ARM Shift Operations

A novel feature of ARM is that all data-processing instructions can include an optional "shift", whereas most other architectures have separate shift instructions. This is actually very useful as we will see later on. The key to shifting is that 8-bit field between Rd and Rm.

<table>
<thead>
<tr>
<th>R type:</th>
<th>1110</th>
<th>000</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Shift</th>
<th>L</th>
<th>A</th>
<th>0</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift Type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 - logical left</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 - logical right</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 - arithmetic right</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 - rotate right</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Shift Amount
0-31 bits
Left Shifts

Left shifts effectively multiply the contents of a register by $2^s$ where $s$ is the shift amount.

MOV R0, R0, LSL 7

R0 before: \[
\begin{array}{cccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1
\end{array}
\]

= 7

R0 after: \[
\begin{array}{cccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0
\end{array}
\]

= 7 * 2^7 = 896

Shifts can also be applied to the second operand of any data processing instruction

ADD R1, R1, R0, LSL 7
**Right Shifts**

Right Shifts behave like dividing the contents of a register by $2^s$ where $s$ is the shift amount, if you assume the contents of the register are unsigned.

**MOV R0, R0, LSR 2**

R0 before: 0000 0000 0000 0000 0000 0100 0000 0000 = 1024

R0 after: 0000 0000 0000 0000 0000 0001 0000 0000 = 1024 / $2^2 = 256
Arithmetic Right Shifts

Arithmetic right shifts behave like dividing the contents of a register by \(2^s\) where \(s\) is the shift amount, if you assume the contents of the register are signed.

\[ \text{MOV R0, R0, ASR 2} \]

| R0 before: | \begin{array}{cccccccccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} | = -1024 |
| R0 after:  | \begin{array}{cccccccccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} | = -1024 / 2^2 = -256 |

This is Java’s “>>>” operator, LSR is “>>” and LSL is “<<”
**ROTATE RIGHT Shifts**

Rotating shifts have no arithmetic analogy. However, they don’t lose bits like both logical and arithmetic shifts. We saw rotate right shift used for the I-type "immediate" value earlier.

**MOV R0, R0, ROR 2**

R0 before: 0000 0000 0000 0000 0000 0000 0000 0111

R0 after: 1100 0000 0000 0000 0000 0000 0000 0001 = -1,073,741,823

Why no rotate left shift?

- Ran out of encodings?
- Almost anything Rotate lefts can do ROR can do as well!

Java doesn't have an operator for this one.
Addressing Modes and Branches

- More on Immediates
- Reading and Writing Memory
- Registers holding addresses
- Pointers
- Changing the PC
  - Loops
  - Labels
  - Calling Functions
Why Built-in Constant Operands?

(Immediates)

- Alternatives? Why not? Do we have a choice?
  - put constants in memory (was common in older instruction sets)

- SMALL constants are used frequently (50% of operands)
  - In a C compiler (gcc) 52% of ALU operations involve a constant
  - In a circuit simulator (spice) 69% involve constants
  - e.g., \( B = B + 1; \) \( C = W \& 0xff; \) \( A = B - 1; \)

- ISA Design Principle:
  - Make the common case easy
  - Make the common case fast
**Moves and ORs**

We can load any 32-bit constant using a series of instructions, one-byte at a time.

```
MOV R0,#85            ; 0x55 in hex
ORR R0,R0,#21760      ; 0x5500 in hex
ORR R0,R0,#5570560    ; 0x550000 in hex
ORR R0,R0,#1426063360 ; 0x55000000 in hex
```

But there are often better, faster, ways to load constants, and the assembler can figure out how for you, even if it needs to generate multiple instructions.

```
MOV R0,=1431655765    ; 0x55555555 in hex
```

Note that an equal sign is used here rather than a hashtag.
ARM is a "Load/Store architecture". That means that only a special class of instructions are used to reference data in memory. As a rule, data is loaded into registers first, then processed, and the results are written back using stores. Load and Store instructions have their own format:

### D type:

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **L** is a "1" for a Load and "0" for a Store
- Why does a "1" imply an immediate operand for ALU types, but "0" for Loads and Stores?

### X type:

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 4 | 3 | 1 | 1 | 1 | 1 | 1 | 4 | 4 |   | 5 | 2 | 1 | 4 |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

- **U** = 0: subtract offset from base, otherwise add them
- **L** = 1 for a Load and 0 for a Store
- The same “shift” options that we saw for the data processing instructions
Load and Store Options

ARM's load and store instructions are versatile. They provide a wide range of addressing modes. Only a subset is shown here.

**LDR** \( \text{Rd, [Rn, #imm12]} \)

Rd ← Memory[Rn + imm12]

Rd is loaded with the contents of memory at the address found by adding the contents of the base register to the supplied constant.

**STR** \( \text{R0, [R1, #-4]} \)

Memory[R1 - 4] ← R0

Offsets can be either added or subtracted, as indicated by a negative sign.

**LDR** \( \text{R2, [R3]} \)

If no offset is specified it is assumed to be zero.

**STR** \( \text{R4, [R5, R6]} \)

The contents of a second register can be used as an offset rather than a constant (using the X-type format).

**LDR** \( \text{R4, [R5, -R6]} \)

Register offsets can be either added or subtracted, like constants.

**STR** \( \text{R4, [R5, R4, LSL 2]} \)

Register offsets can also be optionally shifted, which is great for indexing arrays!
Changing the PC

The Program Counter is special register (R15) that tracks the address of the next instruction to be fetched. There are special instructions for changing the PC.

<table>
<thead>
<tr>
<th>B type:</th>
<th>Cond</th>
<th>101</th>
<th>L</th>
<th>Imm24</th>
</tr>
</thead>
</table>

- 0000 - EQ: equals
- 0001 - NE: not equals
- 0010 - CS: carry set
- 0011 - CC: carry clear
- 0100 - MI: negative
- 0101 - PL: positive or zero
- 0110 - VS: overflow
- 0111 - VC: no overflow
- 1000 - HI: higher (unsigned)
- 1001 - LS: lower or same (unsigned)
- 1010 - GE: greater or equal (signed)
- 1011 - LT: less than (signed)
- 1100 - GT: greater than (signed)
- 1101 - LE: less than or equal (signed)
- 1110 - "": always

Branches are often executed conditionally based on the PSR state set by some previous instruction like CMP or TST.

The "L" bit causes PC+4 to be saved in LP (R14).
Branch using Registers

The standard Branch instruction has a limited range, the 24-bit signed 2’s complement immediate value is multiplied by 4 and added to the PC+8, giving a range of +/- 32 Mbytes. Larger branches make use of addresses previously loaded into a register using the BX instruction.

<table>
<thead>
<tr>
<th>R type:</th>
<th>Cond</th>
<th>000</th>
<th>1 0010 1111 1111 1111 0001</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000 - EQ</td>
<td>- equals</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0001 - NE</td>
<td>- not equals</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0010 - CS</td>
<td>- carry set</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0011 - CC</td>
<td>- carry clear</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0100 - MI</td>
<td>- negative</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0101 - PL</td>
<td>- positive or zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0110 - VS</td>
<td>- overflow</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0111 - VC</td>
<td>- no overflow</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1000 - HI</td>
<td>- higher (unsigned)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1001 - LS</td>
<td>- lower or same (unsigned)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1010 - GE</td>
<td>- greater or equal (signed)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1011 - LT</td>
<td>- less than (signed)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1100 - GT</td>
<td>- greater than (signed)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1101 - LE</td>
<td>- less than or equal (signed)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1110 - “”</td>
<td>- always</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If the condition is true, the PC is loaded with the contents of Rn.

BTW, BX is encoded as a TEQ instruction with its S field set to "0"
Branch Examples

BNE  else

If some previous CMP instruction had a non-zero result (i.e., making the 'Z' bit 0 in the PSR), then this instruction will cause the PC to be loaded with the address having the label "else".

BEQL  func

If some previous CMP instruction set the 'Z' bit in the PSR, then this instruction will cause the PC to be loaded with the address having the label "func", and the address of the following instruction will be saved in R14.

BX   LR

Loads the PC with the contents of R14.

loop:   B loop

An infinite loop.
BTW: This instruction is encoded as: 0xEAFFFE

```
1110  101  0  1111 1111 1111 1111 1111 1111 1110
```

"always"  B-type format  Don't save the return address  -2, which implies PC + 8 -2*4 = PC
A SIMPLE PROGRAM

; Assembly code for
; sum = 0;
; for (i = 0; i <= 10; i++)
; sum = sum + i;
    MOV    R1,#0        ; R1 is i
    MOV    R0,#0        ; R0 is sum
loop:  ADD    R0,R0,R1  ; sum = sum + i
    ADD    R1,R1,#1    ; i++
    CMP    R1,#10      ; i <= 10
    BLE    loop
halt:  B      halt
Load and Stores in Action

An example of how loads and stores are used to access arrays.

Java/C:

```java
int x[10];
int sum = 0;
for (int i = 0; i < 10; i++)
    sum += x[i];
```

Assembly:

```assembly
.align 4
x: .space 40
sum: .word 0
```

```
MOV R0,=x    ; base of x
MOV R1,=sum
LDR R2,[R1]
MOV R3,#0    ; R3 is i
for: LDR R4,[R0,R3 LSL 2]
ADD R2,R2,R4
ADD R3,R3,#1
CMP R3,#10
BLT for
STR R2,[R1]
```
Next time

We’ll write more Assembly programs

Still some loose ends

- Multiplication? Division? Floating point?