ARM Instruction Set

Computer Organization and Assembly Languages
Yung-Yu Chuang

with slides by Peng-Sheng Chen
Introduction

- The ARM processor is easy to program at the assembly level. (It is a RISC)
- We will learn ARM assembly programming at the user level and run it on a simulator.
ARM programmer model

• The state of an ARM system is determined by the content of visible registers and memory.

• A user-mode program can see 15 32-bit general-purpose registers (R0-R14), program counter (PC) and CPSR.

• Instruction set defines the operations that can change the state.
**Memory system**

- Memory is a linear array of bytes addressed from 0 to \(2^{32}-1\)
- Word, half-word, byte
- Little-endian

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>00</td>
</tr>
<tr>
<td>0x00000001</td>
<td>10</td>
</tr>
<tr>
<td>0x00000002</td>
<td>20</td>
</tr>
<tr>
<td>0x00000003</td>
<td>30</td>
</tr>
<tr>
<td>0x00000004</td>
<td>FF</td>
</tr>
<tr>
<td>0x00000005</td>
<td>FF</td>
</tr>
<tr>
<td>0x00000006</td>
<td>FF</td>
</tr>
<tr>
<td>0xFFFFFFF0D</td>
<td>00</td>
</tr>
<tr>
<td>0xFFFFFFF0E</td>
<td>00</td>
</tr>
<tr>
<td>0xFFFFFFF0F</td>
<td>00</td>
</tr>
</tbody>
</table>
Byte ordering

- **Big Endian**
  - Least significant byte has highest address
  Word address 0x00000000
  Value: 00102030

- **Little Endian**
  - Least significant byte has lowest address
  Word address 0x00000000
  Value: 30201000
### ARM programmer model

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>R4</td>
<td>R5</td>
<td>R6</td>
<td>R7</td>
</tr>
<tr>
<td>R8</td>
<td>R9</td>
<td>R10</td>
<td>R11</td>
</tr>
<tr>
<td>R12</td>
<td>R13</td>
<td>R14</td>
<td>PC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>00</td>
</tr>
<tr>
<td>0x00000001</td>
<td>10</td>
</tr>
<tr>
<td>0x00000002</td>
<td>20</td>
</tr>
<tr>
<td>0x00000003</td>
<td>30</td>
</tr>
<tr>
<td>0x00000004</td>
<td>FF</td>
</tr>
<tr>
<td>0x00000005</td>
<td>FF</td>
</tr>
<tr>
<td>0x00000006</td>
<td>FF</td>
</tr>
<tr>
<td>0xFFFFFFFF</td>
<td>00</td>
</tr>
<tr>
<td>0xFFFFFFFD</td>
<td>00</td>
</tr>
<tr>
<td>0xFFFFFFFE</td>
<td>00</td>
</tr>
<tr>
<td>0xFFFFFFFF</td>
<td>00</td>
</tr>
</tbody>
</table>
ARM instructions are all 32-bit long (except for Thumb mode). There are $2^{32}$ possible machine instructions. Fortunately, they are structured.
Features of ARM instruction set

- Load-store architecture
- 3-address instructions
- Conditional execution of every instruction
- Possible to load/store multiple registers at once
- Possible to combine shift and ALU operations in a single instruction
Instruction set

- Data processing
- Data movement (memory access)
- Flow control
Data processing

• They are move, arithmetic, logical, comparison and multiply instructions.

• Most data processing instructions can process one of their operands using the barrel shifter.

• General rules:
  - All operands are 32-bit, coming from registers or literals.
  - The result, if any, is 32-bit and placed in a register (with the exception for long multiply which produces a 64-bit result)
  - 3-address format
Instruction set

MOV<cc><S> Rd, <operands>

MOVCS R0, R1 @ if carry is set
    @ then R0:=R1

MOVS R0, #0 @ R0:=0
    @ Z=1, N=0
    @ C, V unaffected
Conditional execution

- Almost all ARM instructions have a condition field which allows it to be executed conditionally.

\[
\text{movcs R0, R1}
\]

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Condition</th>
<th>Mnemonic</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>Carry Set</td>
<td>CC</td>
<td>Carry Clear</td>
</tr>
<tr>
<td>EQ</td>
<td>Equal (Zero Set)</td>
<td>NE</td>
<td>Not Equal (Zero Clear)</td>
</tr>
<tr>
<td>VS</td>
<td>Overflow Set</td>
<td>VC</td>
<td>Overflow Clear</td>
</tr>
<tr>
<td>GT</td>
<td>Greater Than</td>
<td>LT</td>
<td>Less Than</td>
</tr>
<tr>
<td>GE</td>
<td>Greater Than or Equal</td>
<td>LE</td>
<td>Less Than or Equal</td>
</tr>
<tr>
<td>PL</td>
<td>Plus (Positive)</td>
<td>MI</td>
<td>Minus (Negative)</td>
</tr>
<tr>
<td>HI</td>
<td>Higher Than</td>
<td>LO</td>
<td>Lower Than (aka CC)</td>
</tr>
<tr>
<td>HS</td>
<td>Higher or Same (aka CS)</td>
<td>LS</td>
<td>Lower or Same</td>
</tr>
</tbody>
</table>
Register movement

Syntax: `<instruction>{<cond>}{S}—Rd, N`  immediate, register, shift

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>Move a 32-bit value into a register</td>
<td>Rd = N</td>
</tr>
<tr>
<td>MVN</td>
<td>move the NOT of the 32-bit value into a register</td>
<td>Rd = ¬N</td>
</tr>
</tbody>
</table>

• MOV R0, R2 @ R0 = R2
• MVN R0, R2 @ R0 = ¬R2

move negated

PRE

\[ \begin{align*}
  r5 &= 5 \\
  r7 &= 8
\end{align*} \]

MOV r7, r5 ; let r7 = r5

POST

\[ \begin{align*}
  r5 &= 5 \\
  r7 &= 5
\end{align*} \]
Addressing modes

- Register operands
  ADD R0, R1, R2

- Immediate operands
  a literal; most can be represented by $(0..255) \times 2^n$ $0 < n < 12$
  ADD R3, R3, #1 @ R3 := R3 + 1
  AND R8, R7, #0xff @ R8 = R7[7:0]

This is assembler dependent syntax.

a hexadecimal literal

This is assembler dependent syntax.
Shifted register operands

- One operand to ALU is routed through the Barrel shifter. Thus, the operand can be modified before it is used. Useful for fast multiplication and dealing with lists, table and other complex data structure. (similar to the displacement addressing mode in CISC.)

Some instructions (e.g. **MUL**, **CLZ**, **QADD**) do not read barrel shifter.
# Shifted register operands

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Shift</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSL</td>
<td>logical shift left</td>
<td>$x \text{LSL } y$</td>
<td>$x \ll y$</td>
</tr>
<tr>
<td>LSR</td>
<td>logical shift right</td>
<td>$x \text{LSR } y$</td>
<td>$(\text{unsigned})x \gg y$</td>
</tr>
<tr>
<td>ASR</td>
<td>arithmetic right shift</td>
<td>$x \text{ASR } y$</td>
<td>$(\text{signed})x \gg y$</td>
</tr>
<tr>
<td>ROR</td>
<td>rotate right</td>
<td>$x \text{ROR } y$</td>
<td>$((\text{unsigned})x \gg y) \mid (x \ll (32 - y))$</td>
</tr>
<tr>
<td>RRX</td>
<td>rotate right extended</td>
<td>$x \text{RRX}$</td>
<td>$(c \text{ flag } \ll 31) \mid ((\text{unsigned})x \gg 1)$</td>
</tr>
</tbody>
</table>
Logical shift left

MOV  R0, R2, LSL #2 @ R0:=R2<<2
  @ R2 unchanged

Example: 0...0 0011 0000
Before  R2=0x00000030
After   R0=0x000000C0
        R2=0x00000030
Logical shift right

MOV R0, R2, LSR #2 @ R0:=R2>>2  
@ R2 unchanged

Example: 0...0 0011 0000
Before R2=0x00000030
After  R0=0x00000000C
      R2=0x00000030
Arithmetic shift right

\[
\text{MOV R0, R2, ASR } #2 \quad @ \quad R0 := R2 >> 2 \\
\quad @ \quad R2 \text{ unchanged}
\]

Example: \(1010 \ 0...0 \ 0011 \ 0000\)

Before \(R2 = 0xA0000030\)

After \(R0 = 0xE800000C\)

\(R2 = 0xA0000030\)
Rotate right

MOV R0, R2, ROR #2 @ R0:=R2 rotate
    @ R2 unchanged

Example: 0...0 0011 0001
Before R2=0x00000031
After  R0=0x4000000C
        R2=0x00000031
Rotate right extended

MOV R0, R2, RRX @ R0:=R2 rotate @ R2 unchanged

Example: 0...0 0011 0001
Before R2=0x00000031, C=1
After R0=0x80000018, C=1
R2=0x00000031
Shifted register operands

LSL #5  

ASR #5, positive operand

LSR #5  

ASR #5, negative operand
Shifted register operands

ROR #5

RRX
Shifted register operands

- It is possible to use a register to specify the number of bits to be shifted; only the bottom 8 bits of the register are significant.

@ array index calculation

ADD R0, R1, R2, LSL R3 @ R0 := R1 + R2 × 2^{R3}

@ fast multiply R2 = 35 × R0
ADD R0, R0, R0, LSL #2 @ R0′ = 5 × R0
RSB R2, R0, R0, LSL #3 @ R2 = 7 × R0′
Multiplication

\[
\begin{align*}
&\text{MOV} \quad R1, \ #35 \\
&\text{MUL} \quad R2, \ R0, \ R1 \\
&\quad \text{or} \\
&\text{ADD} \quad R0, \ R0, \ R0, \ LSL \ #2 \quad @ \ R0' = 5 \times R0 \\
&\text{RSB} \quad R2, \ R0, \ R0, \ LSL \ #3 \quad @ \ R2 = 7 \times R0'
\end{align*}
\]
# Shifted register operands

<table>
<thead>
<tr>
<th><em>N</em> shift operations</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td><code>#immediate</code></td>
</tr>
<tr>
<td>Register</td>
<td><code>Rm</code></td>
</tr>
<tr>
<td>Logical shift left by immediate</td>
<td><code>Rm, LSL #shift_imm</code></td>
</tr>
<tr>
<td>Logical shift left by register</td>
<td><code>Rm, LSL Rs</code></td>
</tr>
<tr>
<td>Logical shift right by immediate</td>
<td><code>Rm, LSR #shift_imm</code></td>
</tr>
<tr>
<td>Logical shift right with register</td>
<td><code>Rm, LSR Rs</code></td>
</tr>
<tr>
<td>Arithmetic shift right by immediate</td>
<td><code>Rm, ASR #shift_imm</code></td>
</tr>
<tr>
<td>Arithmetic shift right by register</td>
<td><code>Rm, ASR Rs</code></td>
</tr>
<tr>
<td>Rotate right by immediate</td>
<td><code>Rm, ROR #shift_imm</code></td>
</tr>
<tr>
<td>Rotate right by register</td>
<td><code>Rm, ROR Rs</code></td>
</tr>
<tr>
<td>Rotate right with extend</td>
<td><code>Rm, RRX</code></td>
</tr>
</tbody>
</table>
Encoding data processing instructions

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>cond</td>
</tr>
<tr>
<td>28-25</td>
<td># opcode</td>
</tr>
<tr>
<td>24</td>
<td>S</td>
</tr>
<tr>
<td>23-20</td>
<td>Rn</td>
</tr>
<tr>
<td>19-12</td>
<td>Rd</td>
</tr>
<tr>
<td>11-0</td>
<td>operand 2</td>
</tr>
</tbody>
</table>

- **cond**: Condition code
- **# opcode**: Immediate 32-bit immediate (8 bits)
- **S**: Source operand select
- **Rn**: First operand register
- **Rd**: Destination register
- **# rot**: 8-bit immediate
- **# shift**: Immediate shift length
- **Rm**: Second operand register
- **Rs**: Register shift length
## Arithmetic

- Add and subtraction

Syntax: `<instruction>{<cond>{S} Rd, Rn, N

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>add two 32-bit values and carry</td>
<td>$Rd = Rn + N + \text{carry}$</td>
</tr>
<tr>
<td>ADD</td>
<td>add two 32-bit values</td>
<td>$Rd = Rn + N$</td>
</tr>
<tr>
<td>RSB</td>
<td>reverse subtract of two 32-bit values</td>
<td>$Rd = N - Rn$</td>
</tr>
<tr>
<td>RSC</td>
<td>reverse subtract with carry of two 32-bit values</td>
<td>$Rd = N - Rn - !\text{(carry flag)}$</td>
</tr>
<tr>
<td>SBC</td>
<td>subtract with carry of two 32-bit values</td>
<td>$Rd = Rn - N - !\text{(carry flag)}$</td>
</tr>
<tr>
<td>SUB</td>
<td>subtract two 32-bit values</td>
<td>$Rd = Rn - N$</td>
</tr>
</tbody>
</table>
Arithmetic

- **ADD** R0, R1, R2 @ R0 = R1+R2
- **ADC** R0, R1, R2 @ R0 = R1+R2+C
- **SUB** R0, R1, R2 @ R0 = R1–R2
- **SBC** R0, R1, R2 @ R0 = R1–R2–!C
- **RSB** R0, R1, R2 @ R0 = R2–R1
- **RSC** R0, R1, R2 @ R0 = R2–R1–!C

3-5=3+(-5) → sum<=255 → C=0 → borrow
5-3=5+(-3) → sum > 255 → C=1 → no borrow
Arithmetic

**PRE**

r0 = 0x00000000
r1 = 0x00000002
r2 = 0x00000001

SUB r0, r1, r2

**POST**

r0 = 0x00000001

**PRE**

r0 = 0x00000000
r1 = 0x00000077

RSB r0, r1, #0 ; Rd = 0x0 - r1

**POST**

r0 = -r1 = 0xffffffff89
Arithmetic

**PRE**
\[
\text{cpsr} = \text{nzcvqiFt\_USER} \\
\text{r1} = 0x00000001
\]

\[
\text{SUBS} \ r1, r1, \#1
\]

**POST**
\[
\text{cpsr} = \text{nZCvqiFt\_USER} \\
\text{r1} = 0x00000000
\]

**PRE**
\[
\text{r0} = 0x00000000 \\
\text{r1} = 0x00000005
\]

\[
\text{ADD} \quad \text{r0, r1, r1, LSL \#1}
\]

**POST**
\[
\text{r0} = 0x0000000f \\
\text{r1} = 0x00000005
\]
Setting the condition codes

- Any data processing instruction can set the condition codes if the programmers wish it to

64-bit addition

\[
\begin{align*}
\text{ADD S} & \quad R2, R2, R0 \\
\text{ADC} & \quad R3, R3, R1 \\
\hline
+ & \\
R1 & R0 \\
R2 & R2 \\
\hline
R3 & R2
\end{align*}
\]
## Logical

Syntax: `<instruction>{<cond>>}{S} Rd, Rn, N`

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>logical bitwise AND of two 32-bit values</td>
<td>$Rd = Rn &amp; N$</td>
</tr>
<tr>
<td>ORR</td>
<td>logical bitwise OR of two 32-bit values</td>
<td>$Rd = Rn \mid N$</td>
</tr>
<tr>
<td>EOR</td>
<td>logical exclusive OR of two 32-bit values</td>
<td>$Rd = Rn ^ {N}$</td>
</tr>
<tr>
<td>BIC</td>
<td>logical bit clear (AND NOT)</td>
<td>$Rd = Rn &amp; \sim N$</td>
</tr>
</tbody>
</table>
Logical

- **AND**  \( R0, R1, R2 \)  @  \( R0 = R1 \text{ and } R2 \)
- **ORR**  \( R0, R1, R2 \)  @  \( R0 = R1 \text{ or } R2 \)
- **EOR**  \( R0, R1, R2 \)  @  \( R0 = R1 \text{ xor } R2 \)
- **BIC**  \( R0, R1, R2 \)  @  \( R0 = R1 \text{ and } (\neg R2) \)

**bit clear:**  \( R2 \)  is a mask identifying which bits of  \( R1 \)  will be cleared to zero

\[ R1=0x11111111 \quad R2=0x01100101 \]

BIC  \( R0, R1, R2 \)

\[ R0=0x10011010 \]
Logical

**PRE**

\[
\begin{align*}
  r0 & = 0x00000000 \\
  r1 & = 0x02040608 \\
  r2 & = 0x10305070
\end{align*}
\]

\text{ORR} \quad r0, r1, r2

**POST**

\[
\begin{align*}
  r0 & = 0x12345678
\end{align*}
\]

**PRE**

\[
\begin{align*}
  r1 & = 0b1111 \\
  r2 & = 0b0101
\end{align*}
\]

\text{BIC} \quad r0, r1, r2

**POST**

\[
\begin{align*}
  r0 & = 0b1010
\end{align*}
\]
Comparison

- These instructions do not generate a result, but set condition code bits (N, Z, C, V) in CPSR. Often, a branch operation follows to change the program flow.

Syntax: `<instruction>{<cond>} Rn, N`

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMN</td>
<td>compare negated</td>
<td>flags set as a result of ( Rn + N )</td>
</tr>
<tr>
<td>CMP</td>
<td>compare</td>
<td>flags set as a result of ( Rn - N )</td>
</tr>
<tr>
<td>TEQ</td>
<td>test for equality of two 32-bit values</td>
<td>flags set as a result of ( Rn \land N )</td>
</tr>
<tr>
<td>TST</td>
<td>test bits of a 32-bit value</td>
<td>flags set as a result of ( Rn \land N )</td>
</tr>
</tbody>
</table>
Comparison

compare
• CMP R1, R2 @ set cc on R1-R2

compare negated
• CMN R1, R2 @ set cc on R1+R2

bit test
• TST R1, R2 @ set cc on R1 and R2

test equal
• TEQ R1, R2 @ set cc on R1 xor R2
Comparison

**PRE**

\[ \text{cpsr} = \text{nzcvqiFt}_\text{USER} \]

\[ r0 = 4 \]

\[ r9 = 4 \]

\[ \text{CMP} \quad r0, r9 \]

**POST**

\[ \text{cpsr} = \text{nZcvqiFt}_\text{USER} \]
Multiplication

Syntax: MLA{<cond>}{S} Rd, Rm, Rs, Rn

MUL{<cond>}{S} Rd, Rm, Rs

<table>
<thead>
<tr>
<th>MLA</th>
<th>multiply and accumulate</th>
<th>Rd = (Rm*Rs) + Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL</td>
<td>multiply</td>
<td>Rd = Rm*Rs</td>
</tr>
</tbody>
</table>

Syntax: <instruction>{<cond>}{S} RdLo, RdHi, Rm, Rs

<table>
<thead>
<tr>
<th>SMLAL</th>
<th>signed multiply accumulate long</th>
<th>[RdHi, RdLo] = [RdHi, RdLo] + (Rm*Rs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMULL</td>
<td>signed multiply long</td>
<td>[RdHi, RdLo] = Rm*Rs</td>
</tr>
<tr>
<td>UMLAL</td>
<td>unsigned multiply accumulate long</td>
<td>[RdHi, RdLo] = [RdHi, RdLo] + (Rm*Rs)</td>
</tr>
<tr>
<td>UMULL</td>
<td>unsigned multiply long</td>
<td>[RdHi, RdLo] = Rm*Rs</td>
</tr>
</tbody>
</table>
Multiplication

- **MUL** R0, R1, R2 @ R0 = (R1xR2) [31:0]

- Features:
  - Second operand can’t be immediate
  - The result register must be different from the first operand
  - Cycles depends on core type
  - If S bit is set, C flag is meaningless

- See the reference manual (4.1.33)
Multiplication

- Multiply-accumulate (2D array indexing)
  
  \[ \text{MLA } R4, R3, R2, R1 \ @ R4 = R3 \times R2 + R1 \]

- Multiply with a constant can often be more efficiently implemented using shifted register operand
  
  \[ \text{MOV } R1, \#35 \]
  \[ \text{MUL } R2, R0, R1 \]

  or

  \[ \text{ADD } R0, R0, R0, \text{LSL } \#2 \ @ R0' = 5 \times R0 \]
  \[ \text{RSB } R2, R0, R0, \text{LSL } \#3 \ @ R2 = 7 \times R0' \]
Multiplication

**PRE**

\[
\begin{align*}
  r0 &= 0x00000000 \\
  r1 &= 0x00000002 \\
  r2 &= 0x00000002
\end{align*}
\]

\[
\text{MUL } \ r0, \ r1, \ r2 \quad ; \ r0 = r1 \times r2
\]

**POST**

\[
\begin{align*}
  r0 &= 0x00000004 \\
  r1 &= 0x00000002 \\
  r2 &= 0x00000002
\end{align*}
\]
Multiplication

**PRE**
\[
\begin{align*}
  r0 &= 0x00000000 \\
  r1 &= 0x00000000 \\
  r2 &= 0xf0000002 \\
  r3 &= 0x00000002 \\
\end{align*}
\]

\[
\text{UMULL } r0, r1, r2, r3 \ ; [r1\!,r0] = r2\!*r3
\]

**POST**
\[
\begin{align*}
  r0 &= 0xe0000004 \ ; = \text{RdLo} \\
  r1 &= 0x00000001 \ ; = \text{RdHi}
\end{align*}
\]
Flow control instructions

- Determine the instruction to be executed next

Syntax: B\{<cond>\} label
BL\{<cond>\} label
BX\{<cond>\} Rm
BLX\{<cond>\} label | Rm

<table>
<thead>
<tr>
<th>B</th>
<th>branch</th>
<th>(pc = label)</th>
<th>pc-relative offset within 32MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>BL</td>
<td>branch with link</td>
<td>(pc = label)</td>
<td>(lr = ) address of the next instruction after the BL</td>
</tr>
<tr>
<td>BX</td>
<td>branch exchange</td>
<td>(pc = Rm &amp; 0xffffffff, T = Rm &amp; 1)</td>
<td></td>
</tr>
<tr>
<td>BLX</td>
<td>branch exchange with link</td>
<td>(pc = label, T = 1)</td>
<td>(pc = Rm &amp; 0xffffffff, T = Rm &amp; 1) (lr = ) address of the next instruction after the BLX</td>
</tr>
</tbody>
</table>
Flow control instructions

- Branch instruction
  
  ```assembly
  B label
  ...
  label:
  ...
  ```

- Conditional branches
  
  ```assembly
  MOV R0, #0
  loop:
  ...
  ADD R0, R0, #1
  CMP R0, #10
  BNE loop
  ```
## Branch conditions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Condition flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>equal</td>
<td>Z</td>
</tr>
<tr>
<td>NE</td>
<td>not equal</td>
<td>z</td>
</tr>
<tr>
<td>CS HS</td>
<td>carry set/unsigned higher or same</td>
<td>C</td>
</tr>
<tr>
<td>CC LO</td>
<td>carry clear/unsigned lower</td>
<td>c</td>
</tr>
<tr>
<td>MI</td>
<td>minus/negative</td>
<td>N</td>
</tr>
<tr>
<td>PL</td>
<td>plus/positive or zero</td>
<td>n</td>
</tr>
<tr>
<td>VS</td>
<td>overflow</td>
<td>V</td>
</tr>
<tr>
<td>VC</td>
<td>no overflow</td>
<td>v</td>
</tr>
<tr>
<td>HI</td>
<td>unsigned higher</td>
<td>zC</td>
</tr>
<tr>
<td>LS</td>
<td>unsigned lower or same</td>
<td>Z or c</td>
</tr>
<tr>
<td>GE</td>
<td>signed greater than or equal</td>
<td>NV or nV</td>
</tr>
<tr>
<td>LT</td>
<td>signed less than</td>
<td>Nv or nV</td>
</tr>
<tr>
<td>GT</td>
<td>signed greater than</td>
<td>NzV or nzv</td>
</tr>
<tr>
<td>LE</td>
<td>signed less than or equal</td>
<td>Z or Nv or nV</td>
</tr>
<tr>
<td>AL</td>
<td>always (unconditional)</td>
<td>ignored</td>
</tr>
<tr>
<td>Branch</td>
<td>Interpretation</td>
<td>Normal uses</td>
</tr>
<tr>
<td>--------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>B BAL</td>
<td>Unconditional</td>
<td>Always take this branch</td>
</tr>
<tr>
<td></td>
<td>Always</td>
<td>Always take this branch</td>
</tr>
<tr>
<td>BEQ</td>
<td>Equal</td>
<td>Comparison equal or zero result</td>
</tr>
<tr>
<td>BNE</td>
<td>Not equal</td>
<td>Comparison not equal or non-zero result</td>
</tr>
<tr>
<td>BPL</td>
<td>Plus</td>
<td>Result positive or zero</td>
</tr>
<tr>
<td>BMI</td>
<td>Minus</td>
<td>Result minus or negative</td>
</tr>
<tr>
<td>BCC</td>
<td>Carry clear</td>
<td>Arithmetic operation did not give carry-out</td>
</tr>
<tr>
<td>BLO</td>
<td>Lower</td>
<td>Unsigned comparison gave lower</td>
</tr>
<tr>
<td>BCS</td>
<td>Carry set</td>
<td>Arithmetic operation gave carry-out</td>
</tr>
<tr>
<td></td>
<td>Higher or same</td>
<td>Unsigned comparison gave higher or same</td>
</tr>
<tr>
<td>BVC</td>
<td>Overflow clear</td>
<td>Signed integer operation; no overflow occurred</td>
</tr>
<tr>
<td>BVS</td>
<td>Overflow set</td>
<td>Signed integer operation; overflow occurred</td>
</tr>
<tr>
<td>BGT</td>
<td>Greater than</td>
<td>Signed integer comparison gave greater than</td>
</tr>
<tr>
<td>BGE</td>
<td>Greater or equal</td>
<td>Signed integer comparison gave greater or equal</td>
</tr>
<tr>
<td>BLT</td>
<td>Less than</td>
<td>Signed integer comparison gave less than</td>
</tr>
<tr>
<td>BLE</td>
<td>Less or equal</td>
<td>Signed integer comparison gave less than or equal</td>
</tr>
<tr>
<td>BHI</td>
<td>Higher</td>
<td>Unsigned comparison gave higher</td>
</tr>
<tr>
<td>BLS</td>
<td>Lower or same</td>
<td>Unsigned comparison gave lower or same</td>
</tr>
</tbody>
</table>
Branch and link

- **BL** instruction saves the return address to **R14** (lr)

```assembly
BL    sub    @ call sub
CMP   R1, #5  @ return to here
MOVEQ R1, #0
...
sub: ... @ sub entry point
...
MOV    PC, LR  @ return
```
Branch and link

```
BL    sub1     @ call sub1
...  
use stack to save/restore the return address and registers

sub1:    STMFD R13!, {R0-R2,R14}
BL    sub2
...

    LDMFD R13!, {R0-R2,PC}

sub2:    ...
...
    MOV    PC, LR
```
Conditional execution

CMP   R0, #5
BEQ   bypass @ if (R0!=5) {
ADD   R1, R1, R0 @ R1=R1+R0-R2
SUB   R1, R1, R2 @ }
bypass: ...

Rule of thumb: if the conditional sequence is three instructions or less, it is better to use conditional execution than a branch.
Conditional execution

if ((R0==R1) && (R2==R3)) R4++

CMP R0, R1
BNE skip
CMP R2, R3
BNE skip
ADD R4, R4, #1

skip:

CMP R0, R1
CMPEQ R2, R3
ADDEQ R4, R4, #1
Conditional execution

CMP R0, R1
BNE skip
CMP R2, R3
BNE skip
ADD R4, R4, #1

CMP R0, R1
CMPEQ R2, R3
ADDEQ R4, R4, #1
Data transfer instructions

• Move data between registers and memory
• Three basic forms
  - Single register load/store
  - Multiple register load/store
  - Single register swap: $\text{SWP}(B)$, atomic instruction for semaphore
## Single register load/store

### Syntax:

\[
\begin{align*}
\text{LDR} &\{<\text{cond}>\}\{B\} \ 	ext{Rd}, \text{addressing}^1 \\
\text{LDR} &\{<\text{cond}>\}\text{SB|H|SH} \ 	ext{Rd}, \text{addressing}^2 \\
\text{STR} &\{<\text{cond}>\}\text{H} \ 	ext{Rd}, \text{addressing}^2
\end{align*}
\]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LDR</strong></td>
<td>load word into a register</td>
<td>( Rd \gets \text{mem32}[\text{address}] )</td>
</tr>
<tr>
<td><strong>STR</strong></td>
<td>save byte or word from a register</td>
<td>( Rd \rightarrow \text{mem32}[\text{address}] )</td>
</tr>
<tr>
<td><strong>LDRB</strong></td>
<td>load byte into a register</td>
<td>( Rd \gets \text{mem8}[\text{address}] )</td>
</tr>
<tr>
<td><strong>STRB</strong></td>
<td>save byte from a register</td>
<td>( Rd \rightarrow \text{mem8}[\text{address}] )</td>
</tr>
</tbody>
</table>
## Single register load/store

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDRH</td>
<td>load halfword into a register</td>
<td>$Rd \leftarrow \text{mem16}[\text{address}]$</td>
</tr>
<tr>
<td>STRH</td>
<td>save halfword into a register</td>
<td>$Rd \rightarrow \text{mem16}[\text{address}]$</td>
</tr>
<tr>
<td>LDRSB</td>
<td>load signed byte into a register</td>
<td>$Rd \leftarrow \text{SignExtend}(\text{mem8}[\text{address}])$</td>
</tr>
<tr>
<td>LDRSH</td>
<td>load signed halfword into a register</td>
<td>$Rd \leftarrow \text{SignExtend}(\text{mem16}[\text{address}])$</td>
</tr>
</tbody>
</table>

No **STRSB/STRSH** since **STRB/STRH** stores both signed/unsigned ones.
Single register load/store

- The data items can be a 8-bit byte, 16-bit half-word or 32-bit word. Addresses must be boundary aligned. (e.g. 4’s multiple for LDR/STR)

\[
\begin{align*}
LDR & \quad R0, \; [R1] \quad @ \quad R0 := \text{mem}_{32}[R1] \\
STR & \quad R0, \; [R1] \quad @ \quad \text{mem}_{32}[R1] := R0
\end{align*}
\]

LDR, LDRH, LDRB for 32, 16, 8 bits
STR, STRH, STRB for 32, 16, 8 bits
Addressing modes

• Memory is addressed by a register and an offset.
  \[ \text{LDR } R0, [R1] @ \text{mem}[R1] \]

• Three ways to specify offsets:
  - Immediate
    \[ \text{LDR } R0, [R1, #4] @ \text{mem}[R1+4] \]
  - Register
    \[ \text{LDR } R0, [R1, R2] @ \text{mem}[R1+R2] \]
  - Scaled register
    \[ \text{LDR } R0, [R1, R2, LSL #2] @ \text{mem}[R1+4*R2] \]
Addressing modes

- Pre-index addressing (LDR R0, [R1, #4])
  without a writeback
- Auto-indexing addressing (LDR R0, [R1, #4]!)
  Pre-index with writeback calculation before accessing with a writeback
- Post-index addressing (LDR R0, [R1], #4)
  calculation after accessing with a writeback

<table>
<thead>
<tr>
<th>Index method</th>
<th>Data</th>
<th>Base address register</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preindex with writeback</td>
<td>mem[base + offset]</td>
<td>base + offset</td>
<td>LDR r0,[r1,#4]!</td>
</tr>
<tr>
<td>Preindex</td>
<td>mem[base + offset]</td>
<td>not updated</td>
<td>LDR r0,[r1,#4]</td>
</tr>
<tr>
<td>Postindex</td>
<td>mem[base]</td>
<td>base + offset</td>
<td>LDR r0,[r1],#4</td>
</tr>
</tbody>
</table>
Pre-index addressing

LDR R0, [R1, #4]  @ R0=mem[R1+4]
@ R1 unchanged

LDR R0, [R1, □]
Auto-indexing addressing

LDR  R0, [R1, #4]!  @ R0=mem[R1+4]
    @ R1=R1+4

No extra time; Fast;

LDR  R0, [R1, #]!
Post-index addressing

LDR R0, R1, #4  @ R0=mem[R1]
@ R1=R1+4

LDR R0, [R1], 

R1

+ 

R0
Comparisons

• Pre-indexed addressing

  \texttt{LDR \ R0, [R1, R2] \ @ R0=mem[R1+R2]}

  \ @ \textit{R1 unchanged}

• Auto-indexing addressing

  \texttt{LDR \ R0, [R1, R2]! \ @ R0=mem[R1+R2]}

  \ @ \textit{R1=R1+R2}

• Post-indexed addressing

  \texttt{LDR \ R0, [R1], R2 \ @ R0=mem[R1]}

  \ @ \textit{R1=R1+R2}
Example

**PRE**

\[
\begin{align*}
    r0 &= 0x00000000 \\
    r1 &= 0x00090000 \\
    \text{mem32}[0x00009000] &= 0x01010101 \\
    \text{mem32}[0x00009004] &= 0x02020202
\end{align*}
\]

LDR  \quad r0, [r1, #4]!

Preindexing with writeback:

**POST(1)**

\[
\begin{align*}
    r0 &= 0x02020202 \\
    r1 &= 0x00009004
\end{align*}
\]
Example

**PRE**

\[
\begin{align*}
  r0 &= 0x00000000 \\
  r1 &= 0x00090000 \\
  \text{mem32}[0x00009000] &= 0x01010101 \\
  \text{mem32}[0x00009004] &= 0x02020202
\end{align*}
\]

\[\text{LDR} \quad r0, [r1, \#4]\]

Preindexing:

**POST\(2\)**

\[
\begin{align*}
  r0 &= 0x02020202 \\
  r1 &= 0x00009000
\end{align*}
\]
Example

<table>
<thead>
<tr>
<th>PRE</th>
<th>r0 = 0x00000000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>r1 = 0x00090000</td>
</tr>
<tr>
<td></td>
<td>mem32[0x000009000] = 0x01010101</td>
</tr>
<tr>
<td></td>
<td>mem32[0x000009004] = 0x02020202</td>
</tr>
</tbody>
</table>

LDR r0, [r1], #4

Postindexing:

<table>
<thead>
<tr>
<th>POST(3)</th>
<th>r0 = 0x01010101</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>r1 = 0x000009004</td>
</tr>
</tbody>
</table>
## Summary of addressing modes

Syntax: `<LDR|STR>{<cond>}{{B} Rd, addressing^1 
   LDR{<cond>}SB|H|SH Rd, addressing^2 
   STR{<cond>}H Rd, addressing^2`

<table>
<thead>
<tr>
<th>Addressing^1 mode and index method</th>
<th>Addressing^1 syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preindex with immediate offset</td>
<td><code>[Rn, #+/-offset_12]</code></td>
</tr>
<tr>
<td>Preindex with register offset</td>
<td><code>[Rn, +/-Rm]</code></td>
</tr>
<tr>
<td>Preindex with scaled register offset</td>
<td><code>[Rn, +/-Rm, shift #shift_imm]</code></td>
</tr>
<tr>
<td>Preindex writeback with immediate offset</td>
<td><code>[Rn, #+/-offset_12]!</code></td>
</tr>
<tr>
<td>Preindex writeback with register offset</td>
<td><code>[Rn, +/-Rm]!</code></td>
</tr>
<tr>
<td>Preindex writeback with scaled register offset</td>
<td><code>[Rn, +/-Rm, shift #shift_imm]!</code></td>
</tr>
<tr>
<td>Immediate postindexed</td>
<td><code>[Rn], #+/-offset_12</code></td>
</tr>
<tr>
<td>Register postindex</td>
<td><code>[Rn], +/-Rm</code></td>
</tr>
<tr>
<td>Scaled register postindex</td>
<td><code>[Rn], +/-Rm, shift #shift_imm</code></td>
</tr>
</tbody>
</table>
## Summary of addressing modes

<table>
<thead>
<tr>
<th>Instruction</th>
<th>$r0 =$</th>
<th>$r1 += =$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Preindex with writeback</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>LDR r0,[r1,#0x4]!</strong></td>
<td>mem32[r1+0x4]</td>
<td>0x4</td>
</tr>
<tr>
<td><strong>LDR r0,[r1,r2]!</strong></td>
<td>mem32[r1+r2]</td>
<td>r2</td>
</tr>
<tr>
<td><strong>LDR r0,[r1,r2,LSR#0x4]!</strong></td>
<td>mem32[r1+(r2 LSR 0x4)]</td>
<td>(r2 LSR 0x4)</td>
</tr>
<tr>
<td><strong>LDR r0,[r1,#0x4]</strong></td>
<td>mem32[r1+0x4]</td>
<td>not updated</td>
</tr>
<tr>
<td><strong>LDR r0,[r1,r2]</strong></td>
<td>mem32[r1+r2]</td>
<td>not updated</td>
</tr>
<tr>
<td><strong>LDR r0,[r1,-r2,LSR #0x4]</strong></td>
<td>mem32[r1-(r2 LSR 0x4)]</td>
<td>not updated</td>
</tr>
<tr>
<td><strong>Postindex</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>LDR r0,[r1],#0x4</strong></td>
<td>mem32[r1]</td>
<td>0x4</td>
</tr>
<tr>
<td><strong>LDR r0,[r1],r2</strong></td>
<td>mem32[r1]</td>
<td>r2</td>
</tr>
<tr>
<td><strong>LDR r0,[r1],r2,LSR #0x4</strong></td>
<td>mem32[r1]</td>
<td>(r2 LSR 0x4)</td>
</tr>
</tbody>
</table>
## Summary of addressing modes

Syntax: `<LDR|STR>{<cond>}{B} Rd, addressing¹

LDR{<cond>}SB|H|SH Rd, addressing²
STR{<cond>}H Rd, addressing²

<table>
<thead>
<tr>
<th>Addressing² mode and index method</th>
<th>Addressing² syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preindex immediate offset</td>
<td><code>[Rn, #+/−-offset_8]</code></td>
</tr>
<tr>
<td>Preindex register offset</td>
<td><code>[Rn, +/−Rm]</code></td>
</tr>
<tr>
<td>Preindex writeback immediate offset</td>
<td><code>[Rn, #+/−-offset_8]!</code></td>
</tr>
<tr>
<td>Preindex writeback register offset</td>
<td><code>[Rn, +/−Rm]!</code></td>
</tr>
<tr>
<td>Immediate postindexed</td>
<td><code>[Rn], #+/−-offset_8</code></td>
</tr>
<tr>
<td>Register postindexed</td>
<td><code>[Rn], +/−Rm</code></td>
</tr>
</tbody>
</table>
## Summary of addressing modes

<table>
<thead>
<tr>
<th>Preindex with writeback</th>
<th>Instruction</th>
<th>Result</th>
<th>$r1 + = $</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preindex</td>
<td>STRH r0, [r1, #0x4]!</td>
<td>mem16[r1+0x4]=r0</td>
<td>0x4</td>
</tr>
<tr>
<td>Postindex</td>
<td>STRH r0, [r1, r2]!</td>
<td>mem16[r1+r2]=r0</td>
<td>r2</td>
</tr>
<tr>
<td></td>
<td>STRH r0, [r1, #0x4]</td>
<td>mem16[r1+0x4]=r0</td>
<td>not updated</td>
</tr>
<tr>
<td></td>
<td>STRH r0, [r1, r2]</td>
<td>mem16[r1+r2]=r0</td>
<td>not updated</td>
</tr>
<tr>
<td></td>
<td>STRH r0, [r1], #0x4</td>
<td>mem16[r1]=r0</td>
<td>0x4</td>
</tr>
<tr>
<td></td>
<td>STRH r0, [r1], r2</td>
<td>mem16[r1]=r0</td>
<td>r2</td>
</tr>
</tbody>
</table>
Load an address into a register

- Note that all addressing modes are register-offseted. Can we issue `LDR R0, Table`? The pseudo instruction `ADR` loads a register with an address

  table: .word 10

  ...

  ADR R0, table

- Assembler transfer pseudo instruction into a sequence of appropriate instructions

  sub r0, pc, #12
Application

loop:

ADR R1, table

LDR R0, [R1]
ADD R1, R1, #4
@ operations on R0
...

ADR R1, table

loop:

LDR R0, [R1], #4
@ operations on R0
...

...
Multiple register load/store

- Transfer a block of data more efficiently.
- Used for procedure entry and exit for saving and restoring workspace registers and the return address.
- For ARM7, $2 + N_t$ cycles ($N$: #words, $t$: time for a word for sequential access). Increase interrupt latency since it can’t be interrupted.

Registers are arranged in increasing order; see manual.

```assembly
LDMIA R1, {R0, R2, R5} @ R0 = mem[R1]
  @ R2 = mem[r1+4]
  @ R5 = mem[r1+8]
```
Multiple load/store register

<table>
<thead>
<tr>
<th>LDM</th>
<th>load multiple registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>STM</td>
<td>store multiple registers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>suffix</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA</td>
<td>increase after</td>
</tr>
<tr>
<td>IB</td>
<td>increase before</td>
</tr>
<tr>
<td>DA</td>
<td>decrease after</td>
</tr>
<tr>
<td>DB</td>
<td>decrease before</td>
</tr>
</tbody>
</table>
# Addressing modes

Syntax: `<LDM|STM>{<cond>}{addressing mode} Rn{!},<registers>{}`

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Description</th>
<th>Start address</th>
<th>End address</th>
<th>$Rn!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA</td>
<td>increment after</td>
<td>$Rn$</td>
<td>$Rn + 4*N - 4$</td>
<td>$Rn + 4*N$</td>
</tr>
<tr>
<td>IB</td>
<td>increment before</td>
<td>$Rn + 4$</td>
<td>$Rn + 4*N$</td>
<td>$Rn + 4*N$</td>
</tr>
<tr>
<td>DA</td>
<td>decrement after</td>
<td>$Rn - 4*N + 4$</td>
<td>$Rn$</td>
<td>$Rn - 4*N$</td>
</tr>
<tr>
<td>DB</td>
<td>decrement before</td>
<td>$Rn - 4*N$</td>
<td>$Rn - 4$</td>
<td>$Rn - 4*N$</td>
</tr>
</tbody>
</table>
Multiple load/store register

LDM<mode> Rn, {<registers>}
IA: addr:=Rn
IB: addr:=Rn+4
DA: addr:=Rn-#{<registers>}*4+4
DB: addr:=Rn-#{<registers>}*4
For each Ri in <registers>
   IB: addr:=addr+4
   DB: addr:=addr-4
   Ri:=M[addr]
IA: addr:=addr+4
DA: addr:=addr-4
<!>: Rn:=addr
Multiple load/store register

LDM<mode> Rn, {<registers>}
IA: addr:=Rn
IB: addr:=Rn+4
DA: addr:=Rn-#<registers>*4+4
DB: addr:=Rn-#<registers>*4
For each Ri in <registers>
  IB: addr:=addr+4
  DB: addr:=addr-4
  Ri:=M[addr]
IA: addr:=addr+4
DA: addr:=addr-4
<!>: Rn:=addr
Multiple load/store register

LDM<mode> Rn, {<registers>}
IA: addr:=Rn
IB: addr:=Rn+4
DA: addr:=Rn-#<registers>*4+4
DB: addr:=Rn-#<registers>*4
For each Ri in <registers>
   IB: addr:=addr+4
   DB: addr:=addr-4
   Ri:=M[addr]
IA: addr:=addr+4
DA: addr:=addr-4
<!>: Rn:=addr
Multiple load/store register

LDM<mode> Rn, {<registers>}
IA: addr:=Rn
IB: addr:=Rn+4
DA: addr:=Rn-#<registers>*4+4
DB: addr:=Rn-#<registers>*4
For each Ri in <registers>
  IB: addr:=addr+4
  DB: addr:=addr-4
  Ri:=M[addr]
IA: addr:=addr+4
DA: addr:=addr-4
<!>: Rn:=addr
Multiple load/store register

**LDMIA R0, {R1,R2,R3}**

or

**LDMIA R0, {R1-R3}**

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x010</td>
<td>10</td>
</tr>
<tr>
<td>0x014</td>
<td>20</td>
</tr>
<tr>
<td>0x018</td>
<td>30</td>
</tr>
<tr>
<td>0x01C</td>
<td>40</td>
</tr>
<tr>
<td>0x020</td>
<td>50</td>
</tr>
<tr>
<td>0x024</td>
<td>60</td>
</tr>
</tbody>
</table>

R1: 10
R2: 20
R3: 30
R0: 0x10
Multiple load/store register

LDMIA R0!, {R1,R2,R3}

R1: 10
R2: 20
R3: 30
R0: 0x01C

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x010</td>
<td>10</td>
</tr>
<tr>
<td>0x014</td>
<td>20</td>
</tr>
<tr>
<td>0x018</td>
<td>30</td>
</tr>
<tr>
<td>0x01C</td>
<td>40</td>
</tr>
<tr>
<td>0x020</td>
<td>50</td>
</tr>
<tr>
<td>0x024</td>
<td>60</td>
</tr>
</tbody>
</table>
Multiple load/store register

**LDMIB** R0!, {R1, R2, R3}

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x010</td>
<td>10</td>
</tr>
<tr>
<td>0x014</td>
<td>20</td>
</tr>
<tr>
<td>0x018</td>
<td>30</td>
</tr>
<tr>
<td>0x01C</td>
<td>40</td>
</tr>
<tr>
<td>0x020</td>
<td>50</td>
</tr>
<tr>
<td>0x024</td>
<td>60</td>
</tr>
</tbody>
</table>

R1: 20
R2: 30
R3: 40
R0: 0x01C
Multiple load/store register

`LDMDA R0!, {R1,R2,R3}`

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x010</td>
<td>10</td>
</tr>
<tr>
<td>0x014</td>
<td>20</td>
</tr>
<tr>
<td>0x018</td>
<td>30</td>
</tr>
<tr>
<td>0x01C</td>
<td>40</td>
</tr>
<tr>
<td>0x020</td>
<td>50</td>
</tr>
<tr>
<td>0x024</td>
<td>60</td>
</tr>
</tbody>
</table>

R1: 40  
R2: 50  
R3: 60  
R0: 0x018
Multiple load/store register

LDMDB R0!, {R1,R2,R3}

R1: 30
R2: 40
R3: 50
R0: 0x018

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x010</td>
<td>10</td>
</tr>
<tr>
<td>0x014</td>
<td>20</td>
</tr>
<tr>
<td>0x018</td>
<td>30</td>
</tr>
<tr>
<td>0x01C</td>
<td>40</td>
</tr>
<tr>
<td>0x020</td>
<td>50</td>
</tr>
<tr>
<td>0x024</td>
<td>60</td>
</tr>
</tbody>
</table>
Example

PRE

\[
\begin{align*}
\text{mem32[0x80018]} &= 0x03 \\
\text{mem32[0x80014]} &= 0x02 \\
\text{mem32[0x80010]} &= 0x01 \\
\text{r0} &= 0x00080010 \\
\text{r1} &= 0x00000000 \\
\text{r2} &= 0x00000000 \\
\text{r3} &= 0x00000000
\end{align*}
\]

\text{LDMIA r0!, \{r1-r3\}}

\[\ldots\\\ldots\\\ldots\ldots\\\]
### Example

**LDMIA** $r0!, \{r1-r3\}$

<table>
<thead>
<tr>
<th>Address pointer</th>
<th>Memory address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r0 = 0x8001c$</td>
<td>$0x80020$</td>
<td>$0x00000005$</td>
</tr>
<tr>
<td></td>
<td>$0x8001c$</td>
<td>$0x00000004$</td>
</tr>
<tr>
<td></td>
<td>$0x80018$</td>
<td>$0x00000003$</td>
</tr>
<tr>
<td></td>
<td>$0x80014$</td>
<td>$0x00000002$</td>
</tr>
<tr>
<td></td>
<td>$0x80010$</td>
<td>$0x00000001$</td>
</tr>
<tr>
<td></td>
<td>$0x8000c$</td>
<td>$0x00000000$</td>
</tr>
</tbody>
</table>

$r3 = 0x00000003$  
$r2 = 0x00000002$  
$r1 = 0x00000001$
Example

<table>
<thead>
<tr>
<th>Memory address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80020</td>
<td>0x00000005</td>
</tr>
<tr>
<td>0x8001c</td>
<td>0x00000004</td>
</tr>
<tr>
<td>0x80018</td>
<td>0x00000003</td>
</tr>
<tr>
<td>0x80014</td>
<td>0x00000002</td>
</tr>
<tr>
<td>0x80010</td>
<td>0x00000001</td>
</tr>
<tr>
<td>0x8000c</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

$LDMIB r0!, \{r1-r3\}$

<table>
<thead>
<tr>
<th>Memory address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80020</td>
<td>0x00000005</td>
</tr>
<tr>
<td>0x8001c</td>
<td>0x00000004</td>
</tr>
<tr>
<td>0x80018</td>
<td>0x00000003</td>
</tr>
<tr>
<td>0x80014</td>
<td>0x00000002</td>
</tr>
<tr>
<td>0x80010</td>
<td>0x00000001</td>
</tr>
<tr>
<td>0x8000c</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

$r0 = 0x80010 \rightarrow r3 = 0x00000000$  
$r2 = 0x00000000$  
$r1 = 0x00000000$  

$r0 = 0x8001c \rightarrow r3 = 0x00000004$  
$r2 = 0x00000003$  
$r1 = 0x00000002$
Application

- Copy a block of memory
  - R9: address of the source
  - R10: address of the destination
  - R11: end address of the source

loop: LDMIA R9!, {R0-R7}
STMIA R10!, {R0-R7}
CMP R9, R11
BNE loop
Application

- Stack (full: pointing to the last used; ascending: grow towards increasing memory addresses)

<table>
<thead>
<tr>
<th>mode</th>
<th>POP</th>
<th>=LDM</th>
<th>PUSH</th>
<th>=STM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full ascending (FA)</td>
<td>LDMFA</td>
<td>LDMDA</td>
<td>STMFA</td>
<td>STMIB</td>
</tr>
<tr>
<td>Full descending (FD)</td>
<td>LDMFD</td>
<td>LDMIA</td>
<td>STMFD</td>
<td>STMDB</td>
</tr>
<tr>
<td>Empty ascending (EA)</td>
<td>LDMEA</td>
<td>LDMDB</td>
<td>STMEA</td>
<td>STMIA</td>
</tr>
<tr>
<td>Empty descending (ED)</td>
<td>LDMED</td>
<td>LDMIB</td>
<td>STMED</td>
<td>STMDA</td>
</tr>
</tbody>
</table>

STMFD R13!, {R2-R9} @ used for ATPCS

... @ modify R2-R9

LDMFD R13!, {R2-R9}
Example

<table>
<thead>
<tr>
<th>PRE</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x80018</td>
<td>0x00000001</td>
</tr>
<tr>
<td>sp</td>
<td>0x80014</td>
<td>0x00000002</td>
</tr>
<tr>
<td></td>
<td>0x80010</td>
<td>Empty</td>
</tr>
<tr>
<td></td>
<td>0x8000c</td>
<td>Empty</td>
</tr>
</tbody>
</table>

STMFD  sp!, {r1,r4}

<table>
<thead>
<tr>
<th>POST</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x80018</td>
<td>0x00000001</td>
</tr>
<tr>
<td></td>
<td>0x80014</td>
<td>0x00000002</td>
</tr>
<tr>
<td></td>
<td>0x80010</td>
<td>0x00000003</td>
</tr>
<tr>
<td></td>
<td>0x8000c</td>
<td>0x00000002</td>
</tr>
</tbody>
</table>
Swap instruction

- Swap between memory and register. Atomic operation preventing any other instruction from reading/writing to that location until it completes

Syntax: SWP{B}{<cond>} Rd,Rm,[Rn]

|      | swap a word between memory and a register | \( tmp = \text{mem32}[Rn] \)  \
|      |                                           | \( \text{mem32}[Rn] = Rm \)  \
|      |                                           | \( Rd = tmp \)  \
| SWP  | swap a word between memory and a register | \( tmp = \text{mem8}[Rn] \)  \
|      |                                           | \( \text{mem8}[Rn] = Rm \)  \
|      |                                           | \( Rd = tmp \)  \
| SWPB | swap a byte between memory and a register |
Example

PRE

\[ \text{mem32}[0x9000] = 0x12345678 \]
\[ r0 = 0x00000000 \]
\[ r1 = 0x11112222 \]
\[ r2 = 0x00009000 \]

\[ \text{SWP} \quad r0, \ r1, \ [r2] \]

POST

\[ \text{mem32}[0x9000] = 0x11112222 \]
\[ r0 = 0x12345678 \]
\[ r1 = 0x11112222 \]
\[ r2 = 0x00009000 \]
**Application**

**spin**

```assembly
MOV    r1, =semaphore
MOV    r2, #1
SWP    r3, r2, [r1]; hold the bus until complete
CMP    r3, #1
BEQ    spin

While (1) {
    if (s=0) {
        s=1;
    }
}
// use the
// resource
```

**Process A**

```assembly
While (1) {
    if (s==0) {
        s=1;
    }
}
// use the
// resource
```

**OS**

```
S=0/1
```

**Process B**

```assembly
While (1) {
    if (s==0) {
        s=1;
    }
}
// use the
// resource
```
Software interrupt

- A software interrupt instruction causes a software interrupt exception, which provides a mechanism for applications to call OS routines.

Syntax: SWI{<cond>} SWI_number

| SWI | software interrupt | $lr_{svc} =$ address of instruction following the SWI | $spsr_{svc} =$ cpsr | $pc =$ vectors + 0x8 | $cpsr$ mode = SVC | $cpsr$ I = 1 (mask IRQ interrupts) |
Example

**PRE**

cpsr = nzcVqift_USER
pc = 0x00008000
lr = 0x003ffffff; lr = r14
r0 = 0x12

0x00008000  SWI  0x123456

**POST**
cpsr = nzcVqIfSVC
spsr = nzcVqift_USER
pc = 0x00000008
lr = 0x00008004
r0 = 0x12
Load constants

- No ARM instruction loads a 32-bit constant into a register because ARM instructions are 32-bit long. There is a pseudo code for this.

Syntax: LDR Rd, =constant
ADR Rd, label

<table>
<thead>
<tr>
<th>LDR</th>
<th>load constant pseudoinstruction</th>
<th>Rd = 32-bit constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADR</td>
<td>load address pseudoinstruction</td>
<td>Rd = 32-bit relative address</td>
</tr>
</tbody>
</table>
Immediate numbers

$v = n \text{ ror } 2r$

encoding for data processing instructions
Load constants

- Assemblers implement this usually with two options depending on the number you try to load.

<table>
<thead>
<tr>
<th>Pseudoinstruction</th>
<th>Actual instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR r0, =0xff</td>
<td>MOV r0, #0xff</td>
</tr>
<tr>
<td>LDR r0, =0x55555555</td>
<td>LDR r0, [pc, #offset_12]</td>
</tr>
</tbody>
</table>

Loading the constant `0xff00ffff`

```
LDR r0, [pc, #constant_number-8-{PC}]
: constant_number
  DCD 0xff00ffff
MVN r0, #0x00ff0000
```
Load constants

- Assume that you want to load 511 into R0
  - Construct in multiple instructions
    ```assembly
    mov r0, #256
    add r0, #255
    ```
  - Load from memory; declare L511 .word 511
    ```assembly
    ldr r0, L511
    ldr r0, [pc, #0]
    ```
- Guideline: if you can construct it in two instructions, do it; otherwise, load it.
- The assembler decides for you
  ```assembly
  ldr r0, =255
  mov r0, 255
  ldr r0, =511
  ldr r0, [pc, #4]
  ```
PC-relative modes

Adding (U=1) or subtracting (U=0) an unsigned immediate or register offset

Pre-indexed addressing mode (P=1)
Post-indexed addressing mode (P=0)

Impossible to use direct addressing encoding for data transfer instructions
PC-relative addressing

main:

    MOV  R0, #0
    ADR  R1, a      @ add  r1, pc, #4
    STR  R0, [R1]

PC  SWI  #11

a:  .word   100
    .end

fetch decode exec
fetch decode exec
fetch decode exec
# Instruction set

<table>
<thead>
<tr>
<th>Operation Mnemonic</th>
<th>Meaning</th>
<th>Operation Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Add with Carry</td>
<td>MVN</td>
<td>Logical NOT</td>
</tr>
<tr>
<td>ADD</td>
<td>Add</td>
<td>ORR</td>
<td>Logical OR</td>
</tr>
<tr>
<td>AND</td>
<td>Logical AND</td>
<td>RSB</td>
<td>Reverse Subtract</td>
</tr>
<tr>
<td>BAL</td>
<td>Unconditional Branch</td>
<td>RSC</td>
<td>Reverse Subtract with Carry</td>
</tr>
<tr>
<td>B\langle cc\rangle</td>
<td>Branch on Condition</td>
<td>SBC</td>
<td>Subtract with Carry</td>
</tr>
<tr>
<td>BIC</td>
<td>Bit Clear</td>
<td>SMLAL</td>
<td>Mult Accum Signed Long</td>
</tr>
<tr>
<td>BLAL</td>
<td>Unconditional Branch and Link</td>
<td>SMULL</td>
<td>Multiply Signed Long</td>
</tr>
<tr>
<td>BL\langle cc\rangle</td>
<td>Conditional Branch and Link</td>
<td>STM</td>
<td>Store Multiple</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
<td>STR</td>
<td>Store Register (Word)</td>
</tr>
<tr>
<td>EOR</td>
<td>Exclusive OR</td>
<td>STRB</td>
<td>Store Register (Byte)</td>
</tr>
<tr>
<td>LDM</td>
<td>Load Multiple</td>
<td>SUB</td>
<td>Subtract</td>
</tr>
<tr>
<td>LDR</td>
<td>Load Register (Word)</td>
<td>SWI</td>
<td>Software Interrupt</td>
</tr>
<tr>
<td>LDRB</td>
<td>Load Register (Byte)</td>
<td>SWP</td>
<td>Swap Word Value</td>
</tr>
<tr>
<td>MLA</td>
<td>Multiply Accumulate</td>
<td>SWPB</td>
<td>Swap Byte Value</td>
</tr>
<tr>
<td>MOV</td>
<td>Move</td>
<td>TEQ</td>
<td>Test Equivalence</td>
</tr>
<tr>
<td>MRS</td>
<td>Load SPSR or CPSR</td>
<td>TST</td>
<td>Test</td>
</tr>
<tr>
<td>MSR</td>
<td>Store to SPSR or CPSR</td>
<td>UMLAL</td>
<td>Mult Accum Unsigned Long</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply</td>
<td>UMULL</td>
<td>Multiply Unsigned Long</td>
</tr>
</tbody>
</table>