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:
  - $\text{ADDEQ R0, R1, R2} \quad ; \quad \text{R0 = R1+R2 if Z set}$
## Condition Field (2)

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

- 16 arithmetic & logical operations
- These instructions only work on registers, NOT memory

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>ADD</th>
<th>ADC</th>
<th>SUB</th>
<th>SBC</th>
<th>RSB</th>
<th>RSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical</td>
<td>AND</td>
<td>ORR</td>
<td>EOR</td>
<td>BIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comparisons</td>
<td>CMP</td>
<td>CMN</td>
<td>TST</td>
<td>TEQ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data movement</td>
<td>MOV</td>
<td>MVN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Data-Processing Insts. (2)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Mnemonic</th>
<th>Operation</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>AND</td>
<td>Logical AND</td>
<td>Rd := Rn AND shifter_operand</td>
</tr>
<tr>
<td>0001</td>
<td>EOR</td>
<td>Logical Exclusive OR</td>
<td>Rd := Rn EOR shifter_operand</td>
</tr>
<tr>
<td>0010</td>
<td>SUB</td>
<td>Subtract</td>
<td>Rd := Rn - shifter_operand</td>
</tr>
<tr>
<td>0011</td>
<td>RSB</td>
<td>Reverse Subtract</td>
<td>Rd := shifter_operand - Rn</td>
</tr>
<tr>
<td>0100</td>
<td>ADD</td>
<td>Add</td>
<td>Rd := Rn + shifter_operand</td>
</tr>
<tr>
<td>0101</td>
<td>ADC</td>
<td>Add with Carry</td>
<td>Rd := Rn + shifter_operand + Carry Flag</td>
</tr>
<tr>
<td>0110</td>
<td>SBC</td>
<td>Subtract with Carry</td>
<td>Rd := Rn - shifter_operand - NOT(Carry Flag)</td>
</tr>
<tr>
<td>0111</td>
<td>RSC</td>
<td>Reverse Subtract with Carry</td>
<td>Rd := shifter_operand - Rn - NOT(Carry Flag)</td>
</tr>
<tr>
<td>1000</td>
<td>TST</td>
<td>Test</td>
<td>Update flags after Rn AND shifter_operand</td>
</tr>
<tr>
<td>1001</td>
<td>TEQ</td>
<td>Test Equivalence</td>
<td>Update flags after Rn EOR shifter_operand</td>
</tr>
<tr>
<td>1010</td>
<td>CMP</td>
<td>Compare</td>
<td>Update flags after Rn - shifter_operand</td>
</tr>
<tr>
<td>1011</td>
<td>CMN</td>
<td>Compare Negated</td>
<td>Update flags after Rn + shifter_operand</td>
</tr>
<tr>
<td>1100</td>
<td>ORR</td>
<td>Logical (inclusive) OR</td>
<td>Rd := Rn OR shifter_operand</td>
</tr>
<tr>
<td>1101</td>
<td>MOV</td>
<td>Move</td>
<td>Rd := shifter_operand (no first operand)</td>
</tr>
<tr>
<td>1110</td>
<td>BIC</td>
<td>Bit Clear</td>
<td>Rd := Rn AND NOT(shifter_operand)</td>
</tr>
<tr>
<td>1111</td>
<td>MVN</td>
<td>Move Not</td>
<td>Rd := NOT shifter_operand (no first operand)</td>
</tr>
</tbody>
</table>
Data-Processing Insts. (3)

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, #100  ; (R7 − 100)?
  - 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

<table>
<thead>
<tr>
<th>ASR</th>
<th>Arithmetic shift right</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSL</td>
<td>Logical shift left</td>
</tr>
<tr>
<td>LSR</td>
<td>Logical shift right</td>
</tr>
<tr>
<td>ROR</td>
<td>Rotate right</td>
</tr>
<tr>
<td>RRX</td>
<td>Rotate right extended with C</td>
</tr>
</tbody>
</table>
Shifter Operand (3)

- **LSL**: Logical Left Shift
  - Multiplication by a power of 2

- **LSR**: Logical Shift Right
  - Division by a power of 2

- **ASR**: Arithmetic Shift Right
  - Division by a power of 2, preserving the sign bit

- **ROR**: Rotate Right
  - Bit rotate with wrap around from LSB to MSB

- **RRX**: Rotate Right Extended
  - Single bit rotate with wrap around from CF to MSB
Shifter Operand (4)

- **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 Instructions

- 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 Instrs.

- Load and store instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>Load Word</td>
</tr>
<tr>
<td>LDRB</td>
<td>Load Unsigned Byte</td>
</tr>
<tr>
<td>LDRH</td>
<td>Load Unsigned Halfword</td>
</tr>
<tr>
<td>LDRSB</td>
<td>Load Signed Byte</td>
</tr>
<tr>
<td>LDRSH</td>
<td>Load Signed Halfword</td>
</tr>
<tr>
<td>STR</td>
<td>Store Word</td>
</tr>
<tr>
<td>STRB</td>
<td>Store Byte</td>
</tr>
<tr>
<td>STRH</td>
<td>Store Halfword</td>
</tr>
<tr>
<td>LDM</td>
<td>Load Multiple</td>
</tr>
<tr>
<td>STM</td>
<td>Store Multiple</td>
</tr>
</tbody>
</table>
Addressing Modes (1)

- Immediate offset: 
  \[ [<Rn>, \#+/-%offset_{12}>]\n  - LDR \ R1, [R2, #16] ; R1 = Mem[R2 + 16]

- Register offset: 
  \[ [<Rn>, +/-<Rm>]\n  - LDR \ R1, [R2, -R3] ; R1 = Mem[R2 - R3]

- Scaled register offset: 
  \[ [<Rn>, +/-<Rm>, <shift> \#<shift_imm>]\n  - LDR \ R1, [R2, R3, LSL #2] ; R1 = Mem[R2 + R3*4]
Addressing Modes (2)

- Immediate pre-indexed:
  \[ [<Rn>, \#+/-%<offset_{12}>]! \]
  - LDR \ R1, [R2, \#4]!
    \[; R1 = \text{Mem}[R2 + 4]; R2 = R2 + 4\]

- 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

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register/Address</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADR</td>
<td>R4, a</td>
<td>Get address of a</td>
</tr>
<tr>
<td>LDR</td>
<td>R0, [R4]</td>
<td>R0 = a</td>
</tr>
<tr>
<td>ADR</td>
<td>R4, b</td>
<td>Get address of b</td>
</tr>
<tr>
<td>LDR</td>
<td>R1, [R4]</td>
<td>R1 = b</td>
</tr>
<tr>
<td>ADD</td>
<td>R3, R0, R1</td>
<td>R3 = a + b</td>
</tr>
<tr>
<td>ADR</td>
<td>R4, c</td>
<td>Get address of c</td>
</tr>
<tr>
<td>LDR</td>
<td>R2, [R4]</td>
<td>R2 = c</td>
</tr>
<tr>
<td>SUB</td>
<td>R3, R3, R2</td>
<td>R3 = (a + b) - c</td>
</tr>
<tr>
<td>ADR</td>
<td>R4, x</td>
<td>Get address of x</td>
</tr>
<tr>
<td>STR</td>
<td>R3, [R4]</td>
<td>Store R3 to x</td>
</tr>
</tbody>
</table>
**Example #2**

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADR R4, a</td>
<td>Get address of a</td>
</tr>
<tr>
<td>LDR R0, [R4]</td>
<td>R0 = a</td>
</tr>
<tr>
<td>MOV R0, R0, LSL #2</td>
<td>R0 = a &lt;&lt; 2</td>
</tr>
<tr>
<td>ADR R4, b</td>
<td>Get address of b</td>
</tr>
<tr>
<td>LDR R1, [R4]</td>
<td>R1 = b</td>
</tr>
<tr>
<td>AND R1, R1, #15</td>
<td>R1 = (b &amp; 15)</td>
</tr>
<tr>
<td>ORR R1, R0, R1</td>
<td>R1 = (a &lt;&lt; 2)</td>
</tr>
<tr>
<td>ADR R4, z</td>
<td>Get address of z</td>
</tr>
<tr>
<td>STR R1, [R4]</td>
<td>Store R1 to z</td>
</tr>
</tbody>
</table>
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)

Example

LDMxx r10, {r0,r1,r4}
STMxx r10, {r0,r1,r4}

Base Register (Rb) r10

Increasing Address
LDM/STM (3)

- 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 Instructions

- 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 R0, #9 ; if (R0 > 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++)
\[
f = f + c[i] \times x[i];
\]

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

i = 0;
f = 0;
while (i < N) {
    f = f + c[i] \times x[i];
    i++;}
**FIR Filters (2)**

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

**Assembly Code**

```assembly
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
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 (1)

- **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)
Procedure Linkage (2)

- Example

```assembly
func1:
  STMFD sp!,{regs,lr}
  BL func2
  LDMFD sp!,{regs,pc}
  MOV pc, lr

func2:
```
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**