Computer Architecture

Instruction: Language of the Computer
Part 1. ARM 명령어 기초

Kyusik Chung
kchung@ssu.ac.kr

2장에서 무엇을 배우나 ?

- **ARM** 어셈블리 언어 **(ARM 32 bit CPU** 명령어**)**
- **ARM** 어셈블리 프로그래밍

ARM Assembly Language

ARM 어셈블리 언어

종류	명령어	예	의미	비고
산술	add	ADD r1,r2,r3	r1 = r2 + r3	레지스터 피연산자 3개
	subtract	SUB r1,r2,r3	r1 = r2 - r3	레지스터 피연산자 3개
데이터 전송	load register	LDR r1, [r2,#20]	r1 = Memory[r2 + 20]	워드를 메모리에서 레지스터로
	store register	STR r1, [r2,#20]	Memory[r2 + 20] = r1	워드를 레지스터에서 메모리로
	load register halfword	LDRH r1, [r2,#20]	r1 = Memory[r2 + 20]	하프워드를 메모리에서 레지스터로
	load register halfword signed	LDRHS r1, [r2,#20]	r1 = Memory[r2 + 20]	하프워드를 메모리에서 레지스터로
	store register halfword	STRH r1, [r2,#20]	Memory[r2 + 20] = r1	하프워드를 레지스터에서 메모리로
	load register byte	LDRB r1, [r2,#20]	r1 = Memory[r2 + 20]	바이트를 메모리에서 레지스터로
	load register byte signed	LDRBS r1, [r2,#20]	r1 = Memory[r2 + 20]	바이트를 메모리에서 레지스터로
	store register byte	STRB r1, [r2,#20]	Memory[r2 + 20] = r1	바이트를 레지스터에서 메모리로
	swap	SWP r1, [r2,#20]	r1 = Memory[r2 + 20], Memory[r2 + 20] = r1	레지스터와 메모리 간의 원자적 교환
	mov	MOV r1, r2	r1 = r2	값을 레지스터로 복사
논리	and	AND r1, r2, r3	r1 = r2 & r3	레지스터 피연산자 3개; 비트 대 비트 AND
	or	ORR r1, r2, r3	r1 = r2 | r3	레지스터 피연산자 3개; 비트 대 비트 OR
	not	MVN r1, r2	r1 = ~ r2	레지스터 피연산자 3개; 비트 대 비트 NOT
	logical logical shift left (optional operation)	LSL r1, r2, #10	r1 = r2 << 10	상수만큼 좌측 자리이동
	logical shift right (optional operation)	LSR r1, r2, #10	r1 = r2 >> 10	상수만큼 우측 자리이동
조건부 분기	compare	CMP r1, r2	cond. flag = r1 - r2	조건부 분기를 위한 비교
	branch on EQ, NE, LT, LE, GT, GE, LO, LS, HI, HS, VS, VC, MI, PL	BEQ 25	if (r1 == r2) go to PC + 8 + 100	조건 테스트; PC-상대 주소
무조건 분기	branch (always)	B 2500	go to PC + 8 + 10000	분기
	branch and link	BL 2500	r14 = PC + 4; go to PC + 8 + 10000	프로시저 호출용

21개의
명령어 목록

Bubble Sort in C

- **Non-leaf (calls swap)**

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i ++ ) {
    for (j = i - 1;j >= 0; j --) {
       if (v[j] > v[j + 1]) swap(v,j);
    }
  }
}
```

C 프로그래밍 예제

Sort: Register allocation and saving registers

Register allocation

```
v       RN 0        ; 1st argument address of v
n       RN 1        ; 2nd argument index n
i       RN 2        ; local variable i
j       RN 3        ; local variable j
vjAddr  RN 12       ; to hold address of v[j]
vj      RN 4        ; to hold a copy of v[j]
vj1             RN 5            ; to hold a copy of v[j+1]
vcopy   RN 6        ; to hold a copy of v
ncopy   RN 7        ; to hold a copy of n
```

Saving registers

```
sort:   SUB   sp,sp,#20          ; make room on stack for 5 registers
        STR   lr, [sp, #16]      ; save lr on stack
        STR   ncopy, [sp, #12]   ; save ncopy on stack
        STR   vcopy,  [sp, #8]   ; save vcopy on stack
        STR   j,  [sp, #4]       ; save j on stack
        STR   i,  [sp, #0]       ; save i on stack
```

Bubble Sort를 위한 ARM 어셈블리 프로그래밍1

Sort: Procedure body

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i ++)
    for (j = i - 1; j >= 0;j --)
      if (v[j] > v[j + 1]) swap(v,j);
}
```

Move parameters		MOV	vcopy, v	; copy parameter v into vcopy (save r0)
		MOV	ncopy, n	; copy parameter n into ncopy (save r1)
Outer loop		MOV	i, #0	; i = 0
	for1tst:	CMP	i, n	; if i ≥ n
		BGE	exit1	; go to exit1 if i ≥ n
Inner loop		SUB	j, i, #1	; j = i - 1
	for2tst:	CMP	j, #0	; if j < 0
		BLT	exit2	; go to exit2 if j < 0
		ADD	vjAddr, v, j, LSL #2	; reg vjAddr = v + (j * 4)
		LDR	vj, [vjAddr,#0]	; reg vj = v[j]
		LDR	vj1, [vjAddr,#4]	; reg vj1 = v[j + 1]
		CMP	vj, vj1	; if vj ≤ vj1
		BLE	exit2	; go to exit2 if vj ≤ vj1
Pass parameters and call		MOV	r0, vcopy	; first swap parameter is v
		MOV	r1, j	; second swap parameter is j
		BL	swap	; swap code shown in Figure 2.23
Inner loop		SUB	j, j, #1	; j -= 1
		B	for2tst	; branch to test of inner loop
Outer loop	exit2:	ADD	i, i, #1	; i += 1
		B	for1tst	; branch to test of outer loop

Bubble Sort를 위한 ARM 어셈블리 프로그래밍2

Sort: Restoring registers and return

```
exit1:  LDR   i,  [sp, #0]          ; restore i from stack
        LDR   j,  [sp, #4]          ; restore j from stack
        LDR   vcopy,  [sp, #8]      ; restore vcopy from stack
        LDR   ncopy, [sp, #12]      ; restore ncopy from stack
        LDR   lr, [sp, #16]         ; restore lr from stack
        ADD   sp,sp,#20             ; restore stack pointer
```

Procedure return

```
        MOV   pc, lr                ; return to calling routine
```

Bubble Sort를 위한 ARM 어셈블리 프로그래밍3

고급언어프로그래밍 대 어셈블리프로그래밍 비교

- **(예) 자동차 운전방법에서 AUTO와 STICK 비교**
  - AUTO 방식 운전에서는 주행중 기어 change 를 자동으로 해줌
  - STICK 방식 운전에서는 주행중 기어 change 를 수동으로 해야 함

  - STICK 방식의 운전은 불편하나 이점이 있다. 연비가 좋다. 가속이 빠르다 (성능이 뛰어나다).
  - **고급언어프로그래밍(C, C++, 파이썬,...)은 AUTO 방식에 해당하고 어셈블리프로그래밍은 STICK 방식에 해당함**
  - 고급언어프로그램은 컴파일러 도움을 받아서 machine code로 변환. 어셈블리프로그램은 어셈블러 도움을 받아서 machine code로 변환. 어셈블러에서 명령어를 machine code로 변환하는 것은 1:1 변환하는 단순 작업에 해당함. 반면 컴파일러가 고급언어프로그램을 machine code로 변환하는 것은 최적화가 요구되는 복잡한 작업이다.
  - 성능에 민감한 코드는 어셈블리 프로그램으로 작성하는 게 유리

어셈블리프로그래밍을 위한 **Lab** 내용(**5**주차 ~ **9**주차)

0. 어셈블리프로그래밍 개발환경 구축 및 디버거 사용법 학습

1. Factorial 함수를 어셈블리 코드로 구현 및 **test**

2. 디버거를 이용한 **C pointer** 동작 및 **stack overflow test**

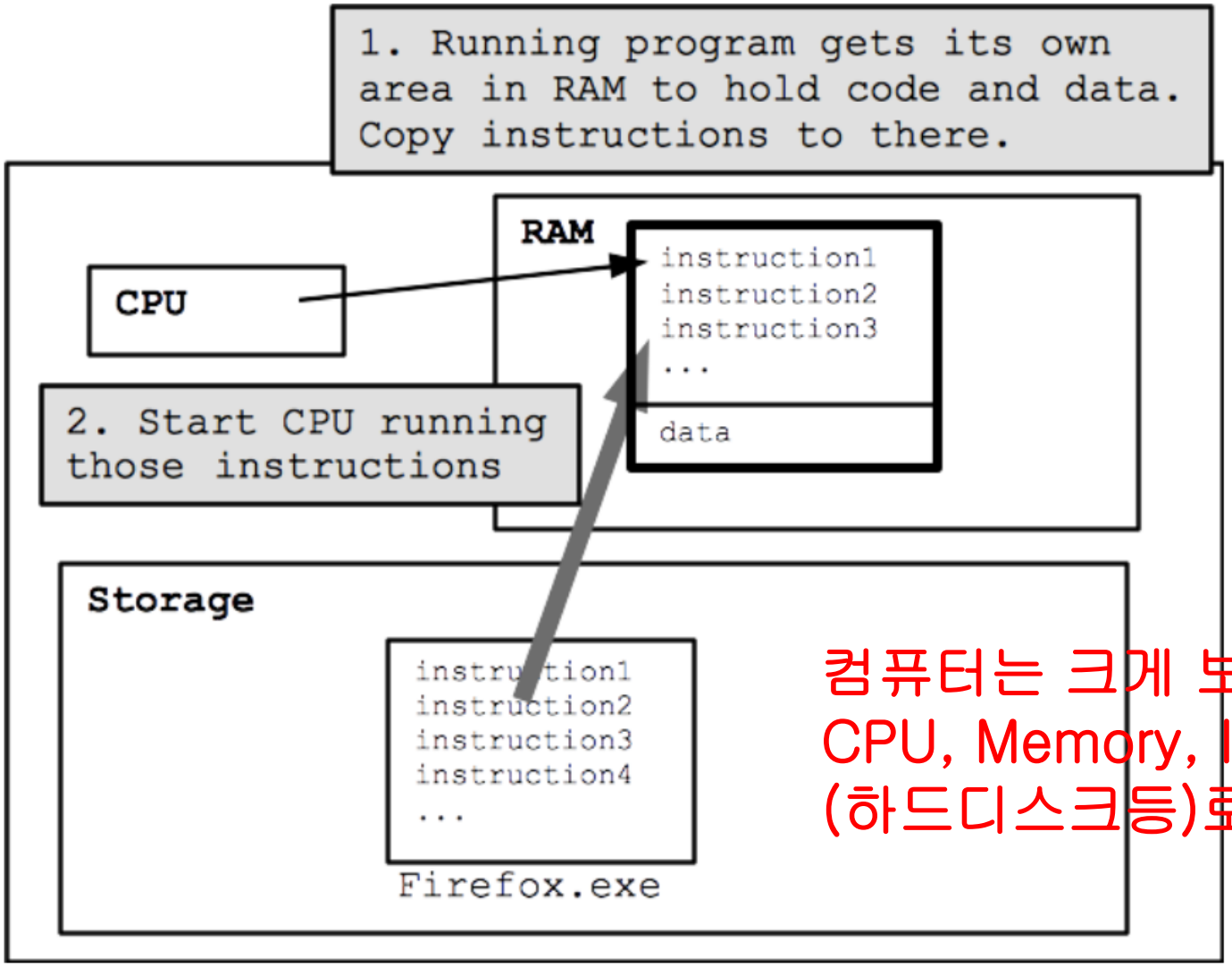
3. Bubble sorting을 위한 어셈블리 코드 분석 및 **test**

4. 매트릭스 곱셈을 어셈블리 코드로 구현 및 **test**

고급언어 프로그램 실행 과정

- 컴퓨터에서 코드 수행
  1) 메모리에서 고급언어 프로그램을 컴파일러 또는 인터프리터를 이용하여 기계코드(이진수)로 변환한뒤 저장장치(하드디스크)에 저장한다(수행파일).
     - 프로그램 내부를 보면 text 영역(명령어들 집합)과 data 영역 (컴퓨터에서 연산시 사용하는 데이터 또는 결과값들이 저장)으로 나눌 수 있다.
  2) 저장장치에서부터 해당 기계코드(수행파일이라 부름)를 메모리에 load한다.
  3) 메모리에 저장된 기계코드를 대상으로 CPU는 명령어를 하나씩 읽어와서(fetch) 순차적으로 수행(execute)한다.
     - 명령어 수행시(연산도중) 데이터가 필요하면 메모리내의 데이터 영역에서 데이터를 읽어온다. 연산후 결과값을 메모리내 데이터 영역에 저장한다.
  4) 최종 결과값을 출력함으로써 코드 수행이 종료된다.

프로그램 실행(1/2)

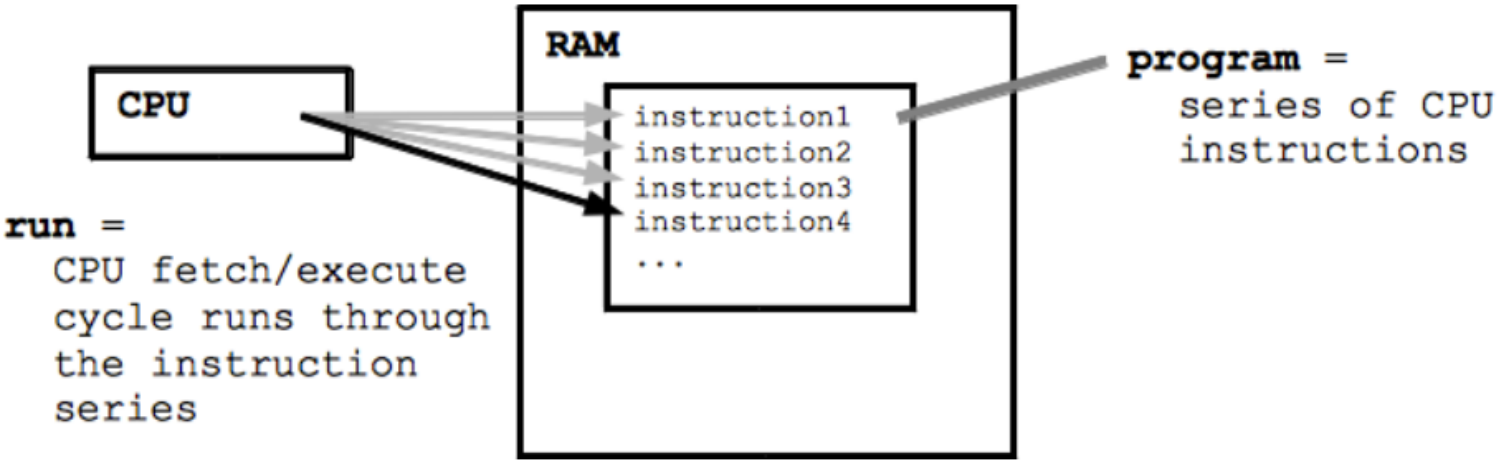


컴퓨터는 크게 보면
CPU, Memory, IO장치
(하드디스크등)로 구성

프로그램 실행(2/2)

How Does a Program Run?

- CPU runs a "fetch/execute cycle"
  -fetch one instruction in sequence
  -execute (run) that instruction, e.g. do the addition
  -fetch the next instruction, and so on
- Run a program = Start CPU running on its 1st instruction
  it runs down through all of the machine code, running the program
  the program will have instructions like "return to step 3" to keep it running
- Super simple machine code instructions run at the rate of 2 billion per-second



고급언어 프로그램에서의 변수 ?(1/2)

- 고급언어 프로그램을 어셈블리코드 또는 기계코드로 변환하는 과정에서 변수는 **main memory**주소로 대체된다.
- 메모리주소:
  - CPU가 명령어를 읽어올 때, 데이터 읽어올 때, 데이터 저장할 때 메모리에 주소를 사용하여 요청. 예를 들면, 메모리 1000번지(명령어)를 읽어주세요. 메모리 2000번지(데이터)를 읽어주세요. 메모리 3000번지에 데이터를 저장해주세요.
- 변수:
  - 고급언어에서 데이터를 나타낼 때 사용하는 용어
  - 수학에서 말하는 변수와 비슷
  - 메모리 특정 번지에 있는 데이터를 추상화(단순화)하여 표현한 것임.
  - (예)
    - 고급언어에서의 표현: x=7 -> 변수 x에 7을 넣으세요.
    - 변환후 기계코드에서의 표현: M[2000]=7 -> 메모리 2000번지에 7을 넣으세요

고급언어 프로그램에서의 변수 ?(2/2)

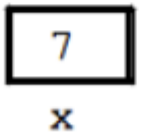
- A "variable" is like a box that holds a value

```
x = 7;
```

- The above line stores the value 7 into the variable named **x**
- Later appearances of **x** in the code retrieve the value from the box
- Using = in this way is called "variable assignment"
- How we use variables in CS101:
  -Assign a value into a variable once
  -Then use that variable many times below

Code

```
x = 7;
```



여기서 box는 메모리를 의미

Byte/Word

- **1 byte = 8 bits**
- **1 word = ? bits**
  - 16 비트 CPU에서는 16 bits
  - 32 비트 CPU에서는 32 bits
  - 64 비트 CPU에서는 64 bits
- **32** 비트 **CPU**와 **64** 비트 **CPU**의 차이점
  - 레지스터 길이가 다르다. 32 비트 CPU에서는 하나의 레지스터 길이가 32 bits. 64 비트 CPU에서는 하나의 레지스터 길이가 64 bits.
  - 메모리에서 CPU로 전송하는 최대 데이터 길이가 다르다. 32 비트 CPU에서는 32bits, 64 비트 CPU에서는 64 bits
- 명령어**(instruction)** 길이
  - ARM: 32 비트 CPU 경우 명령어크기는 16 또는 32 bits, 64 비트 CPU 경우 명령어크기는 32 bits
  - Intel: 32비트 CPU 경우 명령어크기는 8비트 부터 120비트까지 다양, 64비트 CPU 경우 명령어크기는 41비트

32비트 CPU에서 memory addressing

- **byte addressing**이 기본 (본 강의에서 사용)
  - 메모리 1번지에 1 byte가 들어있다고 가정
  - 16 비트, 32 비트, 64 비트 CPU 종류에 무관하게 address 는 지정
  - Byte 단위 메모리주소 0, 1, 2, 3, ... 각 주소에는 1 byte 데이터가 저장
  - CPU가 보내는 메모리주소가 32비트일경우 2^{32} = 4GB(Giga Byte)를 의미
- **Word addressing**을 사용하는 경우
  - 32 비트 CPU를 사용하는 경우 1 word = 32 비트 = 4 byte
  - Word 기준 메모리주소 0, 1, 2, 3, ... 각 주소에는 4 byte 데이터가 저장
  - Word 0은 Byte 주소 0, 1, 2, 3을 묶은 그룹. Word 1은 Byte 4, 5, 6, 7을 묶은 그룹.
  - Byte 단위 메모리주소가 32비트일경우 2^{32} GB= 4GB(Giga Byte). 이를 word 단위 메모리주소로 표현하면 메모리주소가 30비트에 해당 (1 word는 4개의 byte를 차지). 2^{30} GW= 1GW(Giga Word).

C 코드 실행 예제

- (예) 한줄짜리 C 코드 Z=X+Y (각 변수는 32비트 integer 유형)코드가 컴파일되면 아래의 기계어 (instruction, 명령어)들로 바뀐다.
  - 컴파일러는 변수 X, Y, Z를 위해 메모리번지 1000, 1004, 1008번지를 각각 할당한다고 가정
- 변환된 기계어
  - LDR R1, M[1004]: 메모리 1004 데이터(X)를 R1으로 복사
  - LDR R2, M[1008]: 메모리 1008 데이터(Y)를 R2로 복사
  - ADD R3, R1, R2: R1과 R2 더한 결과를 R3에 저장
    (Ch1-Part2 P27 회로에 의해 수행되는 동작)
  - STR M[1000], R3: R3 값을 메모리 1000(Z) 에 저장

2장 Contents

Part 1: ARM 명령어 기초

- **ARM** 명령어 종류
- **ARM** 내부구조
- 숫자표현
- **CPU** 연산후 상태 비트 계산

Part 2:ARM assembly programming

- **ARM** 명령어 사용법 소개

Part 3: 코드 최적화 및 **ARM assembly program** 예제

- 간단한 코드 최적화
- 함수호출
- **Sorting** 예제

2장-Part 1 Contents

Part 1: ARM 명령어 기초**(ARM 32bit CPU** 기준**)**

- **ARM** 명령어 종류
- **ARM** 내부구조
- 숫자표현
- **CPU** 연산후 상태 비트 계산

명령어의 표현

- **명령어는 이진수로 표현된다.**
  - 기계어 또는 기계어 코드라고 불린다.
- **ARM 명령어들은**
  - 32-bit 명령어 워드로 인코딩된다.
  - 연산 코드, 레지스터 번호 등을 인코딩하는 적은 수의 "명령어 포맷"이 존재한다.
  - (포맷상으로 볼 때) 규칙성을 가진다.

ARM 명령어 예제

- **ADD r1, r2, r3**
  - 레지스터 2번 데이터와 레지스터 3번 데이터 더하여 그 결과를 레지스터 1번에 저장하라는 명령어
  - r2, r3를 source operand, r1을 destination operand라 부름. Operand는 데이터를 지정하는 파트임
  - ADD는 명령어 종류 (operation code, 간단히 opcode라 부름)
  - 명령어는 op-code와 operand들로 구성됨

ARM 피연산자

이름	예	설명
레지스터 16개	r0, r1, . . . , r11, r12, sp, lr, pc	고속 데이터 저장소. ARM에서 산술연산을 하기 위해서는 데이터가 레지스터에 있어야 한다.
메모리 워드 2^{30}개	Memory[0], Memory[4],..., Memory[4294967292]	데이터 전송 명령어만 접근 가능. ARM은 바이트 주소를 사용하므로 이웃 워드 간의 주소는 4씩 차이가 난다. 메모리는 자료구조, 배열 및 스필된(spilled) 레지스터 값들을 기억한다.

ARM 명령어 종류

- 명령어의 종류
  - Data Processing (DP) 명령어
    - register에 있는 값을 처리, 예제: ADD r1, r2, r3
  - Data Transfer (DT) 명령어
    - memory값을 register로 복사하거나 (load) , register값을 memory에 저장 (store). 예제: LDR r1, [r2]
  - Control Transfer (B, BL, SWI) 명령어 : 제어 흐름의 변경
    - Branch, 예제: B 2500
    - branch-and-link : return address를 저장해 둔다.
    - trapping into system code : supervisor calls

ARM 명령어 형식

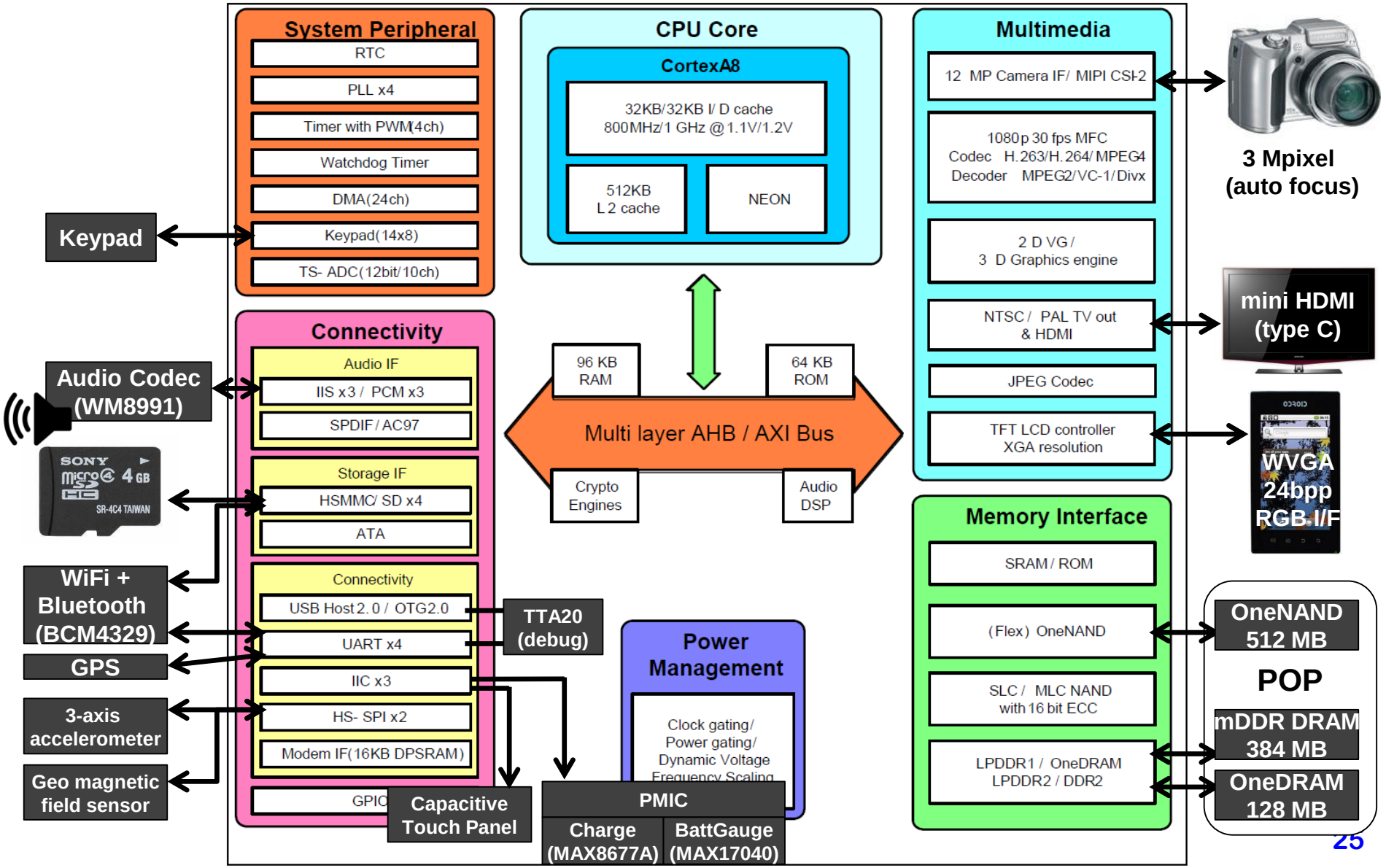
이름	형식	예								비고
필드 크기		4	2	1	4	1	4	4	12	모든 ARM 명령어의 길이는32비트
DP형식	DP	Cond	F	I	Opcode	S	Rn	Rd	Operand2	산술연산 명령어 해석
DT형식	DT	Cond	F	Opcode			Rn	Rd	Offset12	데이터 전송 명령어 해석
필드 크기		4	2	2	24					
BR형식	BR	Cond	F	Opcode	signed_immed_24					B와 BL 명령어

ARM 7 명령어 기본 set

종류	명령어	예	의미	비고
산술	add	ADD r1,r2,r3	r1 = r2 + r3	레지스터 피연산자 3개
	subtract	SUB r1,r2,r3	r1 = r2 - r3	레지스터 피연산자 3개
데이터 전송	load register	LDR r1, [r2,#20]	r1 = Memory[r2 + 20]	워드를 메모리에서 레지스터로
	store register	STR r1, [r2,#20]	Memory[r2 + 20] = r1	워드를 레지스터에서 메모리로
	load register halfword	LDRH r1, [r2,#20]	r1 = Memory[r2 + 20]	하프워드를 메모리에서 레지스터로
	load register halfword signed	LDRHS r1, [r2,#20]	r1 = Memory[r2 + 20]	하프워드를 메모리에서 레지스터로
	store register halfword	STRH r1, [r2,#20]	Memory[r2 + 20] = r1	하프워드를 레지스터에서 메모리로
	load register byte	LDRB r1, [r2,#20]	r1 = Memory[r2 + 20]	바이트를 메모리에서 레지스터로
	load register byte signed	LDRBS r1, [r2,#20]	r1 = Memory[r2 + 20]	바이트를 메모리에서 레지스터로
	store register byte	STRB r1, [r2,#20]	Memory[r2 + 20] = r1	바이트를 레지스터에서 메모리로
	swap	SWP r1, [r2,#20]	r1 = Memory[r2 + 20], Memory[r2 + 20] = r1	레지스터와 메모리 간의 원자적 교환
	mov	MOV r1, r2	r1 = r2	값을 레지스터로 복사
논리	and	AND r1, r2, r3	r1 = r2 & r3	레지스터 피연산자 3개; 비트 대 비트 AND
	or	ORR r1, r2, r3	r1 = r2 | r3	레지스터 피연산자 3개; 비트 대 비트 OR
	not	MVN r1, r2	r1 = ~ r2	레지스터 피연산자 3개; 비트 대 비트 NOT
	logical logical shift left (optional operation)	LSL r1, r2, #10	r1 = r2 << 10	상수만큼 좌측 자리이동
	logical shift right (optional operation)	LSR r1, r2, #10	r1 = r2 >> 10	상수만큼 우측 자리이동
조건부 분기	compare	CMP r1, r2	cond. flag = r1 - r2	조건부 분기를 위한 비교
	branch on EQ, NE, LT, LE, GT, GE, LO, LS, HI, HS, VS, VC, MI, PL	BEQ 25	if (r1 == r2) go to PC + 8 + 100	조건 테스트; PC-상대 주소
무조건 분기	branch (always)	B 2500	go to PC + 8 + 10000	분기
	branch and link	BL 2500	r14 = PC + 4; go to PC + 8 + 10000	프로시저 호출용

명령어 사용법은 Part2에서 소개

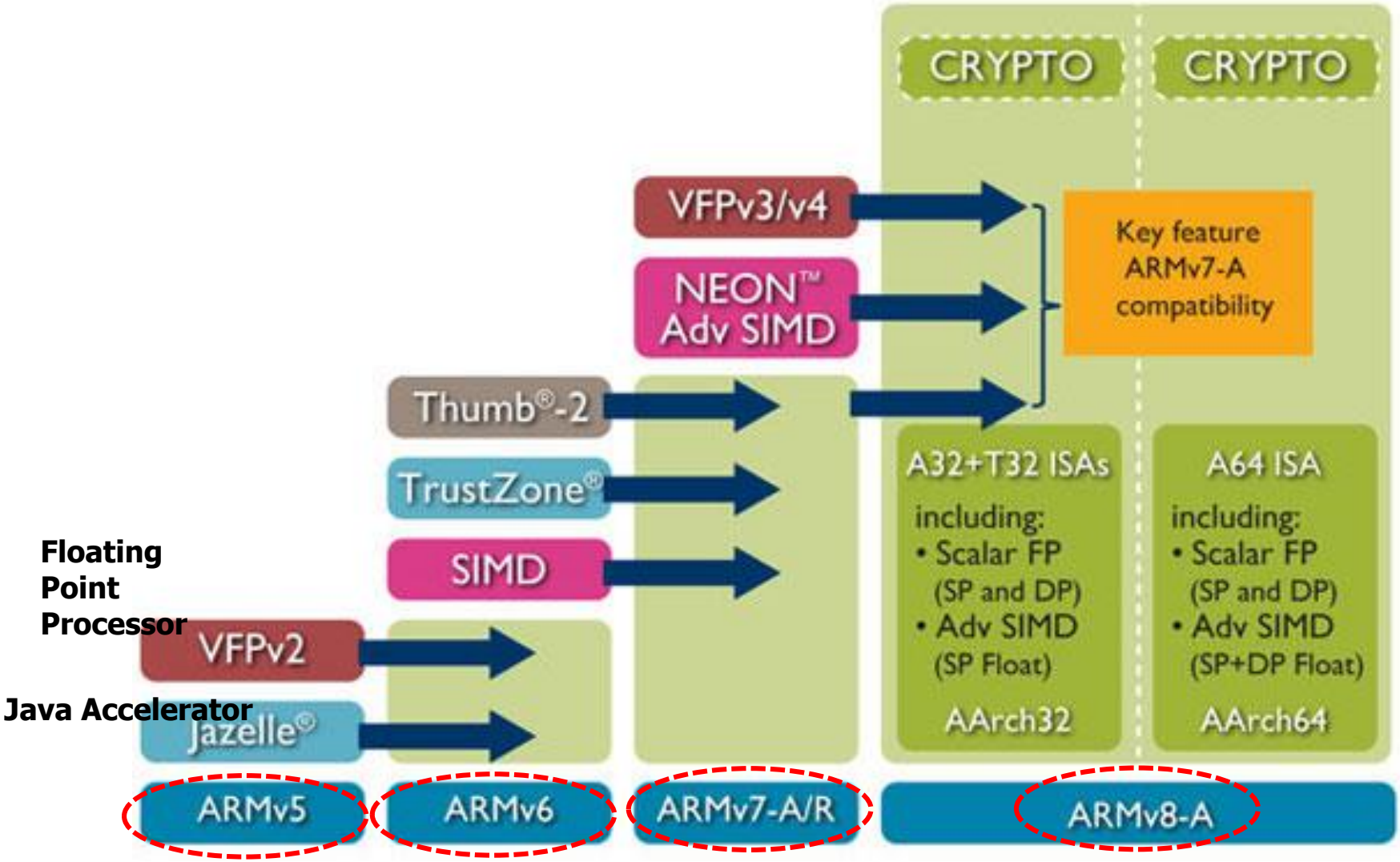
ARM SOC(System On Chip) (예) - Samsung S5PC110



List of ARM microarchitectures

Architecture	Core bit-width	Cores		Profile	References
		ARM Holdings	Third-party		
ARMv8.3-A	64/32	TBA		Application	
ARMv8.2-A	64/32	ARM Cortex-A55,[55] ARM Cortex-A75,[56]		Application	
ARMv8.1-A	64/32	TBA		Application	
ARMv8-R	32	ARM Cortex-R52		Real-time	[42][43][44]
ARMv8-M	32	ARM Cortex-M23,[39] ARM Cortex-M33[40]		Microcontroller	[41]
ARMv8-A	32	ARM Cortex-A32		Application	
ARMv8-A	64/32	ARM Cortex-A35,[45] ARM Cortex-A53, ARM Cortex-A57,[46] [illegible] ARM Cortex-A73[48]	X-Gene, Nvidia Project Denver, AMD K12, Apple Cyclone/Typhoon/Twister/Hurricane/Zephyr, Cavium Thunder X,[49][50][51] Qualcomm Kryo, Samsung M1 and M2 ("Mongoose")[52]	Application	[53][54]
ARMv7E-M	32	ARM Cortex-M4, ARM Cortex-M7		Microcontroller	
ARMv7-R	32	ARM Cortex-R4, ARM Cortex-R5, ARM Cortex-R7, ARM Cortex-R8		Real-time	
ARMv7-M	32	ARM Cortex-M3, SecurCore SC300		Microcontroller	
ARMv7-A	32	ARM Cortex-A5, ARM Cortex-A7, ARM Cortex-A8, ARM Cortex-A9, ARM Cortex-A12, ARM Cortex-A15, ARM Cortex-A17	Qualcomm Krait, Scorpion, PJ4/Sheeva, Apple Swift	Application	
ARMv6-M	32	ARM Cortex-M0, ARM Cortex-M0+, ARM Cortex-M1, SecurCore SC000		Microcontroller	
ARMv6	32	ARM11			
ARMv5TE	32	ARM7EJ, ARM9E, ARM10E	XScale, FA626TE, Feroceon, PJ1/Mohawk		
ARMv4T	32[a 2]	ARM7TDMI, ARM9TDMI, SecurCore SC100			
ARMv4	32[a 2]	ARM8	StrongARM, FA526, ZAP Open Source Processor Core[38]		
ARMv3	32[a 2]	ARM6, ARM7			
ARMv2	32[a 1]	ARM2, ARM250, ARM3	Amber, STORM Open Soft Core[37]		
ARMv1	32[a 1]	ARM1			

ARM Architecture Roadmap



2장-Part 1 Contents

Part 1: ARM 명령어 기초

- ARM 명령어 종류
- ARM 내부구조
- 숫자표현
- CPU 연산후 상태 비트 계산

ARM은 RISC 방식으로 설계된 CPU

Instuction set architecture 설계시 방법론 1) 모든 명령어를 HW로 구현한다 또는 2) 자주 사용되는 명령어들 (명령어 일부)만 HW로 구현하고 나머지 명령어는 SW로 구현한다에 따른 구분

- **CISC(Complex Instruction Set Computer)**
  - 컴파일러를 간단하게 하기 위해 명령어를 복잡하게 설계함.
  - 특정 기능을 소프트웨어가 아닌 하드웨어로 구현함으로써 해당 처리속도 높이는 게 목적
  - 기능 추가되면 하드웨어로 구현이 되는 명령어들이 추가되어 복잡함
  - 많은 어드레스 지정 모드 및 데이터 형식 등이 가능

(예) Intel CPU 설계

- **RISC (Reduced Instruction Set Computer)**
  - 명령어의 수를 줄임으로, 하드웨어가 간단하게 되고 이에 따라 동작 속도를 증가시키는 구조 설계방법(예, 파이프라인 강화)으로 프로세서의 성능 향상
  - 반복적으로 많이 쓰이는 명령어는 하드웨어적으로 구현하고, 다른 명령어들은 하드웨어적으로 구현된 명령어들을 조합하여 소프트웨어적으로 구현.
  - RISC의 장점은 작은 트랜지스터와 작은 실리콘 면적 사용함. 남은 실리콘 면적은 캐시 메모리, 메모리 관리 기능, 부동-소수점 하드웨어 등 성능강화 목적으로 사용
  - RISC는 CISC에 비해 코드 밀도가 낮은 게(코드 크기가 길어짐) 단점
  - 명령어 수가 적은 것과 간단한 명령어는 쉽게 명령어의 길이를 동일하게 할 수 있으므로, 파이프라인의 구현이 용이함.
  - 일반적으로 32비트 고정된 명령어를 사용한다. 이 명령어 구조는 범용의 32개의 32비트 레지스터 뱅크를 가짐으로 load-store 구조를 효율적으로 지원 할 수 있다.

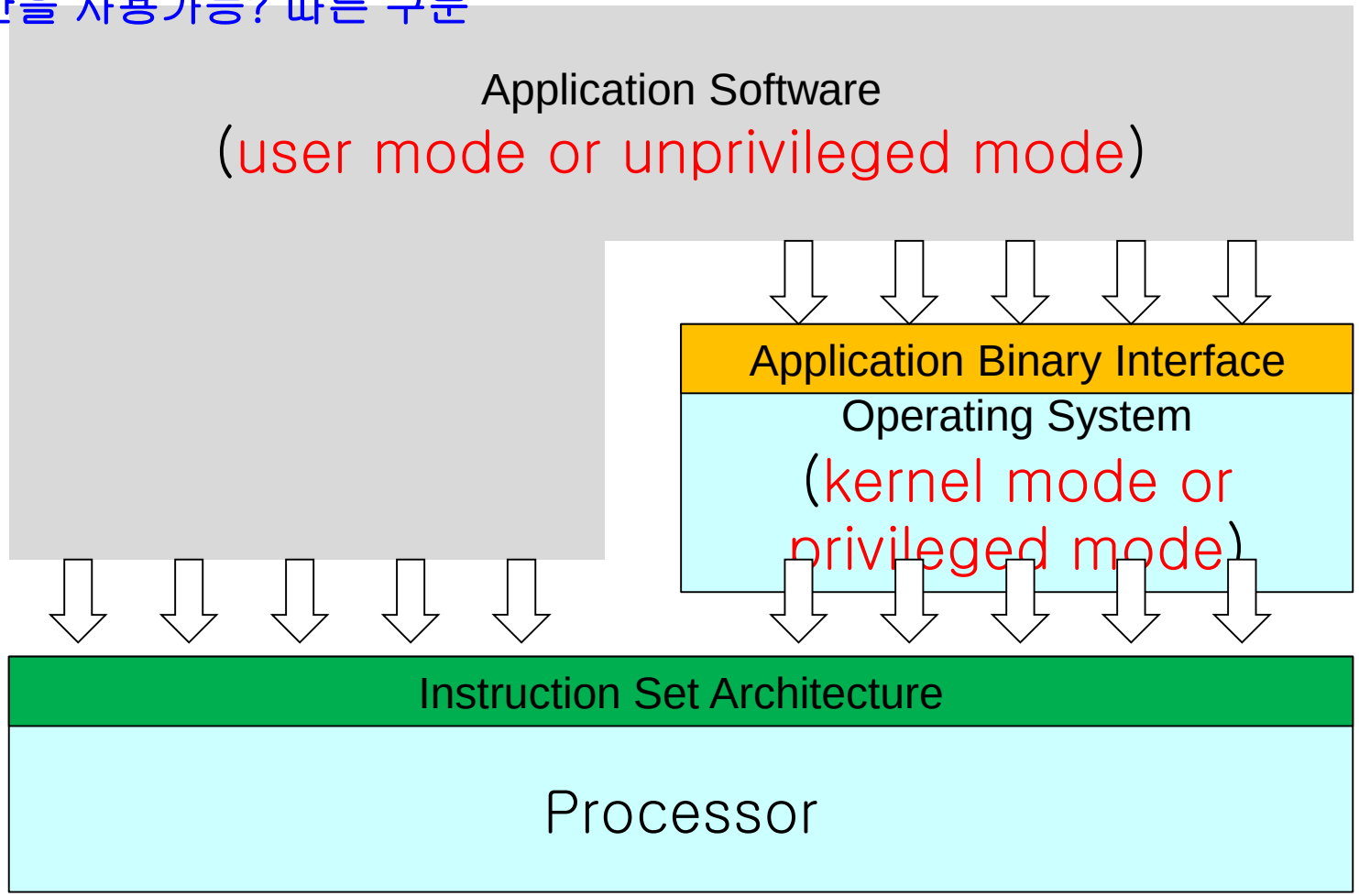
(예) MIPS, ARM CPU 설계

User Mode vs. Kernel Mode

- CPU가 user 코드를 수행하느냐. Kernel(OS) 코드를 수행하느냐로 구분.

Privileged (특권) 모드 vs. Unprivileged (비특권) 모드

- CPU내 명령어를 100% 사용가능 ? 아니면 일부 명령어를 제외한 나머지 명령어만을 사용가능? 따른 구분



ARM 프로세서 모드

- **ARM 프로세서는 기본적으로 7가지의 동작 모드를 갖는다:**
  - 각각의 동작 모드는 자신만의 레지스터 부분 집합을 갖는다 (P35에서 설명)
  - 어떤 명령어는 오직 특권 모드(privileged mode)에서만 수행 가능하다.

예외적인 상황에서 예외처리를 위해 커널코드가 수행되는 모드

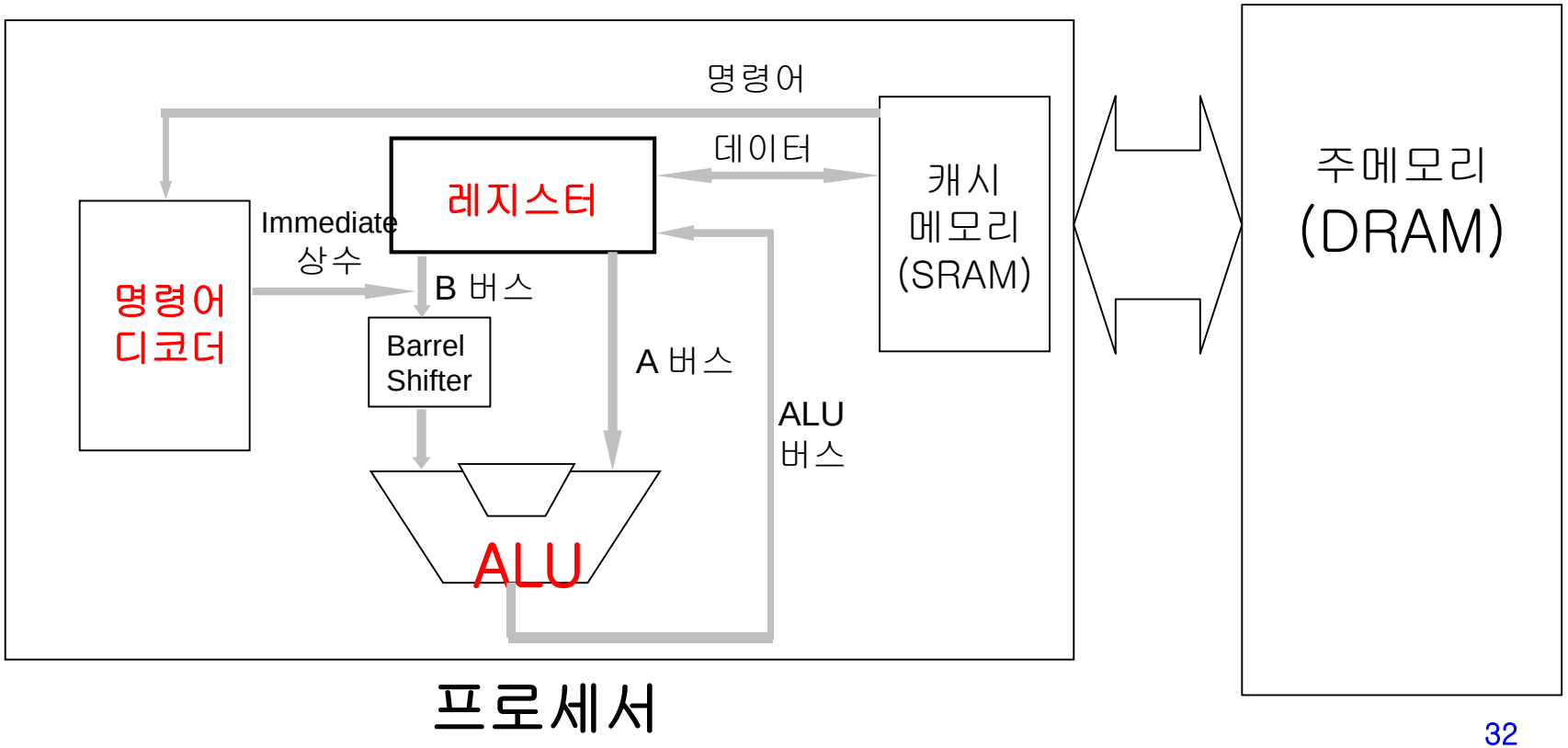
Exception modes

일반적인 상황에서 커널코드가 수행되는 모드

Mode	Description	
Supervisor (SVC)	Entered on reset and when a Software Interrupt instruction (SWI) is executed	Privileged modes
FIQ	Entered when a high priority (fast) interrupt is raised	
IRQ	Entered when a low priority (normal) interrupt is raised	
Abort	Used to handle memory access violations	
Undef	Used to handle undefined instructions	
System	Privileged mode for OS	
User	Mode under which most Applications run	Unprivileged mode

ARM 프로세서의 레지스터

- 레지스터
  - 프로세서가 연산 작업을 하는데 사용되는 값을 임시로 저장하는 공간
  - ARM에는 32비트 길이의 37개의 레지스터가 있다.



ARM 프로세서의 레지스터

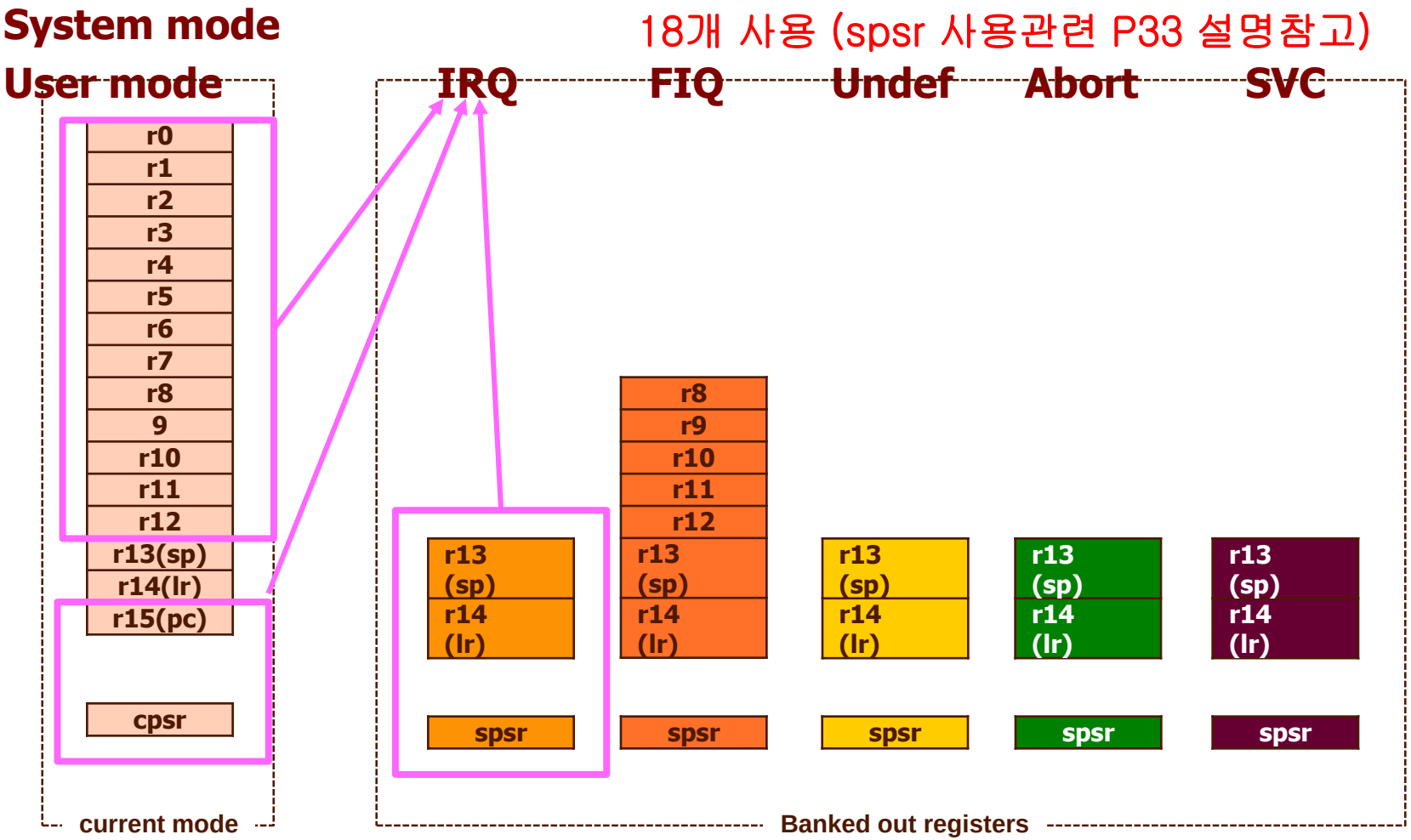
- **30개의 범용(General Purpose) 레지스터**
  - 대부분 데이터 연산 등에 사용
  - 프로세서의 동작(Operating) 모드에 따라 사용되는 레지스터가 제한된다
- **1개의 프로그램 카운터(PC : Program Counter)**
  - 프로그램을 읽어올 메모리의 위치를 나타낸다.
- **1개의 CPSR(Current Program Status Register)**
  - 프로세서가 수행하고 있는 현재의 동작 상태를 나타낸다.
- **5개의 SPSR(Saved Program Status Register)**
  - 이전 모드의 CPSR의 복사본: Exception이 발생하면 ARM이 하드웨어적으로 이전 모드의 CPSR 값을 해당 exception 모드의 SPSR에 복사
  - System 모드를 제외한 모든 privilege 모드에 각각 하나씩 존재

레지스터 vs. 메모리

- 레지스터는 메모리보다 더 빠르게 접근할 수 있다.
- 주 메모리는 복합 데이터를 위해서 사용된다.
  - 배열, 구조체, 동적 데이터
- 산술 연산을 적용하기 위해서는
  - 첫째, 메모리에 있는 변수 값을 레지스터에 적재(load)한다.
  - 둘째, 레지스터들(source) 간에 산술 연산을 수행하여 결과값을 레지스터(destination)에 저장한다.
  - 셋째, 연산결과를 레지스터로부터 메모리에 저장(store)한다.
- 컴파일러는 자주 사용되는 변수들이 레지스터를 사용하도록 실행 코드를 생성해야 한다.
  - 자주 사용되지 않는 변수들은 메모리에 두고 사용해야 한다.
  - 레지스터 최적화가 중요하다.

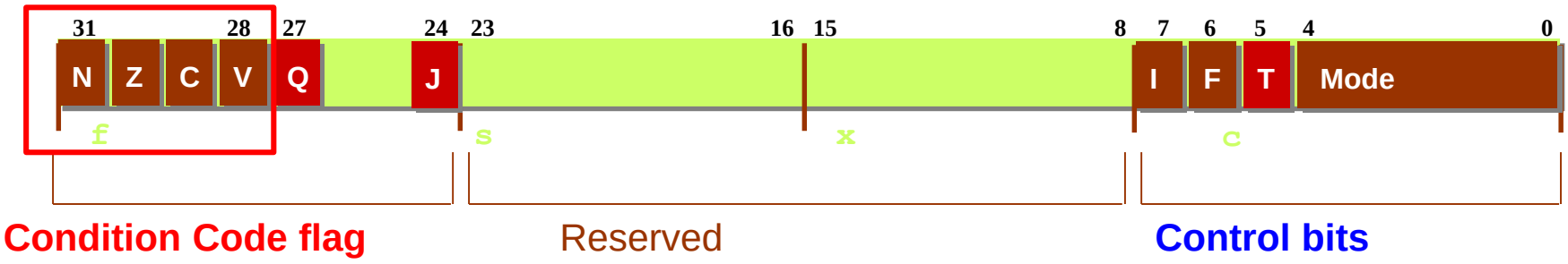
ARM 레지스터 집합

-그림상에 있는 레지스터 총 갯수는 37개
-System mode 또는 User mode 인 경우는 맨 왼쪽 column r0~r15, cpsr 총 17개 사용
-IRQ mode에서는 보라색 color로 표시된 총 18개 사용 (spsr 사용관련 P33 설명참고)

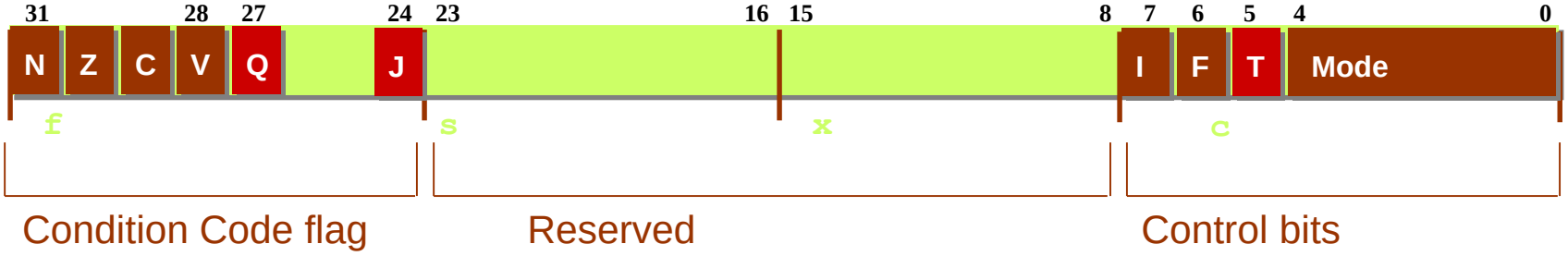


Program Status Register (PSR)

- **ARM의 Program Status Register**
  - 1개의 CPSR(Current Program Status Register)
  - 5개의 SPSR(Saved Program Status Register)
- **PSR 레지스터의 정보**
  - **Condition code flag**
    - ALU의 연산 결과 정보를 가지는 flag 정보를 가지고 있다
    - 이 flag 정보가지고 조건부 분기 명령어 수행시 조건이 참인지 거짓인지 판단한다.
  - **Control bits**
    - 프로세서를 제어하기 위한 비트로 구성되어 있다
  - Reserved



CPSR 레지스터



Flag	논리 연산	산술 연산
Negative (N=1)	사용되지 않는다	**Signed** 연산에서 비트 **31**이 세트 되어 **Negative** 결과 발생
Zero (Z=1)	연산 결과가 모두 **0**	연산 결과가 **0**
Carry (C=1)	**Shift** 동작 결과 **carry** 발생	연산 결과가 **32** 비트를 넘으면 세트
oVerflow (V=1)	사용되지 않는다	연산 결과가 **31** 비트를 넘어 **sign bit** 상실

Negative(N) 대신에 Sign(S) 라는 표현을 사용하기도 함

CPSR 레지스터 – Condition Code 플래그

- **Flag bits**는 **ALU**를 통한 명령의 실행 결과를 나타내는 부분이다.
  - Negative flag ('N' 비트)
    - ALU 연산 결과 마이너스가 발생한 경우 세트
  - Zero flag ('Z' 비트)
    - ALU 연산 결과 0가 발생한 경우 세트
  - Carry flag ('C' 비트)
    - ALU 연산 결과 자리올림이나 내림이 발생한 경우 세트
    - shift 연산에서 carry가 발생한 경우 세트
  - oVerflow flag ('V' 비트)
    - ALU 연산 결과 overflow가 발생하면 세트
  - Q flag ('Q' 비트)
    - ARM Architecture v5TE/J에서 새롭게 추가된 saturation 연산 명령의 수행 결과 saturation이 발생하면 세트된다.
    - 이 비트는 사용자에 의해서만 클리어 된다.
  - 'J' 비트
    - ARM Architecture v5TEJ에서 새롭게 추가된Java 바이트 코드를 수행하는 Jazelle state임을 나타낸다.

CPSR 레지스터 – Control 비트들

- 프로세서의 모드, 동작 **state** 와 인터럽트를 제어
  - I/F 비트
    - IRQ('I' 비트) 또는 FIQ('F' 비트)를 diable(세트) 또는 enable(클리어)
  - T 비트
    - ARM Architecture xT 버전인 경우 Thumb state 임을 나타낸다.
    - 상태를 나타낼 뿐 프로세서는 이 비트에 강제로 값을 write 할 수 없다.
    - BX 명령에 의해서만 제어된다
  - Mode bits
    - ARM의 7 가지의 동작 모드를 나타낸다.

M[4:0]	Operating Mode	M[4:0]	Operating Mode
10000	User 모드	10111	Abort 모드
10001	FIQ 모드	11011	Undefined 모드
10010	IRQ 모드	11111	System 모드
10011	SVC 모드		

2장-Part 1 Contents

Part 1: ARM 명령어 기초

- ARM 명령어 종류
- ARM 내부구조
- 숫자표현
- CPU 연산후 상태 비트 계산

컴퓨터에서 숫자 표현

- 컴퓨터에서 사용하는 숫자는 크게 **signed** 숫자와 **unsigned** 숫자로 나뉜다 **(C**언어에서 **unsigned int x=10, int y=-2)**
- **Unsigned** 숫자는 이진수로 표현
  - (예) 4 비트로 unsigned 숫자1010로 표현하면 그것은 10을 의미
- **Signed** 숫자는 이진수로 표현, 최상위 비트는 **sign**을 나타낸다
  - (예) 4 비트로 signed 숫자 1010로 표현하면 맨앞의 비트 1은 음수를 의미. 숫자의 크기와 무관한 비트임
  - Signed 숫자를 표현하는 방법에는 signed magnitude 방식과 signed 2's complement 방식이 있다. 전자는 사람에게 편한 방식이고 후자는 컴퓨터에서 사용하는 방식
  - (예) signed 숫자 1010 (음수): signed magnitude 방식에서는 -2를 나타낸다. 최상위비트가 1이므로 음수, 숫자의 크기는 최상위 비트를 제외한 나머지 010을 그냥 숫자로 보면 2. signed 2's complement 방식에서는 -6을 나타낸다. (P46을 보세요)
  - (예) signed 숫자 0010(양수): signed magnitude 방식과 signed 2's complement 방식 둘다 +2를 의미한다. 양수의 경우는 동일 (P46을 보세요)

Signed and Unsigned Numbers

- **Unsigned Binary Integers: given an n-bit number**

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

2s-Complement Signed Integers

- **Given an n-bit number**

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1 \times 2^{31} + 1 \times 2^{30} + \ldots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

2s-Complement Signed Integers

- **Bit 31 is sign bit**
  - 1 for negative numbers
  - 0 for non-negative numbers
- **$-(-2^{n-1})$ can't be represented**
- **Non-negative numbers have the same unsigned and 2s-complement representation**
- **Some specific numbers**
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

Signed Negation

- **Complement and add 1**
  - Complement means 1 → 0, 0 → 1

$$x + \bar{x} = 1111\ldots111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate +2
  - +2 = 0000 0000 … 0010_2
  - –2 = 1111 1111 … 1101_2 + 1
    = 1111 1111 … 1110_2

2's complement로 숫자표현

- (예제) 4비트를 사용할 경우
  - 0111 -> +7
  - 0110 -> +6
  - 0101 -> +5
  - 0100 -> +4
  - 0011 -> +3
  - 0010 -> +2
  - 0001 -> +1
  - 0000 -> 0
  - 1111 -> -1
  - 1110 -> -2
  - 1101 -> -3
  - 1100 -> -4
  - 1011 -> -5
  - 1010 -> -6
  - 1001 -> -7
  - 1000 -> -8

1010 -> -6 설명
최상위비트가 1 이므로 음수
크기는 ?
최상위 비트를 제외한 나머지 비트들 010의 2's complment를 구하면 크기가 된다. 010에서 1) complement 를 구하면 101이 되고 2) 여기에 1을 더하면 110이 된다. 크기가 6.

컴퓨터에서 2's complement 를 사용하는 이유 ?
- 뺄셈을 덧셈 방식으로 처리가능 (adder를 갖고 덧셈, 뺄셈 수행가능)
- 숫자 A – B 를 A + (B에 대한 2's complement) 방식으로 처리. 두개의 결과는 동일할까 ? Yes. (P54 오른쪽 상단을 보세요)

Sign Extension

- **Representing a number using more bits**
  - Preserve the numeric value
- **In ARM instruction set**
  - LDRSB, LDRSH: extend loaded byte/halfword
- **Replicate the sign bit to the left**
  - c.f. unsigned values: extend with 0s
- **Examples: 8-bit to 16-bit**
  - +2: 0000 0010 => 0000 0000 **0**000 0010
  - –2: 1111 1110 => 1111 1111 **1**111 1110

2장-Part 1 Contents

Part 1: ARM 명령어 기초

- ARM 명령어 종류
- ARM 내부구조
- 숫자표현
- CPU 연산후 상태 비트 계산

C 코드 예제: If (A > B) goto X

- 이게 컴파일된 후 기계어코드를 보면
  - **CASE 1: A와 B가 unsigned 숫자인 경우**
    - LDR R1, M[1000]: 메모리 1000 데이터(A)를 R1으로 복사
    - LDR R2, M[1004]: 메모리 1004 데이터(B)를 R2로 복사
    - CMP R1, R2: R1 – R2를 수행하여 상태비트(C,S,Z,V) update
    - BHI 2000: 만일 A > B 이면 2000번지로 jump
  - **CASE 2: A와 B가 signed 숫자인 경우**
    - LDR R1, M[1000]: 메모리 1000 데이터(A)를 R1으로 복사
    - LDR R2, M[1004]: 메모리 1004 데이터(B)를 R2로 복사
    - CMP R1, R2: R1 – R2를 수행하여 상태비트(C,S,Z,V) update
    - BGT 2000: 만일 A > B 이면 2000번지로 jump
  - BHI, BGT는 조건부 분기명령어임
    - BHI에서는 조건이 참인지는 C=1 인지 0 인지를 보고 판단한다
    - BGT에서는 조건이 참인지는 S와 V 값을 보고 판단한다
- 이번 Part에서 공부할 내용
  - CPU는 상태비트 (C,S,Z,V) 를 어떻게 계산할 까 ?
  - CPU가 A > B가 참인지를 상태비트 (C,S,Z,V) 중 어느 부분을 보고 판단할까 ? signed와 unsigned 경우가 다르다

Status Bits

- **Status bit**
  - C(carry): Cn
  - S(Sign): bn
  - Z(zero): 모든 bit=0
  - V(overflow): Cn (EXOR) Cn-1

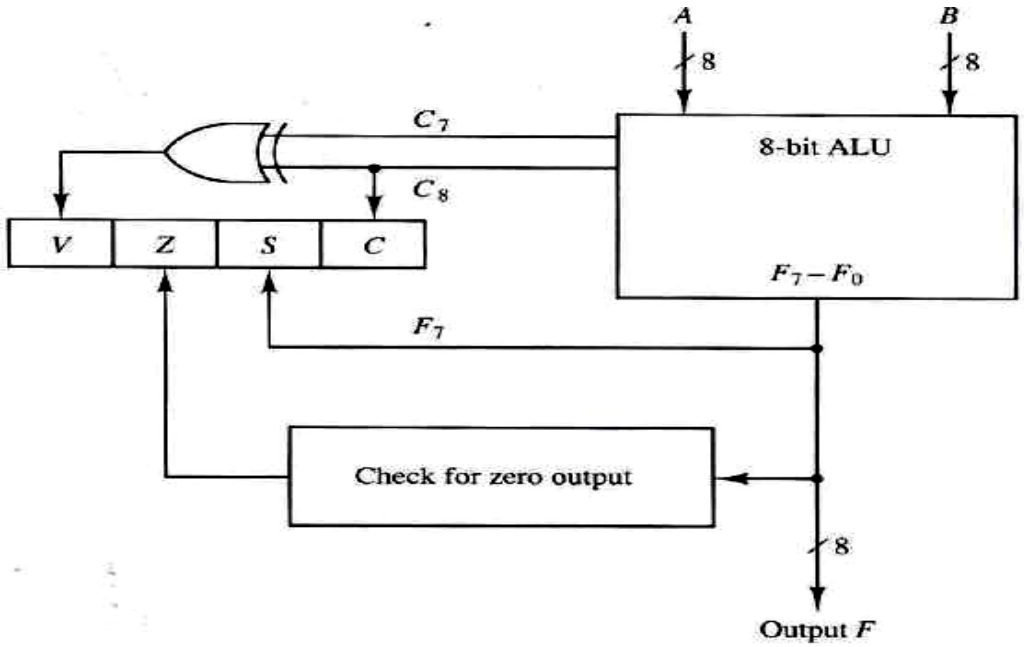
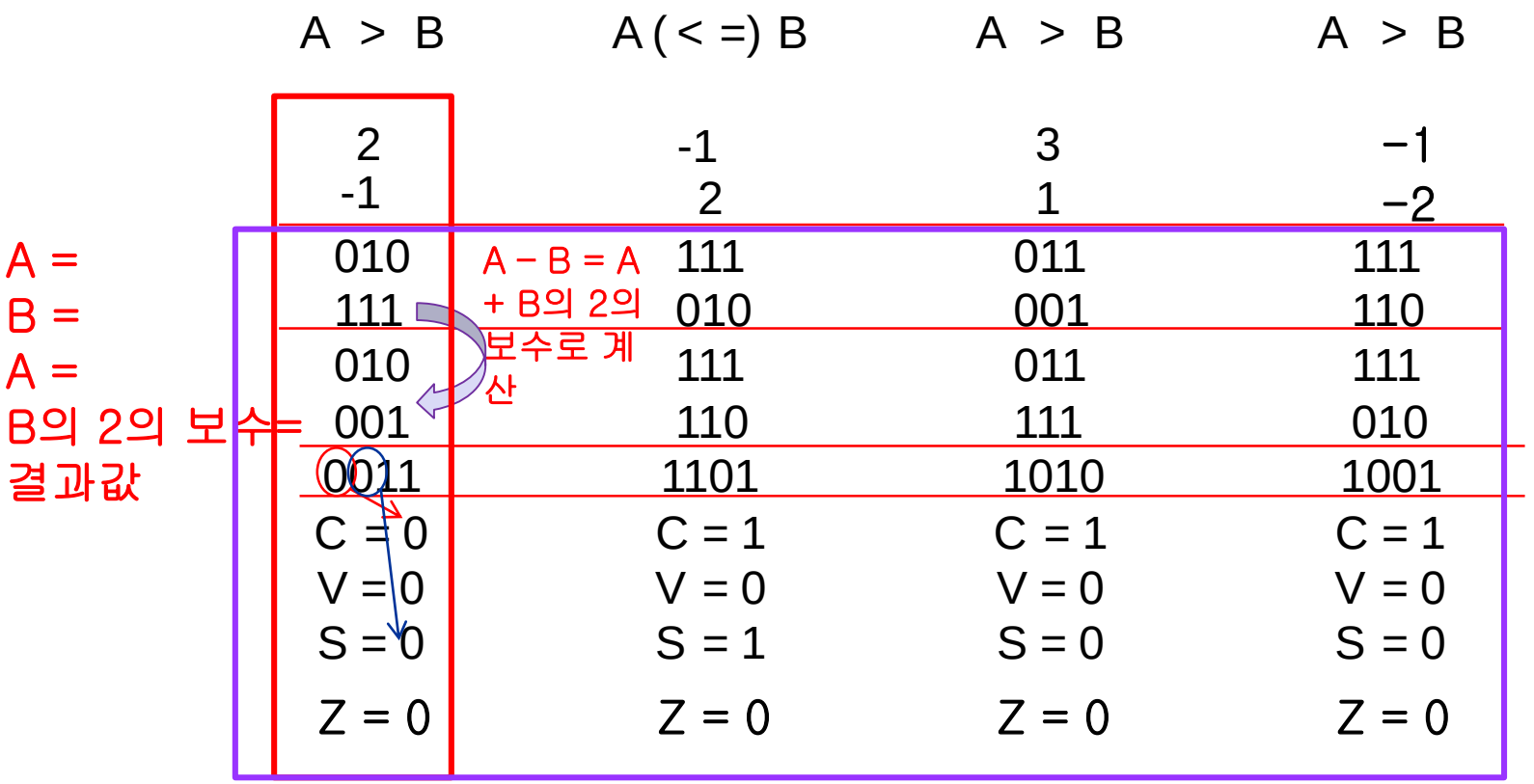


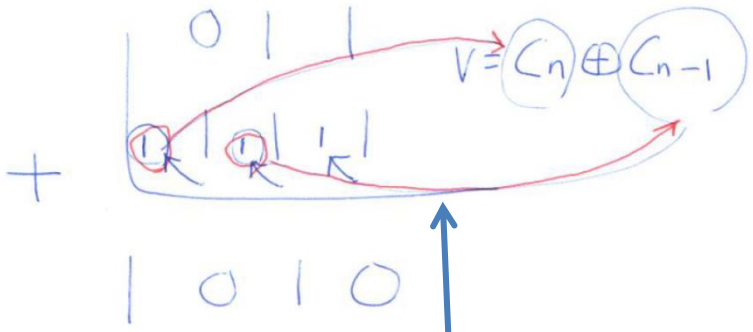
Figure 8-8 Status register bits.

상태 비트 계산 예제(signed 경우)

- 2's complement로 표현 (3 비트사용)



signed 표현 (3 비트사용)

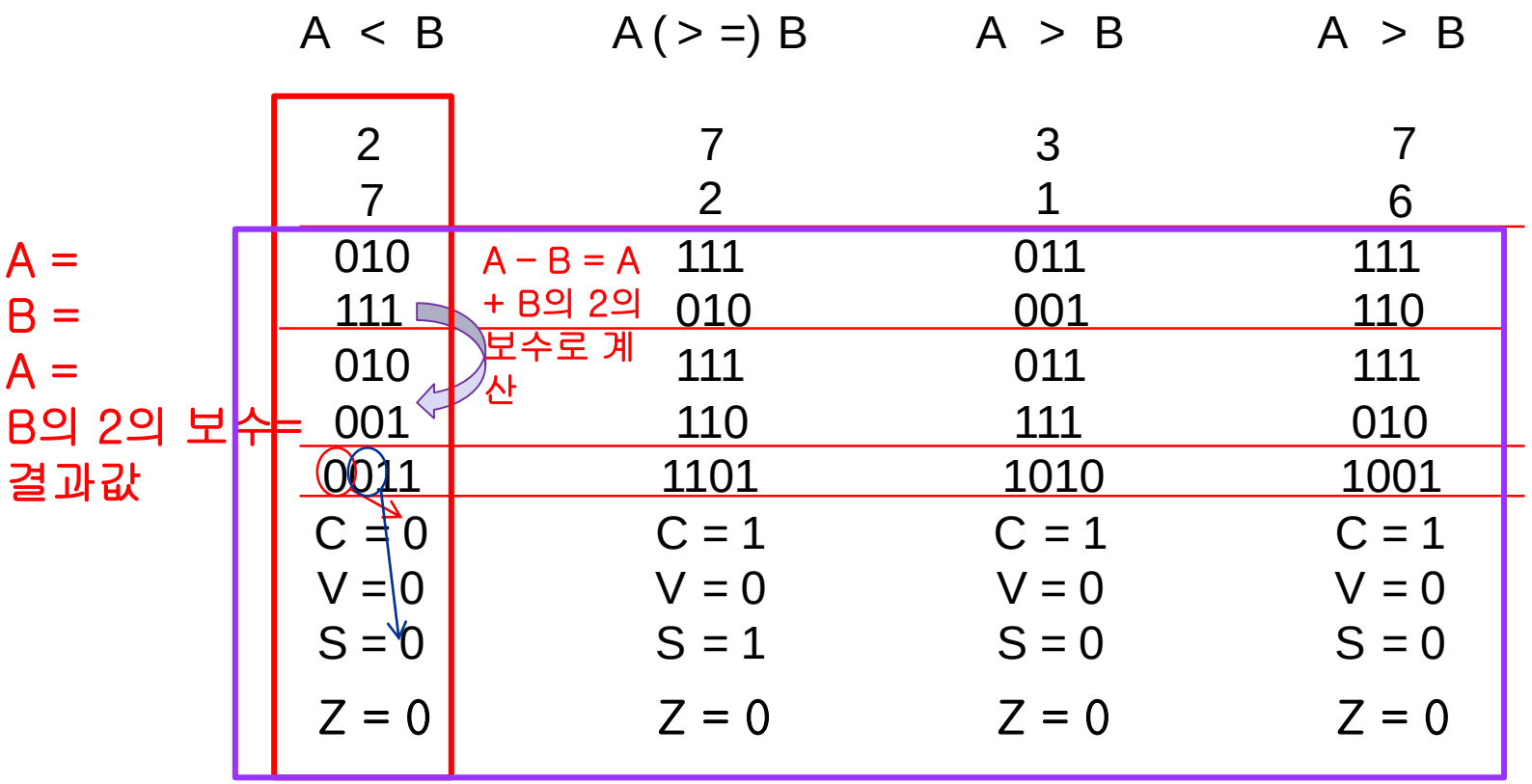


A > B	A > B	A (< =) B	A > B	A > B
3	2	-1	3	−1
-1	-1	2	1	−2
011	010	111	011	111
111	111	010	001	110
011	010	111	011	111
= 001	001	110	111	010
0100	0011	1101	1010	1001
C = 0	C = 0	C = 1	C = 1	C = 1
V = 1	V = 0	V = 0	V = 0	V = 0
S = 1	S = 0	S = 1	S = 0	S = 0
Z = 0	Z = 0	Z = 0	Z = 0	Z = 0

A − B = A + B의 2의 보수로 계산

상태 비트 계산 예제(unsigned 경우)

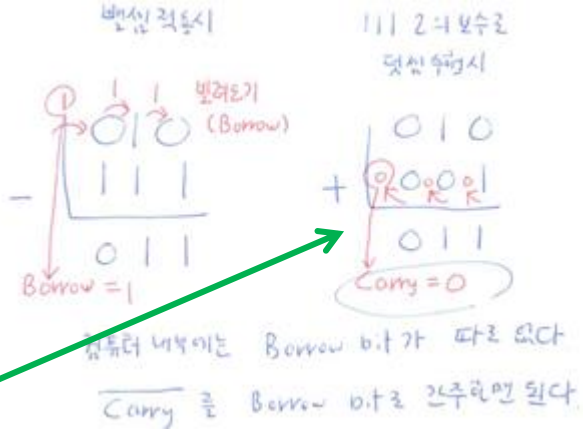
- Unsigned 표현 (3 비트사용)



P51, 53 보라색 상자안 내용이 동일. A, B 이진수값이 동일하면 CPU안에서 C, V, S, Z 계산방식은 동일. CPU는 signed와 unsigned를 구분하기 않고 계산

Unsigned 표현 (3 비트사용)

Unsigned 경우 C=0는 B=1을 의미하므로 borrow 발생, 곧 overflow 발생함을 의미.



A < B	A < B	A (> =) B	A > B
3	2	7	3	7
7	7	2	1	6
011	010	111	011	111
111	111	010	001	110
011	010	111	011	111
= 001	001	110	111	010
0100	0011	1101	1010	1001
C = 0	C = 0	C = 1	C = 1	C = 1
V = 1	V = 0	V = 0	V = 0	V = 0
S = 1	S = 0	S = 1	S = 0	S = 0
Z = 0	Z = 0	Z = 0	Z = 0	Z = 0

A – B = A + B의 2의 보수로 계산

두 숫자 비교시 판단기준

- **Unsigned (**아래 첫번째 표**)**와 **Signed(**아래 두번째 표**)** 를 비교해보면 다르다

Table for Problem 8-26

Relation	Condition of Status Bits
$A > B$	$C = 1$ and $Z = 0$
$A \geq B$	$C = 1$
$A < B$	$C = 0$
$A \leq B$	$C = 0$ or $Z = 1$
$A = B$	$Z = 1$
$A \neq B$	$Z = 0$

Unsigned 숫자를 비교하여 A > B인 경우 분기하는 명령어 "BHI 2000"을 사용
두 숫자가 Unsigned 경우 C=1 & Z=0 조건이 참이면 A>B로 판단.

ow occurs.) Show that the relative magnitude of A
l from inspection of the status bits as specified bel
)

Table for Problem 8-27

Relation	Condition of Status Bits
$A > B$	$(S \oplus V) = 0$ and $Z = 0$
$A \geq B$	$(S \oplus V) = 0$
$A < B$	$(S \oplus V) = 1$
$A \leq B$	$(S \oplus V) = 1$ or $Z = 1$
$A = B$	$Z = 1$
$A \neq B$	$Z = 0$

signed 숫자를 비교하여 A > B인 경우 분기하는 명령어 "BGT 2000"을 사용
두 숫자가 signed 경우 (S EXOR V) = 0 & Z=0 조건이 참이면 A>B로 판단

상태 비트 계산 예제(signed 경우) – overflow ?

- **overflow detection**
  - **signed 2's complement의 경우 C_n exclusive-or C_{n-1}= 1 이면 overflow 발생으로 간주**
  - **unsigned 의 경우 C_n=1이면 overflow는 발생으로 간주**
  - https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Comb/overflow.html

2's complement로 표현 (3 비트사용)

A > B	A > B	A (< =) B	A > B	A > B
3	2	-1	3	−1
-1	-1	2	1	−2
011	010	111	011	111
111	111	010	001	110
011	010	111	011	111
= 001	001	110	111	010
0100	0011	1101	1010	1001
C = 0	C = 0	C = 1	C = 1	C = 1
V = 1	V = 0	V = 0	V = 0	V = 0
S = 1	S = 0	S = 1	S = 0	S = 0
Z = 0	Z = 0	Z = 0	Z = 0	Z = 0

A − B = A + B의 2의 보수로 계산

상태 비트 계산 예제(unsigned 경우) – overflow ?

- Unsigned 표현 (3 비트사용)