Chapter 4

ARM Instruction Sets

Jin-Fu Li
Department of Electrical Engineering
National Central University
Jungli, Taiwan
Outline

- Registers, Memory Access, and Data Transfer
- Arithmetic and Logic Instructions
- Branch Instructions
- Assembly Language
- I/ O Operations
- Subroutines
- Program Examples
Content Coverage

Main Memory System

Central Processing Unit (CPU)
- Operational Registers
- Arithmetic and Logic Unit
- Program Counter
- Control Unit

Cache memory

Instruction Sets

Data/Instruction

Address

Input/Output System
ARM Processor

- ARM processor was designed by Advanced RISC Machine (ARM) Limited Company
- ARM processors are major used for low-power and low cost applications
  - Mobile phones
  - Communication modems
  - Automotive engine management systems
  - Hand-held digital systems
- This chapter introduces the ARM instruction sets based on the ARM 7 processor
  - Different versions of ARM processors share the same basic machine instruction sets
Registers and Memory Access

- In the ARM architecture
  - Memory is byte addressable
  - 32-bit addresses
  - 32-bit processor registers
- Two operand lengths are used in moving data between the memory and the processor registers
  - Bytes (8 bits) and words (32 bits)
- Word addresses must be aligned, i.e., they must be multiple of 4
  - Both little-endian and big-endian memory addressing are supported
- When a byte is loaded from memory into a processor register or stored from a register into the memory
  - It always located in the low-order byte position of the register
Register Structure

R0

R1

R14

R15 (PC)

CPSR

N-Negative
Z-Zero
C-Carry
V-Overflow

Conditional code flags

Processor mode bits

Interrupt disable bits

Status register

Program counter

General Purpose registers
Register Structure

- The use of processor mode bits and interrupt disable bits will be described in conjunction with input/output operations and interrupts in Chapter 5.
- There are 15 additional general-purpose registers called the banked registers:
  - They are duplicates of some of the R0 to R14 registers.
  - They are used when the processor switches into Supervisor or Interrupt modes of operation.
- Saved copies of the Status register are also available in the Supervisor and Interrupt modes.
- The banked registers and Status register copies will also be discussed in Chapter 5.
ARM Instruction Format

- Each instruction is encoded into a 32-bit word.
- Access to memory is provided only by Load and Store instructions.
- The basic encoding format for the instructions, such as Load, Store, Move, Arithmetic, and Logic instructions, is shown below.

```
<table>
<thead>
<tr>
<th>31</th>
<th>28 27</th>
<th>20 19</th>
<th>16 15</th>
<th>12 11</th>
<th>4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition</td>
<td>OP code</td>
<td>Rn</td>
<td>Rd</td>
<td>Other info</td>
<td>Rm</td>
</tr>
</tbody>
</table>
```

- An instruction specifies a conditional execution code (Condition), the OP code, two or three registers (Rn, Rd, and Rm), and some other information.
Conditional Execution of Instructions

- A distinctive and somewhat unusual feature of ARM processors is that all instructions are conditionally executed.
  - Depending on a condition specified in the instruction.
- The instruction is executed only if the current state of the processor condition code flag satisfies the condition specified in bits \( b_{31} - b_{28} \) of the instruction.
  - Thus the instructions whose condition is not met by the processor condition code flag are not executed.
- One of the conditions is used to indicate that the instruction is always executed.
Memory Addressing Modes

- Pre-indexed mode
  - The effective address of the operand is the sum of the contents of the base register Rn and an offset value.

- Pre-indexed with writeback mode
  - The effective address of the operand is generated in the same way as in the Pre-indexed mode, and then the effective address is written back into Rn.

- Post-indexed mode
  - The effective address of the operand is the contents of Rn. The offset is then added to this address and the result is written back into Rn.
## ARM Indexed Addressing Modes

<table>
<thead>
<tr>
<th>Name</th>
<th>Assembler syntax</th>
<th>Addressing function</th>
</tr>
</thead>
<tbody>
<tr>
<td>With immediate offset:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pre-indexed</td>
<td>[Rn, #offset]</td>
<td>EA=[Rn]+offset</td>
</tr>
<tr>
<td>Pre-indexed with writeback</td>
<td>[Rn, #offset]!</td>
<td>EA=[Rn]+offset; Rn←[Rn]+offset</td>
</tr>
<tr>
<td>Post-indexed</td>
<td>[Rn], #offset</td>
<td>EA=[Rn]; Rn←[Rn]+offset</td>
</tr>
<tr>
<td>With offset in Rn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pre-indexed</td>
<td>[Rn, ±Rm, shift]</td>
<td>EA=[Rn]±[Rm] shifted</td>
</tr>
<tr>
<td>Pre-indexed with writeback</td>
<td>[Rn, ±Rm, shift]!</td>
<td>EA=[Rn]±[Rm] shifted; Rn←[Rn]±[Rm] shifted</td>
</tr>
<tr>
<td>Post-indexed</td>
<td>[Rn], ±Rm, shift</td>
<td>EA=[Rn]; Rn←[Rn]±[Rm] shifted</td>
</tr>
<tr>
<td>Relative (Pre-indexed with</td>
<td>Location</td>
<td>EA=Location=[PC]+offset</td>
</tr>
<tr>
<td>Immediate offset)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Shift**: direction #integer, where direction is LSL for left shift or LSR for right shift, and integer is a 5-bit unsigned number specifying the shift format.

±Rm=the offset magnitude in register Rm can be added to or subtracted from the contents of based register Rn.
Relative Addressing Mode

![Diagram](image)

The operand must be within the range of 4095 bytes forward or backward from the updated PC.
Pre-Indexed Addressing Mode

```
STR R3, [R5,R6]

1000

R5
```

```
200

R6
```

1200

Operand

1000=offset
Post-Indexed Addressing with Writeback

Load instruction:
LDR R1, [R2], R10, LSL, #2
Pre-Indexed Addressing with Writeback

27

After execution of Push instruction

2008
2012

Push instruction:
STR R0, [R5,# -4]!

Based register (stack pointer)

TOS (top-of-stack)

R0
R5

27
Load/Store Multiple Operands

- In ARM processors, there are two instructions for loading and storing multiple operands
  - They are called Block transfer instructions
- Any subset of the general purpose registers can be loaded or stored
  - Only word operands are allowed, and the OP codes used are LDM (Load Multiple) and STM (Store Multiple)
- The memory operands must be in successive word locations
- All of the forms of pre- and post-indexing with and without writeback are available
- They operate on a Base register Rn specified in the instruction and offset is always 4
  - LDMIA R10!, {R0,R1,R6,R7}
  - IA: “Increment After” corresponding to post-indexing
Arithmetic Instructions

- The basic expression for arithmetic instructions is
  - OPcode Rd, Rn, Rm
- For example, ADD R0, R2, R4
  - Performs the operation R0 ← [R2] + [R4]
- SUB R0, R6, R5
  - Performs the operation R0 ← [R6] - [R5]
- Immediate mode: ADD R0, R3, #17
  - Performs the operation R0 ← [R3] + 17
- The second operand can be shifted or rotated before being used in the operation
  - For example, ADD R0, R1, R5, LSL #4 operates as follows: the second operand stored in R5 is shifted left 4-bit positions (equivalent to [R5] x 16), and its is then added to the contents of R1; the sum is placed in R0
Logic Instructions

- The logic operations AND, OR, XOR, and Bit-Clear are implemented by instructions with the OP codes AND, ORR, EOR, and BIC.

- For example
  - `AND R0, R0, R1`: performs $R0 \leftarrow [R0] + [R1]$

- The Bit-Clear instruction (BIC) is closely related to the AND instruction.
  - It complements each bit in operand $Rm$ before ANDing them with the bits in register $Rn$.
  - For example, `BIC R0, R0, R1`. Let $R0=02FA62CA$, $R1=0000FFFF$. Then the instruction results in the pattern 02FA0000 being placed in $R0$

- The Move Negative instruction complements the bits of the source operand and places the result in $Rd$.
  - For example, `MVN R0, R3`
Branch Instructions

- Conditional branch instructions contain a signed 24-bit offset that is added to the updated contents of the Program Counter to generate the branch target address.
- The format for the branch instructions is shown as below:

```
  31  28  27  24  23  0
```

  - **Condition**
  - **OP code**
  - **offset**

- Offset is a signed 24-bit number. It is shifted left two-bit positions (all branch targets are aligned word addresses), signed extended to 32 bits, and added to the updated PC to generate the branch target address.
- The updated points to the instruction that is two words (8 bytes) forward from the branch instruction.
The BEQ instruction (Branch if Equal to 0) causes a branch if the Z flag is set to 1.

```
BEQ LOCATION
```

Updated [PC] = 1008
Offset = 92
LOCATION = 1100
Branch target instruction
Setting Condition Codes

- Some instructions, such as Compare, given by
  - CMP Rn, Rm which performs the operation \([Rn]-[Rm]\) have the sole purpose of setting the condition code flags based on the result of the subtraction operation

- The arithmetic and logic instructions affect the condition code flags only if explicitly specified to do so by a bit in the OP-code field. This is indicated by appending the suffix S to the OP-code
  - For example, the instruction ADDS R0, R1, R2 set the condition code flags
  - But ADD R0, R1, R2 does not
An Example of Adding Numbers

LDR    R1, N           Load count into R1
LDR    R2, POINTER     Load address NUM1 into R2
MOV    R0, #0          Clear accumulator R0

LOOP:
LDR    R3, [R2], #4    Load next number into R3
ADD    R0, R0, R3      Add number into R0
SUBS   R1, R1, #1      Decrement loop counter R1
BGT    LOOP            Branch back if not done
STR    R0, SUM         Store sum

Assume that the memory location N, POINTER, and SUM are within the range
Reachable by the offset relative to the PC
GT: signed greater than
BGT: Branch if Z=0 and N ≠0
Assembly Language

- The ARM assembly language has assembler directives to reserve storage space, assign numerical values to address labels and constant symbols, define where program and data blocks are to be placed in memory, and specify the end of the source program text.

- The **AREA** directive, which uses the argument **CODE** or **DATA**, indicates the beginning of a block of memory that contains either program instructions or data.

- The **ENTRY** directive specifies that program execution is to begin at the following LDR instruction.

- In the data area, which follows the code area, the **DCD** directives are used to label and initialize the data operands.
An Example of Assembly Language

Assembler directives

<table>
<thead>
<tr>
<th>AREA</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENTRY</td>
<td></td>
</tr>
</tbody>
</table>

Statements that generate machine instructions

<table>
<thead>
<tr>
<th>LDR</th>
<th>R1, N</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>R2, POINTER</td>
</tr>
<tr>
<td>MOV</td>
<td>R0, #0</td>
</tr>
<tr>
<td>LOOP</td>
<td>LDR R3, [R2], #4</td>
</tr>
<tr>
<td></td>
<td>ADD R0, R0, R3</td>
</tr>
<tr>
<td></td>
<td>SUBS R1, R1, #1</td>
</tr>
<tr>
<td></td>
<td>BGT LOOP</td>
</tr>
<tr>
<td></td>
<td>STR R0, SUM</td>
</tr>
</tbody>
</table>

Assembler directives

<table>
<thead>
<tr>
<th>AREA</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUM</td>
<td>DCD 0</td>
</tr>
<tr>
<td>N</td>
<td>DCD 5</td>
</tr>
<tr>
<td>POINTER</td>
<td>DCD NUM1</td>
</tr>
<tr>
<td>NUM1</td>
<td>DCD 3, -17, 27, -12, 322</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>
Assembly Language

- An EQU directive can be used to define symbolic names for constants
- For example, the statement
  - TEN EQU 10
- When a number of registers are used in a program, it is convenient to use symbolic names for them that relate to their usage
  - The RN directive is used for this purpose
  - For example, COUNTER RN 3 establishes the name COUNTER for register R3
- The register names R0 to R15, PC (for R15), and LR (for R14) are predefined by the assembler
  - R14 is used for a link register (LR)
Pseudo-Instructions

- An alternative way of loading the address into register R2 is also provided in the assembly language.
- The pseudo-instruction `ADR Rd, ADDRESS` holds the 32-bit value `ADDRESS` into `Rd`.
  - This instruction is not an actual machine instruction.
  - The assembler chooses appropriate real machine instructions to implement pseudo-instructions.
- For example,
  - The combination of the machine instruction `LDR R2, POINTER` and the data declaration directive `POINTER DCD NUM1` is one way to implement the pseudo-instruction `ADR R2, NUM1`.
A Branch and Link (BL) instruction is used to call a subroutine.

The return address is loaded into register R14, which acts as a link register.

When subroutines are nested, the contents of the link register must be saved on a stack by the subroutine.

- Register R13 is normally used as the pointer for this stack.
Example

Calling program

LDR    R1, N
LDR    R2, POINTER
BL     LISTADD
STR    R0, SUM

Subroutine

LISTADD  STMFD     R13!, {R3, R14}  Save R3 and return address in R14 on stack, using R13 as the stack pointer

MOV     R0, #0
LOOP    LDR     R3, [R2], #4
         ADD    R0, R0, R3
         SUBS   R1, R1, #1
         BGT    LOOP
         LDMFD  R13!, {R3, R15}  Restore R3 and load return address into PC (r15)
Byte-Sorting Program

for (j=n-1; j>0; j=j-1)
  for (k=j-1; k>=0; k=k-1)
    if (LIST[k] > LIST[j])
      TEMP = LIST[k];
      LIST[k] = LIST[j];
      LIST[j] = TEMP;
  
0 1 2 3 4 5 6 7 n-2 n-1
    k       j
Byte-Sorting Program

ADR  R4,LIST  Load list pointer register R4
LDR  R10,N    Initialize outer loop base
ADD  R2,R4,R10 Register R2 to LIST+n
ADD  R5,R4, #1 Load LIST+1 into R5

OUTER LDRB R0,[R2,# -1]! Load LIST(j) into R0
      MOV  R3,R2    Initialize inner loop base register R3 to LIST+n-1

INNER LDRB R1,[R3, # -1]! Load LIST(k) into R1
      CMP  R1,R0    Compare LIST(k) to LIST(j)
      STRGTB R1,[R2] interchanged LIST(k) and LIST(j)
      STRGTB R0,[R3]
      MOVGT R0,R1    Move (new) LIST(j) into R0
      CMP  R3,R4    If k>0, repeat
      BNE   INNER inner loop
      CMP  R2,R5    If j>1, repeat
      BNE   OUTER outer loop