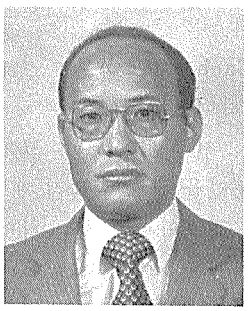


吳 吉 祿
韓國電子通信研究所
주전산기개발본부장 / 工博

다중처리 시스템의 메모리 구조에 관한 연구

1. 서 론

일반적으로 Tightly-coupled된 다중처리 시스템(Multiprocessor System)은 그림 1과 같이 나타낼 수 있지만, Switch의 구현 방법에 따라서 여러 형태의 시스템으로 구현할 수 있다. 즉, 구현 방법에 따라 동시에 두개 이상의 프로세서와 메모리를 연결시킬 수 있는 Cross-bar와 Multistage network이 있고, 한개의 프로세서와 메모리만을 연결할 수 있는 공통버스(Common bus)가 있다. Cross-bar의 경우 높은 연결도(Connectivity)와 전송 속도를 얻을 수 있으나 구현이 어렵고 값이 비싸다. 반면에 공통버스는 연결도와 전송 속도는 낮지만 구현하기가 쉽고 값이 저렴한 장점이 있어 다중처리시스템에서 많이 사용하고 있다. 본 논문에서는 공통버스를 사용하는 다중처리 시스템에 대하여 기술하고자 한다.

공통버스를 사용하는 다중처리 시스템의 경우, 각 프로세서는 메모리의 접근(Access) 과 프로세서 사이의 통신을 위해 버스를 사용해야 한다. 이때 프로세서의 수가 증가하여 버스를 많이 사용하게 되면, 공통버스는 포화 상태가 되어 구조적인 지원없이는 프로세서의 증가에도 불구하고 시스 의 성능은 더 이상 증가할 수 없게 된다. 이런 버스 포화 상태를 막기 위해서 여러 방법이 제안되고 있지만, 본고에서는

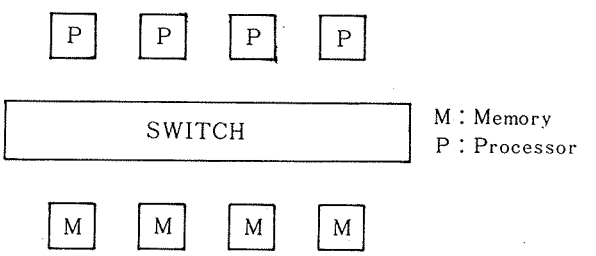


그림 1. Tightly-coupled multiprocessor system

프로세서 사이의 통신 방법으로 공유 메모리(Shared memory) 방식을 사용하는 다중처리 구조 중 프로세서에 Cache메모리를 두는 방법과 Local 메모리를 두는 방법의 특성과 버스이용률(Bus Utilization)에 대해 고찰하고자 한다. 그러므로 본고에서는 Message-based방식에 대한 고찰은 제외한다.

Cache 메모리인 경우 수 Kbyte에서 수십 Kbyte(64Kbyte정도)크기의 빠른 Associative 메모리를 4 byte에서 64byte 정도로 Block (혹은 Line)을 나누어 공유 메모리의 복사를 저장한다. 어느 순간 프로세서가 원하는 데이터가 이 Cache메모리에 없을 때 공통버스를 통하여 공유 메모리를 사용한다. 이 방법은 하드 웨어에 의해 모두 제어되기 때문에 Cache 메모리는 운영체제나 프로그래머에게는 보이지 않는 자원이 된다.

Local 메모리인 경우 수백 Kbyte에서 수 Mbyte크기의 메모리를 프로세서에 할당하여, 그 메모리를 User process의 Context와 운영체제의 Text를 위해 그 프로세서 고유의 영역으로 사용토록 한다. 별도의 공유 메모리는 전체 시스템관리를 위한 운영체제의 Global data structure의 저장이나 프로세스간의 통신을 위한 공유 영역으로 사용한다. 이 방법은 Global data structure를 사용할 때와 프로세서간의 통신 때만 버스를 사용하므로 공통버스의 부담이 줄어들게 된다.

2. Cache메모리와 Local메모리의 특성 및 고려사항

(1) Cache메모리

종래의 Single-processor시스템에서는 메모리 속도가 느려서, 프로세서의 빠른 데이터 요구를 충족시키기 위해 Spatial locality를 강조한 Cache메모리 즉, Block 크기를 크게 하여 Hit ratio를 올리는 형태의 Cache메모리를 설계하였다. 그러나 Cache 메모리의 Block 크기가 클수록, 한정된 Cache 메모리에 올라올 수 있는 Block의 갯수는 상대적으로 적어지므로 Temporal locality에 의한 Hit ratio가 감소하게 되었다. 뿐만 아니라 불필요한 데이터를 전송하는 확률이 높아져 버스 사용이 증가하게 된다. 이런 문제 때문에 최근에는 다중처리 시스템을 설계할 때 버스 사용을 줄이는 방향으로 Cache 메모리를 설계하고 있다.

다중처리 시스템의 Cache 메모리를 설계할 때 다음 사항들을 고려하여야 한다.

가. 전체 Cache크기 : 전체 Cache의 크기는 데이터가 들어가는 부분만을(Tag 메모리 제외) 의미하며, 다음의 어느 것보다 중요한 의미를 갖는다. 그 이유는 전체 Cache크기가 정해지면 Hit ratio가 거의 정해지기 때문이다. 보통 4 Kbyte(VAX 11/750[2])부터 32Kbyte(Multimax[3]) 정도가 일반적이고, 2 Kbyte 이상이 되면 95% 이상의 (Data consistency overhead를 제외하고) hit ratio를 얻을 수 있다.

나. Block(혹은 Line)크기 : Block과 Line을 구별하는 경우도 있으나, 대부분 공유 메모리와 Cache 메모리 사이에 데이터 전달의 기본 단위를 의미한다. 보통 4 byte부터 64byte 정도가 일반적이다(256이나 512byte로 Page 단위와 같은 것도 있음[4]). 앞에서 언급한 것과 같이 Block크기는 Temporal, Spatial Locality와 버스 사용에 영향을 많이 주는 요인이 되므로 버스의 성능, 프로세서의 특성 등을 고려하여 결정해야 한다.

다. Consistency 문제 : Cache 메모리에서 Consistency 문제는 메모리 Write 사이클에서 발생한다. 이 문제를 해결하는 방법으로는 여러가지가 있다. 공유 메모리에 Write할 때 무조건 공유 메모리에 Write하므로써 Cache메모리는 항상 공유 메모리에 최근의 데이터를 갖게 하고, 복사를 갖고 있던 다른 Cache 메모리는 버스를 주시하고 있다가 Write가 발생하면 해당된 Block을 무효화(Invalidate) 시키는 Write Through방법[5], Write가 발생하면 즉시 공유 메모리에 쓰지 않고 변한 데이터를 Cache에만 저장했다가 나중에 공유 메모리로 옮기는 Copy-Back방법[5], 그리고 Write-Once[6], Consistency Protocol[7]방법 등이 있다. 위에

서 언급한 방법의 순서대로 버스 사용은 적어지지만 구현 방법이 복잡해지기 때문에 아직도 Write-though방법을 많이 사용하고 있다. 앞으로 본고에서 언급하는 Cache 메모리는 Write-through방법에 국한된다.

라. 기타 : 그밖에 Associative 메모리를 어떻게 구현할 것인가(Directed map, Set associative, Fully associative 등), 인스트럭션과 데이터 Cache를 구별하여 하드웨어를 구현할 것인가, Virtual 혹은 Physical 어드레스로 Cache를 구현할 것인가, 등이 Cache 메모리를 설계할 때 고려해야 할 사항들이다.

(2) Local 메모리

Local 메모리 방법은 대규모 다중처리 시스템이나 병렬처리 시스템이나 Loosely-coupled된 시스템에서 많이 사용되어 왔다. 이 방법은 공유 메모리를 사용하지 않고 Message passing으로 프로세서간의 통신과 데이타 교환을 하고 있다. 하지만 최근에는 Cache 메모리와 Local 메모리 방법을 병행하여 UNIX 운영체제를 기반으로 하는 다중처리 시스템을 연구하고 있다. [12]. Local 메모리 방법을 사용하여 다중처리 시스템을 설계할 때 고려사항은 다음과 같다.

가. Local 메모리 크기 : Local 메모리의 크기는 Cache 메모리의 크기처럼 Hit ratio에 영향을 받는 것이 아니라 사용자 프로그램의 크기에 따라 주로 결정된다.

나. Load balancing 문제 : 다중처리 시스템에서는 Dynamic load balancing을 위하여 프로세서 사이에 Context switching을 해야 한다. Local 메모리에는 User process의 Context가 저장되어 있기 때문에 Context switching이 일어날 때 버스를 많이 사용하게 된다. 즉, Dynamic load balancing을 위한 알고리즘이나 Context switching 방법의 개선이 필요하다.

다. 공유 데이타의 처리 : 프로세서 사이의 통신이나 프로그램의 수행을 쉽게하기 위해 공유 데이타가 편리한 경우가 있다. 이 경우 공유 데이타를 처리하기 위하여 새로운 공유 메모리를 두게 되면 매우 복잡한 메모리 시스템을 구현하여야 한다.

라. 운영체제와의 관계 : Cache 메모리 경우 대부분 하드웨어로 제어되기 때문에 운영체제에서 따로 Cache 메모리를 관리할 필요가 없지만, Local 메모리인 경우 운영체제가 관리하여야 하기 때문에 운영체제가 더욱 복잡해진다.

3. Cache 메모리의 버스 이용률

Cache 메모리의 Miss ratio (1-hit ratio)가 정해지면 버스이용률(Bus Utilization)은 다음의 식으로 나타낼 수 있다.

$$\text{Bus Utilization (BU)} = \frac{\text{단위 인스트럭션을 수행할 때 이용하는 버스의 시간}}{\text{단위 인스트럭션의 총 수행 시간}} \quad \text{(공식 1)}$$

Cache 메모리를 사용하는 시스템에서 버스를 사용하게 되는 경우는 두가지가 있다. Cache miss가 발생했을 때 버스사용과 Consistency 문제의 해결을 위한 버스의 사용이 그것이다. (공식 2)는 (공식 1)을 Write-Through 방법을 사용한 Cache 메모리의 버스 이용률 계산을 구체화한 것이다. [4]

$$BU = \frac{(mr*mbc+wf*wbc)*rpi}{c+(1-mr-wf)*hc*rpi+(mr*mc+wf*wc)*rpi} \quad \text{(공식 2)}$$

c : 프로세서의 내부적인 수행 시간
hc : hit cost
mr : miss ratio
mbc : cache miss 처리하는 데 사용하는 버스의 시간
mc : cache miss 처리하는 데 총 걸리는 시간
wf : write fraction
wc : write cycle time
wbc : write cycle 때 버스의 사용 시간
rpi : reference per instruction

위 (공식 2)의 변수들 중, 프로세서의 내부 수행 시간은 프로세서에 따라 정해지며, Miss ratio는 2장에서 언급한 설계변수에 의해 정해지며, Consistency 문제 해결을 위한 Write fraction도 그 방법에 따라 정해진다. 그리고 인

스트럭션과 데이타의 비율인 rpi는 프로세서가 정해지면 그 평균치로 구할 수 있다.

Cache 메모리의 버스 이용률을 측정을 위해 세 시스템을 비교해 보았다. 두개의 시스템은 상품화되어 있는 Encore사의 Multimax 시스템 [3]과 Sequent사의 Balance 시스템 모델 8000 [13]이고, 나머지 하나는 앞으로 개발할 모델 시스템이다. Encore의 Multimax 시스템은 Nano 버스(100 Mbyte/sec)라는 공통버스를 사용하고 있으며, 두 개의 NS32032를 내장하고 있는 프로세서 보드당 32Kbyte의 Cache 메모리를 갖고 있다. Balance 시스템의 경우 비교적 낮은 버스 전송 속도(27Mbyte/sec)의 공통버스를 사용하고 있으며, 두개의 NS32032를 내장하고 있는 프로세서 보드당 16Kbyte(8Kbyte/프로세서)의 Cache를 갖고 있다. 모델 시스템은 VME 버스(20Mbyte/sec)에 68020이 내장된 프로세스 보드당 8Kbyte의 Cache를 갖는 가상적인 시스템을 고려했다.

표 1은 이들 세 시스템의 변수들을 [8]과 [9]에 나타나 있는 기본 자료를 기준으로 데이터를 작성한 것이다. 그리고 c는 통계 평균치인 213 ns로, hc는 180ns로 추정하였다. 이 데이터를 기준으로 각 프로세서당 버스 이용률을 계산한 결과, Multimax 시스템의 경우 프로세서당 3% 정도 버스를 사용하고 있어 I/O Controller 가 버스를 사용하는 것을 감안해도 20개 이상의 프로세서가(버스의 이용률 측면만을 고려할 때) 동시에 수행되더라도 버스가 포화되지 않을 것이라고 예측할 수 있다. Balance 시스템 경우 프로세서당 7.3% 정도 버스를 사용하기 때문에 10개 정도의 프로세서가 허용될 수 있으며, 모델 시스템 경우 프로세서당 11.8% 정도로 버스를 사용하기 때문에 I/O에 의한 버스 사용을 감안한다면 6개 정도의 프로세서로 버스는 포화 상태가 된다.

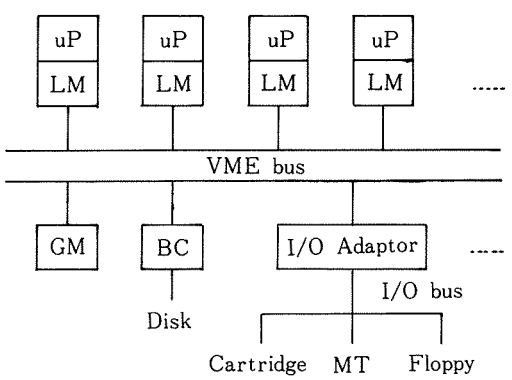
표 1. 버스 이용률 계산을 위한 변수 및 프로세서당 버스 이용률

	mr	mc	mbc	wc	wbc	wf	rpi	BU
Multimax	1%	240ns	80ns	240ns	80ns	20%	2.4	3.0%
Balance	4%	380%	200ns	280ns	100ns	20%	1.2	7.3%
Model	4%	380ns	200ns	380ns	200ns	20%	1.2	11.8%

4. Local 메모리의 버스 이용률

그림 2.는 버스의 부담을 줄이기 위한 한 방법으로 Local 메모리 방법을 사용한 모델 시스템의 구조를 나타내고 있다. Local 메모리의 크기는 현재의 반도체 기술을 감안할 때 1Mbyte에서 수 Mbyte 정도가 가능하며, Dual port로 구성하여 프로세서가 Local 메모리를 사용하는 것과 버스가 Local 메모리에 데이터를 전송하는 것이 동시에 이루어질 수 있다.

앞에서 언급한대로 공유 메모리에는 운영체제의 데이터 Structure를 두어 각 프로세서가 공유토록 하며, Local 메모리에는 운영체제의 인스트럭션 부분(text)과 User process의 모든 부분으로 사용된다. I/O 부분은 일반 시스템과 같다고 가정한다. 프로세서와 Local 메모리 사이에 Cache가 있을 수 있으나, 본고에서는 버스의 Contention을 분석하므로 고려에서 제외한다.



LM : Local Memory
GM : Grobal Shared Memory
BC : Buffer Cache
uP : Microprocessor

그림 2. Local 메모리 방법을 사용한 모델 시스템

Cache 메모리 방법은 하드웨어에서 대부분 관리하기 때문에 운영체제와는 밀접하게 연관되어 있지 않지만, Local 메모리 방법은 운영체

제가 Local 메모리를 관리해야 하기 때문에 운영체제를 떠나서 생각할 수 없다. 그래서 여기에서는 UNIX 운영체제를 기반으로 하는 시스템을 기준으로 Local 메모리 방법의 버스 이용률에 대하여 고찰하기로 한다.

이 방법에서는 앞에서 정의한 버스 이용률을 계산하는 공식(공식 2)을 그대로 적용할 수 없고 새로운 측면에서 공식을 만들어야 한다. 인스트럭션의 평균 수행 시간(Et)은 평균 내부 수행 시간과, Local 메모리 평균접근시간과, 공유 메모리 평균접근시간의 합이 되며, 그리고 평균 버스 사용시간(Bt)은 공유 메모리 평균 접근시간이 된다. 이 두 평균시간을 이용하여 Local 메모리 방식의 버스 이용률(BU)을 구체화하면 (공식 3)과 같이 나타낼 수 있다.

Et = 프로세서 내부 평균 수행 시간 + Local 메모리 평균접근시간 + 공유 메모리 평균접근시간

Bt = 공유 메모리 평균접근시간

$$BU = \frac{Bt}{Et} = \frac{(fg*gc)*rpi}{c+[(1-fg)*lc+fg*gc]*rpi} \quad (공식 3)$$

c : 프로세스의 내부적인 수행 시간
lc : local 메모리 접근 시간
gc : global 공유 메모리 접근 시간
fg : global 공유 메모리의 접근 확률
rpi : reference per instruction

여기서 c, lc, gc, rpi는 하드웨어에 의해서 결정이 되지만 fg는 운영체제와 밀접하게 연관되어 결정되기 때문에 우선 버스 이용률을 fg의 함수로 나타내면(공식 4)가 되고, 그 함수를 그림으로 나타내면 그림 3과 같이 나타난다.

이때 c=213ns, lc=180ns, gc=380ns, rpi=1.2로 가정하였다.

$$BU = \frac{456fg}{240fg+429} \quad (공식 4)$$

이런 모델에서 fg란 시스템 전체 메모리 중 프로세서 자신의 Local 메모리를 제외한 공유 메모리와 다른 프로세서의 Local메모리를 접근할 빈도의 확률을 말한다. UNIX 운영 체제를 기반으로 하는 다중처리 시스템을 개발할 때를 가정하고 fg와 버스 이용률을 유추하기 위해 UNIX 운영체제의 특성을 살펴보면,

가) Kernel의 데이터 구조는 대부분 Global로 선언되어 있기 때문에 모든 프로세서에서 공유해야 한다.

나) 프로세스 내의 Local 변수는 속도를 고려하여 Stack에 두지 않고 주로 레지스터 변수로 선언된다.

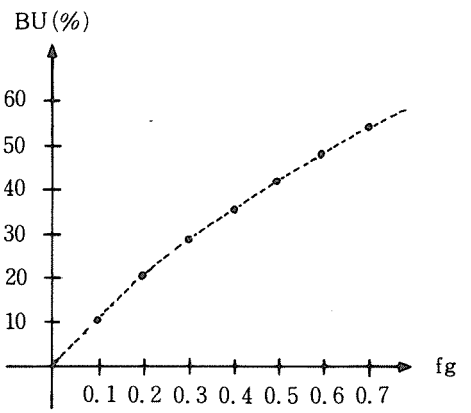


그림 3. fg에 따른 버스 이용률

다) 전항의 이유로 Local 변수의 사용을 위해 Stack을 사용하지 않기 때문에 일반 프로그램에 비하여 Function call과 Return 시점에서 레지스터의 Save와 Restore가 Stack에서 많이 일어난다.

라) Fuction의 매개 변수가 일반 프로그램에 비하여 많으므로 매개변수의 전달을 위해 Stack의 사용이 많다.

마) Global 변수는 기계어로 변환될 경우 Absolute addressing mode로 대치됨으로 Global 변수가 많은 Kernel은 인스트럭션 対 데이터의 비율에 있어 인스트럭션 비율이 일반적인 통계보다 커진다.

위와 같은 특징을 토대로 UNIX의 Iget( )와 Clock( ) Function을 Static하게 분석한 결과 Local 対 Global 데이타의 비율은 약 2대 8 정

도로 나타났다. 그리고 Kernel 모드와 User 모드로 수행되는 비는 4 : 6 정도로 알려져 있으며, 인스트럭션 対 데이터의 메모리 사용 비율도 50대50 정도로 가정할 수 있다.

위의 3가지 가정의 수치를 기준으로 (공식 4)의 fg와 버스 이용률을 구해보면 표2와 같다. 여기서 fg란 Kernel 데이타 중 Global 데이타를 접근할 확률을 의미한다. Case 2는 UNIX 운영체제가 아닌 다중처리 운영 체제의 경우, User 모드 : Kernel 모드의 수행비를 5 : 5 정도로, 그리고 Local 데이타 : Global 데이타의 비율을 1 : 9 정도로 높여 가정한 경우이다.

표2. Local 메모리 모델의 fg와 버스 이용률 계산치

	Case 1	Case 2
User : Kernel	0.6 : 0.4	0.5 : 0.5
Inst.: Data	0.5 : 0.5	0.5 : 0.5
Local : Global	0.2 : 0.8	0.1 : 0.9
fg	0.160	0.225
BU	15.6%	21.2%

위 표에서 보는 바와 같이 Local 메모리 모델인 경우, 프로세서당 15%에서 21% 정도로 버스를 사용하기 때문에 I/O에 의한 버스 사용을 감안한다면 VME 버스를 사용하는 다중처리 시스템에서는 4개 이상의 프로세스를 연결시킬 수 없다는 결론을 얻을 수 있다.

5. 결 론

본고에서는 UNIX 운영체제를 기반으로 하는 다중처리 시스템을 개발할 때 각 프로세스에 허용되는 버스 이용률을 Cache 메모리 모델과 Local 메모리 모델에 대해서 정량적으로 분석하여, 버스 이용률에 의한 공통 버스의 포화상태를 야기시키는 프로세서 개수를 추정하였다. 그 결과 VME와 같은 저속 버스를 사용하는 다중처리 시스템의 경우 다음의 결론을 얻을 수 있었다.

첫째, Local 메모리 모델의 경우 운영체제와 밀접하게 연관되기 때문에 UNIX와 같은 Global 데이타가 많은 운영체제에서는, 특별한 하드웨어나 운영체제의 지원없이는 프로세서가 4개 정도만 되어도 버스포화상태를 초래하여 시스템 성능이 더 이상 증가하지 않는다. UNIX 운영체제와의 관계를 감안해 보면 Local 메모리 모델은 Cache 메모리 모델보다 부적합하다.

둘째, 운영체제와 밀접한 관계가 없는 Cache 메모리 모델의 경우에도 특별한 하드웨어의 지원없이는 프로세서가 10개 정도만 되어도 버스 포화상태를 초래한다.

세째, 이 때문에 Cache 메모리 모델의 경우 버스 포화상태를 초래하지 않고 동시에 20개 정도의 프로세서를 수행시키려면 Multimax 시스템처럼 전송 속도가 빠른 버스를 사용하든지, Balance 시스템처럼 SLIC[13]와 같은 특별한 하드웨어의 지원이 있어야 가능할 것으로 예상된다.

마지막으로 본고에서 미흡한 점은, Cache 메모리 모델의 경우 비교적 정확한 수치로 분석하였지만, Local 메모리 모델 경우 운영체제와 밀접한 관계가 있어 많은 부분이 가정과 경험에 의한 수치로 분석하였다. 앞으로 좀 더 정확한 수치를 얻기 위한 연구가 필요하며, 전송 속도가 빠른 버스를 사용한 다중처리 시스템에서 Cache 메모리 모델의 연구가 계속되어야 할 것이다.

참 고 문 헌

1. Philip Bitar, Alvin M.Despain. "Multiprocessor Cache Synchronization-Issues. Innovations, Evolution," Proc. 12th Annu. Symp. on CA, 1985.
2. VAX Hardware Handbook, DEC, 1982-83.
3. Multimax Technical Summary, Encore Computer Corp, 1985.
4. David R.Cheriton, Gert A.Slavenburg, Patrick D.Boyle, "Software-Controlled Caches in the VMP Multiprocessor," Proc. 13th Annu. Symp. on CA, 1986.
5. A.V.Pohm, O.P.Agrawal, High Speed Memory Systems, Prenctice-Hall, 1983.
6. Goodman J., "Using Cache Memories to Reduce Processor-Memory Traffic," Proc. 10th

Annu. Symp. on CA, 1983.
7. R.H.Katz, S.J.Eggers, D.A.Wood, C.L.Perkins, R.G.Sheldon, "Implementing A Cache Consistency Protocol," Proc. 12th Annu. Symp. on CA, 1985
8. Doug MacGregor, Jon Rubinstein, "A Performance Analysis of MC68020-based Systems," IEEE MICRO, 1985.
9. Anant Agrawal, Richard L. Sites, Mark Horowitz, "ATUM : A New Technique for Capturing Address Traces Using Microcode," Proc. 13th Symp. on CA, 1986
10. Bob Beck, Bob Kasten, "VLSI Assist in Building a Multiprocessor UNIX System," Proc. of 1985 Summer USENIX Conf., 1985.
11. Alan Jay Smith, "Cache Memories," ACM Computing Surveys, 1982. 9.
12. M.D. Janssens, J.K. Annot, and A.J. Van De Goor, "Adapting UNIX for a Multiprocessor Environment," CACM, Vol. 29, No. 9, Sep. 1986.
13. G. Fielland and D. Rodgers, "32-bit Computer System Shares Load Equally Among Up To 12 Processors," Electronic Design, Sep. 6, 1984.
14. 한국전자 통신연구소, Multiprocessor 컴퓨터 시스템에 관한 연구 최종보고서, 과학 기술처, 1987. 5.

用語解說

3次元 Device

Transistor 등의 素子를 3次元(立体)的으로 배치한 集積回路. 현재의 집적회로는 그 소자가 2차원적으로 배치되어 최소 線幅도 1㎛를 밑도는 것까지 미세화시켜 고밀도화를 도모한다. 그러나 2次元 배치에 의한 집적화에도 한계가 있어서, 配線 거리도 단축시킬 수 있고 고밀도화에 적합한 3次元 Device가 개발되고 있다.

3次元 Device를 형성하는 기본 기술은 S_iO_2에 Polysilicon膜을 結晶으로 해서 Transistor를 형성하는 SOI(Silicon on Insulator) 기술이다. 이 기술은 70년대말부터 80년대초에 걸쳐서 Laser 再結晶法에 의해 實用이 되었다. 이에 따라 Amorphous 絶縁膜上에 堆積한 Polysilicon을 Laser Beam으로 한번 溶融시킨 다음 固化시키면, 結晶의 크기는 100배 이상이 되고 MOS FET의 電子 移動度도 Bulk의 Silicon Wafer와 같은 値를 얻을 수 있다.

메사추세츠 工科大学은 79년에 절연막을 지닌 3次元 집적회로의 구상을 발표하고 80년에는 n形 基板上에 MOS FET를 만들고, 그 Gate上에 Gate 酸化膜을 형성시켜, 그 위에 Polysilicon을 설치 Laser Anneal에 의해 再結晶시킨 MOS를 만들었다. 83년에는 Bulk의 p Channel MOS FET와 그 위에 만든 SOI를 조합시킨 3차원 Inverter를 7段으로 접속한 본격적인 3차원 Inverter를 개발하였다.

최근에는 3차원 SOI MOS Device로서 3층 구조의 것이 富士通과 三菱電機에 의해 발표되었다. 前者의 그것은 SRAM에서, 6Transistor로부터 되는 CMOS Memory Cell을 이용한 것으로 Free Frop p Channel CMOS FET는 1階의 Bulk Silicon에, n Channel MOS FET는 2階의 SOI층에, n Channel MOS FET로부터 되는 Transfer Gate는 3階의 SOI에 만든다. 後者는 1, 2, 3階와 n Channel MOS FET를 조합시킨 것으로 각 층간은 CVD 酸化膜으로 메워져 平坦化된다.

3차원 Device의 이점은 高集積化·高密度化 외에 階層에 응하여 적정한 素子를 조합시킬 수 있다. SOI이므로 漂遊容量은 작고, 고속이며, 並行·並列 처리에 적합하며, 脳 및 視神経의 인식 기구와 유사한 階層 처리도 가능하다.

CMOS (Complementary Metal Oxide Semi-conductor (相補性金属酸化膜半導体))

MOS IC중에서 動作 속도(演算 속도)는 늦지만, 消費電力을 아주 작게 한 IC. 電卓, 손목時計 등 휴대용 商品에 사용되는 일이 많다.