segmentation → external fragmentation

paging → memory 절약 편리함. (page table 접근해야함)

⇒ TLB 속도느려지는 문제 해결 → cache에 저장

10. Translation-Lookaside Buffer

▼ Paging ⇒ 느리다!! 라는 단점...

address space → 작고 고정된 사이즈로 chopped into(잘게 나눠짐)

→ 많은 양의 information mapping 필요로 함

- stored in physical memory → 레지스터로 볼까. (page table은 memory에 존재함)
- 각 virtual address에 대한 추가적인 memory lookup 필요로 함 (더 많은 과정들도 필요로 함 (VPN → PFN 과정))

→ 어떻게 address translation 속도를 높일 수 있을까?

- 어떤 hw support 필요로 하는가?
- OS의 간섭 필요로 하는가?

▼ Translation-Lookside Buffer(TLB)

- MMU : address space를 지원하기 위해 hw가 지원하는 것들 ex) address translation pseudo code 등...

  → 메모리와 관련해서 cpu가 지원해야 하는 일을 뜻함.

  ⇒ OS : MMU와 관련된 것들을 잘 설정해줘야 함

  → TLB! MMU 중 일부분

▼ TLB

CPU에 구현되어 있는 MMU(memory management unit)의 한 부분

→ address-translation cache

- 자주 접근하는 아이들을 cache memory에 저장해 놓음 → 메모리 데이터 접근 (address translation 결과를)
  - TLB에 hold 되었던 desired translation이 있다면
    - page table에 대한 접근 없이 빠르게 translation 수행 가능
- Basic algorithm

⇒ accessmemory 한번만!!

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True) // TLB hit
  if (CanAccess(TlbEntry.ProtectBits) == True)
    offset = VirtualAddress & OFFSET_MASK
    PhysAddr = (TlbEntry.PFN << SHIFT) | offset
    Register = AccessMemory(PhysAddr) //접근하고자 하는 메모리 영역
  else
    RaiseException(PROTECTION_FAULT)
else // TLB miss
  PTEAddr = PTBR + (VPN * sizeof(PTE))
  PTE = AccessMemory(PTEAddr)
  if (PTE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
  else if (CanAccess(PTE.ProtectBits) == False)
    RaiseException(PROTECTION_FAULT)
  else
    TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
    RetryInstruction()
```

TLB hit

TLB miss

Input

VPN 비트수

TLB 엔트리가 있는지 먼저 확인

⇒ 다양한 bit 검증적인지

address translation

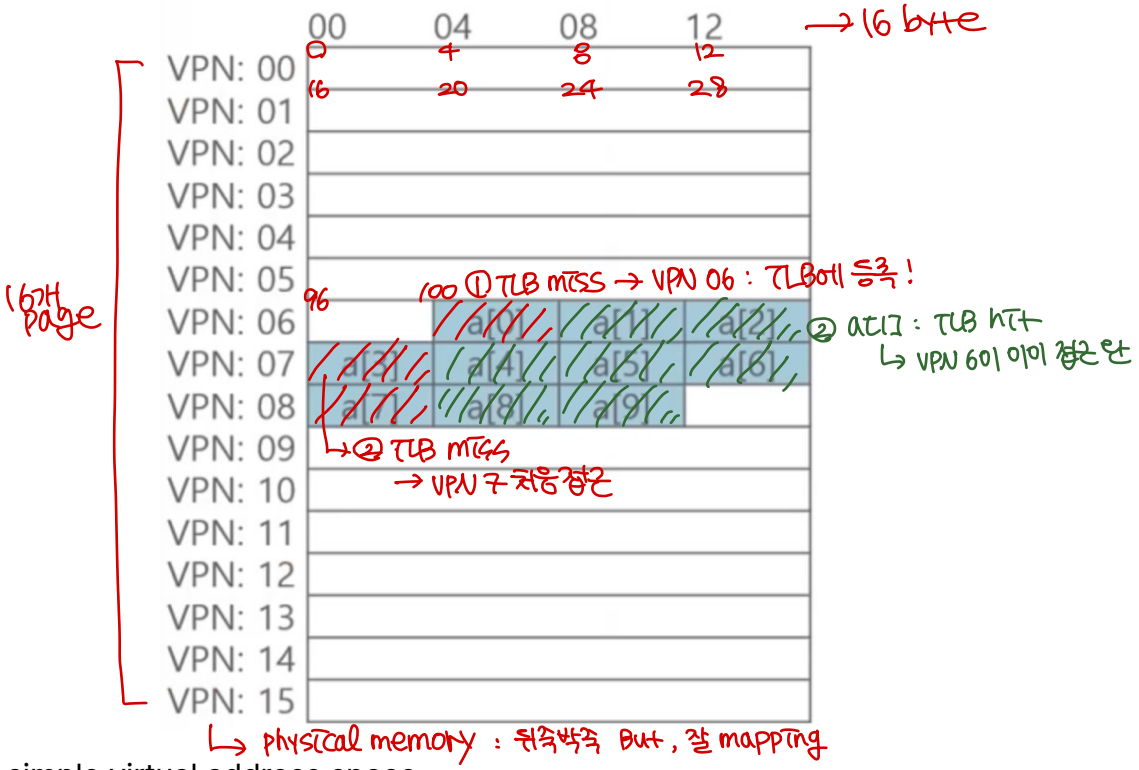
↳ PTE 얻을때도 memory 접근은 한번에도 된다!!

새로운 VPN 엔트리를 TLB에 넣음

↳ TLB hit 다시 시도!!

가정 : a simple linear page table ★, h/w managed TLB

▼ **example**



- a simple virtual address space

    - 8-bit addressing ⇒ offset : 4bit , VPN : 4bit
    - 16-byte pages ⇒ 2^4
- an array of 10 → 4byte integers : 배열이
    - starting at virtual address 100 ⇒ 2^4 * 6 + 4 ⇒ VPN 6번지에 4번부터
- simple loop

```
int sum = 0;
for (i = 0; i < 10; i++){
  sum += a[i];
}
//sum -> register로 해결 -> address translation과 무관한 아이라고 보자
```

    - hit rate : 70%
    - spatial and temporal locality (시간, 공간적으로 비슷한 위치)
        - 프로그램 실행 시 접근하는 메모리 영역의 경향

          → 이미 접근이 이루어진 것의 근처에만 접근하거나 접근이 한 번 이루어진 주소는 자주 접함

▼ Translation-Lookside Buffer(TLB) - 2

누가 TLB Miss를 처리?

▼ who handles the TLB Miss

(CPU)

1. hw managed TLB
    - CISC (ex. x86)
        - hw는 PTBR을 통해 page table이 memory 어디에 위치해 있는지 + 정확한 format을 완전히 알아야 함
    - In x86 → CR3, multi-level page table

    hw → 자체적으로 page table 접근할 수 있는 능력이 필요(TLB miss 대비)
        - OS: PTBR 값을 잘 넣어줘야 함
        - CR3 register → page table base 주소를 잘 가지고 있어야 함 (multi-level page table)

(OS)

2. sw managed TLB

- RISC (example, MIPS) : CPU가 해결하지 않고 linux에게 책임을 넘김

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True) // TLB hit
  if (CanAccess(TlbEntry.ProtectBits) == True)
    offset = VirtualAddress & OFFSET_MASK
    PhysAddr = (TlbEntry.PFN << SHIFT) | offset
    Register = AccessMemory(PhysAddr)
  else
    RaiseException(PROTECTION_FAULT)
else // TLB miss
  RaiseException(TLB_MISS) //exception 발생 -> OS가 알아서 처리
```

- trap handler : TLB update하기 위해 privileged instructions 사용함
  - trap을 일으키는 instruction에 따라 실행을 재개해야 함
  - TLB misses의 infinite chain 일으키지 않기 위해 조심해야 함
- OS : page table을 implement하기 위해 어떤 data structure든 쓸 수 있음

▼ TLB contents

- Fully associative
  - 어떤 주소에 대한 translation이든 모두 TLB 빈 공간에 들어갈 수 있음
  - hw : desired translation 찾기 위해 전체 TLB를 병렬로 search
    - 찾으면 hit, 못 찾으면 miss
- entry가 가지고 있는 정보
  - VPN
  - PFN
  - other bits
    - Vaild bit : entry가 vaild transition을 가지고 있는지 여부를 나타냄
    - Protection bits : page가 page table에 접근 할 수 있는지 나타냄
    - Address space identifier, dirty bit, etc.

▼ Context Switches

- TLB : 현재 실행 중인 procee에만 vaild한 virtual-to-physical transaltion 포함

Hardware can't distinguish
⇒ 100번? 170번? 무엇을 처야하지?

	VPN	PFN	valid	prot
P1	10	100	1	rwx
	-	-	0	-
P2	10	170	1	rwx
	-	-	0	-

- context switch할 때 TLB contents 어떻게 관리?
  - flush the TLB on context switches ⇒ valid 값을 모두 0으로 초기화!
    - OS : process 사이에서 switch 자주 한다면 cost 매우 비쌈 → TLB를 관리하기 위한 overhead만 늘어남
  - Addres space identifier (ASID)

해결!

overhead 해결=>address ID(ASID)

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
-	-	0	-	-
10	170	1	rwx	2
-	-	0	-	-

중복되는 VPN 구분

중복되는 VPNID 가지고 있어도 괜찮음

▼ Sharing of pages page공유되는 case!

- 사용 중인 physical page의 수를 줄이면 memory overhead 줄일 수 있음
  - process의 address space 너무 큼 → 쓸 때만 copy하기
  - 방법 예시 : binaries, shared libraries, fork() ⇒ 공유가능
    - 두 process 중 바꾸는 operation할 때까지 계속 공유 ⇒ address space 커짐. fork() overhead ↑
    - ⇒ 쓸 때만 copy하는 system call (cow) 사용.
- shared memory IPC

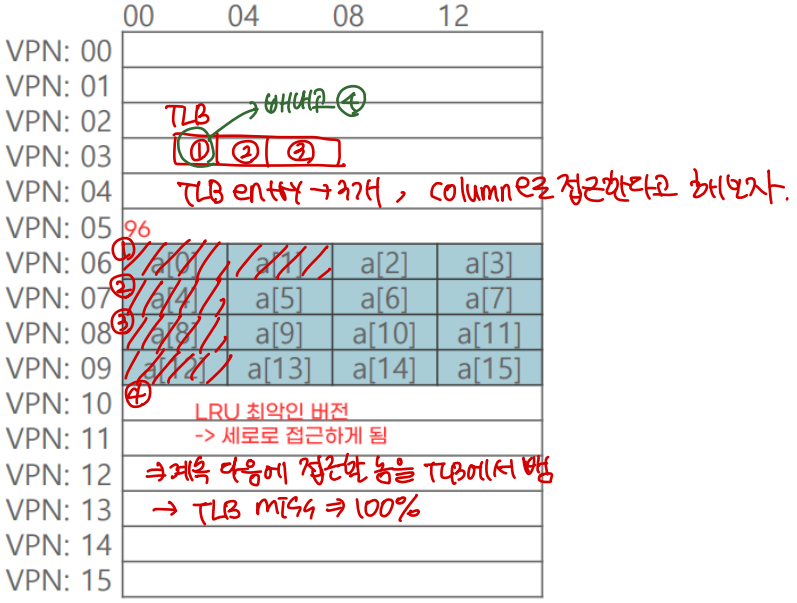
VPN	PFN	valid	prot	ASID
10	101	1	r-x	1
-	-	0	-	-
50	101	1	r-x	2
-	-	0	-	-

vpn → 다름 , pfn → 같음
⇒ shared memory 쓰는 경우

▼ Replacement Policy → TLB가 모두 채워져있는데 새로운 page에 대한 address translation이 발생했을 때 어떻게?

- TLB replacement policy 어떻게 design?
  - 우리가 새로운 TLB entry 추가하려고 할 때 어떤 TLB entry가 교체됨?

    ⇒ 이것에 대한 법칙!!
  - miss rate 줄이기를 목표로 함 ★
- basic polices (대표적인 Algorithm ⇒ 모든 응용 SW에게 정답인 것은 X)
  - ①LRU(Least-Recently-Used)
    - 최근에 안 쓰인 놈은 앞으로도 안 쓸 거라고 가정
      - eviction의 우선 순위가 됨
    - 크기가 n인 TLB로 n+1 page를 loop할 때 unreasonable
      ↳ 되게 좋지 않은 경우!! ※ LRU → 예전것부터 시작한다고 가정

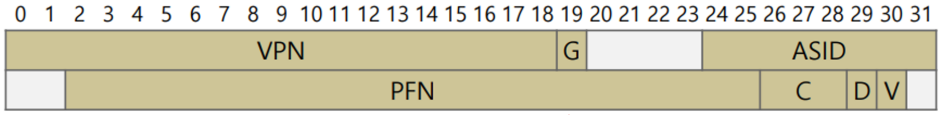


    - 하나 더 더해서 가장 접근한 지 오래 된 놈 빼버림

  - ②Random → 평타는 침!
    - 랜덤으로 eviction

▼ 모든 법칙 적용한 example

- 실제 예시 → MIPS TLB entry
  - 32-bit address space with 4KB page
    - 12-bit offset
    - 20-bit VPN



  - PFN : 24bits → 64GB main memory
  - Global bit(G) : process 사이에 globally shared인 page 표시 위해 사용
  - ASID bits
    - address space 사이를 구별하기 위해 사용
    - PID 보다 더 적은 bit 사용
      - ASID 개수보다 더 많은 process 생성할 수 없음!
      - ASID가 지원되지 않는 것처럼 행동
      - process에게 ASID는 동적으로 할당
  - Coherence bits(C) : page가 hw에 의해 어떻게 cached 되는 지 표시
  - Dirty bit(D)
  - Vaild bit(V)