Modern Assembly Language Programming
with the
ARM processor
Chapter 4: Data Processing and Special Instructions
Thirteen general-purpose registers (r0–r12)
- The stack pointer (r13 or sp)
- The link register (r14 or lr)
- The program counter (r15 or pc)
- Current Program Status Register (CPSR)
Instruction Categories

- Load/Store Instructions
- Data processing Instructions
  - Arithmetic Operations
  - Logical Operations
  - Comparison Operations
  - Data Movement Operations
  - Multiplication Operations
- Branch Instructions
  - Branch with Link (subroutine call)
  - Conditional Branches
- Special Instructions
- Pseudo-Instructions
Hardware-Related Register Use

- All instructions can access $r0-r14$ directly.
- Most instructions also allow use of the program counter ($r15$).
- Specific instructions to allow access to CPSR.
- $r14$, $r15$, and CPSR are “hardware special”.
The CPSR contains four “flag” bits (bits 28-31).

Data Processing Instructions can affect the flag bits (if the programmer wants them to).

The meaning of the flags depends on the type of instruction that set them.

These bits can be used by subsequent instructions to control execution and branching.

*Most* instructions can have a condition attached, to control whether or not they are actually executed.
CPSR Condition Bits

The following table shows the meaning of the four bits depending on the type of instruction that set or cleared them.

<table>
<thead>
<tr>
<th>Name</th>
<th>Logical Instruction</th>
<th>Arithmetic Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>No meaning</td>
<td>Bit 31 of the result has been set. Indicates a negative number in signed operations.</td>
</tr>
<tr>
<td>Z</td>
<td>Result is all zeroes</td>
<td>Result of operation was zero</td>
</tr>
<tr>
<td>C</td>
<td>After Shift operation ‘1’ was left in carry flag</td>
<td>Result was greater than 32 bits</td>
</tr>
<tr>
<td>V</td>
<td>No meaning</td>
<td>Result was greater than 31 bits. Indicates a possible corruption of the sign bit in signed numbers.</td>
</tr>
</tbody>
</table>
## Conditional Execution

\[
\text{op\{<cond>\}} \quad \text{operands}
\]

<table>
<thead>
<tr>
<th>&lt;cond&gt;</th>
<th>English meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>al</td>
<td>always (this is the default &lt;cond&gt;)</td>
</tr>
<tr>
<td>eq</td>
<td>(Z \text{ set (}=))</td>
</tr>
<tr>
<td>ne</td>
<td>(Z \text{ clear} (\neq))</td>
</tr>
<tr>
<td>ge</td>
<td>(N \text{ set and } V \text{ set, or } N \text{ clear and } V \text{ clear} (\geq))</td>
</tr>
<tr>
<td>lt</td>
<td>(N \text{ set and } V \text{ clear, or } N \text{ clear and } V \text{ set} (&lt;))</td>
</tr>
<tr>
<td>gt</td>
<td>(Z \text{ clear, and either } N \text{ set and } V \text{ set, or } N \text{ clear and } V \text{ set} (&gt;)</td>
</tr>
<tr>
<td>le</td>
<td>(Z \text{ set, or } N \text{ set and } V \text{ clear, or } N \text{ clear and } V \text{ set} (\leq))</td>
</tr>
<tr>
<td>hi</td>
<td>(C \text{ set and } Z \text{ clear (unsigned &gt;)})</td>
</tr>
<tr>
<td>ls</td>
<td>(C \text{ clear or } Z \text{ (unsigned } \leq))</td>
</tr>
<tr>
<td>hs</td>
<td>(C \text{ set (unsigned } \geq))</td>
</tr>
<tr>
<td>cs</td>
<td>Alternate name for HS</td>
</tr>
<tr>
<td>lo</td>
<td>(C \text{ clear (unsigned &lt;)})</td>
</tr>
<tr>
<td>cc</td>
<td>Alternate name for LO</td>
</tr>
<tr>
<td>mi</td>
<td>(N \text{ set (result } &lt; 0))</td>
</tr>
<tr>
<td>pl</td>
<td>(N \text{ clear (result } \geq 0))</td>
</tr>
<tr>
<td>vs</td>
<td>(V \text{ set (overflow)})</td>
</tr>
<tr>
<td>vc</td>
<td>(V \text{ clear (no overflow)})</td>
</tr>
</tbody>
</table>
Example Data Processing Instruction

- **Syntax:**
  
  `<Operation>{<cond>}{s|S} Rd, Rn, Operand2`

- **Examples:**

  ```
  add     r0, r1, r2  @ r0=r1+r2
  add     r8, r5, r7  @ r8=r5+r7
  addne   r9, r1, r3  @ if(ne) then r9=r1+r3
  adds    r4, r6, r7  @ r4=r6+r7 and set SPSR flags
  addeqs  r6, r8, r2  @ if(eq) r6=r8+r2 and set SPSR flags
  adds    r0, r1, r2  @ r0=r1+r2 AND the condition flags
                @ are updated
  addcs   r3, r4, r5  @ If the carry bit is set,
                @ then r3=r4+r5,
                @ else ignore this instruction
  adcs    r2,r4,r6    @ Add r4, r6, and the carry bit,
                @ store the result in r2, and set
                @ the SPSR flags
  ```
Operand2

Most of the data processing instructions use Operand2 as the second operand.

Operand2 can be one of three things:

- A register ($r0$–$r15$)
- A shifted or rotated register
- A 32-bit immediate value that can be constructed by shifting, rotating, and/or complementing an 8-bit immediate value
Shifting Operand

There are five mnemonics that can be used to specify an arithmetic or logical shift, or a rotation:

- lsl  Logical Shift Left
- lsr  Logical Shift Right
- asr  Arithmetic Shift Right
- ror  Rotate Right

Note: An arithmetic shift left is equivalent to a logical shift left.

There is also a mnemonic for an extended rotation:

- rrx  Rotate Right with Extend

The RRX operation rotates one place to the right but the Carry flag is used along with Rx to provide a 33 bit quantity to be rotated.
Forms for Operand2

1. \#<immediate> (For 32-bit immediate values that can be constructed by shifting and complementing an 8 bit value.)
2. Rm (Any of the 16 registers r0-r15)
3. Rm, <shift_op> #<shift_imm>
4. Rm, <shift_op> Rs
5. Rm, RRX
More on Immediate values

The assembler must be able to construct the value using only 8 bits of data and a shift or rotate, and/or a complement.

Examples:

- #32  Ok because it is between 0 and 255.
- #1021 Illegal because the number cannot be created from an 8-bit value using shift/rotate/complement.
- #1024  Ok because it is 1 shifted left 10 bits.
- # -1  Ok because it is the one’s complement of 0
- #0xFFFFFFFF  Ok because it is the one’s complement of 1
- #0xEFFFFFFF  Ok because it is the one’s complement of 1 shifted left 31 bits

For immediate values that cannot be constructed by shifting and complementing an 8 bit value, we have to use

```
ldr Rn,=<immediate|symbol>
```

Then Rn can be used as Operand2.
Arithmetic Operations

- Operations:
  - ADD \( Rn + \text{operand2} \) @ Add
  - ADC \( Rn + \text{operand2} + \text{carry} \) @ Add with carry
  - SUB \( Rn - \text{operand2} \) @ Subtract
  - SBC \( Rn - \text{operand2} + \text{carry} - 1 \) @ Subtract with carry (borrow)
  - RSB \( \text{operand2} - Rn \) @ Reverse subtract
  - RSC \( \text{operand2} - Rn + \text{carry} - 1 \) @ Reverse subtract with carry

- Syntax:
  \(<\text{Operation}>\{<\text{cond}>\}\{S\} \text{ Rd, Rn, Operand2}\)

  The optional S specifies whether or not the instruction should affect the bits in the CPSR.

- Examples

  1. \textbf{add} \( r0, r1, r2 \) @ \( r0=r1+r2 \) and don’t set CPSR flags
  2. \textbf{subgt} \( r3, r3, #1 \) @ if (GT) then \( r3=r3-1 \) and don’t set @ CPSR flags\`
  3. \textbf{rsbles} \( r4, r5, #5 \) @ if (LE) then \( r4=5-r5 \) and set CPSR @ flags
Logical Operations

- Operations:
  - AND: Rn & operand2 @ AND
  - EOR: Rn ^ operand2 @ Exclusive OR
  - ORR: Rn | operand2 @ OR
  - ORN: !(Rn | operand2) @ NOR
  - BIC: Rn & !operand2 @ AND NOT (Bit Clear)

- Syntax:
  \(<\text{Operation}>\{<\text{cond}>\}\{\text{S}\} \quad \text{Rd, Rn, Operand2}\)

  The optional S specifies whether or not the instruction should affect the bits in the CPSR.

- Examples

```plaintext
1  and  r0, r1, r2  @ r0=r1&r2 and don’t set CPSR flags
2  biceq r3, r3, #1 @ if (EQ) then r3=r3&!0x00000001 and
   @ don’t set CPSR flags
3   eorles r4, r5, #5 @ if (LE) then r4=R5^0x00000005 and
   @ set CPSR flags
```
Comparison Operations

Comparison operations update the CPSR flags, but have no other effect.

- **Operations:**
  - CMP    Rn − operand2  @ Compare
  - CMN    Rn + operand2  @ Compare Negative
  - TST    Rn & operand2  @ Test
  - TEQ    Rn ^ operand2  @ Test equivalence

- **Syntax:**
  <Operation>{<cond>}  Rn, Operand2

- **Examples**

```plaintext
1  cmp    r0, r1  @ Compare r0 to r1 and set CPSR flags
2  tsteq  r2, #5  @ if (EQ) Compare r2 to 5
     @ and set CPSR flags
```
Data Movement Operations

Operations:
- MOV Rd, operand2  @ Copy operand2
- MVN Rd, !operand2  @ Copy 1’s complement of operand2

Syntax:
<Operation>{<cond>}{S} Rd, Operand2

Examples

1. mov r0, r1  @ r0 = r1
2. movs r2, #10  @ r2 = 10
3. mvneq r1, #1  @ if (EQ) then r1 = -1
4. movles r2, r2, ASR #1  @ if (LE) then r2 = r2 / 2
5.  @ and set CPSR flags
Multiply Operations with 32-bit Results

- **Operation:**
  \[ \text{MUL } Rd = Rm \times Rs \]  @ Multiply with 32-bit result

- **Syntax:**
  \[ \text{MUL\{<cond}\}\{S\} Rd, Rm, Rs \]

- **Operation:**
  \[ \text{MLA } Rd = Rm \times Rs + Rn \]  @ Multiply-accumulate with 32-bit result

- **Syntax:**
  \[ \text{MLA\{<cond}\}\{S\} Rd, Rm, Rs, Rn \]

- **Examples**

<table>
<thead>
<tr>
<th>Line</th>
<th>Command</th>
<th>Register 1</th>
<th>Register 2</th>
<th>Register 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>mul</code></td>
<td><code>r0</code></td>
<td><code>r1</code></td>
<td><code>r2</code></td>
</tr>
<tr>
<td>2</td>
<td><code>mla</code></td>
<td><code>r0</code></td>
<td><code>r1</code></td>
<td><code>r2</code></td>
</tr>
<tr>
<td>3</td>
<td><code>muleq</code></td>
<td><code>r0</code></td>
<td><code>r1</code></td>
<td><code>r2</code></td>
</tr>
<tr>
<td>4</td>
<td><code>mlas</code></td>
<td><code>r0</code></td>
<td><code>r1</code></td>
<td><code>r2</code></td>
</tr>
<tr>
<td>5</td>
<td><code>mulnes</code></td>
<td><code>r0</code></td>
<td><code>r1</code></td>
<td><code>r2</code></td>
</tr>
<tr>
<td>6</td>
<td><code>mlalts</code></td>
<td><code>r0</code></td>
<td><code>r1</code></td>
<td><code>r2</code></td>
</tr>
</tbody>
</table>
Multiply Operations with 64-bit Results

- **Operations:**
  - **SMULL** \( \text{RdHi:RdLo} = \text{Rm} \times \text{Rs} \)** @ Signed multiply with 64-bit result
  - **UMULL** \( \text{RdHi:RdLo} = \text{Rm} \times \text{Rs} \)** @ Unsigned multiply with 64-bit result
  - **SMLAL** \( \text{RdHi:RdLo} = \text{Rm} \times \text{Rs} + \text{RdHi:RdLo} \)** @ Signed multiply-accumulate with 64-bit result
  - **UMLAL** \( \text{RdHi:RdLo} = \text{Rm} \times \text{Rs} + \text{RdHi:RdLo} \)** @ Unsigned multiply-accumulate with 64-bit result

- **Syntax:**
  \(<\text{Operation}>\{<\text{cond}>\}\{S\} \quad \text{RdLo, RdHi, Rm, Rs}\)

- **Examples**
  1. `smull r0, r1, r3, r4`
  2. `smulls r0, r1, r3, r4`
  3. `umlaleq r0, r1, r3, r4`
The divide is available on most ARMv7 processors (Cortex M0 and M1 do not have hardware divide).

- **Operation:**
  
  SDIV  \( \text{Rd} = \text{Rn} \div \text{Rm} \)  \( @ \) Signed divide
  
  UDIV  \( \text{Rd} = \text{Rn} \div \text{Rm} \)  \( @ \) Unsigned divide

- **Syntax:**
  
  SDIV|UDIV\(<\text{cond}>\)  \( \text{Rd}, \text{Rm}, \text{Rn} \)

- **Example**

```
1  div  r0, r1, r2
```
Accessing the CPSR and SPSR

- **Operations:**
  - **MRS** @ Move from Status Register
  - **MSR** @ Move to Status Register

- **Syntax:**
  - `MRS{<cond>} Rd, CPSR{_<fields>}`
  - `MSR{<cond>} SPSR{_<fields>}, Rd`
  
  `<fields>` is any combination of:
  - `c` control field
  - `x` extension field
  - `s` status field
  - `f` flags field

- **Example Usage:**

```
1. mrs  R0, CPSR    @ Read the CPSR into r0
2. bic  R0,R0, #0xF0000000 @ Clear all of the flags
3. msr  CPSR_f, R0  @ Write the flags field to CPSR
```
Operating System Calls

- **Operation:**
  - SWI perform software interrupt
- **Syntax:**
  - SWI \(<\text{syscall\_number}>\)
- In Linux, the \(<\text{syscall\_number}>\) is ignored. The actual system call number is passed in register r7.
- **Example**

```
1     mov r0, #1   @ fd -> stdout
2     ldr r1, =msg @ buf -> msg
3     ldr r2, =len @ count -> len(msg)
4     mov r7, #4   @ write is syscall #4
5     swi #0      @ invoke syscall
```
The ARM processor has an alternate mode where it executes a 16-bit instruction set known as Thumb. This instruction allows us to change the mode and branch to Thumb code.

- **Operation:**
  
  BX Like BL, but also change to Thumb mode.

- **Syntax:**
  
  BX{<cond>} <target_address>

- **Example**

  ```
  bx my_thumb_code
  ```
Chicken and Egg Problem

- You have to learn the register set, instruction set and assembler directives before you can write assembly.
- You have to write assembly in order to learn the register set, instruction set, and assembler directives.

You may feel unsure of what you are doing and not understand everything at first.

That’s ok. Ask questions if you get stuck.
## Instruction Summary

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Add with carry</td>
<td>MRS</td>
<td>Move from CPSR or SPSR register</td>
</tr>
<tr>
<td>ADD</td>
<td>Add</td>
<td>MSR</td>
<td>Move to CPSR or SPSR register</td>
</tr>
<tr>
<td>AND</td>
<td>AND</td>
<td>MUL</td>
<td>Multiply</td>
</tr>
<tr>
<td>B</td>
<td>Branch</td>
<td>MVN</td>
<td>Move Negative</td>
</tr>
<tr>
<td>BIC</td>
<td>Bit Clear</td>
<td>ORR</td>
<td>OR</td>
</tr>
<tr>
<td>BL</td>
<td>Branch and Link</td>
<td>RSB</td>
<td>Reverse Subtract</td>
</tr>
<tr>
<td>BX</td>
<td>Branch and Exchange</td>
<td>RSC</td>
<td>Reverse Subtract with Carry</td>
</tr>
<tr>
<td>CDP</td>
<td>Coprocessor Data Processing *</td>
<td>SBC</td>
<td>Subtract with Carry</td>
</tr>
<tr>
<td>CMN</td>
<td>Compare Negative</td>
<td>SDIV</td>
<td>Signed integer division</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
<td>STC</td>
<td>Store Coprocessor register *</td>
</tr>
<tr>
<td>EOR</td>
<td>Exclusive OR</td>
<td>STM</td>
<td>Store Multiple</td>
</tr>
<tr>
<td>LDC</td>
<td>Load Coprocessor Register *</td>
<td>STR</td>
<td>Store Register</td>
</tr>
<tr>
<td>LDM</td>
<td>Load Multiple Registers</td>
<td>STREX</td>
<td>Store Register Exclusive</td>
</tr>
<tr>
<td>LDR</td>
<td>Load Register</td>
<td>SUB</td>
<td>Subtract</td>
</tr>
<tr>
<td>LDREX</td>
<td>Load Register Exclusive</td>
<td>SWI</td>
<td>Software Interrupt</td>
</tr>
<tr>
<td>MCR</td>
<td>Move to Coprocessor Register *</td>
<td>SWP</td>
<td>Swap register with memory</td>
</tr>
<tr>
<td>MLA</td>
<td>Multiply Accumulate</td>
<td>TEQ</td>
<td>Test bitwise equality</td>
</tr>
<tr>
<td>MOV</td>
<td>Move Register or Constant</td>
<td>TST</td>
<td>Test bits</td>
</tr>
<tr>
<td>MRC</td>
<td>Move from Coprocessor Register *</td>
<td>UDIV</td>
<td>unsigned integer division</td>
</tr>
</tbody>
</table>

* Not covered yet.