ARM Insturction Set

Outline

1. ARMv7-A architecture
2. Endianness
3. ARM instruction set
4. Assembly language - compare with C code
5. Conditional execution
6. ARM instruction classes
7. Data Processiing Instructions
8. Load / Store Instruction
9. Branch Instruction
10. Cycle Timing of Instruction Classes
11. Case Studies : CNN

1. ARMv7-A architecture

- RISC features : small instruction & many instruction
  - Large uniform register file : 요구 register 많은 대신 일반화
  - Load/store architecture : main memory에 간섭하지 않음. operand 연산을 포함한 모든 연산 처리는 register을 걸쳐서 진행하며, ALU가 레지스터에만 연결됨.
  - simple addressing mode
- Additional Feature
  - Instructions combining *shift* with arithmetic/logical operation

```
add $t0 $s0 $t1 3
shift $t1 3-bit, then add $s0 and put in $t0
```

  - Auto-increment/decement addressing modes
  - Load/sotre multiple instructions
    - 여러 instruction을 동시에 load / store
  - conditional execution of almost all instructions
    - PSR을 통해 이전 condition을 저장.
    - instruction에 따라 바꾸는 PSR은 다르다.

Revisit PSR

- PSR(Program Status Register)
  - Condition flags
    - N = Negative result from ALU(set if negative)
    - Z = Zero result from ALU(set if zero)
    - C = ALU operation carried out(set if carried out)
    - V = ALU operation overflowed(set if over/underflow)

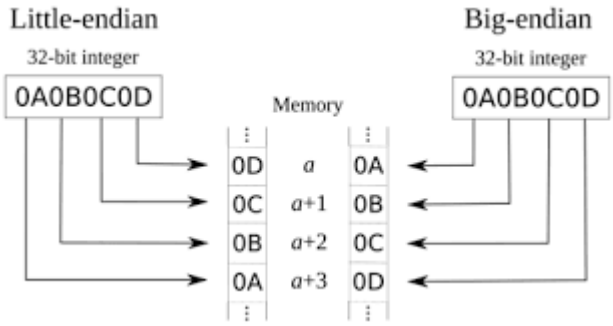
- Mode field : Current mode of processor

- CPSR : Current Program Status Register
- SPSR : Saved Program Status Register
- mode가 바뀔때 CPSR을 SPSR에 저장

2. Endianness

- 데이터를 저장하는 방법
  - 4-byte width
  - Little-endian memory system
  - Big-endian memory system
- Little-endian memory system : LSB is lowest address
- Big-endian memory system : MSB is lowest address

```
        32bit
[ 3 ][ 2 ][ 1 ][ 0 ]
 MSB            LSB
    Big-endian memory system
4   MSB [3|0] LSB
5       [2|1]
6       [1|2]
7   LSB [0|3] MSB
8       [ | ]
              Little-endian memory system
```



3. ARM Instruction Set

- All instructions are 32-bit
- Almost all instructions can be executed conditionally
- instruction consists of
  1. Opcode with (optional) condition code and "s" to set conditional code flags
     - 특수한 명령어가 아니라면 "s"가 안붙어있으면 condtional code flages 초기화 안한다.

        - "s"가 붙더라도 모든 condtional code flags가 초기화되는 것은 아님. 각 명령어들마다 초기화되는 bit는 정해져있다.
    2. Destination operand
    3. 1st source operand
    4. 2nd source operand

```
ADD R0 R1 R2
R0 : destination operand
R1 : 1st source operand
R2 : 2nd source operand
```

4. Assembly Language

- one-to-one relationship with machine instruction
- knowledge of assembly
    1. highly optimization
    2. **C <-> assembly decoding**
    3. writing compiler
    4. developing operation system

```
AREA armex, CODE, READONLY  ; 시작을 알림 + 코드의 성격을 알려줌
ENTRY
EQU     54

MOV     r0, #10
MOV     r0, #else
ADD     r2, r0, r1      ; this for comment
. . .
DCD     0xAB00321A
END
```

5. Conditional Execution

Condition Code Flags

- Included in Program Status Register(PSR)

Flag	Bit	Name	Description
N	31	Negative	set when result < 0
Z	30	Zero	set when result = 0
C	29	Carry	same as carry-out frome result or shifted out bit
V	28	Overflow	set when over/underflow

- C vs V
    1. C : unsigned over/underflow & shifted out bit

2. V : signed over/underflow
    - 그렇다면, 연산 결과가 0일때도 C가 set되는 이유는(CS)? 예외처리라 보면 되는것인가?

Condition Codes

Suffix	Flags	Meaning
EQ	Z set	equal
NE	Z clear	not equal
CS/HS	C set	Carry set / HIGH
CC/LO	C clear	Carry clear / LOW
MI	N set	Negative(Minus)
PL	N clear	Posivie or zero(PLUS)
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	Higher(unsigned >)
LS	C clear and Z set	Lower or Same(unsigned <)
GE	N and V the same	Great or Equal(signed >=)
LT	N and V differ	Less than(signed <)
GT	Z clear, N and V differ	Greater than(signed >)
LE	Z st, N and V differ	Less or Equal(signed <=)
AL	Any	Always

- 해당 조건을 만족 못하면, 코드 자체를 스킵해버린다.
    - examples

```
//Condition Passed()
//==================

boolean Condition Passed()
    cond = CurrentCond();

    // Evaluate base condition
    case cond<3:1> of
        when '000' result = (APSR.Z == '1');
        // EQ or NE
        when '001' result = (APSR.C == '1');
        // EQ or NE
        when '010' result = (APSR.N == '1');
        // EQ or NE
        when '011' result = (APSR.V == '1');
        // EQ or NE
```

```
            when '100' result = (APSR.C == '1')&&(APSR.Z == '0');
            // EQ or NE
            when '101' result = (APSR.Z == APSR.N);
            // EQ or NE
            when '110' result = (APSR.N == APSR.Z)&&(APSR.Z =='0');
            // EQ or NE
            when '111' result = TRUE;
            // EQ or NE
        // Contional flag values in the set '111x' indicate the instruction
        // is always executed
        // otherwise, invert condition if necessary
        if cond<0> == '1' && cond != '1111' then
            result = !result;
        return result;
```

Conditional Instruments

- normal instruments can't change condition flags except using 's',

```
  r0[0x005]
  r1[0x00A]

          CMP     r0, r1;         // set condition flag-------1
          ADDGT   r2, r2, #1;     // r2 = r2 + 1 if r0 > r1---2
          ADDLS   r3, r3, #1      // r3 = r3 + 1 if r0 < r1---3


         N  Z  C  V
  CPSR [ 1  0  X  0 ] ...    -> r0 < r1

  inst 2, 3 은 수행하는동안 CPSR을 업데이트 못하므로 CPSR의 상태는 inst 1~3까지
같은 상태를 유지한다.

  r1[0x005]
  r2[0x008]
  r3[0x001]

  loop    ADD     r2, r2, r3
          SUBS    r1, r1, #0x001; loop 4
          BNE     loop

              N  Z  C  V
  1st : CPSR [ 0  0  0  0 ] ...    -> r1 = 5
   .
   .
   .
              N  Z  C  V
  5st : CPSR [ 0  1  0  0 ] ...    -> r1 = 1 -> Z bit가 set이므로 BNE 작동 X
```

- example

- 1st

```
ADD     r0, r1, r2;
// DON'T UPDATE FLAG
ADDCS   r0, r1, r2;
// if carry bit set, r0 = r1 + r2. DON'T UPDATE FLAG
ADDSCS  r0, r1, r2;
// if carry bit set, r0 = r1 + r2 AND update flag
CMP     r0, r1;
// compare r0 and r1
```

- 2nd

```
C source code               unconditional           conditional
if(r0== 0)                  CMP r0, #0              CMP     r0, #0
{                           BNE else                ADDEQ   r1, r1, #1
    r1 = r1 + 1;            ADD r1, r1, #1          ADDNE   r2, r2, #1
}                           B end
else                    else
{                           ADD r2, r2, #1
    r2 = r2 + 1;        end
}
```

- 3rd : gcd(Euclid)

```
int GCD(int a, int b0){
    while(a != b){
        if(a > b)
            a = a - b;
        else
            b = b - a;
    }
    return a;
}
```

```
Conditional branches                All instructions Conditional
gcd     CMP     r0, r1              gcd     CMP     r0, r1
        BEQ     end                         SUBGT   r0, r0, r1
        BLT     less                        SUBLT   r1. r1. r0
        SUB     r0, r0, r1                  BNE     gce
        B       gcd
less
        SUB     r1, r1, r0
        B       gcd
end
```

6. Assembly Instruction Classes

- ARM instructions process data held in *registers* and only access *memory* w/ load/store instructions.
- ARM instruction classes
    - Data processing instructions
    - Branch instructions
    - Load-store instructions
    - Software instructions
    - Software instructions
    - Program Status Register Instructions

7. Data Processing Instruction

- Syntax : **Operation {cond} {s} Rd, Rn, Operand2**
- operand 2 에선 shift연산을 우선실행한다.(flexible operand)
- 대부분의 명령어들은 s가 붙었을 경우에만 일부 CPSR을 업데이트 한다.

Arithmetic Operations

Opcode	Operands	Descriptions	Functions
ADC	Rd, Rn, Op2	Add with carry	Rd = Rn + Op2 + C
ADD	Rd, Rn, Op2	Add	Rd = Rn + Op2
MOW	Rd, Op2	Move	Rd = Op2
MVN	Rd, Op2	Move NOT	Rd = !Op2
RSB	Rd, Rn, Op2	Reverse Subtract	Rd = Op2 - Rn
RSC	Rd, Rn, Op2	Reverse Subtract with Carry	Rd = Op2 - Rn - !C
SBC	Rd, Rn, Op2	Subtract with Carry	Rd = Rn - Op2 - !C
SUB	Rd, Rn, Op2	Subtract	Rd = Rn - Op2

Logical Operations & Flag-Setting Instructions

- Logical operations

Opcode	Operands	Descriptions	Functions
AND	Rd, Rn, Op2	AND	Rd = Rn &Op2
BIC	Rd, Rn, Op2	Bit Clear	Rd = Rn & ~Op2
EOR	Rd, Rn, Op2	Exclusive OR	Rd = Rn ^ Op2
ORR	Rd, Rn, Op2	OR	(Rd = Rn

- Flag setting instructions

Opcode	Operands	Descriptions	Functions
CMP	Rn, Op2	Compare	Rn - Op2

Opcode	Operands	Descriptions	Functions
CMN	Rn, Op2	Compare negative	Rn + Op2
TEQ	Rn, Op2	Test Equivalence	Rn ^ Op2
TST	Rn, Op2	Test	Rn & Op2

ALU & Flag Setting Instructions

- SUBS vs CMP
  - 둘 다 Rn - Op2 라는 연산을 하고 Conditino Code Flag를 업데이트 하는것은 동일하지만, SUBS는 해당 연산의 결과를 Rd에다 저장한다. 그에 반해 CMP는 연산 결과를 따로 저장해두지는 않음.

Flexible Operand

- 2nd Source Operand(Op2) being flexible(shift & immediate value)
  1. shift : 5-bit unsigned inter or constant
  2. Multiply
  3. Immediate Value

```
RSB R0, R1, R2, LSL #2;   // R0 = (R2<<2) -R1
MOV R0, R1, LSR R3;       // R0 = R1<<R3
ADD R0, R5, R5, LSL #1;   // R0 = R5 + R5<<2 = 3*R5
SUB R0, R1, #100;         // R0 = R1 - 100
```

Shift

- Shift Operation
  - LSL
    - 왼쪽으로 shift : 2의 거듭제곱만큼 곱하는 형태
    - LSB의 비는 비트는 0으로 채운다
    - 마지막으로 나온 MSB는 condition flag의 Carry Flag Bit에 저장

```
                  LSL A 2                     LSL A 1
A[0x0123] C[X]  ------->  A[0x2300] C[1]  ------->  A[0x3000] C[2]
```

  - LSR
    - 오른쪽으로 shift
    - 마지막으로 나온 LSB는 마찬가지로 condition flag의 Carry Flag Bit에 저장
    - ASR
    - LSR의 signed 연산
    - 2's complement 방식으로 연산하는 경우, 음수를 shift하는 경우 부호가 바뀐다 -> 따라서 ASR을 사용

```
                  LSR A 2                     LSR A 1
A[0x0123]  C[X]  ------->  A[0x0001]  C[2]  ------->  A[0x0000]  C[1]
```

- ROR
  - immediate 값만큼 오른쪽으로 shift연산 실행
  - LSB쪽에서 버려진 값을 MSB쪽으로 끼워 넣는다

```
                  ROR A 2                     ROR A 1
A [0x0123]  C[X]  ------->  A [0x2301]  C[2]  ------->  A [0x1230]
C[1]
```

- RRX
  - Carry Flag bit에 있는값을 공백에 채운다. 이때, 밀려난 비트는 Carry Flag Bit에 저장한다.

```
                  RRX A 2                     RRX A 1
A[0x0123]  C[0]  ------->  A[0x3001]  C[2]  ------->  A[0x2300]  C[1]
```

Multiply

- Executed in dedicated unit
  - 32 * 32 multiply w/ 32-bit truncation - LS Word (2~5 cycles) : 결과 최대 64 bit 중 하위 32 bit를 저장. `text MUL R0 R1 R2 ; // Multiply : R0 = R1 * R2 만약 32bit 이상일때 하위 32비트만 가져온다. MLA R0 R1 R2 R3 ; // Multiple and add : R0 = (R1 * R2) + R3`
  - 32 * 32 multiply w/o truncation (64 bit result)

```
[U | S]MULL R4 R5 R2 R3
; // [R5 : R4] = R2 * R3 32 bit 이상의 결과를 2개의 레지스터에 나누어 저장
// U : unsigned, S : signed
[U | S]MLAL R4 R5 R2 R3
; // [R5 : R4] = R2 * R3 + [R5 : R4]
; // 초기화 되기 전 [R5 : R4]를 더해준다
; // 그렇다면 둘의 합이 0xFFFF를 넘었을땐? 하위 4bit만 저장하는가?

MLAL R4 R5 R2 R3 = [R5 : R4] = R2 * R3 + [R5 : R4]
R2 [0x000A] = 10            R2 [0x0008]
R3 [0x0008] =  8     ->     R3 [0xFFFF] R2 * R3 = 0x0FFF F000
R4 [0x0010] = 16            R4 [0xF010] 0xF000 + 0x0010 = 0xF010
R5 [0x0001] =  1            R5 [0x1000] 0x0FFF + 0x0001 = 0x1000
```

Opcode	Operands	Description	Function
MLA	Rd, Rn, Rm, Ra	Multiply accumulate(MAC)	Rd = Ra + (Rm * Rn)
MLS	Rd, Rn, Rm, Ra	Multiply and Subtract	Rd = Ra - (Rm * Rn)

Opcode	Operands	Description	Function
MUL	Rd, Rn, Rm	Multiply	Rd = Rn * Rm
SMLAL	RdLo, RdHi, Rn, Rm	Signed 32-bit multiply with a 64-bit accumulate	[RdHi : RdLo] += Rn * Rm
SMULL	RdLo, RdHi, Rn, Rm	Signed 64-bit multiply	[RdHi : RdLo] = Rn * Rm
UMLAL	RdLo, RdHi, Rn, Rm	Unsigned 64-bit MAC	[RdHi : RdLo] += Rn * Rm
UMULL	RdLo, RdHi, Rn, Rm	Unsigned 64-bit Multiply	[RdHi : RdLo] = Rn * Rm

Divide

- 일반적인 암코어 프로세서는 디바이드 프로세서 없음. -> shift를 이용 or function 불러옴. 주로 나누는 값을 연속적으로 빼는 형식으로 주로 구현.

8. Load / Store Instruction

- store : reg의 데이터를 memory에 저장
- load : memory의 데이터를 reg로 가져옴

Single Resigster Data Transfer

- Syntax
  - load : LDR (cond) (size) Rd (address)
  - store : STR (cond) (size) Rd (address)
- Supported access sizes
  - LDR, STR : word
  - LDRB, STRB : byte
  - LDRH, STRH : half word
  - LDRSB : singed byte load
  - LDRSH : singed half byte load
- Addressing Mode
  1. Register addressing : 가장 일반적인 LDR

```
LDR R0 [R1]   ; address pointed by R1
```

  2. Pre-indexed addressing : 더하는 연산을 load하기 이전에 수행하기 때문에 pre-index라고 불림.

```
LDR R0 [R1 R2]         ; address pointed to by R1 + R2
LDR R0 [R1 #32]        ; address pointed to by R1 + 32
LDR R0 [R1 R2 LSL #2]  ; address pointed to by R1 + (R2<<2)
```

3. Pre-indexed with write-back : 본래 레지스터도 같이 변경(write-back)

```
LDR R0 [R1 #32]!       ; address pointed to by R1 + 32
                       ; then R1 := R1+32
```

4. Post-indexed with write-back : 데이터 참조 하고 address 변화

```
LDR R0 [R1] #32
; Read R0 from address pointed to by R1, then R1 : = R1 +32
```

- example

```
STR R0 [R1 #12] : ! 붙으면 write-back 또한 수행.
0x20c [0x05] <===== 3. R0 [0x05]
0x208 [    ]
0x204 [    ]        2. offset [0x00c]
0x200 [    ] <===== 1. R1 [0x200]

STR R0 [R1] #12
0x20c [0x05] <===== R1 [0x20c](NEW!)
0x208 [    ]
0x204 [    ]        offset [0x00c]
0x200 [0x05] <===== R1 [0x200], R0 [0x05]
```

- Memory Block Copy : Post-index addressing

```
loop    LDR r0, [r8], #4
        STR r0, [r10], #4
        CMP r0, r9
        BLT loop
    1. r0에 r1의 가리키는 주소의 데이터를 가져오고, r1에 4를 더한다.
    2. r0의 값을 r10이 가르키는 destination에 저장하고 r10에 4를 더한다.
    3. r8과 r9의 값을 비교한다

- 어째서 r8과 r9을 비교하는가? r8과 r9은 둘다 데이터가 변하지 않는다
- r8가 가르키는 주소에 포인터가 들어있다면 해당 포인터를 한번 더 참조하는가?

        Area    Word    Code    Readonly
num     EQU     20
        ENTRY
start
        LDR     r0      =src
        LDR     r1      =dst
        MOW     r2      #num
wordcopy
```

```
        LDR     r3      [r0]    #4
        STR     r3      [r1]    #4
        SUBS    r2      r2      #1
        // 여기선 비교 대신 움직일때마다 워드를 하나씩 빼는 형식으로 구현
        BNE     wordcopy

stop                //프로그램을 끝내는 부분. 굳이 신경쓰지 말자.
        MOV     r0      #0x18
        LDR     r1      =0x20026
        SWI     0x123456
```

Multiple Register Data Transfers

- syntax
  - LDM (cond) (addr_mode) Rd{!}, (reg_list)
  - STM (cond) (addr_mode) Rd{1}, (reg_list)
- addressing mode LDMIA / STMIA = increment after(default) SDMIB / STMIB = increment before LDMDA / STMDA = decrement after LDMDB / STMDB = decrement before

```
                                r0 [0x012]  r1[0x015] r4 [0x019]

                                   IA      IB      DA      DB
                                 [     ] [0x019] [     ] [     ]
LDMxx r10, {r0, r1, r4}          [0x019] [0x015] [     ] [     ]
STMxx r10, {r0, r1, r4}          [0x015] [0x012] [     ] [     ]
                           r10-> [0x012] [     ] [0x012] [     ]
                                 ____________________________________
                                 [     ] [     ] [0x015] [0x012]
                                 [     ] [     ] [0x019] [0x015]
                                 [     ] [     ] [     ] [0x019]
```

- Memory Block Copy

```
; r8 points to start of source data
; r9 points to end of source data                          [     ]  | 이동
; r10 points to start of destination data                  [     ]  | <----
                                                      r10 [     ]---     |
loop      LDMIA   r8!,    {r0-r7} ;load 32 bytes          [     ]        |
          STMIA   r10!,   {r0-r7} ; and store them    r9  [     ]--      |
          CMP     r8, r9          ; check for the end     [     ] |      |
          BLT     loop            ; and loop              [     ] | data--
                                                      r8  [     ]--
```

1. Load data r0~r7 to r8~. Then, write-back r8
2. Store data r0~r7 to r10~. Then, write-back r10
3. if start of source data is same as end of source data, end

9. Branch Instruction

Branches with LDR

```
LDR pc, =label        ;       load address of label into PC

branches anywhere within the 4GB address space are thus possible

- PC를 바꿀땐 branch와 비슷한 역할. 그러면 pc자리에 레지스터 주소가 들어가면?

      LDR r0, =src
_____________________________________________
src   DCD 1, 2, 1, 3, 5, 6                  <--------literal pool address data
```

- Literal Pool : 리터럴 데이터들이 모여있는 곳
- Literal
    1. 프로그래밍 언어에서 직접 값을 나타내는 자구 단위. 예를 들면, 14는 실정 14를 나타내고, ‘APRIL’은 문자열 APRIL을 나타내며, 3.0005E2는 수 300.05를 나타낸다.
    2. 원시 프로그램 중에 있는 기호 또는 양(quantity). 다른 데이터를 참조하지 않고 그 자신이 데이터가 되어 있는 것으로, 원시 프로그램의 번역 중에 그 값을 변경해서는 안 되는 것이다.

Branch Instructions

- Syntax : B{L} {cond} label
    - Subroutine calls with optional "L"
- 24-bit address offest (left shifted by 2) for a relative branch range of 32MB
- Causes a pipeline flush

```
      B start <------ perfom PC relative branch to label "start"
.
.
.
start         <------ continue execution from here
```

Subroutines : branch and link

- return address stored in link register(1r/r14)
- branch to address of +-32MB
- returned by restoring pc from 1r

10. Cycle Timing of Instruction Classes

- Cycle time은 외부 요인(Cache, Pipeline 등)에 영향을 많이 받으므로 정확하게 측정 불가.
- 주로 사용하는 측정 방법
    1. not based on (static) instruction / cycle count

2. Averaged over a sufficiently long period of time
3. timer-based measurement preferred

11. Case Studies : CNN