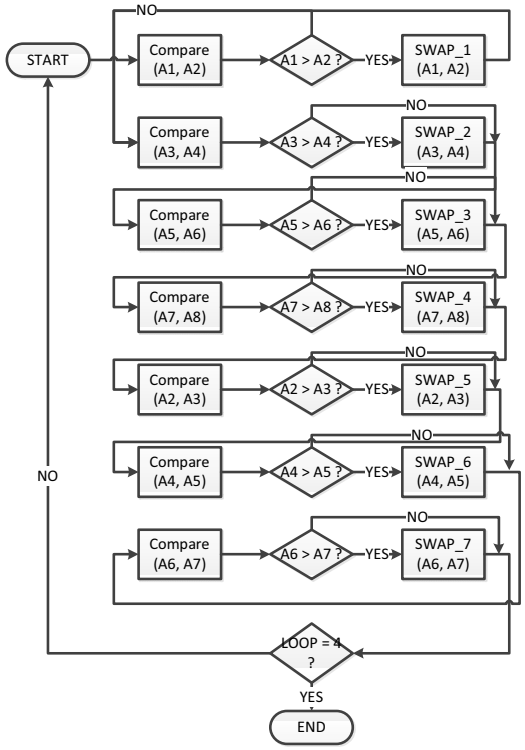

Figure 5. The flowchart of the 8 byte sorting progrram.

Note that the final program written for the sorting function is done on a 9-bit address architecture. From figure 3, it is observed that the 1 bit function code is appended onto the first address (8-bits) in a 3-tuple instruction. In figure 4, it is observed that the SBN instructions are synthesized to perform higher-level functions, such as CLEAR, JUMP and MOVE. The flowchart of the written program is described in the figure 5.

C. The Finite State Machine (FSM) Controller

The TISC's FSM controller generates a total of 14 control signals. Within 9 clock cycles, a complete instruction is being process and carried out throughout the registers and multiplexers in the architecture (from reading operands to computation completion). During each clock cycles, operations from program counter increment, to loading data item, to write the MDR register, storing the results from the ALU are being carried. Each registers are controlled and the characteristics of the data movement within MISC is predefined and fixed. The N register is used as an indicator to trigger the controller's outputs for branching. If a branch instruction occurs, the current values in the program counter register will be overwritten with a new target address. This target address was embedded in the last byte of the instructions and the PC will start at this target address over the next clock cycle. The table 4 below explains the data movement within the MISC's data-path driven by the nine master clocks.

IV. RESULTS AND DISCUSSIONS

In this section, we are presenting you the hardware implementation results of the TISC Weave-sorter. The architecture is implemented, tested and verified using the DK Design Suite software environment, which provides a Handel-C Hardware Descriptive language to ease the design process. A Celoxica RC10 board which houses the Spartan 3 XCS1500L-4 FPGA is used. Throughout the design process, you have chosen to use only the on board LED for display indicator, for minimum hardware initialization and occupancy. In the table 5 below, the logic utilization report is shown and the number of slice flip flop used is less than a hundred (91 slices). In table 6, the total amount of occupied slices is amounted at 144. As for the results for other components, the TISC sorter design only uses 1 RAMB16, which is shown in table 7.

TABLE IV. THE EXPLANATION OF THE DATA MOVEMENT WITH RESPECT TO EACH CLOCK CYCLES.

Clock Cycle	Data_A	Data_B	Remarks
1	-	-	PC to MAR ,OP to OP_reg
2	Data_A loaded to R_reg	-	-
3	-	-	PC + 1 to MAR
4	-	Data_B loaded out	-
5	Data_A read from R_reg	Data B read	The output is selected via MUX
6	-	-	The computed data is stored in MDR
7	-	-	PC + 2 to MAR
8	-	-	Branch code loaded, if -ve branch to PC + 2 + 'branch_address'
9	-	-	PC + 3 (reset)

For related work comparisons, we refer to [1] and [3]. The parallel sorting strategy in [1] was implemented onto a Virtex 2 XC2V2000-6bf957, capable of sorting 128 characters (127 characters + 1 sentinel) and utilizing 4316 slices. On the other hand, the Weave-sorter in [3] was implemented onto a Virtex XCV300-4bg352, capable of sorting 8 characters and utilizing 2703 slices. The comparisons are shown in table 8 below.

TABLE V. THE LOGIC UTILIZATION OF THE 9-BIT ARCHITECTURE FOR TISC WEAVE-SORTER.

Logic Utilization	Quantity	Total	Usage
No. of Slice Flip Flops	91	26,624	1%
No. of 4 Input LUTs	241	26,624	1%

TABLE VI. THE LOGIC DISTRIBUTION OF THE 9-BIT ARCHITECTURE FOR WEAVE-SORTER.

Logic Distribution	Quantity	Total	Usage
No. of occupied Slices	144	13,312	1%
No. of Slice containing only related logic	144	144	100%
No. of Slice containing unrelated logic	0	144	0%

TABLE VII. THE HARDWARE UTILIZATION REPORT (OTHER COMPONENTS).

Components	Quantity	Total	Usage
No. BUFGMUXs	3	8	37%
No. DCMs	1	4	25%
No. External IOBs	28	221	12%
No. LOCed IOBs	28	28	100%
No. of RAMB16s	1	32	3%
No. of Slices	144	13,312	1%
No. of SLICEMs	1	6656	1%

TABLE VIII. COMPARISON TO OTHER RELATED WORKS.

Components	Our work (TISC-sorter)	Parallel Sorter [1]	Weave-sorter [3]

Components	Our work (TISC-sorter)	Parallel Sorter [1]	Weave-sorter [3]
No. of Slices	144	4316	2703
No. BUFGMUXs	3	1	-
No. External IOBs	28	19	-
No. of RAMB16s	1	1	-
Clock Frequency (MHz)	40	51.67	45
No. of Slice Flip Flops	91	-	-

TABLE IX. THE RESULTS FOR VARIOUS ARCHITECTURES OVER N = 8 SORTS.

Sorts (N)	Cycles per sort	Total cycles		
		Our work (TISC-sorter)	*Parallel Sorter [1]*	*Weave-sorter [3]*
8	58,7,3,1,1,1,1,1	2196	1234	2304
8	56,5,2,1,1,1,1,1	2196	1229	2304

One instruction requires 9 clocks cycles to complete. The total amount of memory space occupied for the program is 403 bytes, which is well fitted into a 9-bit (512 bytes) memory. The breakdowns of the memory allocation are the following: data memory = 40 bytes, program memory = 363 bytes. Since an instruction is formed with 3 bytes, it means that we have used a total of 121 instructions (7 for CBLs and 114 for SBNs). Since there are iterated loops and reused code blocks within the TISC architecture, we have identified that there were 71 instructions that was iterated 4 times (worst case). The final count for the total instructions are (71 * 4) + 50 = 334. This yields an amount of 334 * 9 = 3006 (worst case) clock cycles at 40 MHz. The comparisons of clock cycle for different architectures can be found in table 9 above. The limitation of the TISC is bounded by the von-Neumann's architecture. The total clock cycles can be improved when using a Harvard architecture, which is projected to use less than 9 clocks per instructions, with the tradeoff of having two separate block memories.

V. CONCLUSION

In this paper, we have presented a simple and low-complexity architecture for BWT suffix sort application. With only 2 instruction sets (the SBN and CBL), 2 ALUs (the Adder and Comparator), a small data-path, and a 9-bits address block ram (amounting to 512 byte spaces). The TISC architecture that we have presented utilizes a stunning amount of 91 slice flip flops, 144 slices and only 1 RAMB16, running at 40MHz. Our design has outperformed other similar works in terms of minimum hardware utilization and design complexity.

REFERENCE

[1] J. Martinez, R. Cumplido, and C. Feregrino, "An FPGA Parallel Sorting Architecture for the Burrows Wheeler Transform," presented at the Proceedings of the 2005 International Conference on Reconfigurable Computing and FPGAs (ReConFig'05) on Reconfigurable Computing and FPGAs, 2005.

[2] 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 International Symposium on*, 2010, pp. 60-65.

[3] A. Mukherjee, N. Motgi, J. Becker, A. Friebe, C. Habermann, and M. Glesner, "Prototyping of efficient hardware algorithms for data compression in future communication systems," in *Rapid System Prototyping, 12th International Workshop on, 2001.*, 2001, pp. 58-63.

[4] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," ed, 1994.

[5] V. A. Pedroni, R. P. Jasinski, and R. U. Pedroni, "Panning sorter: A minimal-size architecture for hardware implementation of 2D Data Sorting Coprocessors," in *Circuits and Systems (APCCAS), 2010 IEEE Asia Pacific Conference on*, 2010, pp. 923-926.

[6] A. Mukherjee, *Introduction to nMOS and CMOS VLSI systems design*: Prentice-Hall, 1986.

[7] W. F. Gilreath and P. A. Laplante, *Computer architecture: a minimalist perspective*: Kluwer Academic Publishers, 2003.

[8] F. M. a. B. Parhami, "URISC: The ultimate reduced instruction set computer," Department of Computer Science, University of WaterlooJune 1987 1987.

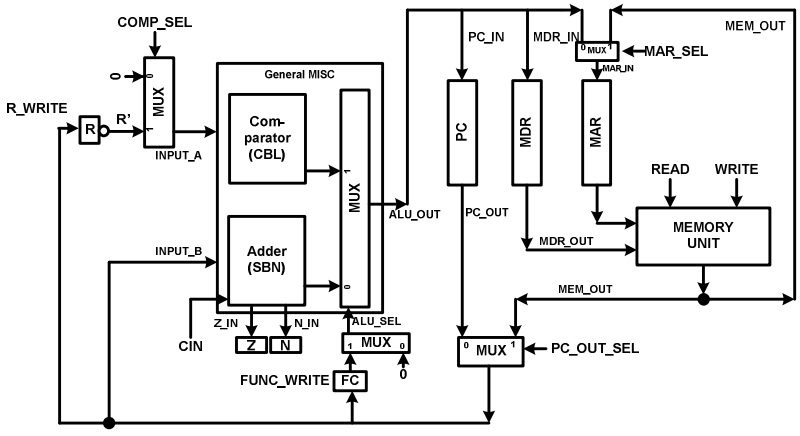

Figure 6. The abstraction of the architecture for general reconfigurable TISC