ARBITRARY LONG DIGIT INTEGER SORTER HW/SW CO-DESIGN

Shun-Wen Cheng

Tamkang University
Taipei, TAIWAN
E-mail: swcheng@ieee.org

Abstract— The coming of multimedia era and information security era indicates that must process longer digit integer data. Previous sort researches focus on pure performance of large amount of finite fixed digit/bit number. This paper discusses on effectively solving arbitrary long digit integer sorting problem by HW/SW co-design under the Area×Time^2 (AT^2) price-performance constraint. The work proposes multi-level (two-level) sort architecture to attain the object: an accomplished fixed-digit (k-bit) hardware sorter implements the first or basic level sorting, software programmed radix 2^k sort implements the second or higher level sorting. By Super Radix Sorting HW/SW co-design and reuse techniques, the work makes fixed-digit HW sorters more flexible and useful.

Index Term — HW/SW co-design, Reusable & Embedded Cores, Sorting, Radix Sort, Technology Independent Methodologies, System-on-a-Chip (SoC), VLSI design.

1. INTRODUCTION

Sorting is one of the most important problems in computer science. Many fundamental processes in computing and communication systems require sorting of data. Sorting network play a key role in the areas of parallel computing, multi-access memories and multiprocessing [3], [4], [5], [6], [11], [13], [14], [19].

Compare and swap elements of data are vital for sorting, as depicted in Fig. 1. But if someone needs to process very long digit integer sorting, then directly design a corresponding digit integer hardware sorter, the comparators and networks will become very huge. The circuit schematics of 1, 2, 4, and 16 bit magnitude comparators are depicted in Fig. 2. And about bus, if it is designed for 32-digit integer, every bus represents 32-bit line. And if it is designed for 64-digit integer, every bus represents 64-bit line. That means it needs double wire structures and areas.

More importantly, circuit cost/complexity of a (2k)-bit comparator are not only twice than a k-bit comparator, as shown in Table 1. Also, the ability of CMOS circuit fan-out is limited; it still needs to add some additional buffers in the comparator circuits.

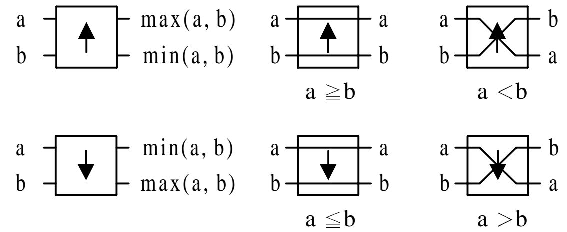
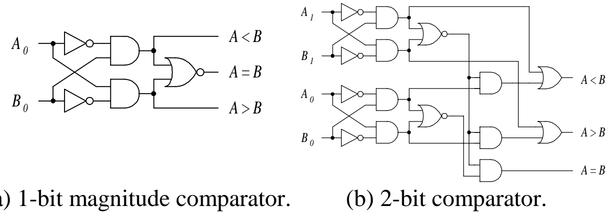
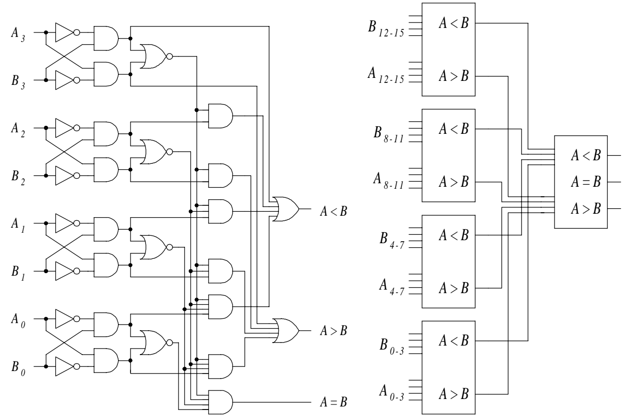


Figure 1. Compare & swap elements are vital for sorting



(a) 1-bit magnitude comparator. (b) 2-bit comparator.



(c) 4-bit comparator. (d) 16-bit comparator.

Figure 2. The Circuit of magnitude comparators.

magnitude comparator	1-bit	2-bit	4-bit	16-bit
CMOS Cost (gate count)	12 P + 12 N	39 P + 39 N	87 P + 87 N	399P + 399N

Table 1. Circuit cost/complexity of a long bit/digit comparator are more higher than a short bit/digit one.

m number single sorter chip design	Function blocks of one-sorter cell	Cost of each functional block (CMOS transistor gate count)	Number of cells	Number of clock cycles
16-bit Enumeration Sorter [23] (1982)	Two 16-bit data registers One 16-bit comparator One 8-bit counter	256P + 256N 399P + 399N 136P + 136N	m	2 N
16-bit VLSI Sorter [16] (1983)	Two 16-bit data registers One 16-bit comparator Two 16-bit 2-way multiplexers	256P + 256N 399P + 399N 64P + 64N	m / 2	4 N
16-bit Rebound Sorter [8] (1978 [8], 1989 [2])	Two 8-bit data registers One 8-bit comparator Two 8-bit 2-way multiplexers	128P + 128N 195P + 195N 32P + 32N	m	2 N
16-bit Bit-Serial Sorter [1] (1991)	One 1-bit comparator One 16-bit shift register Two 1-bit 2-way multiplexers Two 1-bit delay elements	12P + 12N 452P + 452N 4P + 4N 2P + 2N	m	N + 1

Table 2. Chip Comparison of m 16-bit hardware sorter designs.

In Table 2, some sorter chip designs had shown hardware expandable properties [1], [2], [8]. But they are not good enough for arbitrary long digit integer sorter design. The time performance of a fixed-digit (k-bit) hardware sorter is often better than a same digit software sort program, as displayed in Table 3. But a pure hardware sorter still has higher area cost and some restrictions, so it is not popular yet on common commercial CPUs.

Base on the physical considerations, the author focuses on effectively solving arbitrary long digit integer sort problem by HW/SW co-design under Area-Time^2 (AT^2) cost-performance trade-off constraint [20], [21]. Several AT^2-optimal sorting networks under different word length models have been proposed in [7], [9], [15], and [17].

For embedded systems, a uniprocessor software solution is often not applicable due to the insufficient I/O and performance, while realizing multiprocessor sorting methods on parallel computers is much too expensive with respect to area cost and power consumption.

When the trends of data processing migrate from 32-bit to 64-bit, 128-bit or uncertainly higher, a fixed-digit pure HW sorter cannot content demands alone. All of the sorting algorithms or circuits in this paper are based on commonly known algorithms and structures. But make an accomplished hardware sorter reusable [12], make a pure HW sorter more flexible and balance its cost-performance, are very valuable and necessary.

This paper is organized as follows. Section 2 briefly introduces the basic LSD radix sort algorithm. Then a cost-benefit balanced multi-level (two-level) HW/SW mixed sort architecture is given and discussed in Section 3. Finally conclude the major findings and outline the future work.

Design	Area Perf. (A)	Time Perf. (Td)
Uniprocessor Heapsort	log N	$N (\log_2 N)^2$
$(1 + \log_2 N)$ – processor Mergesort	$(\log_2 N)^2$	$N \log_2 N$
$(\log_2 N)^2$ – processor Bitonic Sort	$(\log_2 N)^3$	N
N–processor Bitonic Sort on Mesh	$N (\log_2 N)^2$	sqrt(N)
N–processor Bitonic Sort on Shuffle-Exchange Net [19]	$N^2 / (\log_2 N)^2$	$(\log_2 N)^3$
$N (\log_2 N)^2$ – Comparator Bitonic Sort	$N^2 / \log_2 N$	$(\log_2 N)^2$
N^2 comparators Bubble Sort	$N \log_2 N$	N

Table 3. Area-Time Bounds for the finite and fixed bit/digit number sorting problem [21].

II. STRAIGHT RADIX SORT ALGORITHM

This approach begins with the least significant key first, and is known as LSD (Least Significant Digit) sort. Following the sort on a key, the piles are put together to obtain a single pile that is then sorted on the next significant key. This process is continued until the pile is sorted on the most significant key [13]. And the sorted sequence is obtained.

Complexity As shown in Fig. 3, it takes $\boldsymbol{n}$ steps to put all the elements in queue AUX, and $\boldsymbol{d}$ steps to initialize the queues Q[i]. The main loop of the algorithm, which is executed $\boldsymbol{m}$ times, pops each element from AUX and pushes it into one of the Q[i]s.

It also concatenates all the Q[i]s together. So the overall running time of the algorithm is $O(\boldsymbol{m}\ n)$. But if $\boldsymbol{m}$ is limited or small, it can be ignored. So the time complexity of the algorithm is $O(n)$, this is a common condition under a common CPU.

Algorithm **Straight_Radix_Sort** (A[ ], ***n***, ***k***)
(* **Input**: A[ ](an array of integer, each with ***k*** digits, in the rage 1 to ***n***).
Output: A[ ] (the array in sorted order). *)
begin
Assume that all elements are initially in a auxiliary queue AUX;
(* The use of AUX is for simplicity; it can be implemented by Array A *)

for *i:= 1* **to** *d* **do**
(* *d* is the possible digits; *d* = 10 in case of decimal numbers *)
Initialize queue Q[*i*] *to be empty;*

for *i:=* ***k*** **downto** d **do**
while *AUX is not empty* **do**
Pop x from AUX;
d := the i-th digit of x;
Insert x into Q[*d*]*;*

for *j:= 1* **to** *d* **do**
Insert Q[*j*] *into AUX;*

for *i:= 1* **to** ***n*** **do**
Pop A[*i*] *from AUX;*

end.

A Radix -10 Sorting Example:

232, 321, 213, 231, 111, 112, 132, 123, 221
1S → 321, 231, 111, 221
2S → 232, 112, 132
3S → 213, 123

321, 231, 111, 221, 232, 112, 132, 213, 123
10S → 111, 112, 213
20S → 321, 221, 123
30S → 231, 232, 132

111, 112, 213, 321, 221, 123, 231, 232, 132
100S → 111, 112, 123, 132
200S → 213, 221, 231, 232
300S → 321

Result: 111, 112, 123, 132, 213, 221, 231, 232, 321

Figure 3. Basic straight radix sort algorithm and a radix-10 sorting example.

III. A MULTI-LEVEL MIXED ARCHITECTURE: SUPER RADIX SORT

Figure 3 also displays an LSD radix-10 sorting example using linked allocation [9]. But when the radix is very large, linked list allocation will become ineffective.

From this example, the benefits of LSD radix sort are directly unfolded: (1) the key size can be changed easily; (2) there is no recursive function call, no stack size problem. For solving arbitrary long digit integer sorting problem under cost-performance trade-off constraint, the LSD radix benefits will be extended to the utmost edge.

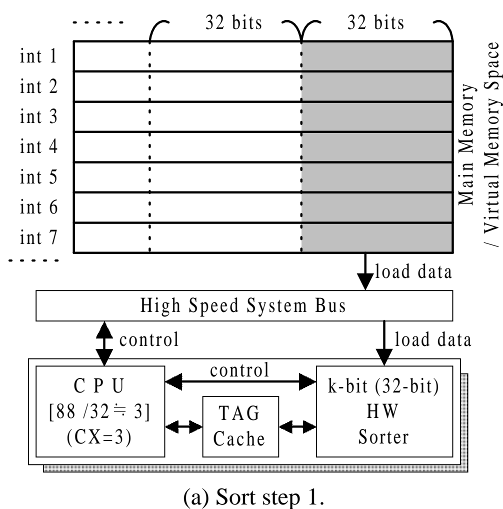
And because of very long digit integer, using bit field structure to reduce memory requirement, and accelerate sort process, is necessary. As depicted in Fig. 4, a two-level HW/SW mixed sort architecture are proposed: an accomplished fixed-digit (k-bit) hardware sorter implements the first/bottom level sorting, software programmed LSD-radix (radix 2^k) sort implements the second/higher level sorting by way of CPU. Thus sort operation will appear in assembly codes, as Fig. 5 shows.

It can directly handle maximum $2^{32} \times$ k-digit integers sorting job (if 32 is the length of common register). If k=16, it can handle max $2^{32} \times 16$ digit integer sorting job. If the number of digit is still higher then the quota, similar multi-level mixed sort architecture can be considered. Of course, if the input sequence is also arbitrary long, some special design have provided solutions [24]. Or the sequence is separated into several pieces, and then merges them to get the total result after sorting.

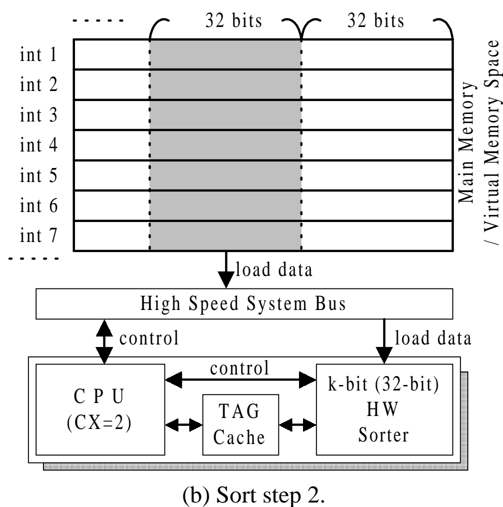
Because the bit length of numbers is very long, compare two numbers than directly swap them is very ineffective [18]. An indirect method -- only record swapped indices and hold them in cache is a good idea.

If the system only has an common CPU and the bits of the longest number is *m*, and the sort algorithm is radix sort, the average overall running time of the proposed method is $\boldsymbol{m} \times O(\text{N})$. But if the system has an accomplished fixed-digit (k-bit) hardware sorter on the system and the bits of the longest number is *m*, the overall running time of the proposed method becomes $\boldsymbol{m}$ / k × Td. If the HW sorter is N $(\log_2 \text{N})^2$ – Comparator Bitonic Sorter, the overall running time is $\boldsymbol{m} / \text{k} \times O((\log_2 \text{N})^2)$. Some comparisons are shown in Table 4.

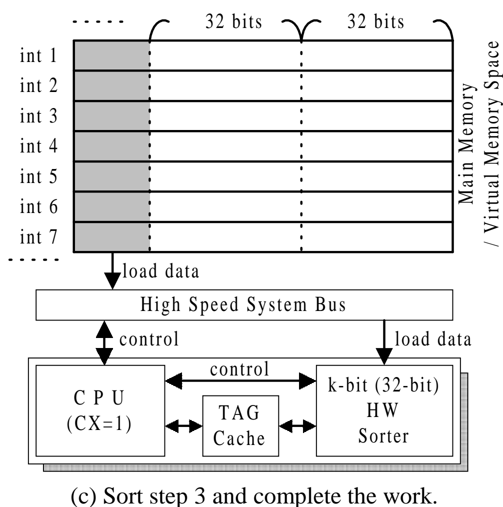
The proposed HW/SW mixed super radix sorting architecture can process and change HW/SW partitioning ratio easily, as displayed in Fig. 6, to get a cost-benefit balanced flexible HW/SW mixed design. And the accomplished fixed-digit (k-bit) hardware sorter can choose any your favor or your own design.



(a) Sort step 1.



(b) Sort step 2.



(c) Sort step 3 and complete the work.

Figure 4. Two-level HW/SW mixed sort architecture.

```
DATA SEGMENT
   NUM   DB  ............
   ...............
DATA ENDS
..........................

CODE SEGMENT
   ..........................
SORT_START:
   MOV   ESI, OFFSET NUM     ; source address
   MOV   EBX, 88D   ; digit of number = 88
   MOV   EDX, 9D    ; there are 9 numbers to be sorted
   SORT
    .........................
CODE ENDS
```

Figure 5. Why does not instruction SORT appear in instruction sets of nowadays?

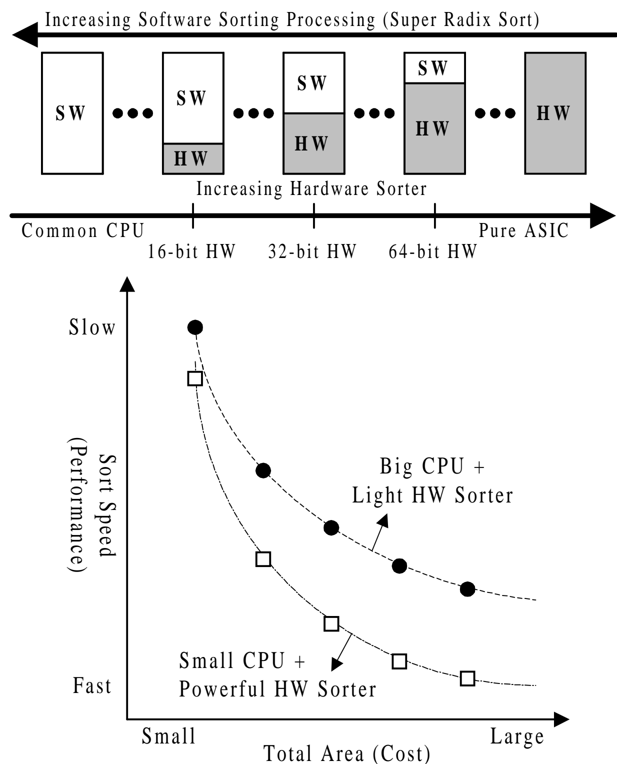


Figure 6. Impact and challenge of hardware/software co-design trade-off.

Figure 7 demonstrates a super radix sort: 88-digit integer SW LSD radix-4,294,967,296 (radix-2^{32}) sort with 32-bit HW sorter mixed sorting, it needs 3 steps. And it is processed by 88-digit integer SW LSD Radix-65,536 (Radix-2^{16}) sort with 16-bit HW sorter mixed sorting, it will needs 6 steps. If the hardware sorter can be easily decomposed to several stages then pipeline, the hardware sorter can get more higher hardware sharing and throughputs, as Fig. 8 depicts.

	Tag[X]	num[X][2]		num[X][1]		num[X][0]	
Origin:	[1]	00000110	0100000010010000	1010000000010100	0010101000000000	0001000000101000	0001010101010000
	[2]	10101000	0010101010101000	0000101010001010	0100000010101011	0000000000000010	0000000000000000
	[3]	00000000	0000010111110000	0000000000000000	0000000010101010	0000010101000000	0101100000001100
	[4]	00000011	0111100000000100	1100000011110000	0001110000010010	0000000000000000	0000000000000010
	[5]	00000000	0000000000000000	0000000000000000	0000010100001000	0000000000000000	0000001100110011
	[6]	00000000	0000000000000000	0000000000000001	0000100010000000	0000000000000100	0010100000010000
	[7]	00000000	0000000000000001	0001010000001000	0000000100101000	0000000101000000	0000000000111000
	[8]	00000000	0000000000000000	0000000100100000	0000001000010100	0010001000101000	0000001010000100
	[9]	00000000	0001000010101000	0000100001110000	0000100000011000	0000000000001111	0000000001100000
Step 1:	[4]	00000011	0111100000000100	1100000011110000	0001110000010010	0000000000000000	0000000000000010
	[5]	00000000	0000000000000000	0000000000000000	0000010100001000	0000000000000000	0000001100110011
	[2]	10101000	0010101010101000	0000101010001010	0100000010101011	0000000000000010	0000000000000000
	[6]	00000000	0000000000000000	0000000000000001	0000100010000000	0000000000000100	0010100000010000
	[9]	00000000	0001000010101000	0000100001110000	0000100000011000	0000000000001111	0000000001100000
	[7]	00000000	0000000000000001	0001010000001000	0000000100101000	0000000101000000	0000000000111000
	[3]	00000000	0000010111110000	0000000000000000	0000000010101010	0000010101000000	0101100000001100
	[1]	00000110	0100000010010000	1010000000010100	0010101000000000	0001000000101000	0001010101010000
	[8]	00000000	0000000000000000	0000000100100000	0000001000010100	0010001000101000	0000001010000100
Step 2:	[3]	00000000	0000010111110000	0000000000000000	0000000010101010	0000010101000000	0101100000001100
	[5]	00000000	0000000000000000	0000000000000000	0000010100001000	0000000000000000	0000001100110011
	[6]	00000000	0000000000000000	0000000000000001	0000100010000000	0000000000000100	0010100000010000
	[8]	00000000	0000000000000000	0000000100100000	0000001000010100	0010001000101000	0000001010000100
	[9]	00000000	0001000010101000	0000100001110000	0000100000011000	0000000000001111	0000000001100000
	[2]	10101000	0010101010101000	0000101010001010	0100000010101011	0000000000000010	0000000000000000
	[7]	00000000	0000000000000001	0001010000001000	0000000100101000	0000000101000000	0000000000111000
	[1]	00000110	0100000010010000	1010000000010100	0010101000000000	0001000000101000	0001010101010000
	[4]	00000011	0111100000000100	1100000011110000	0001110000010010	0000000000000000	0000000000000010
Step 3:	[5]	00000000	0000000000000000	0000000000000000	0000010100001000	0000000000000000	0000001100110011
	[6]	00000000	0000000000000000	0000000000000001	0000100010000000	0000000000000100	0010100000010000
	[8]	00000000	0000000000000000	0000000100100000	0000001000010100	0010001000101000	0000001010000100
	[7]	00000000	0000000000000001	0001010000001000	0000000100101000	0000000101000000	0000000000111000
	[3]	00000000	0000010111110000	0000000000000000	0000000010101010	0000010101000000	0101100000001100
	[9]	00000000	0001000010101000	0000100001110000	0000100000011000	0000000000001111	0000000001100000
	[4]	00000011	0111100000000100	1100000011110000	0001110000010010	0000000000000000	0000000000000010
	[1]	00000110	0100000010010000	1010000000010100	0010101000000000	0001000000101000	0001010101010000
	[2]	10101000	0010101010101000	0000101010001010	0100000010101011	0000000000000010	0000000000000000

Step 1:
Input Sequence: A[1][0], A[2][0], A[3][0], A[4][0], A[5][0], A[6][0], A[7][0], A[8][0], A[9][0]
After HW Sorting: A[4][0], A[5][0], A[2][0], A[6][0], A[9][0], A[7][0], A[3][0], A[1][0], A[8][0]
ONLY Record swapped index: 4, 5, 2, 6, 9, 7, 3, 1, 8.

Step 2:
Input Sequence: A[4][1], A[5][1], A[2][1], A[6][1], A[9][1], A[7][1], A[3][1], A[1][1], A[8][1]
After HW Sorting: A[3][1], A[5][1], A[6][1], A[8][1], A[9][1], A[2][1], A[7][1], A[1][1], A[4][1]
ONLY Record swapped index: 3, 5, 6, 8, 9, 2, 7, 1, 4.

Step 3:
Input Sequence: A[3][2], A[5][2], A[6][2], A[8][2], A[9][2], A[2][2], A[7][2], A[1][2], A[4][2]
After HW Sorting: A[5][2], A[6][2], A[8][2], A[7][2], A[3][2], A[9][2], A[4][2], A[1][2], A[2][2]
ONLY Record swapped index: 5, 6, 8, 7, 3, 9, 4, 1, 2.
The final index is the answer.
* Swap the original whole number in these sorting steps is unnecessary.

Figure 7. Super Radix Sort: 88-digit integer SW LSD radix- 4,294,967,296 (radix- 2^{32}) sort with 32-bit HW sorter mixed sorting.

HW / SW	Pure Common CPU / Radix Sort	Pure Common CPU / Quick Sort [11]	CPU & One 32-bit HW Sorter* / Super Radix Sort	CPU & One 64-bit HW Sorter* / Super Radix Sort	CPU & One 256-bit HW Sorter* / Super Radix Sort
Running time (Avg. case)	$\boldsymbol{m} \times O(N)$	$\boldsymbol{m} \times O(N \log N)$	$\boldsymbol{m} / 32 \times$ Td.	$\boldsymbol{m} / 64 \times$ Td.	$\boldsymbol{m} / 256 \times$ Td.

* If the HW sorter is an N $(\log_2 N)^2$ – comparator bitonic processor, the order of Td is $O((\log_2 N)^2)$.

Table 4. The performance order comparison between original architectures and new mixed architectures.

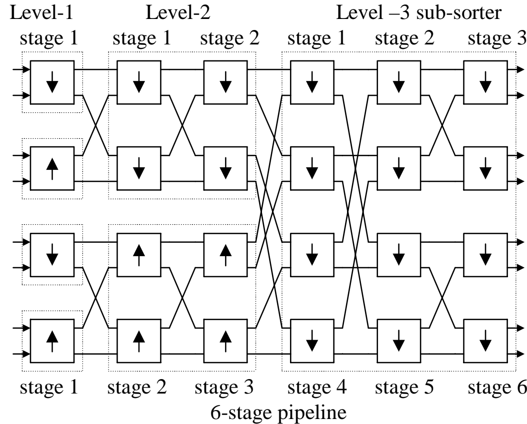


Figure 8. A three-level bitonic sorter. Pipeline this type circuit can get higher hardware sharing and throughputs.

Parameter	Old design	New design
Range	Fixed *k*-digit	$2^{32} \times k$ - digit
Layout cost	1	No change (Slight modification)
Hardware reusing	0	High

Table 5. New design has high hardware reusing.

IV. Concluding Remark

This paper discusses on effectively solving arbitrary long digit integer sorting problem by HW/SW co-design under Area x Time^2 (AT^2) price-performance constraint. The work introduces a two-level (multi-level) sort architecture can attain the object: an accomplished fixed-digit (k-digit) hardware sorter implements first level sorting, software programmed LSD radix (radix 2^k) sort implements second level sorting.

As Table 5 shows, by HW/SW co-design and reuse methodology, the proposed mixed super radix sorting architecture makes accomplished hardware sorters more flexible and useful: It is time to put a hardware sorter on a common commercial CPU or network processor.

References

[1] M. Afghahi, "A 512 16-b Bit-serial Sorter Chip," *IEEE J. Solid-State Circuits*, vol. 26, pp. 1452–1457, Oct. 1991.

[2] B. Ahn and J. M. Murray, "A Pipelined, Expandable VLSI Sorting Engine Implemented in CMOS Technology," in *Proc. IEEE Int'l. Symp. on Circuits and Systems*, 1989, pp. 134–137.

[3] S. G. Akl, *Parallel Sorting Algorithms*. Reading, New York: Academic Press, 1985.

[4] K. E. Batcher, "Sorting Networks and Their Applications," in *Proc. AFIPS 1968 Spring Joint Computer Conference*, pp. 307–314, Apr. 1968.

[5] G. Baudet and D. Stevenson, "Optimal Sorting Algorithms for Parallel Computer," *IEEE Trans. Computers*, vol. 27, pp.84–87, Jan. 1978.

[6] R. Beigel and J.Gill, "Sorting n Objects with a k-sorter," *IEEE Trans. Computers*, vol. 39, pp.714–716, May 1990.

[7] G. Bilardi and F. P. Preparata, "A Minimum Area VLSI Network for *O*(log n) Time Sorting," *IEEE Trans. Computers*, vol. 34, pp.336–343, May 1985.

[8] T. C. Chen, Vincent Y. Lum, and C. Tung, "The Rebound Sorter: An Efficient Sort Engine for Large File," in *IEEE Proc. 4th Int'l Conf. on Very Large Data Bases*, pp. 312–318, Sep. 1978.

[9] R. Cole and A. R. Seigel, "Optimal VLSI Circuits for Sorting," *JACM*, vol. 35, pp.777-809, 1988.

[10] Edward. H. Friend, "Sorting on Electronic Computer Systems," *JACM*, vol. 3, pp.134-168, 1956.

[11] C. A. R. Hoare, "Quicksort," *Computing Journal*, vol. 5, pp. 10–15, 1962.

[12] M. Keating and P. Bricaud, *Reuse Methodology Manual.* Reading: Kluwer, 1998.

[13] D. E. Knuth, *The Art of Computer Programming, Vol 3: Sorting and Searching*. Reading: Addison-Wesley, 1973.

[14] J.-G. Lee and B.-G. Lee, "Realization of Large-scale Distributors Based on Batcher Sorters," *IEEE Trans. Communications*, vol. 47, pp. 1103–1110, July 1999.

[15] T. Leighton, "Tight Bounds on The Complexity of Parallel Sorting," *IEEE Trans. Computers*, vol. 34, pp. 344–354, Apr. 1985.

[16] G. S. Miranker, Luong Tang, and Chak-Kuen Wong, "A 'Zero-Time' VLSI Sorter," *IBM J. Research & Development*, vol. 27, pp. 140–148, Mar. 1983.

[17] S. Olariu, M. C. Pinotti, and S. Q. Zheng, "How to Sort N Items Using a Sorting Network of Fixed I/O Size," *IEEE Trans. Parallel and Distributed Sys.*, vol. 10, pp 487–499, May 1999.

[18] B. Parhami and D.-M. Kwai, "Data-driven Control Scheme for Linear Arrays: Application to a Stable Insertion Sorter," *IEEE Trans. Parallel and Distributed Sys.*, vol. 10, pp 23–28, Jan. 1999.

[19] H. S. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE Trans. Computers*, vol. 20, pp.153–161, Feb. 1971.

[20] C. D. Thompson, "Area-Time Complexity for VLSI," in *Proc. 11th Annual ACM Symp. on Theory of Comp.*, pp. 81–88, Apr. 1979.

[21] C. D. Thompson, "The VLSI Complexity of Sorting," *IEEE Trans. Computers*, vol. 32, pp.1171–1184, Dec. 1983.

[22] N. H. E. Weste and K. Eshraghian, Principle of CMOS VLSI Design, 2nd Ed., Reading: Addison–Wesley, 1993.

[23] H. Yasuura, N. Takagi, and S. Yajima, "The Parallel Enumeration Sorting Scheme for VLSI," *IEEE Trans. Computers*, vol. 31, pp.1192–1201, Dec. 1982.

[24] S. Q. Zheng, S. Olariu, and M. C. Pinotti, "A Systolic Architecture for Sorting an Arbitrary Number of Elements," in *Proc. 1997 3rd Int. Conf. Algorithms and Architectures for Parallel Processing*, pp. 113 -126, 1997.