

ARM Instruction Set Architecture

Jin-Soo Kim (*jinsookim@skku.edu*)

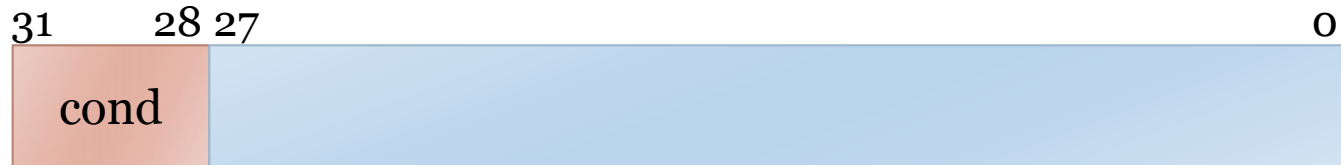
Computer Systems Laboratory

Sungkyunkwan University

<http://csl.skku.edu>

Condition Field (1)

- Most ARM instructions can be conditionally executed.
- NOP if the flags in CPSR do not satisfy the condition.
- Every instruction contains a 4-bit condition code field:
 - `ADDEQ R0, R1, R2` ; $R0 = R1+R2$ if Z set



Condition Field (2)

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0, N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-

Data-Processing Insts. (1)

Opcode	Mnemonic	Operation	Action
0000	AND	Logical AND	$Rd := Rn \text{ AND shifter_operand}$
0001	EOR	Logical Exclusive OR	$Rd := Rn \text{ EOR shifter_operand}$
0010	SUB	Subtract	$Rd := Rn - \text{shifter_operand}$
0011	RSB	Reverse Subtract	$Rd := \text{shifter_operand} - Rn$
0100	ADD	Add	$Rd := Rn + \text{shifter_operand}$
0101	ADC	Add with Carry	$Rd := Rn + \text{shifter_operand} + \text{Carry Flag}$
0110	SBC	Subtract with Carry	$Rd := Rn - \text{shifter_operand} - \text{NOT}(\text{Carry Flag})$
0111	RSC	Reverse Subtract with Carry	$Rd := \text{shifter_operand} - Rn - \text{NOT}(\text{Carry Flag})$
1000	TST	Test	Update flags after $Rn \text{ AND shifter_operand}$
1001	TEQ	Test Equivalence	Update flags after $Rn \text{ EOR shifter_operand}$
1010	CMP	Compare	Update flags after $Rn - \text{shifter_operand}$
1011	CMN	Compare Negated	Update flags after $Rn + \text{shifter_operand}$
1100	ORR	Logical (inclusive) OR	$Rd := Rn \text{ OR shifter_operand}$
1101	MOV	Move	$Rd := \text{shifter_operand}$ (no first operand)
1110	BIC	Bit Clear	$Rd := Rn \text{ AND NOT}(\text{shifter_operand})$
1111	MVN	Move Not	$Rd := \text{NOT shifter_operand}$ (no first operand)

Data-Processing Insts. (2)

- 16 arithmetic & logical operations
- Most instructions take two source operands (except MOV & MVN)
 - One is always a register
 - The other (shifter operand) is an immediate value or a register (a shift applied optionally)
- CMP, CMN, TST, TEQ always update the condition codes. Add “S” to the instruction mnemonic for other instructions.

Shifter Operand (1)

- Immediate operand
 - An 8-bit constant rotated (to the right) by an even number of bits (0, 2, 4, 8, ..., 30)
 - MOV R0, #0 ; R0 = 0
 - ADD R3, R3, #1 ; R3 += 1
 - CMP R7, #1000 ; (R7 - 1000)?
 - BIC R9, R8, #0xFF00
- Register operand
 - R0 – R15

Shifter Operand (2)

- Shifted register operand
 - The value of a register, shifted (or rotated) before it is used as the data-processing operand
 - The shift amount is given by an immediate value or by the value of register

ASR	Arithmetic shift right
LSL	Logical shift left
LSR	Logical shift right
ROR	Rotate right
RRX	Rotate right extended with C

Shifter Operand (3)

- Example:

- MOV R2, R0, LSL #2 ; R2 = R0*4
- ADD R9, R5, R5, LSL #3 ; R9 = R5*9
- RSB R9, R5, R5, LSL #3 ; R9 = R5*7
- SUB R1, R9, R8, LSR #4 ; R1 = R9 - R8/16
- MOV R2, R4, ROR R3
; R2 = R4 rotated right by value of R3

Multiply Insts.

- Normal multiply:
32-bit x 32-bit -> bottom 32-bit result
 - MUL (Multiply)
 - MLA (Multiply and Accumulate)
 - No multiply-by-constant instructions
- Example
 - MUL R4, R2, R1 ; $R4 = R2 * R1$
 - MLA R7, R8, R9, R3 ; $R7 = R8 * R9 + R3$

Load/Store Insts.

- Load and store instructions

LDR	Load Word
LDRB	Load Byte
LDRH	Load Unsigned Halfword
LDRSB	Load Signed Byte
LDRSH	Load Signed Halfword
STR	Store Word
STRB	Store Byte
STRH	Store Halfword
LDM	Load Multiple
STM	Store Multiple

Addressing Modes (1)

- Immediate offset:
[<Rn>, #+/-<offset_12>]
 - LDR R1, [R2, #16] ; R1 = Mem[R2 + 16]
- Register offset: [<Rn>, +/-<Rm>]
 - LDR R1, [R2, -R3] ; R1 = Mem[R2 - R3]
- Scaled register offset:
[<Rn>, +/-<Rm>, <shift> #<shift_imm>]
 - LDR R1, [R2, R3, LSL #2]
; R1 = Mem[R2 + R3*4]

Addressing Modes (2)

- Immediate pre-indexed:
[<Rn>, #+/-<offset_12>]!
 - LDR R1, [R2, #4]!
; R1 = Mem[R2 + 4]; R2 = R2 + 4
- Register pre-indexed:
[<Rn>, +/-<Rm>]!
- Scaled register pre-indexed:
[<Rn>, +/-<Rm>, <shift> #<shift_imm>]!

Addressing Modes (3)

- Immediate post-indexed:
[<Rn>], #+/-<offset_12>
 - LDR R1, [R2], #4
; R1 = Mem[R2]; R2 = R2 + 4
- Register post-indexed:
[<Rn>], +/-<Rm>
- Scaled register post-indexed:
[<Rn>], +/-<Rm>, <shift> #<shift_imm>

ADR Pseudo-op

- ADR: Set register to address
- Cannot refer to an address directly in an instruction
- Generate value by performing arithmetic on PC

```
ADR R1, foo
LDR R0, [R1]
...
foo: .word 123
```

```
ADD R1, PC, #4
LDR R0, [R1]
...
foo: .word 123
```

Example #1

- $x = (a + b) - c;$

```
ADR  R4, a           ; Get address of a
LDR  R0, [R4]        ; R0 = a
ADR  R4, b           ; Get address of b
LDR  R1, [R4]        ; R1 = b
ADD  R3, R0, R1      ; R3 = a + b
ADR  R4, c           ; Get address of c
LDR  R2, [R4]        ; R2 = c
SUB  R3, R3, R2      ; R3 = (a + b) - c
ADR  R4, x           ; Get address of x
STR  R3, [R4]        ; Store R3 to x
```

Example #2

- $z = (a \ll 2) | (b \& 15);$

```
ADR  R4, a           ; Get address of a
LDR  R0, [R4]        ; R0 = a
MOV  R0, R0, LSL #2  ; R0 = a << 2
ADR  R4, b           ; Get address of b
LDR  R1, [R4]        ; R1 = b
AND  R1, R1, #15     ; R1 = (b & 15)
ORR  R1, R0, R1      ; R1 = (a << 2) | (b & 15)
ADR  R4, z           ; Get address of z
STR  R1, [R4]        ; Store R1 to z
```


LDM/STM (1)

- Load/store a subset of GPRs to/from Mem.
- IA (Increment After) -- default
 - LDM R8, {R0, R2, R9}
 - LDM R8!, {R0, R2, R9}
; The final address is written back into R8
- IB (Increment Before)
- DA (Decrement After)
- DB (Decrement Before)

LDM/STM (2)

- **Stack addressing**
 - ARM uses Fully Descending stack
 - SP points to the last used location
 - Grow towards decreasing memory addresses
 - Use STMFD to push a set of registers to stack:
STMFD = STMDB (Decrement Before)
 - Use LDMFD to pop a set of data from stack:
LDMFD = LDMIA (Increment After)
- **Example:**
 - STMFD SP!, {R0-R12, LR}
 - LDMFD SP!, {R0-R12, PC}

Branch Insts.

- Unconditional jump
 - B Exit ; Jump to the label Exit
 - BX LR ; Jump, switching to Thumb mode
- Conditional branches
 - Use condition codes
 - BEQ, BNE, BCS, BCC, BMI, BPL, BVS, BVC, BHI, BLS, BGE, BLT, BGT, BLE
 - CMP Ro, #9 ; if (Ro > 9) goto L1;
 BGT L1

FIR Filters (1)

- Finite Impulse Response (FIR) filter

$$f = \sum_{0 \leq i < N} c_i x_i$$

```
for (i = 0, f = 0; i < N; i++)  
    f = f + c[i] * x[i];
```



```
i = 0;  
f = 0;  
while (i < N) {  
    f = f + c[i] * x[i];  
    i++;  
}
```

FIR Filters (2)

```
i = 0;
f = 0;
while (i < N) {
    f = f + c[i] * x[i];
    i++;
}
```

```
MOV  R0, #0    ; R0 = 0 (i)
MOV  R8, #0    ; R8 = 0 (i * 4)
ADR  R2, N     ; Get address of N
LDR  R1, [R2]  ; R1 = N
MOV  R2, #0    ; R2 = 0 (f)
ADR  R3, c     ; Get address of c
ADR  R5, x     ; Get address of x
```

```
loop
LDR  R4, [R3, R8] ; R4 = c[i]
LDR  R6, [R5, R8] ; R6 = x[i]
MUL  R4, R4, R6   ; R4 = c[i]*x[i]
ADD  R2, R2, R4   ; f += c[i]*x[i]
ADD  R8, R8, #4   ; R8 += 4
ADD  R0, R0, #1   ; i += 1
CMP  R0, R1       ; if (i < N)
BLT  loop        ; Continue
```

Procedure Linkage

- Procedure call
 - `BL foo ; LR = PC, goto foo()`
 - `BLX foo ; Calls Thumb subroutine`
 - `BLX R1 ; Calls ARM or Thumb subroutine`
(Thumb if `R1 & 0x1 == 1`)
- Procedure return
 - `MOV R15, R14 ; PC = LR`
 - Return value in `R0` (and `R1`, if needed)

Miscellaneous Instructions

- **MRS: Move PSR to GPR**
 - `MRS R1, CPSR` ; R1 = CPSR
- **MSR: Move to PSR from GPR**
 - `CPSR_[c|x|s|f]`: control, extension, status, flags
 - `MSR CPSR_c, R1` ; Set the control field
- **MRS & MSR used to enable/disable interrupts/FIQs**