# Lecture #5 – February 6, 2004

# Status Flags, Conditional Execution, & Binary

## Status Flags and Conditional Execution

Four ***status bits*** in the ***program status register*** (***PSR***) are set or cleared as a result of executing other instructions. Arithmetic instructions (such as `ADD`, `SUB`, `AND`, `ORR`, etc.) normally *do not* affect the status bits without a special notational tag. Appending an `S` to the end of each (writing `ADD` as `ADDS` and `SUB` as `SUBS`, for example), forces the instruction results to change the status bits appropriately. The four comparison instructions `CMP`, `CMN`, `TST`, and `TEQ` automatically set the status bits appropriately and do not require the `S` suffix. Instructions to load and store registers to and from memory never change the status bits.

The four status bits used by the conditional checks are `Z` (zero), `N` (negative), `V` (overflow), and `C` (carry). For any instruction form that affects the status bits, the `Z` flag is set to 1 if the result of the instruction is *exactly zero* and is cleared to 0 if the result is any other value. The `N` flag is set to 1 if the numeric result can be interpreted as negative, and is cleared to 0 if the numeric result is "not negative" (positive or zero). The `V` flag is set to 1 if an addition or subtraction results in an overflow condition (the sum of two positive numbers is negative, for example), and is cleared to 0 if no overflow occurs. The `C` flag is the ***carry bit***, and is the "$33^{RD}$ bit" of an addition or subtraction. The `C` flag is added into the sum in an `ADC` (add with carry), but not in a plain `ADD` instruction.

The ARM is an unusual design in that *every* instruction may be conditionally executed. There are 16 possible conditions, including "always" (no condition) and "never" (which is never actually used, but is a code reserved for expansion in future versions of the chip). The remaining 14 conditions test the `Z`, `N`, `V`, and `C` status bits, either singly or in combination. These 14 conditions are indicated by a two-letter code, appended to the end of any instruction requiring conditional execution. The two-letter code is omitted when the instruction must always be executed. If an instruction requires conditional execution and must also set the status flags as a result, the condition code is appended first, followed by the `S`. The eight two-letter codes that test single flag bits are shown below:

```
CODE  TEST  MEANING
EQ    Z=1   Equal (to Zero)
NE    Z=0   Not Equal
CS    C=1   Carry Set
CC    C=0   Carry Clear
MI    N=1   Negative
PL    N=0   Positive or Zero
VS    V=1   Overflow
VC    V=0   No Overflow
```

The remaining two-letter codes that use multiple flags are shown below:

| CODE | TEST | MEANING |
|---|---|---|
| HI | ((NOT C) OR Z)=0 | Unsigned Higher |
| LS | ((NOT C) OR Z)=1 | Unsigned Lower or Same |
| GE | (N EOR V)=0 | Signed Greater Than or Equal |
| LT | (N EOR V)=1 | Signed Less Than |
| GT | (Z OR (N EOR V))=0 | Signed Greater Than |
| LE | (Z OR (N EOR V))=1 | Signed Less Than or Equal |

## Example

Consider the ADD instruction as an example. By itself, the ADD opcode means to always perform the addition. By adding the CS condition suffix, the ADDCS opcode means to perform the addition only if C=1 (i.e., if the carry status bit is set to 1). Appending the S to create ADDS means that the result of the unconditional addition must modify the status flags. The opcode combination ADDCSS performs the addition only if C=1, but also modifies the status bits afterwards (of course, the instruction doesn't modify the status bits if it isn't executed).

Conditional execution allows the assembly language programmer to create very compact code for certain high-level structures. Consider the following If-Then-Else pseudo-code structure:

```
If Z=1 Then
     R0 := R0 + 1
Else
     R0 := R0 - 1 ;
```

In "traditional" assembly languages, this must be written as two independent blocks of isolated code, with jumps placed so that only one of the blocks is ever executed. By using conditional execution, the ARM code to perform the same task requires only two lines:

```
ADDEQ  R0,R0,#1      If Z=1 Then R0 := R0 + 1
SUBNE  R0,R0,#1      If Z=0 Then R0 := R0 - 1
```

It is important to realize that for this construction to work correctly these two instructions *must not* also set the status flags. If the first instruction is written as ADDEQS, executing the code fragment when Z=1 causes Z to reflect the result of the addition; in all cases where the sum isn't zero the subtraction *will be performed as well*!

## Binary Representations

A decimal number such as 1204 is really shorthand for the following expansion:

$$1\times10^3 + 2\times10^2 + 0\times10^1 + 4\times10^0$$

A ***binary number*** has a similar expansion, but uses powers of 2 instead of powers of 10. For example, the binary number 110101 is shorthand for the following expansion:

$$1\times2^5 + 1\times2^4 + 0\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$$

By reducing out the powers, this expression can be rewritten as:

$$1\times32 + 1\times16 + 0\times8 + 1\times4 + 0\times2 + 1\times1$$

All digits in binary are either 0 or 1, so reducing out the multiplications is trivial:

$$32 + 16 + 0 + 4 + 0 + 1$$

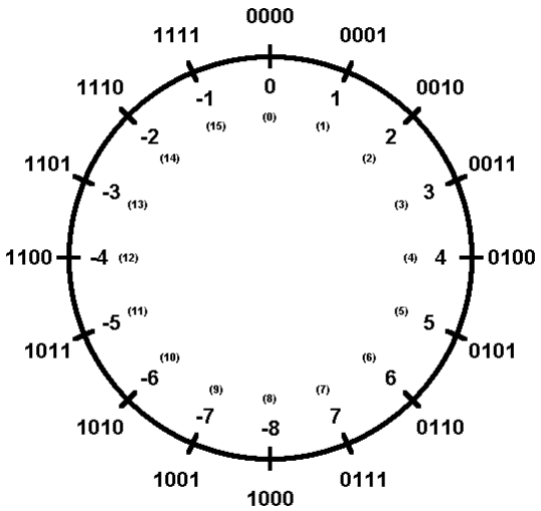
Computing the final sum gives us 53, which is the decimal equivalent to the binary number 110101. Any binary number can be converted into its decimal equivalent by using the same technique. Enumerating all possible binary patterns in 32 bits gives us over four billion results:

```
00000000000000000000000000000000   =    0
00000000000000000000000000000001   =    1
00000000000000000000000000000010   =    2
00000000000000000000000000000011   =    3
00000000000000000000000000000100   =    4
00000000000000000000000000000101   =    5
                  …                      …
11111111111111111111111111111100   =    4,294,967,292
11111111111111111111111111111101   =    4,294,967,293
11111111111111111111111111111110   =    4,294,967,294
11111111111111111111111111111111   =    4,294,967,295
```

This is a little too large a workspace to use as a teaching tool, so most people cut this down a bit! For all intents and purposes, we can cover the same topics and concepts using four bits instead of 32, and we can examine all possible behaviors and combinations of behaviors. With four bits, there are exactly sixteen distinct ***unsigned binary*** patterns to consider ($2^4$=16):

```
0000 =    0
0001 =    1
0010 =    2
…         …
1111 =    15
```

The next binary pattern after `1111` (decimal 15) is `10000` (decimal 16), but since that pattern requires five bits instead of four we can legitimately say that `1111+1=0000`. Thus, we can write all such binary numbers around the rim of a wheel:




By this diagram we can see that while `1111` is equal to decimal +15, it can also be considered as -1 *at the same time*. By counting counterclockwise, we label all negative numbers until we have an equal number of negatives as positives. All such negatives share the common characteristic that the leftmost bit equals 1. Examining the leftmost bit is how we determine if a ***signed binary*** number is either positive or negative, so `0000` is considered to be positive. With the special pattern `1000`, considered negative, we have one more negative number than we have positives. In the result of an arithmetic instruction, the leftmost bit of the result is used to set or clear the `N` status bit.

Now, adding +5 to +4 (counting clockwise from zero) we get +9 if we treat all numbers as *unsigned* integers. At the same time, adding +5 to +4 gives us -7 if we treat all numbers as *signed* integers; this is a ***signed overflow condition*** for signed arithmetic. When a signed overflow is detected the `V` status bit is set to `1`, but if the result is valid under signed arithmetic the `V` bit is cleared to `0`. Signed overflows happen when the boundary between +7 and +8/-8 is crossed, in either direction (adding two positives and getting a negative result, or adding two negatives and getting a positive result).

In unsigned arithmetic adding +10 to +7 normally gives us +17 (binary `10001`), but in four bits it results in binary `0001` (decimal +1) with the carry bit set to `1`. Thus, in addition setting `C=1` indicates an ***unsigned overflow condition***. Unsigned overflows, involving the carry bit, happen when the boundary between +15 and 0 is crossed.

Consider the example that adds `1010` to `1001`. The true binary result is `10011`. In our 4-bit situation, the 4-bit result is `0011`, and `C` is set to `1`. At the same time, `Z` is cleared to `0` (the answer isn't exactly `0000`), `N` is cleared to `0` (`0011` is considered to be positive because its leftmost bit is `0`), and `V` is set to `1` (two negative numbers, -6 and -7, were added together and the result is positive).

In this example, the C, V, and N bits are either examined or ignored according to the interpretation of the numbers. If the numbers are unsigned, C is meaningful but V and N are ignored as meaningless. The situation is reversed if the numbers are considered to be signed. Regardless of the interpretation, the Z, C, V, and N bits are all set or cleared by the addition instruction according to their respective rules. It is the job of the programmer to determine which bits to use and which to ignore.

## Conclusions

The situation with respect to the Z, C, V, and N bits is identical in the ARM to our 4-bit example, just extended to 32 bits. When instructions are configured to modify the status bits, all 32 bits must be zero for the Z bit to be set to 1, and the leftmost of those 32 bits contains the sign information for use by the N and V bits. A carry out of bit 32 goes to the C bit.

In the ARM an ADD instruction adds just its two 32-bit operands, but an ADC adds the two operands and the current value of the carry bit. Both produce a 32-bit sum, and the $33^{RD}$ bit, 0 or 1, becomes the new value of C if the instruction is configured to modify the status bits (ADDS or ADCS).