ARM System Developer's Guide
Designing and Optimizing System Software

ENC Lab.

Contents

ARM Embedded System

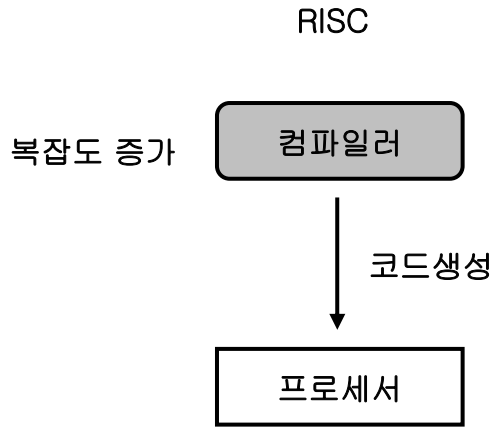
1st Semester in Master's
yhheo@enc.hanyang.ac.kr

Outline

- **RISC**의 특징
- **ARM** 프로세서의 특징
- **ARM** 하드웨어 임베디드 시스템 아키텍처
- 소프트웨어 계층 구조

RISC의 특징

- Reduced Instruction Set Computer
  - Fixed length for all instructions
  - Emphasis on software
  - Register-to-register instructions
  - Explicit LOAD and STORE instructions
  - Large code sizes



- Complex Instruction Set Computer
  - Varying instruction length
  - Emphasis on hardware
  - Better high-level language support
  - Memory-to-memory instruction
  - Small code sizes

ARM 프로세서의 특징 [1/2]

- Advanced RISC Machines (ARM)
  - 휴대형 임베디드 시스템
    - 배터리 전력 요구(저전력)
    - 작은 다이(die) 사이즈로 설계
  - ARM 프로세서내에 하드웨어 디버그 기술을 포함
  - 핵심요소 : 전체적으로 효율적인 시스템 성능 및 전력 소모
  - Low-power, low-cost embedded applications
    - Mobile telephones, PDA, etc.

ARM 프로세서의 특징 [2/2]

- 임베디드 시스템을 위한 명령어 세트
  - 가변 사이클로 실행되는 명령어
    - Ex) 다중 로드-스토어(multiple load-store) 명령어
  - 인라인 베럴 시프터(Inline Barrel Shifter)
    - 명령어에 의해 사용되기 전에 입력 레지스터 중의 하나를 미리 처리 해주는 H/W 컴포넌트
  - 16비트 Thumb 명령어
  - 조건부 실행
  - DSP 확장 명령어

  ☞ 성능향상과 코드 집적도 향상

- 가장 일반적인으로 사용되는 32bit 임베디드 프로세서 코어 중 하나

ARM 하드웨어 임베디드 시스템 아키텍처 [1/4]

- ARM 프로세서
  - 임베디드 디바이스를 제어
- 컨트롤러
  - 시스템에서 주요 기능을 담당하는 블록
- 주변장치
  - 칩 외부로의 입출력 기능을 제공해주는 컴포넌트
- 버스
  - 주변 장치간의 통신을 담당

ARM 하드웨어 임베디드 시스템 아키텍처 [2/4]

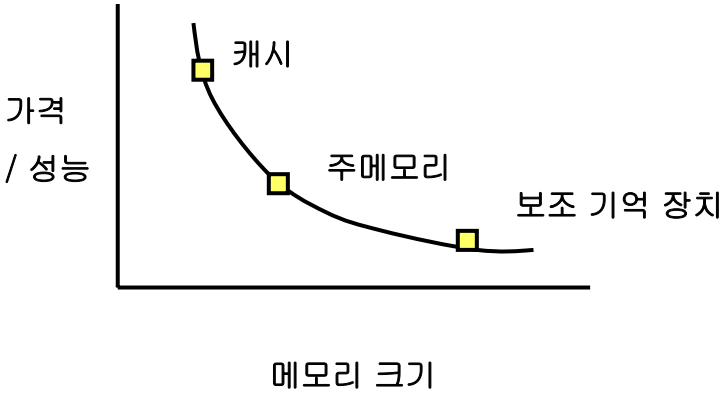
- ARM 버스 기술
  - 칩 내부 버스 사용
  - 다른 주변 장치가 ARM 코어와 내부에서 연결

- 버스에 연결될 수 있는 디바이스
  - 버스 마스터
    - 다른 디바이스로 데이터를 전송할 수 있는 논리 장치
  - 버스 슬레이브
    - 버스 마스터로부터 전송 요청이 있을 때에만 동작

- AMBA 버스 프로토콜 (Advanced Microcontroller Bus Architecture)
  - 프로세서와 주변장치를 연결
  - ARM은 버스 프로토콜만 규정
  - - ASB, APB => AHB

ARM 하드웨어 임베디드 시스템 아키텍처 [3/4]

- Memory



[그림] 메모리의 트레이드오프

- **메모리폭**
  - **한번에 액세스 할 수 있는 비트 수**
  - **보통 8, 16, 32,64비트 사용**
- **메모리의 종류**
  - ROM, DRAM, SRAM, SDRAM

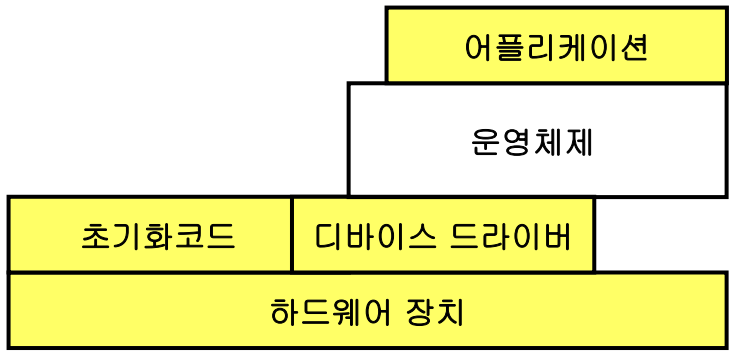
ARM 하드웨어 임베디드 시스템 아키텍처 [4/4]

- 주변장치
  - 모든 ARM 주변장치는 메모리에 매핑(memory-mapped)

- 인터럽트 컨트롤러
  - 인터럽트 컨트롤러 레지스터 안에 해당 비트를 1로 설정
  - 디바이스 혹은 주변장치가 특정시간에 인터럽트를 발생시킬 수 있도록 할 지를 결정함
  - 표준 인터럽트 컨트롤러, 벡터 인터럽트 컨트롤러(인터럽트에 우선순위 부여)가 있음

소프트웨어 계층 구조

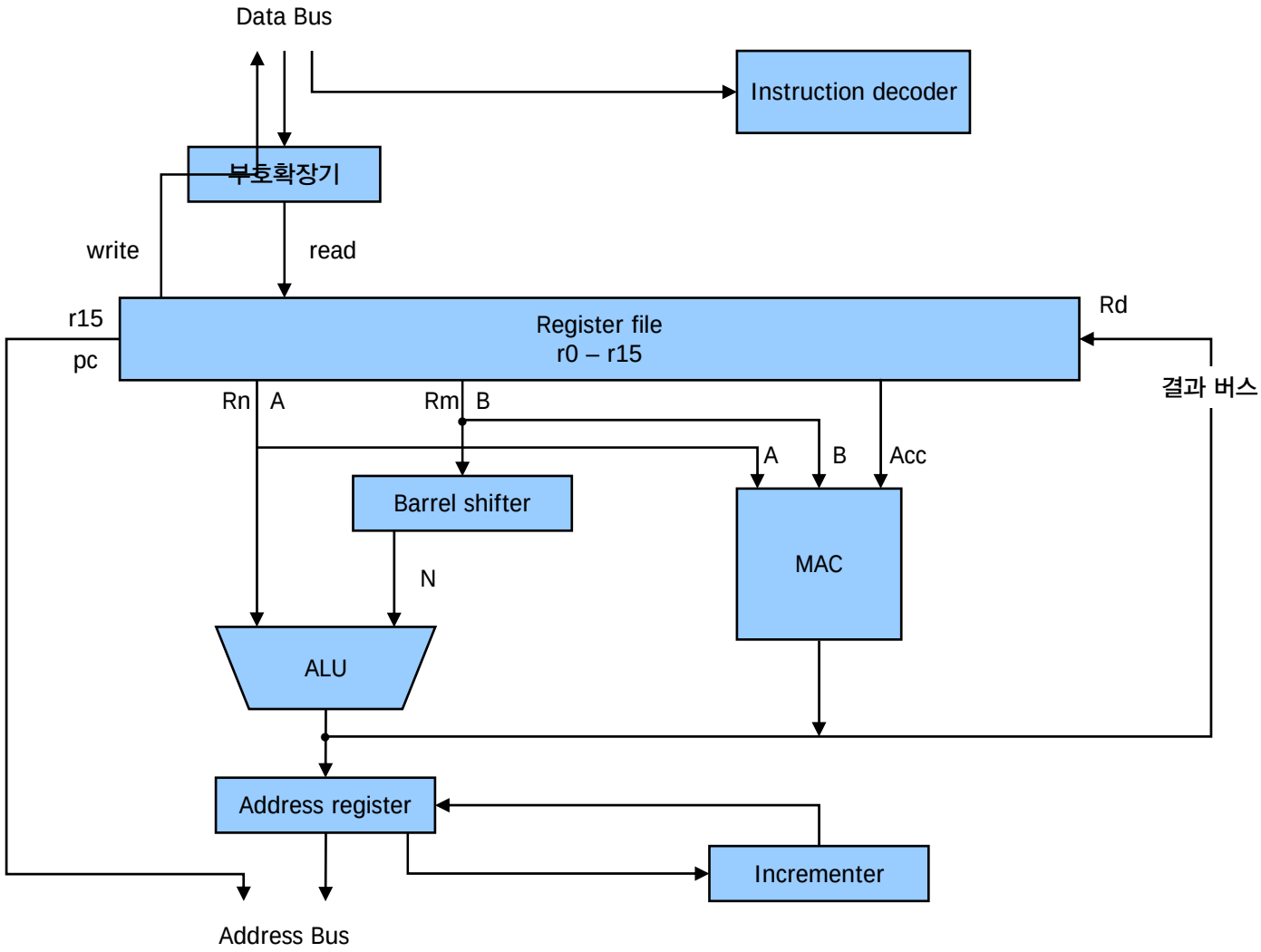
- 초기화 코드
  - 하드웨어 장치들을 초기화 해주는 코드
- 운영체제
  - 초기화 과정 다음에 실행될 코드
  - 하드웨어 리소스와 인프라스트럭처를 사용하기 위한 일반적인 프로그래밍 환경 제공
- 디바이스 드라이버
  - 주변 장치와의 표준 인터페이스를 제공
- 어플리케이션
  - 임베디드 시스템의 특정 테스크를 수행



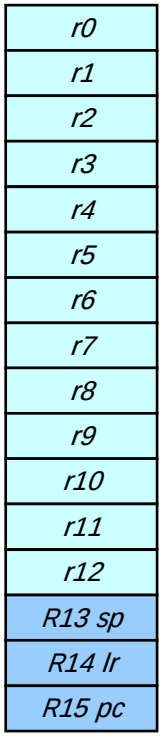
ARM Processor Fundamentals

Jae Hyoung Jung
jhjung@enc.hanyang.ac.kr

ARM core data flow model



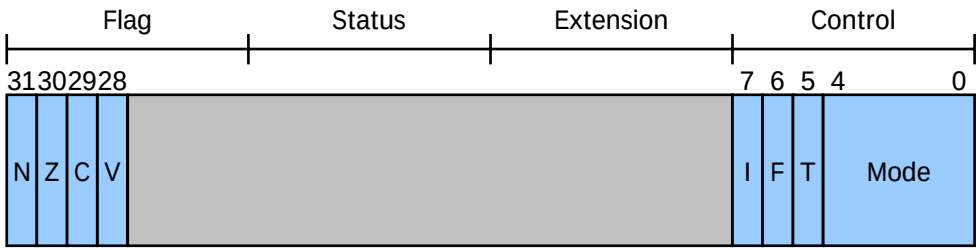
General purpose register



- 데이터나 주소를 저장
- 16개의 데이터 레지스터(r0 ~ r15)와 상태 레지스터로 구성
- Special purpose register
  - r13 : Stack pointer
    - 프로세서 모드의 스택 맨 위의 주소값을 저장
  - r14 : Link register
    - 코어가 서브루틴을 호출할 때마다 그 복귀 주로를 저장
  - r15 : Program counter
    - 프로세서가 읽어드린 다음 명령어의 주소를 저장

CPSR register

- CPSR : Current Program Status Register
  - ARM 코어 내부동작 모니터링 및 제어 족적으로 사용
  - 전용 32비트 레지스터로 레지스터 파일 안에 위치
  - 32비트를 8비트씩 각각 Flag, Status, Extension, Control의 4 영역으로 나누어짐
- SPSR : Stored Program Status Reigster



Current Program Status Register

Processor mode

- 프로세서 모드란?
  - 어떤 레지스터가 활성화되고, cpsr 레지스터를 액세스할 수 있는 권리를 갖게 될 지를 결정
- 프로세서 모드의 형태
  - 특권모드 – cpsr read/write
  - 일반모드 – cpsr read only, status flag read/write
- cpsr을 직접 제어하여 모드 변경을 한 경우에는 cpsr이 spsr에 복사되지 않음
- spsr은 특권 모드에서만 수정/읽기가 가능, user 모드에서 접근 불가

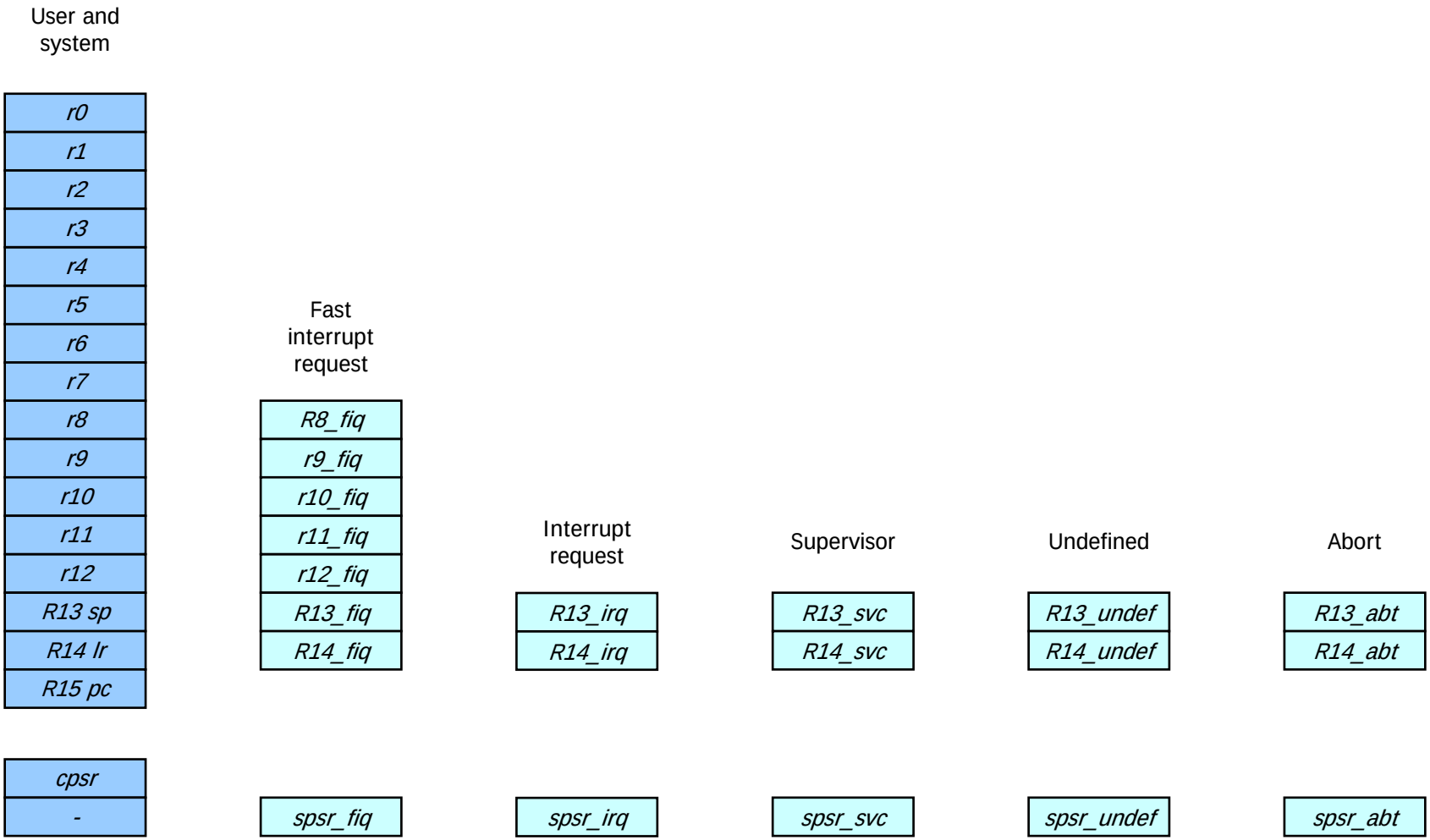
Processor mode

프로세서 모드의 종류

- Abort mode
  - 메모리 억세스가 실패했을 경우 진입
- FIQ & IRQ mode
  - ARM의 두 가지 인터럽트 레벨을 위한 모드
- Supervisor mode
  - 프로세서 리셋 시 진입, 일반적으로 OS 커널이 동작하는 모드
- System
  - User 모드의 특수한 버전 cpsr을 완전히 읽고 쓰는 것이 가능
- Undefined mode
  - 프로세서가 정의되지 않은 명령어나 지원되지 않은 명령어를 만났을 때 진입
- User mode
  - 프로그램과 어플리케이션을 위해 사용

모드	약자	특권기능	모드 비트 [4:0]
Abort	abt	yes	10111
Fast interrupt request	fiq	yes	10001
Interrupt request	irq	yes	10010
Supervisor	svc	yes	10011
System	sys	yes	11111
Undefined	und	yes	11011
User	usr	no	10000

Bank register



Processor state & Instruction set

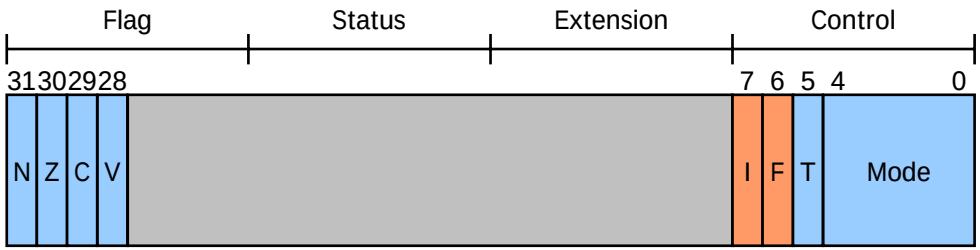
- ARM processor's instruction set
  - ARM instruction
  - Thumb instruction (16bits)
  - Jazelle instruction (Java extension)

	ARM (cpsr T = 0)	Thumb (cpsr T = 1)
명령어 크기	32비트	16비트
코어 명령어	58	30
조건부 실행	대부분 가능	분기 명령어서만 가능
데이터 처리 명령어	배럴 시프터와 ALU의 액세스 가능	배럴 시프터 명령어와 ALU 명령어가 분리됨
프로그램 상태 레지스터	특권 모드에서만 읽고 쓰기 가능	직접 액세스 불가
레지스터 사용	15개의 범용 레지스터 + pc	8개의 범용 레지스터 + 7개의 상위 레지스터 + pc

	Jazelle (cpsr T = 0, J = 1)
명령어 크기	8비트
코어 명령어	Java 코드의 60% 이상은 하드웨어적으로 구현되었고 나머지는 소프트웨어적으로 구현됨

Interrupt mask

- 인터럽트 마스크란?
  - 특정 인터럽크 소스가 프로세서에게 인터럽트 요청을 할 수 없도록 하는 것
  - ARM 프로세서서 코어는 IRQ와 FIQ 두 가지의 인터럽트 요청 레벨을 가짐
  - cpsr의 6^{th} bit – F : FIQ mask
  - cpsr의 7^{th} bit – I : IRQ mask



Current Program Status Register

Status flag

- 비교명령어나 산술 명령어 뒤에 S가 붙은 경우에 업데이트
- 상태플래그에 따라 조건부 실행 가능

플래그	플래그명	1로 세트되는 경우
Q	Saturation	오버플로우나 포화가 발생하는 경우
V	oVerflow	Signed 오버플로우가 발생하는 경우
C	Carry	Unsigned 캐리 발생하는 경우
Z	Zero	결과가 0 인 경우, 종종 동일함을 표현하기 위해 사용
N	Negative	결과의 31번째 비트가 1인 경우

Conditional Execution

- 조건부 실행 : Conditional Execution
  - 어떤 명령어를 실행할 지의 여부를 제어가능
  - 명령어들은 코어가 그것을 실행할지 여부를 결정할 수 있는 조건 인자를 가지고 있음
  - 명령어를 실행하기 전에 코어는 자신이 가지고 있는 조건 인자와 cpsr의 상태 플래그 비교
    - 일치하는 경우 명령어 실행, 불일치 명령어는 무시

Mnemonic	Meaning	Status flag
EQ	equal	Z
NE	not equal	z
CS HS	carry set/unsigned higher or same	C
CC LO	carry set/unsigned lower	c
MI	minus/negative	N
PL	plus/positive or zero	n
VS	overflow	V
VC	no overflow	v
HI	unsigned higher	zC
LS	unsigned lower or same	Z or c
GE	signed greater than or equal	NV or nv
LT	signed less than	Nv or nV
GT	signed greater than	NzV or nzv
LE	signed less than or equal	Z or Nv or nV
AL	always (always run)	ignored

Pipeline

- RISC의 파이프라인 (3 stages)
  - Fetch — 메모리에서 명령어를 로드
  - Decode — 실행한 명령어를 해독
  - Execute — 명령어를 처리, 결과를 레지스터에 저장

- 파이프라인의 특징
  - 분기 명령어로 실행되거나 pc 값을 직접 수정하여 분기하는 경우에는 파이프라인이 깨짐
  - ARM10은 분기 예측 방식을 사용, 명령어를 실행하기 전에 가능한 분기를 미리 예측하여 새로운 분기 주소를 로드하여 파이프라인이 깨지는 상황을 줄여줌
  - 실행단계에 있는 명령어는 인터럽트가 발생하더라도 그 과정을 완료

- ARM의 파이프라인
  - ARM7 3단 파이프라인 — Fetch – Decode – Execute
  - ARM9 5단 파이프라인 — Fetch – Decode – Execute – Memory – Write
  - ARM10 6단 파이프라인 — Fetch – Issue – Decode – Execute – Memory – Write

ARM 프로세서의 익셉션

- 익셉션이나 인터럽트 발생시 프로세서는 pc에 특정 메모리 주소값을 넣음
  주소값은 벡터 테이블이라는 “특정 주소 영역”안에 있는 값

- 익셉션 벡터 테이블의 종류
  - Reset vector
    - 전원이 공급될 때 프로세서에 의해 처음으로 실행되는 명령어의 위치, 이 명령어에서 초기화 코드로 분기
  - Undefined instruction vector
    - 프로세서가 명령어를 분석할 수 없는 경우에 사용
  - Software interrupt vector
    - SWI 명령어를 실행시켰을 때 호출, SWI는 운영체제 루틴에서 벗어나고 할 경우에 주로 사용
  - Prefetch abort vector
    - 프로세서가 정확한 접근권한없이 어떤 주소에서 명령어를 읽어드리고 시도할 때 발생, 실제 abort는 파이프라인의 디코드 단계에서 발생
  - Data abort vector
    - Prefetch abort와 유사, 명령어가 정확한 접근권한 없이 데이터 메모리를 억세스하려고 시도할 때 발생
  - Interrupt request vector
    - 프로세서의 현재 흐름을 중단시키기 위해 외부 하드웨어 장치에 의해 사용, IRQ가 cpsr에서 마스크 되지 않은 경우에만 발생
  - Fast Interrupt request vector
    - IRQ와 유사, 더욱 빠른 응답시간을 요하는 하드웨어에 할당, FIQ가 cpsr에서 마스크되지 않은 경우에만 발생

Architecture Naming

- ARM {x}{y}{z}{T}{D}{M}{I}{E}{J}{F}{-S}
  - x Family
  - y MMU/MPU
  - z Cache
  - T Thumb extension
  - D JTAG Debug
  - M Hard multiplier
  - I Embedded ICE extension
  - E DSP enhanced
  - J Java extension
  - F VFP (Vector Floating-Point unit) enhanced
  - S Synthesizible version

ARM 포로세서의 버전별 특징

버전	코어의 예	향상된 ISA
ARMv1	ARM1	최초의 ARM 프로세서 26비트 어드레싱
ARMv2	ARM2	32비트 곱셈기 32비트 코프로세서 지원
ARMv2a	ARM3	온칩 캐시 swap 명령어 캐시 관리를 위한 코프로세서 15
ARMv3	ARM6 & ARM7DI	32비트 어드레싱 cpsr과 spsr의 분리 새로운 모드 추가 : undefined instruction과 abort MMU 지원 : 가상 메모리
ARMv3M	ARM7M	Signed/unsigned 곱셈 확장 명령어
ARMv4	Strong ARM	Signed/unsigned 하프워드/바이트 로드-스토어 명령어 새로운 모드 추가 : system 아키텍쳐별로 정의된 동작을 위한 reverse SWI 공간 26비트 주소지정방식을 더 이상 지원하지 않음
ARMv4T	ARM7TDMI & ARM9T	Thumb
ARMv5TE	ARM9E & ARM10E	ARMv4T의 수퍼셋 ARM과 Thumb 명령어 사이에서 상태를 변경하기 위한 명령어가 추가됨 향상된 곱셈 명령어 추가된 DSP형 명령어 보다 빠른 곱셈 누산기
ARMv5TEJ	ARM7EJ & ARM926EJ	Java 가속기
ARMv6	ARM11	향상된 프로세서 명령어 정렬되지 않고 섞인 엔디안 데이터 처리 새로운 멀티미디어 명령어

ARM 제품군별 특징 비교 & ARM 프로세서 종류

	ARM7	ARM9	ARM10	ARM11
Pipeline stages	3	5	6	8
Speed	80	150	260	335
mW/MHz	0.06mW/MHz	0.19mW/MHz (+cache)	0.5mW/MHz (+cache)	0.4mW/MHz (+cache)
MIPS/MHz	0.97	1.1	1.3	1.2
Architecture	Von-Neumann	Harvard	Harvard	Harvard
Multiplier	8x32	8x32	16x32	16x32

CPU core	MMU/MPU	Cahche	Jazelle	Thumb	ISA	E
ARM7TDMI	none	none	no	yes	v4T	no
ARMEJ-S	none	none	yes	yes	V5TEJ	yes
ARM720T	MMU	통합된 8K cache	no	yes	v4T	no
ARM920T	MMU	분리된 16K/16K D+ I cache	no	yes	v4T	no
ARM922T	MMU	분리된 8K/8K D+ I cache	no	yes	v4T	no
ARM926EJ-S	MMU	분리된 캐시와 조절 가능한 TCM	yes	yes	v5TEJ	yes
ARM940T	MPU	분리 조절 가능한 TCM	no	yes	v4T	no
ARM946E-S	MPU	분리 조절 가능한 TCM	no	yes	v5TE	yes
ARM966E-S	none	분리된 32K/32K D+ I cache	no	yes	v5TE	yes
ARM1020E	MMU	분리된 16K/16K D+ I cache	no	yes	v5TE	yes
ARM1022E	MMU	분리된 캐시와 조절 가능한 TCM	no	yes	v5TE	yes
ARM1026EJ-S	MMU & MPU	분리된 캐시와 조절 가능한 TCM	yes	yes	v5TE	yes
ARM1136J-S	MMU	분리된 캐시와 조절 가능한 TCM	yes	yes	v6	yes
ARM1136JF-S	MMU	분리된 16K/16K D+ I cache	yes	yes	v6	yes

제품군별 특징

- ARM7 군
- ARM9 군
- ARM10 군
  - 파이프라인을 6단계로 확장
  - VFP (Vector Floating-Point) 사용시 pipeline stage의 7번 째 stage로 추가 가능
- ARM11 군
  - 고성능 저전력의 어플리케이션을 위해 설계
- 이외의 특별한 제품군
  - Strong ARM
    - Digital Semiconductor 와 합작, 현재 intel 사에서 라이센스 보유
    - Harvard architecture, 5 stages pipeline
  - XScale
    - Strong ARM의 다음 버전 (up to 1GHz)
    - v5TE ISA, Harvard architecture, Strong ARM과 유사한 MMU
  - SC100
    - 저전력 보안 장치를 위해 설계
    - ARM7TDMI 기반, MPU

Summary

- ARM processor's components
  - ALU, Barrel shifter, MAC, Register file, Instruction decoder, Address register, incrementer, 부호확장기

- ARM의 특징
  - ARM, Thumb, Jazelle 명령어 세트
  - Register file에 총 37개가 포함, 한 시점에서는 17개 or 18개의 레지스터만 사용가능, 나머지는 프로세서 모드에 따라 숨겨져 있음
  - cpsr은 현재 프로세서의 상태, 인터럽트 마스크, 상태 플래그, 상태정보 및 프로세서 모드가 저장됨, 상태정보는 명령어세트를 결정
  - ARM 프로세서는 코어와 버스로 인터페이스 되어 있는 주변 장치들로 이루어짐

- 확장된 코어는 다음과 같은 것들이 포함됨
  - Cache, TCM, MMU, Co-processor

Introduction to the ARM Instruction Set

Juyoung Kim

jykim@enc.hanyang.ac.kr

Outline

- Introduction
- Characteristics
- Category
  - Arithmetic
  - Branch
  - Single-Register Load-Store
  - Multiple-Register Load-Store
  - others

Introduction

- ARM instruction set
  - 하위 버전의 ISA와 호환성을 가진다.
  - 버전에 따라 새로운 명령어가 추가되기도 한다.
  - 버전에 따라 기존 명령어의 기능이 확장되기도 한다.
    - Mnemonic이 추가되기도 하고 기존 것을 유지하기도 한다.

Characteristic

- ALU
  - Barrel shifter
    - 한 operand의 bit shift 전처리 기능을 한다.
    - Shift에 소모되는 cycle을 절약하여 성능을 향상시킨다.
    - 종류
      - LSL , LSR : Logical shift (Unsigned operation)
      - ASR : Arithmetic shift (Signed operation)
      - ROR : Rotation shift
      - RRX
    - 표기법
      - {register} <명령어> {shift value}
    - 특징
      - 상수 값을 2^n으로 곱하거나 나눌 때 유용.
      - 몇몇 명령어에서는 사용 불가.

- CPSR (Current Program state register)
  - 명령어 뒤에 'S'가 붙으면 갱신된다.
  - 구성
    - **C** : Carry
    - **N** : Negative
    - **Z** : Zero
    - **V** : Overflow
  - 특징
    - 각 flag들을 참조하여, 산술 및 비교연산을 수행한다.

Arithmetic

- Arithmetic instruction
  - 산술 연산
    - 표기법
      - <명령어>{<조건>}{S} Rd, Rn, N
    - 종류
      - ADD, ADC : (Carry를 고려한)덧셈
      - SUB, SBC : (Carry를 고려한)뺄셈
      - RSB, RSC : (Carry를 고려한)반전
  - 논리 연산
    - 표기법
      - <명령어>{<조건>}{S} Rd, Rn, N
    - 종류
      - AND, ORR EOR, BIC
  - 비교 연산
    - 표기법
      - <명령어>{<조건>}{S} Rn, N
    - 종류
      - CMN, CMP : (음/양)비교
      - TEQ , TST : (같음/대소)비교
  - 곱셈 연산
    - 표기법
      - MLA{<조건>}{S} Rd, Rm, Rs, Rn
      - MUL{<조건>}{S} Rd, Rm, Rs

- Rd : Destination Register
- Rn, Rm : Source Register
- N : Constant

Branch

- 표기법
  - B<X><L>{<조건>} lable | Rm
    - L : Sub routine으로 분기
      - lr (Link register) 에 복귀할 주소를 저장.
      - Sub routine 이 끝나면 PC에 lr을 대입.
    - X : Thumb mode로 전환
      - Rm 에 저장된 주소를 Thumb mode의 주소로 사용.
      - CPSR의 Thumb mode를 나타내는 T flag가 1이 됨.

Single-Register Load-Store

- 표기법
  - <LDR|STR>{<조건>}{B} Rd, addressing
  - LDR{<조건>}SB|H|SH Rd, addressing
  - STR{<조건>}H Rd, addressing
    - Word단위 연산이 기본, H인 경우 Half word, B인 경우 Byte 연산.

- Indexing
  - Auto-index
    - [base, offset]! ➔ *(base + offset); base += offset;
  - Pre-index
    - [base, offset] ➔ *(base + offset);
  - Post-index
    - [base], offset ➔ *(base); base += offset;
  - Offset
    - Signed/Unsigned Immediate, Register value 가 사용 가능
    - Barrel shift를 통한 연산이 가능.

Multiple-Register Load-Store

- 표기법
  - <LDM|STM>{<조건>}<주소 지정 방식> Rn{!}, <Register>{^}
  - !는 Auto-indexing
  - ^는 CPSR register의 추가
- 특징
  - Code size의 감소
  - Memory에서 여러 block data를 전송하는데 용이
  - Syntax과 Stack을 저장, 복구하는데 효과적
- 주소 지정 방식
  - <I/D><A/B>
    - Increase, Decrease
      - Address가 Word 단위로 증(I)감(D).
    - After, Before
      - Base register의 위치를 포함하는가(After) 아닌가(Before).

others

- Data swap
  - 표기법
    - SWP{B}{<조건>} Rd, Rm, [Rn]
      - Word단위 교환, B인 경우 Byte.
- Software interrupt
  - 표기법
    - SWI{<조건>} SWI_number
- PSR set
  - MRS{<조건>} Rd, <cpsr | spsr>
    - CPSR, SPSR의 값을 register로 읽어오기
  - MSR{<조건>} <cpsr | spsr>_<fields> ,Rm
  - MSR{<조건>} <cpsr | spsr>_<fields> ,#Immediate
    - Register나 Immediate의 값을 CPSR, SPSR에 대입.
- Conditional execution
  - CPSR의 상태에 따라 명령어의 실행 여부를 결정.
  - Example
    - ADREQ : Z flag 가 1 일 때만 명령어를 실행.
  - 특징
    - 특정 알고리즘 수행 시 필요한 명령어의 수행을 줄일 수 있다.

Introduction to the Thumb Instruction Set

Juyoung Kim

jykim@enc.hanyang.ac.kr

Outline

- Introduction
- Characteristics
- ARM-Thumb inter-working
- Category
  - Branch
  - Arithmetic
  - Load-Store
  - Stack control
  - Software interrupt (SWI)

Introduction

- Thumb mode의 필요성
  - Low memory capacity system 에서 code size를 줄임
  - 16 bits bus system 에서 bus utilization 을 높임
  - ARM4T 모델부터 사용 가능하게 됨

- 효과
  - Code size
    - 동일한 기능의 ARM instruction code에 비해 30%의 code size 절약
  - Bus utilization
    - 16 bits bus system에서 2번 전송이 필요한 ARM 명령어와 달리 Thumb 명령어는 1번의 전송으로 가능

- Code density

+ ARM

```
        MOV     R3,#0
LOOP    SUBS    R0,R0,R1
        ADDGE   R3,R3,#1
        BGE     LOOP
        ADD     R2,R0,R1
```

5 * 4 Bytes = 20 Bytes

+ Thumb

```
        MOV     R3,#0
LOOP    ADD     R3,#1
        SUB     R0,R1
        BGE     LOOP
        SUB     R3,#1
        ADD     R2,R0,R1
```

6 * 2 Bytes = 12 Bytes

Characteristic

- Half-word aligned
- Difference between Thumb mode and ARM mode
  - Barrel shift를 이용한 연산을 사용하지 못함.
    - 명령어 길이의 제약
    - Shift 연산 명령어를 독립적으로 가짐
  - Conditional execution
    - Branch 명령어만 가능
  - Register access
    - 하위 Register[r0 – r7] : 모든 명령어가 접근 가능
    - 상위 Register[r8 – r12] : 몇몇을 제외한 명령어는 접근 불가능
    - SP, LR, PC : 제한적 접근 (명령어를 통한 접근)
    - CPSR : 간접적 접근 (자동 접근)
    - SPSR : 접근 불능

ARM-Thumb inter-working

- LSB of the address of an instruction is always Zero
  - ARM mode에서 Thumb mode로 전환될 때, LSB 가 1 인 것을 유효한 Thumb mode로 인식
- 명령어
  - BX Rm
    - PC = Rm & 0xfffffffe
    - CPSR의 T = Rm[0]
  - BLX Rm | Label
    - LR = current PC + 1
    - PC = Label | Rm & 0xfffffffe
    - CPSR의 T = 0 | Rm[0]
- 만약, LSB가 1이 아니라면 일반 Branch 명령어와 동일한 기능을 함.

Branch

- 표기법
  - B {<조건>} label
    - 유일하게 조건적으로 실행되는 명령어
    - <조건> 이 있을 경우 signed 8 bits, 없을 경우 signed 11 bits 의 offset field를 가진다.
  - BL label
    - 조건부 실행이 불가
    - Signed 11 bits * 2개 만큼의 offset field를 가짐.
      - BL 명령어는 한 번의 사용 시, 두 개(High offset, Low offset)의 BL명령어로 변환됨.

Arithmetic

- Arithmetic instruction
  - Barrel shift를 사용할 수 없음.
    - ASR, LSL, LSR등의 명령어를 독립된 형태로 가짐
  - 하위 Register[r0 – r7] 에서만 동작
    - MOV, ADD, SUB , CMP 인 경우에만 가능
  - CPSR을 update함
    - 상위 Register사용 시, CMP를 제외한 명령어는 CPSR을 update하지 않음

Load-Store

- Single-register instruction
  - ARM 에서와 유사.
    - 하지만 Conditional execution이 없음
  - CPSR update ‘S’가 없음

- Multi-register instruction
  - 표기법
    - <LDM|STM>IA Rn!, {하위 레지스터 리스트}
  - 주소 지정방식 : IA (Increase ,After) 만을 지원
  - Auto-index (!) 가 필수

SWI

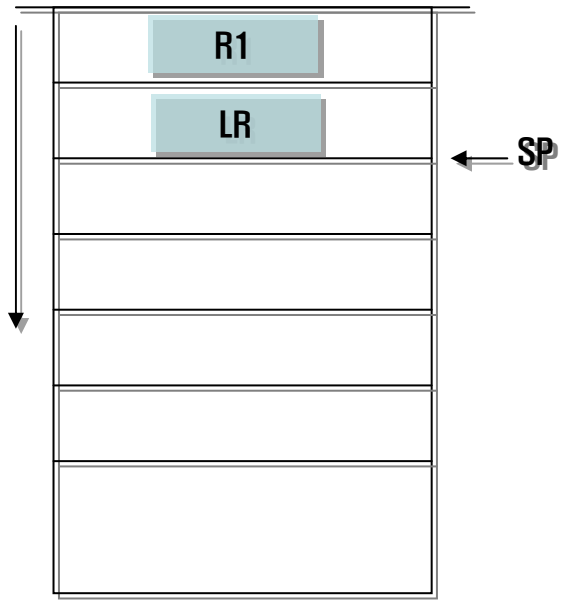
- Software Interrupt가 발생하면 exception 처리를 위해 Thumb에서 ARM으로 mode가 변경됨
- CPSR의 I = 1 , T = 0으로 변환.

Stack control

- 표기법
  - POP {low_register_list {,PC}}
  - PUSH {low_register_list {, LR}}
- Stack pointer가 R13으로 고정되어 있으며, 항상 auto-update됨
- Full descending stack만을 지원
- 예제

Thumb

```
PUSH {R1, LR}     : R1, LR을 stack에 저장.
MOV  R0,#2
POP  {R1, PC}     : R1을 복원, PC를 LR로 복원
```



Efficient C Programming

Seong Min Jo

smcho@enc.hanyang.ac.kr

Contents

C 컴파일러에 대한 다양한 오해

- 코드 최적화
  - 자주 실행되고 성능에 중요한 영향을 끼치는 함수를 중점으로 최적화
  - 성능 측정 툴 사용 권장
- C 컴파일러
  - 일반적으로 보수적인 (conservative) 특징
  - 효과적인 C 코드 작성 방법
    - 어떤 경우에 C 컴파일러가 보수적
    - 사용되는 프로세서 아키텍처의 한계
    - 특정 C 컴파일러의 제약사항
- 책에서 나오는 예제에 사용되는 컴파일러
  - armcc : ARM에서 제공하는 컴파일러
  - arm-elf-gcc 2.95.2 : GNU C 컴파일러

C 컴파일러에 대한 다양한 오해

```
void memclr (char *data, int N)
{
    for (; N>0; N--)
    {
        *data = 0;
        data ++;
    }
}
```

- N = 0?
- 4-byte 정렬?
- N이 4의 배수

적절한 데이터 형의 선택

- ARM 프로세서
  - 32-bit register
  - RISC load-store architecture

아키텍처	명령어	설명
ARMv4 이전	LDRB	unsigned 8비트 값을 레지스터로 읽어 들임
	STRB	signed 또는 unsigned 8비트 값을 메모리에 저장
	LDR	signed 또는 unsigned 32비트 값을 레지스터로 읽어 들임
	STR	signed 또는 unsigned 32비트 값을 메모리에 저장
ARMv4	LDRSB	signed 또는 8비트 값을 레지스터로 읽어 들임
	LDRH	unsigned 16비트 값을 레지스터로 읽어 들임
	LDRSH	signed 16비트 값을 레지스터로 읽어 들임
	STRH	signed 또는 unsigned 16비트 값을 메모리에 저장
ARMv5	LDRD	signed 또는 unsigned 64비트 값을 레지스터로 읽어 들임
	STRD	Signed 또는 unsigned 64비트 값을 메모리에 저장

적절한 데이터 형의 선택

- ARMv4 이전 프로세서
  - char : ARM 컴파일러에서 자동적으로 unsigned 8비트로 인식
  - 포팅 시 문제
  - 예)

```
...
for(i = 10; i >= 0; i--){
        ...
}
...
```

  - armcc
    - Warning message : unsigned comparison with 0
  - gcc
    - Command-line option : -fsigned-char

C 데이터 형	기능
char	unsigned 8비트 (바이트)
short	signed 16비트 (하프워드)
int	signed 32비트 (워드)
long	signed 32비트 (워드)
long long	signed 64비트 (더블워드)

적절한 데이터 형의 선택

- 지역변수
  - 8-bit 또는 16-bit 특성을 이용하지 않는 산술연산일 경우 int 형 사용
- 메모리에 저장되는 배열 또는 전역 변수
  - 필요한 데이터를 저장할 수 있는 가장 작은 데이터 형 사용
    - 메모리 절약
    - ARMv4 아키텍처의 어드레싱을 이용
      - 배열 포인터를 증가시키며 데이터 엑세스
    - 단 LDRH의 경우 short 형을 지원하지 않기 때문에 short 형 배열은 피해야 함
  - 배열의 각 요소 또는 전역 변수를 읽거나 지역 변수를 저장할 때
    - 명시적인 캐스트 연산 사용
- 형 변환 (Casting)
  - 명시적이든 암시적이든 narrow 캐스트는 피함
    - 추가적인 사이클 필요
  - 로드-스토어 명령어는 명령어 자체가 캐스트를 수행함
- 함수 인자, 리턴값
  - char, short 형을 사용하지 않도록 함
    - 컴파일러에 의한 불필요한 캐스트 연산을 수행하지 않음

예제 – 32비트 체크섬 계산 1

C 코드

```
int checksum_v1(int *data)
{
    char i;
    int sum = 0;
    for(i = 0; i < 64; i++)
    {
        sum += data[i];
    }
    return sum;
}
```

Assembly 코드

```
checksum_v1
    MOV r2, r0              ; r2 = data
    MOV r0, #0              ; sum = 0
    MOV r1, #0              ; i = 0
checksum_v1_loop
    LDR r3, [r2, r1, LSL #2]  ; r3 = data[i]
    ADD r1, r1, #1          ; r1 = i + 1
    AND r1, r1, #0xff       ; i = (char)r1
    CMP r1, #0x40           ; i와 64비교
    ADD r0, r3, r0          ; sum += r3
    BCC checksum_v1_loop    ; (i < 64) 이면 loop로 분기
    MOV pc, r14             ; sum 리턴
```

예제 - 32비트 체크섬 계산 2

C 코드

```
int checksum_v2(int *data)
{
    unsigned int i;
    int sum = 0;
    for(i = 0; i < 64; i++)
    {
        sum += data[i];
    }
    return sum;
}
```

Assembly 코드

```
checksum_v2
    MOV r2, r0              ; r2 = data
    MOV r0, #0              ; sum = 0
    MOV r1, #0              ; i = 0
checksum_v2_loop
    LDR r3, [r2, r1, LSL #2]    ; r3 = data[i]
    ADD r1, r1, #1          ; r1++
    CMP r1, #0x40           ; i와 64비교
    ADD r0, r3, r0          ; sum += r3
    BCC checksum_v2_loop    ; (i < 64) 이면 loop로 분기
    MOV pc, r14             ; sum 리턴
```

예제 – 16비트 체크섬 계산 1

C 코드

```
short checksum_v3(short *data)
{
        unsigned int i;
        short sum = 0;
        for(i = 0; i < 64; i++)
        {
          sum = (short)(sum + data[i]);
        }
        return sum;
}
```

Assembly 코드

```
checksum_v3
        MOV r2, r0              ; r2 = data
        MOV r0, #0              ; sum = 0
        MOV r1, #0              ; i = 0
checksum_v3_loop
        ADD r3, r2, r1, LSL #1  ; r3 = &data[i]
        LDRH r3, [r3, #0]       ; r3 = data[i]
        ADD r1, r1, #1          ; i++
        CMP r1, #0x40           ; i와 64비교
        ADD r0, r3, r0          ; r0 = sum + r3
        MOV r0, r0, LSL #16
        MOV r0, r0, ASR #16     ; sum = (short)r0
        BCC checksum_v3_loop    ; (i < 64) 이면 loop로 분기
        MOV pc, r14             ; sum 리턴
```

예제 – 16비트 체크섬 계산 2

C 코드

```
short checksum_v4(short *data)
{
        unsigned int i;
        int sum = 0;
        for(i = 0; i < 64; i++)
        {
          sum = *(data++);
        }
        return (short)sum;
}
```

Assembly 코드

```
checksum_v4
        MOV r0, #0              ; sum = 0
        MOV r1, #0              ; i = 0
checksum_v4_loop
        LDRSH r3, [r0], #2      ; r3 = *(data++)
        ADD r1, r1, #1          ; i++
        CMP r1, #0x40           ; i와 64비교
        ADD r2, r3, r2          ; sum += r3
        BCC checksum_v4_loop    ; (i<64)이면 loop분기
        MOV r0, r2, LSL #16
        MOV r0, r0, ASR #16     ; r0=(short)sum
        MOV pc, r14             ; sum 리턴
```

예제

C 코드

```
short add_v1(short a, short b)
{
    return (short)(a + (b >> 1));
}
```

Assembly 코드

+ armcc

```
add_v1
    ADD r0, r0, r1, ASR #1          ; r0=(int)a + ((int)b >> 1)
    MOV r0, r0, LSL #16
    MOV r0, r0, ASR #16             ; r0 = (short)r0
    MOV pc, r14
```

+ gcc

```
add_v1_gcc
    MOV r0, r0, LSL #16
    MOV r1, r1, LSL #16
    MOV r1, r1, ASR #17             ; r1 = (int)b >> 1
    ADD r1, r1, r0, ASR #16         ; r1 += (int)a
    MOV r1, r1, LSL #16
    MOV r0, r1, ASR #16             ; r0 = (short)r1
    MOV pc, lr                      ; r0 리턴
```

속도 향상을 고려한 C 루프문

- 효과적인 루프문 코딩
  - 0으로 다운카운트 루프 사용
  - 디폴트로 unsigned 루프 카운터 사용
    - 반복 조건으로 i != 0을 사용
  - 루프가 한번 이상 실행되는 경우 do-while문을 사용
  - 루프 오버헤드를 줄여야 하는 중요한 루프문만 언롤링
    - 언롤링을 너무 많이 하지 않도록 함
      - 코드 사이즈 증가
      - 캐시 성능에 좋지 않은 영향
  - 배열에서 요소들의 수를 4나 8의 배수가 되도록 정렬

예제 – 업카운터 루프문

- C 코드

```
int checksum_v5(int *data)
{
    unsigned int i;
    int sum = 0;
    for(i = 0; i < 64; i++)
    {
        sum += *(data++);
    }
    return sum;
}
```

- 어셈블리 코드

```
checksum_v5
    MOV r2, r0                  ; r2 = data
    MOV r0, #0                  ; sum = 0
    MOV r1, #0                  ; i = 0
checksum_v5_loop
    LDR r3, [r2], #4            ; r3 = *(data++)
    ADD r1, r1, #1              ; i++
    CMP r1, #0x40               ; i 와 64 비교
    ADD r0, r3, r0              ; sum += r3
    BCC checksum_v5_loop        ; (I < 64)이면 loop로 분기
    MOV pc, r14                 ; sum 리턴
```

예제 – 다운카운터 루프문

- C 코드

```
int checksum_v6(int *data)
{
    unsigned int i;
    int sum = 0;
    for(i = 64; i != 0; i--)
    {
        sum += *(data++);
    }
    return sum;
}
```

- 어셈블리 코드

```
checksum_v6
    MOV r2, r0                  ; r2 = data
    MOV r0, #0                  ; sum = 0
    MOV r1, #0x40               ; i = 64
Checksum_v6_loop
    LDR r3, [r2], #4            ; r3 = *(data++)
    SUBS r1, r1, #1             ; i-- & 플래그 업데이트
    ADD r0, r3, r0              ; sum += r3
    BNE checksum_v6_loop        ; (i != 0)이면 loop로 분기
    MOV pc, r14                 ; sum리턴
```

예제 – 가변길이 루프문

C 코드

```
int checksum_v7(int *data, unsigned int N)
{
    int sum = 0;
    for(; N != 0; N--)
    {
        sum += *(data++);
    }
    return sum;
}
```

Assembly 코드

```
checksum_v7
    MOV r2, #0              ; sum = 0
    CMP r1, #0              ; N과 0을 비교
    BEQ checksum_v7_end     ; (N == 0)이면 end로 분기
checksum_v7_loop
    LDR r3, [r0], #4        ; r3 = *(data++)
    SUBS r1, r1, #1         ; N-- & 플래그 업데이트
    ADD r2, r3, r2          ; sum += r3
    BNE checksum_v7_loop    ; (N != 0)이면 loop로 분기
checksum_v7_end
    MOV r0, r2              ; r0 = sum
    MOV pc, r14             ; r0 리턴
```

예제 – 한번 이상 수행하는 루프문

C 코드

```
int checksum_v8(int *data, unsigned int N)
{
    int sum = 0;
    do
    {
        sum += *(data++);
    } while(--N != 0);
    return sum;
}
```

Assembly 코드

```
checksum_v8
    MOV r2, #0              ; sum = 0
checksum_v8_loop
    LDR r3, [r0], #4        ; r3 = *(data++)
    SUBS r1, r1, #1         ; N-- & 플래그 업데이트
    ADD r2, r3, r2          ; sum += r3
    BNE checksum_v8_loop    ; (N != 0)이면 loop로 분기
    MOV r0, r2              ; r0 = sum
    MOV pc, r14             ; r0 리턴
```

예제 - 루프 언롤링

C 코드

```
int checksum_v9(int *data, unsigned int N)
{
    int sum = 0;
    do
    {
      sum += *(data++);
      sum += *(data++);
      sum += *(data++);
      sum += *(data++);
      N -= 4;
    } while (N != 0);
    return sum;
}
```

Assembly 코드

```
checksum_v9
        MOV r2, #0                  ; sum = 0
checksum_v9_loop
        LDR r3, [r0], #4            ; r3 = *(data++)
        SUBS r1, r1, #4             ; N -= 4 & 플래그 업데이트
        ADD r2, r3, r2              ; sum += r3
        LDR r3, [r0], #4            ; r3 = *(data++)
        ADD r2, r3, r2              ; sum += r3
        LDR r3, [r0], #4            ; r3 = *(data++)
        ADD r2, r3, r2              ; sum += r3
        LDR r3, [r0], #4            ; r3 = *(data++)
        ADD r2, r3, r2              ; sum += r3
        BNE checksum_v9_loop        ; (N != 0)이면 loop로 분기
        MOV r0, r2                  ; r0 = sum
        MOV pc, r14                 ; r0 리턴
```

예제 – 배수가 아닌 루프의 언롤링

C 코드

```
int checksum_v10(int *data, unsigned int N)
{
        unsigned int i;
        int sum = 0;
        for(i = N/4; i != 0; i--)
        {
            sum += *(data++);
            sum += *(data++);
            sum += *(data++);
            sum += *(data++);
        }
        for(i = N&3; i != 0; i--)
        {
            sum += *(data++);
        }
        return sum;
}
```

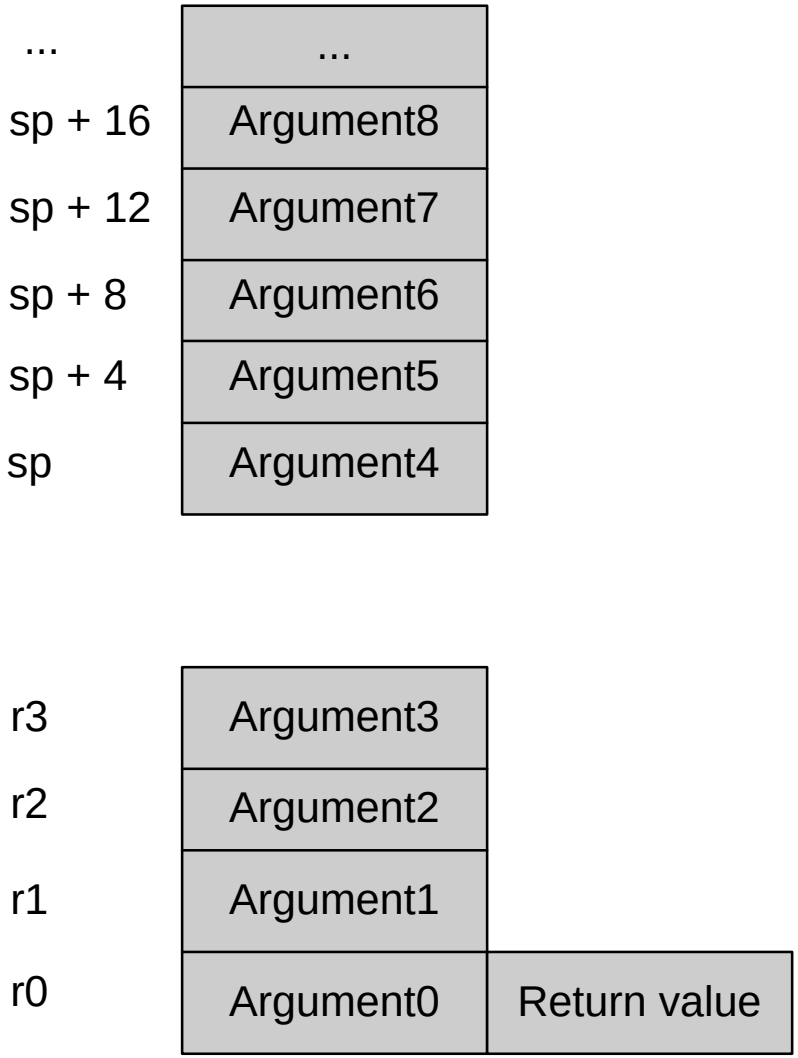
레지스터 매핑

- 효율적인 레지스터 할당
  - 함수의 내부 루틴에서 사용하는 지역 변수의 수를 12개로 제한
    - 이론적으로 14개가 가능
    - r12 및 임시 작업용 레지스터들로 인하여 최대 12개의 지역 변수만 사용하는 것을 권장
  - 어떤 변수가 중요한지 컴파일러에게 알릴 것을 권장
    - 하지만 register 키워드는 사용을 피함

C 컴파일러에서의 레지스터 사용

레지스터 번호	레지스터 이름	레지스터 사용
r0	a1	매개 변수 레지스터. 이들은 함수 호출 시 처음 4개의 함수 인자와 함수 복귀 시에 리턴 값을 가질 수 있다. 어떤 함수는 이 레지스터들을 중복해서 사용할 수 있고 함수 내에 범용 스크래치 레지스터로 사용할 수 있다.
r1	a2	
r2	a3	
r3	a4	
r4	v1	범용 변수 레지스터. 함수는 이 레지스터들의 callee 값을 유지한다.
r5	v2	
r6	v3	
r7	v4	
r8	v5	
r9	v6 sb	범용 변수 레지스터. RWPI로 컴파일할 때를 제외하고 함수는 이 레지스터의 callee 값을 유지한다. r9은 정적 베이스 주소를 저장하고 있으며 이것은 읽고 쓸 수 있는 주소값이다.
r10	v7 sl	범용 변수 레지스터. 스택 리미트를 체크하며 컴파일할 때를 제외하고 함수는 이 레지스터의 callee 값을 유지한다. r10은 스택 리미트 주소를 저장하고 있다.
r11	v8 fp	범용 변수 레지스터. 프레임 포인터를 사용하여 컴파일할 때를 제외하고 함수는 이 레지스터의 callee 값을 유지한다. 이전 버전의 armcc에서만 프레임 포인터를 사용한다.
r12	ip	함수가 임시로 사용할 수 있는 스크래치 레지스터. 이것은 함수 베니어나 다른 내부 프로시저 호출 요구사항을 위한 범용 스크래치 레지스터로 사용된다.
r13	sp	Full descending 스택을 가리키는 포인터
r14	lr	링크 레지스터. 함수 호출 시에 복귀할 주소를 저장한다.
r15	pc	프로그램 카운터

ATPCS (ARM-Thumb Procedure Call Standard) 인자 패싱



서브루틴 호출 시 매개변수 처리

- 효과적인 함수 호출
  - 함수 인자 수를 4로 제한
    - 많은 인자의 경우 구조체의 포인터를 이용하여 인자 전달
  - 작은 함수들은 같은 소스 파일 안에 이들을 호출하는 함수 앞에 정의
    - 함수 호출 최적화
    - 작은 함수를 인라인화
  - 크리티컬한 함수는 inline 키워드 사용

서브루틴 호출 시 매개변수 처리

C 코드

```
char *queue_bytes_v1(char *Q_start,
        char *Q_end, char *Q_ptr, char *data,
        unsigned int N)
{
        do
        {
            *(Q_ptr++) = *(data++);
            if(Q_ptr == Q_end)
            {
                Q_ptr = Q_start;
            }
        } while(--N);
        return Q_ptr;
}
```

Assembly 코드

```
queue_bytes_v1
        STR r14, [r13, #-4]!        ; 스택에 lr을 저장
        LDR r12, [r13, #4]          ; r12 = N
queue_v1_loop
        LDRB r14, [r3], #1          ; r14 = *(data++)
        STRB r14, [r2], #1          ; *(Q_ptr++) = r14
        CMP r2, r1                  ; (Q_ptr == Q_end) 이면
        MOVEQ r2, r0                ; {Q_ptr = Q_start;}
        SUBS r12, r12, #1           ; --N & 플래그 업데이트
        BNE queue_v1_loop           ; r0 = Q_ptr
        LDR pc, [r13], #4           ; r0 리턴
```

서브루틴 호출 시 매개변수 처리

C 코드

```
typedef struct{
        char *Q_start;
        char *Q_end;
        char *Q_ptr;
} Queue;
void queue_bytes_v2(Queue *queue,
        char *data, unsigned int N)
{
        char *Q_ptr = queue->Q_ptr;
        char *Q_end = queue->Q_end;
        do
        {
            *(Q_ptr++) = *(data++);
            if(Q_ptr == Q_end)
            {
                Q_ptr = queue->Q_start;
            }
        } while(--N);
        queue->Q_ptr = Q_ptr;
}
```

Assembly 코드

```
queue_bytes_v2
        STR r14, [r13, #-4]!        ; 스택에 lr을 저장
        LDR r3, [r0, #8]            ; r3 = queue->Q_ptr
        LDR r14, [r0, #4]           ; r14 = queue->Q_end
queue_v2_loop
        LDRB r12, [r1], #1          ; r12 = *(data++)
        STRB r12, [r3], #1          ; *(Q_ptr++) = r12
        CMP r3, r14                 ; (Q_ptr == Q_end) 이면
        LDREQ r3, [r0, #0]          ; {Q_ptr = queue->Q_start;}
        SUBS r2, r2, #1             ; --N & 플래그 업데이트
        BNE queue_v2_loop           ; (N != 0)이면 loop로 분기
        STR r3, [r0, #8]            ; queue->Q_ptr = r3
        LDR pc, [r13], #4           ; r0 리턴
```

서브루틴 호출 시 매개변수 처리

C 코드

```
unsigned int nybble_to_hex(unsigned int d)
{
        if(d < 10)
        {
            return d + '0';
        }
        return d – 10 + 'A';
}
void uint_to_hex(char *out, unsigned int in)
{
        unsigned int i;
        for(i = 8; i != 0; i--)
        {
            in = (in << 4) | (in >> 28);
            *(out++) = (char)nybble_to_hex(in & 15);
        }
}
```

Assembly 코드

```
uint_to_hex
        MOV r3, #8                  ; i = 8
uint_to_hex_loop
        MOV r1, r1, ROR #28         ; in = (in << 4) || (in >> 28)
        AND r2, r1, #0xF            ; r2 = in & 15
        CMP r2, #0xA                ; (r2 >= 10) 이면
        ADDCS r2, r2, #0x37         ; r2 += 'A' – 10
        ADDCC r2, r2, #0x30         ; 아니면 r2 += '0'
        STRB r2, [r0], #1           ; i-- & 플래그 업데이트
        BNE uint_to_hex_loop        ; (i != 0)이면 loop로 분기
        MOV pc, r14                 ; 리턴
```

포인터 앨리어싱

- 포인터 앨리어싱 피하기
  - 2개의 포인터가 같은 주소를 가리키고 있는 경우
  - 한 포인터를 이용하여 값을 기록할 경우 다른 포인터의 읽은 값에 영향을 주게 됨
  - 메모리를 제어하는 공통 표현법을 없애기 위해 컴파일러에 의존하지 않도록 함
    - 그 표현 방법이 저장하고 있는 새로운 지역 변수를 이용
  - 지역 변수의 주소 값을 사용하는 것을 피하여야 함

포인터 앨리어싱

C 코드

```
void timers_v1(int *timer1, int *timer2,
               int *step)
{
    *timer1 += *step;
    *timer2 += *step;
}
```

Assembly 코드

```
timers_v1
    LDR r3, [r0, #0]      ; r3 = *timer1
    LDR r12, [r2, #0]     ; r12 = *step;
    ADD r3, r3, r12       ; r3 += r12
    STR r3, [r0, #0]      ; *timer1 = r3
    LDR r0, [r1, #0]      ; r0 = *timer2
    LDR r2, [r2, #0]      ; r2 = *step;
    ADD r0, r0, r2        ; r0 += r2
    STR r0, [r1, #0]      ; *timer2 = r0
    MOV pc, r14           ; 리턴
```

포인터 앨리어싱

C 코드

```
typedef struct {int step;} State;
Typedef struct{int timer1, timer2;} Timers
void timers_v2(State *state, Timers *timers)
{
    int step = state->step;
    timers->timer1 += step;
    timers->timer2 += step;
}
```

포인터 앨리어싱

- 지역변수의 주소
  + 포인터 앨리어싱 발생

- C 코드

```
int checksum_next_packet()
{
    int *data;
    int N, sum = 0;
    data = get_netxt_packet(&N);
    do
    {
        sum += *(data++);
    } while(--N);
    return sum;
}
```

- Assembly 코드

```
checksum_next_packet
    STMFD r13!, {r4, r14}      ; 스택에 r4, lr을 저장
    SUB r13, r13, #8           ; 스택에 두 변수 공간을 마련
    ADD r0, r13, #4            ; r0 = &N, 스택에 저장된 N
    MOV r4, #0                 ; sum = 0
    BL get_next_packet         ; r0 = data
checksum_loop
    LDR r1, [r0], #4           ; r1 = *(data++)
    ADD r4, r1, r4             ; sum += r1
    LDR r1, [r13, #4]          ; r1 = N (스택에서 읽음)
    SUBS r1, r1, #1            ; r1-- & 플래그 업데이트
    STR r1, [r13, #4]          ; N = r1 (스택에 씀)
    BNE checksum_loop          ; (N != 0)이면 loop로 분기
    MOV r0, r4                 ; r0 = sum
    ADD r13, r13, #8           ; 스택에 저장된 변수를 삭제
    LDMFD r13!, {r4, pc}       ; r0 리턴
```

구조체

- 구조체의 레이아웃
  - 성능과 코드 집적도에 영향
  - 효율적인 구조체 정렬
    - 요소의 크기가 점점 증가하는 순으로 구조체를 레이아웃
    - 너무 큰 구조체 사용하지 않음
      - 계층적 구조 사용
    - 이식성을 위하여 API 구조체에 패딩 추가
    - API 안에 enum 형 사용하지 않음
      - 컴파일러에 따라 크기가 다름

전송 크기	명령어	바이트 주소
1바이트	LDRB, LDRSB, STRB	바이트 주소 정렬
2바이트	LDRH, LDRSH, STRH	2의 배수
4바이트	LDR, STR	4의 배수
8바이트	LDRD, STRD	8의 배수

구조체

```
struct{
        char a;
        int b;
        char c;
        short d;
}
```

	+3	+2	+1	+0
+0	pad	pad	pad	a
+4	b[31,24]	b[23,16]	b[15,8]	b[7,0]
+8	d[15,8]	d[7,0]	pad	c

```
struct{
        char a;
        char c;
        short d;
        int b;
}
```

	+3	+2	+1	+0
+0	d[15,8]	d[7,0]	c	a
+4	b[31,24]	b[23,16]	b[15,8]	b[7,0]

```
_packed struct{
        char a;
        int b;
        char c;
        short d;
}
```

	+3	+2	+1	+0
+0	b[23,16]	b[15,8]	b[7,0]	a
+4	d[15,8]	d[7,0]	c	b[31,24]

비트 필드

- 비트필드
  - 가능하면 비트필드를 사용하지 않도록 함
    - 컴파일러마다 배치가 다름
    - 비트필드는 구조체의 포인터를 사용하여 접근하므로 효율성도 좋지 않음
    - #define 또는 enum 사용
  - 논리연산 AND, OR, EOR와 마스크 값을 사용하여 정수형에 비트 필드를 테스트, 토클, 세트하도록 함
    - stage |= STAGEA : state A 활성화
    - stage &= ~STAGEB : state B 비활성화
    - stage ^= STAGEC : stage C 토글

비트필드

C 코드

```
void dostageA();
void dostageB();
void dostageC();
typedef struct{
    unsigned int stageA : 1;
    unsigned int stageB : 1;
    unsigned int stageC : 1;
} Stages_v1;
void dostages_v1(Stages_v1 *stages)
{
    if(stages->stageA) dostageA();
    if(stages->stageB) dostageB();
    if(stages->stageC) dostageC();
}
```

Assembly 코드

```
dostages_v1
    STMFD r13!, {r4, r14}       ; r4, lr을 스택에 집어넣음
    MOV r4, r0                  ; stages를 r4로 이동
    LDR r0, [r0, #0]            ; r0 = stage bitfiled
    TST r0, #1                  ; (statges->stageA) 이면
    BLNE dostageA               ; dostageA();
    LDR r0, [r4, #0]            ; r0 = stages bitfield
    MOV r0, r0, LSL #30         ; 비트 1을 비트 31로 시프트
    CMP r0, #0                  ; (bit31)이면
    BLLT dostageB               ; dostageB();
    LDR r0, [r4, #0]            ; r0 = stages bitfield
    MOV r0, r0, LSL #29         ; 비트 2를 비트 31로 시프트
    CMP r0, #0                  ; (!bit31)이면
    LDMLTFD r13!, {r4, r14}     ; 리턴
    BLT dostageC                ; dostageC();
    LDMFD r13!, {r4, pc}        ; 리턴
```

비트필드

C 코드

```
void dostageA();
void dostageB();
void dostageC();
typedef unsigned long Stages_v2;
#define STAGEA (1ul << 0);
#define STAGEB (1ul << 1);
#define STAGEC (1ul << 2)
void dostages_v2(Stages_v2 *stages_v2)
{
        Stages_v2 stages = *stages_v2
        if(stages & STAGEA) dostageA();
        if(stages & STAGEB) dostageB();
        if(stages & STAGEC) dostageC();
}
```

Assembly 코드

```
dostages_v1
        STMFD r13!, {r4, r14}       ; r4, lr을 스택에 집어넣음
        LDR r4, [r0, #0]            ; stages = *stages_v2
        TST r4, #1                  ; (statges & STAGEA) 이면
        BLNE dostageA               ; dostageA();
        TST r4. #2                  ; (stage & STAGEB) 이면
        BLNE dostageB               ; dostageB();
        TST r4, #4                  ; (!(stage & STAGEC)) 이면
        LDMNEFD r13!, {r4, r14}     ; 리턴
        BNE dostageC                ; dostageC();
        LDMFD r13!, {r4, pc}        ; 리턴
```

비정렬 데이터와 엔디안

- 엔디안과 정렬
  - 가능하다면 비정렬 데이터는 사용하지 않음
  - 어떤 바이트 정렬이 될 수 있는 데이터를 위해서 char* 형 사용
  - __packed 지시어
    - 포팅 시 유용
    - 성능에는 좋지 않는 영향
  - 포인터 정렬과 프로세서 엔디안에 따라 함수를 사용하여 구현
    - 비정렬 구조체를 빠르게 엑세스

비정렬 데이터와 엔디안

C 코드

```
int readint(__packed int *data)
{
        return *data;
}
```

Assembly 코드

```
readint
        BIC r3, r0, #3          ; r3 = data & 0xFFFFFFFC
        AND r0, r0, #3          ; r0 = data & 0x00000003
        MOV r0, r0, LSL #3      ; r0 = 데이터 워드의 비트 오프셋
        LDMIA r3, {r3, r12}     ; r3, r12 = r3에서 읽은 8바이트값
        MOV r3, r3, LSR r0      ; 이들 세 명령어
        RSB r0, r0, #0x20       ; 64 비트값 r12, r3을 시프트
        ORR r0, r3, r12, LSL r0 ; r0 비트만큼 오른쪽으로 시프트
        MOV pc, r14             ; r0 리턴
```

리틀 엔디안 설정

명령어	폭 (비트)	b31 .. b24	b23 .. b16	b15 .. b8	b7 .. b0
LDRB	8	0	0	0	B(A)
LDRSB	8	S(A)	S(A)	S(A)	B(A)
STRH	8	X	X	X	B(A)
LDRH	16	0	0	B(A+1)	B(A)
LDRSH	16	S(A+1)	S(A+1)	B(A+1)	B(A)
STRH	16	X	X	B(A+1)	B(A)
LDR / STR	32	B(A+3)	B(A+2)	B(A+1)	B(A)

빅 엔디안 설정

명령어	폭 (비트)	b31 .. b24	b23 .. b16	b15 .. b8	b7 .. b0
LDRB	8	0	0	0	B(A)
LDRSB	8	S(A)	S(A)	S(A)	B(A)
STRH	8	X	X	X	B(A)
LDRH	16	0	0	B(A)	B(A+1)
LDRSH	16	S(A)	S(A)	B(A)	B(A+1)
STRH	16	X	X	B(A)	B(A+1)
LDR / STR	32	B(A)	B(A+1)	B(A+2)	B(A+3)

나눗셈

- 나눗셈
  - 가능하면 나눗셈의 사용을 피함
    - 원형 버퍼 처리를 위해 사용하지 않음
  - 나눗셈 루틴이 몫과 나머지를 함께 생성한다는 사실 이용
  - 같은 분모 d로 반복해서 나누기를 할 경우 $s = (2^k - 1) / d$를 이용하여 계산

예제 - 원형 버퍼 구현

■ 나눗셈을 이용한 코드

```
offset = (offset + increment) % buffer_size
```

■ 나눗셈을 이용하지 않은 코드

```
offset += increment;
if(offset >= buffer_size)
{
        offset -= buffer_size;
}
```

예제 – 스크린 버퍼의 offset 버퍼 (x, y) 위치

C 코드

```
typedef struct{
        int x;
        int y;
} point;
point getxy_v1(unsigned int offset,
                    unsigned int bytes_per_line)
{
        point p;
        p.y = offset / bytes_per_line;
        p.x = offset – py * bytes_per_line;
        return p;
}
```

예제 – 스크린 버퍼의 offset 버퍼 (x, y) 위치

C 코드

```
typedef struct{
        int x;
        int y;
} point;
point getxy_v2(unsigned int offset,
                unsigned int bytes_per_line)
{
        point p;
        p.x = offset % bytes_per_line;
        p.y = offset / bytes_per_line;
        return p;
}
```

Assembly 코드

```
getxy_v2
        STMFD r13!, {r4, r14}     ; r4와 lr을 스택에 집어넣음
        MOV r4, r0                ; p를 r4로 이동
        MOV r0, r2                ; r0 = bytes_per_line
        BL __rt_udiv              ; (r0, r1) = (r1 / r0, r1 % r0)
        STR r0, [r4, #4]          ; p.y = offset / bytes_per_line
        STR r1, [r4, #0]          ; p.y = offset % bytes_per_line
        LDMFD r13!, [r4, pc]      ; 리턴
```

예제 – Scaling

C 코드

```
void scale( unsigned int *dest,
            unsigned int *src,
            unsigned int d,
           unsigned int N)
{
    unsigned int s = 0xFFFFFFFFu / d;
    do
    {
       unsigned int n, q, r;
       n = *(src++);
       q = (unsigned int)(((unsigned long long)n * s) >> 32);
       r = n – q * d;
       if(r >= d)
       {
          q++;
       }
       *(dest++) = q;
    } while(--N);
}
```

부동소수점

- 부동 소수점
  - ARM 프로세서 제품군 대부분 하드웨어 부동소수점 지원하지 않음
  - C 라이브러리를 이용한 소프트웨어 부동소수점 연산 지원
    - 정수 알고리즘보다 훨씬 느림
  - 빠른 실행과 분수 값이 필요할 경우
    - 고정소수점 알고리즘
    - 블록소수점 알고리즘

인라인 함수 및 인라인 어셈블리

- 인라인 함수와 어셈블리
  - C 컴파일러가 지원하지 않는 새로운 함수나 원칙을 정의하기 위해 인라인 함수 사용
    - #define 매크로보다 인라인을 사용이 더 좋음
      - #define 매크로는 함수의 인자와 리턴값 데이터 형을 체크하지 않음
  - C 컴파일러가 지원하지 않는 ARM 명령어를 엑세스하기 위해 인라인 어셈블리 사용

예제 – Saturating Multiply Double Accumulate Primitive

```
__inline int qmac_v1(int a, int x, int y)
{
    int i;
    i = x* y;
    if(i >= 0){
        i = 2 * i;
        if(i < 0) i = 0x7FFFFFFF;
        if(a + i < a) return 0x7FFFFFFF;
        return a + i;
    }
    if(a + 2 * i > a)  return 0x80000000;
    return a + 2 * i;
}
```

```
int sat_correlate_v1(short *x, short *y, unsigned int N
{
    int a = 0;
    do{
        a = qmac_v1(a, *(x++), *(y++));
    } while(--N);
}
```

예제 – Saturating Multiply Double Accumulate Primitive

```
__inline int qmac_v2(int a, int x, int y)
{
        int i;
        const int mask = 0x80000000;
        i = x * y;
#ifdef __ARMCC_VERSION
        asm{
            ADDS i, i, i
            EORVS i, mask, i, ASR 31
            ADDS a, a, i
            EORVS a, mask, a, ASR 31
        }
#endif
#ifdef __GNUC__
        asm("ADDS %0, %1, %2":" = r" (i) : "r" (i), "r" (i) : "cc");
        asm("EORVS %0, %1, %2, ASR #31" : " = r" (i) : "r" (mask), "r" (i) : "cc");
        asm("ADDS %0, %1, %2" : " = r" (a) : "r" (a), "r" (i) : "cc");
        asm("EORVS %0, %1, %2, ASR #31" : " = r" (a) : "r" (mask), "r" (a) : "cc");
        return a;
}
```

예제 – Saturating Multiply Double Accumulate Primitive

```
__inline int qmac_v3(int a, int x, int y)
{
    int i;
    __asm
    {
        SMULBB i, x, y
        QDADD a, a, I
    }
    return a;
}
```

이식성 문제

- 이식성 문제 요약
  - char 형
    - ARM에서는 char형은 unsigned
  - int 형
    - 오래된 아키텍처는 16-bit int 사용
  - 비정렬 데이터 포인터
    - ARMv5E까지 ARM 아키텍처는 비정렬 포인터를 지원하지 않음
  - 엔디안 가정
    - C 코드는 메모리 시스템의 엔디안에 대한 가정을 함
  - 함수 프로토타입
    - ARM 컴파일러는 매개변수를 narrow 캐스트를 통하여 형을 변경하여 보냄
    - ANSI 프로토타입을 사용할 것을 권장
  - 비트필드 사용
    - 엔디안 의존적
  - 열거형
    - enum의 할당된 비트수가 다를 수 있음
  - 인라인 어셈블리
    - 포팅이 어려워질 수 있음
  - volatile 키워드
    - ARM 메모리 매핑된 주변 장치의 형 정의를 할 때 volatile 키워드 사용

Summary

- 지역 변수, 함수 인자, 리턴 값
  - signed나 unsigned int 형 사용
- 루프문의 가장 효과적인 형식
  - 다운 카운트, do-while 루프문
- 루프 언롤링
  - 루프문의 오버헤드를 줄일 수 있음
- 포인터 앨리어싱
  - 반복적인 메모리 엑세스를 하지 않도록 최적하는데 방해
- 함수 인자
  - 4개로 제한

Summary

- 구조체
  - 요소들의 크기가 증가하는 순서대로 배치
- 비트필드를 사용하지 않음
  - 마스크와 논리 연산 이용
- 나눗셈
  - 그에 상응하는 곱셈으로 바꾸어 사용
- 비정렬 데이터
  - 사용하지 않음
  - 데이터가 정렬되어 있지 않으면 char* 포인터 형 이용
- 인라인 어셈블러
  - C 컴파일러가 지원하지 않는 명령어를 엑세스 하거나 최적화를 위해서 사용

Writing and Optimizing ARM Assembly Code

Young Hoi Heo

yhheo@enc.hanyang.ac.kr

Outline

- 어셈블리 코드 작성
- 최적화 도구
- 명령어의 스케줄링
- 적절한 레지스터의 선택
- 조건 분기 명령어의 활용
- 최적의 루프문 구현
- 효율적인 조건 분기
- 비정렬 데이터의 처리

어셈블리 코드 작성 [1/3]

C 함수

```
#include <stdio.h>
int square( int i);
int main(void)
{
    int i;
        for( i = 0; i<10; i++)
        {
            printf("Square of %d is %d₩n", i, square(i) );
        }
}

int square( int i )
{
    return i * i ;
}
```

어셈블리 함수

```
    AREA | .text |, CODE, READONLY

    EXPORT square

    ; int square( int i )
square                  ; code lable
    MUL r1, r0, r0      ; r1 = r0 * r0
    MOV r0, r1          ; r0 = r1
    MOV pc, lr          ; r0을 리턴
    END

; ARM 코드를 C 파일에 컴파일 하는 경우

    BX lr               ; r0을 리턴
;  Thumb 코드로 컴파일 하는 경우
;  BX 명령어 (ARMv4T 이상)
```

어셈블리 코드 작성 [2/3]

■ 어셈블리 루틴에서 서브루틴을 호출

```
AREA | .text |, CODE, READONLY

    EXPORT main

    IMPORT |Lib$$Request$$armlib |, WEAK
    IMPORT __main          ; C 라이브러리 시작
    IMPORT printf          ; stdout 으로 출력

i   RN 4
    ; int main (void)
main
    STMFD   sp!, { i, lr }
    MOV     i, #0
loop
    ADR     r0, print_string
    MOV     r1, i
    MUL     r2, i, i
    BL      printf
    ADD     i, i, #1
    CMP     i, #10
    BLT     loop
    LDMFD   sp!, { i, pc }

print_ string
    DCB     "Square of %d₩n", 0
    END
```

Thumb 코드

```
    LDMFD   sp!, { i, lr }
    BX      lr
```

어셈블리 코드 작성 [3/3]

■ 어떤 정수 값들을 합하는 sumof 라는 함수정의

```
#include <stdio.h>
/* N은 덧셈을 할 값의 수를 의미한다. */
int sumof( int N, ...);
int main(void)
{
    printf("Empty sum = %d₩n", sumof( 0 ) );
    printf("1 = %d₩n", sumof( 1, 1 ) );
    printf("1 + 2 = %d₩n", sumof( 2, 1, 2 ) );
    printf("1 + 2 + 3 = %d₩n", sumof(3, 1, 2, 3 ) );
    printf("1 + 2 + 3 + 4 = %d₩n", sumof(4, 1,2,3,4 ) );
    printf("1 + 2 + 3 + 4 + 5 = %d₩n", sumof(5,1,2,3,4,5));
    printf("1 + 2 + 3 + 4 + 5 + 6 = %d₩n", sumof(6,1,2,3,4,5,6) );

}
```

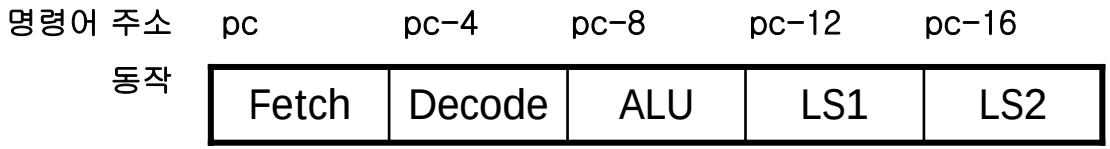
```
    AREA | .text |, CODE, READONLY
    EXPORT sumof
N     RN 0                  ; 합할 요소들의 수
Sum RN 1                    ; 현재의 합
    ; int sumof( int N, ... )
sumof                       ; code lable
    SUBS    N, N, #1        ; 하나의 요소를 가짐
    MOVLT   sum, #0         ; 합할 요소가 없음
    SUBS    N, N, #1        ; 두 개의 요소들을 가짐
    ADDGE   sum, sum, r2
    SUB     N, N, #1        ; 세 개의 요소들을 가짐
    ADDGE   sum, sum, r3
    MOV     r2, sp          ; 스택의 맨 위
loop
    SUBS    N, N, #1        ; 또 다른 요소를 가짐
    LDMGEFD r2!, {r3}       ; 스택으로부터 데이터를 읽음
    ADDGE   sum, sum, r3
    BGE     loop
    MOV     r0, sum
    MOV     pc, lr          ; r0를 리턴
END
```

최적화 도구

- 최적화 과정의 첫 번째 과정
  - 크리티컬한 루틴이 어떤 것인지 찾아 내어 현재 성능을 측정해 보는 것
- Profiler
  - 각 루틴에서 소요되는 시간과 처리 사이클을 측정
  - 컴파일 중 또는 실행 중 일정함수나 Code block의 성능 결과
- Cycle counter
  - 특정 루틴에서 소요되는 사이클의 수를 측정
  - 최적화 전후로 하여 주어진 서브루틴을 벤치마킹 하는데 사용
- ADS1.1 Debugger 에서 사용되는 ARMulator
  - Profiling, Cycle counting 기능을 제공

명령어의 스케줄링 [1/7]

- ARM9TDMI 프로세서는 5단계의 동작을 병렬로 수행
  - Fetch – pc가 가리키는 메모리 주소로부터 명령어를 읽는다.
  - Decode – 이전 사이클에서 읽어 들인 명령어를 해독한다.
  - ALU – 이전 사이클에서 해독한 명령어를 실행한다.
  - LS1 – 로드-스토어 명령어에 의해 규정된 데이터를 읽거나 저장한다.
  - LS2 – 바이트 또는 하프워드 로드 명령어에 의해 로드 된 데이터를 제로 또는 부호 확장한다.



- 현재의 명령어는 그 이전 명령어가 완전히 처리되어야만 다음 단계로 이동
- 만약 명령어가 사용할 수 없는 이전 명령어의 결과를 필요로 한다면, 그 프로세서는 정지 상태

  ☞ 파이프라인 해저드(hazard) 또는 파이프라인 인터로크(interlock)

명령어의 스케줄링 [2/7]

- Interlock 가 없는 경우

```
ADD     r0, r0, r1
ADD     r0, r0, r2
```

→ 2 사이클 소요

- Interlock 가 발생한 경우

```
LDR     r1, [r2, #4]
ADD     r0, r0, r1
```

→ 3 사이클 소요

첫 번째 사이클 – ALU는 r2 + 4의 주소를 계산하면서 ADD 명령어를 해독

두 번째 사이클 – 로드 명령어가 아직 r1의 값을 읽어 들이지 못했기 때문에 ADD 수행될 수 없다.

명령어의 스케줄링 [3/7]

- 지연된 로드 사용으로 인한 한 사이클의 인터로크

```
LDRB    r1, [r2, #1]
ADD     r0, r0, r2
EOR     r0, r0, r1
```

→ 4 사이클 소요

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	EOR	ADD	LDRB		
Cycle 2	...	EOR	ADD	LDRB	
Cycle 3		...	EOR	ADD	LDRB
Cycle 4		...	EOR	-	ADD

명령어의 스케줄링 [4/7]

- 분기 명령어를 처리하는데 3 사이클 소요

```
        MOV     r1, #1
        B       case1
        AND     r0, r0, r1
        EOR     r2, r2, r3
        . . .
case1
        SUB     r0, r0, r1
```

→ 5 사이클 소요

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	AND	B	MOV	...	
Cycle 2	EOR	AND	B	MOV	...
Cycle 3	SUB	-	-	B	MOV
Cycle 4	...	SUB	-	-	B
Cycle 5		...	SUB	-	-

분기로 인한 파이프라인 플러시

명령어의 스케줄링 [5/7]

- 로드 명령어의 스케줄링
  - 메모리를 많이 사용하는 태스크 예제
  - 함수 str_tolower는 0으로 끝나는 문자열을 in에서 out으로 복사 (소문자로 변경)

```
void str_tolower( char *out, char *in )
{
    unsigned int c;
    do
    {
        c = *( in++ );
        if ( c >= 'A' && c <= 'Z' )
        {
            c = c + ( 'a' – 'A' );
        }
        *(out++) = (char) c;
    } while (c);
}
```

```
str_tolower
    LDRB    r2, [r1], #1        ; c = *( in++ )
    SUB     r3, r2, #0x41       ; r3 = c – 'A'
    CMP     r3, #0x19           ; ( c <= 'Z' – 'A' ) 이면
    ADDLS   r2, r2, #0x20       ;      c += 'a' – 'A'
    STRB    r2, [r0], #1        ; *(out++) = (char)c
    CMP     r2, #0              ; ( c != 0 ) 이면
    BNE     str_tolower         ;     str_tolower 로 분기
    MOV     pc, r14             ; 리턴
```

+ LDRB 명령어 바로 뒤에서 c 사용으로, 2 사이클 동안 정지

→ 어셈블리를 사용함으로써, 2사이클을 피하는 알고리즘 구조
프리로딩(preloading) 및 언롤링(unrolling)에 의한 로드 스케줄링 방법

명령어의 스케줄링 [6/7]

프리로딩에 의한 로드 스케줄링

```
out         RN 0                ; 출력 스트링을 가리킴
in          RN 1                ; 입력 스트링을 가리킴
c           RN 2                ; 로드된 문자
t           RN 3                ; 임시 레지스터
    ; void str_tolower_preload ( char *out, char *in )
    str_tolower_preload
    LDRB    c, [in], #1         ; c = * ( in++ )
loop
    SUB     t, c, #'A'          ; t = c – 'A'
    CMP     t, #'Z' – 'A'       ; ( t <= 'Z' – 'A' ) 이면
    ADDLS   c, c, #'a' – 'A'    ;    c += 'a' – 'A' ;
    STRB    c, [out], #1        ; *( out++ ) = ( char )c;
    TEQ     c, #0               ; c == 0 인지를 테스트
    LDRNEB  c, [in], #1         ; ( c != 0 ) 이면  { c = * in++ ;
    BNE     loop                ;                    goto loop;   }
    MOV     pc, lr              ; 리턴
```

루프문이 문자당 11사이클에서 9사이클로 감소, 속도 1.22배 향상

명령어의 스케줄링 [7/7]

- 언롤링에 의한 로드 스케줄링
  - 이 방법은 루프문을 언롤링한 다음 인터리빙
  - Ex) i, i+1, i+2 씩 인터리빙하여 루프 반복을 수행

```
loop_next3
 LDRB       ca0, [in], #1          ; ca0 = *in++;
 LDRB       ca1, [in], #1          ; ca1 = *in++;
 LDRB       ca2, [in], #1          ; ca2 = *in++;
 ...
 ...
```

문자당 7사이클을 소요 , 속도 1.57배 향상

→ 코드의 길이 두 배 이상 증가

마지막 액세스 하다가 data abort가 발생할 수 있다.

적절한 레지스터의 선택

- 레지스터에 변수 할당
  - 변수를 표현하기 위해 r0, r1 과 같은 레지스터 명명법을 사용하는 것보다 변수 이름 사용하는 것이 좋다.
    → 변수에 사용되는 레지스터를 쉽게 변경
    → 최적화 코드의 가독성, 확실성을 높여줌
- 지역 변수로 14개 이상을 사용하는 경우
  - 스택에 저장
  - 내부 루프가 성능에 가장 큰 영향 – ATPCS는 알고리즘의 내부 루프에서 바깥쪽으로 작업
- 레지스터의 최대 활용
  - 메모리보다 레지스터 액세스가 효율적
  - 몇몇 32비트보다 작은 값들을 하나의 32비트 레지스터에 저장하여 코드 사이즈를 줄이고 성능을 향상시킬 수 있다.

조건 분기 명령어의 활용

- 대부분의 if 문은 조건부 실행으로 구현 가능
  - 조건 분기를 사용하는 것보다 효율

- If 문을 그 자체가 조건문이 되는 비교 명령어를 사용해서 유사한 조건들을 논리 AND와 OR 연산을 통해 구현 가능

논리 관계 변환

변경된 표현	동일한 표현
!(a && b)	(!a) || (!b)
!(a || b)	(!a) && (!b)

최적의 루프문 구현

- ARM 루프 카운터 구현
  - 플래그를 설정하기 위한 뺄셈 명령어, 조건 분기 명령어의 2가지 명령어를 필요
- 루프의 성능 향상을 위해 루프를 언롤링
  - 단, 지나친 언롤링은 캐시의 성능을 저하
  - 적은 수의 반복문은 비효율적
- 중복된 루프문은 하나의 루프 카운터 레지스터만 필요
  - 다른 용도로 레지스터들을 사용할 수 있으므로 효율성 향상
- 이외의 카운트 루프
  - 음수 인덱스 루프를 구현
  - 로그 인덱스 루프를 구현

효율적인 조건 분기

- 0 =< x < N 범위에서의 switch
  - 함수 주소 테이블을 사용하는 것
    -> x에 표기된 테이블에서 pc 값을 읽어옴

- 일반적인 값 x를 가질 때의 switch
  - 해시 함수 적용

비정렬 데이터의 처리

- 성능이 문제가 안 된다면,
  - LDMB나 STMB 명령어를 사용하여 비정렬 데이터를 액세스
  - 접근방법 : 포인터 정렬이나 메모리 시스템에 정해진 엔디안에 상관없이 주어진 엔디안의 데이터를 액세스

- 성능이 문제가 된다면,
  - 각각 가능한 메모리 정렬 방식을 이용하여 최적화한 여러 개의 루틴으로 구분하여 사용
  - 루틴들을 자동으로 생성 – 어셈블러 MACRO 지시어를 사용

Optimized Primitives

Young Hoi Heo

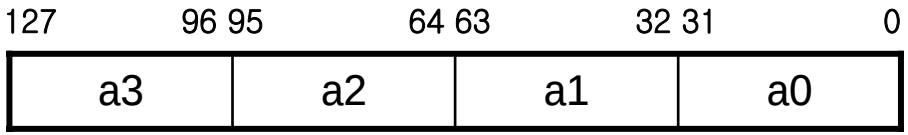
yhheo@enc.hanyang.ac.kr

Outline

- 배정밀도 정수 곱셈
- 정수 정규화와 CLZ
- 나눗셈
- 초월함수 : LOG, EXP, SIN, COS
- 엔디안 반전과 비트 연산
- 포화
- 난수 생성

배정밀도 정수 곱셈 [1/2]

- UMULL, SMULL 명령어
  - 32비트 폭 곱하기 연산가능
  - Unsigned, signed 정수들의 곱하여 64비트 또는 128비트의 결과
  - Multiword 값 – Little Endian 적용



4 개의 32 비트 값으로 구현한 128 비트

- long long 곱셈
  - 두 64 비트 값인 b 와 c를 곱하여 새로운 64 비트(signed or unsigned) long long 값인 a

배정밀도 정수 곱셈 [2/2]

- 128비트의 결과를 만들어 내는 unsigned 64 x 64 곱셈
  - 첫 번째 방법
    - ARM7M 에서 보다 빠르게 동작
    - MAC(곱셈+덧셈)명령어는 곱셈 명령어에 비해 추가로 한 사이클이 더 소요
  - 두 번째 방법
    - ARM9TDMI와 ARM9E에서 보다 빠르게 동작
    - MAC 명령어는 곱셈 명령어 만큼 빠름
    - ARM9E에서는 결과값을 사용함으로써 생기는 인터로크를 피하기 위해서 곱셈 명령어를 스케줄링

- 128비트의 결과를 만들어 내는 signed 64 x 64 곱셈
  - signed 64비트 정수는 상위 signed 32비트와 하위 unsigned 32비트로 나뉨
  - b의 상위 부분과 c의 하위 부분을 곱하려면 signed x unsigned 곱셈 명령어 필요
  - ARM에서는 위 명령어 없지만, 매크로를 사용하여 구현
  - 매크로 USMLAL
    - unsigned x signed MAC(곱셈+덧셈) 연산을 지원
    - 두 값을 signed 라고 가정 곱
    - 최상위 비트가 1이라면 signed 곱셈은 $b-2^{32}$곱, $c2^{32}$ 를 더하여 그 결과를 보정

정수 정규화와 CLZ

- 정수의 가장 중요한 비트인 leading one 이 알려진 비트 위치에 있을 때
  → 정규화

- Newton-Raphson 나눗셈 구현, 부동소수점 형식으로 변환 - 정규화 필요

- CLZ (Count Leading Zero) 명령어 – 첫 번째 유효숫자 앞에 0이 몇 개 인지 셈
  → 한 비트도 없다면 32를 리턴

- Counting Trailing Zeros

나눗셈

- ARM 코어는 나눗셈을 위한 하드웨어를 지원하지 않음

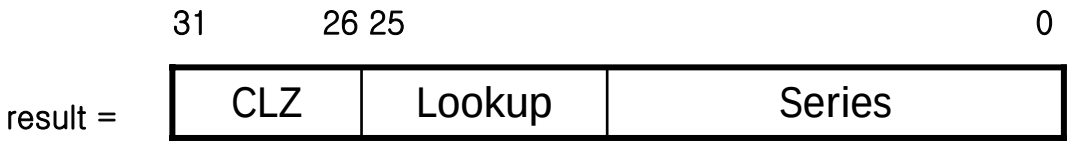
- 두 수를 나누기 위해서는 표준 산술 명령어를 사용

  그 결과를 계산하는 소프트웨어 루틴을 호출

- ARM9E에서의 Newton-Raphson 나눗셈 루틴은 하드웨어 나눗셈 구현 장치처럼 사이클당 한 비트 만큼 빠르게 실행

초월함수 : LOG, EXP, SIN, COS

- 표 검색(table lookup)과 급수 전개(series expansion)사용
- 정확한 32비트의 결과를 생성하기도 하지만, 실제 많은 어플리케이션에서는 이를 초과하는 결과를 생성
- 밑이 2인 로그함수
  - 결과의 [31:26] 비트를 찾기 위해 CLZ 명령어를 사용
  - 근사값을 찾기 위해 처음 5개 비트의 표 검색을 사용
  - 정확하게 근사적인 에러 값을 계산하기 위해 급수 전개를 사용



엔디안 반전과 비트 연산

- 엔디안 반전
  - ARM 코어의 32비트 데이터 버스를 사용할 때에는 최대 효율성을 위해 8비트 배열과 16비트 배열을 한 번에 4 바이트씩 로드-스토어 하기를 원함
  - 여러 바이트 로드시 저장 순서에 영향

- 비트 조작
  - 비트 반전 : k 비트와 31 - k비트 값의 교환을 수행
  - 비트 이동 : k < 16 일 때에는 비트 k를 비트 2k로 이동
    k >=16 일 때에는 2k – 31 비트 만큼 이동
  - DES 초기 조작 : DES는 데이터 암호화 표준 (Data Encryption Standard)로서 벌크 데이터 암호화를 위한 일반적인 알고리즘, 암호화 전후로 해서 데이터에 64비트 조작을 수행

포화

- 포화는 오버플로우를 방지하기 위해 고정된 범위로 결과를 제한하는 것
- saturate 16(x) : x 를 -0x00008000에서 + 0x00007fff 사이의 값으로 제한
- saturate 32(x) : x 를 -0x80000000에서 + 0x7fffffff 사이의 값으로 제한

난수 생성

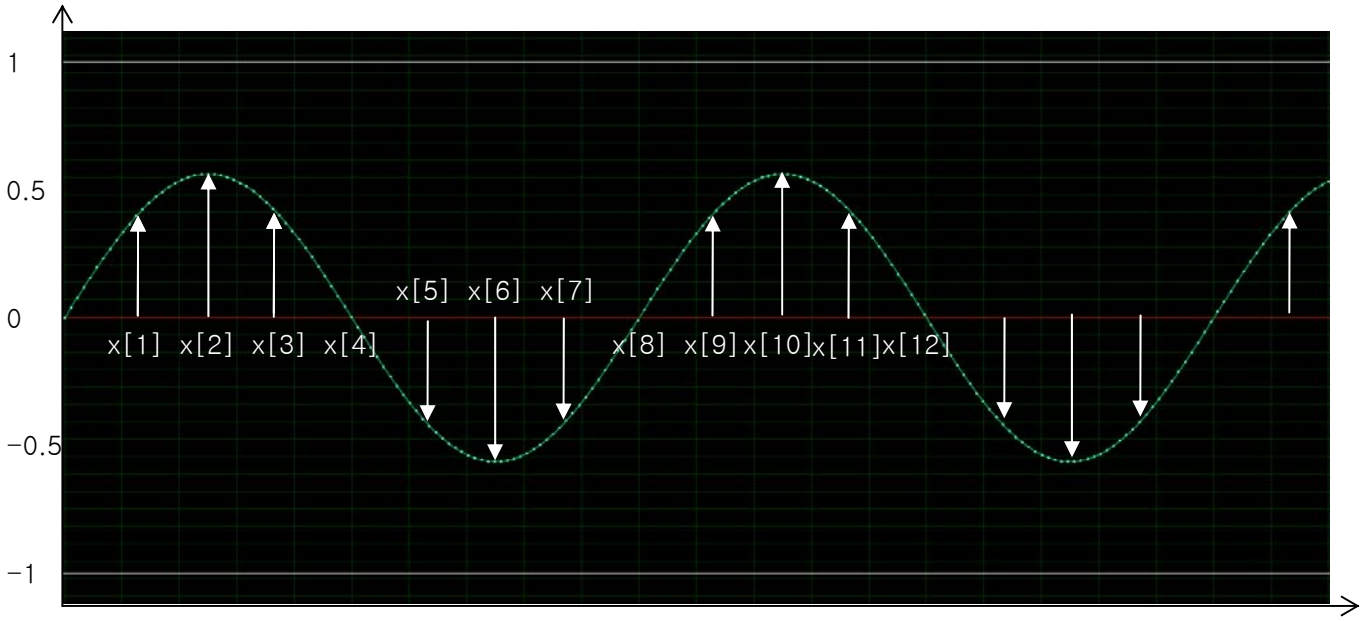
- 실제 난수를 만들어 내는 것은 랜덤 노이즈의 소스로 동작하는 특정 하드웨어 필요
- 어플리케이션들은 보통 가짜 난수 사용
- 가짜 난수 – 실제론 전혀 랜덤하지 않고 반복적인 순서로 그 수들을 생성, 하지만 그 순서가 매우 길고 흩어져 있기 때문에 랜덤한 것처럼 보임
- MAC(Multiply Accumulate)명령어를 사용

Digital Signal Processing

Jin-won Jung

jwjung@enc.hanyang.ac.kr

디지털 신호의 표현



디지털화된 아날로그 신호

1. 신호의 동적 범위 (Dynamic Range)
   - M = max|x[t]| t=0, 1, 2, 3, ...

2. 표현할 때 필요한 정확성
   - E = M x 0.0001 = 0.0001 볼트

표현 방법 선택

- 부동 소수점
  - 동적 범위와 정확성 제한을 만족
  - C의 float로 조작이 쉬움
  - ARM에서 지원하지 않음
- 고정 소수점
  - 분수를 표현하기 위해 분수에 곱을 해서 만든 정수 이용
    - X[t] = round_to_nearest_integer(5000xx[t])
  - 성능을 위해 2^k만큼 곱하는 것이 유리
    - X[t] = round_to_nearest_integer(2^kxx[t])
    - K=13일 경우 X[t]의 범위는 -8192~8191
      - 일반적인 경우 k=15, short의 최대 값으로 확장 (-32768~32767)
- 포화 (saturation)
  - 최대값의 범위보다 큰 값을 갖는 sample 값에 대해 최대값으로 포화
  - 큰 신호는 왜곡하지만 정확성 높아짐

고정소수점 형식으로 저장된 값의 연산

- 오른쪽 시프트에서의 라운딩
  - 오른쪽 시프트는 근처의 정수로 라운딩되지 않고 -∞로 라운딩된다.

$$y = (x + (1 << (shift - 1))) >> shift$$

- 나눗셈에서의 라운딩

$$y = (x + (d >> 1)) / d$$

- 헤드룸(Headroom)
  - 정수변수에 저장 가능한 최대 크기 : 발생할 수 있는 최대 크기
  - Q13을 16bit 정수에 저장하면 headroom은 4배 혹은 2bit

- Q 표현 방법의 변환
  - X[t]가 x[t]의 Qn 표현일 때
  - X[t]<<k는 x[t]의 Q(n+k) 표현
  - X[t]>>k는 x[t]의 Q(n-k) 표현

고정소수점 신호의 덧셈과 뺄셈

- $y[t] = x[t] + c[t]$ 신호를 고정소수점 형식으로 변경

$$Y[t] = 2^d\, y[t] = 2^d\,(x[t] + c[t]) = 2^{d-n}\, X[t] + 2^{d-m}\, C[t]$$

$$Y[t] = (X[t] << (d-n)) + (C[t] << (d-m))$$

  - Shift된 결과가 overflow가 되지 않도록 주의

$X[t], C[t], Y[t]$는 $x[t], c[t], y[t]$의 Qn, Qm, Qd표현

- Overflow 방지 방법
  - X[t]와 C[t]표현 방법이 각각 여유 있는 headroom 1bit를 갖는 지 확인한다.
  - X와 C보다는 Y를 위한 정수형을 더 크게 한다.
  - Y[t]를 위해 더 작은 Q 표현법을 사용한다.
  - Saturation을 사용

고정소수점 신호의 곱셈

- $y[t] = x[t] \times c[t]$ 신호를 고정소수점 형식으로 변경

$$Y[t] = 2^d\, y[t] = 2^d\,(x[t] + c[t]) = 2^{d-n}\, X[t] + 2^{d-m} C[t]$$

$$Y[t] = (X[t] \times C[t]) >> (n + m - d)$$

- 일반적인 사용 방법
  - 결과의 전체 시리즈를 더하기를 원한다면 X[t]와 C[t]보다 Y[t]를 저장하는 정수형을 크게하고, d=n+m으로 설정

$$Y[t] = X[t] \times C[t] \,//\, 곱셈$$

$$Y[t] += X[t] \times C[t] \,//\, 곱셈 + 덧셈$$

  - d=n으로 설정

$$Y[t] = (X[t] \times C[t]) >> m$$

고정소수점 신호의 나눗셈

- $y[t] = \frac{x[t]}{c[t]}$ 신호를 고정소수점 형식으로 변경

$$Y[t] = 2^d\, y[t] = \frac{2^d\, x[t]}{c[t]} = 2^{d-n+m}\, \frac{X[t]}{C[t]}$$

$$Y[t] = (X[t] << (d - n + m)) / C[t]$$

  - Shift가 overflow를 발생시키지 않는 지 확인
  - 일반적인 경우 n=m

$$Y[t] = (X[t] << d) / C[t]$$

디지털 신호를 표현하는 방법

신호 연산	동일한 정수 고정소수점
$y[t] = x[t]$	$Y[t] = X[t] <<< (d-n)$
$y[t] = x[t] + c[t]$	$Y[t] = (X[t] <<< (d-n)) + (C[t] <<< (d-m))$
$y[t] = x[t] - c[t]$	$Y[t] = (X[t] <<< (d-n)) - (C[t] <<< (d-m))$
$y[t] = x[t] \times c[t]$	$Y[t] = (X[t] \times C[t]) <<< (d-n-m)$
$y[t] = x[t] / c[t]$	$Y[t] = (X[t] <<< (d-n+m)) / C[t]$
$y[t] = sqrt(x[t])$	$Y[t] = isqrt(X[t] <<< (2 \times d - n))$

❖ <<<: 신호의 양의 부호에 따라 왼쪽이나 오른쪽 시프트를 갖는 연산

ARM에서의 DSP 코드 작성을 위한 가이드

- Saturation은 추가의 사이클을 소요하기 때문에 saturation이 필요하지 않도록 DSP 알고리즘 설계
- Load와 store를 최소화하도록 DSP 알고리즘 설계
  - 데이터를 load한 다음에 그 데이터를 최대한 많이 사용 (재사용 증가)
- 프로세서 interlock를 피하기 위해 ARM 어셈블리를 작성
  - Load와 곱셈 결과는 정지 사이클을 추가하지 않고서는 다음 명령어세서 사용이 불가능
- 내부 루프에서 레지스터를 14개 이하로 사용하도록 DSP 알고리즘 설계

ARM7TDMI에서의 DSP

- 메모리를 효율적으로 액세스하기 위해 다중 load-store 명령어 (LDM, STM) 사용
  - Load 명령어는 3사이클 소요
  - 다중 load-store 명령어는 첫 번째 워드 다음에 전송되는 워드 처리에 1사이클 소요
- 곱셈 명령어는 두 번째 오퍼랜드를 기반으로 한 초기값 사용
  - 예측 가능한 성능을 위해 상수 계수나 다중 레지스터를 규정하는 레지스터 오퍼랜드 사용
- 경우에 따라 MLA 명령어를 MUL과 ADD로 분리
  - 상수값을 곱하여 더하는 작업을 수행할 때 ADD 명령어와 함께 배럴 시프터 사용 가능
- 시프트를 가진 산술 명령어를 사용하면 고정된 계수를 빠르게 곱할 수도 있다.
  - 고정된 계수에 대해 시프트를 가진 ADD와 SUB는 MLA보다 빠른 MAC 명령어 제공

ARM9TDMI에서의 DSP

- 로드된 값을 2사이클 동안 사용하지 않도록 코드를 스케줄링하면 load 명령이 빠름
  - 다중 로드 명령어의 이점이 없으므로 16bit 데이터는 16bit short 형 배열에 저장
  - 데이터를 load하기 위해 LDRSH 사용
- 곱셈 명령어는 두 번째 오퍼랜드를 기반으로 한 초기값 사용
  - 예측 가능한 성능을 위해 상수 계수나 다중 레지스터를 규정하는 레지스터 오퍼랜드 사용
- 곱셈은 MAC 명령어와 속도 동일
  - 분리된 곱셈과 덧셈 보다는 MLA 명령 사용
- 시프트를 가진 산술 명령어를 사용하면 고정된 계수를 빠르게 곱할 수도 있다.
  - 고정된 계수에 대해 시프트를 가진 ADD와 SUB는 MLA보다 빠른 MAC 명령어 제공

StrongARM에서의 DSP

- signed byte와 하프워드 load는 피한다.
  - signed byte와 하프워드를 load하는 경우 2사이클 소요 (그 외에는 1사이클)
- 곱셈 명령어는 두 번째 오퍼랜드를 기반으로 한 초기값 사용
  - 예측 가능한 성능을 위해 상수 계수나 다중 레지스터를 규정하는 레지스터 오퍼랜드 사용
- 곱셈은 MAC 명령어와 속도 동일
  - 분리된 곱셈과 덧셈 보다는 MLA 명령 사용

ARM9E에서의 DSP

- 로드에 의한 interlock를 피할 수 있도록 코드를 스케줄링
  - Load-store 명령어는 ARM9TDMI와 성능이 동일함
- 32bit 정수를 곱하기 위해 MUL과 MLA만 사용 (16bit의 경우 SMULxy, SMLAxy)
- 곱셈은 MAC 명령어와 속도 동일
  - 분리된 곱셈과 덧셈 보다는 SMLAxy 명령 사용

ARM10E에서의 DSP

- 가능한 다중 load-store 명령어 사용
  - 높은 메모리 대역을 주기 위해 백그라운드에서 실행
  - 백그라운드 load가 완료되기 전에 데이터를 사용하지 않도록 코드 스케줄링
- 다중 load-store 명령어가 사이클당 두 워드를 전송할 수 있도록 데이터 배열이 64bit로 정렬되어 있는지 확인
- 32bit 정수를 곱하기 위해 MUL과 MLA만 사용 (16bit의 경우 SMULxy, SMLAxy)
- SMLAxy 명령어는 SMULxy보다 한 사이클 더 소요
  - MAC 명령에 대해 곱셈과 덧셈을 분리하는 것이 유용할 수 있음

인텔 Xscale에서의 DSP

- LDRD (load-double-word) 명령어는 한 사이클에 두 워드 전송
  - 처음 로드된 레지스터는 2사이클 동안, 두 번째 레지스터는 3사이클 동안 사용하지 않도록 코드 스케줄링
- LDRD를 사용할 수 있도록 데이터 배열이 64bit로 정렬되어 있는지 확인
- 곱셈의 결과는 즉시 사용할 수 없다.
  - 프로세서 정지 상태를 막기 위해 곱셈 명령어와 로드 명령어를 섞어서 코드 스케줄링
- 32bit 정수를 곱하기 위해 MUL과 MLA만 사용 (16bit의 경우 SMULxy, SMLAxy)

FIR (Finite Impulse Response) 필터

- 신호의 특정 주파수 대역을 변경하거나 특별한 효과를 얻기 위해 사용

$$y_t = \sum_{i=0}^{M-1} c_i x_{t-i}$$

y_t : 필터링 된 샘플
x_t : 필터링 되지 않은 샘플
c_i : 필터 계수
M : 필터의 길이

- FIR 필터의 구현
  - 입력 신호와 필터계수의 고정 소수점 표현 (각각 Qn, Qm 표현 적용)

$$X[t] = round(2^n xt)$$

$$C[i] = round(2^m ci)$$

  - 다음의 A[t]값을 계산하여 필터를 구현

$$A[t] = C[0]X[t] + C[1]X[t-1] + \Lambda + C[M-1]X[t-M+1]$$

FIR 필터 cont.

- FIR 필터 비트 정밀도 구하는 방정식

$$|A[t]| \le \max\{|X[t-i]|, 0 \le i < M\} \times \sum_{i=0}^{M-1} |C[i]|$$

$$|A[t]| \le \sqrt{\sum_{i=0}^{M-1} |X[t-i]|^2} \times \sqrt{\sum_{i=0}^{M-1} |C[t]|^2}$$

- Example 8.8 FIR로 구현한 High pass filter

$$y_t = -0.45x_t + 0.9x_{t-1} - 0.45x_{t-2}$$

  - x_i 와 c_i는 각각 Qn과 Qm 고정소수점 신호 X[t]와 C[i] (X[t]는 16bit 정수)

$$|A[t]| \le 2^{15} \times 1.8 \times 2^m = 1.8 \times 2^{15+m}$$

  - 가장 큰 계수 정확성을 위해 m=15

$$A[t] = -14746 \times X[t] + 29491 \times X[t-1] - 14746 \times X[t-2]$$

  - 출력 Y[t]에 대해 Y[t]=A[t]>>15로 설정
    - Y[t]값이 16bit 정수를 넘지 않도록 saturation하거나, Q(n-1) 표현을 사용하여 결과 저장

블록 FIR 필터

- 필터의 치수 M이 너무 큰 경우 레지스터 안에 필터 계수를 저장할 수 없다.
  - A[t]를 구하기 위해 너무 많은 load 명령이 사용
- RxS 블록 필터 내부 루프가 반복되는 동안 S개의 데이터와 계수 값을 한 번에 읽음
  - 각 루프에서는 RxS 곱을 R 누산기로 보내어서 더함

	X_{t-M+1}	X_{t-M+2}	X_{t-M+3}	X_{t-M+4}	X_{t-M+5}	X_{t-M+5}	X_{t-M+5}
A_t	C_{M-1}	C_{M-2}	C_{M-3}				
A_{t+1}		C_{M-1}	C_{M-2}	C_{M-3}			
A_{t+2}			C_{M-1}	C_{M-2}	C_{M-3}		
A_{t+3}				C_{M-1}	C_{M-2}	C_{M-3}	

4x3 블록 필터 구현

ARM exception handling

Juyoung Kim

jykim@enc.hanyang.ac.kr

Outline

- Introduction
- Exception handling
- Interrupt
- Interrupt handling scheme

Introduction

- Exception?
  - 외부 시스템으로부터 온 Event, Interrupt, Error 등으로 인해 Normal execution state에서 변환된 상태
- Exception handling
  - Exception을 발생시킨 원인과 상태에 따라 해당하는 작업을 수행하도록 하는 일
  - 목적
    - 프로그램의 순차적 수행을 중지시키고 특정 작업을 수행 할 수 있음
    - 시스템 성능 향상.

Exception Handling [1/6]

- Exception examples
  - Reset (System initialize)
  - Memory access failure
  - Undefined instruction
  - Software interrupt
  - External interrupt

- General operations
  - backup
    - SPSR <= CPSR
    - LR <= PC
  - Mode transition
    - CPSR <= [exception CPSR]
    - CPSR_T <= 0 (ARM mode로만 실행)
    - Exception mode가 Reset이나 FIQ인 경우
      - CPSR_F <= 1 (Disable FIQ interrupt)
    - CPSR_I (Disable normal interrupt)
  - Branch to exception handler
    - PC <= [Exception handler address]
      - Vector table을 참조하여 exception handler의 address를 알아냄

Exception Handling [2/6]

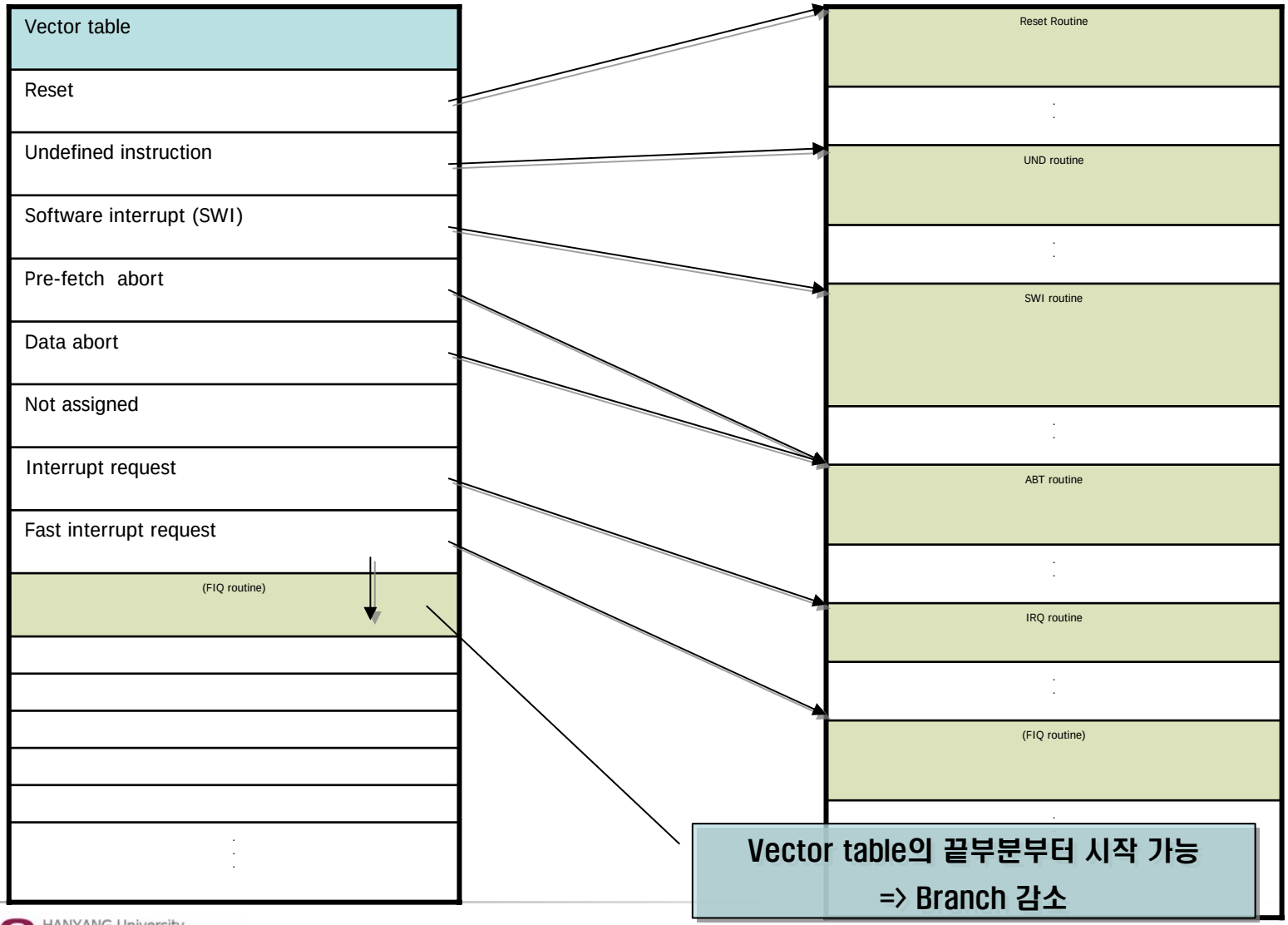
- ARM exception types

Exception	Mode	Purpose	Priority	Vector offset
Reset	-	HW/SW reset	1	Base + 0x00
Undefined instruction	UND	Software emulation	6	Base + 0x04
Software interrupt (SWI)	SVC	Safe-mode for OS	6	Base + 0x08
Pre-fetch abort	ABT	Memory protection	5	Base + 0x0C
Data abort	ABT	Memory protection	2	Base + 0x10
Not assigned	-	-	-	Base + 0x14
Interrupt request	IRQ	Normal interrupt 처리	4	Base + 0x18
Fast interrupt request	FIQ	Fast interrupt 처리	3	Base + 0x1C

- Base 는 0x00000000 또는 특정 주소 값이 가능
  - Address 0x00000000 에 jump 명령어를 넣고, 지정된 Base로 가도록 함

Exception Handling [3/6]

- Vector table



Exception Handling [4/6]

- ARM exception priority

Priority	Exception	I	F
1	Reset	1	1
2	Data abort	1	-
3	Fast interrupt request	1	1
4	Interrupt request	1	-
5	Pre-fetch abort	1	-
6	Software interrupt (SWI)	1	-
6	Undefined instruction	1	-

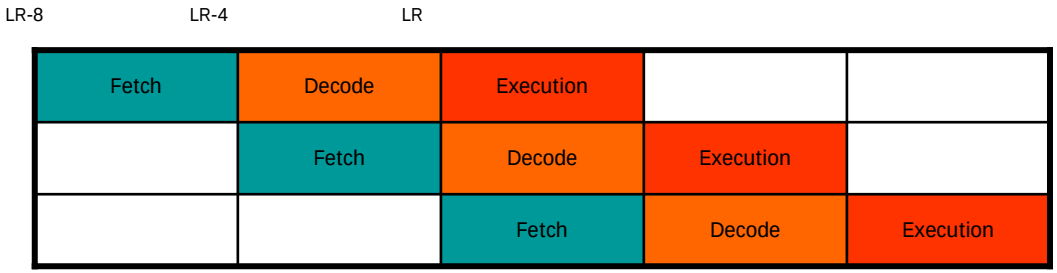
- I (Interrupt flag), F (Fast interrupt flag) 의 체크
  - 동등, 하위 priority를 가진 다른 interrupt가 현재 handler의 처리 도중 memory access를 일으킬 경우를 막기 위함

- SWI 와 UND는 같은 priority를 가짐
  - SWI는 Define되어 있는 명령어의 처리이므로 UND와 동시에 일어날 수 없음
  - 동일한 Priority로 두어도 상관 없음

Exception Handling [5/6]

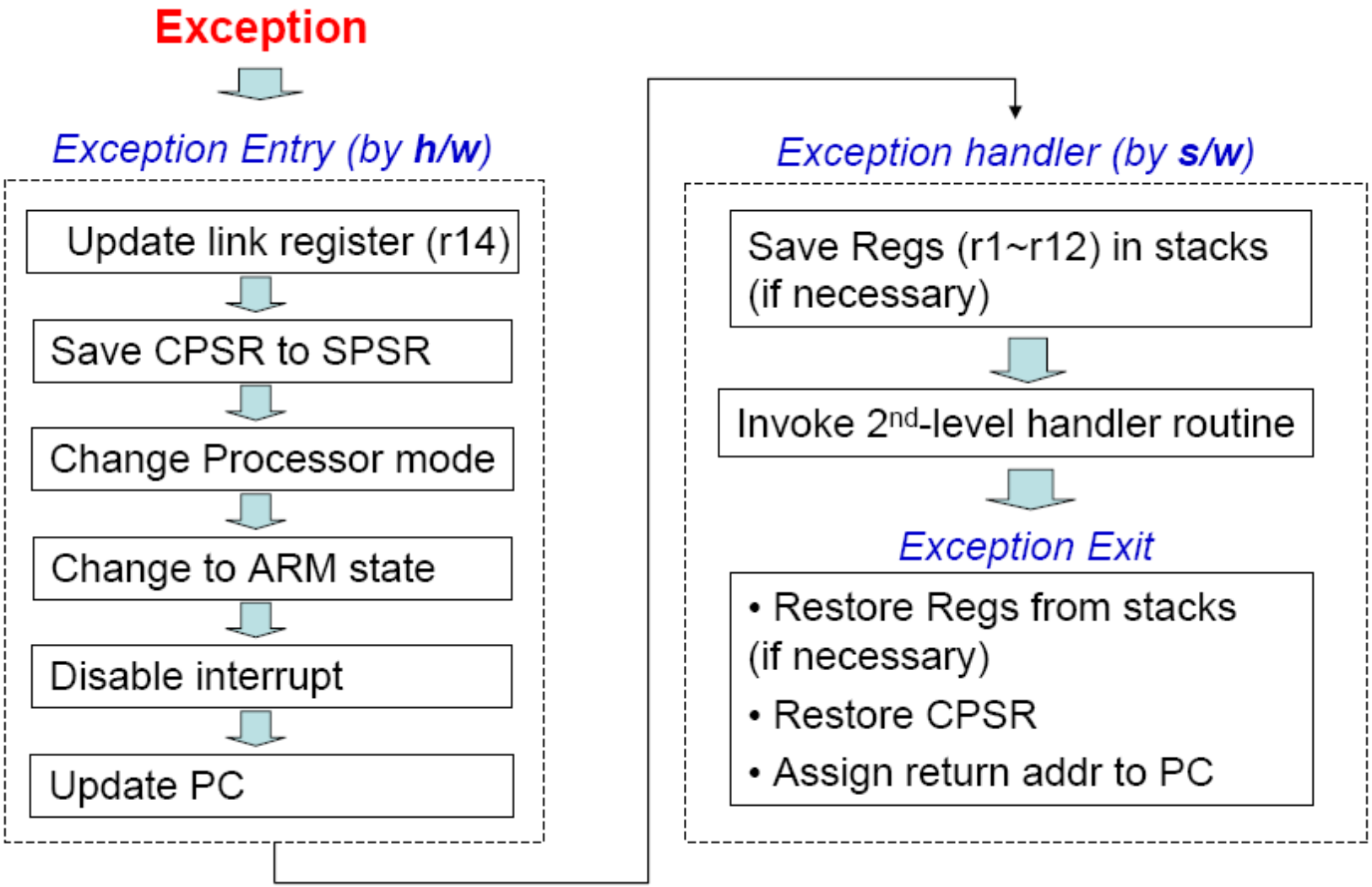
- LR (PC + 8) offset
  - 기존 LR = 마지막으로 실행된 명령어 주소 + 8

Priority	Exception	LR	Target
1	Reset	-	정의되지 않음
2	Data abort	LR - 8	Data abort가 발생한 명령어
3	Fast interrupt request	LR - 4	FIQ handler 복귀 주소
4	Interrupt request	LR - 4	IRQ handler 복귀 주소
5	Pre-fetch abort	LR - 4	Pre-fetch abort가 발생한 명령어
6	Software interrupt (SWI)	LR	다음 명령어
6	Undefined instruction	LR	다음 명령어



Exception Handling [6/6]

- Summary



Interrupt [1/4]

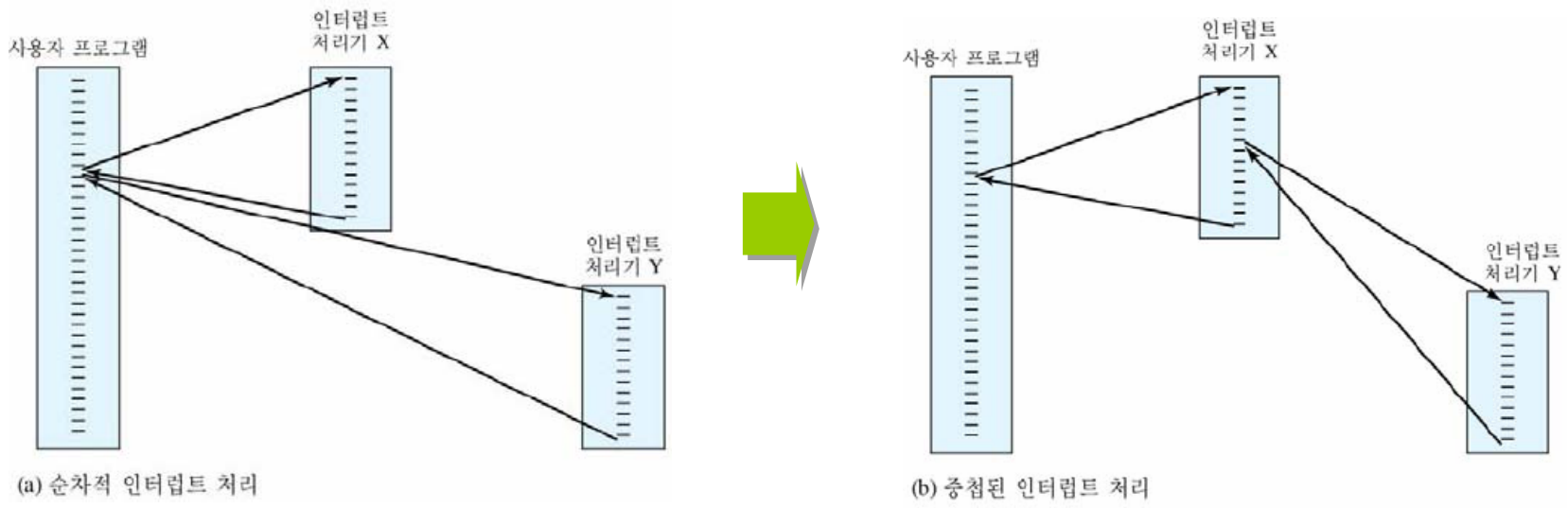
- ARM interrupt Types
  - External interrupt
    - FIQ
    - IRQ
  - Instruction interrupt
    - SWI

- Interrupt allocation
  - FIQ
    - 빠른 응답 시간을 요구하는 장치
    - Interrupt latency가 짧아야 함
  - IRQ
    - 보통 응답 시간을 요구하는 장치
  - SWI
    - User mode에서 Supervisor mode로 전환 시

Interrupt [2/4]

- Interrupt latency
  - Interrupt가 발생하고 나서 ISR (Interrupt Service Routine)을 시작할 때까지의 기간
  - 요청된 interrupt들의 latency가 최소화 되어야 함
  - Methods
    - Overlapped interrupt handler
      - ISR의 실행 중, 다른 ISR을 먼저 수행할 수 있음
    - Priority adaptation
      - 현재 실행중인 ISR보다 Priority 가 높은 interrupt에 대해서만 overlapped ISR을 가능케 함.

Priority of X < Priority of Y



Interrupt [3/4]

- General operations
  - Mode change
  - Backup
    - SPSR <= CPSR
    - LR <= PC
  - Interrupt disable
    - IRQ인 경우
      - CPSR_I <= 1 (동일 priority인 IRQ만 disable)
    - FIQ인 경우
      - CPSR_I <= 1; CPSR_F <= 1 (동일 priority인 FIQ와 낮은 priority인 IRQ를 disable)
    - MRS 명령어를 이용하여 직접 CPSR을 수정함
  - Branch to ISR

Interrupt [4/4]

- Interrupt stack
  - 각 mode별로 메모리 상에 full descending stack을 보유함
  - ISR의 실행 시 필요한 context를 stack에 저장
  - Overlapped ISR을 허용하면 요구 stack의 크기는 커짐
    - Interrupt의 depth에 따라 보유해야 하는 데이터의 크기 증가

Stack overflow 시에도 Vector table에는 손상이 없음

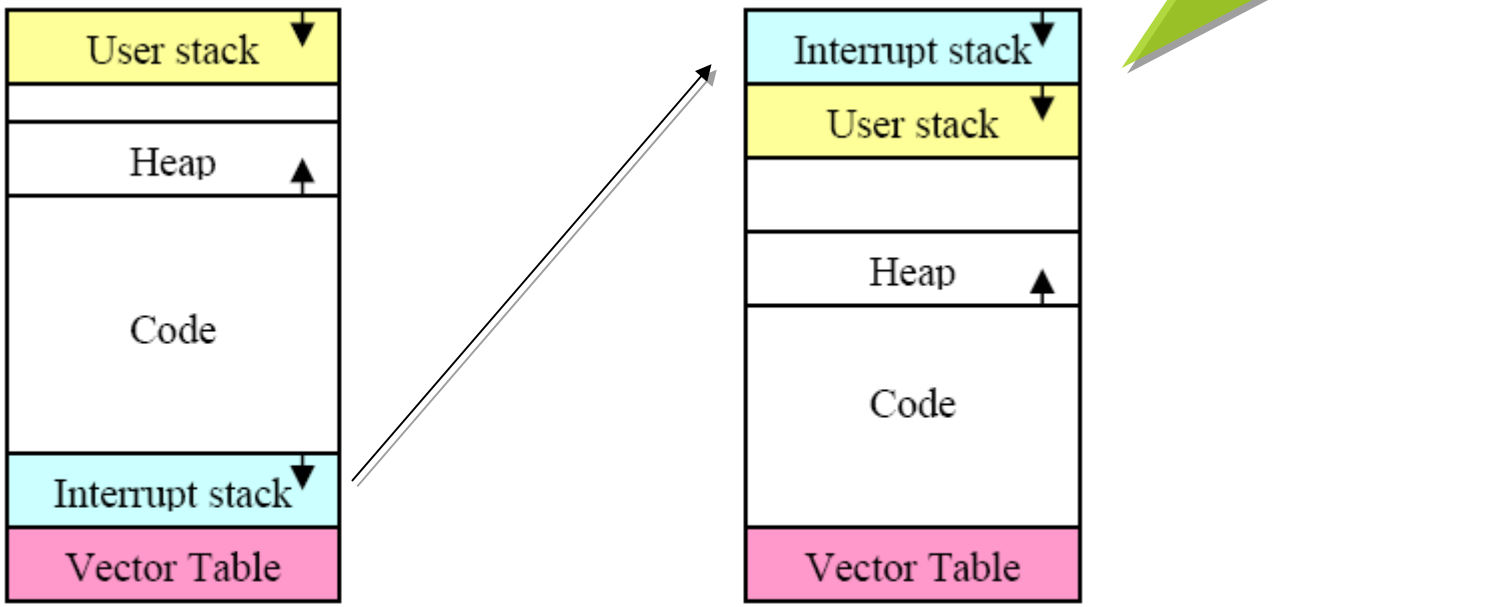


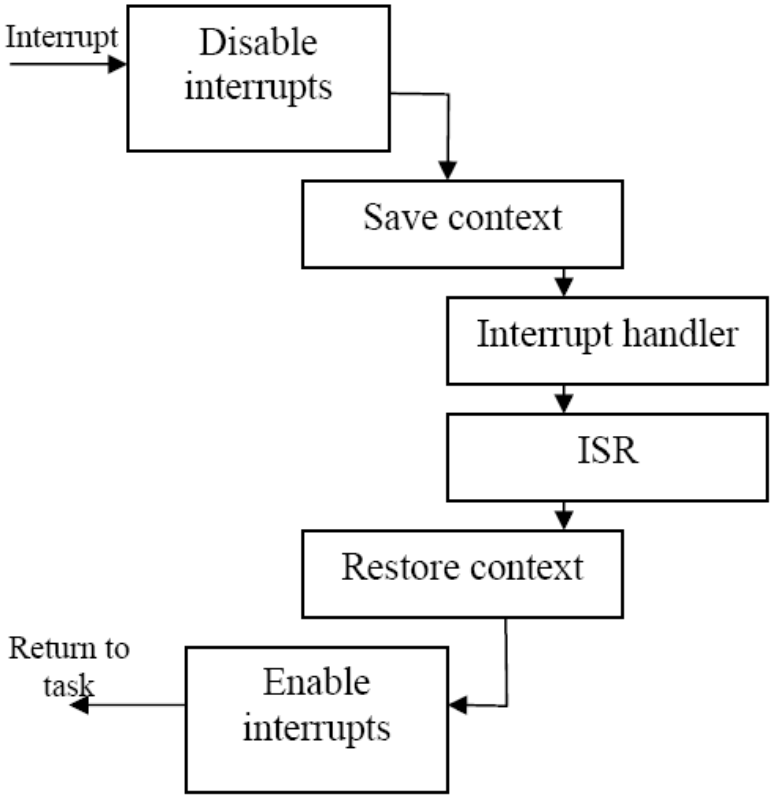
Figure 3 Typical Memory Layouts

Interrupt handling scheme [1/5]

- Schemes
  - Nested/Non-nested
  - Prioritizes Simple/Standard/Direct/Grouped
  - Re-entrant
  - VIC PL190 based

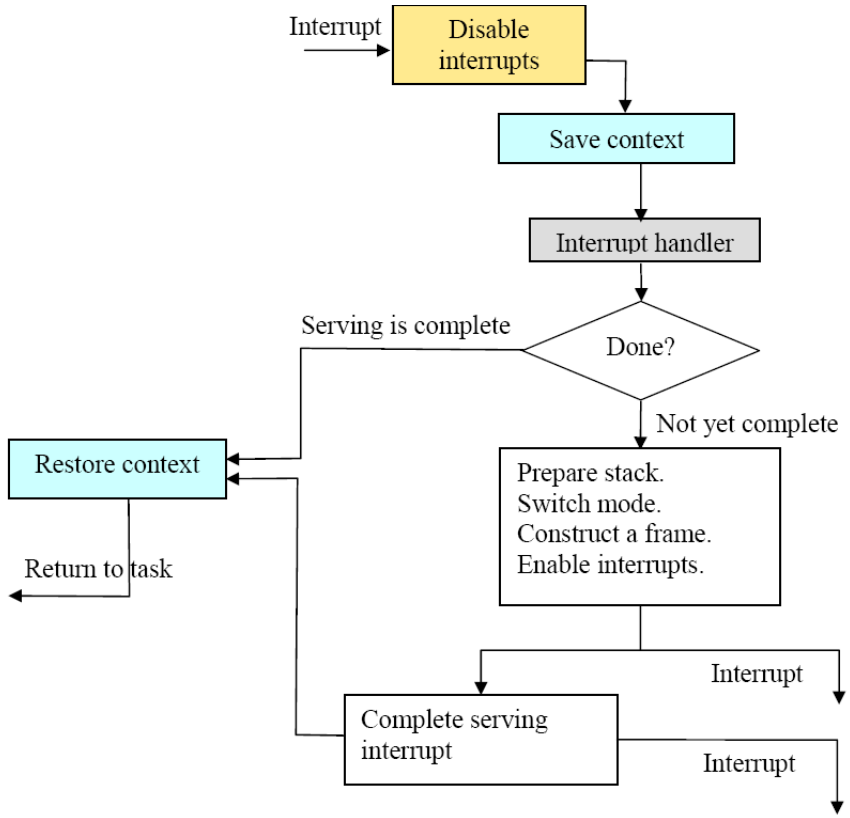
Interrupt handling scheme [2/5]

- Non-nested interrupt handler
  - 중첩을 허용하지 않아 한번에 하나의 interrupt만을 처리
  - Interrupt latency가 길다
  - Implementation & debugging 이 쉽다
  - 복잡한 system에 적합하지 않다



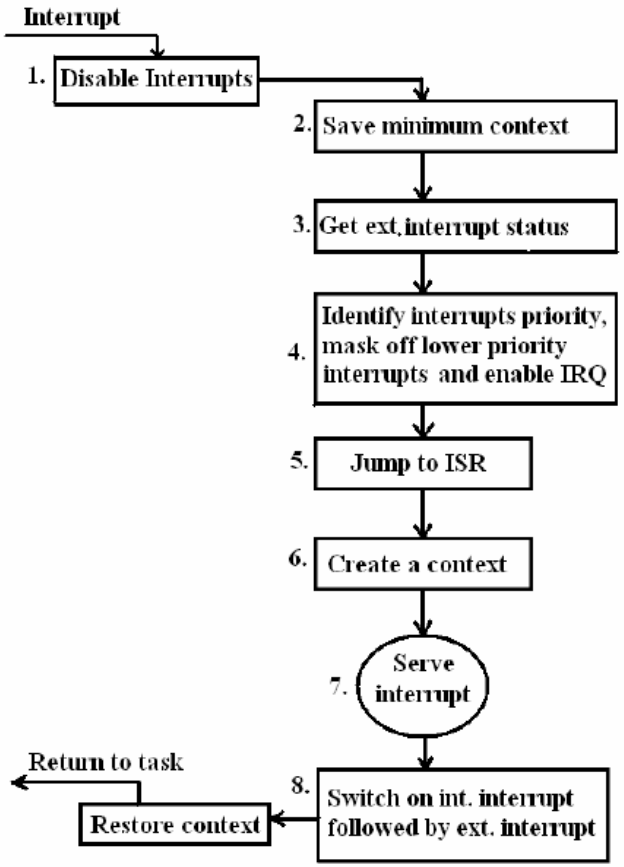
Interrupt handling scheme [3/5]

- Nested interrupt handler
  - Priority를 고려하지 않고 interrupt를 중첩되게 처리한다
  - Interrupt latency가 다소 길다
  - 다른 ISR이 수행되는 도중에도 interrupt 를 처리할 수 있다.
  - Low priority인 task가 High priority인 task의 처리를 막을 수 있다.



Interrupt handling scheme [4/5]

- Prioritizes Simple interrupt handler
  - 들어온 interrupt들 중 Priority가 가장 높은 것을 처리한다
  - Interrupt latency가 짧고, 예측 가능하다
  - Implementation & debugging 이 쉽다
  - 복잡한 system에 적합하지 않다



Interrupt handling scheme [5/5]

- Other interrupt handlers
  - Re-entrant interrupt handler
    - 실행중인 interrupt의 처리 중 다른 priority의 interrupt를 가 오면 현재 것을 중지하고 새로운 것을 수행
    - 장점 : Interrupt가 re-enable될 수 있어 interrupt latency를 줄일 수 있다
    - 단점 : Handler의 복잡도 감소
  - Prioritized standard interrupt handler
    - 들어온 interrupt들 중 Priority가 높은 것부터 차례로 실행
    - 장점 : Lower priority task 보다 Higher priority task의 interrupt latency가 짧다
    - 단점 : Interrupt latency를 예측 불가
  - Prioritized grouped interrupt handler
    - 동일 Priority의 interrupt를 group으로 묶어 처리
    - 장점 : Handler 의 복잡도가 감소. System response time을 줄임
    - 단점 : Interrupt를 group에 넣어야 하는지에 대한 복잡도가 증가

Firmware

Hyo Jung Yun

hjyun@enc.hanyang.ac.kr

펌웨어와 부트로더

- 펌웨어
  - 하드웨어와 어플리케이션/운영체제 레벨의 소프트웨어 사이의 인터페이스를 제공하는 하위 레벨 소프트웨어
  - ROM에 상주, 임베디드 하드웨어 시스템에 전원이 공급될 때 실행
  - 기본적인 시스템 동작 지원
  - 어플리케이션의 종류에 따라서 내용이 달라짐.
- 부트로더
  - 운영체제나 응용 프로그램을 하드웨어 타겟으로 다운로드 하는데 사용하는 작은 어플리케이션
  - 일반적으로 펌웨어에 통합되어 있음.

펌웨어 실행 과정 (1)

- 타겟 플랫폼 셋업 단계
  - 하드웨어 시스템 레지스터 프로그램
  - 플랫폼 확인
  - 진단 프로그램
  - 디버그 인터페이스
    - RAM에 브레이크포인트 설정
    - 메모리의 상태를 표시 및 수정
    - 현재 프로세서의 레지스터 내용 표시
    - 메모리를 ARM이나 Thumb 명령어로 역어셈블하기

- 하드웨어 추상화 단계
  - 하드웨어 추상 레이어 (HAL)
    - 일련의 정의된 프로그래밍 인터페이스 제공
  - 디바이스 드라이버
    - 특정 하드웨어 주변 장치와의 통신 역할을 담당하는 HAL 소프트웨어
    - 특정 주변 장치에 데이터를 읽고 쓰기 위한 API 제공

펌웨어 실행 과정 (2)

- 부트 이미지 로드
  - 이미지 포맷
    - 2진 이미지 (plain binary) : 헤더나 디버깅 정보를 포함하고 있지 않다.
    - ELF (Excutable and Linking Format)
      - ARM 기반의 시스템에서 가장 알려진 이미지 포맷
      - 재배치 오브젝트 파일
      - 실행 가능 오브젝트 파일
      - 공유 오브젝트

- 제어권 양도
  - 펌웨어가 운영체제나 어플리케이션에게 플랫폼의 제어권을 넘기는 작업
  - ARM 시스템
    - 벡터 테이블을 업데이트하고 pc 값을 수정하는 것을 의미
    - 벡터 테이블 업데이트는 특정 익셉션과 인터럽트 벡터를 수정하는 작업을 포함함
    - pc는 운영체제 엔트리 포인터 주소를 가리킬 수 있도록 변경되어야 함.

ARM Firmware Suite

- AFS (ARM Firmware Suite)
  - 펌웨어 패키지
  - ARM 기반의 임베디드 시스템을 위해 설계
  - XScale, StrongARM 프로세서 등 지원
  - µHAL
    - 서로 다른 통신 장치들, 직렬 통신 장치 사이의 하위 레벨 디바이스 드라이버 제공
    - 표준 함수 프레임워크 제공
    - 시스템 초기화
      - 타겟 플랫폼과 프로세서 코어 셋업
    - 폴링 방식의 시리얼 드라이버
    - LED 지원
    - 타이머 지원
      - 주기적인 인터럽트 설정 가능
    - 인터럽트 컨트롤러
  - Angel
    - 호스트 디버거와 타겟 플랫폼 사이의 통신 지원
    - 메모리 수정, 이미지 다운로드/실행
    - 브레이크포인트, 프로세서 레지스터 안의 내용 표시

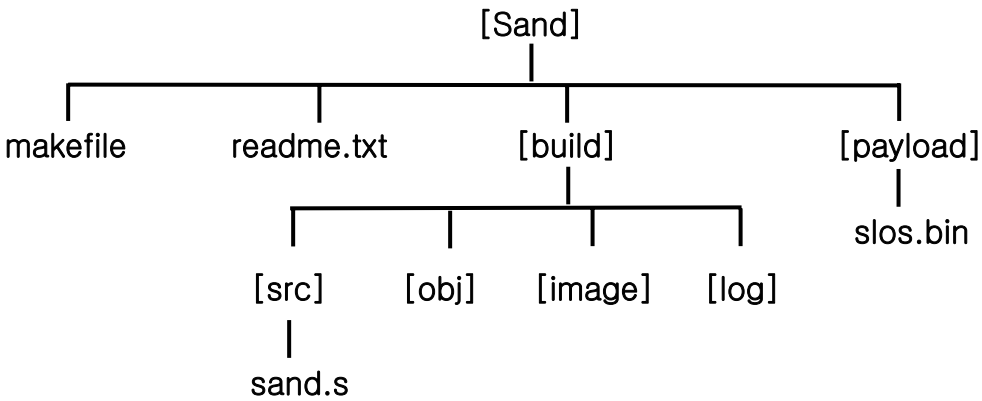
레드햇 RedBoot

- RedBoot
  - Red Hat 사에서 개발된 펌웨어 툴
  - ARM, MIPS, SH 등 여러 CPU에서 실행될 수 있도록 설계
  - GDB (GNU Debugger)를 이용한 디버그 기능 제공
  - 통신
    - 직렬 통신이나 이더넷을 선택적으로 사용 가능
    - XModem 프로토콜 : 직렬통신
    - TCP : 이더넷 통신
    - Bootp, telnet, tftp와 같은 다양한 네트워크 표준 지원
  - 플래시롬 메모리 관리 장치
    - 플래시롬에 이미지 다운로드, 업데이트, 삭제 하는데 사용될 수 있는 파일 시스템 루틴 제공
    - 이미지 압축, 압축풀기 기능 제공
  - 완전한 운영체제 지원
    - 임베디드 리눅스, 레드햇 eCos 등등 유명한 운영체제들의 로딩과 부팅을 지원
    - 부팅시, 커널로 바로 보내게 될 파라미터들을 정의하는 기능 지원

타겟

- ARM Evaluator-7T (Core : ARM7TDMI)

특 징	구 성
코드	ARM 명령어만 사용
툴 체인	ARM Developer Suite 1.2
이미지 크기	700 바이트
소스 크기	17 KB



Sandstone 코드 구조

실행 과정

단 계	설 명
1	Reset 익셉션 처리
2	하드웨어 초기화
3	메모리 리매핑
4	통신 하드웨어 초기화
5	부트로더 : 페이로드 복사 및 제어권 양도

1단계 : Reset 익셉션 처리

- Default Vector Table에 Reset 벡터만 정의
- 이 외의 벡터는 무한 루프 수행하는 더미 핸들러로 분기
- 1단계 결과
  - 더미 핸들러가 셋업
  - 하드웨어를 초기화하기 위해 제어권이 코드로 넘어감.

```
sandstone_start
  B  sandstone_init1      ; reset vector
  B  ex_und               ; undefined vector
  B  ex_swi               ; swi vector
  B  ex_pabt              ; prefetch abort vector
  B  ex_debt              ; data abort vector
  NOP                     ; 사용되지 않음
  B  int_irq              ; irq vector
  B  int_fiq              ; fiq vector
```

```
ex_und      B  exund      ; 무한 루프
ex_swi      B  ex_swi     ; 무한 루프
ex_dabt     B  ex_dabt    ; 무한 루프
ex_pabt     B  ex_pabt    ; 무한 루프
int_irq     B  int_irq    ; 무한 루프
int_fiq     B  int_fiq    ; 무한 루프
```

2단계 : 하드웨어 초기화

- 시스템 레지스트리 셋업
  - 0x03ff0000으로 셋업
  - 하드웨어 시스템 레지스터가 ROM과 RAM에서 떨어진 위치에 올 수 있도록 하여 주변장치와 메모리를 분리시켜줌.

- 2단계 결과
  - 시스템 베이스 주소 0x03ff0000에서 설정된다.
  - 그 과정을 디스플레이 하기 위해 세그먼트 디스플레이 장치가 설정된다.

```
sandstone_init1
        LDR     r3, =SYSCFG
        LDR     r4, =0x03ffffa0
        STR     r4, [r3]
```

3단계 : 메모리 리매핑

- SRAM 을 초기화하고 메모리를 재배치한다.
- 리매핑

메모리 종류	시작 주소	끝 주소	크기
플래시롬	0x01800000	0x01880000	512K
SRAM 뱅크 0	0x00000000	0x00040000	256K
SRAM 뱅크 1	0x00040000	0x00080000	256K

- 3단계 결과
  - 메모리 표 재배치
  - pc 는 다음 단계를 가리킨다. 이 주소는 새로 리맵된 플래시롬에 위치해 있다.

```
LDR         r14, =sandstone_init2
LDR         r4, =0x01800000             ; 새로운 플래시롬 위치
ADD         r14, r14, r4
ADRL        r0, memorymaptable_str
LDMIA       r0, {r1-r12}
LDR         r0, =EXTDBWTH               ; =(SYSCFG + 0x3010)
STMIA       r0, {r1-r12}
MOV         pc, r14                     ; 재배치된 메모리로 분기
sandstone_init2
                ; sandstone_init2 다음 코드는 +0x1800000 에서 시작한다.
```

4단계 : 통신 하드웨어 초기화

- 시리얼 포트 설정, 표준 배너 출력 작업

- 4단계 실행 결과
  - 시리얼 포트가 초기화됨
    - 9600bps, 패러티 비트 없음, 정지 비트 하나, 흐름제어 없음
    - 시리얼 포트를 통해 보내진 Sandstone 배너는 다음과 같다.

```
Sandstone Firmware (0.01)
- paltform ..... e7t
-status ..... alive
-memory ..... remapped
+ booting payload ...
```

5단계 : 부트로더 – 페이로드 복사 및 제어권 양도

- 페이로드를 복사하여 pc의 제어권을 복사된 페이로드로 넘겨준다.
- 부트로더 코드는 페이로드가 암호화나 압축이 불필요한 간단한 2진 이미지라고 가정함.
- 5단계 실행 결과
  - 페이로드가 SRAM에 복사된다. 이 주소는 0x00000000 이다.
  - pc의 제어권이 페이로드로 넘어간다. pc = 0x00000000

```
sandstone_load_and_boot
                MOV     r13, #0                         ; 목적 주소
                LDR     r12, payload_start_address      ; 시작 주소
                LDR     r14, payload_end_address        ; 마지막 주소
_copy
                LDMIA   r12!, {r0-r11}
                STMIA   r13!, {r0-r11}
                CMP     r12, r14
                BLT     _copy
                MOV     pc, #0
```

요약 정리

- 펌웨어
  - 하드웨어를 어플리케이션이나 운영체제와 인터페이스 해주는 하위 레벨의 코드를 말한다.
- 부트로더
  - 운영체제나 어플리케이션을 메모리로 다운로드 한 후, 그 소프트웨어로 pc 제어권을 넘겨주는 작업을 수행하는 소프트웨어를 말한다.
- ARM Firmware Suite, RedBoot
  - ARM Firmware Suite는 ARM 기반의 시스템만을 위해서 설계됨
  - RedBoot는 보다 범용적인 제품으로 다른 프로세서에서도 사용
- Sandstone
  - Reset 익셉션 처리
  - 하드웨어 초기화
  - 메모리맵 변경
  - 부트로더가 이미지를 SRAM에 로드한 후 pc의 제어권을 이미지에게 넘겨준다.

Embedded Operating Systems

Young Hoi Heo

yhheo@enc.hanyang.ac.kr

Outline

- 기본 컴포넌트
- 예 : SLOS
  - SLOS 디렉토리 레이아웃
  - 초기화
  - 메모리 모델
  - 인터럽트와 익셉션 처리
  - 스케줄러
  - 문맥전환
  - 디바이스 드라이버 프레임워크

기본 컴포넌트

- 초기화 코드
  - 처음으로 실행되는 운영체제 코드 (내부 데이터 구조체와 전역 변수, 하드웨어 셋업 포함)
  - 초기화 과정은 펌웨어가 제어권을 넘겨주자마자 시작
- 메모리 관리
  - 시스템 스택과 태스크 스택을 셋업하는 일
  - 스택의 위치는 태스크나 시스템을 위해 메모리를 어떻게 사용할 것인지에 따라 다름
- 정적 태스크
  - 운영체제 이미지 안에 포함
  - SLOS는 정적 태스크 기반의 운영체제
- 동적 태스크
  - 운영체제가 설치되고 실행된 다음에 로드되어 실행
  - 스택은 태스크가 생성될 때에 셋업

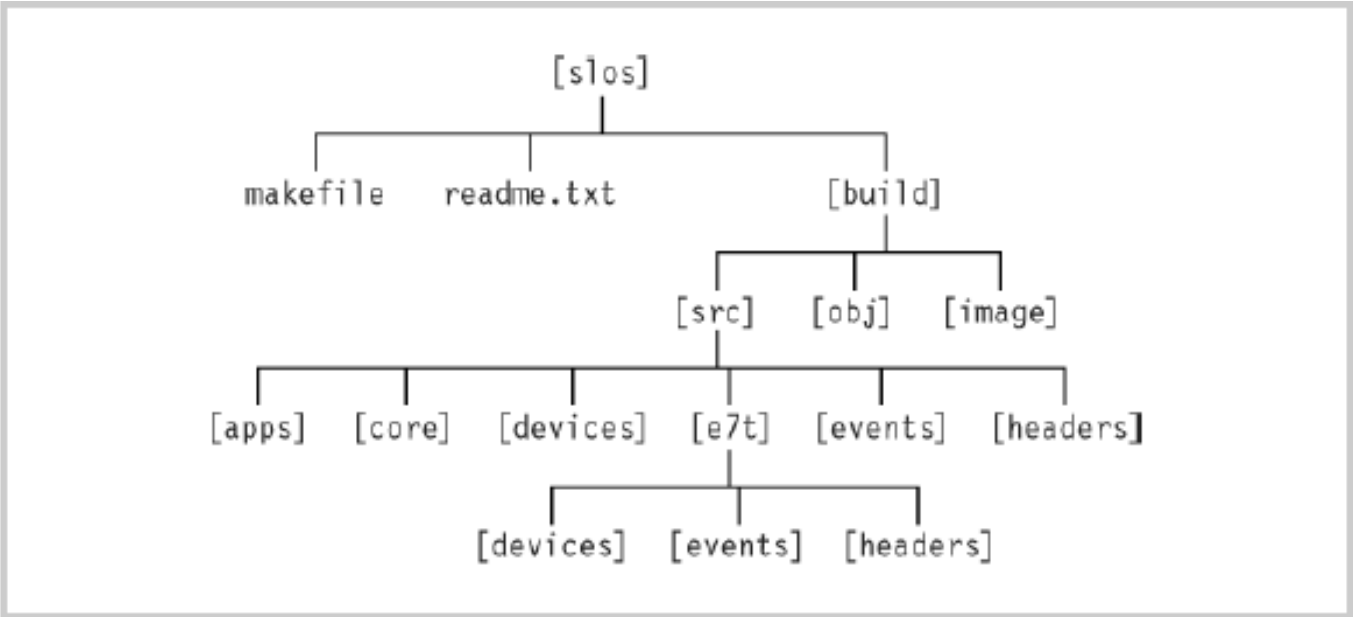
기본 컴포넌트

- 스케줄러
  - 다음으로 어떤 태스크가 실행될지를 결정하는 알고리즘
  - 간단한 방법 – 라운드 로빈 알고리즘

- 디바이스 드라이버 프레임워크
  - 운영체제가 서로 다른 주변장치간에 일관된 인터페이스를 제공하기 위해 사용하는 방법
  - 주변장치를 안전하게 액세스할 수 있는 방법을 제공
    - Ex) 여러 어플리케이션들이 동일한 주변 장치를 동시에 액세스하지 못하게 한다.

예 : SLOS

- SLOS 디렉토리 레이아웃



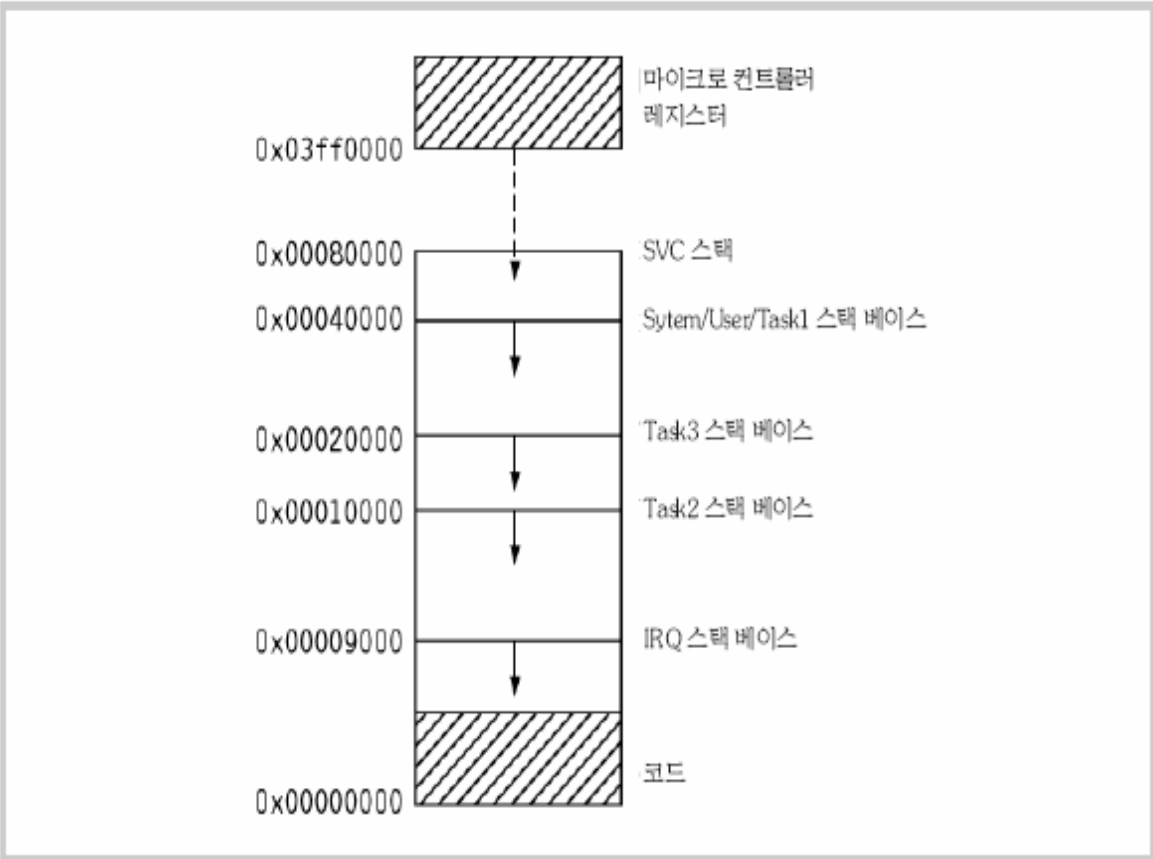
예 : SLOS

- 초기화
  - 스타트업
    - FIQ 레지스터와 system, SVC, IRQ 모드 스택을 셋업
  - PCB (Process Control Block) 셋업 코드 실행
    - 각 태스크의 상태를 포함하고 있는 PCB 가 모든 ARM 레지스터들을 포함하면서 셋업
    - 문맥교환이 이루어지는 동안 태스크의 상태를 저장하고 복구하는데 사용
    - 셋업 코드는 PCB를 초기 시작 상태로 설정
  - C 초기화 코드 실행
    - 디바이스 드라이버
    - 이벤트 핸들러
    - 주기적인 타이머 초기화 루틴을 설정

→ 작업들이 완료되어야 첫 번째 태스크가 실행될 수 있다.

예 : SLOS

- 메모리 모델


0x03ff0000
마이크로 컨트롤러
레지스터
0x00080000
SVC 스택
0x00040000
Sytem/User/Task1 스택 베이스
0x00020000
Task3 스택 베이스
0x00010000
Task2 스택 베이스
0x00009000
IRQ 스택 베이스
0x00000000
코드

예 : SLOS

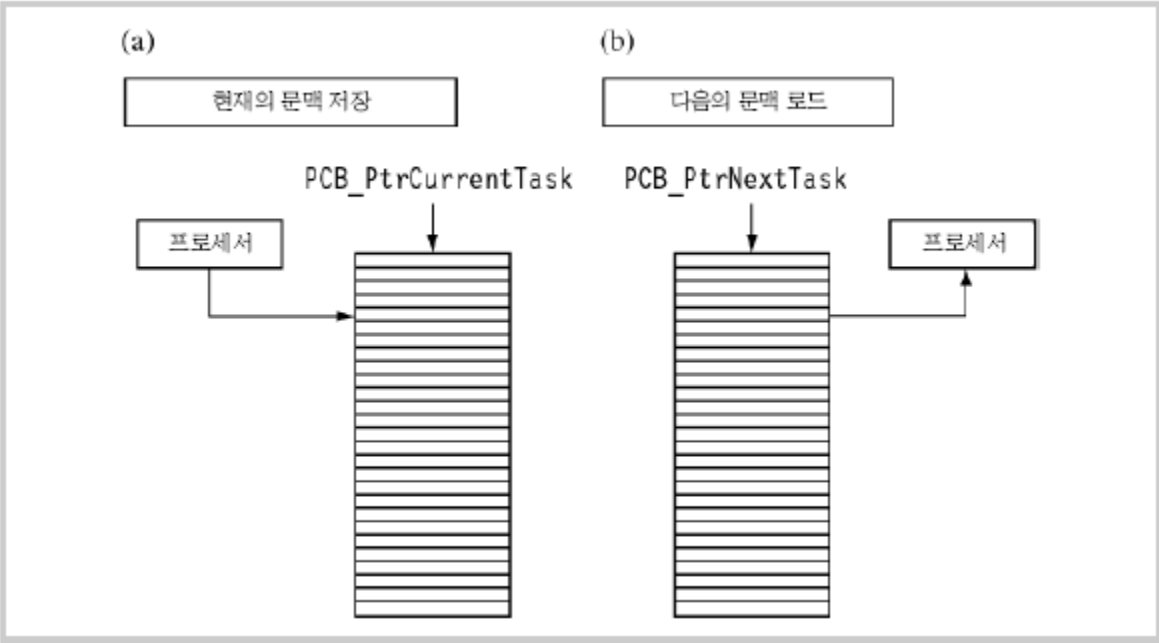
- 인터럽트와 익셉션 처리

익셉션 할당	목적
Reset	운영체제의 초기화
SWI	디바이스 드라이버를 액세스 하기 위한 메커니즘
IRQ	이벤트를 서비스하기 위한 메커니즘

예 : SLOS

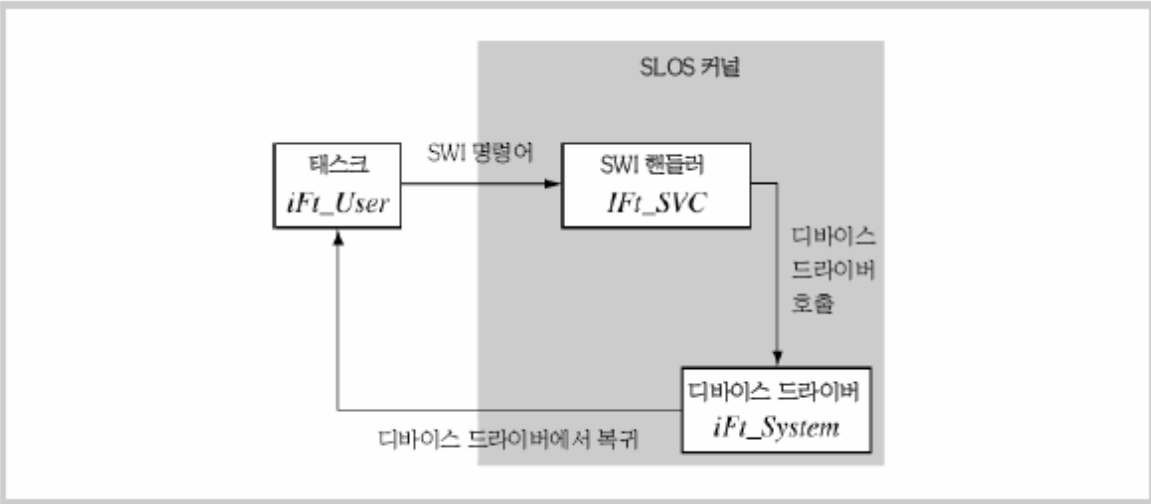
- 스케줄러
  - 간단한 정적 라운드 로빈 알고리즘 사용
  - 정적이란 운영체제가 초기화될 때에만 태스크가 생성
  - 태스크는 활성화 상태에 있을 때에는 생성되지도 제거되지도 않음

- 문맥전환



예 : SLOS

- 디바이스 드라이버 프레임워크 (DDF : Device Driver Framework)
  - SWI 명령어를 사용하여 구현
  - DDF는 어플리케이션이 하드웨어를 직접 액세스 못하도록 함으로써 운영체제를 보호
  - 태스크들을 위한 일관된 표준 인터페이스를 제공



디바이스 드라이버 호출

Caches

Juyoung Kim

jykim@enc.hanyang.ac.kr

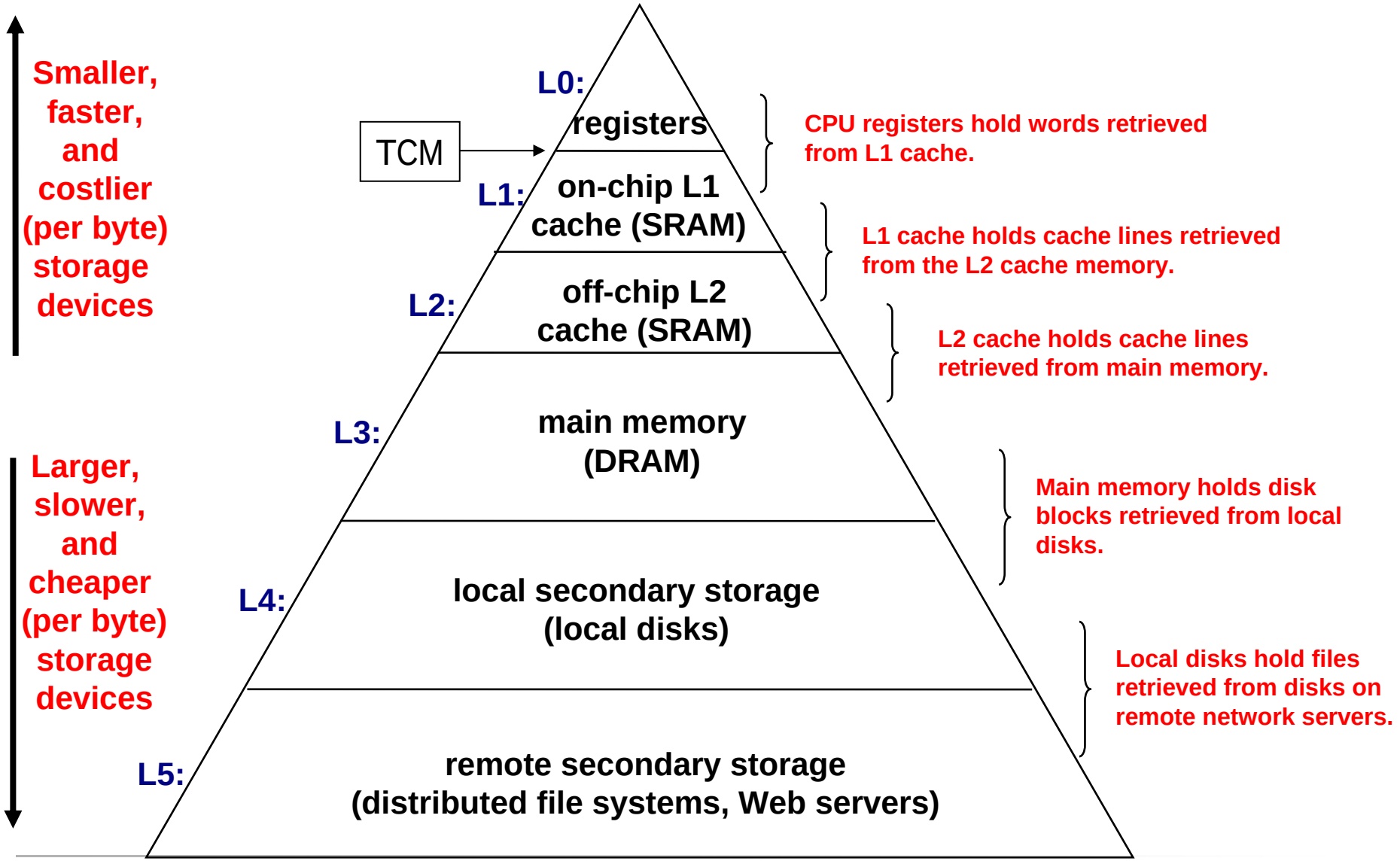
Outline

- Introduction
- Memory hierarchy and cache memory
- Cache architecture
- Cache policy
- Coprocessor 15 and cache
- Cache flush & Cache clean
- Cache lockdown
- Cache & SW performance

Introduction

- Cache?
  - Feature
    - Fast speed & small size
    - Allocated between processor core and main memory
  - Role
    - Memorize which is recently referenced main memory data
    - To solve problems that caused by slow speed of main memory
- Write buffer?
  - Feature
    - Small FIFO memory
    - Allocated between processor core and main memory
  - Role
    - For the relief of delay of inter-working between processor and main memory

Memory hierarchy and cache memory



Memory hierarchy and cache memory

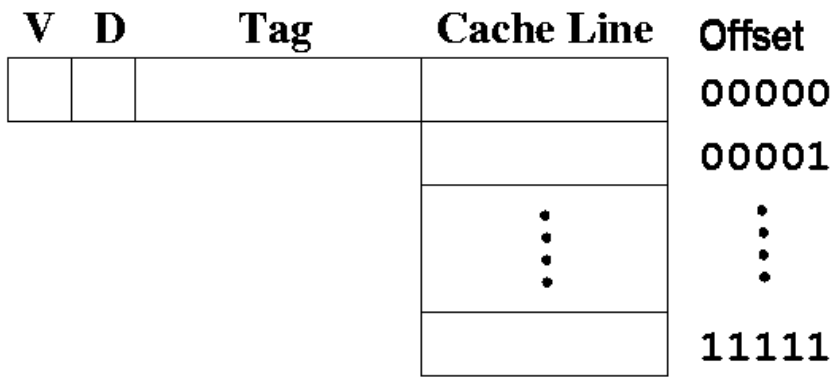
- Cache & MMU
  - If above the MMU, then cache is logically (virtually) addressed
    - Virtual cache has lower latency because it avoids translation time
  - If below the MMU, then physically addressed
    - Physical cache has higher latency because it has translation time
- Performance improvement with cache
  - Locality
    - Temporal locality
      - Programs tends to use recent information
    - Spatial locality
      - Memory location referenced recently is more likely to be referenced than farther memory location
  - Program access a relatively small portion of the address space at any instant of time.

Cache architecture

- Types of cache architecture
  - Unified cache
    - Adapted to Von Neumann architecture
    - merged instruction & data path
  - Split cache
    - Adapted to Harvard architecture
    - Separated instruction & data path
- Component of cache architecture
  - Cache memory
  - Cache controller
  - Address from processor
    - # of cache line : [Cache-tag][set index][data index]

Cache architecture

- Cache memory
  - Cache-tag
    - upper bits of the address
    - The size of the tag is **32 - lg N** where **N** is the number of bytes in the data part of the slot
  - Status
    - V (valid): Indicating whether the slot holds valid data
    - D (dirty): Indicates that the data in the slot (to be discussed momentarily) has been modified (written to) or not
  - Data block
    - the actual data itself. There are N bytes, where N is a power of 2



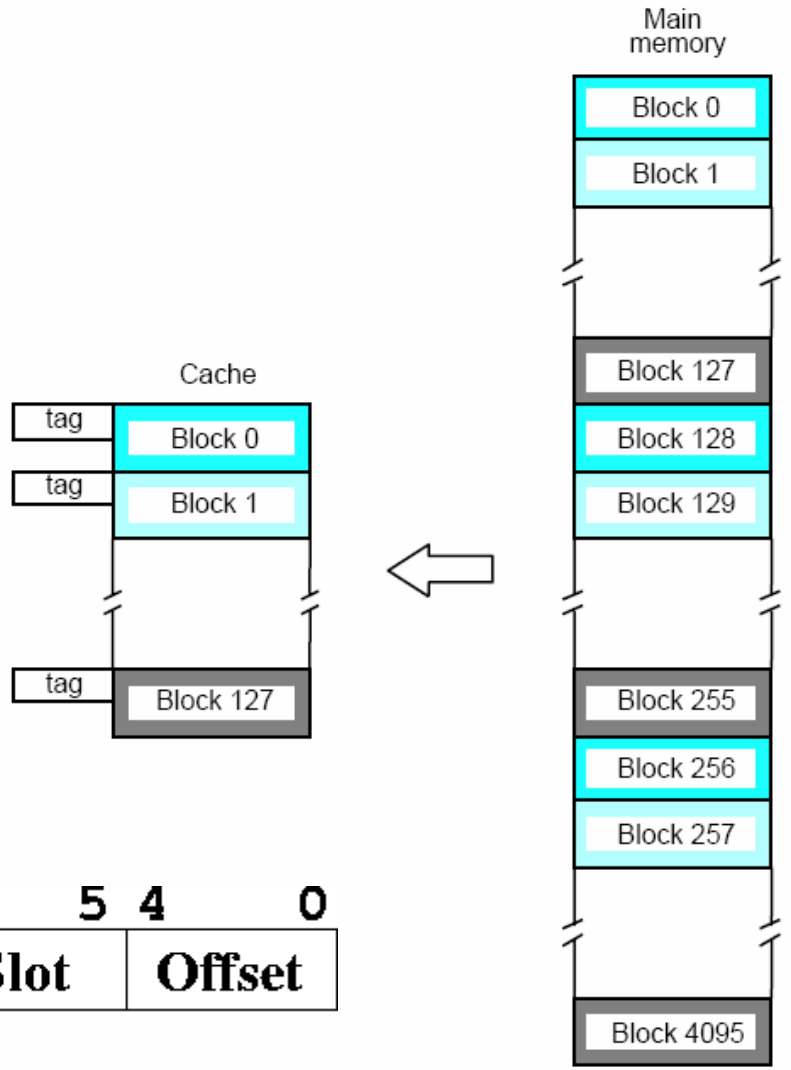
Single Slot of Cache

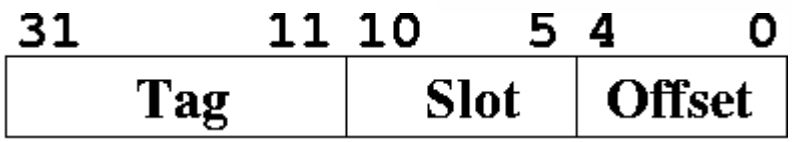
Cache architecture

- Cache controller
  - Sequence
    1. Select cache line by Set index
    2. Check status (V, D)
       1. Hit
          1. Load data from data block with consider offset by data index
       2. Miss
          1. Cache line fill
  - Hit & miss
    - Hit: When you look for data at a given address, and find it stored in cache
    - Miss: not hit

Cache architecture

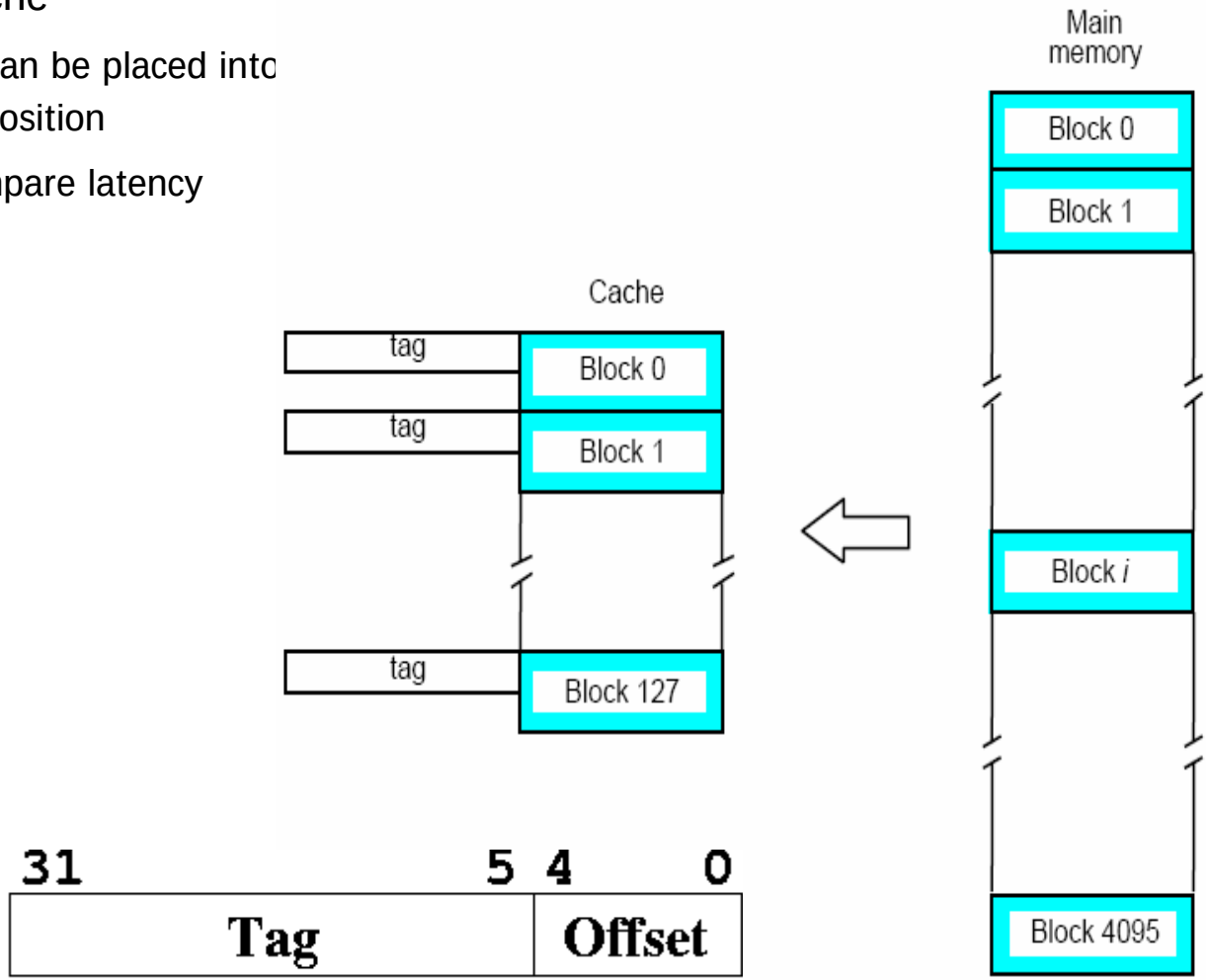
- Memory mapping type
  - Direct mapped cache
    - Block (i) of the main memory maps onto block (i modulo 128) of cache





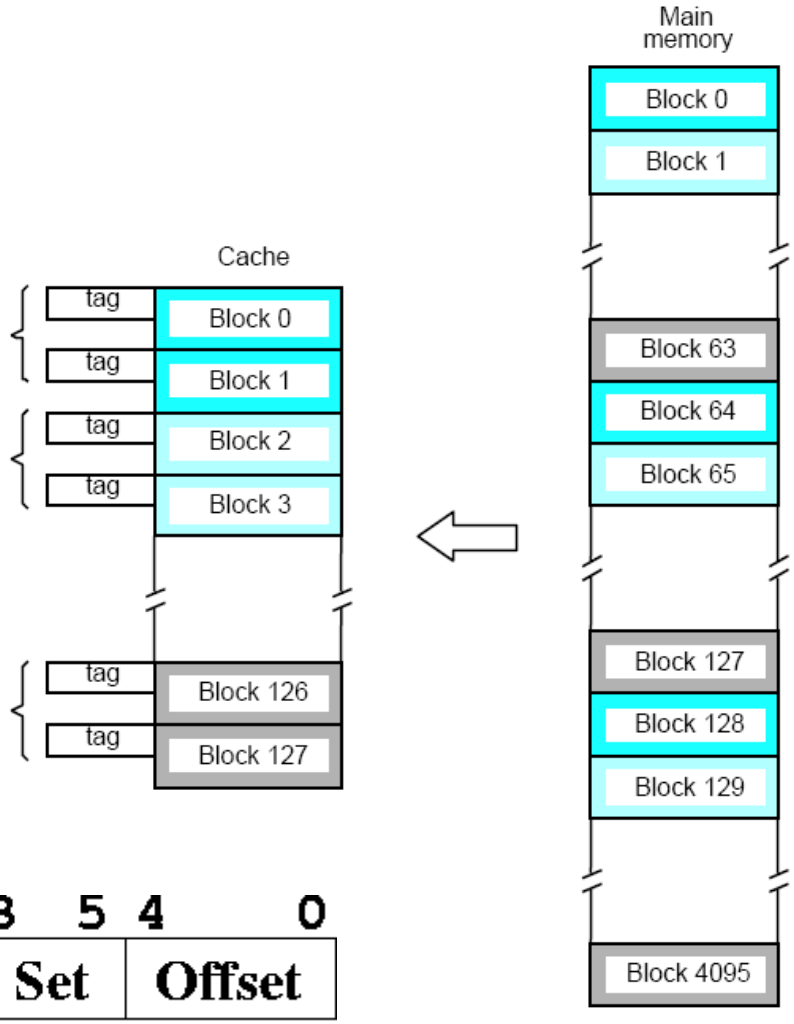
Cache architecture

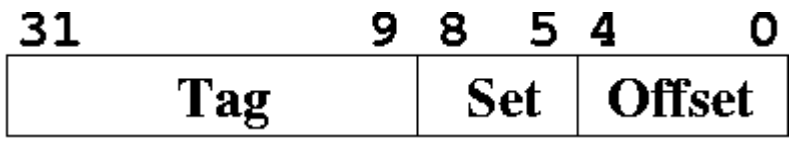
- Memory mapping type
  - Fully associative cache
    - A memory block can be placed into any cache block position
    - The most tag compare latency



Cache architecture

- Memory mapping type
  - Set associative cache
    - n-ways mapped cache, Block (i) of the main memory maps onto block (i modulo 2^(7-√n) of cache
    - Increase tag compare latency n time more than direct mapped cache



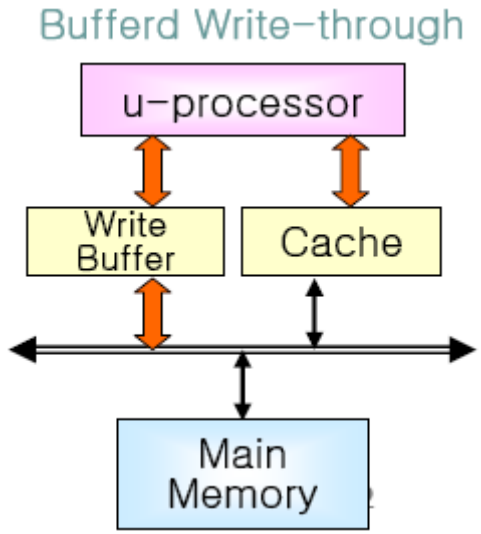


Cache architecture

- Content-addressable memory (CAM)
  - Special type of computer memory used in certain very high speed searching applications
  - Divide cache memory to unit called “Way”
  - On a way, cache lines have same set index
  - ARM920T, ARM940T adapted 64-ways set associative cache

Cache architecture

- Write buffer
  - A variation of write-through where the cache uses a "write buffer" to hold data being written back to main memory
  - Problem
    - Data which is reside write buffer to write, can't read until be saved to main memory
      - So, FIFO length should to be shorten



Cache architecture

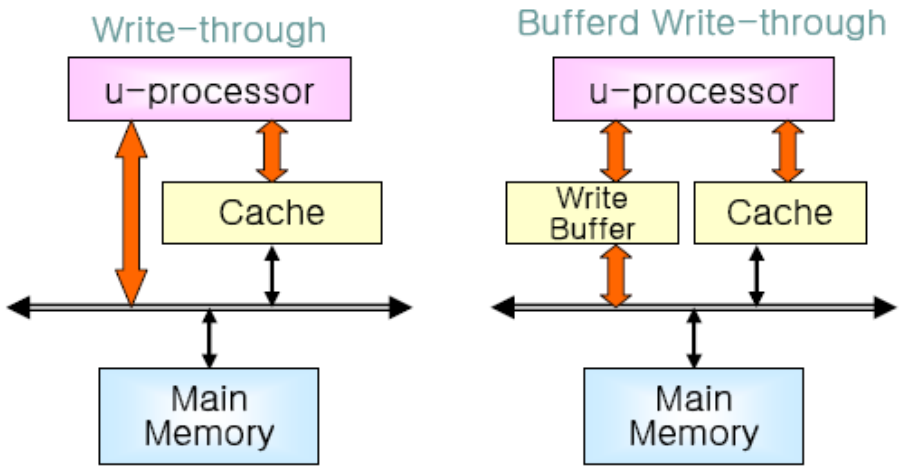
- Cache performance measurement
  - Hit rate = (Hit count / memory request count) * 100
  - Miss rate = 100 – Hit rate

Cache policy

- Policies
  - Write policy
  - Replacement policy
  - Allocation policy

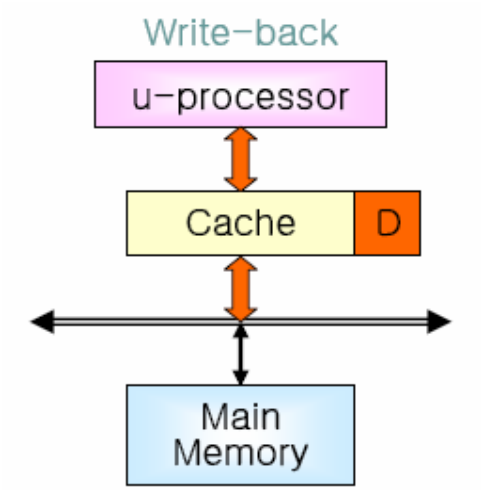
Cache policy

- Write policy
  - Write-through
    - Write : Memory is updated at the same time of cache updating
    - Characteristics
      - Always maintain cache coherence
      - Increase bus traffic, Performance degradation
      - Write no-allocate
    - Enhanced by Buffed write-through
      - Reducing CPU write cycle time
      - When frequent writing -> Needs Second level Cache



Cache policy

- Write policy
  - Write-back (Copy-back)
    - Write : Memory is updated at line replacement
    - Characteristics
      - Hardware overhead: Dirty bit is maintained
      - Write allocate



Cache policy

- Replacement policy
  - Motivation
    - **Victim block** : When a new block is to be brought into the cache and all the positions that it may occupy are full, the cache controller must decide which of the old blocks to overwrite
    - **Eviction** : the cache controller must update victim block must update to the main memory before overwrite it
  - Replacement strategy
    - Direct-mapped cache
      - No replacement strategy exists
    - Associative and set-associative caches
      - Round-robin strategy
        - Target : Next line
      - Pseudo-random strategy
        - Target : Random line
        - Using victim counter : Select a block has highest counter to victim block

Cache policy

- Allocation policy
  - Allocate cache line when occurred cache miss
  - Read-allocated
    1. Update dirty cache line to main memory
    2. allocate cache line
    3. read from main memory
  - Read-write-allocate
    - On read operation, use read-allocated policy
    - On write operation, allocate cache line if victim cache line is not valid
      otherwise, update cache line and allocate

Coprocessor 15 and cache

- CP15
  - Cache control register for ARM core with cache

Function	Base register	Sub register	Op-code
Cache clean & flush	C7	C5, C6, C7, C10, C13, C14	0, 1 , 2
Drain buffer write	C7	C10	4
Cache lockdown	C9	C0	0, 1
Round-robin replacement	C15	C0	0

Cache flush & Cache clean

- **Cache operation**
  - **Flush**
    - Clear valid bit to zero to invalidate data cache
    - For data block initialization
  - **Clean**
    - Force update dirty block to main memory
    - For maintain consistence cache with main memory
  - CP15 control cache operation using cache control registers
    - I-cache flush & Clean (C5)
    - D-cache flush(C6) / clean (C7)
    - I & D cache flush & clean (C14)

Cache lockdown

- Lockdown
  - Mean : Do not eviction
  - Effect
    - Better performance for frequently used instruction or data
    - Smaller usable cache size
  - Target
    - Vector interrupt table
    - Interrupt service routine
    - Exclusive critical algorithms
    - Global variable which used frequently

Cache & SW performance

- Effective SW programming consider cache
  - Using write buffer
    - Shorten average memory access time
  - Consider spatial locality
    - Increase hit rate
  - suitable data structure for cache block size
    - For effective data transfer
  - Avoid using linked list
    - Linked list can prompt cache miss

Memory Protection Units

KEUK HAN KWON
khkwon@enb.hanyang.ab.kr

목차

- 보호 영역

- MPU, 캐시, 쓰기 버퍼의 초기화
  - 영역의 크기와 위치 정의
  - 접근 권한
  - 캐시 및 쓰기 버퍼의 속성 설정
  - 영역 및 MPU 활성화

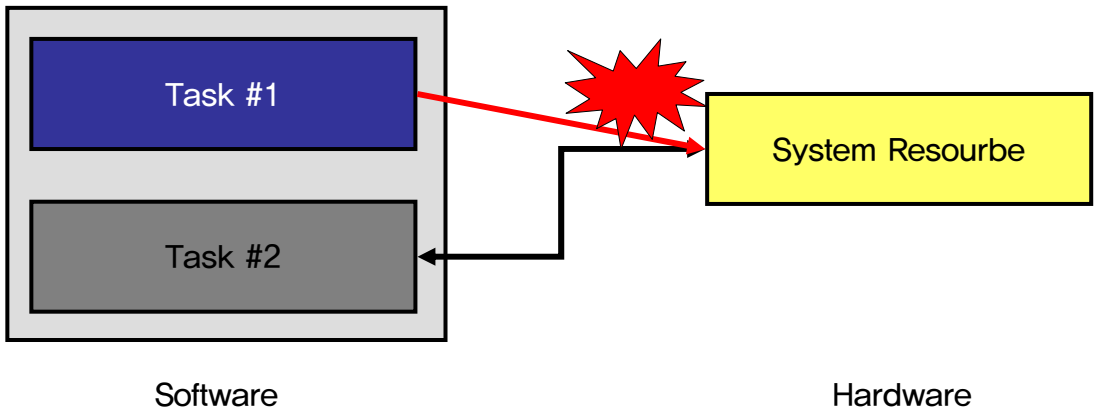
Overview

- Protection
  - 원치 않은 액세스로부터 시스템 자원과 태스크들을 막아주는 것

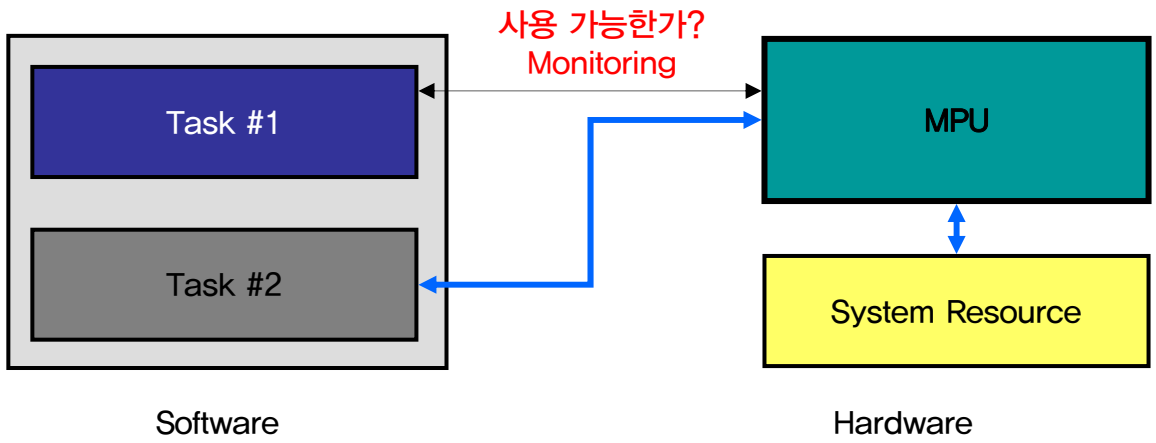
- 시스템 자원의 액세스 제어 방법
  - 비보호 (Unprotected)
    - 소프트웨어에만 의존하여 시스템 자원을 보호
  - 보호 (Protected)
    - 하드웨어와 소프트웨어에 의존하여 시스템 자원을 보호

Overview

- 비보호 시스템 (Unprotected)
  - 동작중에 메모리와 주변 장치를 사용하도록 강요하는 전용 하드웨어가 없음
  - 시스템 자원을 액세스하기 위해 각 태스크들이 다른 모든 태스크들과 협력 해야 함



- 보호 시스템 (Protected)
  - 시스템 자원의 액세스를 확인하고 제한하는 전용 하드웨어를 가지고 있음
    - 자원의 소유권
  - 태스크들은 운영체제에 의해 정의되고 하드웨어에 의해 강요된 규칙에 따라 동작
    - 프로그램에게 하드웨어 레벨에서 자원들을 모니터링 하고 제어하는 특별한 권한을 줌
  - 한 태스크가 또 다른 태스크의 자원을 사용하지 못하도록 사전에 예방



보호 시스템 (Protected)

- 모니터링을 필요로 하는 주요 자원
  - 메모리 시스템, 주변 장치
  - ARM 주변 장치는 일반적으로 메모리에 매핑되어 있기 때문에 MPU는 두 자원들을 보호하기 위해 동일 한 방법 사용

- 영역 (Region)
  - 시스템 보호를 위해 영역을 이용
  - 메모리 영역과 관련되어 있는 속성들의 집합
  - 프로세서 코어는 몇몇의 CP15 레지스터 안에 이 속성값들을 저장하고 있으며, 각 영역은 0에서 7까지의 번호로 규정
  - 메모리 경계는 **시작주소**와 **크기**의 속성값을 사용하여 설정
  - 운영체제는 이 영역에게 **접근권한**, **캐시** 및 **쓰기 버퍼** 정책과 같은 추가 속성을 할당
  - 액세스는 RW, RO, No-access 중 하나로 설정
  - 프로세서 모드를 기초로 한 추가 권한을 할당할 수 있음
  - 캐시 및 쓰기 버퍼 속성을 제어하는 캐시 쓰기정책을 가짐

- 프로세서가 주 메모리 안의 영역을 액세스할 때 MPU는 어떻게 동작할지를 결정하기 위해 영역 접근권한 속성과 현재 프로세서 모드를 비교
  - 요청 사항이 액세스 영역을 만족하면 코어는 주 메모리를 읽거나 쓸 수 있도록 허락
  - 요청이 메모리 액세스 금지 구역이라면 MPU는 **Abort 신호**를 발생

보호 영역

- ARM740T, ARM946E-S, ARM1026-S
  - 8개의 보호 영역
  - 명령어 영역, 데이터 영역이 통합
  - 접근 권한과 캐시 정책이 명령어 메모리와 데이터 메모리에 동일하게 적용
  - 영역은 코어의 아키텍처에 상관없이 독립적
  - 각 영역은 0에서 7사이의 번호로 할당됨

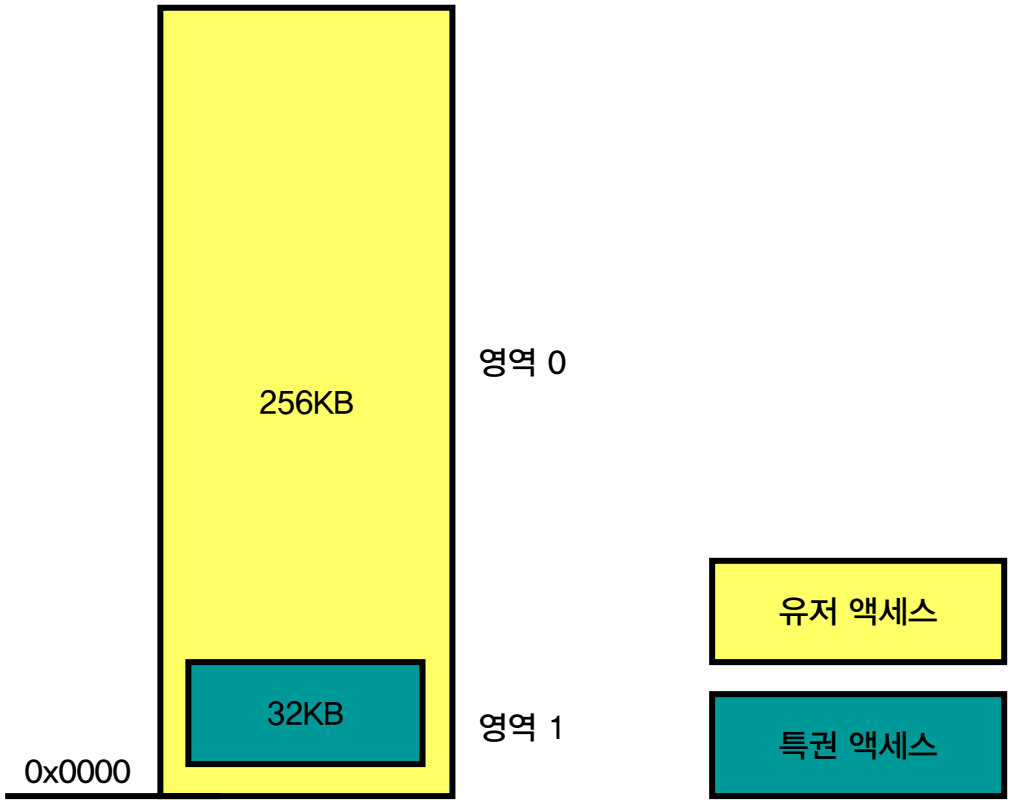
- ARM940T
  - 16개의 보호 영역
  - 명령어 메모리와 데이터 메모리를 제어하기 위한 분리된 영역을 갖기 때문에, 코어는 명령어와 데이터 영역에 대해 다른 영역 크기와 시작 주소를 할당
  - 캐시 코어에서 추가로 8개의 영역을 갖음
  - 영역번호는 데이터 영역과 명령어 영역의 쌍을 이루며, 0에서 7까지 할당 됨

영역을 제어하기 위한 규칙

- 영역은 다른 영역에 오버랩 될 수 있음
- 영역은 영역에 할당되어 있는 특권에 독립적인 우선순위가 할당
- 영역이 오버랩되면 가장 높은 우선순위 번호를 가진 영역 속성이 다른 영역에 대해 우선순위를 갖는다. 우선순위는 오버랩된 영역내의 주소에 대해서만 적용
- 영역의 시작 주소는 그 크기의 배수이어야 함
- 영역의 크기는 4KB와 4GB 사이에 있는 2의 거듭 자리가 될 수 있음
- 정의된 영역 밖에 있는 주 메모리의 영역을 액세스 하면 Abort를 발생
  - 코어가 명령어를 fetbh하고 있었다면, prefetbh abort를 발생, 데이터를 위한 메모리 요청이었다면 data abort를 발생

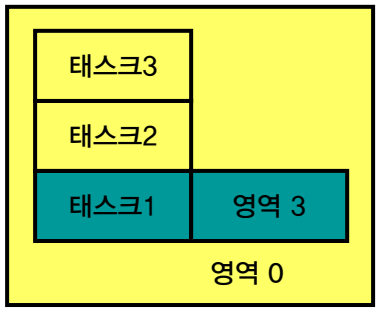
오버랩 영역

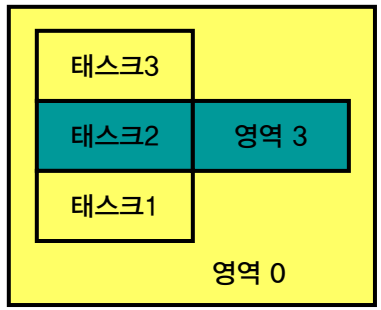
- 한 영역에 할당되어 있는 메모리 공간의 일부가 또 다른 영역에 할당되어 있는 메모리 공간 안에 있는 경우

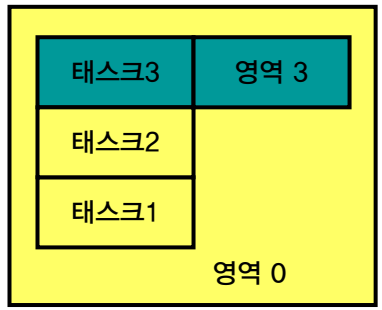


배경 영역

- 큰 메모리 영역에 동일한 속성을 할당하는 데 사용되는 가장 낮은 우선 순위의 영역







MPU, 캐시, 쓰기 버퍼의 초기화

MPU를 제어하는 코프로세서 레지스터

- 영역 크기와 위치
  - 기본 레지스터 : C6
  - 명령어 및 데이터 영역의 크기와 위치를 정의한
- 영역 접근권한
  - 기본 레지스터 : C5
  - 각 영역에 대한 접근권한을 설정
- 영역 쓰기 버퍼 속성
  - 기본 레지스터 : C3
  - 영역을 위한 캐시 속성을 지정
- 영역 캐시 속성
  - 기본 레지스터 : C2
  - 영역을 위한 쓰기 버퍼 속성을 지정
- 시스템제어
  - 기본 레지스터 : C1
  - 캐시와 MPU를 활성화(Enable) /비활성화(Disable)

영역의 크기와 위치 정의

- ARM740T, ARM946E-S, ARM1026EJ-J 프로세서는 8개의 영역 존재
- CP15:b6:bX 안의 보로 레지스터에 값을 설정하여 한 영역의 크기와 위치를 결정

영역의 크기와 위치를 설정해주는 bP15:b6 레지스터 포멧

필드 이름	비트필드	설명
베이스 주소	[31:12]	4KB 이상의 주소는 [5:1]에 나타난 크기의 배수
SBZ	[11:6]	0
N	[5:1]	영역의 크기는 $2^{N+1,}$ (11<= N <= 31)
E	[0]	1 = Enable, 0 = Disable

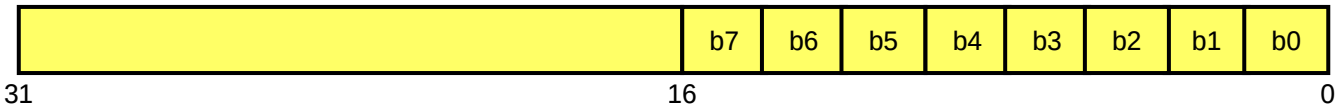
CP15:b6:b0에서 CP15:b6:b7까지의 레지스터의 비트필드

접근권한

특권 모드의 태스크	유저 모드의 태스크	표준 AP 값 인코드	확장 AP 값 인코드
No access	No access	00	0000
Read/write	No abbess	01	0001
Read/write	Read only	10	0010
Read/write	Read/write	11	0011
Unpredictable	Unpredictable	-	0100
Read only	No abbess	-	0101
Read only	Read only	-	0110
Unpredictable	Unpredictable	-	0111
Unpredictable	Unpredictable	-	1000~1111

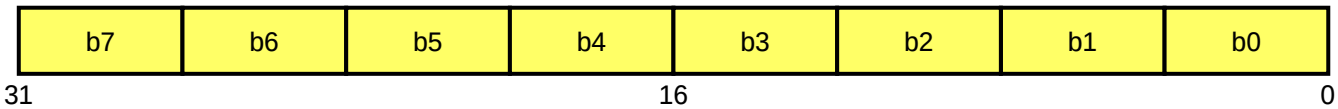
CP15:c5:c0:0 표준 명령어 영역 AP

CP15:c5:c0:1 표준 데이터 영역 AP



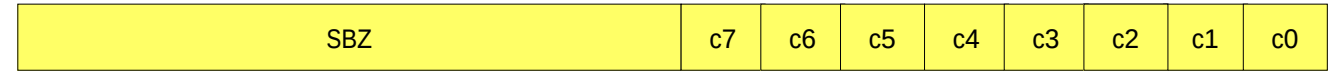
CP15:c5:c0:2 확장 명령어 영역 AP

CP15:c5:c0:3 확장 데이터 영역 AP



캐시 및 쓰기 버퍼 속성 설정

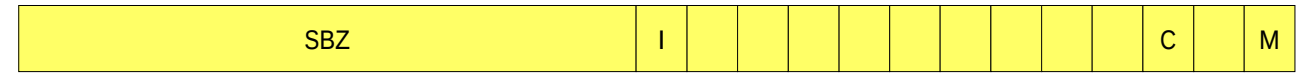
CP15:b2:b0:0 : 데이터 캐시
CP15:b2:b0:1 : 명령어 캐시



CP15:b3:b0:0 : 쓰기 버퍼

명령어 캐시		데이터 캐시		
캐시 비트 CP15:c2:c0:1	영역 속성	캐시 비트 CP15:c2:c0:0	캐시 비트 CP15:c3:c3:c0:0	영역 속성
0	Not cached	0	0	NCNB (not cached, not buffered)
1	cached	0	1	NCB (Not cached, buffered)
		1	0	WT (cached, writethrough)
		1	1	WB (cached, writebback)

영역 및 MPU 활성화



컨트롤 레지스터에서 메모리 보호 장치 제어 비트

비트	활성화된 기능	값
0	MPU	0=Disable, 1=Enable
2	데이터 캐시	0=Disable, 1=Enable
12	명령어 캐시	0=Disable, 1=Enable

MPU 시스템의 예

하드웨어 특징

- MPU를 가지고 있는 ARM코어
- 0x0에서 시작하고 0x40000으로 끝나는 256KB의 물리 메모리
- 0x10000000에서 0x12000000까지의 몇 메가 바이트 이상을 차지하는 메모리 매칭된 몇 가지 주변장치

소프트웨어 구성 요소

- 시스템 소프트웨어는 크기가 64KB 이내로, Exception을 지원하기 위해 Vector Table, Exception Handler, Data Cache를 포함
- 시스템 소프트웨어는 User모드로부터 Access되어서는 안 된다. 즉, User모드는 이 영역 안에 있는 코드나 데이터를 액세스하기 위해서 시스템 콜을 사용해야 함
- 크기가 64KB 이내인 공유 소프트웨어가 있다. 이것은 유저 태스크 사이에 메시지를 보내기 위해서 공통으로 사용되는 라이브러리와 데이터 공간을 포함
- 시스템에서 독립적인 기능을 제어하는 3개의 유저 태스크가 있다. 이 태스크들은 32KB 이내의 크기를 가지며, 이것들은 실행될 때 다른 2개의 태스크에 의한 액세스로부터 보호되어야 함

보호 시스템 예의 메모리 맵

기능	액세스 레벨	시작주소	크기	영역
메모리 매핑된 주변 장치를 보호	시스템	0x10000000	2MB	4
보호 시스템	시스템	0x00000000	4GB	1
공유 시스템	유저	0x00010000	43KB	2
유저 태스크 1	유저	0x00020000	32KB	3
유저 태스크 2	유저	0x00028000	32KB	3
유저 태스크 3	유저	0x00030000	32KB	3

보호 시스템 예의 영역 할당과 메모리맵

START ADDRESS	SIZE	IAP	DAP	CB
0x10000000	19 (1M)	5 (RONA)	1 (RWNA)	0 (ccb)
0x00028000	14 (32K)	6 (RORO)	3 (RWRW)	6 (CWT)
0x00010000	15 (64K)	6 (RORO)	3 (RWRW)	6 (CWT)

FIQ 스택 베이스
IRQ 스택 베이스
Supervisor 스택 베이스
Undef 스택 베이스
Abort 스택 베이스

The future of the ARM architecture

Juyoung Kim

jykim@enc.hanyang.ac.kr

Outline

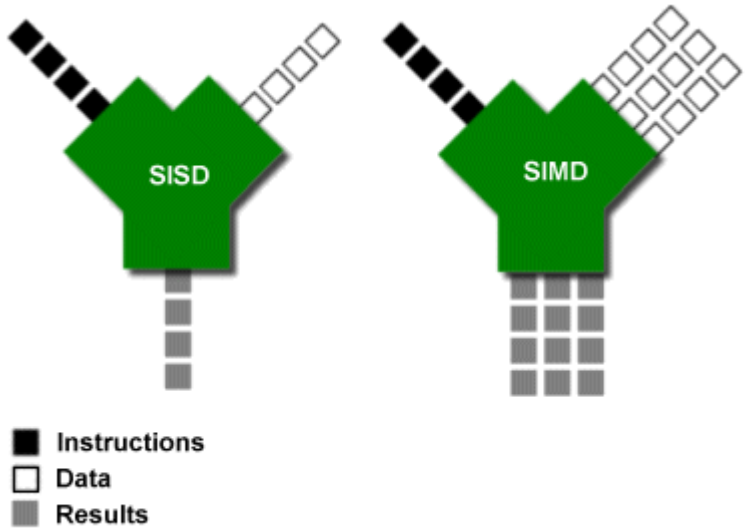
- Introduction
- ARMv6 DSP & SIMD instructions
- ARMv6 system engine & multiprocessor
- Feature of the ARMv6
- The coming advanced technology

Introduction

- Meet to requirements of client
  - Advanced technology would be required :Especially on the markets of mobile computing
    - DSP
    - AV data processing
    - Inter-working between homogeneous system
    - Effective synchronous of multiprocessing system
    - **Power efficiency**

ARMv6 DSP & SIMD instructions

- Advance the DSP processing
  - SIMD (Single Instruction Multiple Data)
    - Strength
      - Higher code density
      - Lower power consumption
    - Weakness
      - Flexibility of data align



ARMv6 DSP & SIMD instructions

- DSP arithmetic operation of ARMv6
- 8, 16 bits operation supported
  - Support signed, unsigned, saturating signed, saturating unsigned instructions
  - Manage overflow & wrap around
  - Processing parallelism
    - 8 bits * 4 operation or 16 bits * 2 operation at once on 32 bits ARM processors
  - GE (Greater or Equal) bits
    - CPSR[19:16]
    - Possible conditional execution
    - Ex)

    SEL Rd, Rn, Rm

    =>

    Rd[31:24] = GE[3]? Rn[31:24] : Rm[31:24]

    Rd[23:16] = GE[2]? Rn[23:16] : Rm[23:16]

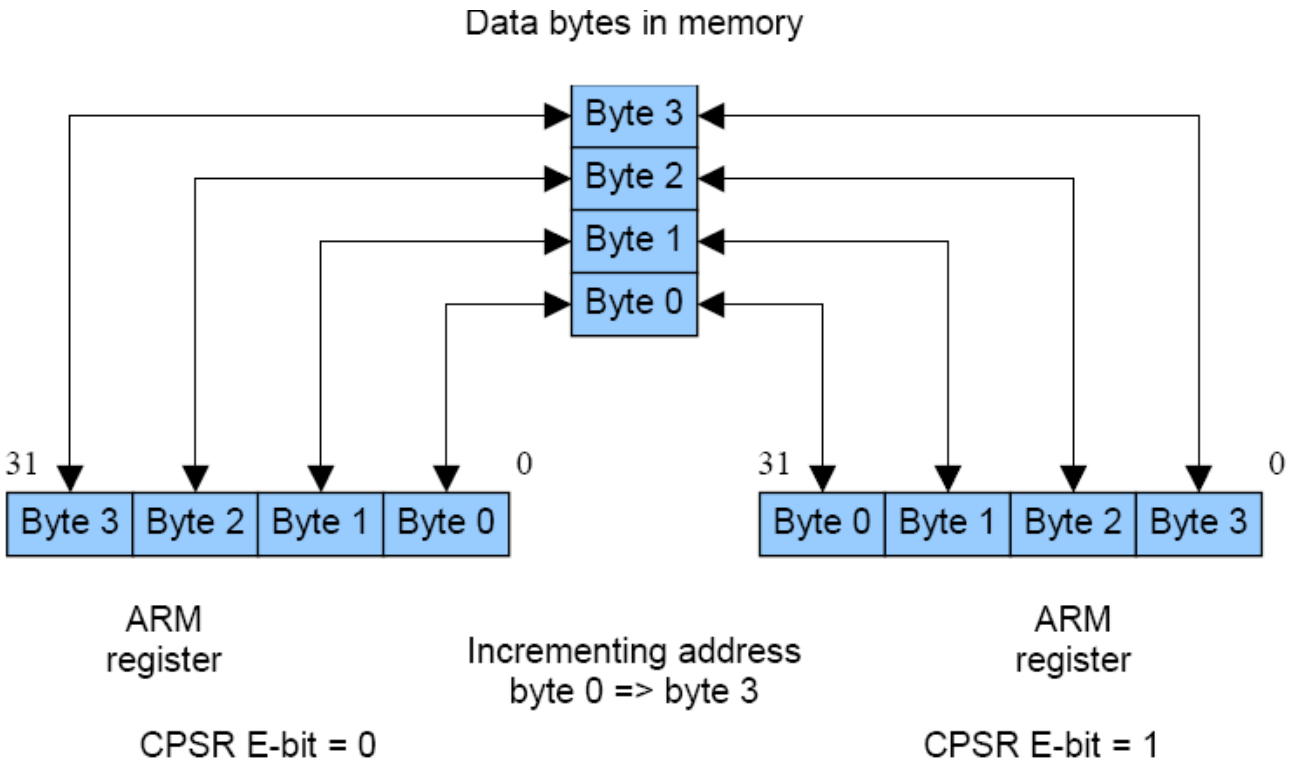
    Rd[15:08] = GE[1]? Rn[15:08] : Rm[15:08]

    Rd[07:00] = GE[0]? Rn[07:00] : Rm[07:00]

ARMv6 DSP & SIMD instructions

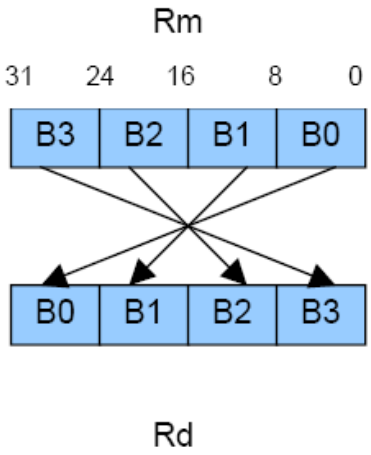
- Instruction compression
  - Compress two 16 bits data to one 32 bits data
- Complex arithmetic operation
  - EX) FFT (Fast Fourier Transform) operation
- Overflow instruction
- SAD (Sum of Absolute Difference) instruction
- Dual 16 bits multiply instruction
- Words multiply instruction
- Encrypt multiply

- Homogeneous endian support
  - Using E flag
    - CPSR[9]

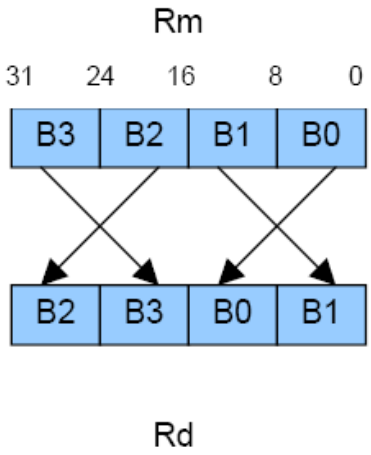


- Exception handling
  - Advanced inversion instruction for stack control

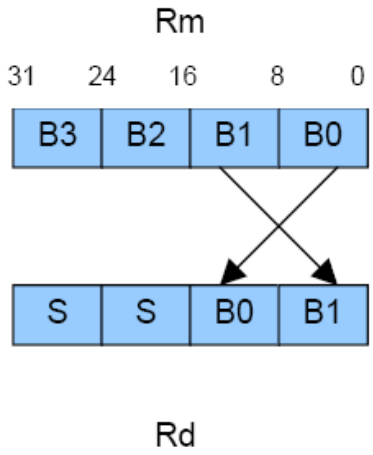
REV{<cond>} Rd, Rm



REV16{<cond>} Rd, Rm



REVSH{<cond>} Rd, Rm



ARMv6 system engine & multiprocessor

- Multiprocessing
  - Problem of SWP instruction
    - Bus wait until semaphore released
    - Bottleneck of performance
  - Advanced synchronous instruction
    - No bus wait
    - LDREX (Load Exclusive)
    - STREX (Store Exclusive)

Feature of the ARMv6

- Pipeline
  - Load / Store
  - Multiply / Add
- Parallel load/store unit (LSU)
- Transition from virtual tag cache to physical tag cache

The coming technology

- TrustZone
  - Found in ARMv6KZ and later application core architecture
  - Provides a low cost alternative to adding an additional dedicated security core to a SoC, by providing two virtual processors backed by hardware based access control.
  - Purpose
    - Security of online market

The coming technology

- TrustZone
  - Found in ARMv6KZ and later application core architecture
  - Provides a low cost alternative to adding an additional dedicated security core to a SoC, by providing two virtual processors backed by hardware based access control.
  - Purpose
    - Security of online market
- Thumb-2
  - Made its debut in the **ARM1156 core**, announced in 2003
  - Extends the limited 16 bits instruction set of Thumb with additional 32 bits instructions

THANK YOU