NEON Instruction Set

Outline

1. Singe-Intruction Multiple-Data(SIMD)
2. NEON Instruction Set
3. Instruction Syntax
4. Vectorizing
5. Compiler Options
6. Case Studies

1. Single-Instruction Multiple-Data(SIMD)

- The Same operation performed on multiple operands : 여러 operand에 같은 operation을 적용

```
Four 32-bit instruction
|  A0 [ ] |  A1 [ ] |  A2 [ ]  |  A3 [ ] -->128 bit
|+ B0 [ ] |+ B1 [ ] |+ B2 [ ]  |+ B3 [ ]
|_____________________________________
  C0 [ ]    C1 [ ]    C3 [ ]     C4 [ ]

Single SIMD instruction

 | A0 [ ]  A1 [ ]  A2 [ ]   A3 [ ] --> 128 bit
 | B0 [ ]  B1 [ ]  B2 [ ]   B3 [ ]
+|_____________________________________
   C0 [ ]  C1 [ ]  C3 [ ]   C4 [ ]

  이때, 각 line마다 carryed bit를 옮기면 안된다.(= carry가 prapogate되지 않음)
```

- Regiser split into elements of equal size and type
- Operantion performed on the same element.

```
NEON Vector addition(4x,short)

VADD.U16    D2, D1, D0
; // -->destination, Source operand 1, Source openrad 2

| 0x1001| 0x1234|    0x7|   0xAB|
|   +   |   +   |   +   |   +   |
|  0xFF0| 0x5678| 0xFFF8|   0xCD|
|   =   |   =   |   =   |   =   |
| 0x1FF1| 0x58AC| 0xFFFF|  0x178|
```

- **그렇다면 원래 다른 레지스터에 있던 데이터를 NEON 전용 Register로 가져와서 연산하는 것인가?**
- **만약 overflow가 생긴다면 어떻게 처리하는가?**

2. NEON Instruction Set

- Fewer cycles needed than for scalar processor
  - In general, halves number of cycles : 더 빠르게 수행
  - can sleep sooner, thereby saving power consumption
- Cost effective than a seperate DSP
  - Easier to program
  - Using the same tool chain to ARM : just difference of compiler
- Implemented as a part of ARM

3. Register File

- Physically separate 256-byte register file
  - called entension registers
- Two different views
  - 32 * doubleword(64 bits) registers
  - 16 * quad word(128 bits) registers
  - 서로 다른 레지스터가 아니라, 256 bit 레지스터를 어떤 시각으로 바라보느냐의 차이

```
32 bit = word
64 bit = doubleword

              128-bit
[0x1234     5678] [0x1234     5678] --> 2 * 64 bit data(Q)
[0x1234] [0x5678] [0x1234] [0x5678] --> 4 * 32 bit data(D)
[  ][  ] [  ][  ] [  ][  ] [  ][  ] --> 8 * 16 bit data

다만, Q + D + D 같은 형식은 불가능하다.
```

  - 레지스터의 길이는 data size에 의해 결정된다.
- Flexible register selection
  - D or Q, depending on vector lenght & data type
  - Example

```
     -----> Data type
     |
VADD.U16  D2, D1, D0
; // four 16-bit additions ----> 4 * 16 = 64 bit(1 D)
VADD.U8   D2, D1, D0
; // eight 8-bit additions ----> 8 * 8 = 64 bit(1 D)
VADD.U16  Q2, Q1, Q0
; // eight 16-bit additions ---> 8 * 16 = 128 bit(1 Q)
; // 이때, Q레지스터를 썼으므로 데이터의 크기는 총 128 bit가 됨
```

    - 64~128 bit 임의로 사용 가능?

- 최대 8 bit의 데이터를 16개까지 동시 연산 가능.(8 * 16 = 128)

4. Instructions Syntax

```
V{<mod>} <op> {<shape>} {<cond>} {.<dt>} {<dest>}, src1, src2

<mod> - Instruction Modifiers : 자주 사용하지 않음
    Q : indicates the operation uses saturating arithmetics (e.g. VQADD)
    H : indicates the operation halves the result (e.g. VHADD)
    D : indicates the operation double the result (e.g. VQDMUL)
    R : indicates the operation performs rounding (e.g. VRHADD)

<shape> - Operand Shapes : 2개의 source operand와 destination의 data size가 서로 다
를때, 어떤것을 기준으로 맞출것인지 알려줌
    L : Long operation - double the width of both operands
    W : Wide operation - double the width of th last operand
    N : Narrow operantion - halve the width of the result

<cond> - Conditional, used with IT instruction
<.dt> - Data type(one vector of vesource operand 2 우선)
      - 나머지는 operand shape을 보고 미루어 짐작한다.
<dest> - Desitination
<src1>, <src2> - source operand
```

Data Types

- Supported data types
  - Unsigned integer(U8, U16, U32, U64)
    - D : 8, 4, 2, 1 개의 vector
    - Q : 16, 8, 4, 2 개의 vector
  - Signed integer(S8, S16, S32, S64)
  - Integer of unspecified type(I8, I16, I32, I64)
  - **Floating point number(F16, F32) ------> float 형이 2바이트로 표현 가능?**
  - Polynomia(P8)
- Encoded into instruction
  - Defines the size and shape of source operand 2, if any(otherwise, source operand 1 or the equal
  - Other operands and result data type implied bu the instruction(according to operand types)
- Example

```
VADD.I16 D2, D1, D0 ; D0 contains four 16-bit integers
                    ; D1 & D2 contain the same types
ADD(op) - Add
I16(.dt) - 16-bit Integer of unspecified
```

Operand Shapes

- Suppoted operand shapes

- Long(L), Wide(W) : operand element promoted before operation
  - 연산을 진행하기 전에 D 형의 data들을 Q 형으로 확장
- Narrow(N) : operand elements demoted before operation

```
Qx : quard word(128-bit)     Dx : double word(64-bit)
1. VADDL.I16   Qd, Dn, Dm
2. VADDW.I16   Qd, Qn, Dm
3. VADDN.I32   Dd, Qn, Qm

1 : 결과는 64-bit, operands는 32-bit 일때 사용
2 : 결과는 64-bit, operand 중 하나가 32-bit 일때 사용
3 : 결과는 32-bit, operands는 64-bit 일때 사용 -> 사라지는 bit가 0이 아니라면 어찌될까?

어찌됐든 vector length는 무조건 동일해야한다
```

Example Instructions

- Base addition / subtraction

```
VADD.I16  D0, D1, D2
;// adds 4 16-bit(double-word) integer elements
;// i vector = 16bit, D = 64bit -> 4 vectors
VADD.F32  Q4, Q7, Q8
;// add 4 32-bit floating elements
;// 1 vector = 32bit, Q = 128bit -> 4 vectors
VSUBL.S32 Q8, D1, D5
;// subtract 4 16-bit(double-word) integer elements
;// result is 64-bit
```

- Absolute Difference

```
VABD.S16  D0, D1, D2
;// absolute difference of 4 16-bit integer elements
```

- Saturating Math : 자주 안쓴다.

```
VQADD.S16 D0, D1, D2
;//saturated add of 4 16bit integer elements
```

- Multiplication

```

```

```
VMUL.I8 Q0, Q1, Q2
;// multiplies 16 8-bit integaer elements-128bit = Q
VMUL.I16  D1, D7, D4[2]
;// multiplies 4 16-bit integer elements by 16-bit scalar
;// element 2 contained D4
```

- Logical operations

```
VORR Q0, Q1, Q15
; // logical OR of 128-bit
VAND D4, D7, D20
; // logical AND of 64-bit
```

- General data processing : 복사

```
VDUP.32 Q0, R3
; // R3 = 4 byte = 32 bit
; // duplicates 32-bit scalar in R3 across register Q0
; // R3 = 0xFABC -> Q0 = 0xFABC FABC FABC FABC
```

- Load / Store instructions

```
VLD1.32 {D0, D1}, [R1]!
; // load 4 32-bit elements into D0 and D1 and update R1
; // 단, vector 당 size : 32, 각 D마다 2개의 vector 할당
; // VLD1은 일반적인 경우
VST1.32 {D0, D1}, [R1]!
; // store 4 32-bit elements into D0 and D1 and update R0
```

Instruction Details

- SIMD vectors

5. Vectorizing

- Example : vector addition

```
void add_int(int * __restrict pa, int * __restrict pb, unsigned int n, int
x){
  unsigned int i;
  for(i=0; i<(n&~3); i++)
  // 기존 vector length의 하위 두 비트를 항상 0으로  만들어 가까운 4의 배수로 내
림한 수를 vector length로 설정한다.
  // 하나의 항씩 연산할때, vector length만큼 루프를 돈다.
```

```
  // 해당 코드에선 4개의 항씩 연산하므로 vector lenght/4 만큼 루프를 돈다.
    {
      pa[i] = pb[i] + x
      // 한번 연산할때마다 4개의 멤버를 한꺼번에 연산하므로 +4
    }
}
```

```
pa : pointer of A
n : vector size
pb : pointer of B

     n & ~3 bit
pb | [      ][      ][      ][      ] <--B
   |+[      ][      ][      ][      ] <--[x][x][x][x]
   |_________|________|________|________
pa | [      ][      ][      ][      ] <--A
 4word 단위로 carry bit가 관여 안함.
```

Compiler Vectorizing

- Generation of as parallel code as possible : 하나의 벡터를 연산할때 가급적 많은 항들을 우겨넣으려고 한다.
- Iteration count given as the vector length divided by (regisster length/data size) : (처리해야하는 정보량) / (레지스터당 정보량) / (data size)

example 1

```
Data type short : 2B
Register : 16B
-> vector length : 8
-> 1 register = 2 memory : 한번에 두 칸의 메모리를 불러오고, 이후 write-back 또한 이를 생각하며 해야된다.

vldmia    r2!, {d16-d17}
; // d16~d17 = q8
vldmia    r1!, {d18-d19}
; // d18~d19 = q9
vadd.i16  q8, q9, q8
; // q9과 q8의 값을 더하고 q8에 저장 -> result data는 q8에만 들어있음!
vstmia    r3!, {d16-d17}
; // q8(=d16~d17) 리턴
cmp       r3, r0
; // 루프 지속 여부를 결정.
bne       0x10b45c, <main+172>

interation count = 64 / (16B/2B) = 8
--> 전체 64개를 진행해야되는데 8개씩 한번에 진행. 이를 8번 반복.
```

```
Data type int : 4B
Register : 16B
-> vector length : 4
-> 1 register = 2 memory
  vldmia    r2!, {d16-d17}
; // d16~d17 = q8
vldmia    r1!, {d18-d19}
; // d18~d19 = q9
vadd.i32  q8, q9, q8
; // q9과 q8의 값을 더하고 q8에 저장 -> result data는 q8에만 들어있음!
vstmia    r3!, {d16-d17}
; // q8(=d16~d17) 리턴
cmp       r3, r0
; // 루프 지속 여부를 결정.
bne       0x10b45c, <main+172>

interation count = 64 / (16B/4B) = 16
```

example 2

```
usigned int vector_add_of_n(unsigned int *ptr, unsigned int items){
  unsigned int result = 0;
  unsigned int i;
  for(i=0; i<(item*4); i++)
  //
    result += ptr[i];
  return result;
}
```

```
r0[ ptr ]
r1[items]
r2[     ]
r3[     ]

vector_add_ofn PROC
    LSLS     r3, r1, #2
    MOV      r2, r0
    MOV      r0, #0
    BEQ      |L1.72|
    LSL      r3, r1, #2
    VMOV.I8  q0, #0
    LSRS     r1, r3, #2
    BEQ      |L1.48|
    ; // r2에 r0의 value(address)를 저장.
    ; // q0 register를 모두 0으로 초기화
    ;_____________________
    VLD1.32  {d2, d3}, [r2]!  ; // write back 주의
    ; // r2_value 주소에서 word * 4만큼의 데이터를 가져오고 q1(= d2,d3)에 저장
```

```
    VADD.I32 q0, q0, q1
    ; // q0 = q0 + q1
    SUBS     r1, r1, #1
    ; // r1이 0이 될때까지 반복
    ; // r1 = items 이므로
    BNE      |L1.32|        ;--> four 32-bit addition per iteration
    ;_____________________ 따라서 여기서 items만큼 반복 = c 코드 상에서 4 * items
만큼 반복
    CMP      r3, #4
    BCC      |L1.72|
    VADD.I32 d0, d0, d1
    VPADD.I32 d0, d0, d0
    VMOV.32   r1, d0[0]
    ADD       r0, r0, r1
    BX        1r
```

example 3

```
Void add_int(int * __restrict pa, int * __restrict pb, unsigned int n, int x){
  unsigned int i;
  for(i=((n&~3)>>2); i; i--)
    {
      *(pa+0) = *(pb+0) + x;
      *(pa+1) = *(pb+1) + x;
      *(pa+2) = *(pb+2) + x;
      *(pa+3) = *(pb+3) + x;
      pa+=4; pb+=4;
    }
}
```

```
r0 [   pa]
r1 [   pb]
r2 [    n]
r3 [    x]

w/ vectorizing compilation          w/o vectorizing compilation
add_int PROC                        add_int PROC
        BICS    r12, r2, #2                 MOV   r12, #0
        BEQ     |L1.40|                    PUSH  {r4}
        VDUP.32 q1, r3                     BICS  r2, r2, #3
        LSRS    r2, r2, #2                 BEQ   |L1.44|
        BEQ     |L1.40|            |L1.20|
|L1.20|                                      LDR   r4,[r1,r12,LSL, #2]
        VLD.32  {d0,d1}, [r1]!             ADD   r4, r4, r3
        VADD.I32 q0, q0, q1                STR   r4, [r0,r12,LSL,#2]
        SUBS    r2, r2, #1                 ADD   r4, r4, r3
        VST1.32 {d0, d1}, [r0]!            CMP   r4, [r0,r12,LSL,#2]
        BNE     |L1.20|                   BHI   |L1.20|
|L1.40|                                  |L1.40|
```

```
        BX      1r                          POP   {r4}
        ENDP                                BX    1r
                                            ENDP
```

- w/ vectorizing
    1. iteration count : (n & ~3)/4
    2. loop 당 처리하는 word : 4개
- w/o vectorizing
    1. iteration count : n
    2. loop 당 처리하는 word : 1개

4. Compiler Options

- Optimization of code space
- Optimization of time(speed)
- Optimization w/ knowledge of loop count
    - Plus auto-vectorizing
- Optimization w/ keyword **restrict**

example : vector addtion

```
void add_int(int *pa, int *pb, unsigned int n, int x){
  unsigned int i;
  for(i=0; i<n; i++)
    pa[i] = pb[i] + x;

}
```

- initial register setting

```
  r0[   pa]
  r1[   pb]
  r2[    n]
  r3[    x]
```

1. code space

```
add_int PROC
        PUSH    {r4, r5, 1r}
        MOW     {r4, #0}
|L0.8|
        CMP     r4, r2
        LDRCC   r5, [r1,r4,LSL,#2]
        ADDCC   r5, r5, r3
        STRCC   r5, [r0, r4, LSL, #2]
```

```
        ADCC    r4, r4, #1
        BCC     |L0.8|
        POP     {r4, r5,pc}
```

- Iteration count : n

2. time(speed)

```
add_int PREC
        CMP   r2,#0
        BXEQ  lr
_____________________________
        TST   r2,#1   ; // r2(n) and 1 -> LSB 한비트만 살려둠
                      ; // 근데 s 없어도 condition flag update?
        SUB   r1,r1,#4
        SUB   r0,r0,#4
        PUSH  {r4}
        BEQ   |L0.48| ; // 짝수면 바로 넘어간다.
        LDR   r4, [r1, #4]!
        ADD   r12, r0, #4
        ADD   r4, r4, r3
        STR   r4, [r0, #4]
        MOV   r0, r12
; // 만약 iteration 횟수가 홀수라면, 이후 진행을 위해 짝수로 맞춰준다.
_____________________________
|L0.48|
        LSRS  r2, r2, #1 ; // r2 = r2/2 -> interation 한번당 2번씩 add
        BEQ   |L0.48|    ; // 0이면 set
_____________________________
|L0.56|
        LDR   r12, [r1, #4]
        SUBS  r2, r2, #1
        ADD   r12, r12, r3
        STR   r12, [r0, #4]
        ADD   r12, r0, #8
        LDR   r4, [r1, #8]!
        ADD   r4, r4, r3
        STR   r4, [r0, #8]
        MOV   r0, r12
        BNE   |L0.56|
; // over-hanging iteration : 한번의 iteration를 돌때마다 2개씩 addtion을 실행
_____________________________
|L0.96|
        POP   {r4}
        BX    lr
```

- Iteration time : n/2 or n/2+1

3. Optimization w/ knowledge of loop count

```
void add_int(int *pa, int *pb, unsigned int n, int x){
  unsigned int i;
  for(i=0; i<(n&~3); i++)
    pa[i] = pb[i] + x;
}
```

```
add_int
    BICS  r2, r2, #3
    ; // r2 = n & ~3
    BXEQ  lr
    LSR   r2, r2, #2
    ; // r2 = n& ~3 / 4
    SUB   r1, r1, #4
    SUB   r0, r0, #4
    LSL   r2, r2, #1
    PUSH  {r4}
_____________________
|L0.28
    LDR   r12,{r1,#4}
    SUBS  r2,r2,#1
    ADD   r12,r12,r3
    STR   r12,[r0,#4]
    ADD   r12, r0, #8
    LDR   r4, [r1, #8]!
    ADD   r4, r4, r3
    STR   r4, [r0,#8]
    MOV   r0, r12
    BNE   |L0.28|
_____________________
; // tow 32-bit additions per iteration --> over-hanging iteration
    POP   {r4}
    BX    lr
```

- 2번과는 달리 4의 배수로 n을 제한하기 때문에 약간 더 짧다.

4. Optimization w/ keword

```
  void add_int(int * __restrict pa, int * __restrict pb, unsigned int n, int x){
  // restrict : pa와 pb array의 영역이 서로 겹치지 않는다는것을 명시해준다.
  // 컴파일러는 loop안에서의 data-depedance를 무시하고 vectorize 가능
    unsigned int i;
    for(i=0;i=(n&~3); i++)
      {
        pa[i] = pb[i] + x
        // 한번 연산할때마다 4개의 멤버를 한꺼번에 연산하므로 +4
      }
  }
  ```
```
```

```
add_int PROC
    BICS      r12, r2, #3
    BEQ       |L0.36|
    VDUP.32   q1,r3
    LSR       r2,r2,#2
|L0.16|
    VLD1.32   {d0,d1},[r1]!
    SUBS      r2,r2,#1
    VADD.I32   q0,q0,q1
    VST1.32   {d0,d1},[r0]!
    BNE       |L0.16|
|L0.36|
    BX        lr
```