Lecture04

17/25

버츄얼메모리

-무제한 메모리를 쓰는 환상(가상의 메모리)

-프로텍션(페이지테이블을 통해 실제 주소로 연결)

같은 가상주소라도 실제 주소로 연결되는건 다르다

페이지테이블

-피지컬메모리와 로지컬메모리를 맵핑

-모든 데이터 접근마다 페이지테이블이 필요하다(많은 시간소요)

TLB

페이지테이블의 캐시.

접근마다 페이지테이블을 가는게 아닌 한번만.(성능 향상)

MMU

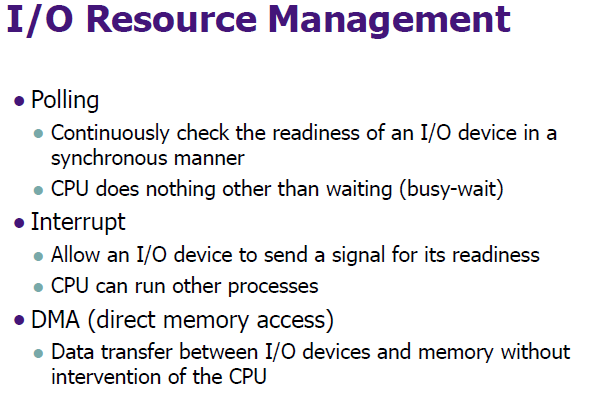
-가상메모리를 피지컬메모리로 변환시켜줌

-CPU가 MMU가 있다면 가상메모리를 쓰는 CPU

19/25

Memory mapped I/O

I/O mapped I/O



21/25

Polling = waiting (휴대폰 배터리, 안테나 강도)

-cpu가 device상태를 계속 체크,일 중단x

Interrupt = 중간에 끊고 들어온다 (전화가 왔을 때,화면터치)

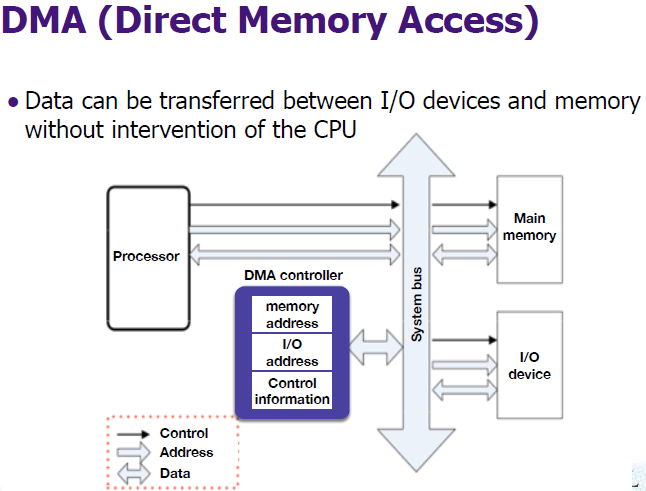
모든 Interrupt는 다른 Interrupt Service Routine이 있다

Interrupt Vector

인터럽트 발생 시 점프하는 주소테이블

DMA

CPU가 신경쓸 필요없이 device와 메모리 사이의 데이타이동



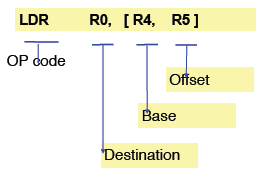
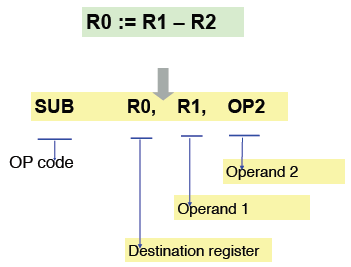
Lecture05

ARM

CPU의 디자인을 설계 (제작X) (팔컴,애플,삼성..등이 ARM 디자인의 CPU 사용)

9/25

[ ] 표시는 주소라는 뜻



11/25

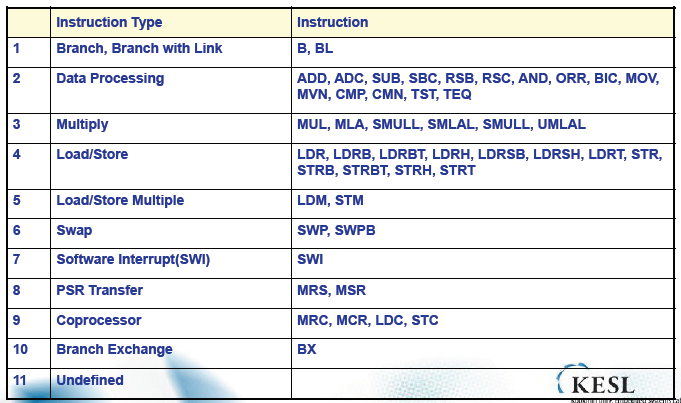
pipelines : 여러 명령어를 단계를 나눠 동시에 처리

(멀티 프로세싱 같은 효과) -> 퍼포먼스 향상

14/25

branch : 폰노이만 형식이여서 branch가 없으면 명령어처리를 순서대로만...

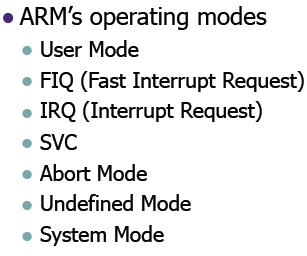
CMP : SUB를 통해서 수행



16/25

ARM의 Operating Mode

사용자가 일반이냐 슈퍼냐에 따라 하드웨어도 변경



인터럽트 발생 시

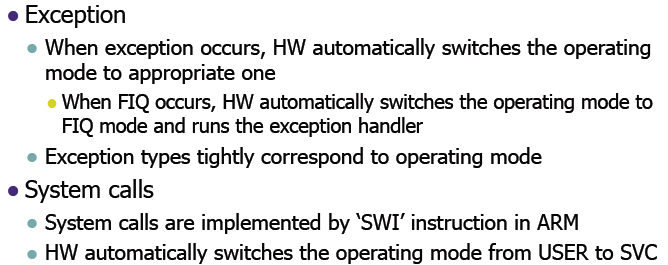
CPU가 FIQ (급한일 일때 (배터리x)) or IRQ로 변경

Abort Mode -> Page Fault(적절한 맵핑 없을 시, 자주겪는다)일 시 handling 한다

명령어 식별이 안될때 Undefined Mode (명령어 op코드 오류 등)

17/25

인터럽트 발생 -> CPU 모드가 바뀜 -> 인터럽트 핸들러 호출



CPU모드가 없을 시

os코드에 if then else 등 핸들러 부분으로 주소 이동 (페이지폴트 시, 인터럽트 시)

시스템콜을 사용 시 CPU가 SWI모드로 자동변경 -> 시스템콜 핸들러 호출

10/25

레지스터가 아닌 상수를 쓸땐 # 을 쓴다

11/25

파이프라인은 병렬처리 처럼 보이게 한다

Dependency . 명령어의 처리결과가 아직 안나왔는데 그 결과값이 입력값인 경우

Hazard가 발생한다.

파이프라인 3단계의 이점 : low power.

14/25

branch : 컨트롤 플로우를 바꾼다. 많이 안쓰느것을 권장. (break pipeline)

multiply : 파이프라인의 길이가 다르다. (계산 길이가 길어서 따로 분류)

시간도 오래걸리고 파워도 많이 필요

-> shift 로 많이 대체하여 사용

load/store multiple : 주소 복사에 자주쓰인다 (array)

SWI : 시스템콜 할때 사용

시스템콜 대표 5 : open close read write IOCTL

Exception : 외부에서 자극을 줘서 발생한 상황에 대한 반응 (ex 인터럽트)

(스마트폰 화면 터치 )

기존에 실행중이던 프로세스는 exception이 발생한 것을 모른다

-> transparent. hidden.

가장 대표적인 exception 은 page fault.

cpu가 exception이 걸리면

인터럽트번호를 본 후

자동으로 cpu모드를 바꾼다.

그 후 인터럽트 핸들러 호출

SVC : 관리자모드. System call 을 사용할때의 모드

(인터럽트 중에 시스템콜 사용 불가능 IRQ ->SVC 안된다)

Abort Mode : 페이지 테이블의 번호를 기록해둔다

모든 Exception 은 외부에서 온다.

SVC 만 내부에서 온다. (시스템콜)

fd = open();

-> swi # 넘버

a = read( , );

-> swi # 넘버

넘버를 통해서 명령어 구분

Undefined Mode : 명령어의 op코드를 봤는데도 무슨 명령어인지 모를때

페이지폴트에는 두가지의 있다

Instruction -> Prefetch

Prefetch 랑 Data Abort 더 알아둘것

-Instruction fetch. 코드 페이지를 갖고 온다. (디맨드 페이징)

-data access 할때

Data Abort 를 발생시키는 것 -> load ,store

자바의 바이트코드가 스택머신이다

스택포인터는 가장 위를 가르키고 있다

스택은 각각의 어플리케이션 마다 있다

-> 스택포인터는 어플리케이션이 바뀔때 마다 바껴야한다 (스택포인터 위치 저장)

프로그램 카운터도 저장

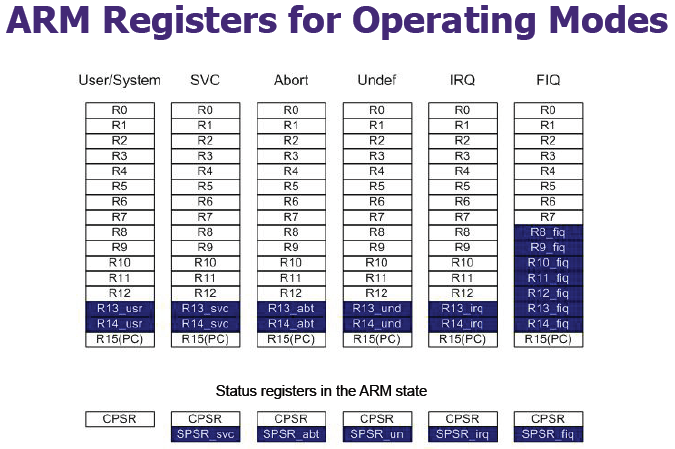
-> 각각의 어플리케이션마다 보따리가 있다(데이터를 저장해둔)

CPU모드가 계속 바뀐다

User Mode 일때 CPSR.

User Mode -> IRQ 가 되면 User Mode는 SPSR , IRQ 는 CPSR

20/25



FIQ 제외하고는 Register를 13개씩 공유

모드가 바뀔때 Register의 내용을 메모리에 save했다가

다시 원래 모드로 돌아왔을 때 restore 한다.

FIQ 가 빠른 이유: 메모리를 안 거칠 수 있다 -> FIQ에 할당된 Register가 많다.

(배터리를 뺐을 때 아주 잠깐동안 중요한 내용을 Flash memory에 저장한다)

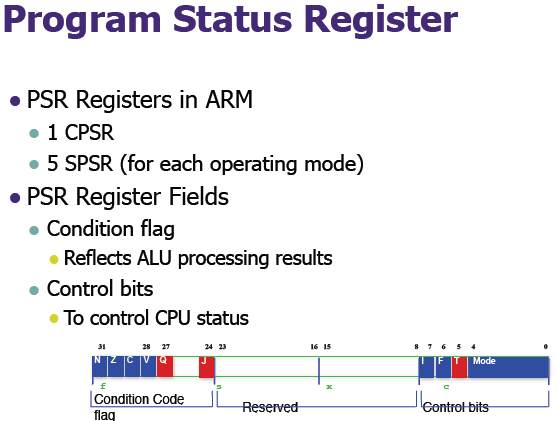
R13 은 보통 Stack Pointer. R14은 return address

(버퍼 오버플로우 : return address를 조작한 해킹행위)

ARM 의 특징 : return address를 스택의 탑에 두지 않는다

-> 하드웨어적으로 버퍼 오버플로우를 막는다

25/25



I F T MODE

0 0 IRQ,SVC,...

0 0 일경우 인터럽트 발생하지 않는다

N Z C V

negative zero carry Overflow

0일경우 1표시

음수일 경우 1표시

ALU 계산을 할 경우 바뀌는 부분이다

(ex CMP R0 R1 -> 0 이면 같다 -> Z 부분에 1 이 된다)

N Z C V (컨디션 플래그) 를 통해 명령어 문구길이를 줄일 수 있다

(IF 랑 BRANCH 를 없앨 수 있다)

-> break pipeline 을 막는다

-> 퍼포먼스 향상

PSR 에 플래그가 있다

Conditional Execution

ARM 명령어에는 N Z C V 가 모두 포함돼있다

branch 명령어는 메인메모리에 access 한다 ( 전력소모 높높 )

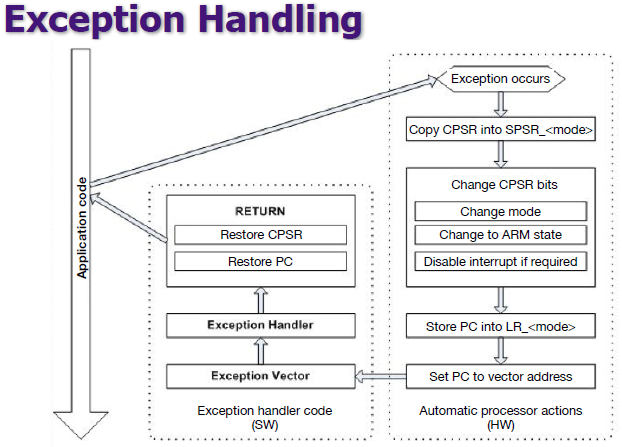
Interrupt ( 타임 인터럽트 , 페이지 폴트 , ... )

Gi Vi : 리눅스의 대표적인 타임 인터럽트

PSR 을 보면 알 수 있는 것

: 어떤 모드인지(User,FIQ), 현재 인터럽트가 on/off 인지

10/16

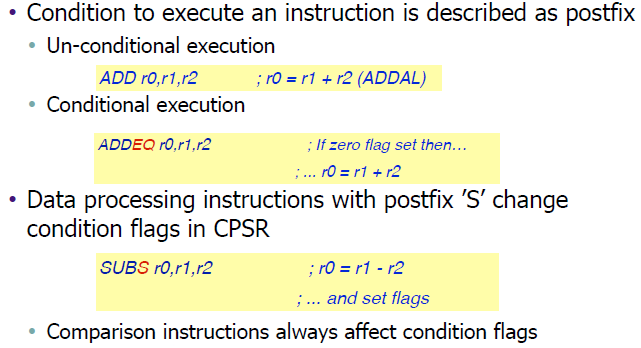


Store PC into LR\_<mode> : 리턴 어드레스를 저장 (LR : 링크 레지스터)

LR 이 없다면 리턴어드레스를 스택의 탑에 저장해야함

Restore PC : 리턴 어드레스로 부터

Restore CPSR : SPSR 로 부터



SUBS 같은 명령어가 N Z C V 에 영향을 준다

ADDEQ 와 같이 N Z C V 를 비교

lecture07

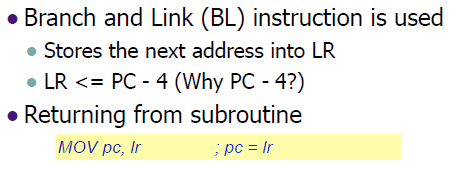
02/16

B : Branch : 그냥 주소로 점프

BL : Branch Link : 주소로 점프 후 다시 복귀 (복귀할 주소를 LR에 저장)

04/16

pipeline 때문에 return address 는 PC-4



07/16

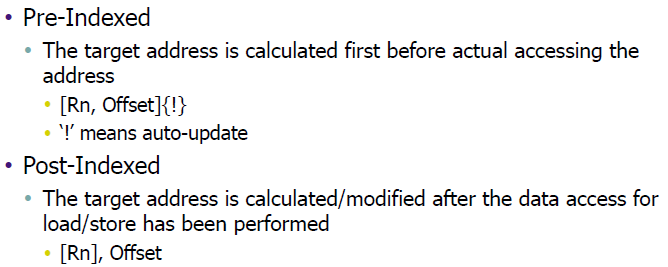
EORS r1,r3,r0 r3과r0를 Exclusive OR 해서 값이 0이면 NZCV 의 Z가 SET

(결과가 플래그에 반영된다)

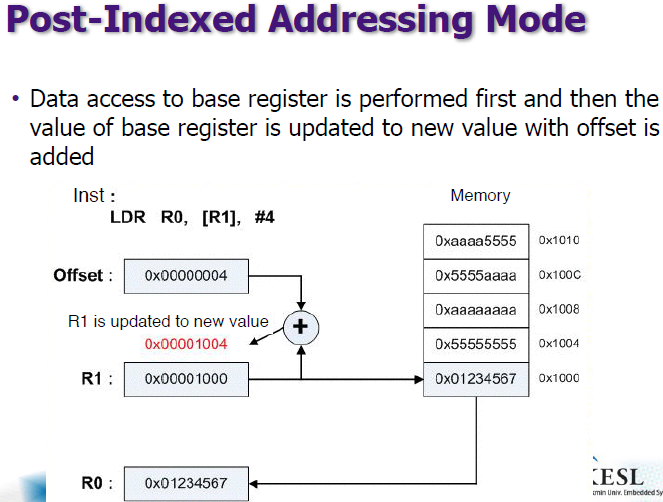
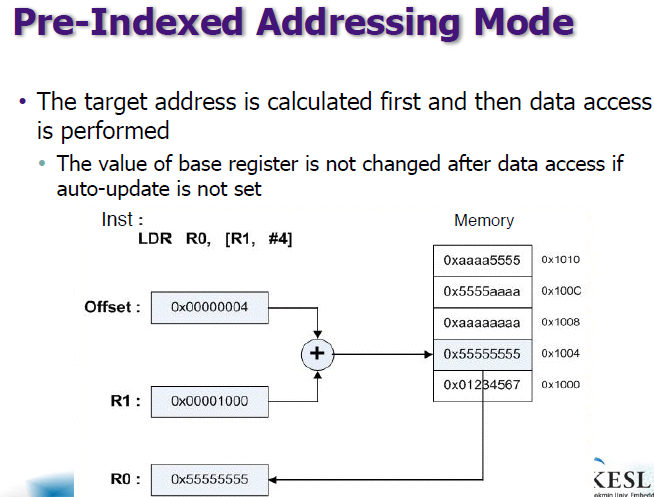
CMP r0,r1 값이 0이면 플래그에 결과 반영 (뺄셈) r0 값이 더 크면 NZ,NN

Word 단위가 가장 빠르다

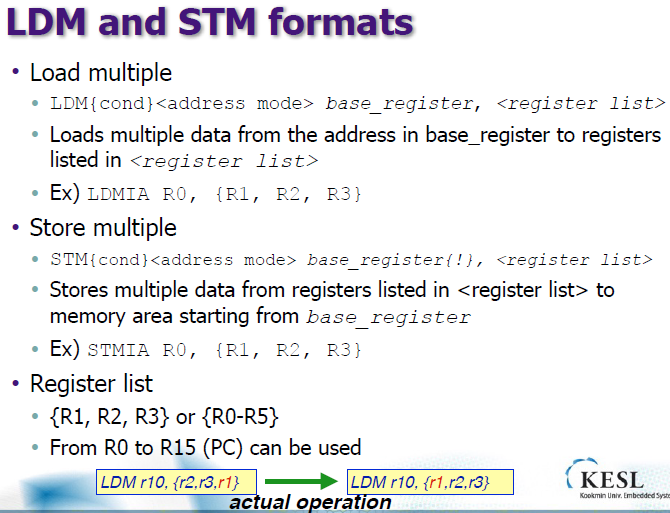
Byte 단위는 word단위로 한번 읽었다가 다시 필요한 만큼 byte로 ...

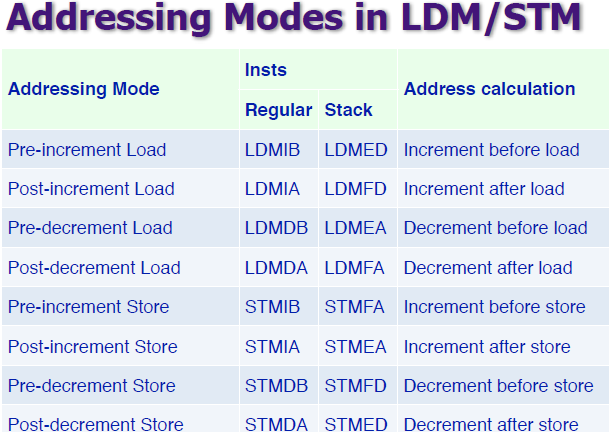


! 마크가 있을 시 레지스터 값 갱신되어 저장 (배열 같은 loop에 자주 쓰임)



12/21





I increment

D decrement

A after

B before

Stack의 종류

-Asc -Desc

Full Empty Full Empty

Full 스택은 스택포인터가 먼저 변한 후 push

Empty 스택은 push를 한 후 스택포인터가 변한다

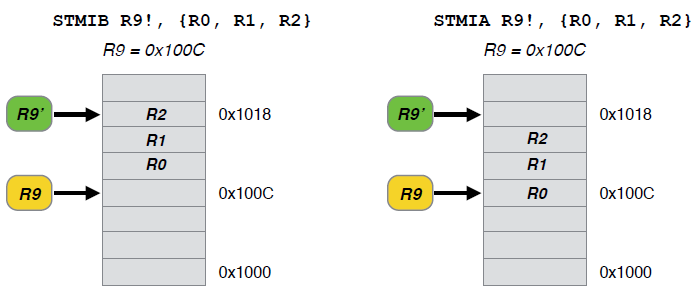
Asc Full 의 경우 push 할 때 IB (먼저 증가 한 후 저장)

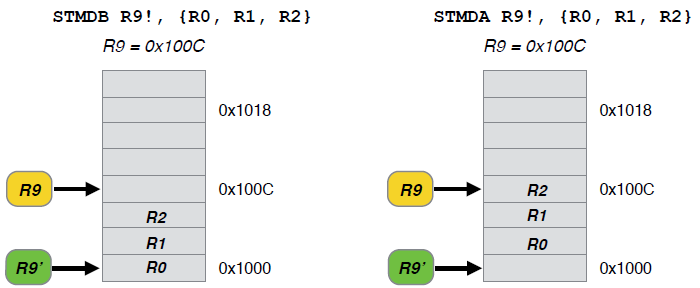
Empty의 경우 push 할 때 IA (저장 후 증가)

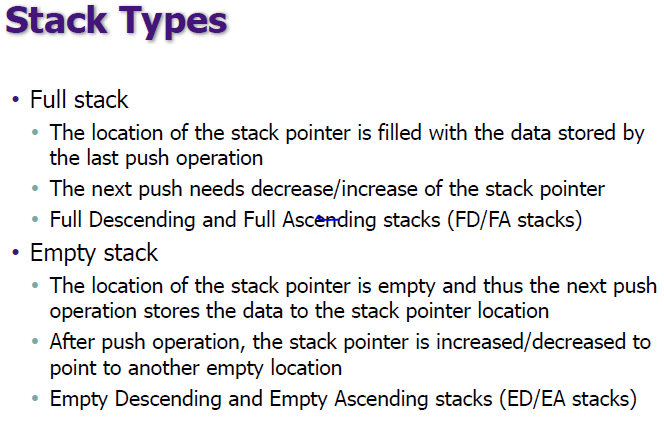
Dsec Full 의 경우 pop 할 때 IA

Empty의 경우 pop 할 때 IB

STMIB = STM Full Asc



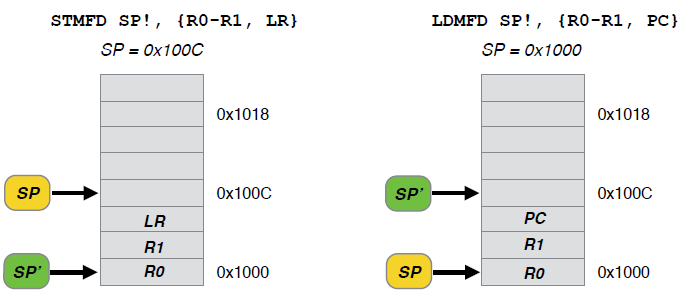




stack frame : function call 을 위한 부분

레드 블랙 트리 : 대표적인 이진트리로 커널의 스케쥴러에 사용된다

R0-R1 : R0 부터 R1 까지

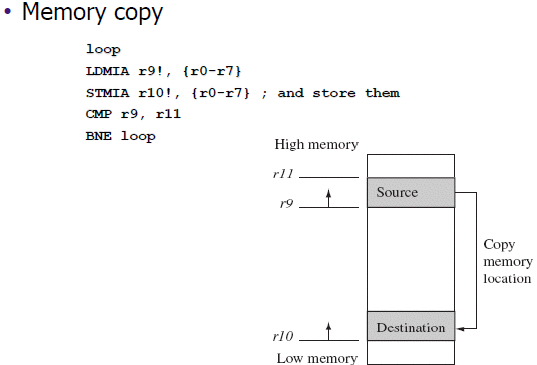


LR : 링크 레지스터

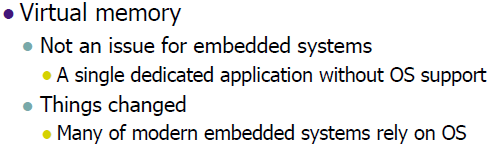
R0 - R1 , LR : 오른쪽에 있는게 스택의 위쪽에 위치한다

r9 : start pointer

r11 : last pointer



r9과 r11이 같아질때까지 반복



버츄얼메모리

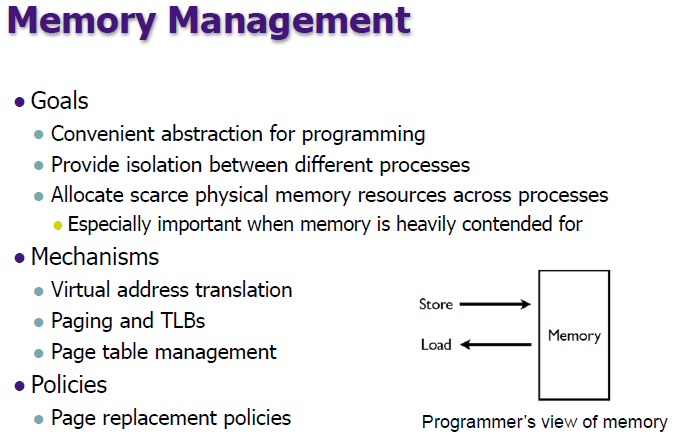
- 무제한 메모리 환상

- 프로텍션

페이지 테이블은 메인메모리에 있다

-> 페이지 테이블 access를 줄이면 성능이 좋아진다

-> TLB (Translation lookaside buffer) 사용 : 페이지테이블의 캐시



페이지 하나의 메모리 : 4K

-> 4M 일 시 오래걸린다. 페이지 메모리 낭비 크다

-> 1K 일 시 페이지를 너무 자주 가져온다.

시간절약, 낭비절약

1960년대의 페이지 : 항상 작동중

현대 : 유저가 작동했을 때 작동 ( 이벤트 발생 시 )

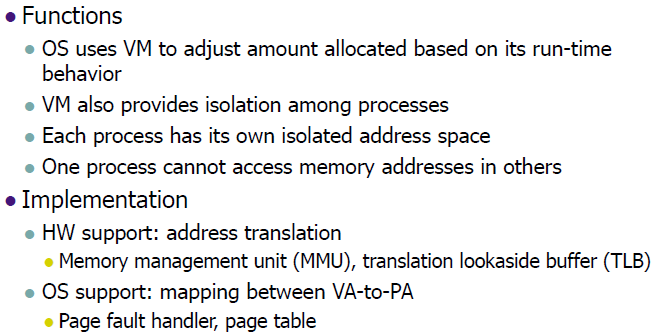
메모리의 관점 : 메모리가 꽉 차 있을 시, 새로운 페이지를 받아들이려면

기존에 있던것과 새로운것을 바꿔야 한다 ( 대체 )

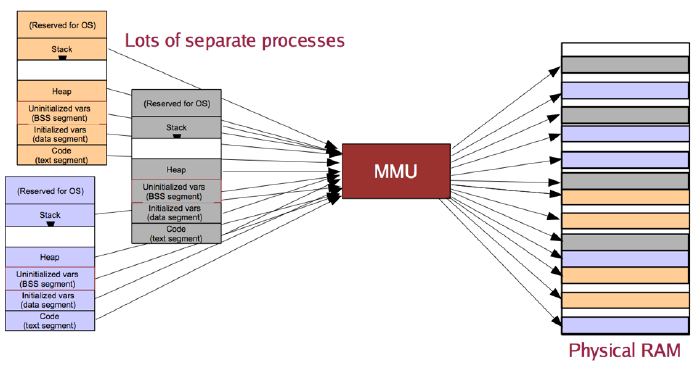
LRU 알고리즘 : Least Recently Used ex) 캐시

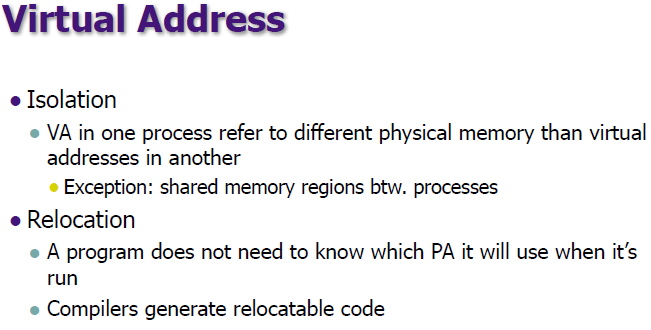
캐시는 우리가 사용하는 것보다 용량이 작다 -> 대체를 많이 함

( Replace for limited space )

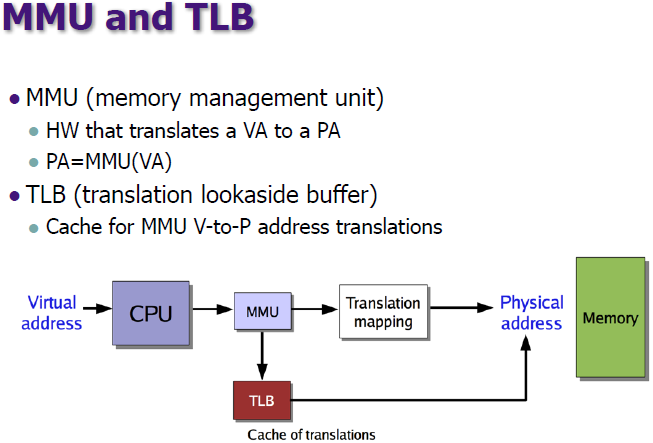


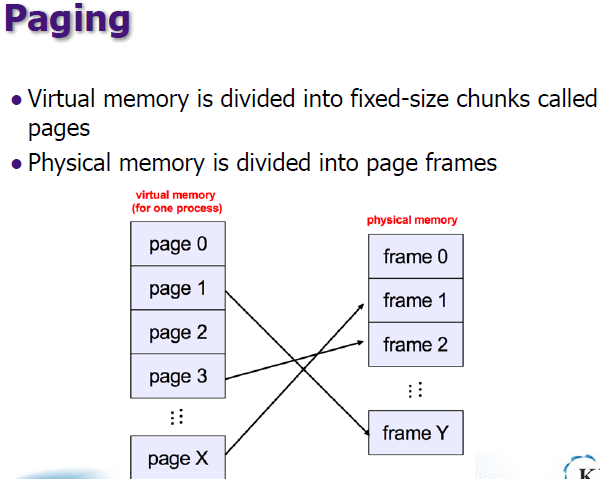
하드웨어적 : 자동으로 페이지테이블을 찾고, 없을 시 페이지 폴트

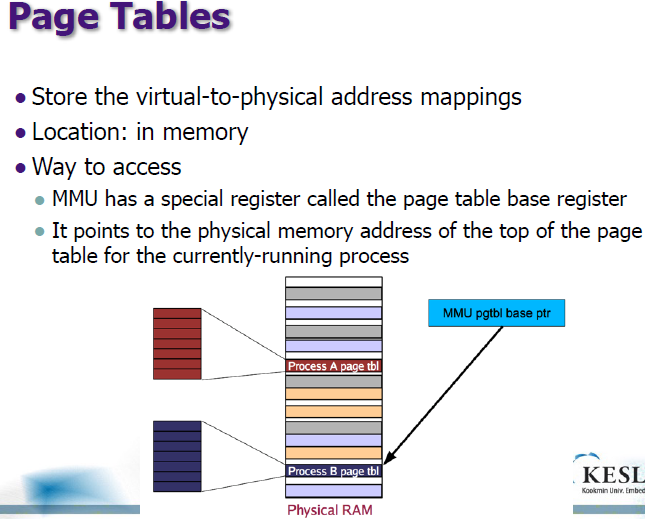


12/18

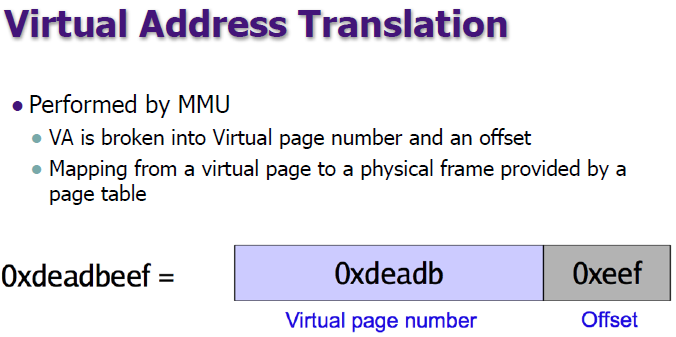
page 는 고정된 사이즈. 항상 같은 사이즈

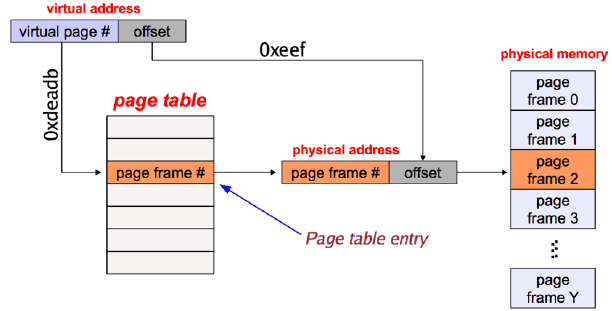


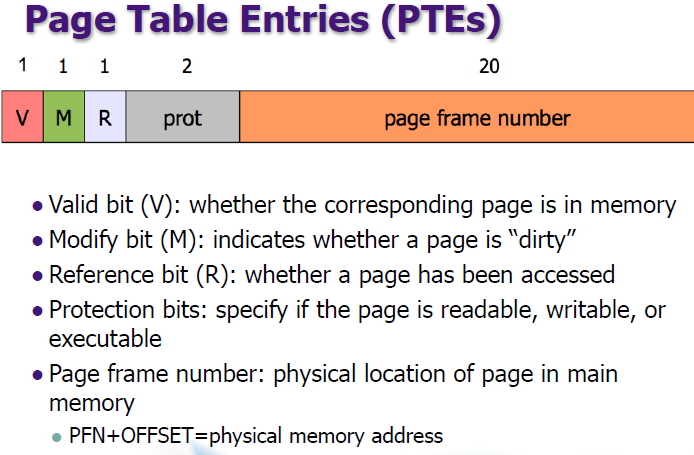




15/18







virtual address : 페이지 테이블을 찾기 위한 정보.index,information to pagetable

physical address : 실제 주소,위치

VA : idx | offset

여러개의 페이지 테이블을 중첩시킨다

VA PA

VA PA

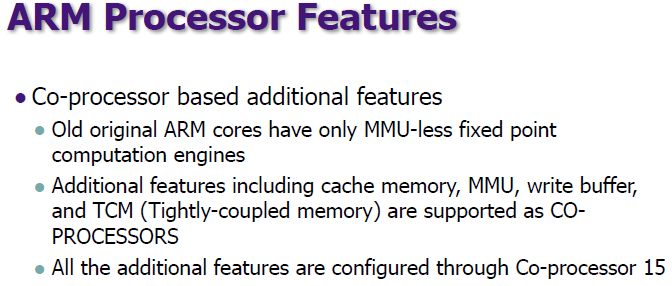
VA 새로운 페이지 테이블의 주소 ( 포인터 역할 )

-----> VA : idx1 | idx2 | offset

idx2 을 두번째 테이블에서 사용

intel 은 3단계의 페이지 테이블을 쓴다

MMU-less -> No Virtual Memory



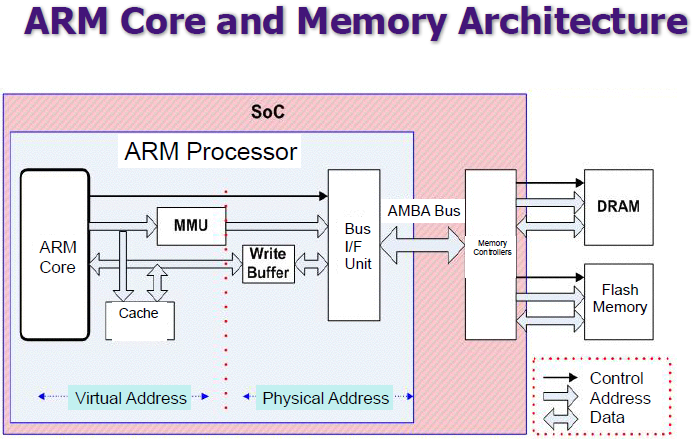
CO-PROCESSORS 15 의 특징

Virtual Memory (MMU)

Cache memory

Write buffer

ARM Core 가 CPU 였는데 진화,개발되어 SoC 가 CPU



컨텍스트 스위치 할 때

예전 프로세스의 Cache, TLB 는 현재 프로세스에 노쓸모

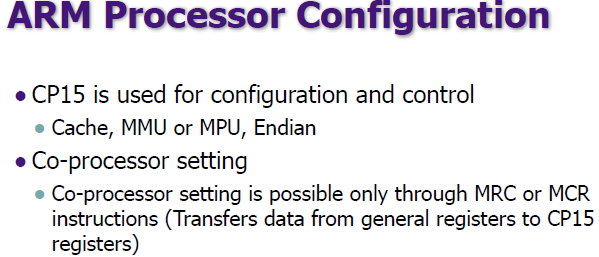
-> 캐시 메모리 삭제(Flush) : 밑으로 내려보낸다

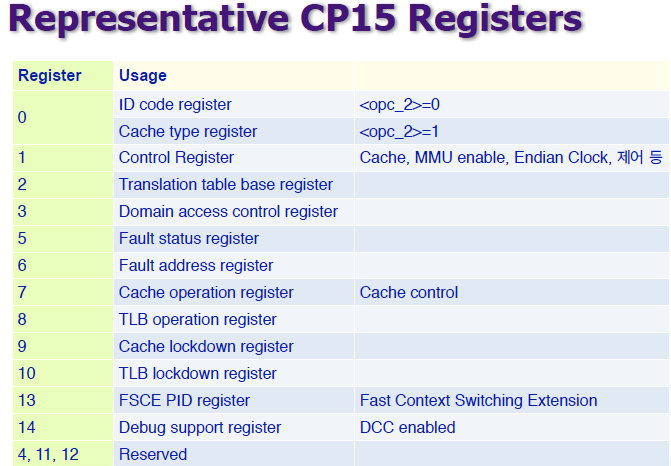
컨텍스트 스위치 자주하면 성능이 떨어진다

write through Cache 캐시에 쓰면 메모리에 바로 써진다

CO-PROCESSOR 15

Cache Flush, Cache Locking한다





register 1 중요하므로 조심 , 캐시 MMU 온오프 가능

-> 메모리 데이터와 캐시 데이터가 불일치 하는 문제

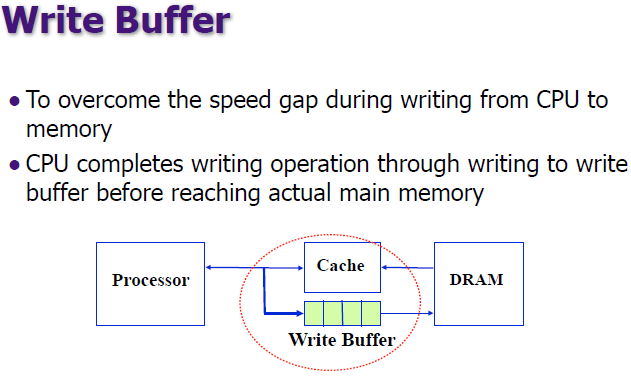
캐시 : 최신 데이터

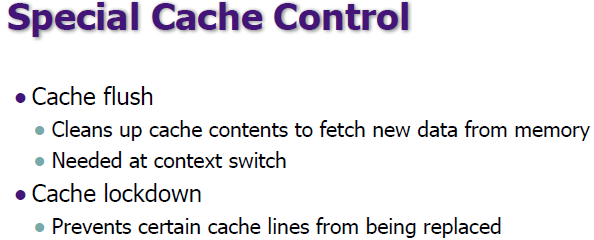
메모리 : 오래된 데이터

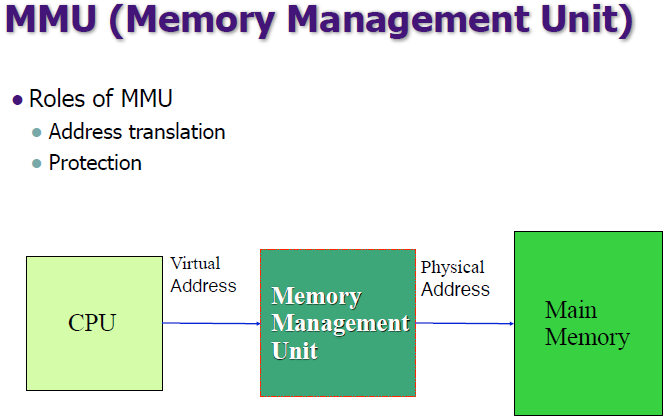
Cache Coherence problem

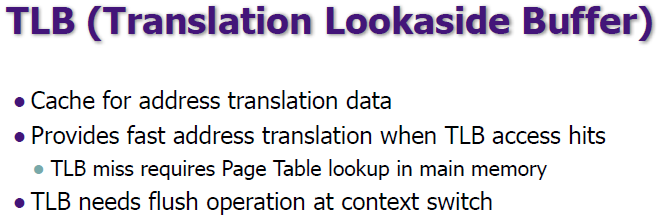
:multiple CPU , share memory 일 시 생기는 문제

한 캐시 데이터 -> 메모리, 다른 캐시는 메모리의 데이터 사용(오래된 데이터)





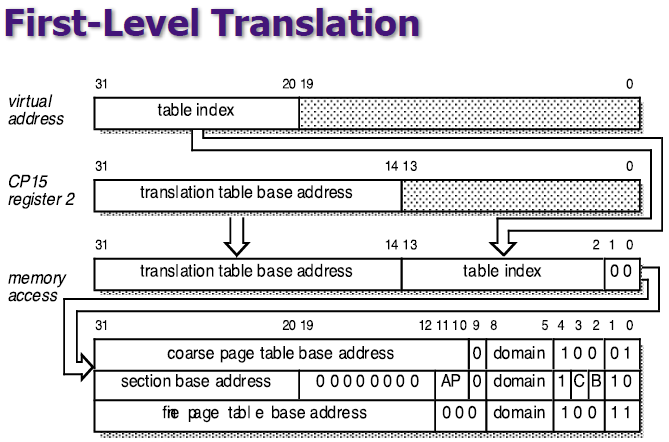


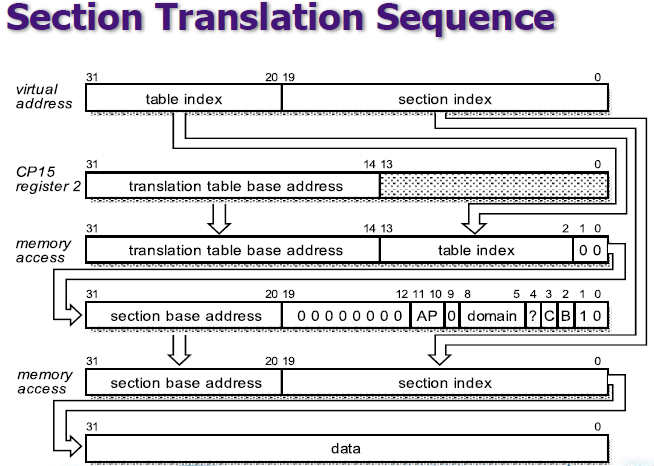


19/21

VA page 가 1M 일시 20 bits 가 필요

4k 는 12bits 필요 -> 나머지 비트는 또 다른 index를 쓴다





Memory Access

두번째줄 AP 는 사용자 모드에 따라 (관리자 X)

C 가 1 이면 캐싱 X

B 가 1 이면 write buffer X

맨뒤 두 비트 1 0 이면 1M 페이지

0 1 이면 4k 페이지