MMU (Memory Management Unit)

http://recipes.egloos.com/5232056

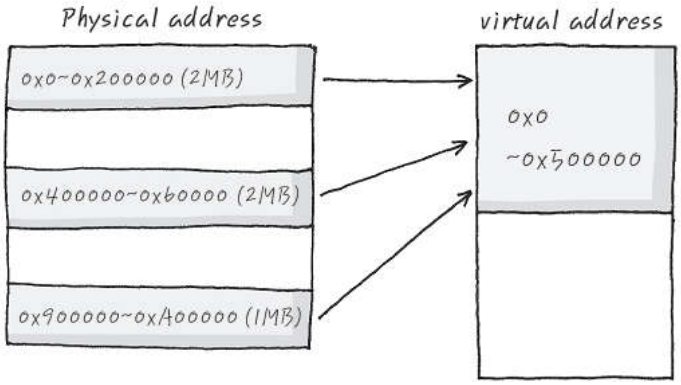
MMU는 CPU의 Memory 주소를 감쪽같이 속이는 거짓말쟁이에요. MMU는 표현하고 행동해요. CPU가 Memory를 Access할 때 마다 주소를 속인답니다. 주소를 속여서 어떻게 하느냐, Physical Address와 Virtual Address (Logical Address라고도 부르죠)의 Mapping을 자기가 갖고서 장난을 치는 거죠. CPU는 자기가 속았는지도 몰라요~

MMU를 왜 사용하느냐,
MMU를 사용함으로써, 도대체 System에 무슨 이득이 있을까요.

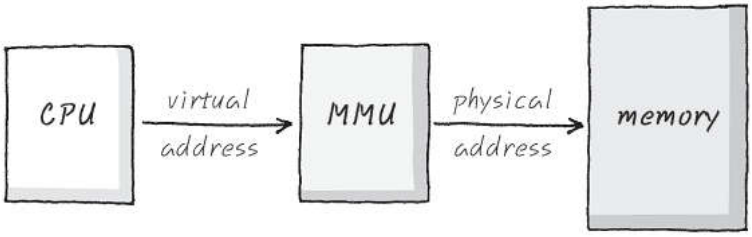
Task 마다 또는 Program마다 똑같은 주소를 사용해도 되게 만들어 주는 거에요. 모든 Task는 0x0~0x3000 번지까지 자리 잡도록 Compile하더라도, MMU를 사용한다면 문제 없답니다. 왜냐! Physical하게는 0x3000~0x5FFF번에 올라와 있는 A Task나, 0x6000~0x8FFF에 올라와 있는 B Task나 MMU가 장난치기만 한다면 0x3000~0x5FFF도 0x0~0x3000처럼, 0x6000~0x8FFF도 0x0~0x3000처럼 장난 칠 수 있다는 말이에요.

또! 두 개의 CS를 사용하는 RAM의 물리적 주소가, 0x00000~0x10000 CS1, 0x20000~0x30000 CS2 이렇게 나뉘어져 있어도, MMU만 잘 사용하면, 0x0000~0x20000 으로 연속적인 것처럼 장난 칠 수 있다는 말이죠.

또 어디다 이용할 수 있느냐! 하면, Physical Address상에서는 쓰고 남은 조각조각 난 Memory들을 한데 모아서 마치 연속적인 Memory처럼 사용할 수도 있지요. Address에 관한 한 MMU를 사용하면 내 멋대로 조작할 수 있어요. 이런데 어울리는 컨셉이 있죠.



뭐 이런건데요, 잘 보시면 Physical Memory의 주소를 마구 다른 주소로 바꿔 버릴 수 있는 거에요.

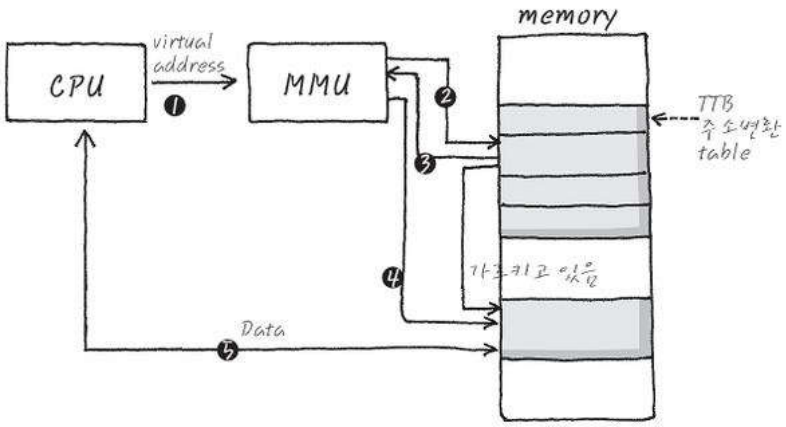


그러니까, 개념으로만 보면, CPU는 Virtual Address를 발생시키면, MMU가 Physical Memory의 주소로 바꿔줘서 실제 Physical Memory를 Access한다. 뭐 이런 셈인 거죠. 그리고, MMU를 사용해서, Memory 영역의 특성을 조작할 수도 있어요. 특성이라는 게 Cache랑 많이 연관되는데, 그 중에서도 Cache영역과 Non Cache영역, 그리고 쓰기 관련해서는 Write Bufferable과 Non Write Bufferable 영역으로 나눌 수도 있고요. 심지어, 어떤 영역은 Write를 못하게 Readonly 영역으로도 지정할 수 있어요. 오호라, 너무나 편리한 녀석이죠. MMU를 잘 다루려면 또 용어를 잘 알아야 하겠지요. 자, 생각해 보시죠. CPU에게서 받은 Virtual Address를 Physical Memory로 Mapping하려면 뭐가 필요할까요? 한마디로 주소 가지고 장난치려면 뭔가 있어야겠죠.

ⓐ Virtual Address와 Physical Address를 연결해 주는 Table이 있어야겠죠.
ⓑ Table이 존재하는 위치를 알아야겠죠.

ⓐ를 Page Table이라고 부르고요, ⓑ TTB (Translation Table Base Address)라고 해서 MMU의 Register중 하나에 저장되어 있어요. 여기에서 한 가지 알아둬야 할 것은 Virtual Address와 Physical Address를 바꿔주는 Table은 외부 Memory에 존재한다는 거죠. (외부 Memory라 함은 느린 외부 Main Memory 일 수도 있고, 빠른 성능을 위해서 MCU 내부에 TCM Memory에 위치시킬 수도 있어요)
여하튼,
① CPU는 Memory의 어딘가를 Access하기 위해서 Virtual Address를 발생해요.

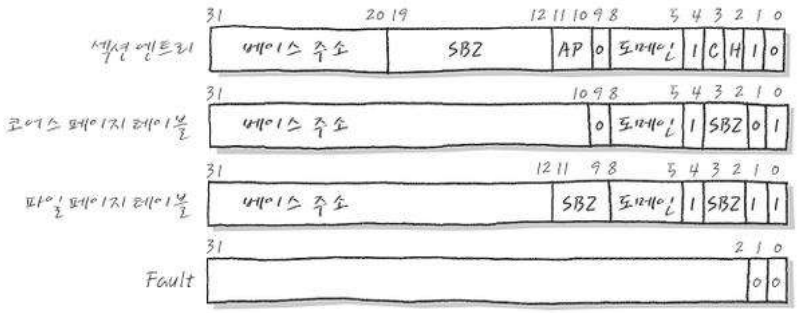
② MMU는 이 Virtual Address를 받아서 Memory의 TTB에서부터 시작해서
존재하는 Page Table을 Access하고요,
③ 찾아간 Page Talbe안에 Physical 주소를 찾아내어 주소 신호를 발생하고,
④ Memory는 해당 Physical 주소안에 Data를 출력해서 CPU에게 전달하는 과정인거죠.
⑤ Data가 CPU에 전달 됨.



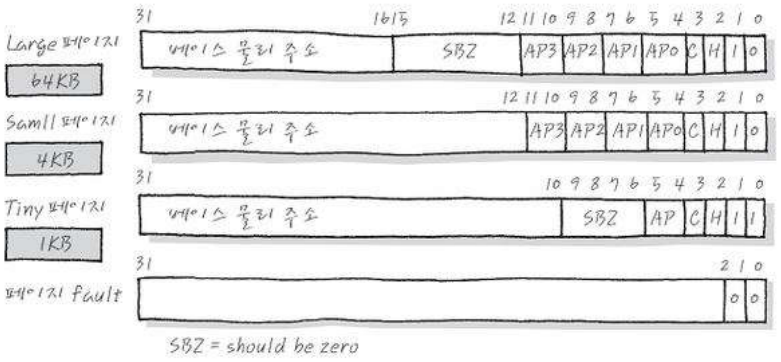
오, 간단하네요. 여기에서 한가지 더. 일단 Page Table 크기는 얼마나 될까..?하는 의문이 들고요, Virtual Address는 어떻게 Physical Address로 변환 될까 하는 의문이 드네요. 일단 Virtual Address는 32Bit Address System (Register 크기가 32Bit니까 그래요) 이므로 2^32 = 4GB를 나타낼 수 있는 거에요. 이때! Page Table 역시 4GB를 나타낼 수 있어야겠지요. 이때! Page Table의 한 개 Entry (32Bit)는 1MB씩을 가리킬 수 있고요 - 그림의 한 칸 이라고 보믄 되고요 32Bit 크기에요 - 1MB씩 가리킬 수 있으니까 이런 Entry가 4096개 필요 한 거죠.

4096 * 32Bit = 16KB 가 Page Table의 기본 크기가 되는 거랍니다. 우후후.

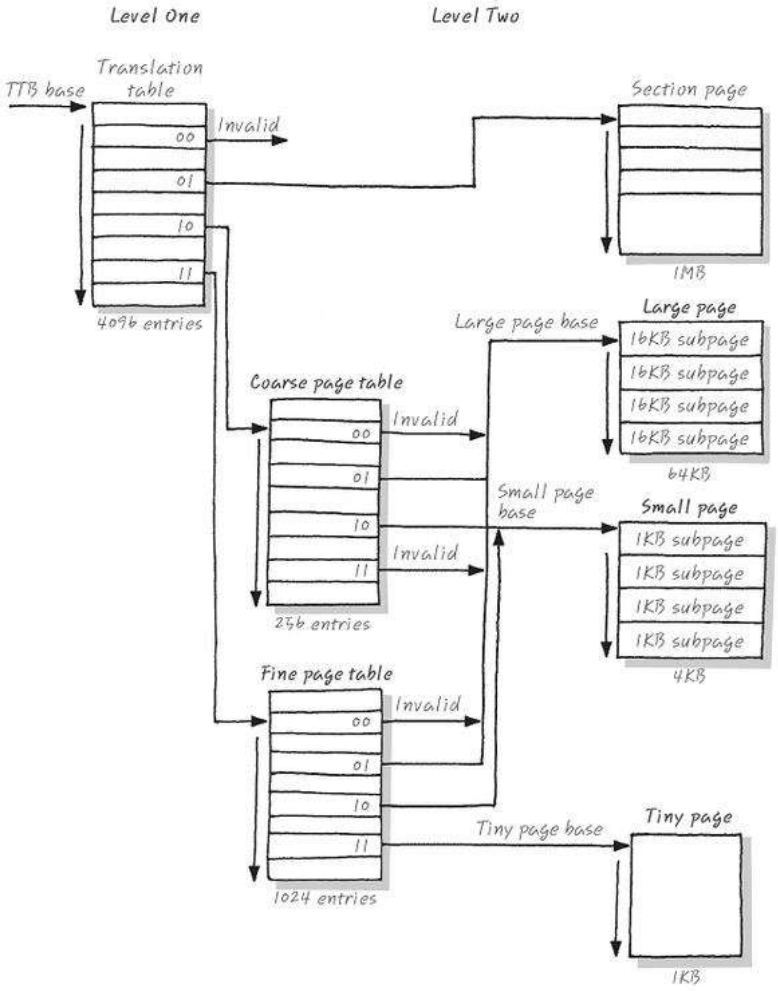
그러니까 Virtual Address는 무조건 1MB 단위로 기본 설정이 되어 있고요 이런 걸 Section이라고 불러요. 그런데 무조건 1MB씩만 할당 가능하다면 너무 답답하겠지요. 그래서 더 작은 단위로도 나눌 수 있는 Option이 있는데요. 그걸 64KB나, 4KB 단위로 나눌 수 있다고 해서 Fine, Coarse Page Table이라고 불러요. 자, 1MB 단위로 4096개로 나누어진 Page Table을 Level 1 Page Table이라고 부르고요, 각 32Bit 씩의 단위를 Level 1 Page Table Entry라고 부르죠.



이런 Page Table Entry는 요렇게 4가지 종류를 갖고요. 모두 1MB를 마크하고 있고요. Coarse Page Table과 Fine Page Table은 그 1MB를 더 나눌 수 있게 해주니까, 또 다른 Page Table을 가리키는 거에요. 그런 또 다른 Page Table을 Level 2 Page Table이라고 부르는 거구요. 어렵다. AP니 Domain이니 하는 거는 여기서 일단 Skip하고요. 너무 복잡하니까. Coarse Page Table이나, Fine Page Table이 가리키는 Level 2 Page Table은 어떻게 생겨먹었는지 한번 볼까요?



우후~ Level 2로 넘어 가면 더 작은 Page들에 대한 Page Table Entry가 있을 테죠. 자 이걸 전체적인 그림으로 보면,



여기서 예를 잠시 보고 넘어가면 상당히 쉽죠. 자, 우리 TTB부터 한번 볼까요. 어떤 System의 TTB가 0xA41400 이라고 치고요. TTB가 00A24000 인 경우에, 실제 0x00A24000을 보면, 1MB로 4096개의 Page Table이 있음을 확인할 수 있어요. 그럼, 이 MMU Table을 어떻게 해석할 것이냐 하면 한 칸당 1MB라고 보시면 됩니다요. 아래는 실제 PageTable이 어떻게 Virtual Address와 Physical Address를 Mapping 하는지를 한눈에 결과를 미리 알 수 있을 거에요. (Size 포함 되어 있어요)

```
Virtual address_______|_physical_____________|_size____|
  C:00000000--008FFFFF|
  C:00900000--009FFFFF|  A:00900000--009FFFFF| 00100000
  C:00A00000--00B00FFF|
  C:00B01000--00B0FFFF|  A:00B01000--00B0FFFF| 00001000|
  C:00B10000--00BFFFFF|  A:00B10000--00BFFFFF| 00010000|
  C:00C00000--014FFFFF|  A:00C00000--014FFFFF| 00100000|
  C:01500000--0153FFFF|  A:01500000--0153FFFF| 00010000|
  C:01540000--01546FFF|  A:01540000--01546FFF| 00001000|
  C:01547000--01EFFFFF|
  C:01F00000--01FFFFFF|  A:01F00000--01FFFFFF| 00100000|
  C:02000000--16CFFFFF|
  C:16D00000--177FFFFF|  A:16D00000--177FFFFF| 00100000|
  C:17800000--178BFFFF|  A:17800000--178BFFFF| 00010000|
  C:178C0000--178CFFFF|  A:178C0000--178CFFFF| 00001000|
  C:178D0000--1790FFFF|  A:178D0000--1790FFFF| 00010000|
  C:17910000--1791EFFF|  A:17910000--1791EFFF| 00001000|
  C:1791F000--B0053FFF|
  C:B0054000--B0055FFF|  A:00AC2000--00AC3FFF| 00001000|
  C:B0056000--B005FFFF|
  C:B0060000--B006FFFF|  A:01550000--0155FFFF| 00010000|
  C:B0070000--B007CFFF|
  C:B007D000--B007DFFF|  A:0156A000--0156AFFF| 00001000
  C:B007E000--B007FFFF|
  C:B0080000--B00FFFFF|  A:01580000--015FFFFF| 00010000
```

요런 Virtual Address ↔ Physical Address의 Mapping이 어떻게 해서 나오게 되는지 알아보시도록 하시죠. 요 녀석이 MMU page table이구요. Level 1 Page Table이에요.

```
___address__|________0_0123
ASD:00A24000|>00000000 ....  ①
ASD:00A24004| 00000000 ....
ASD:00A24008| 00000000 ....
ASD:00A2400C| 00000000 ....
ASD:00A24010| 00000000 ....
ASD:00A24014| 00000000 ....
ASD:00A24018| 00000000 ....
ASD:00A2401C| 00000000 ....
ASD:00A24020| 00000000 ....
ASD:00A24024| 00900D62 b...   ② → 00000000100100000000110101100010
ASD:00A24028| 00000000 ....
ASD:00A2402C| 00A41561 a...   ③ → 00000000101001000001010101100001
ASD:00A24030| 00C00D6A j...   ④ → 00000000110000000000110101101010
ASD:00A24034| 00D00D6A j...
ASD:00A24038| 00E00D6A j...
ASD:00A2403C| 00F00D6A j...
ASD:00A24040| 01000D6A j...
ASD:00A24044| 01100D6A j...
ASD:00A24048| 01200D6A j. .
ASD:00A2404C| 01300D6A j.0.
ASD:00A24050| 01400D6A j.@.    ⑤ ~ 요기까지~
ASD:00A24054| 00A41961 a...
ASD:00A24058| 00000000 ....
ASD:00A2405C| 00000000 ....
ASD:00A24060| 00000000 ....
ASD:00A24064| 00000000 ....
```

잘 보시면 앞에서부터 9칸이 비어 있는 걸 볼 수 있을 테죠. 이 9칸은 아무런 Physical Address에 Mapping되어 있지 않아요.

①

```
___address__|________0_0123
ASD:00A24000|>00000000 ....   0~ 1MB
ASD:00A24004| 00000000 ....   1~ 2MB
ASD:00A24008| 00000000 ....   2~ 3MB
ASD:00A2400C| 00000000 ....   3~ 4MB
ASD:00A24010| 00000000 ....   4~ 5MB
ASD:00A24014| 00000000 ....   6~ 7MB
ASD:00A24018| 00000000 ....   7~ 8MB
ASD:00A2401C| 00000000 ....   8~ 9MB
ASD:00A24020| 00000000 ....   9~ 10MB
```

실제 Vitual Address 영역을 보면,

```
Virtual address_______|_physical_________|_size___|
  C:00000000--008FFFFF|
```

이 영역은 Virtual Address가 아무데도 Mapping되어 있지 않기 때문에 CPU가 Access하려고 시도하면 Abort가 나요. 으흐흐. Fault page라고도 하지요.

② 그렇다면,10번째 Entry를 볼까요?

```
___address__|________0_0123
ASD:00A24024| 00900D62 b...   →   00000000100100000000110101100010
```

요거는 끝에 2bit가 10으로 끝나네요. 그러면, Section Entry인 거지요. 자 그러면 크기는 1MB일 것이고, 앞에 12bit가 base 주소를 가리키겠죠. 나머지 bit을 0으로 clear시키면 Physical Address Base가 나와요. 그 값은 00000000100100000000000000000000이고요, 0x900000이네요. 9MB부터 1MB를 가리키겠네요.

```
Virtual address_______|_physical__________|_size___|
 C:00900000--009FFFFF|  A:00900000--009FFFFF| 00100000
```

오, 정말 그르네요? 신기하다.

③ 11번째 Entry는 아무것도 없으니 12번째 Entry를 볼까요?

```
ASD:00A2402C| 00A41561 a...  → 00000000101001000001010101100001
```

끝에 2bit가 01이니까 coarse page네요. [31:10] 까지가 base주소니까 00000000101001000001100000000000이 되고 이건 Hex로 0xA41400니까 거길 보죠. 여기에 Level 2 Page Table이 있는 거에요.

```
___address__|________0_0123
ASD:00A41400|>00000000 ....     ⓐ 여기는 비어 있겠네요.
ASD:00A41404| 00B01FFA ....     ⓑ → 00000000101100000001111111111010
ASD:00A41408| 00B02FFA ./..
ASD:00A4140C| 00B03FFA .?..
ASD:00A41410| 00B04FFA .o..
ASD:00A41414| 00B05FFA ._..
ASD:00A41418| 00B06FFA .o..
ASD:00A4141C| 00B07FFA ....
ASD:00A41420| 00B08FFA ....
ASD:00A41424| 00B09FFA ....
ASD:00A41428| 00B0AFFA ....
ASD:00A4142C| 00B0BFFA ....
ASD:00A41430| 00B0CFFA ....
ASD:00A41434| 00B0DFFA ....
ASD:00A41438| 00B0EFFA ....
ASD:00A4143C| 00B0FFFA ....     ~ 요기까지~
ASD:00A41440| 00B10FF9 ....     ⓒ → 00000000101100010000111111111001
ASD:00A41444| 00B10FF9 ....
ASD:00A41448| 00B10FF9 ....
ASD:00A4144C| 00B10FF9 ....
ASD:00A41450| 00B10FF9 ....
ASD:00A41454| 00B10FF9 ....
ASD:00A41458| 00B10FF9 ....
ASD:00A4145C| 00B10FF9 ....
ASD:00A41460| 00B10FF9 ....
ASD:00A41464| 00B10FF9 ....
ASD:00A41468| 00B10FF9 ....
ASD:00A4146C| 00B10FF9 ....
ASD:00A41470| 00B10FF9 ....
ASD:00A41474| 00B10FF9 ....
ASD:00A41478| 00B10FF9 ....
ASD:00A4147C| 00B10FF9 ....
ASD:00A41480| 00B20FF9 ....
ASD:00A41484| 00B20FF9 ....      주으으으욱~
```

ⓐ 는 비어 있으니까 넘어가 구요, ⓑ 는 00000000101100000001111111111010니까요. 끝이 10이네요. 그러면 small page 4K짜리구요, Physical Address는 하위 11번째 bit 까지를 clear하면 되니까, 00000000101100000001000000000000 → 0xB01000 이고요. 0xB01000에서부터 4K 만큼을 차지하겠네요. 그러면 그 뒤로 주르르륵 0xB01000에서부터 0xB0FFFF까지 모두 4KB로 차지하고 있겠네요. 음냐.

```
Virtual address_______|_physical__________|_size___|
 C:00B01000--00B0FFFF|  A:00B01000--00B0FFFF| 00001000|
```

자, 그럼 ⓒ는 어떨까요? coarse page table이고요, 아까에 이어서! 붙어 있는 거에요. 끝이 0x00B10FF9이고요,00000000101100010000111111111001이니까, 끝이 01이네요. 01이라는 얘기는 Large page 16KB짜리구요. 이거는 [15:0]을 0으로 clear하면 주소니까, 00000000101100010000000000000000이구요, Hex로는 0xB10000 이랍니다. 그러면 0xB10000부터 16KB씩 늘어나는 거에요~ 주르르륵 끝날 때까지. 이 결과를 보면,

```
Virtual address_______|_physical__________|_size___|
 C:00B10000--00BFFFFF|  A:00B10000--00BFFFFF| 00010000|
```

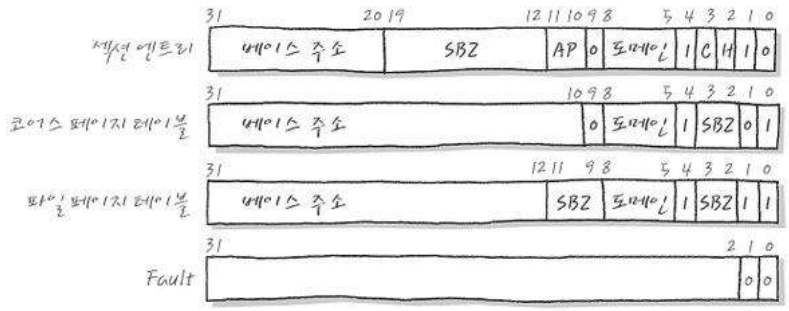
뭐 이런 셈이죠. 우후후~ 실은 하도 table 길이가 길어서 자른 거에요. 이제 0xB00000 의 1MB 영역을 다 봤으니까, 다시 Level 1으로 올라봐 BoA요. ④번인데요, 00A24030| 00C00D6A 이니까, ④ → 00000000110000000000110101101010 이고요, 끝이 10이니까 Section이겠네요! 그럼 또 1MB짜리. 그런데 가만히 보면 ⑤까지 계속 비스므리한 값들이 연속되어 있네요? 결과 볼까요?
0xC00000~0x14FFFFF까지 계속 1MB 단위로 Mapping되어 있는 거죠?

C:00C00000--014FFFFF| A:00C00000--014FFFFF| 00100000|

캬캬. 정말 그러네요. 신기해라. 요기서 또 신기한 걸 하나 더 볼게요. Cache관련한 설정 값들인데요. 위의 Page Table은 실은 몇 가지 복잡한 내용을 생략한 내용들인데, 자세히 살펴 보면 Virtual ↔ Physical Address Mapping이 있을 뿐 더러, Cache설정도 같이 있는 거에요. 그리고, MMU관련한 Permission도 같이 표기가 되어요. 아래의 Page Table보면, Permission이 readwrite가 있고요, acceess의 경우에는 uncached, unbuffered, write_through/ bufferred 뭐 이런 거 있지요. 이것이 바로 cache와 MMU memory protection 기능에 대한 표기에요. MMU page 단위로 이런걸 설정할 수가 있는데요. 찬찬히 뜯어보지요.

```
____address_____|_physical__________|_size___|permission|_access_____________
00000000-008FFFFF|                   |        |          |
00900000-009FFFFF|A:00900000--009FFFFF|00100000|readwrite |uncached/unbuffered    ⓐ
00A00000-00B00FFF|                   |        |          |
00B01000-00B0FFFF|A:00B01000--00B0FFFF|00001000|readwrite |write-through/buffered
00B10000-00BFFFFF|A:00B10000--00BFFFFF|00010000|readwrite |write-through/buffered
00C00000-014FFFFF|A:00C00000--014FFFFF|00100000|readwrite |write-through/bufferedⓑ
01500000-0153FFFF|A:01500000--0153FFFF|00010000|readwrite |write-through/buffered
01540000-01546FFF|A:01540000--01546FFF|00001000|readwrite |write-through/buffered
01547000-01EFFFFF|                   |        |          |
01F00000-01FFFFFF|A:01F00000--01FFFFFF|00100000|readwrite |uncached/buffered
02000000-16CFFFFF|                   |        |          |
16D00000-177FFFFF|A:16D00000--177FFFFF|00100000|readwrite |write-through/buffered
17800000-178BFFFF|A:17800000--178BFFFF|00010000|readwrite |write-through/buffered
178C0000-178CFFFF|A:178C0000--178CFFFF|00001000|readwrite |write-through/buffered
```

뭐, 일단 달라 보이는 ⓐ와 ⓑ를 다시 한번 찬찬히 뜯어 볼까요? ㅋ ⓐ는 원래 page table Entry에 어떻게 씌어 있었는지 확인해 보시죠.



자리 10987654321098765432109876543210
값 00000000100100000000110101100010

자, 이건 Section Entry였구요, [11:2]까지의 bit가 뭔가 의미를 가지고 있었죠. 이걸 다시 분석해 보면, AP(Access Permission)는 [11:10] = 11 이네요. 이건 의미가 접근 권한이에요. 11은 ReadWrite라는 뜻이고요, Domain은 [8:5] = 1011이네요. 이건 의미 잠시만요 Cache는 [3] = 0 이니까 Uncached, Write Buffer는 [2] = 0 이니까 Unbufferable 인 거에요.

C:00900000--009FFFFF|A:00900000--009FFFFF|00100000|readwrite |uncached/unbuffered ⓐ
자, 어때요. 그렇지요?

그럼 두번째 ⓑ를 볼까요?
이것도 Page Table의 내용이 00C00D6A 이었고요,
10987654321098765432109876543210
00000000110000000000110101101010

AP는 [11:10] = 11 이네요. 이건 의미가 접근 권한이에요. 11은 ReadWrite라는 뜻이고요, Domain은 [8:5] = 1011이네요. 이것도 잠시만요. Cache는 [3] = 1 이니까 Cached Write Buffer는 [2] = 0 인데, Cache를 쓰고 있으니까, Write-through Buffered인 거에요. (만약에 Cache가 1인데, B도 1이면 Write-back buffer에요)

C:00C00000--014FFFFF|A:00C00000--014FFFFF|00100000|readwrite |write-through/buffered ⓑ

C와 B의 의미는 뭐 이런 식인 게죠.
C B
0 0 : Cache Off, Write Buffer Off
1 0 : Cache On (Write Through), Write Buffer On
1 1 : Cache On (Write Back) Write Buffer On

아. 복잡하다. 글치요? Domain이 하나 남았는데요, Domain은 Total 16개의 Domain으로 정해져 있고요, DACR은 이런 Domain에 관한 값이 예를 들어 00400001라면, 2bit씩 16개의 Domain의 속성을 각각 가리키는 거에요.

DACR이 0x00400001라면, 2bit씩 자르면 되어요. 결국 00/00/00/00/01/00/00/00/00/00/00/00/00/00/00/01 이 binary 값이니까 11번째와 0번째가01값을 갖고요. 01의 의미는 Client에요. (00 : No Access, 01 : Client, 10 : Reserved, 11 : Manager, Master는 무조건 Access가능이에요)
자... 보면은 D0와 D11이 Client이지요? Client 이외의 것들은 모두 no access니까 Access하면 Fault가 나요.

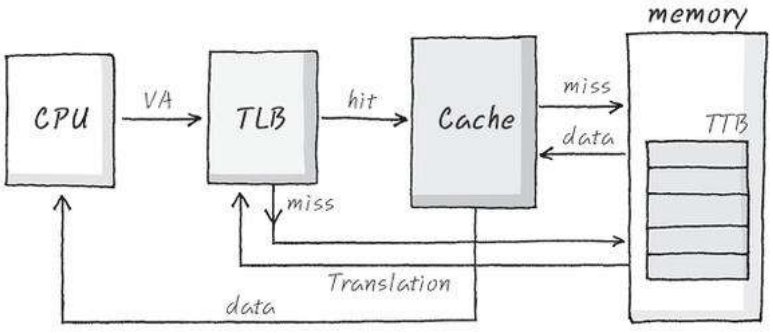
D15 no access D14 no access D13 no access D12 no access
D11 client D10 no access D9 no access D8 no access
D7 no access D6 no access D5 no access D4 no access
D3 no access D2 no access D1 no access D0 client

위에서 본 Domain은 4bit로 이루어져 있지요. 그러니까 Entry를 16개의 Domain중 하나의 Domain에 속하게 할 수 있어요. 그 중에 client에 속하게 되면 AP (Access Permission)에 의한 Access권한이 주어지고요 이 경우에는 둘 다 1011이니까 결국 11번째 Domain에 속한다는 의미이고요. 11번째 Domain은 Client이고요. 이 Domain에 대한 access 권한도
System Design하는 Engineer가 잘 정해줘야 해요. 오오, MMU는 Memory에 관한 한 엄청난 일들을 하는군요! 헥헥. 자, 이제 끝난 줄 알았겠지만, 마지막으로 하나 더 할 것이 남아 있어요. TLB라는 건데요. TLB(Translation Look Aside Buffer)는 그 정체가 Cache에요. Entry에 대한 Cache인 거지요.

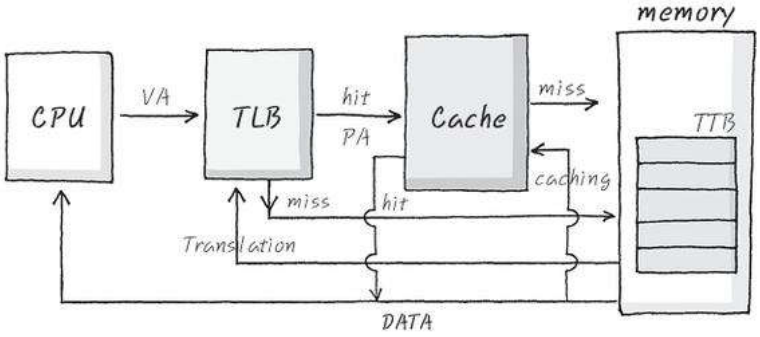
Virtual Address를 MMU에 입력을 하면 MMU는 일단 느린 Main Memory의 Page Table을 Access 하겠지요. 이러면 또또또 곤란한 거에요. Page Table에 갔다 오는 거 자체가 너무 느리니까 Entry자체도 Cache Memory를 마련해 놓고 거기다가 캐싱을 하는 거죠. 요 Cache Memory가 TLB인 거에요. 그래서 Page Table이 Entry안에 있으면 굳이 Main Memory까지 갔다 오지 않고요, 기냥~ TLB안에 있는 Entry를 가져다 쓰면 훨씬 빠르겠지요.

TLB안에 Entry가 없는 경우에만 Main Memory에 가서 Page Table을 가져오는 일을 하게 되는 거죠
① CPU가 Virtual Address를 발생하면,
② TLB에 Virtual Address Entry가 올라와 있는지 확인하고요,
③ 있으면 곧바로 Memory에 Access해서 Data 가져오고요,
④ 없으면 Memory의 TTB부터 위치해 있는 page table에서 Physical Address구해와서
⑤ 다시 Memory에 Access해서 Data를 가져오는 거죠.

만약에 TLB가 없다면 느려터진 Memory를 2번은 꼭 Access해야 Data를 가져올 수 있는 거죠.



여기에다가, Cache까지 한꺼번에 그려넣으면 좀더 상상하기 편해질 라나요?



뭐.
엄청 긴 얘기가 되어 살짝 미안해 져 버린 저에요.

이렇게 분석을 했지만, 실은 로꾸꺼 Page Table을 만들 수 있어야 해요. 남이 만든 System은 이렇게 분석을 하겠지만 서도, 내가 만드는 System은 내가 꾸며줘야겠지요. 게다가, MMU를 쓰게 되면 이제부터 Memory Map, 특히나 Scatterloading 따위도 무조건 Virtual Address 기준으로 만들어야 한답니다. 쩝.

덧글 | 덧글 쓰기

highseek 2010-01-24 22:57
그림중에 오타가 있네요.

> 히언 2010-01-24 23:13
> 으악! 어디죠?

> highseek 2010-01-24 23:32
> 세번째 그림에 가르키고 있음(x) -> 가리키고 있음(o)
>
> 그리고 fine page table이 죄다 파일 페이지 테이블이라고..

> 히언 2010-01-25 01:01
> 으흐.. 그르네요!
>
> 파일 -> 파인 이 맞습니다.
> 그림은 수정되는대로 올려야겠어요!
> 완전 실수 했는데요 크흑.

지니 2010-01-25 13:21
안녕 하세요 강좌를 아주 잘읽고 있는 독자 입니다.

그림 6번째 것에서

10이 Section Entry 이고

01이 coarse page 인것 같은데 맞는지 ——a

글과 그림이 달라서 수정이 필요한 부분 인것 같습니다만...

> 히언 2010-01-25 15:55
> 아~ 그르네요~ 지송합니다.. 이 부분에 대해서 그림에 오자가 좀 있네요!
> 이해해 주시면 정말 감사하겠습니다~ 수정 할께요 크흑.

이세종 2010-01-26 13:42
안녕하세요..
저는 그림이 너무 예뻐 그림을 보고 나서 이 글을 읽게 되었는데요.
그림은 직접 그리시는 건지요?
너무 잘 그리시네요. 이전 글도 봤는데, 삽화그림도 잘 그리시네요.
위 그림(메모리구조)은 포토샵으로 그린건지.. 아님 다른 툴이 있는건가요?
좋은 글 써주셔서 감사합니다.

> 히언 2010-01-26 21:32
> 으흐흐 안녕하세요.
> 몇번 물어보시는 분들이 있었는데,
> 툴을 사용하진 않았습니다~
> 모두 손그림이에요.
>
> 냐호호~

ruring 2010-01-28 10:07
에이알두안대 손그림이라니!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

> 히언 2010-01-28 22:17
> 으흐흐 발그림이 아니니 다행이죠.. 으후후후훗!

김경진 2010-01-29 13:30

5번째 그림 large page의 [1:0]bit가 2'b01인데 오타 같네요 ^^
그리고 내용중 "9MB부터 1MB를 가리키겠네요."는 10MB죠?

히언 2010-01-29 15:37
맞습니다~ 01이 맞아요 T.T 크흑.
그리고 9MB부터 1MB니까 10MB를 가르키는 거죠~ T.T
완전 엉망이에요 죄송해요. 얼릉 수정하고, 정오표 만들겠습니다~ T.T

soto 2010-01-30 06:34
6번째 그림:
coarse pagetable에서 large page base와 small page base가 있는데,
large, small에서 각각 16kb, 1kb subpage가 뜻하는 것이 16kb, 1ke씩 나누어 진다는 뜻인지 아님 그림 아래에 있는 64kb, 1kb로 나누어 진다는 것인지 조금 헤갈린다는 느낌이 들어요 ^^;

히언 2010-01-31 21:00
으흐흐 그르네요..

김경진 2010-01-30 21:30
그래도 MMU를 1번에 설명을 하시는 센스...대단하세요 ^^

히언 2010-01-31 21:01
이쁘게 봐주시는 김경진님,
더 열심히 잘 할께요 ^^

풀리비 2010-02-09 19:05
댓글은 처음이네요^^;;

ARMv6부터 page table descriptor 형식이 바뀐 것 같습니다.
ARM 매뉴얼보다가 coarse와 fine 페이지 테이블은 죽어도 안나오고 large page와 small page라고만 나와 왜 그런가 했네요..
ARMv7 매뉴얼 보면 ARMv6까지는 legacy support로 coarse page table을 지원하는 것 같더군요.

히언 2010-02-09 21:44
으흐흐~ 맞습니다.

일단은 이 concept을 이해한다면,
Page table을 사용하는 모든 system에 활용할 수 있습니다.
예를 들면, File system중에 EFS2라는 것이 있는데, 이녀석도 대충 이런 형식의 page table을 사용하고 있는 거죠. page table은 모든 page mapping architectrue를 이해하는 basic이 된답니다. 냐호~
풀리비님 감사~

덧글 쓰기