ARM NEON

유튜브 주소 : https://youtu.be/J55KVQUxbuY

NEON 개요

NEON: ARM 아키텍처의 SIMD 기술

- 병렬 연산을 통해 성능을 극대화하는 데이터 처리 기술

- SIMD(Single Instruction Multiple Data)
  - 하나의 명령어로 여러 데이터를 동시에 처리

- 128 bit 벡터 레지스터 사용
  - 128 bit 벡터 레지스터를 통해 8, 16, 32, 64 bit 데이터 타입을 병렬로 처리 가능

- 지원 데이터 타입
  - 8, 16, 32, 64 bit 정수 데이터
  - 32, 64 bit 부동 소수점 데이터

NEON 개요

NEON: ARM 아키텍처의 SIMD 기술

- NEON 장점
  - 연산 성능 및 효율 극대화 -> 고성능, 저전력 연산 가능
  - 다양한 데이터 타입 지원
    - 정수 및 부동소수점 데이터를 모두 병렬로 처리 가능

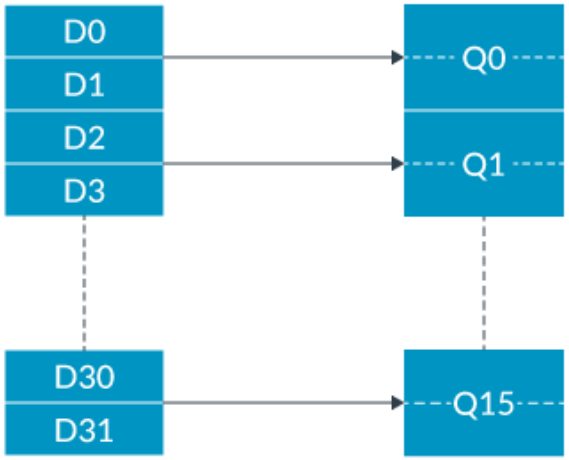
- NEON 단점
  - 메모리 대역폭이 좁은 경우 성능에 제약 발생 가능
    - NEON의 병렬 연산은 많은 메모리를 필요로 함
  - 호환성에 대한 제한 존재
    - ARM 아키텍처에서만 사용 가능하므로, 크로스 플랫폼 개발 불가

ARM SIMD 기술 타임라인

- ARMv5(1999)
  - 별도의 SIMD 기능은 제공하지 않았으나, 일부 DSP 기능을 지원
    - DSP: 디지털 신호 처리 기술로, 정수형 데이터 병렬 연산에 특화된 기능
- ARMv6(2001)
  - DSP 기능을 확장하여 정수형 데이터(8, 16bit)에 대한 SIMD 연산이 가능해짐
- ARMv7-A(2008)
  - NEON 기능을 도입, 64bit 레지스터를 사용
  - 8,16,32,64bit 정수 및 32bit 부동 소수점 병렬 연산 지원
- ARMv8-A(2011)
  - 레지스터가 128bit로 확장되었으며, 64bit 부동 소수점 연산을 추가로 지원
- ARMv8.2-A(2016)
  - SVE 도입 -> 벡터 길이를 128 ~ 2048bit까지 가변적으로 조절 가능
- ARMv9(2021)
  - SVE2 도입 -> NEON과 SVE의 기능을 통합

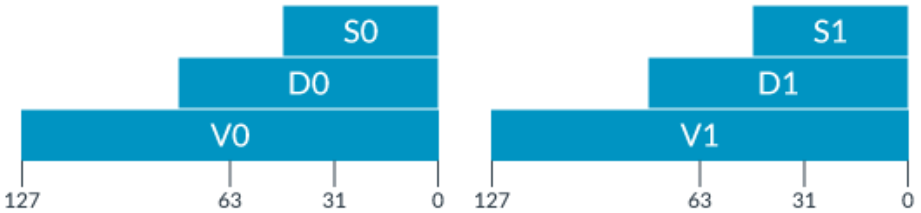
NEON 레지스터 구조

- ARMv7-A: 32개의 64bit 레지스터로 구성
  - D0부터 D31로 명명
  - D레지스터 2개를 128bit 레지스터인 Q레지스터 1개로 맵핑 가능



ARMv7 레지스터 구조

- ARMv8: 32개의 128bit 레지스터로 구성
  - V0부터 V31로 명명
  - 각 벡터 레지스터는 32bit, 64bit 뷰로도 접근 가능



ARMv8 레지스터 구조

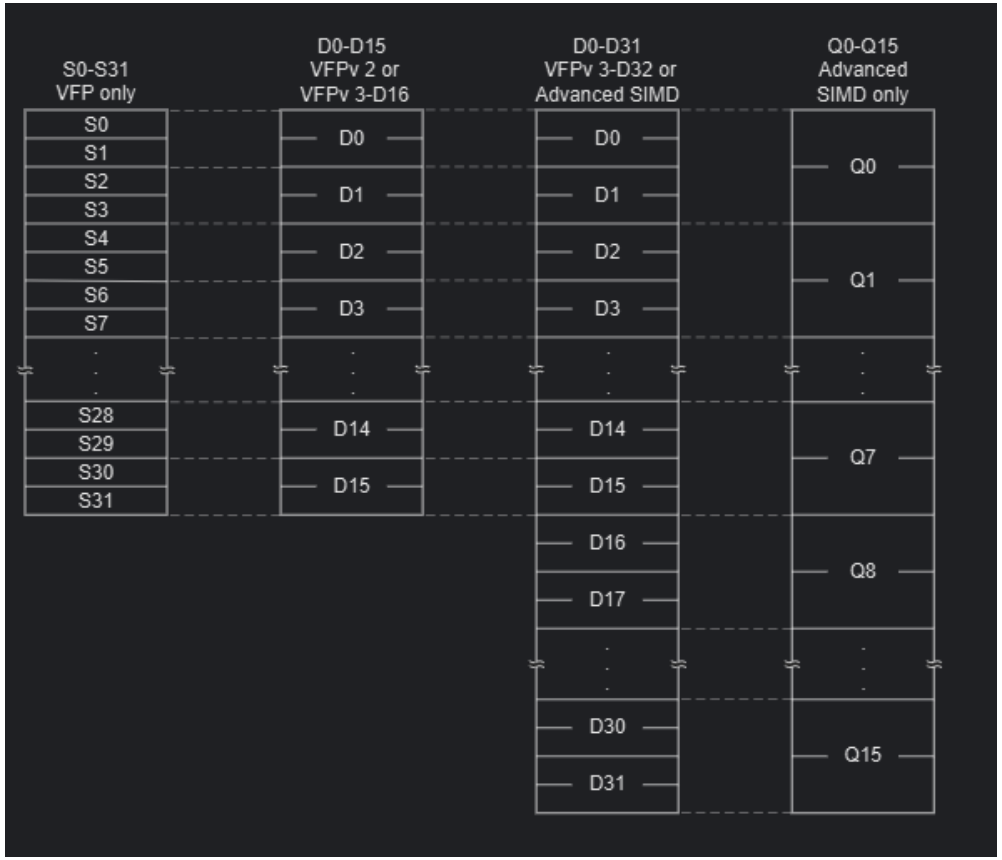
NEON 레지스터 구조

- SIMD와 VFPv3가 동시에 사용될 경우, 동일한 레지스터 뱅크 공유
  - VFPv3: 부동 소수점 연산기

- 32개의 64비트 레지스터를 지원
  - 32개의 부동 소수점 레지스터 또한 지원
    - VFPv3가 VFPv3-D32방식으로 구현될 경우

- 레지스터 뱅크를 다양한 뷰로 접근 가능
  - 쿼드 워드 -> 128 bit 뷰 (Q0 – Q15)
  - 더블 워드 -> 64 bit 뷰 (D0 – D31)
  - 단정밀도 부동 소수점 연산 -> 32 bit 뷰 (S0 – S31)

- 명령어에 따라 자동으로 적절한 뷰에 접근



NEON 인트린직 함수

- NEON 인트린직 함수
  - C/C++ 코드 내에서 NEON 명령어를 호출할 수 있도록 해주는 함수
  - 각 NEON 명령어를 함수 형태로 제공
  - 어셈블리 코드보다 쉽게 병렬 연산 구현 가능

- NEON 사용 방법 2가지
  - 어셈블리 코드 직접 작성
    - 인트린직 함수보다 더 높은 수준의 최적화 가능
    - 코드 복잡도가 인트린직 함수에 비해 매우 높아짐
  - NEON 인트린직 함수 사용
    - 고수준 언어에서 NEON의 기능을 간단하게 사용 가능
    - 일반적으로는 어셈블리보다 인트린직 함수를 많이 활용

NEON 인트린직 함수

- 주요 NEON 인트린직 함수
- 데이터 로드/스토어 함수
  - Vld1q_u8: 메모리에서 128 bit 벡터로 데이터를 로드하는 함수
  - Vst1q_f32: 128 bit 벡터의 데이터를 메모리에 저장하는 함수

- 벡터 연산 함수
  - Vaddq_u8: 128 bit unsigned 정수를 요소 별로 더하는 함수
  - Vmulq_f32: 128 bit float형 데이터를 요소 별로 곱하는 함수

- 타입 변환 함수
  - Vcvtq_f32_s32: 32 bit 정수 벡터를 float 벡터로 변환하는 함수
  - Vcvtq_s32_f32: float 벡터를 32 bit 정수 벡터로 변환하는 함수

NEON 인트린직 함수 예시 코드

128bit 간 덧셈 구현

```c
// 레퍼런스 C 구현
void add_vectors_reference(uint8_t *a, uint8_t *b, uint8_t *result, int length) {
    for (int i = 0; i < length; i++) {
        result[i] = a[i] + b[i];
    }
}
```

```c
// NEON 인트린직 함수 구현
void add_vectors_neon(uint8_t *a, uint8_t *b, uint8_t *result, int length) {
    int i;
    for (i = 0; i <= length - 16; i += 16) {
        uint8x16_t vec_a = vld1q_u8(&a[i]);
        uint8x16_t vec_b = vld1q_u8(&b[i]);
        uint8x16_t vec_result = vaddq_u8(vec_a, vec_b);
        vst1q_u8(&result[i], vec_result);
    }
    for (; i < length; i++) {
        result[i] = a[i] + b[i];
    }
}
```

```c
// 어셈블리 코드 구현
void add_vectors_asm(uint8_t *a, uint8_t *b, uint8_t *result, int length) {
    int i = 0;
    for (; i <= length - 16; i += 16) {
        __asm__ volatile(
            "ld1 {v0.16b}, [%[a]], #16\n"
            "ld1 {v1.16b}, [%[b]], #16\n"
            "add v2.16b, v0.16b, v1.16b\n"
            "st1 {v2.16b}, [%[result]], #16\n"
            : [a] "+r" (a), [b] "+r" (b), [result] "+r" (result)
            :
            : "v0", "v1", "v2", "memory"
        );
    }
    for (; i < length; i++) {
        result[i] = a[i] + b[i];
    }
}
```

어셈블리 구현과 인트린직 구현의

```
Execution time (Reference C): 3.050581 seconds
Execution time (NEON Intrinsics): 0.217088 seconds
Execution time (Assembly): 0.226368 seconds
```

Q & A