논문 2008-45SD-9-14

마스터와 슬레이브에 따른 싱글버스와 다중버스 토폴로지의 성능분석

(Performance Analysis of Single and Multiple Bus Topology Due to Master and Slave)

이 국 표*, 윤 영 섭*

(Kook-Pyo Lee and Yung-Sup Yoon)

요 약

SoC의 버스 구조에는 싱글버스와 다중버스로 구분된다. 싱글버스는 전송을 원하는 여러 개의 마스터 중 선택된 하나의 마스터만이 데이터 트랜잭션을 수행할 수 있다. 반면에 다중버스는 개별적으로 동작이 가능한 버스를 브리지를 통해 연결하여 각각의 버스에서 여러 데이터를 병렬 처리할 수 있다. 그러나 현재의 버스에서 다른 버스로 데이터 통신을 수행할 경우, 레이턴시가 급격하게 증가할 수 있다. 게다가, 다중버스의 성능은 마스터의 개수, 슬레이브의 종류 등에 따라 쉽게 바뀔 수가 있다. 이에 본 논문에서는 TLM(Transaction Level Model) 시뮬레이션 방법을 이용하여 마스터의 개수, SDRAM, SRAM, 레지스터 등의 슬레이브 종류에 따른 싱글버스와 다중버스 아키텍처의 성능을 정량적으로 비교·분석하였다.

Abstract

The SoC bus topology is classified to single and multiple bus systems due to bus number. In single bus system, the selected only one master among the masters that try to initiate the bus transaction can execute its data transaction. On the other hand, in multiple bus system, as several buses that can be operated independently are connected with bridge, multiple data can be transferred parallel in each bus. However, In the case of data communication from one bus system to the other, data latency has remarkably increased in multiple bus. Furthermore, the performance of multiple bus can be easily different from master number, slave type and so on. In this paper, the performance of single and multiple bus architecture is compared and quantitatively analysed with the variation of master number and slave type especially applying SDRAM, SRAM and register with TLM simulation method.

Keywords: SoC, bus topology, single bus, multiple bus

Ⅰ. 서 론

SoC는 System on a Chip의 줄임말로서 하나의 칩으로 시스템을 구현한다는 의미이다. 그만큼 SoC의 장점은 소형 경량화라 말할 수 있고, 칩의 수가 줄어들기 때문에 고장이 적어지고 시스템의 성능은 향상된다[1~3]. 따라서 소비자들의 욕구를 충족시켜줄 수 있기 때문에 설계자들의 목표이기도 하다. 현재 SoC 시장의 70%이상을 점유하고 있는 ARM 프로세서에서 채택한 버스 구조인 AMBA(Advanced Microcontroller Bus Architecture)가 온 칩 통신의 표준이 되고 있다.[4] 그 중 AHB(Advanced High performance Bus)는 고성능 버스를 의미하며, 일반적인 AHB의 구성요소에는 하나의 버스 내에 여러 개의 마스터, 슬레이브, 아비터, 디코더로 구성되어있다. 마스터는 CPU, DMA, DSP 등과 같이 어드레스나 제어신호를 내보냄으로써 "read"나 "write"의 동작을 할 수 있도록 하는 주체를 말하며 슬

* 정회원, 인하대학교 전자공학과
(Dept. of Electronics Engineering, Inha University)
접수일자: 2008년4월2일, 수정완료일: 2008년8월28일

레이브는 DRAM, SRAM 컨트롤러 등과 같이 주어진 어드레스 공간 내에서 "read"나 "write"를 가능하게 해주는 장치를 말한다. 그리고 아비터는 여러 개의 마스터가 동시간대에 버스를 사용할 수 없기 때문에 이를 중재하는 역할을 수행하며, 중재하는 방식에 따라 시스템의 성능을 향상 시킬 수 있어 별개의 IP의 성능 향상과는 별도로 많은 연구를 하고 있는 부분이다. 마지막으로 디코더는 마스터로부터 나오는 어드레스의 상위 비트를 가지고 적절한 슬레이브를 선택해주는 역할을 수행한다. 또한 시스템의 성능을 향상시키기 위한 연구로서 버스 구조를 바꾸는 방법도 제안되고 있다. 일반적인 AHB의 경우는 하나의 버스를 동시간대에 이용할 수 없기 때문에 이를 해결하기 위해 최근에는 버스를 다중으로 두고 그 사이에 브리지를 연결한 다중버스가 구성되어 있다. 그러나 다중버스의 경우는 버스 1에 있는 마스터가 버스 2에 있는 슬레이브를 이용하고자 할 경우 브리지를 통해서 데이터를 처리해야하기 때문에 브리지의 레이턴시에 의한 지연이 발생할 수 있다[5]. 또한 싱글버스나 다중버스에 관계없이 마스터나 슬레이브의 종류에 따라 버스의 성능은 크게 달라질 수 있다.

이에 본 논문에서는 TLM(Transaction Level Model) 방법을 이용하여 마스터의 개수에 따른 싱글버스와 다중버스의 성능을 분석하였고, 슬레이브 SDRAM과 SRAM에 따른 싱글버스와 다중버스의 성능을 분석하여 각각에 조건에 따라 적절한 버스 시스템을 찾고자 하였다.

Ⅱ. 본 론

1. 다중 버스 아키텍처 모델구성

그림 1에는 다중버스 구조의 버스통신이 나타나 있는데 공용버스1과 공용버스2는 개별적으로 데이터 전송이 가능하다. 그러므로 싱글버스 구조와 달리 여러 데이터를 동시에 전송할 수 있는 장점을 가지고 있다.

그러나 공용버스1의 마스터와 공용버스2의 슬레이브

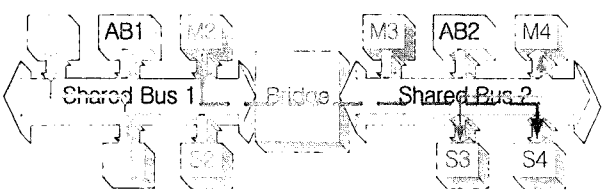


그림 1. 다중버스 구조의 버스 통신
Fig. 1. Bus communications of multiple bus architecture.

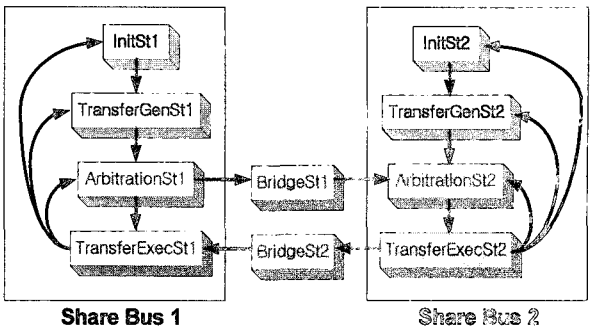


그림 2. 다중 버스아키텍처 모델의 상태도
Fig. 2. State machine of multiple bus architecture model.

사이에서 데이터 전송을 할 경우, 브리지 블록을 통과해야 하는데, 이때 브리지는 데이터를 받아들이는 슬레이브 기능과 데이터를 전송하는 마스터 기능을 동시에 해야 한다. 결국 버스 사이의 데이터 전송은 복잡하게 진행되고 레이턴시(latency)가 매우 크기 때문에, 다중버스 구조의 성능이 크게 하락될 수 있다.

그림1에 2중 버스에 대한 브리지 버스아키텍처의 상태도가 나타나 있다. 공용버스1과 공용버스2는 개별적인 데이터 처리가 이루어지며, 만약 공용버스1의 데이터를 공용버스2의 슬레이브로 전송할 경우 브리지1상태를 통과하여 처리된다. 이에 대한 모델링을 구성하기 위해서 표 1, 표 2와 같은 알고리즘을 구현하였다. 표1에서 보듯이 최종 사이클 (Final_Cycle)까지 진행될 동안 공용버스1과 공용버스2는 bus1_execute(Bus1_Arbitration_Type), bus2_execute(Bus2_Arbitration_Type) 함수의 수행이 반복된다. 만약 공용버스를 더 추가하고 싶으면 bus3_execute, bus4_execute 함수를 추가할 수 있다. 표 2에는 bus1_ execute에 대한 알고리즘이 나타나 있으며, bus2_ execute, bus3_execute의 경우도 bus1_execute와 동일하게 구현할 수 있다.

표 1. 다중 버스 아키텍처 알고리즘
Table 1. Algorithm of multiple bus architecture.

```
1:  multiple_bus_model(Bus1_Arbitration_Type,Bus2_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
8:      bus1_execute(Bus1_Arbitration_Type)
9:      bus2_execute(Bus2_Arbitration_Type)
            .............
10:   end while
11: end multiple_bus_model(Bus1_Arbitration_Type,Bus2_Arbitration_Type)
```

표 2. 공용버스1에 대한 버스 아키텍처 알고리즘
Table 2. Algorithm of bus architecture about shared bus1 execution.

```
1:  bus1_execute(Bus1_Arbitration_Type)
2:  begin
3:      // ------------------------------------------------------ //
4:      // ------------------- State is InitSt1. ----------------- //
5:      // ------------------------------------------------------ //
6:    If (State=InitSt1) then
7:      Idle_Gen(Master_Signal[selected masters])
8:      if(Master_Signal.Idle_Cycle[all masters]=) then
9:        Cur_Cycle++
10:       Detail_Cycles_Cal(NULL,NULL,InitSt)
11:     end if
12:     else then
13:       Execute_Bus_Model(Master_Signal[all masters],NULL,InitSt)
14:       State=TransferGenSt1
15:     end else
16:   end if
17:     // ------------------------------------------------------ //
18:     // ------------------ State is TransferGenSt1. ---------- //
19:     // ------------------------------------------------------ //
20:   else if (State=TransferGenSt1) then
21:     Req_Gen(Master_Signal[selected masters])
22:     Addr_Gen(Master_Signal[selected masters])
23:     Data_Gen(Master_Signal[selected masters])
24:     Transfer_Signal_Gen(Master_Signal[selected masters])
25:     State=ArbitrationSt1
26:   end else if
27:     // ------------------------------------------------------ //
28:     // ------------------ State is ArbitrationSt1. ---------- //
29:     // ------------------------------------------------------ //
30:   else if (State=ArbitrationSt1) then
31:     Arbitration(Req[selected masters],Arbitration_Type)
32:     if (ARBITRATION_CYCLE!=0) then
33:       ARBITRATION_CYCLE--
34:       Cur_Cycle++
35:       Detail_Cycles_Cal(NULL,NULL,ArbitrationSt1)
36:       Execute_Bus_Model(Master_Signal[all masters],NULL,ArbitrationSt1)
37:     end if
38:     else if (bus1.transaction_bus=1) then
39:       if (TransferExecSt2 & (bus2.transaction_bus=1)) then
40:         Cur_Cycle++
41:         Detail_Cycles_Cal(NULL,NULL,ArbitrationSt1)
42:         Execute_Bus_Model(Master_Signal[all masters],NULL,ArbitrationSt1)
43:       end if
44:       else then
45:         State=TransferExecSt1
46:       end else
47:     end else if //else if (bus1.transaction_bus=1) then
48:     else if (bus1.transaction_bus=2) then
49:     If (TransferExecSt2 & (bus2.transaction_bus=2)) then //State is BridgeSt1.
50:         Cur_Cycle++
51:         Detail_Cycles_Cal(NULL,NULL,ArbitrationSt1)
52:         Execute_Bus_Model(Master_Signal[all masters],NULL,ArbitrationSt1)
53:       end if
54:       else then
55:         State=TransferExecSt1
56:       end if
57:     end else if //else if (bus1.transaction_bus=2) then
58:   end else if //else if (State=ArbitrationSt) then
59:     // ------------------------------------------------------ //
60:     // ------------------ State is TransferExecSt1. --------- //
61:     // ------------------------------------------------------ //
62:   else if (State=TransferExecSt1) then
63:    If ((Master_Signal.Data_Size[selected master]+Slave_Signal.
64     Slave_Latency[selected slave])=) then
65:       Cur_Cycle++
66:       Detail_Cycles_Cal(Master_Signal[all masters],Slave_Signal[all slaves],
          TransferExecSt1)
67:       Execute_Bus_Model(Master_Signal[all masters],Slave_Signal[all slaves],
          TransferExecStSt1)
68:     end if
69:     Master_Signal.Req[selected master]=False
70:     if     (Master_Signal.Req[some masters]=True)     State=ArbitrationSt1
71:     else if(Master_Signal.Idle_Cycle[some masters]=0) State=TransferGenSt1
72:     else                                             State=InitSt1
73:   end else
74: end bus1_execute(Bus1_Arbitration_Type)
```

InitSt1 상태는 모든 마스터로부터 데이터전송 요청이 발생하지 않아서 기다리는 구간이며, 사이클 Cur_Cycle은 계속 증가하게 된다. 만약 마스터의 데이터 요청이 발생하면 TransferGenSt1 상태로 진입하게 된다. 여기서 Detail_Cycles_Cal(Master, Slave, State)은 버스 트랜잭션 사이클, 버스 요청 사이클, Idle 사이클 등 소요 사이클을 계산하는 함수이다.

그리고 Execute_Bus_Model(Master, Slave, State)는 각각의 상태(State)에 따른 버스의 진행을 규정하는 함수로 데이터 통신을 수행한다. TransferGenSt1 상태는 데이터전송에 필요한 어드레스, 데이터, 전송신호 등을 만드는 구간이며, 사이클 cur_cycle이 증가 없이 ArbitrationSt1 상태로 진행하게 된다.

ArbitrationSt1 상태는 여러 마스터의 데이터전송 요청 중에서 우선순위에 따라 마스터를 선택하는 구간으로서 Arbitration(request_master,arbitration_type) 함수에 의해서 해당 마스터를 선택하게 된다. 다음으로 버스중재 사이클(ARBITRATION_CYCLE) 만큼 시간지연 후, 데이터 전송이 이루어진다. 만약 전송하려는 데이터의 슬레이브가 공용버스2에 있을 경우 BridgeSt1 상태로 진행하게 되며, 전송하려는 데이터의 슬레이브가 공용버스1에 있을 경우 공용버스2의 데이터 전송이 이루어지고 있지 않으면 데이터 전송을 시작할 수 있다.

TransferExecSt1 상태는 데이터 전송이 실제로 이루어지는 구간으로서, 슬레이브 레이턴시가 고려되어 데이터 트랜잭션이 발생한다. 슬레이브에는 대표적으로 레지스터, SRAM, SDRAM 컨트롤러 등이 있으며, 일반적으로 refresh 시간, pre-charge 시간, CAS 레이턴시 등에 의해서 SDRAM의 레이턴시가 가장 길다. 이에 반하여 SRAM과 레지스터의 경우에는 "쓰기", "읽기" 명령에 따라 즉시 데이터 전송이 발생하며 레이턴시가 짧아서 고속 데이터 처리에 유리하다. 본 연구에서는 SDRAM, SRAM, 레지스터 모델링을 통하여 슬레이브 종류에 따른 성능을 분석해 보았다.

그림 3과 같이 데이터는 길이에 따라 싱글 데이터와 버스트 데이터로 나뉘며, 버스트 데이터는 길이에 따라 4, 8, 16 등을 가질 수 있는데, 그림3에 자세히 나타나

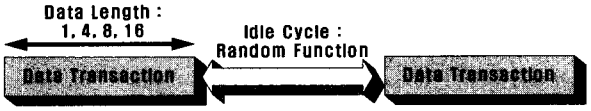


그림 3. 데이터 길이와 idle 사이클
Fig. 3. Data length and idle cycle.

있다. 본 연구에서는 랜덤함수를 이용하여 데이터 길이를 무작위로 발생시켜 전송시켰다. 그리고 싱글버스와 다중버스의 성능차이를 명확하게 보기 위해서 idle 사이클의 크기의 평균값을 25로 하였으며 랜덤함수로 0-50까지의 값이 발생하도록 하였다.

2. 다중 버스 아키텍처 성능

그림 4에 싱글버스와 다중버스의 데이터 트랜잭션에 대해 나타나 있다. 최종 사이클 (Final_Cycle)은 1,000,000 사이클로 하였으며, 버스트 데이터의 효과를 명확하게 보여주기 위해서, 싱글데이터와 버스트 크기가 16인 버스트 데이터를 무작위로 발생시켜서 전송시켰다. 그리고 그림4의 버스 중재 방식은 fixed priority 방식으로 하였다.

다중버스의 경우, 마스터가 50% 확률로 동일 버스의 슬레이브에 데이터 전송을 하고, 나머지 50% 확률로 다른 버스의 슬레이브에 데이터 전송을 하는 것으로 정하였다. 만약 마스터가 동일 버스의 슬레이브에 데이터를 전송할 확률을 50% 보다 높인다면 브리지 통과 확률이 적어지므로 다중버스의 성능은 보다 크게 향상될 것이며, 반대의 경우에는 성능향상이 그림1 보다 작아질 것이다. 일반적으로 SoC 칩의 사양을 결정할 때, 다중버스에서 마스터가 접근하는 슬레이브는 동일버스에 연결시켜서 성능을 높이려 하기 때문에 실제 성능은 그림4 보다 우수하다.

그림 4(a)는 슬레이브 SDRAM에 따른 데이터 트랜잭션 사이클을 보여주고 있다. 슬레이브 SDRAM의 레이턴시가 SRAM 보다 훨씬 크기 때문에 그림 4(b)보다 성능이 떨어짐을 알 수 있다. 싱글버스의 전체 트랜잭션 사이클이 629,230 사이클이고, 다중버스의 전체 트랜잭션 사이클은 882,192 사이클로 다중버스가 싱글버스보다 성능이 약 40% 이상 향상되었음을 알 수 있다.

또한 싱글버스의 경우, 마스터 M5, M6, M7, M8에서 버스점유권을 현저하게 받지 못하는 스타베이션(starvation)현상이 발생하였다. 그림 4(b)는 슬레이브 SRAM 또는 레지스터에 따른 데이터 트랜잭션 사이클을 보여주고 있다.

싱글버스의 전체 트랜잭션 사이클이 995,333 사이클이고, 다중버스의 전체 트랜잭션 사이클은 1,155,383으로 다중버스가 싱글버스보다 성능이 약 16% 이상 향상되었음을 알 수 있으며 싱글버스의 경우 마스터 M6, M7, M8에서 스타베이션 현상이 발생하였다.

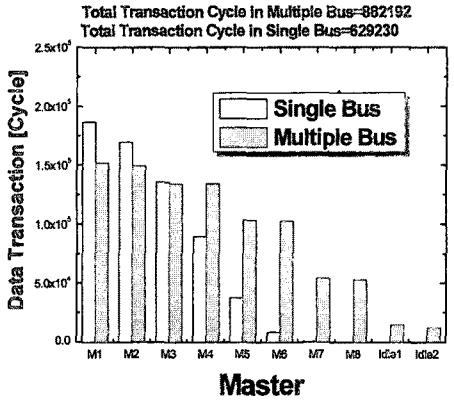
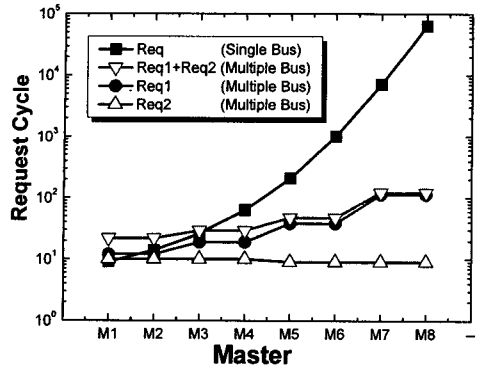


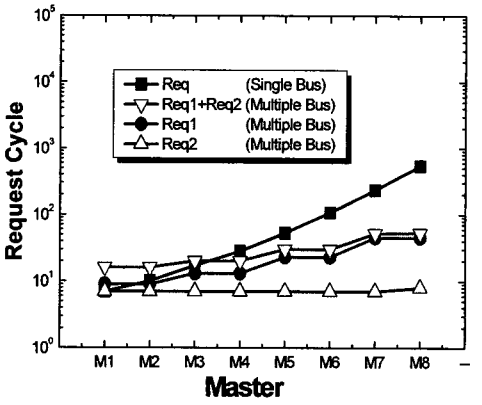
그림 4. (a) 슬레이브 SDRAM와 (b) 슬레이브 SRAM 또는 레지스터에 따른 데이터 트랜잭션 사이클
Fig. 4. Data transaction cycle due to (a) slave SDRAM and (b) slave SRAM or register.

그림 4(b)의 슬레이브는 레이턴시가 작아서 신속하게 데이터처리를 하기 때문에 그림 4(b)의 다중버스의 idle 사이클이 그림 4(a)의 다중버스의 idle 사이클 보다 훨씬 크다. 결국 그림 4(b) 슬레이브의 경우 성능향상이 작은 것처럼 보이지만, 그림 3의 idle 사이클의 평균값을 설정된 25에서 더 줄이거나, 마스터 개수를 늘려서 실험한다면 성능향상을 더욱 증대시킬 수 있을 것이다. 그림 5는 슬레이브에 따른 버스요청 사이클을 보여준다. 슬레이브 SDRAM에 적용하여 레이턴시가 길어짐에 따라 버스 대기시간도 길어지게 되어 그림 5(a)의 버스요청 사이클이 그림 5(b)에 비하여 크며, 특히 우선순위가 낮은 마스터 M5 이상에서 싱글 버스의 버스요청 사이클이 크다. 그러나 다중버스는 상대적으로 버스요청 사이클이 작으며, 마스터별 편차도 크지 않다.

그림 4와 그림 5를 통해 우리는 슬레이브의 종류에 따라 싱글버스와 다중버스의 성능 향상의 차이는 있지만, 슬레이브의 종류에 관계없이 다중버스의 성능이 싱



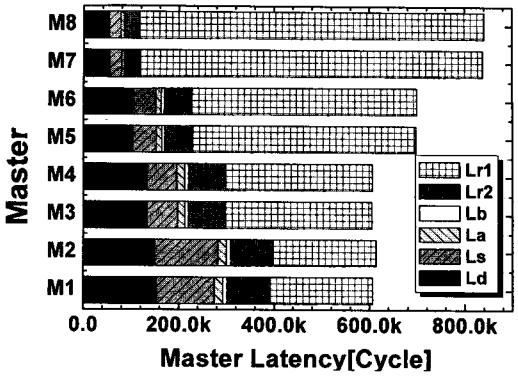
(a)



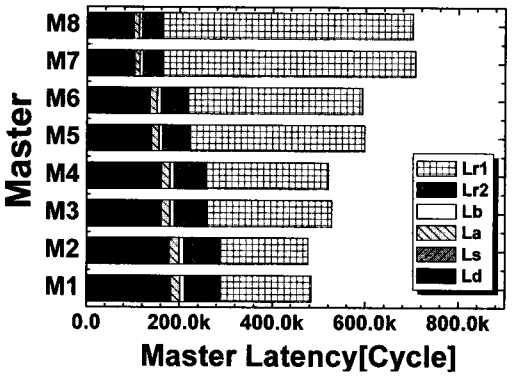
(b)

그림 5. (a) 슬레이브 SDRAM에 따른 버스요청 사이클, (b) 슬레이브 SRAM 또는 레지스터에 따른 버스 요청 사이클

Fig. 5. Bus request cycle due to (a) slave SDRAM and (b) slave SRAM or register.



(a)



(b)

그림 6. (a) 다중버스에서 SDRAM 슬레이브와 (b) SRAM 또는 레지스터 슬레이브의 레이턴시 비교

Fig. 6. (a) Latency comparison between (a) SDRAM and (b) SRAM or register slaves in multiple bus.

표 1. 마스터의 개수에 따른 전체 트랜잭션 사이클과 버스요청 사이클

Table 1. Total transaction cycle and bus request cycle due to the number of master.

Slave	SDRAM				SRAM or register			
	Total Transaction Cycle		Total Bus Request Cycle		Total Transaction Cycle		Total Bus Request Cycle	
Master Number	Single Bus	Multiple Bus	Single Bus (Req)	Multiple Bus (Req1+Req2)	Single Bus	Multiple Bus	Single Bus (Req)	Multiple Bus (Req1+Req2)
2	410824	419634	7	6	480895	470386	4	4
4	606758	686541	96	55	822288	884170	41	18
6	629183	824097	1266	165	957163	1020520	201	66
8	629230	882192	71596	440	985457	1155383	996	237
10	629707	894406	2161553	1520	986110	1217854	995	458
12	630388	896043	4307609	10354	989756	1232694	6534	1296
14	629920	898285	6453398	14826	990070	1241746	48780	4386
16	628480	898024	8605670	2014562	990123	1240359	563239	18448

글버스보다 앞서 있음을 알 수 있다.

표 1은 마스터의 개수에 따른 전체 트랜잭션 사이클과 버스요청 사이클을 보여주고 있다. SDRAM의 경우, 싱글버스의 경우, 마스터의 개수가 증가해도 마스터의 개수가 6개 이상일 때부터는 전체 트랜잭션 사이클이 약 629,000 사이클로 일정해지는 것을 알 수 있으며, 다중버스의 경우도 마스터의 개수가 10개 이상일 때부터는 전체 트랜잭션 사이클이 약 900,000 사이클로 일정해지는

것을 알 수 있다. 속도가 SDRAM에 비해 빠른 SRAM 또는 레지스터의 경우도 싱글버스의 경우 마스터의 개수가 8개 이상일 때부터 전체 트랜잭션 사이클이 약 980,000 사이클로 일정하며 다중버스의 경우는 마스터의 개수가 약 10개 이상일 때부터 1,200,000 사이클로 일정해짐을 보이고 있다.

전체 버스요청 사이클의 경우는 SDRAM이나 SRAM 또는 레지스터에 상관없이 마스터의 개수가 증가할수록 꾸준히 증가하고 있음을 보이고 있다. 결국 데이터 전송은 버스 아키텍처의 성능에 크게 좌우되기 때문에 마스터의 개수가 늘어나고, 버스 요청을 많이 한다고 할지라도 일정 수준에서 포화됨을 알 수 있다.

그림 6은 다중버스에서 슬레이브 종류에 따른 레이턴시를 보여준다. 브리지 통신에 소요되는 총 레이턴시를 식으로 표현하면 식(1)과 같다.

$$L_{tot} = L_{r1} + L_{r2} + L_b + L_a + L_s + L_d \quad (1)$$

여기서 L_{tot}는 브리지통신에 소요되는 총 레이턴시, L_{r1}은 공용버스1과 공용버스2에서 각각 버스요청 할때 소요되는 레이턴시이며, L_{r2}는 브리지를 통과하여 두번째 버스에서 발생하는 버스요청 레이턴시, L_b는 브리지를 통과할 때 소요되는 레이턴시, L_a는 아비터에서 버스중재시 소요되는 레이턴시, L_s는 슬레이브 접근에 소요되는 레이턴시, L_d는 실제 데이터 전송에 소요되는 레이턴시이다.

브리지 레이턴시 L_b는 1 사이클로 정하였는데, 이는 일반적으로 데이터 전송에서 브리지를 경과할 때 1 사이클 소요되는 상황을 감안한 것이다. 그리고 실제상황을 고려하여 아비터 레이턴시 L_a는 1 사이클로 정하였는데, 만약 브리지를 통과할 경우 총 2 사이클이 소요된다.

마스터의 우선순위가 낮아질수록 L_{r1}이 커짐을 알 수 있다. 이는 우선순위가 낮을수록 버스 점유권을 얻기 위해 기다리는 시간이 많이 걸리기 때문이다. 그리고 브리지를 통과하는 통신의 경우 브리지에서 버스요청을 받을 동안 버스는 대기상태에 있다. 본 연구에서는 브리지에서 버스를 요청하는 경우를 아비터에서 가장 빨리 처리하도록 규정하여 브리지에서 버스를 요청하는 레이턴시 L_{r2}는 상대적으로 적은 값을 유지하게 하였다.

그림 6(a)와 그림 6(b)의 가장 큰 차이점은 L_s이다. 일반적으로 SDRAM은 precharge 사이클, refresh 사이클, 로우 컬럼 액세스 사이클 등을 고려하기 때문에 슬레이브에서 긴 레이턴시가 발생하는데 비해, SRAM이나 레지스터의 경우는 슬레이브에서 지연되는 사이클이 없으므로 L_s가 존재하지 않는다. 결국 그림6(a)는 그 영향에 의해 식(1)의 모든 레이턴시가 상대적으로 큰 값을 나타내고 있다.

Ⅲ. 결 론

본 논문에서 우리는 TLM(Transaction Level Model) 방법을 이용하여 마스터의 개수와 슬레이브의 종류에 따른 싱글버스와 다중버스의 성능을 비교 분석하였다. 싱글버스는 마스터의 개수가 6개 이후로는 성능이 일정하였고 다중버스의 경우는 10개까지는 성능이 꾸준히 증가하였다. 그렇기 때문에 싱글버스보다는 다중버스가 성능이 좋음을 검증할 수 있었다. 슬레이브의 경우는 SDRAM과 SRAM에 따른 데이터 트랜잭션 사이클을 비교하여 보았는데, 싱글버스일 경우가 더 많은 마스터의 스타베이션 현상이 발생하였다. 또한 SRAM과 SDRAM에 따른 성능차이를 정량적으로 파악하였으며, 다중버스와 싱글버스의 성능차이와 임계점을 확인하였다. 본 논문은 설계하려는 사양에 따라 효율적인 버스 아키텍처를 선택할 때, 중요한 자료가 될 것으로 생각된다.

참 고 문 헌

[1] K. Lahiri, A. Raghunathan, and S. Dey, "Design Space Exploration for Optimizing On-Chip Communication Architectures", in IEEE Trans. on Computer-Aided Design, pp.952-961, Jun, 2004.

[2] L. Benini and G. D. Micheli, "Networks on chips: A new SoC paradigm," IEEE Comput., vol.35, pp.70-78, Jan. 2002.

[3] M. Gasteier and M. Glesner, "Bus-based communication synthesis on system level," ACM Trans. Design Automation Electron. Syst., vol. 4, no. 1, pp.1-11, 1999.

[4] AMBA TM Specification(AHB) (Rev 2.0), ARM Ltd, May 1999.

[5] K. Sekar, K. Lahiri, A. Raghunathan, and S. Dey, "FLEXBUS: A high performance system-on-chip communication architecture with a dynamically configurable topology", in Proc. Design Autom. Conf., pp.571-574, 2005.

저 자 소 개

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

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