ARM 어셈블리

https://youtu.be/Yby-zH60YTY

ARM

- ARM은 Advanced RISC Machine의 약자,
  임베디드 기기에 주로 사용되는 32bit 프로세서
- 모바일 기기 또는 IoT 디바이스에 많이 사용됨
- ARMv8 부터는 32비트 지원을 유지하면서 64bit 명령어 지원

Cortex-A : 애플리케이션 실행 능력에 중점
Cortex-R : Real-Time 용도에 특화
Cortex-M : 마이크로 컨트롤러 시장 타겟

ARM 동작 모드

M 프로파일을 제외

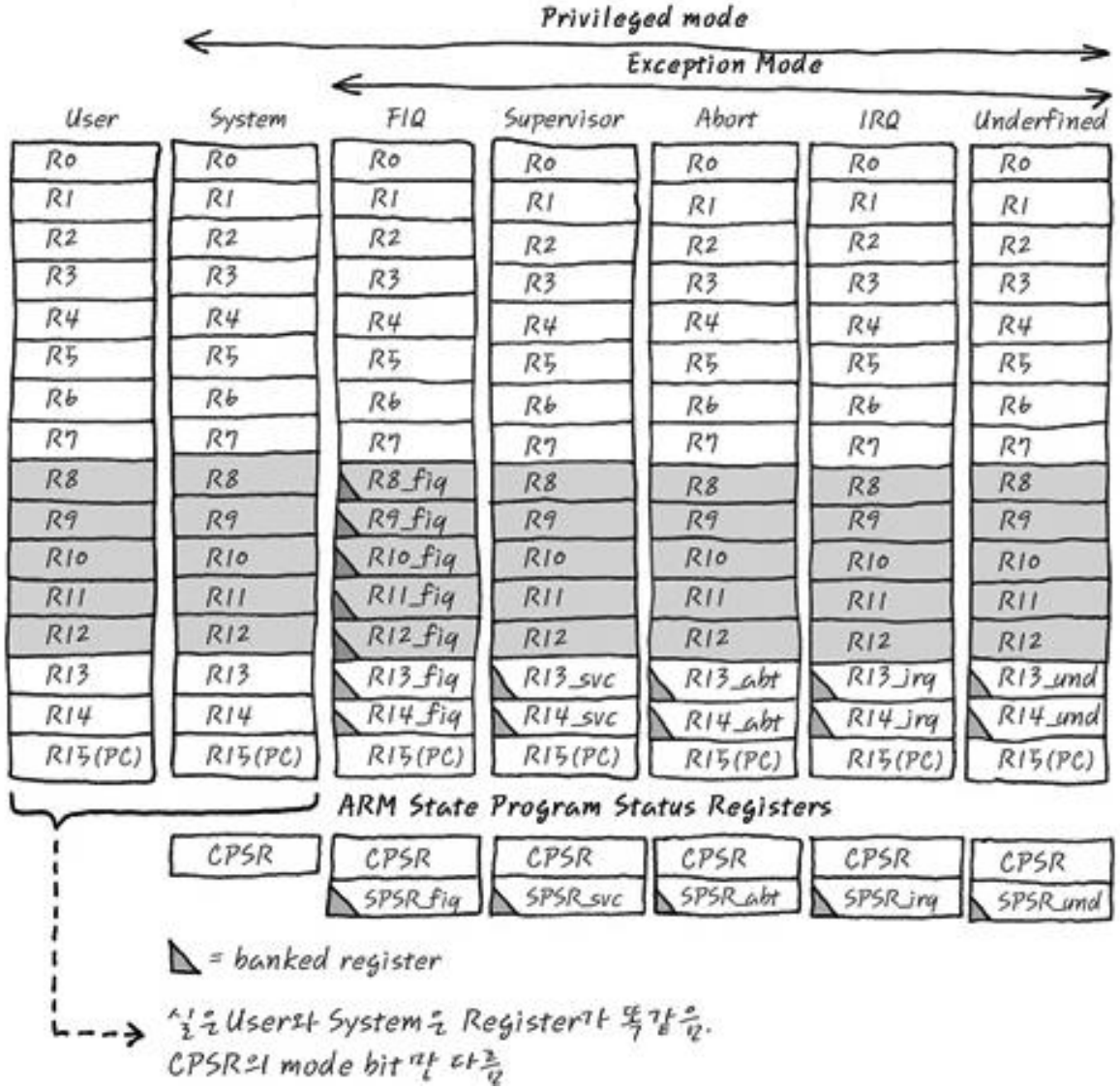
권한	모드	진입 유발 예외	설명
Un-privileged	User		User 태스크나 어플리케이션 수행시의 모드
privileged	FIQ	Fast Interrupt	빠른 인터럽트 처리 모드
	IRQ	Standard Interrupt	일반적인 인터럽트 처리모드
	SVC	Reset, Power On, SWI	시스템 자원(memory, I/O, reg)을 자유롭게 관리하기위한 모드
	Abort	Memory falult	메모리에서 명령, 데이터를 r/w시 오류발생 처리 하기 위한 모드
	Undefined	Undefined Instruction	Fetch 한 명령어가 디코더에 정의되지 않은 경우를 처리하기 위한 모드
	System		User 모드와 동일한용도

ARM 동작 모드

모드 필요 : 현재 프로세서가 어떤 권한을 가지고
어떤 종류의 작업을 하고 있는지 나타내기 위해서

Ex) 예외가 발생할 때
모드 X
현재하던 일을 멈춤 ->
현재 하던 일에 대한 내용을 백업 ->
예외처리를 한 뒤에 ->
백업한 것을 다시 복구 한 뒤 돌아온다

모드 O
공동으로 쓰는 범용 레지스터와 달리
bankedregister 사용->
백업 필요 없음



Registers

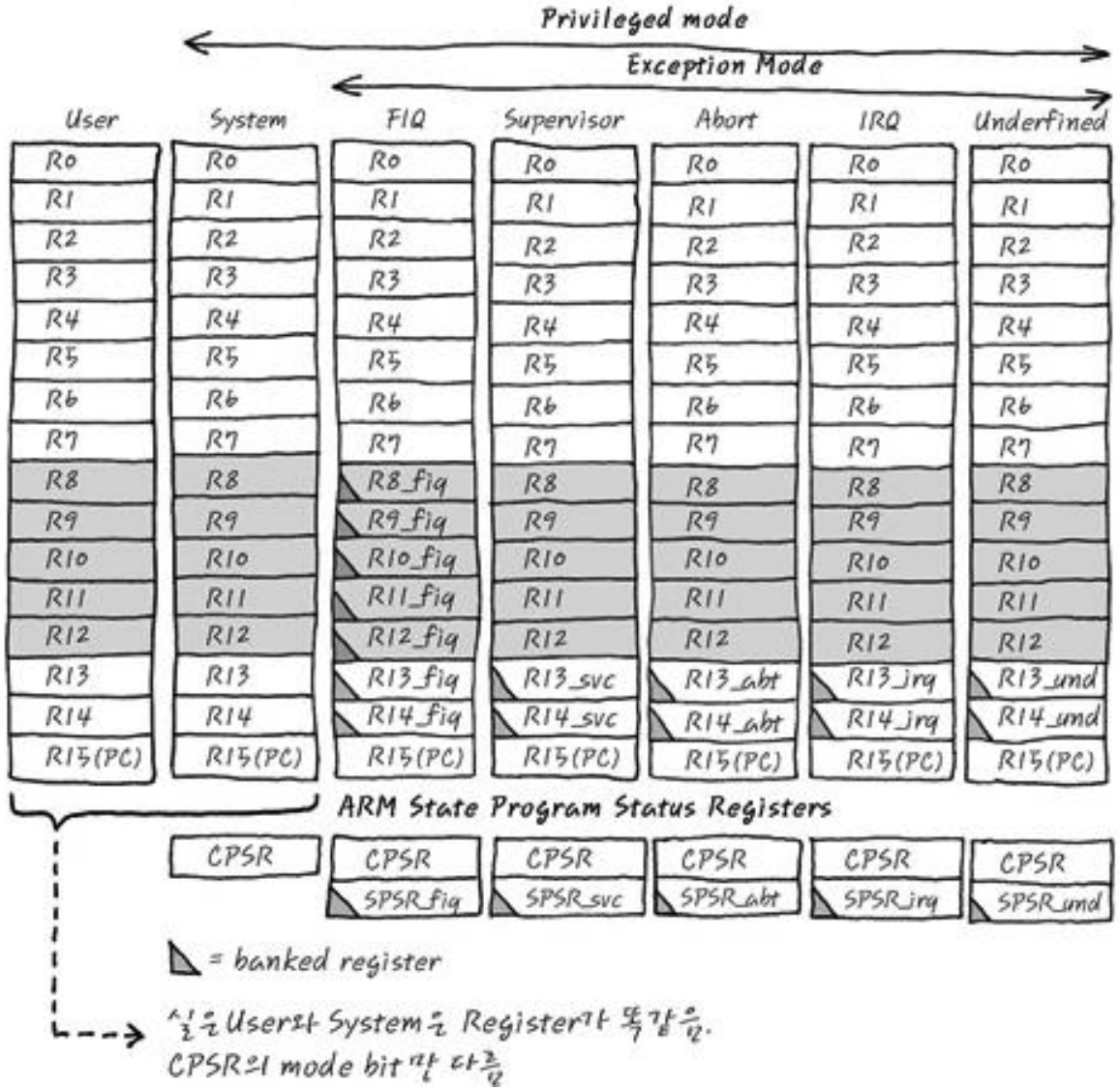
- ARM은 32비트 길이의 레지스터를 37개

- 30개의 범용 레지스터와 한 개의 PC, 한 개의 CPSR, 5개의 SPSR

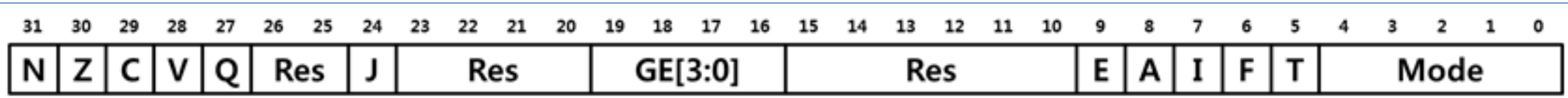
1. 범용 레지스터
2. PC (Program Counter)
3. CPSR(Current Program Status Register)
4. SPSR(Saved Program Status Register)

범용 레지스터, PC, SPSR(Saved Program Status Register)

- R0 ~ R7 모든 CPU 모드 에서 동일
- R0 ~ R3 함수에 인수를 전달하는데 사용
- R4 ~ R11 초기값 리턴 필요
  PUSH {R0-R3,R12,LR}
  POP {R0-R3,R12,PC}
- R8 ~ R12는 FIQ 모드를 제외한 모든 CPU 모드에서 동일
  FIQ 모드에는 고유 한 R8 ~ R12 레지스터
- SPSR R13 및 R14는 시스템 모드를 제외한 모든 권한있는 CPU 모드에서 뱅킹
- R13은 SP (스택 포인터)
- R14는 LR (링크 레지스터)
- R15는 PC, 프로그램 카운터

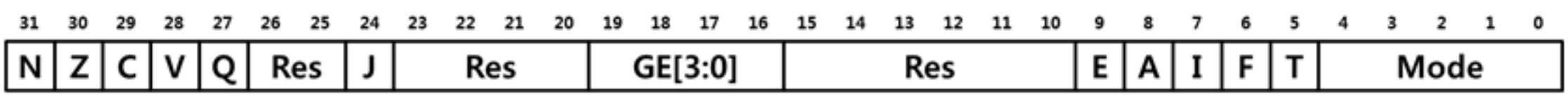


CPSR(Current Program Status Register)



N	음수(Negative)플래그, 플래그 설정 연산 결과를 31 비트에 기록한다.
Z	제로 (Zero) 플래그, 플래그 설정 연산 결과가 0인 경우에 기록한다.
C	캐리 (Carry) 플래그, 덧셈에 대한 unsigned overflow를, 뺄셈에 대한 not-borrow를 기록한다. 쉬퍼트 회로에 의해서도 사용된다.
V	오버플로우(Overflow) 플래그, 플래그 설정 연산에 대해 signed overlfow를 기록한다.
Q	포화(Saturation) 플래그, 포화시에 일부 명령어가 이 플래그를 1로 설정한다.
J	J = 1은 자바 실행이 가능함을 의미한다.(이때 T = 0 이어야 함) 이 비트를 변경하기 위해서는 BXJ 명령어를 사용한다. (ARMv5E 이상)
Res	이 비트들은 확장을 위해 예약되어 있다. 소프트웨어는 이 비트들에 저장되어 있는 값을 유지해야함

CPSR(Current Program Status Register)



GE[3:0] SIMD greater-or-equal 플래그

E 데이터 엔디안값을 제어한다.

A A = 1은 불확실한 data abort를 비활성화 한다.

I I = 1 은 IRQ 인터럽트를 비활성화 한다.

F F = 1 은 FIQ 인터럽트를 비활성화 한다.

T T = 1은 Thumb 상태를 가리키고, T = 0은 ARM 상태를 가리킨다. 이 비트를 변경하기 위해서는 BX 또는 BLX 명령을 사용해야 한다.

Mode 현재 프로세서 모드

기타 기술

Thumb

- 16비트 포맷으로 ARM 명령어 세트의 서브셋을 다시 인코딩하여 생성되는 두 번째 명령어 세트
- 16비트 코드로 코드를 축약한다.
- 분기 명령어만 조건문으로 사용 가능, r0~r7만 사용 가능

Thumb-2

- 16비트 명령어세트에 32비트 명령어를 추가하여 확장
- ARM 명령어 세트를 사용하지 않고도 처리 가능

기타 기술

트러스트존(TrustZone)

- ARM에서 제공하는 보안 기술로, 현재 모든 Cortex-A 클래스 프로세서에 통합됨
- 운영체제로부터 보호되는 두 번째 환경을 생성해 보안성을 제공 하는 방식

NEON

- SIMD(Single Instruction Multiple Data)명령어 세트
- NEON 명령어를 사용하면 한번의 사이클에 연산이 여러 데이터를 레지스터에 작업하게 되어 높은 성능을 얻을 수 있음

ARM 어셈블리

레이아웃 포맷

레이블: 명령어; 주석

- 레이블 : 메모리 위치를 참조하는 일반적인 방법. 레이블은 분기명령어에 사용됨.
- 명령어 : ARM 명령어 혹은 어셈블러 디렉티브
- 주석 : ; 이후에 나오는 모든 문자는 주석으로 간주

```
Start :
        MOV r1, #20 ; 20이라는 값을 레지스터 r1에 넣는다.
        MOV r2, #22 ; 22이라는 값을 레지스터 r2에 넣는다.
        ADD r0, r1, r2 ; r1과 r2를 더하면 r0에는 42 값이 들어간다.
        b end : 'end'로 분기
```

ARM 어셈블리

명령어 포맷

<op>{cond}{flags} Rd, Rn, Operand2

- <op> : 3개의 문자로 구성되며 피연산자를 호출
- {cond} : 2개의 문자로 구성된 조건코드, 옵션
- {flags} : 추가적인 플래그, 옵션
- Rd : 목표 레지스터
- Rn : 첫 번째 레지스터
- Operand2: 두번째 레지스터 혹은 두번째 피 연산자

ARM 어셈블리 - 이동

MOV{<cond>}{S} Rd, Rs

: 데이터를 레지스터에 복사.
소스(source)는 레지스터나 즉시값(0~ 65535)
목적지(destination)는 항상 레지스터

MVN{<cond>}{S} Rd, Rs

: 부정값을 레지스터(destination = NOT(source))로 복사
mov로 처리할 수 없는 값을 저장할 때 유용.
즉시 값으로 표현할 수 없는 숫자(0xFF00FFF)

NEG{<cond>}{S} Rd, Rs

: Rs 값을 -1과 곱한 결과를 Rd에 저장

ARM 어셈블리 - 산술

ADD{<cond>}{S} Rd, Rm, Rs
: Rm과 Rs를 더하여 Rd에 결과를 저장

ADDS {<cond>}{S} Rd, Rm, Rs
: Rm과 Rs를 더하여 Rd에 결과를 저장
사용시 결과가 레지스터 길이를 넘으면 캐리 플래그를 업데이트한다.

ADC{<cond>}{S} Rd, Rm, Rs
: Rm과 Rs를 더하여 Rd에 결과를 저장.
캐리 플래그가 설정 되있다면 캐리비트를 더한다.
레지스터 길이를 넘으면 캐리 플래그를 업데이트

ARM 어셈블리 - 산술

SUB{<cond>}{S} Rd, Rm, Rs
: Rm에서 Rs를 빼서 Rd에 결과를 저장

SBC{<cond>}{S} Rd, Rm, Rs
: SUB와 유사하나 SUB에서 캐리 플래그가 설정되면 결과에 캐리비트를 뺀다.

RSB{<cond>}{S} Rd, Rm, Rs
: SUB와 연산의 순서가 반대, Rs에서 Rm를 빼서 Rd에 결과를 저장.
배럴 시프터가 필요 할 때 명령어 사용을 절약 할 수 있다.
RSB r1, r2, r3, LSL #1 ; r1 = (r3*2) – r2

RSC{<cond>}{S} Rd, Rm, Rs
: RSB와 유사하나 SUB에서 캐리 플래그가 설정되면 결과에 캐리비트를 뺀다.

ARM 어셈블리 – 데이터 전송

LDR {<cond>}{B|H} Rd, addressing

: 시스템 메모리에서 하나의 데이터 항목을 레지스터로 이동하기 위해 사용

STR

: 레지스터를 가져와서 시스템 메모리에 32비트를 저장한다.

Ex)

```
wordcopy
        LDR    r3, [r0], #4
        STR    r3, [r1]], #4
        SUBS   r2, r2, #1
        BNE    wordcopy
```

ARM 어셈블리 – 논리

AND
: 2개의 피연산자를 사용해 논리 AND 수행하여 결과를 목적 레지스터에 저장

EOR
: 비트와이즈 연산에 유용, 비트를 효율적으로 ‘스위칭‘

ORR
: 2개의 레지스터를 사용해 논리 OR을 수행하여 그 결과를 저장

BIC
: AND Not, C에서는 operand1 & (!operand2)

CLZ`{<cond>}Rd, Rm`
레지스터 `Rd`을 가져와서 앞부분의 `0`의 개수를 센 후에 그 결과를 `Rm`에 저장

ARM 어셈블리 – 비교

CMP

: 2개의 숫자를 비교하는데 빼는 연산을 사용, 결과에 따라 상태 플래그를 업데이트

```
CMP r0, r1 ;
```

TST

: operand1 AND operand2와 같은 동작으로 레지스터의 비트가 클리어 혹은 설정 되었는지 확인하는 테스트 명령어

결과에 따라 상태 플래그를 업데이트

```
LDR r0, [r1] ; r1이 가르키는 메모리를 r0으로 로드한다.
TEQ r0, 0x80; r0의 비트 7이 1인가?
BEQ another_routine ; 그렇다면 분기
```

TEQ

2개의 숫자를 비교하는데 OR 연산을 사용, 결과에 따라 상태 플래그를 업데이트

ARM 어셈블리 – 분기

B there

: 라벨이 there인 곳으로 무조건 분기한다. (Branch)

BEQ there

: 플래그가 0이면 there로 분기한다.
  아니면 다음 명령어를 수행한다. (Branch Equal)

BNE there

: 플래그가 0이 아니면 there로 분기한다.
  아니면 다음 명령어를 수행한다. (Branch Not Equal)

BL sub+ROM

: 계산된 위치의 서브루틴을 호출한다. (Branch with Link)

최적화 방법들 - 카운트를 높이지 않고 줄이기

- 카운터를 증가시키는 코드를 작성하는게 일반적이지만
  0으로 감소시키는게 더 빠르다

```
mov r0, 0

loop:
        ADD r0, r0, #1
        CMP r0, #15
        BLE loop
```

```
mov r0, 0

loop:
        SUBS r0, r0 #1
        BNE loop
```

최적화 방법들 - 정수

- 부동 소수점의 처리는 정수 연산에 비해 많은 실행 시간 필요
- 일반적으로 사용하는 정수는 원하지 않은 연산을 피하기 위해 시스템 버스와 같은 너비를 갖게 하므로
  u16이 필요하더라도 32비트 시스템에서는 u32 사용이 더 빠르다
- 양수의 값만 처리한다면 unsigned로 만드는 것이 더 빠름
- Ex) 100.01사용시 100을 곱하여 사용하기

최적화 방법들 - 나눗셈

- 나눗셈은 피할 수 있다면 피할 것
- 나눗셈을 시프트로 변경할 수 있는지 고민할 것
- Ex) (a / b) > c를 a>(c*b)로 작성

최적화 방법들 – 적당한 파라미터, 객체 대신 포인터

- ARM에서 파라미터는 서브루틴을 호출하기 전에
  레지스터 r0부터 r3까지에 파라미터 값을 저장해 넘김
- 그 외의 파라미터는 스택에 저장
- 파라미터가 너무 클 경우에도 스택에 푸쉬됨
- 객체의 어드레스를 서브루틴에 전달하여 스택에 푸시 되지 않도록 함

최적화 방법들 – 시스템 메모리를 자주 업데이트 X

```
Void loopit(void)
{
        u32 i; // 지역변수
        iGlobal = 0; // 전역변수

        //32비트 인덱스 감소
        for (i=16; i !=0; i--)
        {
                iGlobal++;
        }
}
```

```
Void loopit(void)
{
        u32 i; // 지역변수
        u32 j;
        iGlobal = 0; // 전역변수

        //32비트 인덱스 감소
        for (i=16; i !=0; i--)
        {
                j++;
        }
        iGlobal = j; //지역변수의 값을 전역변수에 복사
}
```

최적화 방법들 – 특별한 루틴

- 임베디드 시스템에서는 특정 함수를 위해
  더욱 최적화된 루틴을 생성해야 하는 일이 있음

- 주로 수학 연산에 관련된 경우

- Ex) 10으로 곱하기 위한 빠른 루틴

MOV r1, r0, asl #3 ; r0에 8을 곱한다.

ADD r0, r1, r0, asl #1; 결과에 r0를 두 번 더한다.

최적화 방법들 – 하드웨어

- 주파수 설정
  프로세서가 최대 속도로 동작할 필요가 있을 때
  시스템 호출은 프로세서로 하여금 최대 속도로 동작
  연산이 끝나면 낮은 속도로 돌아와 에너지를 절약

- 캐시 설정
  명령어 캐시, 데이터 캐시, 캐시 라인 잠굼, Thumb 사용

사이트

- 라즈베리 파이 ARM 어셈블러

https://thinkingeek.com/arm-assembler-raspberry-pi/

간단한 코드

1. vim, nano 같은 편집기로 first.s 파일 생성
2. 코드작성 후 어셈블한다.
   as -o first.o first.s
3. first.o를 링크하여 실행파일 생성
   gcc –o first first.o
4. 실행
   ./first
5. ./first; echo $?
   확인명령어
6. 에러코드 7값이 출력

```
.global main

main:
        mov r0, #2
        bx lr
```

```
all: first

first: first.o
        gcc -o $@ $+

first.o : first.s
        as -o $@ $<

clean:
        rm -vf first *.o
```

Q & A