In this lecture, we will consider some aspects of ARM instruction set architecture (ISA) in some details.

We shall consider the format of some instruction codes and their relationship with the assembly instruction.

We start with a simple Branch and Branch with Link instruction:

Note that the top 4 bits [31:28] are always used to specify the conditions under which the instruction is executed.

The L-bit (bit 24) is set if it is a branch with link instruction.

BL is jump to subroutine instruction - r14 <- return address

24-bit signed offset specifies destination of branch in 2's complement form. It is shifted left by 2 bits to form a word offset.

The range of branch is +/- 32 Mbytes.

Data Processing Instructions

Uses 3-address format: first operand - always register; second operand - register/shifted register/immediate value; result - always a register

For second register operand, it can be logical/arithmetic/rotate. This is specified in "shift-type"

How much to shift by is either a constant #shift or a register

For immediate value second operand, only rotation is possible

S-bit controls condition code update

N flag - set if result is negative (N equals bit 31 of result)

Z flag - set if result is zero

C flag - set if there is a carry-out from ALU during arithmetic operations, or set by shifter

V flag - set in an arithmetic operation if there is an overflow from bit 30 to bit 31. It is significant only when operands are viewed as 2's complement signed values

ARM condition codes fields

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Mnemonic extension</th>
<th>Interpretation</th>
<th>Status flag state for execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>EQ</td>
<td>Equal / equals zero</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>Nequals V</td>
</tr>
<tr>
<td>1011</td>
<td>LT</td>
<td>Signed less than</td>
<td>Nis not equal to V</td>
</tr>
<tr>
<td>1100</td>
<td>GT</td>
<td>Signed greater than</td>
<td>Z set or N equals V</td>
</tr>
<tr>
<td>1101</td>
<td>LE</td>
<td>Signed less than or equal</td>
<td>Z set or N is not equal to V</td>
</tr>
<tr>
<td>1110</td>
<td>AL</td>
<td>Always</td>
<td>any</td>
</tr>
<tr>
<td>1111</td>
<td>NV</td>
<td>Never (do not use!)</td>
<td>none</td>
</tr>
</tbody>
</table>

Data Processing Instruction Binary encoding
## ARM data processing instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Mnemonic</th>
<th>Meaning</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>AND</td>
<td>Logical bit-wise AND</td>
<td>Rd := Rs AND Op2</td>
</tr>
<tr>
<td>0001</td>
<td>EOR</td>
<td>Logical bit-wise exclusive OR</td>
<td>Rd := Rs EOR Op2</td>
</tr>
<tr>
<td>0010</td>
<td>SUB</td>
<td>Subtract</td>
<td>Rd := Rs - Op2</td>
</tr>
<tr>
<td>0011</td>
<td>RSB</td>
<td>Reverse subtract</td>
<td>Rd := Op2 - Rs</td>
</tr>
<tr>
<td>0100</td>
<td>ADD</td>
<td>Add</td>
<td>Rd := Rs + Op2</td>
</tr>
<tr>
<td>0101</td>
<td>ADC</td>
<td>Add with carry</td>
<td>Rd := Rs + Op2 + C</td>
</tr>
<tr>
<td>0110</td>
<td>SBC</td>
<td>Subtract with carry</td>
<td>Rd := Rs - Op2 + C - 1</td>
</tr>
<tr>
<td>0111</td>
<td>RSC</td>
<td>Reverse subtract with carry</td>
<td>Rd := Op2 - Rs + C - 1</td>
</tr>
<tr>
<td>1000</td>
<td>TST</td>
<td>Test</td>
<td>Scn on Rs AND Op2</td>
</tr>
<tr>
<td>1001</td>
<td>TEQ</td>
<td>Test equivalence</td>
<td>Scn on Rs EOR Op2</td>
</tr>
<tr>
<td>1010</td>
<td>CMP</td>
<td>Compare</td>
<td>Scn on Rs - Op2</td>
</tr>
<tr>
<td>1011</td>
<td>CMN</td>
<td>Compare negated</td>
<td>Scn on Rs + Op2</td>
</tr>
<tr>
<td>1100</td>
<td>ORR</td>
<td>Logical bit-wise OR</td>
<td>Rd := Rs OR Op2</td>
</tr>
<tr>
<td>1101</td>
<td>MOV</td>
<td>Move</td>
<td>Rd := Op2</td>
</tr>
<tr>
<td>1110</td>
<td>BIC</td>
<td>Bit clear</td>
<td>Rd := Rs AND NOT Op2</td>
</tr>
<tr>
<td>1111</td>
<td>MVN</td>
<td>Move negated</td>
<td>Rd := NOT Op2</td>
</tr>
</tbody>
</table>

### Example of data processing instructions

- ADD r5, r1, r3  
  E081 5003
- ADDNE r0, r0, r0, LSL #2  
  1090 0100

## Data Transfer Instructions (LDR/STR)

- **P = 1** means pre-indexed, i.e. modify the address BEFORE use
- **P = 0** means post-indexed, i.e. modify the address AFTER use
- **B = 1** selects unsigned byte transfer (default is word transfer)
- `<offset>` may be `+/-` 12-bit immediate value (i.e. constant)
- `<offset>` may also be `+/-` register
- write-back (or "!") = 1 if the base register is updated
- All the shift parameters are the same as before
Multiple register transfer instructions

STMIA r13!, {r0-r2, r14}  

Multiply Instructions

- ARM has a number of multiply instructions
  - Produce product of two 32-bit binary numbers held in registers.
  - Results of 32-bit*32-bit is 64 bits. Some ARM processors store the entire 64-bit results in registers. Other ARM processors only stores the LOWER 32-bit products.
  - Multiply-Accumulate instruction also add product to accumulator value to form a running total.

Multiply Instructions

- Multiply Instructions
  - Produce product of two 32-bit binary numbers held in registers.
  - Results of 32-bit*32-bit is 64 bits. Some ARM processors store the entire 64-bit results in registers. Other ARM processors only stores the LOWER 32-bit products.
  - Multiply-Accumulate instruction also add product to accumulator value to form a running total.

Multiply Instructions

- Multiply Instructions
  - Produce product of two 32-bit binary numbers held in registers.
  - Results of 32-bit*32-bit is 64 bits. Some ARM processors store the entire 64-bit results in registers. Other ARM processors only stores the LOWER 32-bit products.
  - Multiply-Accumulate instruction also add product to accumulator value to form a running total.

Multiply Instructions

- Multiply Instructions
  - Produce product of two 32-bit binary numbers held in registers.
  - Results of 32-bit*32-bit is 64 bits. Some ARM processors store the entire 64-bit results in registers. Other ARM processors only stores the LOWER 32-bit products.
  - Multiply-Accumulate instruction also add product to accumulator value to form a running total.

Multiply Instructions

- Multiply Instructions
  - Produce product of two 32-bit binary numbers held in registers.
  - Results of 32-bit*32-bit is 64 bits. Some ARM processors store the entire 64-bit results in registers. Other ARM processors only stores the LOWER 32-bit products.
  - Multiply-Accumulate instruction also add product to accumulator value to form a running total.

Example of using ARM Multiplier

- This calculates a scalar product of two vectors, 20 long.
- r8 and r9 points two the two vectors
- r11 is the loop counter
- r10 stores results

Example of using ARM Multiplier

- This calculates a scalar product of two vectors, 20 long.
- r8 and r9 points two the two vectors
- r11 is the loop counter
- r10 stores results

Example of using ARM Multiplier

- This calculates a scalar product of two vectors, 20 long.
- r8 and r9 points two the two vectors
- r11 is the loop counter
- r10 stores results

Example of using ARM Multiplier

- This calculates a scalar product of two vectors, 20 long.
- r8 and r9 points two the two vectors
- r11 is the loop counter
- r10 stores results

Multiply with constants

- When multiplying by a constant value, it is possible to replace the general multiply with a fixed sequence of adds and subtracts that have the same effect.
- For instance, multiply by 5 could be achieved using a single instruction:

Multiply with constants

- When multiplying by a constant value, it is possible to replace the general multiply with a fixed sequence of adds and subtracts that have the same effect.
- For instance, multiply by 5 could be achieved using a single instruction:

Multiply with constants

- When multiplying by a constant value, it is possible to replace the general multiply with a fixed sequence of adds and subtracts that have the same effect.
- For instance, multiply by 5 could be achieved using a single instruction:

Multiply with constants

- When multiplying by a constant value, it is possible to replace the general multiply with a fixed sequence of adds and subtracts that have the same effect.
- For instance, multiply by 5 could be achieved using a single instruction:

Multiply with constants

- When multiplying by a constant value, it is possible to replace the general multiply with a fixed sequence of adds and subtracts that have the same effect.
- For instance, multiply by 5 could be achieved using a single instruction: