A Bus Architecture for Crosstalk Elimination in High Performance Processor Design

Wen-Wen Hsieh
Department of Computer Science
National Tsing Hua University
HsinChu, Taiwan 300, R.O.C
wwhsieh@cs.nthu.edu.tw

Po-Yuan Chen
Department of Computer Science
National Tsing Hua University
HsinChu, Taiwan 300, R.O.C
pychen@cs.nthu.edu.tw

TingTing Hwang
Department of Computer Science
National Tsing Hua University
HsinChu, Taiwan 300, R.O.C
tingting@cs.nthu.edu.tw

ABSTRACT

In deep sub-micron technology, the crosstalk effect between adjacent wires has become an important issue, especially between long on-chip buses. This effect leads to the increase in delay, in power consumption, and in worst case, to incorrect result. In this paper, we propose a de-assembler/assembler structure to eliminate undesirable crosstalk effect on bus transmission. By taking advantage of the prefetch process where the instruction/data fetch rate is always higher than instruction/data commit rate in high performance processors, the proposed method would hardly reduce the performance. In addition, the required number of extra bus wires is only 7 as compared with 85 needed in [6] when the bus width is 128 bits.

Categories and Subject Descriptors

B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; C.1 [**Processor Architecture**]: Miscellaneous

General Terms

Performance, Design

Keywords

Crosstalk, Architecture, Instruction/Data Bus, High Performance

1. INTRODUCTION

In deep sub-micron technology, coupling capacitance between interconnects is the dominant factor in the total wire capacitance. It derives from one signal and its neighboring wire switching at different directions. This effect, *crosstalk*, will lead to additional delay and power consumption of a signal. Even worse, in some cases, it may cause malfunction of a circuit. Thus, elimination of crosstalk has become a very important design issue. Since in a bus structure, a number of wires are laid in parallel for a long distance, the crosstalk problem in a bus structure is especially salient.

Two major categories of crosstalk elimination approaches have been proposed. One category is designed for power consumption and its objective is to minimize the total crosstalk in all wires. Pervious work such as spacing and shielding [1] is two famous approaches in this category. Other approaches such as [2, 3, 4] are also designed to reduce the total crosstalk. Another category is designed for performance and its objective is to minimize the maximum crosstalk effect among all wires. Kuo et al. [5] proposed techniques at post-compiler level for performance improvement. In addition, [6] and [7] use bus-encoding methods to achieve this goal. Both of them [6, 7] proposed encoding data to be crosstalk free before it is transmitted on buses. At receiving end of the bus, a decoder logic decodes the data into the original one. In this paper, we will focus on the second problem, i.e., the elimination of certain data transmission patterns so that the maximum crosstalk effect is minimized. In this regard, Victor et al. [6] proved theoretically that the maximum wire number for encoding n-bits bus is $\lfloor \log F_{n+2} \rfloor$ where F_n is the n_{th} number of Fibonacci sequence. These bus encoding methods become impractical when the number of bus lines become large. For example, a 128-bit bus will be encoded with 171 wires in theory and with 213 wires in practice. For a high performance processor like superscalar and VLIW architecture, the width of a bus is usually wide. Using methods as such are not appropriate.

In this paper, a new bus structure is proposed for wide bus architecture in high-performance processors. To hide memory latencies, a common technique used in high-performance processors is prefetching. This technique is to prefetch instructions or data into buffers before they are used by the processors. By inserting a de-assembler and an assembler at the sending and receiving ends of the bus, respectively, certain transmission patterns that cause undesirable crosstalk can be eliminated. Moreover, our method takes advantage of the prefetch process where the instruction/data fetch rate is always higher than instruction/data commit rate. Therefore, in our approach there is almost no penalty in terms of dynamic instruction count.

The rest of this paper is organized as follows. Section 2 describes the crosstalk model. Section 3 gives our motivation. Section 4 presents our novel bus architecture. Section 5 shows the experiment results. Finally, Section 6 concludes this paper.

CODES+ISSS'06, October 22–25, 2006, Seoul, Korea.

2. CROSSTALK MODEL

There are two kinds of capacitance with which a single wire is associated. One is the capacitance C_{ground} between the wire and ground, and the other is the coupling capacitance C_{couple} between the wire and its neighboring wires. The total capacitance C_{total} of a signal wire is calculated as follows.

$$C_{total} = C_{ground} + n \times C_{couple}, \quad 0 \leq n \leq 4$$

where n depends on the types of coupling of its neighboring wires. A more detailed analysis of C_{total} on delay can be found in [9].

Table 1: The bit pattern of different crosstalk types.

crosstalk type	time	bit pattern (w_{i-1}, w_i, w_{i+1})
1C	T_{t-1}	(b,b,b) $(b,b,\bar{b})$ $(b,\bar{b},\bar{b})$ $(\bar{b},\bar{b},b)$
	T_t	$(b,\bar{b},\bar{b})$ $(\bar{b},\bar{b},b)$ (b,b,b) (b,b,b)
2C	T_{t-1}	(b,b,b) $(\bar{b},b,b)$ $(b,b,\bar{b})$ $(\bar{b},b,\bar{b})$ $(b,b,\bar{b})$ $(\bar{b},b,b)$
	T_t	$(b,\bar{b},b)$ $(\bar{b},\bar{b},b)$ $(b,\bar{b},\bar{b})$ $(\bar{b},\bar{b},\bar{b})$ $(\bar{b},\bar{b},b)$ $(b,\bar{b},\bar{b})$
3C	T_{t-1}	$(b,\bar{b},b)$ $(b,\bar{b},b)$ $(\bar{b},\bar{b},b)$ $(b,\bar{b},\bar{b})$
	T_t	$(b,b,\bar{b})$ $(\bar{b},b,b)$ $(\bar{b},b,\bar{b})$ $(\bar{b},b,\bar{b})$
4C	T_{t-1}	$(b,\bar{b},b)$
	T_t	$(\bar{b},b,\bar{b})$

The coupling capacitance of a wire can be classified into four types *1C*, *2C*, *3C* and *4C* according to the C_{couple} of two wires [7]. Let the crosstalk effect on a single wire *(victim)* depends on the signal transition of its neighboring wires *(aggressors)*. We use a tri-tuple (w_{i-1}, w_i, w_{i+1}) to represent the wire signal pattern at a certain time, where w_i represents the victim while w_{i-1} and w_{i+1} are aggressors. Table 1 shows the relations between crosstalk type and the wire signal transition at time T_{t-1} and time T_t, where (b, $\bar{b}$) $\in$ {0, 1} and $\bar{b}$ is the complement of b. Figure 1 shows the *4C* crosstalk examples on three wires w_{i-1}, w_i, and w_{i+1}. The signal patterns transmitted on the wires are *(1, 0, 1)* at time T_{t-1} and *(0, 1, 0)* at time T_t in Figure 1(a), while *(0, 1, 0)* at time T_{t-1} and *(1, 0, 1)* at time T_t in Figure 1(b).

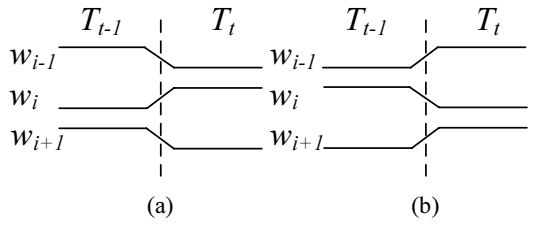

Figure 1: The examples of *4C* crosstalk sequence.

Note that the transmission of a pattern (b,b,b) followed by any other patterns would never cause signals on adjacent wires switching at different direction, since the signals in pattern (b,b,b) are the same. Therefore, transmission the pattern with all 0's (or all 1's) followed by any other pattern will never incur *3C*/*4C* crosstalk.

3. MOTIVATION

In order to study 1) the relationship between instruction/data fetched rate and instruction/data committed rate and 2) the percentage of *3C* and *4C* crosstalk patterns incurred in bus transmission, the transmission on the instruction bus for the DSPstone benchmark is profiled. Experiments were performed by using Simplescalar 3.0 [10], and the out-of-order 4-issue superscalar architecture is used to simulate the speculative fetching.

Figure 2 is the percentage of committed instructions to the total fetched instructions for different examples in the benchmark set. It shows that the number of committed instructions is only about 30% to 40% of the total number of fetched instructions for all example. In other words, the instruction fetch rate is much higher than instruction commit rate in bus transmission.

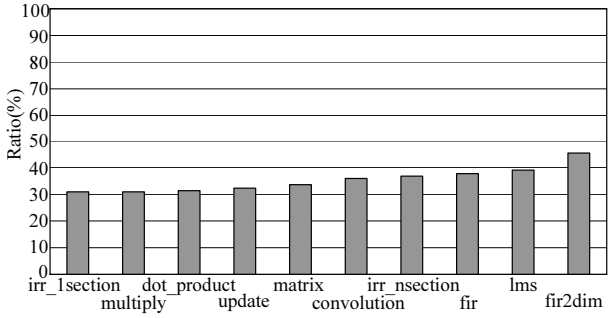

Figure 2: The percentage of instruction committed.

Table 2 is the second profiling result. The column labeled *bits of instruction* gives the total bit number of fetched instructions, and the column labeled *bits of 3C and 4C* is the bit number of *3C* and *4C* crosstalk patterns. The column labeled *ratio of 3C and 4C* shows the ratio of *3C* and *4C* crosstalk bits to the total fetched bits. From the table, we know that the ratio of *3C* and *4C* crosstalk patterns is very low.

Table 2: The percentage of *3C* and *4C* patterns.

benchmark	bits of instruction	bits of *3C* and *4C*	ratio of *3C* and *4C*(%)
multiply	180736	6430	3.6
update	576480	20256	3.5
convolution	168192	5914	3.5
dit_product	108256	4070	3.8
fur2dim	195296	7500	3.8
fir	134016	5048	3.8
irr_nsection	301440	10698	3.5
irr_1section	197600	7120	3.6
matrix	107424	3983	3.7
lms	2036064	73427	3.6

Since *3C* and *4C* types of crosstalk take only a small portion of the total transmitted data but cause serious delay penalty, we propose a de-assembler and an assembler structure on both ends of the bus to eliminate these two types of crosstalk.

4. THE DE-ASSEMBLER AND ASSEMBLER TECHNIQUES

We develop a bus structure to de-assemble/assemble data on a bus such that *3C* and *4C* crosstalk patterns are eliminated. Figure 3 shows the overall architecture.

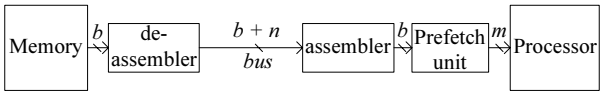

Figure 3: The overall architecture.

4.1 Basic Idea

In our technique, a bus is first partitioned into several channels, $channel_1$, $channel_2$...$channel_n$. Data transmitted on a channel is referred to as a *data segment* which is denoted as $data_{t,\ i}$, where t is the time stamp and i is the channel position index. Each *data segment* is regarded as a basic data transmission unit. Figure 4 illustrates how our de-assembling and assembling mechanisms work at cycle T_t.

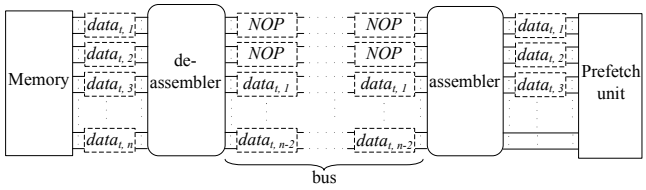

Figure 4: The sending end and the receiving end.

In Figure 4, $data_{t,\ 1}$ represents the *data segment* prepared to send on the first channel position in the current cycle, and $data_{t-1,\ 1}$ represents the *data segment* sent on the first channel position in the pervious cycle. The *data segments* $data_{t-1,\ i}$ sent in the pervious cycle T_{t-1} are stored in the registers in the de-assembler. At the beginning of sending data, the $data_{t,\ i}$ and the $data_{t-1,\ i}$ are checked to see if any *3C* or *4C* crosstalk incurs. If no *3C* or *4C* crosstalk is found, then $data_{t,\ i}$ is transmitted on the $channel_i$. Otherwise, the $data_{t,\ i}$ is shifted to the next one channel position $channel_{i+1}$ and a *data segment* with all 0's (or all 1's) called an *NOP segment* is inserted onto the $channel_i$. Once a $data_{t,\ i}$ is shifted to $channel_{i+1}$, it is required to be checked with $data_{t-1,\ i+1}$ to see if any crosstalk incurs between them. The checking continues until $data_{t,\ i}$ finds a position $channel_j$ where $data_{t,\ i}$ and $data_{t-1,\ j}$ incurs no crosstalk, or it reaches the last channel of the bus. Those *data segments* $data_{t,\ i}$ which are not able to be sent during this current cycle T_t due to the *NOP segments* insertion would be shifted to the next cycle T_{t+1}. For example, in Figure 4, assume $data_{t,\ 1}$ with *3C/4C* crosstalk occurs between $data_{t-1,\ 1}$ and $data_{t-1,\ 2}$. Then the $data_{t,\ 1}$ is shifted two channel positions and will be sent at position $channel_3$. Since the *data segments* are shifted two channel positions, $data_{t,\ n-1}$ and $data_{t,\ n}$ would be sent in the next transmission cycle T_{t+1}.

As to the assembler, it is required to remove all inserted *NOP segments* and pack the valid *data segments* to form the completed instructions as shown in Figure 4. After the packing, the assembler would inform the processor the number of completed instructions transmitted during the current cycle. Those *data segments* which cannot be packed into a complete instruction will be stored in a *buffer queue* waiting for the next assembling processing.

The worst case of transmission happens when the *3C* or *4C* crosstalk occurs between $data_{t,\ 1}$ and every *data segment* transmitted at cycle T_{t-1}. Thus, the bus is filled with all *NOP segments* at current cycle transmission. Since the *NOP segments* do not result in crosstalk with any other data patterns in the next transmission cycle, all data segments can be sent without incurring any *3C/4C* crosstalk patterns. Therefore, the worst case is to double the transmission cycles, that is, one cycle for *data segments* transmission and one cycle for *NOP segments* alternately.

4.2 Insertion of Separation Bits

Since crosstalk may occur across the boundary of two adjacent *channels*, shielding wires have to be inserted between every pairs of *channels*. Moreover, whether all 0's (or all 1's) pattern is an *NOP segment* or a real *data segment* requires a mechanism to make distinction. For these two purposes, our *separation bits*, s, are designed as follows.

We say a set of bit-patterns is a *crosstalk free cyclic* if any pairs of the patterns in the set does not incur *3C/4C* crosstalk. For example, a set of patterns, (*000*, *001*, *100*, *101*, and *111*) is a *crosstalk free cyclic*. Hence, in addition to acting as a state remembering bit, the *separation bit* must be designed to be a *crosstalk free cyclic*.The appropriate *separation bits* is chosen to form a *(|s|+2)*-bit *crosstalk free cyclic*, where $|s|$ is the length of *separation bits* and the 2 are the last bit of $data_{t,\ i}$ and the first bit of $data_{t,\ i+1}$.

Since we have 4 patterns for $data_{t,\ i}$ and $data_{t,\ i+1}$ combination and two more patterns to tell $data_{t,\ i}$ to be an *NOP segment* or a *data segment*, we need to find a set of codes which is *crosstalk free cyclic* and of size at least 6. For $|s| = 1$, the maximum size of its *crosstalk free cyclic* codes has only size of five (*000*, *001*, *100*, *101*, and *111*). These codes are not enough to accommodate 6 different patterns.

Table 3: The four possible choices of separate bits.

NOP segment = all 0's		NOP segment = all 1's	
s_{data}	s_{nop}	s_{data}	s_{nop}
10	00	00	10
11	01	01	11

Let the size of s be increased to 2. The maximum number of the *crosstalk free cyclic* codes is now over 6. In fact, for $|s| = 2$, there are more than one choices. Table 3 shows all possible choices. For example, when the *NOP segment* is designed to be all 0's pattern, two codes for s bits can be used. One is to have $s = 10$ for $data_{t,\ i}$ being a *data segment* and $s = 00$ for $data_{t,\ i}$ being an *NOP segment*. Similarly, if the *NOP segment* is designed to be all 1's pattern, two codes for s bits, *(00, 10)* and *(01, 11)* can be used. Figure 5 is an example of using all 0's pattern as the *NOP segment* and the selected codes for s are the *(10, 00)* pair. In this case, the first two patterns, *(0-1-0-0)* and *(0-1-0-1)*, at the left tell that $data_{t,\ i}$ is a real *data segment*, and the two patterns, *(0-0-0-0)* and *(0-0-0-1)*, at right tell that $data_{t,\ i}$ is an *NOP segment*. Moreover, the six patterns form a *crosstalk free cyclic*.

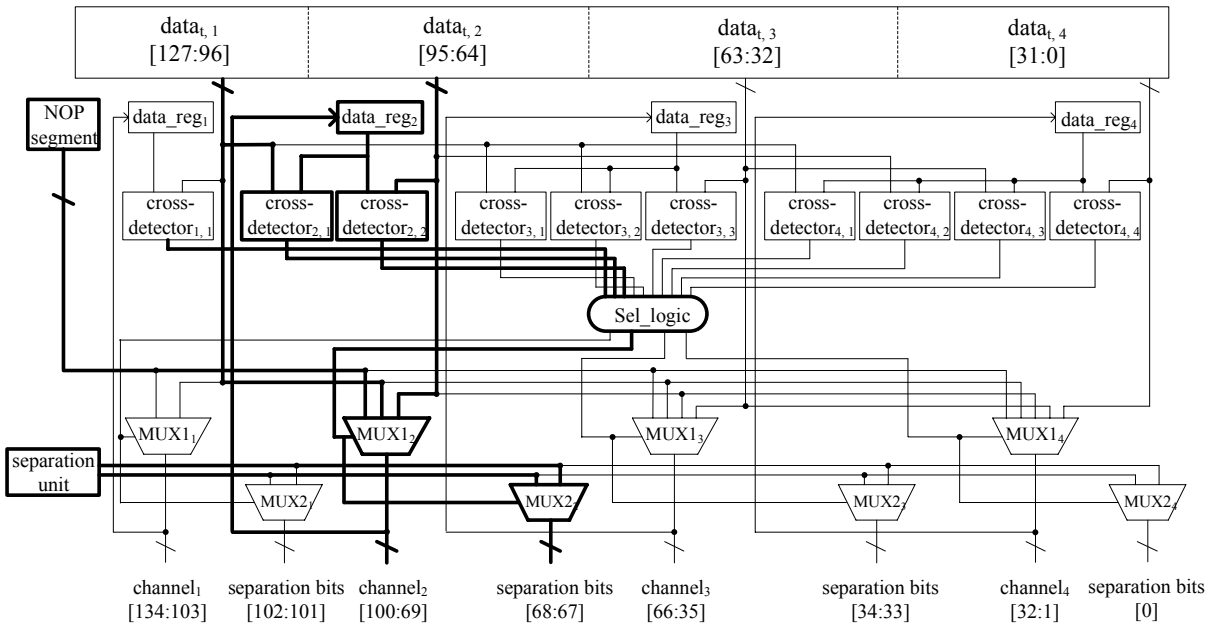

Figure 6: The de-assembler architecture.

data segment $data_{t,\,i}$	separation bits s	data segment $data_{t,\,i+1}$
· · · 0	1, 0	0 · · ·
· · · 0	1, 0	1 · · ·
· · · 1	1, 0	0 · · ·
· · · 1	1, 0	1 · · ·

$data_{t,\,i}$ is a data segment

data segment $data_{t,\,i}$	separation bits s	data segment $data_{t,\,i+1}$
· · · 0	0, 0	0 · · ·
· · · 0	0, 0	1 · · ·

$data_{t,\,i}$ is a NOP segment

Figure 5: One selection of separation bits.

Finally, one special condition is designed for the last channel position $channel_n$. Since the last channel has no adjacent channel $channel_{n+1}$, only one bit is required to decide whether the data sent on the last channel position is an *NOP segment* or not.

4.3 The De-assembler and Assembler Architectures

In order to check if any crosstalk occurs between *data segment* to be sent at current cycle and the *data segment* already sent at pervious cycle in parallel rather than in sequential, we design a parallel checking architecture. In this section, we describe our de-assembler and assembler architectures. The de-assembler architecture is shown in Figure 6. In this example, the width of the whole bus is 128 bits and the width of each channel is set to 32. Hence, the bits from 127 to 96 are grouped as $channel_1$, the bits from 95 to 64 are grouped as $channel_2$,... etc. and the total number of channels is 4.

To detect if a crosstalk occurs between the current *data segment*, $data_{t,\,i}$ and the data sent in $channel_i$ at pervious cycle, two logic elements named *data_reg* and *cross_detector* are designed. The $data\_reg_i$ is used to store the *data segment* sent on $channel_i$ at pervious cycle. For each $channel_i$, the $cross\_detector_{i,\,j}$, where j from 1 to i, is a combinational logic used to check if $data\_reg_i$ and $data_{t,\,j}$ induce crosstalk. Note that in order to check if $data_{t,\,i}$ can be sent on $channel_k$, for k from i to n in parallel, one or more $cross\_detector_{i,\,j}s$ are designed for each channel position $channel_i$. For a $data\_reg_j$, it is checked with all *data segment* $data_{t,\,i}$ to be sent, for i from 1 to j as shown in the Figure 6.

Next, all the output signals of the $cross\_detector_{i,\,j}s$ are sent to a logic element named *Sel_logic*. With inputs from all *cross_detector*s, *Sel_logic* will decide which *data segment* is to be sent on $channel_i$. Then, the output of *Sel_logic* is passed to the first level multiplexor, $MUX1_i$, where the inputs to $MUX1_i$ are $data_{t,\,j}$ for j from 1 to i and *NOP segment*. This multiplexor is used to select the *data segment* or *NOP segment* to be sent. Finally, the output of $cross\_detector_{i,\,j}s$ are also sent to the second level multiplexor, $MUX2_i$, which is used to determine what the *separation bits* are.

Now, taking $data_{t,\,2}$ as an example, two crosstalk detectors, $cross\_detector_{2,\,1}$ and $cross\_detector_{2,\,2}$, are used to detect if crosstalk occurs between $data\_reg_2$ and $data_{t,\,1}$, and between $data\_reg_2$ and $data_{t,\,2}$. The output of $cross\_detector_{1,\,1}$, $cross\_detector_{2,\,1}$ and $cross\_detector_{2,\,2}$ are sent to the *Sel_logic*. Then, the outputs of *Sel_logic* are used as the *select* signal of $MUX1_2$. The inputs to the $MUX1_2$ includes $data_{t,\,1}$, $data_{t,\,2}$ and *NOP segment*. Finally, the $MUX2_2$ is used to choose *separation bits*.

At the receiving side of the bus, an assembler is designed to remove the *NOP segments*. The architecture for the assembler is shown in Figure 7. The input of the assembler is a set of *data segments* with *separation bits* interleaving within them. First, a *DSel_logic* is constructed to determine which incoming data is *data segment* and which channel position be passed to. The inputs to *DSel_logic* contain two kinds of signals. One is the *separation bits* which record the information to distinguish a *data segment* from a *NOP segment*. The other is the number of *data segments* left unpacked at

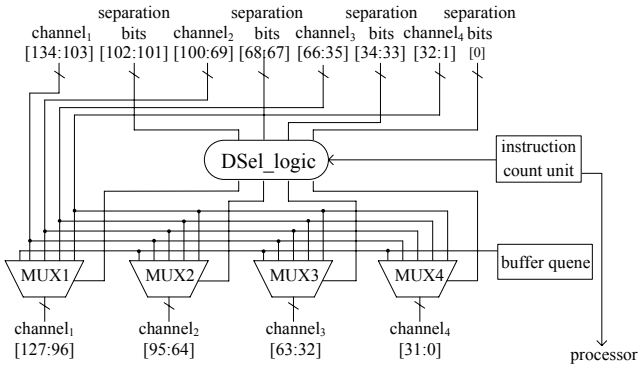

Figure 7: The assembler architecture.

previous cycle T_{t-1}. The output of *DSel_logic* is the number of channel position to be left shifted for each *data segment*, which is the *select* signal of the multiplexor, MUX_i. The input *data segment* to the MUX_i includes the *data segment* left in the buffer queue in the previous cycle and all incoming *data segments*.

Finally, an *instruction control unit* is designed to detect how many instructions are packed. The result of the *instruction control unit* is then sent to the processor.

5. EXPERIMENT RESULTS

In order to demonstrate the effectiveness and efficiency of our method, a set of experiments are conducted. The sim-outorder simulator from Simplescalar 3.0 [10] incorporated with our de-assembler and assembler architecture is used to simulate the out-of-order superscalar architecture without caches. We take instruction bus as the demonstration example. Each instruction is 32-bit long, and four instructions are issued in parallel so that the total bus width is 128 bits. We adopt DSPstone as the benchmarks.

The first experiment is to understand how many extra cycles are needed to execute a program. Table 4 shows the results. The columns labelled TCC and *pen* are the total cycle counts of the original circuit and the cycle penalty using our architecture, respectively. It can be seen that there is almost no cycle count overhead (less than 1%) for 8-bit, 16-bit, 32-bit channel sizes. In the worst case, the cycle count overhead is only 0.21% (dot_product when channel size is 32).

Table 4: The cycle count overhead for different channel size.

	TCC	channel size					
		8		16		32	
		pen	*ratio(%)*	*pen*	*ratio(%)*	*pen*	*ratio(%)*
dot_product	2355	0	0	3	0.13	5	0.21
fir2dim	12084	1	0.08	7	0.06	22	0.18
fir	3702	5	0.01	1	0.02	3	0.08
lms	4010	3	0.07	5	0.12	6	0.15
matrix	44360	4	0.01	27	0.06	11	0.02
matrix1x3	2841	2	0.07	3	0.11	5	0.18
average			0.04		0.08		0.14

The second experiment is to understand the extra wire overhead. The area overhead includes the extra wires required for separation bits and the area of the de-assembler/ assembler. Table 5 shows the comparisons of our results to Victor's memoryless approach [6]. Four cases for different channel sizes by using our method (4-bit, 8-bit, 16-bit and 32-bit per channel) and two cases (theoretical and practical) in Victor's paper are shown. The results show that when the number of bus width is getting wider, the effectiveness of our approach becomes more significant. For example, when the bus width is 128 and the channel size is 32, the number of extra wires using our method is only 7 as compared with 59 and 85 needed for the theoretical and practical cases, respectively, proposed in Victor's paper.

Table 5: The number of extra wires.

bus width	Ours				Victor's [6]	Victor's [6]
	Channel size					
	4	8	16	32	theoretical	practical
32	15	7	3	1	14	21
64	31	15	7	3	28	45
128	63	31	15	7	59	85

As to the area overhead for the de-assembler and the assembler, we choose the case of 128-bit bus width with 32-bit per channel for experiment. Two logic circuits are designed using Verilog and synthesized by the Synopsys Design Compiler. Table 6 shows the comparisons of our results to Victor's memoryless approach [6]. The gate count is obtained by synthesizing circuits using only NOR gates and inverters, and the area is synthesized with the TSMC 0.13μm cell library. The result shows that the de-assembler in our design takes more area than the encoder in Victor's approach [6]. This overhead is mainly from the logic for *crosstalk_detectors*. In addition, registers are needed in our approach because the de-assembler have to store the *data segments* transmitted in the pervious cycle.

Table 6: The area comparison of two designs.

method		Ours	Victor's[6]
De-assembler/ Encoder	gate count	3860	885
	area(μm)	8556.54	2359.39
	Num of register	128	0
Assembler/ Decoder	gate count	879	1402
	area(μm)	2053.854	3381.22
	Num of register	0	0

The third experiment is to see how much performance improvement can be obtained by eliminating *3C* and *4C* crosstalk. The result is simulated with Spice[12], and the case of 128-bit bus width with 32-bit per channel is taken. The values of capacitances for C_{ground} and C_{couple} in different technology, are obtained from the Berkeley predictive technology model (BPTM) [11]. Table 7 shows the simulation result. In this table, the first column gives the process technology (65nm, 90 nm). The second column gives different bus length (3mm and 5mm). The third column to the seventh column report the wire delay without and with crosstalk. The next two columns report the critical path delay for the de-assembler and assembler. All the delay information is normalized to the wire delay without crosstalk (*0C*). The last column reports the improvement ratio of our design, it is calculated by the formula

$$1 - \frac{2C\ wire\ delay\ +\ deassembler\ delay\ +\ assembler\ delay}{4C\ wire\ delay}$$

Table 7: The timing analysis of wire and the de-assembler/assembler.

tech	bus length	0C	1C	2C	3C	4C	deassembler	assembler	ratio(%)
90nm	3mm	1.00	1.26	3.00	5.38	7.42	2.80	1.45	2.23
	5mm	1.00	1.18	2.63	5.07	6.87	1.19	0.61	35.44
65nm	3mm	1.00	1.31	3.10	5.69	7.68	2.63	1.35	7.75
	5mm	1.00	1.13	2.13	4.45	6.16	0.99	0.51	41.07

From the table, we can see that the wire delay caused by $3C/4C$ crosstalk is serious. For example, the wire delay caused by $4C$ crosstalk is at least twice as the $2C$ crosstalk caused.(6.16 by $4c$ and 2.13 by $2C$ when bus length is 5mm in 65nm technology). It implies that we can shorten the clock cycle length to 50%. In addition, the extra delay caused by the de-assembler and assembler is less significant when the bus length increasing, therefore, the improvement rate achieved about 35% in 90nm technology and 41% in 65nm technology when bus length is 5mm as the table shown. We can predict that the improvement ratio will become more significant in deeper technology.

The last experiment is to understand the performance improvement rate for different channel sizes in different technologies. We take channel size is 16 and channel size is 32 as examples since the two channel sizes are more practical. The performance improvement rate is calculated as

$$improvement\_rate = \frac{new\_tcc \times rate}{orig\_tcc} \times 100\%$$

where ori_tcc and new_tcc are the total transmission cycle count of the original circuit and the new circuit, respectively, and $rate$ are the clock length reduction rate for 65nm and 90nm technologies. Figure 8 shows that the improvement rate for different cases can achieve 35% in 90nm technology and achieve about 40% in 65nm technology. It shows that the improvement rate of performance is less significant when the channel size is smaller. In addition, the improvement rate is getting higher when the process scales down.

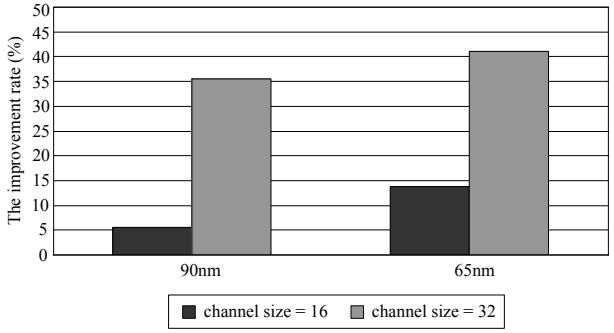

Figure 8: The improvement rate for different technologies.

6. CONCLUSION

In this paper, we have proposed a new bus structure to eliminate $3C/4C$ crosstalk effect during data transmission. By inserting a de-assembler and an assembler at the sending and receiving ends of the bus, respectively, certain transmission patterns that cause undesirable crosstalk can be eliminated. We take advantage of the prefetch process where the instruction/data fetch rate is always higher than instruction/data commit rate in high performance processors. According to the experimental results, our method achieves 40% in 65nm technology and more performance improvement-rate at the expand of a small number of extra wires as compared with the original design.

7. REFERENCES

[1] R. Arunachalam, E. Acar and S. R. Nassif, "Optimal Shielding/Spacing Metrics for Low Power Design," *IEEE Computer Society Annual Symposium on VLSI*, pp. 167-172, February 2003.

[2] J. D. Z. Ma, L. He, E. Acar, and S. R. Nassif, "Towards Global Routing With RLC Crosstalk Constraints," *Design Automation Conference*, pp. 669-672, June 2002.

[3] L. Li, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "A Crosstalk Aware Interconnect with Variable Cycle Transmission," *Design, Automation and Test in Europe Conference and Exhibition*, vol. 1, pp. 102-107, February 2004.

[4] S.-K. Wong and C.-Y. Tsui, "Re-configurable Bus Encoding Scheme for Reducing Power Consumption of the Cross Coupling Capacitance for Deep Sub-micron Instruction Bus," *Design, Automation, and Test in Europe Conference and Exhibition*, vol. 1, pp. 130-135, November 2004.

[5] W. A. Kuo, Y. L. Chiang, T. Hwang, and Allen C.-H. Wu, "Performance-Driven Crosstalk Elimination at Post-Compiler Level," *IEEE International Symposium on Circuits and Systems*, pp. 3041-3044, May 2006.

[6] B. Victor and K. Keutzer, "Bus encoding to prevent crosstalk delay," *IEEE/ACM International Conference on Computer Aided Design*, pp. 57-63, November 2001.

[7] C. Duan, A. Tirumala, and S. P. Khatri, "Analysis and Avoidance of Cross-talk in On-Chip Buses," *Hot Interconnects*, pp. 133-138, August 2001.

[8] C. Duan and S. P. Khatri, "Exploiting Crosstalk to Speed up On-Chip Buses," *Design, Automation and Test in Europe Conference and Exhibition*, pp. 778-783, February 2004.

[9] P. P. Sotiriadis and A. Chandrakasan, "Reducing Bus Delay in Submicron Technology Using Coding," *IEEE Asia and South Pacific Design Automation Conference*, pp. 109-114, January-February 2001.

[10] "http://www.simplescalar.com/"

[11] "http://www-device.eecs.berkeley.edu/~ptm"

[12] L. Nagel, "Spice: A computer program to simulate computer circuits," in Universiv of Culifomiu, Berkeley UCBERL Memo M520, May 1995.