Lecture 3

Flags

Flags are used to give a signal that something either has or has not happened.

For example, the flag is raised to half mast on in front of buildings as a signal that someone important is dead.

Microprocessors also use “flags” to signify what has happened.

Flags and Processors

Flags are essentially 1 bit memory devices and different microprocessors have different flags.

The ARM microprocessor has four flags that are commonly found in the majority of microprocessors.

These flags are the zero flag (Z), the negative flag (N), the carry flag (C), and the overflow flag (V).

Flags are also know as ‘condition codes’.

Flags – example

If an instruction such as a subtraction produces a result which is 0x00000000 then the zero flag would be set to 1.

An ARM instruction that would set the zero flag is :

SUBS r2, r1, r1

That is to subtract value in r1 from itself and put the difference (zero) in r2.

Flags

Note that the mnemonic is SUBS rather than SUB
-- these are actually two different instructions with different machine codes.
-- they perform the same function except SUBS will set or clear the flags and SUB will not change the flags from their current setting.

This applies to all ALU instructions we have covered so far and also MOV.

Flags

The following instructions do not alter the flags:

MOV, ADD, SUB, RSB, MUL, MLA, AND, EOR, ORR, BIC

The following instructions do set or clear the flags as required:

MOVS, ADDS, SUBS, RSBS, MULS, MLAS, ANDS, EORS, ORRS, BICS

What are flags used for?

Flags are used in two ways:

The main use of flags is to determine if another instruction is executed or not. This is called conditional execution.

The carry flag can also be used in some arithmetic instructions as an additional value (we will return to this later).

Conditional Execution

- Conditional execution means that an instruction either does or does not execute depending upon the state of the flags (or 'condition codes').
- Almost all ARM instructions can be conditionally executed and this is indicated by the addition of two letters in the mnemonic of the instruction
- e.g.
  - MOVCS r12, #114
    - means move 114 into r12 if the carry flag is set.

Condition Fields

- There are 15 different condition fields which may be appended to (almost) any mnemonic.
- The common ones are:
  - EQ meaning ‘equal’ and an instruction with this extension is executed only if the zero flag is set.
  - NE - ‘not equal’ - executed if Z is clear.
  - CS - ‘carry set’ - executed if C is set.
  - CC - ‘carry clear’ - executed if C is clear.

More Condition Fields

- MI meaning ‘minus’ and an instruction with this extension is executed only if N is set.
- PL - ‘plus’ or ‘positive’ - executed if N is clear.
- VS - ‘overflow’ - executed if V is set.
- VC - ‘no overflow’ - executed if V is clear.
- There are 7 more condition fields including ‘always’ which is the default if no condition field is specified. (page 113 or see next slide)

Question

- What values are held in r4 and r5 after the execution of the following?
  - SUBS r4, r7, r7 ;
    - subtract r7 from r7
  - MOVS r4, #17 ;
    - move 17_{10} into r4
  - ADDNE r5, r4, #76 ;
    - add r4 to 76_{10}
- What happens to the zero flag?

15 different condition fields

Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

- AL is the default condition if there is no condition field used in the mnemonic.
- Note that some conditions apply to unsigned integers and others are used with signed integers, that is 2's complement.

Branches

- An important use of conditional execution is in branches.
- Conditional branches are used when a processor performs one function rather than another
  - e.g.
    - display text messages rather than receive an incoming call.
- First consider the unconditional branch.

The unconditional branch.

- An unconditional branch always executes and it reloads the program counter with a new value.
- The program counter, r15, automatically increments by four as each instruction is executed EXCEPT when a branch instruction is executed.
- The purpose of a branch instruction is to continue the program at a different location in memory.

Branch - example

- Consider the following example program:

Memory address	Instruction
0x00000080	EOR r4, r5, r9
0x00000084	B 0x0000008C ; branch
0x00000088	AND r5, r9, r2
0x0000008C	MLA r8, r6, r6, r2
0x00000090	B 0x00000088 ; branch

- These instructions are executed in the following order: EOR, B 0x8C, MLA, B 0x88, AND, MLA, B 0x88, AND, MLA, B 0x88, *etc*.

Conditional branches

- Unconditional branches are of very little use but conditional branches have many uses.
- Any condition field can be used with branch
  - e.g.
    - BNE 0x08C00000
      - means branch to 0x08C00000 if not equal and
      - either the instruction at address 0x08C00000 if Z is clear or
      - the instruction immediately following the branch if Z is set will be executed next.

Question

- What values are held in r4, r5 and r15 after the execution of the following?

Memory Address	Instruction
0x00008000	SUBS r4, r7, r7
0x00008004	BEQ 0x0000800C
0x00008008	MOVS r4, #17
0x0000800C	ADD r5, r4, #76

Branch - restrictions

- The memory address in the branch instruction must be contained within the instruction itself.
- 24 bits of the 32 bit instruction code are used to determine the destination address of the branch as an offset from the current program counter value.
- This means that the destination address must be within ±32MB of the memory address of the branch instruction.

Branch - mnemonics

- In general assembly language programs are written without any knowledge of the memory address for the corresponding machine code.
- So the mnemonic for branch uses ‘labels’ rather than actual memory addresses and
- The assembler program determines the actual value used.
- E.g.
  - BNE continue
  - SUB r4, r7, r7
  - continue ADD r5, r4, #76

Use of the zero flag

- The zero flag is commonly used to test if a program ‘loop’ has been executed a certain number of times;
- for example
  - if one wished to add together 10 numbers from memory
  - this could be implemented in a loop that repeated 10 times.
  - A register would be chosen as the ‘loop counter’;
  - during each loop the counter would decrease by 1 and
  - the loop repeats until the counter reaches 0.

Use of the zero flag - example

- Program to add together 10 values from memory 0x00000080 onwards:

```
        MOV r0, #9 ; set loop counter
        MOV r1, #0x080 ; set address for data
        LDR r2, [r1], #4 ; load 1st value
loop    SUBS r0, r0, #1 ; decrement counter
        LDR r3, [r1], #4 ; load next value
        ADD r2, r2, r3 ; add to running total
        BNE loop ; branch back to loop if zero flag is cleared
          because r0 does not hold zero.
```

The carry flag

- The carry flag is used in many arithmetic and logic instructions.
  - The use of the carry flag is best illustrated by looking at addition.
- Consider a 32 bit processor such as the ARM7.
  - If the sum of two numbers is greater than 0xFFFFFFFF (= 4,294,967,29510) then
  - the sum will have more than 32 bits and it cannot be fitted into a 32 bit register.

Setting the carry flag

- The carry flag will be set if the sum is greater than 0xFFFFFFFF using the instruction ADDS.
- Hence the following instructions will set the carry flag:

```
MOV r2, 0xF2000000
MOV r7, 0x11000000
ADDS r9, r2, r7
```

- The sum should be 0x103000000 or $100000011000000000000000000000000_2$ but the register can not hold the most significant bit.

Question

- What happens to the zero and carry flags after each addition in the following?

```
MOV r5, #0xFFFFFFFF
ADDS r4, r5, #0 ;add 0 to r5
ADDS r4, r5, #1 ;add 1 to r5
ADDS r4, r5, #2 ;add 2_10 to r5
```

Use of the carry flag

- The carry flag can be used in two ways:
  - In common with the other flags it can be used to determine if a conditional instruction is executed or not.
    - E.g.
      - ADDCS r1, r1, #1 will only execute if the carry flag is set and
      - ‘BCC label’ will only branch if the carry flag is clear.
  - The other use of the carry flag is in the addition instruction ADC (and the subtraction instructions SBC and RSC).

Add with carry

- The instruction ADC ‘add with carry’ adds together the two values and adds another 1 if the carry flag is set.
  - E.g.
    - ADC r0, r1, #3 ;
      - with value in r1 equal to 2
      - if carry flag is clear the sum in r0 is 5 (=3 + 2) but
      - if the carry flag is set the sum in r0 is 6.

Using add with carry

- ADC is used when we add together numbers greater than (2^{32}-1)
  - e.g.
    - $12{,}000{,}000{,}000_{10}$ added to $14{,}000{,}000{,}000_{10}$ which in hexadecimal is 0x2CB417800 added to 0x342770C00.
    - Both numbers are 34 bits in length and each one can be stored in two registers.
    - E.g.
      - r0 could hold the value 0xCB417800 and r1 could hold the value 0x00000002 for $12{,}000{,}000{,}000_{10}$

Using add with carry

- Example:
  - r0 holds 0xCB417800 and r1 holds 0x00000002
  - r2 holds 0x42770C00 and r3 holds 0x00000003

    ADDS r4, r2, r0
  - The sum of r2 and r0 is greater than 0xFFFFFFFF
  - so the carry flag is set and r4 holds the lowest 32 bits: 0x0DB88400.

Using add with carry

- Next instruction:

  ADC r5, r3, r1

- Because the carry flag is set, an extra 1 is added into the sum so r5 will hold 0x00000006.
- Taken together r5 and r4 hold the value 0x60DB88400 which is $26{,}000{,}000{,}000_{10}$

Negative numbers

- There are two main methods for representing negative numbers in microprocessors.
- These are:
  1) Sign magnitude.
  2) Two's complement.
- In each case the most significant bit - 'm.s.b.' - indicates the sign (1 for -ve and 0 for +ve).

The sign bit

- If the m.s.b. or ‘sign bit’ is 1 the number is -ve and if the m.s.b. is 0 the number is +ve.
- To find out if a number is -ve or +ve we could use:

  MOVS rx, rx

- The value in register rx remains unchanged but the negative flag is set if the m.s.b. is 1 and it is cleared if the m.s.b. is 0.

Sign magnitude

- Using the sign magnitude method a negative number is the same as a positive number but with the m.s.b. or ‘sign bit’ equal to 1.
- E.g. in 16 bits:

Decimal	Binary	Hexadecimal
+160	0000000010100000	0x00A0
-160	1000000010100000	0x80A0
+20640	0101000010100000	0x50A0
-20640	1101000010100000	0xD0A0

Sign magnitude

- In general any hexadecimal number is negative if it starts with 8 or greater and it is positive if it starts with 7 or less.
- So in 32 bits the number 0x800050A0 is negative and 0x000050A0 is positive using the sign magnitude method.
- The magnitude of a ‘sign magnitude’ number is easy to find:
  - simply AND with 0x7FFFFFFF.

Sign magnitude

- However sign magnitude numbers cannot be used for arithmetic.
- For example: 3 + (-3) should be 0.

Decimal	Hexadecimal
3	0x00000003
+(-3)	+ 0x80000003
?	0x80000006

- The answer is -6 rather than 0 !

2’s complement

- In the two’s complement method a negative number, -x, is given by the value (2^n - x) for an n bit processor.
- For example -3 in a 32 bit processor is:

$$\begin{array}{r} 0x100000000 \\ -3 \\ \hline 0xFFFFFFFD \end{array}$$

- So 0xFFFFFFFD is the 2’s complement representation of -3 in a 32 bit processor.

2's complement

- The two's complement method automatically sets the m.s.b. or 'sign bit' to 1.
- The following method can be used to find a 2's complement representation of a negative number,
  - e.g.
    - -20640
    - First find the positive value: 0x000050A0 or 0000 0000 0000 0000 0101 0000 1010 0000_2
    - Next invert all bits (0→ 1, 1 → 0).
      - 1111 1111 1111 1111 1010 1111 0101 1111_2 or
      - 0xFFFFAF5F.
    - And then add 1 ➔0xFFFFAF60

2’s complement

- So 0xFFFFAF60 is the 2’s complement representation of -20640_{10} .
- The inversion of bits can be implemented in hexadecimal rather than binary as follows:
  - Inverted No: F E D C B A 9 8 7 6 5 4 3 2 1 0
  - Original No: 0 1 2 3 4 5 6 7 8 9 A B C D E F

Question

- What is the two's complement of the following numbers in 32 bits?

$-1{,}500{,}000{,}000_{10}$
($1{,}500{,}000{,}000_{10}$ = 0x59682F00)

-114_{10} (114_{10} = 0x00000072)

-2006_{10} (2004_{10} = 0x000007D6)

  - Inverted No: F E D C B A 9 8 7 6 5 4 3 2 1 0
  - Original No: 0 1 2 3 4 5 6 7 8 9 A B C D E F

2’s complement

- This method also works in reverse so if you have a two’s complement number
  - e.g.
    - 0xFFFFFF60 and you want to know it’s value in decimal.
    - First the sign bit is 1
    - so it is a negative number.
    - Therefore invert and add 1:
    - 0x0000009F + 0x00000001 = 0x000000A0
    - which is 160 in decimal.
    - Hence 0xFFFFFF60 is the 2’s complement representation for -160_{10}

2’s complement

- Unlike sign magnitude, arithmetic is simple in two’s complement e.g. in 8 bits

Decimal	Binary
3	00000011
+(-3)	+ 11111101
0	1 00000000

- Note that if the carry bit (9th bit) is ignored the answer is 0 which is correct.

2's complement arithmetic

- In two's complement any hexadecimal number which has an 8 or greater for the most significant digit is negative since the 'sign bit' or m.s.b. is 1.
- So if we add two very big numbers so that the sum is greater than $(2^{31} - 1)$ then that number will be negative if we are working in 2's complement.
- E.g.
  - if we add $1{,}500{,}000{,}000_{10}$ to $1{,}100{,}000{,}000_{10}$
  - that is 0x59682F00 added to 0x4190AB00:

2’s complement arithmetic

```
   0x59  68 2F 00
+  0x41  90 AB 00
   ---------------
   0x9A F8 DA 00
```

- If we are working in two’s complement the sum is a negative number (= -1,694,967,296$_{10}$) and is clearly incorrect.
- If the instruction ADDS is used the negative flag would be set but the carry flag would be cleared because the sum is still a 32 bit result.

The overflow flag

- The overflow flag can be used to identify when a 2's complement result goes 'out of range'.
- E.g. for the last example in binary.

```
  0101 1001 0110 1000 0010 1111 0000 0000
+ 0100 0001 1001 0000 1010 1011 0000 0000
  1001 10...        ← result (top 6 bits)
```

- The overflow flag, V, is set because there is an overflow into the 'sign bit'.

Adding negative numbers

- Using two's complement we can do sums such as x + (-y) for example if we add $1{,}500{,}000{,}000_{10}$ to $-1{,}100{,}000{,}000_{10}$
- First find the 2's complement of $-1{,}100{,}000{,}000_{10}$

Positive number	0x4190AB00
Invert	0xBE6F54FF
Add 1	0xBE6F5500

- and now add this to 0x59682F00

Adding negative number

```
   0x  59 68 2F 00
+  0x  BE 6F 55 00
  ------------------
  0x1 17 D7 84 00
```

- Because we are working in 2's complement we can ignore the carry 1 and the answer in the lowest 32 bits is 0x17D78400 or $400{,}000{,}000_{10}$ which is correct.
- The carry flag is set but the overflow and negative flags are cleared.

Adding negative number

- Consider this in binary:

```
  0101 1001 0110 1000 0010 1111 0000 0000
+ 1011 1110 0110 1111 0101 0101 0000 0000
  0001 01...    ← result (top 6 bits)
1 1111 00...    ← carry from previous column
```

- The sign bit of the result is 0 indicating a positive result. A carry out occurs so C is set but the 32 bit result is correct so V is cleared.

A negative result

- What happens if we add two negative numbers or if we add a smaller positive number to a bigger negative number - the result should be negative.
- E.g.
  - if we add $1{,}100{,}000{,}000_{10}$ to $-1{,}500{,}000{,}000_{10}$
  - The 2's complement of $-1{,}500{,}000{,}000_{10}$ is 0xA697D100 and add this to 0x4190AB00.

```
      0x41 90 AB 00
+     0xA6 97 D1 00
      -------------
      0xE8 28 7C 00
```

A negative result

$$\begin{array}{rr} & 0x41\ 90\ AB\ 00 \\ + & \underline{0xA6\ 97\ D1\ 00} \\ & 0xE8\ 28\ 7C\ 00 \end{array}$$

- The result is negative as expected since the sign bit is 1 but is it correct?
  - Apply 2's complement to the result to find it's positive value
  - invert 0xE8287C00 to find 0x17D783FF and add 1 to find 0x17D78400
  - which is $400{,}000{,}000_{10}$ as expected.

A negative result

- Consider this in binary:

```
  0100 0001 1001 0000 1010 1011 0000 0000
+ 1010 0110 1001 0111 1101 0001 0000 0000
  1110 10...      ← result (top 6 bits)
0 0000 11...      ← carry from previous column
```

- The sign bit of the result is 1 indicating a negative result. There is no carry out and the 32 bit result is correct so C and V are both cleared.

2’s complement subtraction

- Subtraction, such as x - y, is implemented in a microprocessor by finding the two’s complement of (-y) and then performing the addition, x + (-y)
- E.g.
  - to subtract 0x4190AB00 from 0x59682F00 the microprocessor first finds the 2’s complement of 0x4190AB00 which is 0xBE6F5500 and then this is added to 0x59682F00 to find the result.

A negative overflow?

- So if we subtract a big positive number from a small negative number so that the difference is less than -2^{31} then that number will be positive if we are working in 2's complement.
- E.g.
  - if we subtract $1{,}500{,}000{,}000_{10}$ from $-1{,}100{,}000{,}000_{10}$ that is 0x59682F00 subtracted from 0xBE6F5500:

A negative overflow.

- 0xBE6F5500 – 0x59682F00 becomes 0xBE6F5500 + 0xA697D100

```
   0x  BE  6F 55 00
+  0x  A6  97 D1 00
  ---------------
  0x1 65  07 26  00
```

- Ignoring the carry 1 the 32 bit result 0x65072600 is +ive (= $1{,}694{,}967{,}296_{10}$) and clearly incorrect.
- Again this would set the overflow flag.

Flags - summary.

- The zero flag, Z, is set when the result (not including any carry out) is 0x00000000.
- The negative flag, N, is set when the most significant bit of the 32 bit result is 1.
- The carry flag, C, is set when the result (taken as an unsigned integer) is greater than (2^{32} - 1).
- The overflow flag, V, is set when the result (taken as a two's compliment number) is greater than (2^{31} - 1) or less than -2^{31}

Other number formats.

- Using 32 bits we can represent unsigned integers from 0 to (2^{32} - 1) and signed integers from -2^{31} to (2^{31} - 1) using the two's complement format.
- What if we want to represent numbers bigger than 4.295×10^9 (= 2^{32}) or fractional number less than 1 but not equal to 0?
- We can use two registers, that is 64 bits, to represent numbers up to 1.845×10^{19} but this uses twice as many bits and does not help with fractional numbers.

Floating point numbers

- The floating point format uses the equation $A \times 2^n$ to represent numbers.
- Using the IEEE 754 standard the A value has a sign and 24 bits and the exponent value, n, has 8 bits.
- The number $5{,}637{,}144{,}576_{10}$ ($= 2^{32} + 2^{30} + 2^{28}$) is too big for a 32 bit unsigned integer.
- In binary this number is
  - $1\ 0101\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2$
- First we must normalize this number as follows:

5,637,144,57610 in IEEE 754

- $1\ 0101\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 1.0101\ 0000\ 0000\ 0000\ 0000\ 000_2 \times 2^{32}$
- The exponent is $32_{10} = 100000_2$ but in the IEEE 754 format we must add $127_{10} = 1111111_2$to this to give 10011111_2 $(=159_{10})$
- In 32 bits this becomes:

  0100 1111 1010 1000 0000 0000 0000 0000
- Note that the normalization means that the m.s.b. of the A value is always 1 so this can be omitted.
- The m.s.b. of the 32 bit word is a sign bit.

8 bits for exponent
How to represent both negative and positive exponent?

0000 0000 to 1111 1111
Use middle point as zero -> 0111 1111 = 127
Less than 127 to represent negative exponent
Greater than 127 to represent positive exponent

IEEE 754 – special cases

- There are a number of special cases in the definition of IEEE 754 single precision format.
- They are: zero, plus and minus infinity, very small numbers less than 1.17×10^{-38} and ‘NaN’, not a number.
- Zero is represented by all zeros in both the exponent and the fraction although the sign bit can be either 0 or 1.

IEEE 754 – special cases

- Infinity is represented by all ones in the exponent and all zeros in the fraction with the sign bit used to indicate plus/minus so that 0x7F800000 is + ∞ and 0xFF800000 is – ∞.
  - 0111 1111 1000 0000 0000 0000 0000 0000
  - 1111 1111 1000 0000 0000 0000 0000 0000
- Very small numbers that can be expressed as $A \times 2^{-126}$ where A is of the form $\pm 0.xxxxxxxxxx_2$ are represented by the sign bit, followed by 8 zeros, followed by 23 bits of the fraction without the leading 0, i.e. $s00000000xxxxxxxxxxxxxxx_2$

IEEE 754 – special cases

- Not a number, NaN, is represented by either 0 or 1 for the sign bit, all ones for the exponent and any non-zero value for the fraction.
- NaN is used to indicate the result of an illegal floating point operation such as ‘divide by zero’ or ‘logarithm of a negative number’.
- The IEEE 754 definition also covers ‘double precision’ which is a 64 bit code.

0xE2600000 ?

- What is 0xE2600000?
  - -1.033×10^{21} in IEEE 754 single precision format,
  - −497,025,024 as a 2's complement number,
  - 3,797,942,272 as an unsigned integer or
  - RSB r0, r0, #0x0 as machine code?
- The context, i.e. the computer program / programmer, determines what the contents of a particular memory location represents.