**Status Flags**

Now it is time to discuss what *status flags* are available. These five status flags are kept in a special register called the Program Status Register (PSR). The PSR also contains other important bits that control the processor.

- **N** - set if the result of an operation is negative (Most Significant Bit (MSB) is a 1)
- **Z** - set if the result of an operation is "0"
- **C** - set if the result of an operation has a carry out of its MSB
- **V** - set if a sum of two positive operands gives a negative result, or if the sum of two negative operands gives a positive result
- **Q** - a sticky version of overflow created by instructions that generate multiple results (more on this later on).
**Comparison Instructions**

These instructions modify the status flags, but leave the contents of the registers unchanged. They are used to test register contents, and they **must** have their "S" bit set to "1". They also don't modify their Rd, and by **convention**, Rd is set to "0000".

<table>
<thead>
<tr>
<th>R type:</th>
<th>1110</th>
<th>000</th>
<th>Opcode 1</th>
<th>Rn</th>
<th>0000</th>
<th>00000000</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>I type:</td>
<td>1110</td>
<td>001</td>
<td>Opcode 1</td>
<td>Rn</td>
<td>0000</td>
<td>Rotate</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

- **CMP R0, R1**  
  PSR flags set for the result  
  \( R_0 - R_1 \)

- **CMN R2, R3**  
  PSR flags set for the result  
  \( R_2 + R_3 \)

- **TST R4, #8**  
  PSR flags set for the result  
  \( R_4 \& 8 \)

- **TEQ R5, #1024**  
  PSR flags set for the result  
  \( R_5 \oplus 1024 \)
Register Transfer

These instructions are used to transfer the contents of one register to another, or simply to initialize the contents of a register. They make use of only one operand, and, by convention, have their Rn field set to "0000".

\[
\begin{align*}
\text{MOV} & \quad \text{R0, R3} \\
& \quad R0 \leftarrow R3 \\
\text{MOV} & \quad \text{R1, #4096} \\
& \quad R1 \leftarrow 4096 \\
\text{MVN} & \quad \text{R2, R4} \\
& \quad R2 \leftarrow -R4 \\
\text{MVNS} & \quad \text{R3, #1} \\
& \quad R3 \leftarrow -1, \text{ and set PSR (Z = 0, N = 1, V = 0, C = 0)}
\end{align*}
\]
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>4</th>
<th>3</th>
<th>4</th>
<th>1</th>
<th>4</th>
<th>4</th>
<th>5</th>
<th>2</th>
<th>1</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110</td>
<td>000</td>
<td>Opcode</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>Shift</td>
<td>L</td>
<td>A</td>
<td>0</td>
<td>Rm</td>
</tr>
</tbody>
</table>

**Shift Type**
- 00 - logical left
- 01 - logical right
- 10 - arithmetic right
- 11 - rotate right

**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: | 0000 0000 0000 0000 0000 0000 0000 0111 | = 7 |
| R0 after: | 0000 0000 0000 0000 0000 0011 1000 0000 | = 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.

MOV R0, R0, ASR 2

R0 before: 1111 1111 1111 1111 1111 1111 1100 0000 0000 = -1024
R0 after: 1111 1111 1111 1111 1111 1111 1111 0000 0000 = -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 01 11
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)*

<table>
<thead>
<tr>
<th>I type:</th>
<th>4</th>
<th>3</th>
<th>4</th>
<th>1</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1110</td>
<td>001</td>
<td>Opcode</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>Rotate</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

- 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

  **How large of constants should we allow for? If they are too big, we won’t have enough bits leftover for the instructions or operands.**
Rotations to make constants

Recall that immediate constants are encoded in two parts: Thus, only a subset of 4096 32-bit numbers can be used directly as an operand.

There are actually only 3073 distinct constants. There are 16, "0s" and 4 ways to represent all powers 2.

From last time, how might you encode 256?

<table>
<thead>
<tr>
<th>Rotate</th>
<th>Bits Used</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0 - 255</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>-2147483648 - 1073741887</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>-2147483648 - 1879048207</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>-2147483648 - 2080374787</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>-2147483648 - 2130706432</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>4194304 - 1069547520</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>1048576 - 267386880</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>262144 - 66846720</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>65536 - 16711680</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>16384 - 4177920</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>4096 - 1044480</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>1024 - 261120</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>256 - 605280</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>64 - 16320</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>16 - 4080</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>4 - 1020</td>
</tr>
</tbody>
</table>
**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.
Load and Store Instructions

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:**
```
 4 3 1 1 1 1 1 4 4 12
```

```
D type:
1110 010 1 U 0 0 L   Rn   Rd   Imm12
```

**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
```

```
X type:
1110 011 1 U 0 0 L   Rn   Rd   Shift [L A] 0   Rm
```

**If U is "0" subtract offset from base, otherwise add them.**

**L is a "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  $Rd,[Rn,#imm12]  \[\text{Rd} \leftarrow \text{Memory}[Rn + \text{imm12}]\]  
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  $R0,[R1,#-4]  \[\text{Memory}[R1 - 4] \leftarrow R0\]  
Offsets can be either added or subtracted, as indicated by a negative sign.

LDR  $R2,[R3]  \[\text{if no offset is specified it is assumed to be zero}\]

STR  $R4,[R5,R6]  \[\text{The contents of a second register can be used as an offset rather than a constant (using the X-type format)}\]

LDR  $R4,[R5,-R6]  \[\text{Register offsets can be either added or subtracted, like constants}\]

STR  $R4,[R5,R4,LSL 2]  \[\text{Register offsets can also be optionally shifted, which is great for indexing arrays!}\]
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.

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).

<table>
<thead>
<tr>
<th>Cond</th>
<th>101</th>
<th>L</th>
<th>Imm24</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>EQ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>NE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>CS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>CC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>MI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>PL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>VS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>VC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>HI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>LS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>GE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>LT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>GT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>LE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- equals
- not equals
- carry set
- carry clear
- negative
- positive or zero
- overflow
- no overflow
- higher (unsigned)
- lower or same (unsigned)
- greater or equal (signed)
- less than (signed)
- greater than (signed)
- less than or equal (signed)
- always
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>Cond</th>
<th>000</th>
<th>10010 1111 1111 1111 0001</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 - EQ</td>
<td>equals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001 - NE</td>
<td>not equals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010 - CS</td>
<td>carry set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011 - CC</td>
<td>carry clear</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 - MI</td>
<td>negative</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101 - PL</td>
<td>positive or zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110 - VS</td>
<td>overflow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111 - VC</td>
<td>no overflow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000 - HI</td>
<td>higher (unsigned)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001 - LS</td>
<td>lower or same (unsigned)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010 - GE</td>
<td>greater or equal (signed)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011 - LT</td>
<td>less than (signed)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100 - GT</td>
<td>greater than (signed)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101 - LE</td>
<td>less than or equal (signed)</td>
<td></td>
<td></td>
</tr>
<tr>
<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
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?