논문 2010-47SD-1-10

SoC를 위한 새로운 플라잉 마스터 버스 아키텍쳐 구조의 제안과 검증

(Proposal of a Novel Flying Master Bus Architecture For System On a Chip and Its Evaluation)

이 국 표*, 강 성 준**, 윤 영 섭*

(Kook Pyo Lee, Seong Jun Kang, and Yung Sup Yoon)

요 약

고성능의 SoC를 구현하기 위해서, 우리는 버스 프로토콜과 상관없이 선택된 슬레이브에 직접 액세스하는 특별하게 정의된 마스터인 플라잉 마스터 버스 아키텍쳐 구조를 제안한다. 제안한 버스 아키텍쳐는 베릴로그와 하이닉스 0.18um 공정을 디자인 맵핑하여 실행하였다. 마스터와 슬레이브 래퍼는 150여개의 로직 게이트 카운트를 가지기 때문에, SoC 디자인에 있어서 모듈의 고유 영역인 면적용적은 여전히 고려해야 한다. TLM 성능분석 시뮬레이션을 통해 제안한 아키텍쳐가 기존의 버스아키텍쳐와 비교해서 트랜잭션 사이클이 25~40%, 버스 효율성이 43~60% 증가하였고, 요청 사이클이 43~77% 감소하였다. 결론적으로, 우리가 제안한 플라잉 마스터 버스 아키텍쳐 구조는 성능과 효율성의 측면에서 버스 아키텍쳐 분야를 선도할 주요 후보 중 하나라고 여겨진다.

Abstract

To implement the high performance SoC, we propose the flying master bus architecture that a specially defined master named as the flying master directly accesses the selected slaves with no regard to the bus protocol. The proposed bus architecture was implemented through Verilog and mapped the design into Hynix 0.18um technology. As master and slave wrappers have around 150 logic gate counts, the area overhead is still small considering the typical area of modules in SoC designs. In TLM performance simulation about proposed architecture, 25~40% of transaction cycle and 43~60% of bus efficiency are increased and 43~77% of request cycle is decreased, compared with conventional bus architecture. Conclusively, we assume that the proposed flying master bus architecture is promising as the leading candidate of the bus architecture in the aspect of performance and efficiency.

Keywords : SoC, performance improvement, flying master bus architecture

I. Introduction

As SoC (System-on-Chip) has dominated the ASIC world, the competitiveness of SoC performance has increased and triggered the market to move accordingly. Especially, SoC performance depends on the bus topology that consists of shared bus protocol, master, slave, arbiter, decoder and so on. World-class electronics companies have internally developed the bus topology such as AMBA, Core Connect and Silicon Micro-Networks[1~3]. However, although several masters and slaves are connected in the shared bus architecture, only one data transaction initiated by a selected master can be transferred to a selected slave. In the case that several masters try to

* 정회원, 인하대학교 전자공학과
(Dept. of Electronics Engineering, Inha University)
** 정회원, 전남대학교 전기 및 반도체공학과
(Dept. of Electrical and Semiconductor Engineering, Chonnam National University)
접수일자: 2009년9월15일, 수정완료일: 2009년12월30일

transfer the data transaction, the sequence of a data communication is decided by the bus arbitration policy.

If the multiple data transactions of several masters communicate simultaneously on the shared bus, the innovative performance improvement can be realized. However, as the multiple data transactions cannot be loaded to the shared bus concurrently in the bus protocol, most of the conventional bus topologies do not support the parallel data transaction on the shared bus. Therefore, we propose the novel bus architecture that is capable of transferring the multiple data transactions simultaneously, as well as using existing IP and bus protocol with simple design modification.

The key contributions of this paper are summarized as follows. First, we discuss why the flying master bus concept is introduced. Second, we describe which components are consisted and how much design overhead is needed to implement this architecture. Third, we verify how much the performance improvement is realized, using the probability approach and the TLM (Transaction Level Model) method that is recently used to explore the bus architecture deeply[4].

II. Flying master bus architecture

1. Concept of flying master bus

Introducing the flying master bus concept (next, we call flying bus), we propose that a specially defined master named as the flying master directly accesses the selected slaves with no regard to the bus protocol. When the master M1 and M2 concurrently try to initiate the data transactions to the slave S1 and S2 respectively on the conventional shared bus, the data transaction of master M1 is first transferred on the shared bus and that of master M2 is performed with the next sequence. On the other hand, as the bus architecture of figure 1(a) introduces the flying master concept, the flying master FM(M1) transfers the data transaction to slave S1 without the

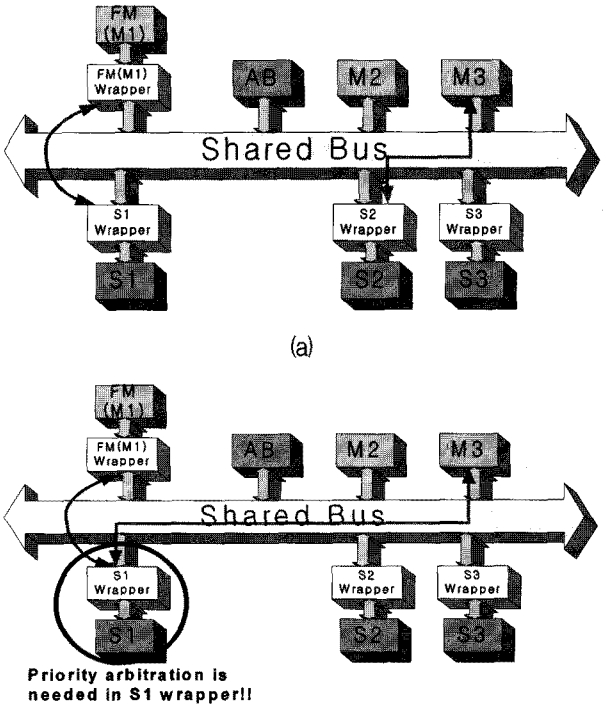


그림 1. (a) 동시 발생시의 (b) 마스터 FM(M1)과 M3(AB)의 버스 우선권 데이터 통신시의 제안하는 플라잉 마스터 버스 아키텍쳐 구조

Fig. 1. Proposed flying master bus architecture about (a) concurrent and (b) priority data communication of master FM(M1) and M3 (AB: arbiter).

shared bus and the data transaction of master M3 is performed through the shared bus. In this case, as the data transactions of master FM(M1) and M3 can be concurrently transferred, the performance of bus architecture improves definitely. If flying master and the other masters try to transfer the data transactions to same slave, data transactions communicate not concurrently but step by step with the priority decision of slave wrappers shown in figure 1(b).

The flying bus can be compared with the multiple bus system that the additional bus is introduced for concurrent data communication. However, the multiple bus system needs the required components containing a bridge block for the operation of multiple buses. Furthermore, it is inevitable that the long latency problems happen in the case that the inter-bus data communicate through a bridge. In general, it is

suitable to apply the multiple bus system when the architecture is comprised of more than 10 masters and has relatively low probability that the data transactions communicate through a bridge. Because the SoCs having the number of more than 10 masters are over a specification in the industrial companies of Siemens, Fuzitsu and Samsung, and not required in the aspect of application, most of the SoCs are not considered to adopt the multiple bus architecture. Consequently, it is necessary to improve the performance of single bus architecture such as proposed flying bus.

2. Design of flying bus

The wrappers of flying master and slave shown in figure 1 need to interface between the flying master and the shared bus signals in the flying master bus system. We use AHB bus model[1] to present the example of our wrappers shown in figure 2. Figure 2(a) shows the flying master wrapper that composes of a decoder and several multiplexers. HRDATA[31:0], HRESP[1:0] and HREADY input signals are connected in master wrapper and outputted through multiplexers to choose appropriate signals with the slave selection signal HSEL_s[x:0]. The slave wrapper has a role to select the priority between the data transactions of the flying master and the other masters with the HSEL selection block shown in figure2(b). in this wrapper, HREADY and HESL out signals are generated after comparing the transaction signal HTRANS[1:0]. We notice that the wrappers o f the flying master and the slaves are simply designed with using only several multiplexers without additional controller blocks.

3. Logic delay and area overhead

The proposed bus architecture was implemented through Verilog and mapped the design into Hynix 0.18um technology through the Synopsys Design Compiler. The logic delay due to the master and the slave wrappers along the bus is shown in table 1. The max delay becomes near 3.0ns and 2.5ns for master and slave wrappers respectively. In the synthesized netlists, master and slave wrappers have around 124 - 164 logic gate counts. However, the area

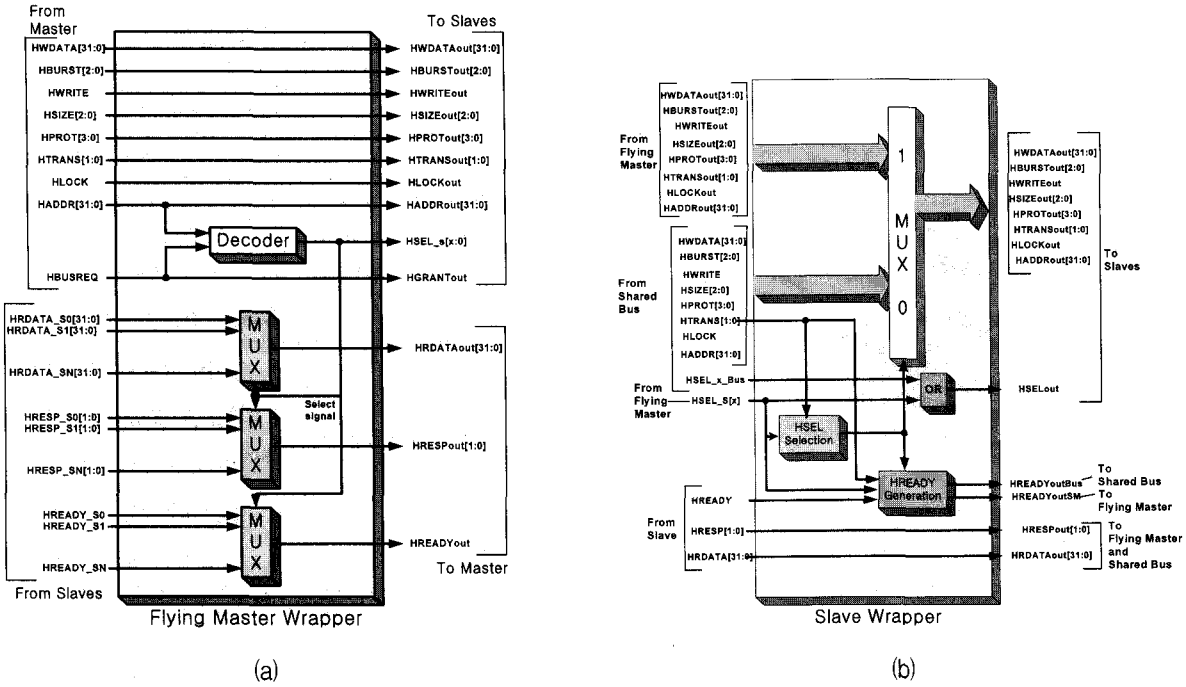


그림 2. (a) 플라잉 마스터와 (b) 슬레이브 래퍼의 블록 다이어그램
Fig. 2. Block diagram of (a) flying master and (b) slave wrappers.

표 1. 최대 지연시간과 게이트 카운트
Table 1. Max delay and gate count.

	Master wrapper	Slave wrapper
Max delay	2.95ns	2.48ns
Gate count	124	164
Critical path	From HBUSREQ To HRDATAout	From HSEL_S To HADDR_S

overhead is still small considering the typical area of modules in SoC designs.

Finally, we find that it is simple and easy to implement the flying bus and the design overhead is rather small, which is only composing of several multiplexers and bypass interconnection components.

Ⅲ. Performance analysis

1. Static performance analysis using probability

If flying master and the other masters try to transfer the data transactions to same slave, data transactions communicate not concurrently but step by step. Therefore, the total cycle of the flying bus is obtained as following equation:

$$C_{tot} = \alpha \cdot C_{fm} + C_{mx} \quad (1)$$

where, C_{tot}, C_{fm} and C_{mx} are total cycle, the cycle that is spent by the flying master, and the cycle that is spent by the other masters, respectively. α is the probability that flying master and the other masters transfer the data transactions to same slave and obtained as following equation:

$$\alpha = \sum_{n=1}^{n} \sum_{p=1}^{p} P_{FMSp} \cdot P_{MnSp} \quad (2)$$

where, P_{FMSp} and P_{MnSp} are the probabilities that the data transactions of flying master and the other masters are transferred to p slave respectively. Consequently, total cycle C_{tot} increases in accordance with the increment of α that is presented in Eq.(2).

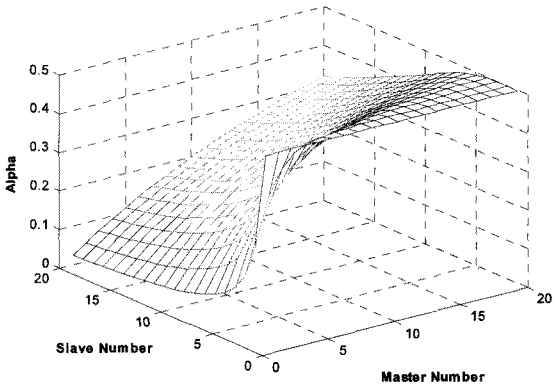


그림 3. 플라잉 마스터와 다른 마스터들이 동일 슬레이브에 전송하는 데이터 트랜잭션 확률
Fig. 3. Probability that the data transactions of flying master and the other masters are transferred to same slave.

The α graph of figure 3 is obtained from Eq. (2), provided that the probability that masters transfer the data transactions, is same as 50% with no regard to masters and slaves.

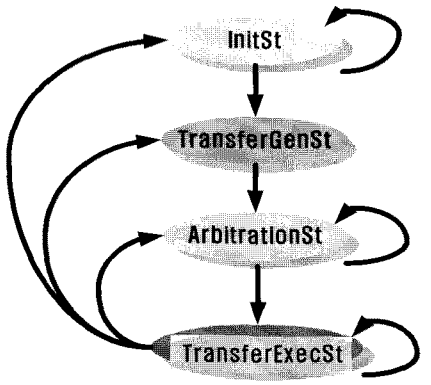
For example, when the numbers of master and slave are all 4, we obtain that α value is about 0.2 from figure 3. Then, we calculate that more than 40% of total cycle decreases from Eq. (1), provided that the usage proportion of flying master and the other masters is same. Finally, introducing the flying master bus architecture, we expect the remarkable improvement of bus performance.

2. Dynamic performance analysis using TLM

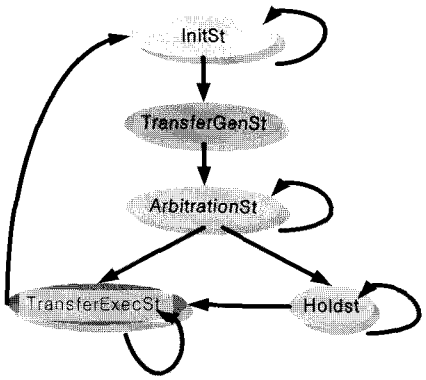
(1) Transaction level model

In order to implement the TLM (Transaction Level Model) method of bus architecture, we make the state machine and the algorithm of normal bus shown in figure 4(a) and table 2. There are InitSt, TransferGenSt, ArbitrationSt and TransferExecSt states in our normal bus architecture model. InitSt is the initial state that the bus requests of masters do not happen.

When a master tries to initiate the data transaction, our state moves on *TransferGenSt* where all kinds of bus signals related with bus



(a)



(b)

그림 4. (a) 일반 버스와 (b) 플라잉 버스의 버스 아키텍쳐 모델 상태도

Fig. 4. State machine of bus architecture model: (a) normal bus and (b) flying bus.

표 2. 일반 버스 아키텍쳐의 알고리즘

Table 2. Algorithm of normal bus architecture.

```
1: normal_bus_model(Arbitration_Type)
2: begin
3:    Cur_Cycle=0
4:    State=InitSt
5:    Master_Signal[all    masters]=Idle
6:    Slave_Signal[all    slaves]=Idle
7: while(Cur_Cycle<Final_Cycle)do
//  ------------------------------------------------------------------ //
// ---------------------- State is InitSt. ---------------------- //
//  ------------------------------------------------------------------ //
8:    if(State==InitSt)then
9:       Idle_Gen(Master_Signal[selected    masters])
10:      while(Master_Signal.Idle_Cycle[allmasters]--)do
11:         Cur_Cycle++
12:         Detail_Cycles_Cal(NULL,NULL,InitSt)
13:      endwhile
14:      Execute_Bus_Model(Master_Signal[all    masters],NULL,InitSt)
15:      State=TransferGenSt
16:   endif
//  ------------------------------------------------------------------ //
// ---------------------- State is TransferGenSt. ---------------------- //
// ------------------------------------------------------------------ //
17: elseif(State==TransferGenSt)then
18:    Req_Gen(Master_Signal[selected    masters])
19:    Addr_Gen(Master_Signal[selected    masters])
20:    Data_Gen(Master_Signal[selected    masters])
21:    Transfer_Signal_Gen(Master_Signal[selected    masters])
22:    State=ArbitrationSt
23: endelseif
//  ------------------------------------------------------------------ //
// ---------------------- State is ArbitrationSt. ---------------------- //
//  ------------------------------------------------------------------ //
24: elseif(State==ArbitrationSt)then
25:    Arbitration(Req[selected    masters],Arbitration_Type)
26:    while(ARBITRATION_CYCLE--)do
27:       Cur_Cycle++
28:       Detail_Cycles_Cal(NULL,NULL,ArbitrationSt)
29:       Execute_Bus_Model(Master_Signal[all    masters],NULL,ArbitrationSt)
30:    endwhile
31:    State=TransferExecSt
32: endelseif
//  ------------------------------------------------------------------ //
// ---------------------- State is TransferExecSt. ---------------------- //
// ------------------------------------------------------------------ //
33: elsethen//StateisTransferExecSt.
34: while((Master_Signal.Data_Size[selectedmaster]+Slave_Signal.
Slave_Latency[selected slave])--) do
35:       Cur_Cycle++
36:       Detail_Cycles_Cal(Master_Signal[all    masters],
Slave_Signal[all slaves],TransferExecSt)
37:       Execute_Bus_Model(Master_Signal[all    masters],
Slave_Signal[all slaves],TransferExecStSt)
38:    endwhile
39:    Master_Signal.Req[selected    master]=False
40:    Master_Signal.Granted[selected    master]=False
41:    if(Master_Signal.Req[somemasters]==True)State=ArbitrationSt
42:    elseif(Master_Signal.Idle_Cycle[somemasters]==0)State=TransferGenSt
43:    elseState=InitSt
44:    endif
45: endelse
46: endnormal_bus_model(Arbitration_Type)
```

communication are created. Next state is *ArbitrationSt* that the master assigning a bus grant is selected by using the previously defined arbitration policy. Next, our state moves on *TransferExecSt* state that the data of selected master communicate on the shared bus. After data communication is finished, our state moves on *InitSt, TransferGenSt* or *ArbitrationSt* state due to the bus request of master.

It directly moves on ArbitrationSt in the case that the bus request signal Req[master] is '1', If Req[master] and idle cycle idle_cycle[master] are all '0', TransferGenSt state is selected. For the residue, it moves on InitSt state. Table 2 gives a full detail of our normal bus model operation.

Next, we make the state machine and the algorithm of flying bus shown in figure 4(b) and table 3. ArbitrationSt and TransferExecSt states are

표 3. 플라잉 마스터 버스 아키텍쳐의 알고리즘
Table 3. Algorithm of flying master bus architecture.

```
1: flying_bus_model()
2: begin
3: Cur_Cycle=0
4: State=InitSt
5: Master_Signal[flying master]=Idle
6: Slave_Signal[all slaves]=Idle
// ------------------------------------------------------------ //
// ---------------------- State is InitSt. ---------------------- //
// ------------------------------------------------------------ //
7: if(State==InitSt)then
8:    Idle_Gen(Master_Signal[flying    master])
9:    while(Master_Signal.Idle_Cycle[flyingmaster]--)do
10:      Detail_Cycles_Cal(NULL,NULL,InitSt)
11:   endwhile
12:   Execute_Bus_Model(Master_Signal[flying    master],NULL,InitSt)
13:   State=TransferGenSt
14: endif
// ------------------------------------------------------------ //
// ---------------------- State is TransferGenSt. ---------------------- //
// ------------------------------------------------------------ //
15: elseif(State==TransferGenSt)then
16:   Req_Gen(Master_Signal[flying    master])
17:   Addr_Gen(Master_Signal[flying    master])
18:   Data_Gen(Master_Signal[flying    master])
19:   Transfer_Signal_Gen(Master_Signal[flying    master])
20:   State=ArbitrationSt
21: endelseif
// ------------------------------------------------------------ //
// ---------------------- State is ArbitrationSt. ---------------------- //
// ------------------------------------------------------------ //
22: elseif(State==ArbitrationSt)then
23:   Arbitration(Req[selected    masters],Arbitration_Type)
24:   while(ARBITRATION_CYCLE--)do
25:      Cur_Cycle++
26:      Detail_Cycles_Cal(NULL,NULL,ArbitrationSt)
27:      Execute_Bus_Model(Master_Signal[all    masters],NULL,ArbitrationSt)
28:   endwhile
29: if(TransferExecSt[grantedmaster]&(addr[grantedmaster]==addr[flyingmaster]))then
30:      State=HoldSt
31:   endif
32:   elsethen
33:      State=TransterExecSt
34:   endelse
35: endelseif
// ------------------------------------------------------------ //
// ---------------------- State is HoldSt. ---------------------- //
// ------------------------------------------------------------ //
36: elseif(State==Holdst)then
37:   while(TransferSt[grantedmaster])do
38:      Detail_Cycles_Cal(NULL,NULL,ArbitrationSt)
39:      Execute_Bus_Model(Master_Signal[flying    master],NULL,ArbitrationSt)
40:   endwhile
41:   State=TransferExecSt
42: endelseif
// ------------------------------------------------------------ //
// ---------------------- State is TransferExecSt. ---------------------- //
// ------------------------------------------------------------ //
43: elsethen//StateisTransferExecSt.
44:   while((Master_Signal.Data_Size[flyingmaster]+Slave_Signal.
Slave_Latency[selected slave])--) do
45:      Detail_Cycles_Cal(Master_Signal[flying    master],Slave_Signal[all
slaves],TransferExecSt)
46:   Execute_Bus_Model(Master_Signal[flying master],Slave_Signal[all slaves],
TransferExecStSt)
47:   endwhile
48:   Master_Signal.Req[flying    master]=False
49:   State=InitSt
50: endelse
51: endwhile
52: endflying_bus_model()
```

modified in additional to HoldSt, compared with normal bus. After ArbitrationSt state, next state moves on TransferExecSt or HoldSt state. If the target slave of flying master and the other master is same, our state moves on HoldSt. For the residue, it moves on TransferExecSt state. After finishing a data transfer in TransferExecSt state, our state directly moves on InitSt state in order to wait the next communication of flying master.

In this paper, we apply this bus model to AMBA system that has been used in more than 50% embedded system of all over the world [1]. The data length of transaction can be random changed to 1, 4, 8 and 16 shown in figure 5. The idle cycle between the data transactions of master is also changeable using random function that is operated with the range and the mean values of idle cycle. The models of SDRAM and SRAM controllers are used as slave component. SDRAM has relatively long latency due to pre-charge, refresh, row/column access cycles and so on. However, SRAM has no latency while writing or reading. In our simulation, though the numbers of master and slave are changeable, those of master and slave are set to all 4. In order to accurately confirm the results, final cycle is set to more than 1,000,000.

The arbitration policies of fixed priority, round-robin, TDMA and Lottery are applied in our performance simulation[5~8]. Fixed priority and round-robin policies are classical schemes and used in many commercial SoCs. On the other hand, TDMA and Lottery is recently developed and can control a master priority using slot number and probability respectively. In our simulation, the master priority considered, the slot number of each master from master M1 to master M4 is assigned to 3, 1, 1, and 1 respectively in TDMA scheme. Similarly, the probability of a bus use from master M1 to master M4 is assigned to 3/6, 1/6, 1/6, and 1/6 respectively

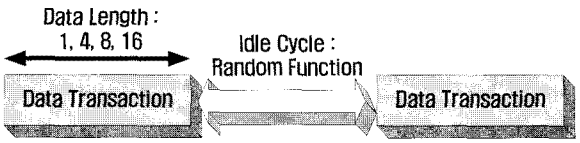


그림 5. 버스 아키텍쳐 모델의 데이터 길이와 idle 사이클

Fig. 5. Data length and idle cycle of bus architecture model.

표 4. 데이터 길이와 idle 사이클의 조건

Table 4. Condition of data length and idle cycle.

Data length (Random function)		Single, Burst (4,6,8,16)
Idle cycle (Random function)	Range	0~30
	Mean	15

in Lottery scheme in order to be compared with the results of TDMA scheme.

(2) Experimental results

Figure 6 shows the dynamic performance comparison with flying and normal buses due to the arbitration policies.

In the simulation until 1,000,000 cycles, we find that the total bus transaction cycles of flying bus (FB) and normal bus (NB) are about 700,000 and 500,000 respectively, in the case of SDRAM slave shown in figure 6(a), irrespective of arbitration schemes. Especially, the master M1 assigned by a flying master can transfer many data transactions relatively in all arbitration policies, though the transaction cycles of all masters increase in flying bus. For instance, the master M1 of a flying bus transfers the data transaction nearly twice as much transaction cycle in round-robin scheme. This result is similar to the case of SRAM slave shown in figure 6(b).

The systems of SoCs have one processor at least to operate the other masters and perform the software commands. Though the processor is only one of the masters, the role of a processor is relatively important and the processing load is also heavy. Especially, a proportion of processor's usage is

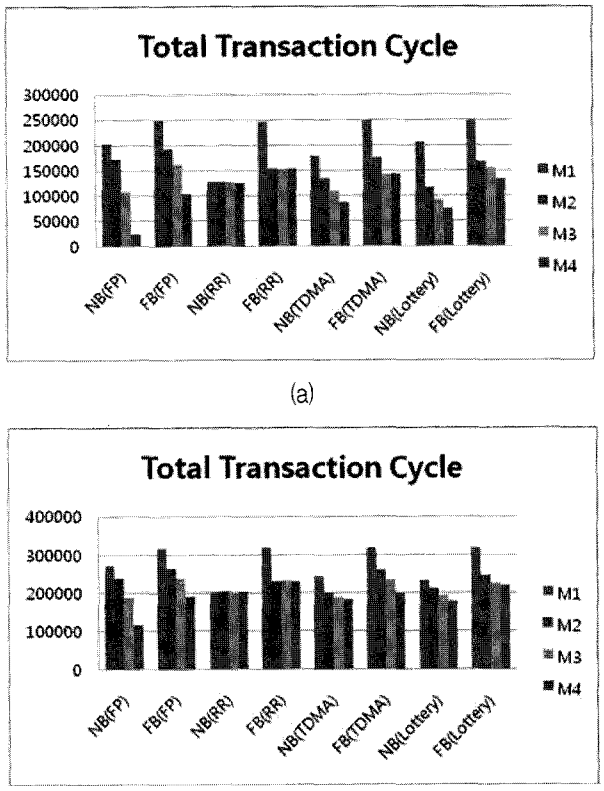
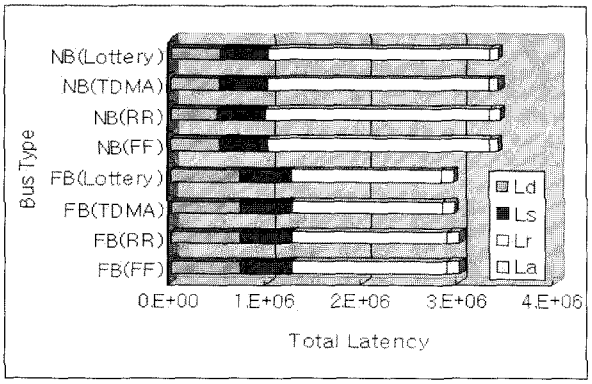


그림 6. (a) SDRAM 슬레이브, (b) SRAM 슬레이브의 플라잉 마스터 버스구조와 일반 버스구조의 데이터 트랜잭션 사이클 비교

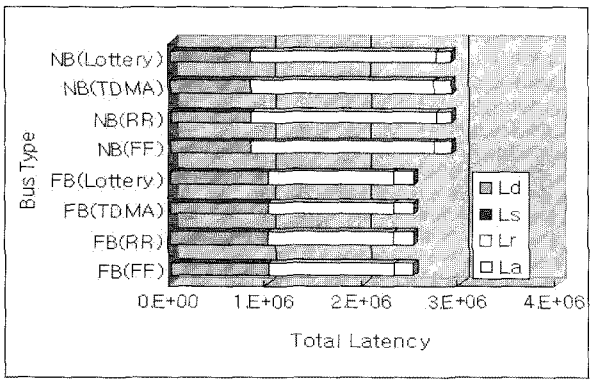
Fig. 6. Data transaction cycle comparison between flying and normal bus: (a) SDRAM slave, (b) SRAM slave. (FB: Flying Bus, NB: Normal Bus, FP: Fixed Priority, RR: Round-Robin).

around 50% in the communication MCU chips, the performance of which is dependent on the processor performance. So, it is noticeable that the performance of master M1 assigned as a flying master predominates. Finally, we propose that the master M1 of flying bus is employed as important component like a processor for high performance.

As the total bus transaction cycles of flying bus (FB) and normal bus (NB) are about 1,000,000 and 800,000 respectively in the case of SRAM slave, an increase rate relatively slows. That is why the performance of normal bus is already high, as slave latency Ls of SRAM is small shown in figure 7. Conclusively, from data transaction cycle comparison until 1,000,000 cycles, we find that the performance of flying bus is superior to that of normal bus around



(a)



(b)

그림 7. (a) SDRAM 슬레이브, (b) SRAM 슬레이브의 플라잉 마스터 버스구조와 일반 버스구조의 총 레이턴시 비교

Fig. 7. Total latency comparison between flying and normal bus: (a) SDRAM slave, (b) SRAM slave. (FB: Flying Bus, NB: Normal Bus, FP: Fixed Priority, RR: Round-Robin).

40% and 25% about SDRAM and SRAM slaves respectively.

Figure 7 shows the latency comparison about normal and flying buses. Then, the total latency is obtained as following equation:

$$L_{tot} = L_d + L_s + L_r + L_a \qquad 3)$$

where, L_{tot} is total latency of bus system, L_d is the latency of data transaction, L_s is the latency of target slave, L_r is the latency of bus request and L_a is the latency of arbitration. L_d, L_s, L_r and L_a values of figure 7 are total cycles that summate the latency cycles of each master. From figure 7, we confirm that L_d, L_s, L_r and L_a values are unrelated to arbitration policies and have similar results about all

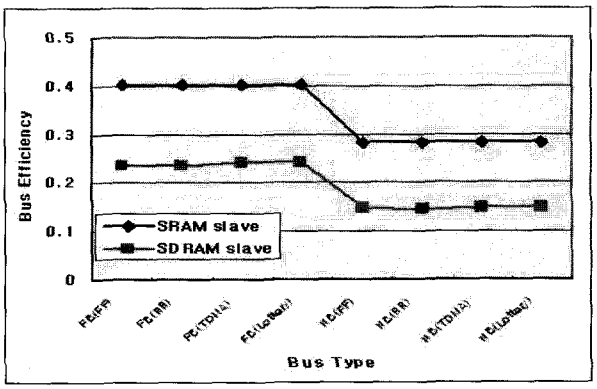


그림 8. 플라잉 마스터 버스구조와 일반 버스구조의 버스 효율성 비교

Fig. 8. Bus efficiency between flying and normal bus. (FB: Flying Bus, NB: Normal Bus, FP: Fixed Priority, RR: Round-Robin).

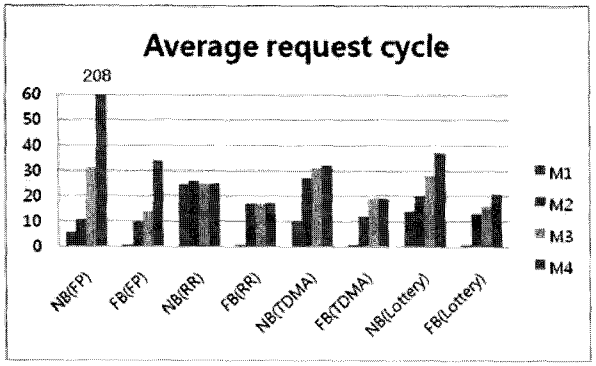
arbitration policies.

In the AMBA system, as one cycle is required in order to transfer one data, the data latency L_d of figure 7 is same as the data transaction cycle of figure 6. We find that the difference between figure 7(a) and 7(b) is caused by the slave latency that is spent while writing or reading slave data.Generally, SDRAM has relatively long latency due to pre-charge, refresh, row/column access cycles and so on. However, as SRAM has no latency while writing or reading, more data can be transferred speedily.

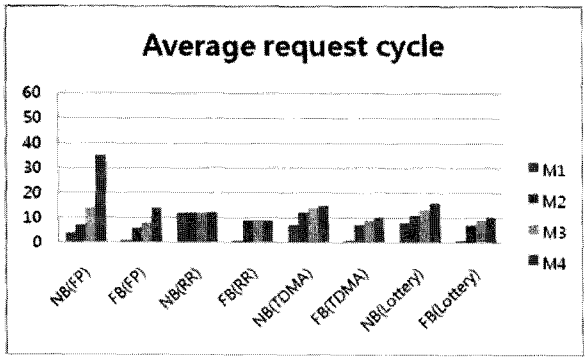
The efficiency of bus transaction is obtained as following equation:

$$\lambda = \frac{L_d}{L_{tot}} \qquad (4)$$

where, λ is the efficiency of bus system. From figure 8 and Eq. (4), we notice that λ of flying bus is larger than that of normal bus around 60% and 43% about SDRAM and SRAM slaves respectively. These results cause that the total bus request latency of flying bus is relatively small. Figure 9 shows the average bus request cycles until getting a bus grant. The request cycle of master M1 is only one cycle in the flying bus, which is independent of slave type and arbitration scheme. We infer that the master M1 assigned flying master is easy to get a bus grant in



(a)



(b)

그림 9. (a) SDRAM 슬레이브, (b) SRAM 슬레이브 경우의 중재 방식에 따른 평균 요청 사이클 비교

Fig. 9. Average request cycle due to arbitration policies in (a) SDRAM and (b) SRAM slave case (FB: Flying Bus, NB: Normal Bus, FP: Fixed Priority, RR: Round-Robin).

spite of heavy traffic situation. On the other hand, the master M4 having the lowest priority takes many request cycles in the fixed priority of normal bus, shown in figure 9, which leads to the starvation phenomenon hardly getting a bus grant for long time. However, Introducing flying master, we can decrease the request cycle of master M4 from 208 to 34 in the case of SDRAM slave and from 35 to 14 in that of SRAM slave.

The request cycle of round-robin priority is nearly equal except the case of master M1 in flying bus and that of TDMA and Lottery policies similarly increase due to master priority. Finally, we find that the request cycle of flying bus is outperformed as well as the transaction cycle meaning a transferring amount.

Also, we notice that the graph shapes of figure 9 are decided only by arbitration policies despite the striking difference of request cycle between SDRAM and SRAM.

IV. Conclusion

We propose the flying master bus architecture that is capable of transferring the multiple data transactions simultaneously to realize high

표 5. 일반 버스 방식과 플라잉 마스터 버스 방식의 총 성능 비교

Table 5. Total performance comparison between normal and flying buses.

Item	Total transaction cycle				Total bus efficiency				Total request cycle			
Slave type	SDRAM		SRAM		SDRAM		SRAM		SDRAM		SRAM	
Bus type	Normal bus	Flying bus	Normal bus	Flying bus	Normal bus	Flying bus	Normal bus	Flying bus	Normal bus	Flying bus	Normal bus	Flying bus
Fixed priority	506868	707951	815041	1008818	0.15	0.24	0.28	0.40	256	59	60	29
Round-robin	505614	707866	815430	1009195	0.15	0.24	0.28	0.40	101	52	48	28
TDMA	507661	708640	815387	1009747	0.15	0.24	0.28	0.40	100	51	48	27
Lottery	508183	707294	815212	1009019	0.15	0.24	0.28	0.40	99	51	48	27

performance SoC.

Table 5 summarizes the total performance comparison between normal and flying buses. Compared items are transaction cycle, bus efficiency and request cycle. Applying the flying bus, we find that 25~40% of transaction cycle and 43~60% of bus efficiency are increased and 43~77% of request cycle is decreased.

Conclusively, we conclude that the proposed flying master bus architecture is promising as the leading candidate of the bus architecture in the aspect of performance and efficiency.

Reference

[1] ARM, Limited. AMBA Specification, 1999.

[2] IBM, Armonk, NY, "CoreConnect bus architecture," 1999.

[3] Sonics, Inc., Mountain View, CA, "Silicon micronetworks technical overview," 2002.

[4] K. Lee and Y. Yoon, "Architecture exploration for performance improvement of SoC chip based on AMBA system," ICCIT, pp.739-744, 2007.

[5] K. Lahiri, A. Raghunathan, and G. Lakshminarayana, "The LOTTERYBUS On-Chip Communication Architecture," IEEE Trans. VLSI Systems, vol.14, no.6, 2006.

[6] M. Jun, K. Bang, H. Lee and E. Chung, "Latency-aware bus arbitration for real-time embedded systems," IEICE Trans. Inf. & Syst., vol.E90-D, no.3, 2007.

[7] Y. Xu, L. Li, Ming-lun Gao, B.Zhand, Zhao-yu Jiand, Gao-ming Du, W. Zhang, "An Adaptive Dynamic Arbiter for Multi-Processor SoC", Solid-State and Integrated Circuit Technology International Conf., pp.1993-1996, 2006.

[8] Chiung-San Lee, "High-Fair Bus Arbiter for Multiprocessors," IEICE Trans. Inf. & Syst., vol.E80-D, no.1, 1997.

저 자 소 개

이 국 표 (정회원)
대한전자공학회 논문지
제45권 SD편 제4호 참조

강 성 준(정회원)
대한전자공학회 논문지
제46권 SD편 제2호 참조

윤 영 섭 (정회원)
대한전자공학회 논문지
제45권 SD편 제4호 참조