GABRIEL MOISE

DIGITAL SYSTEMS

SHEET 1

① Some Thumb instructions that have the effect of setting register r0 to zero are:

```
2000      movs   r0, #0
1a00:     subs   r0, r0, r0
0fc0:     lsrs   r0, r0, #31     (am I allowed to shift by 32 positions? why? how do I encode that? does it mean I leave it unchanged?)
4c40:     eors   r0, r0
```

They all make the Z bit equal to 1, as the result is zero, and the other ones, N, V, C are 0. In general, movs can only affect the N and Z bits depending on the value that is stored at the destination, subs can affect any of them depending on the values used for the subtraction, lsrs can affect only N, Z and C as we cannot overflow when shifting and eors affects only N and Z, depending on the result.

Some Thumb instructions that have the effect of copying register r1 to register r0 are:

```
0008:     movs   r0, r1
0008:     lsls   r0, r1, #0
```

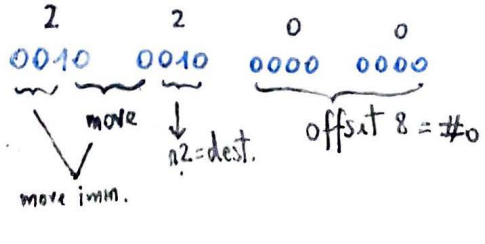
} They are equivalent

②

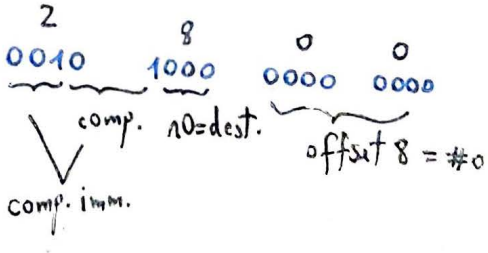
```
            foo:
c0:  2200   movs   r2, #0
            loop:
c2:  2800   cmp    r0, #0
c4:  d002   beq    ce <done>
c6:  3801   subs   r0, #1
c8:  1852   adds   r2, r2, r1
ca:  e7fa   b      c2 <loop>
            done:
cc:  0010   movs   r0, r2
ce:  4770   bx     lr
```

Decoding:

0x 2200:



0x 2800:



0x d002: 1101 0000 0000 0010

1101 → cond. branch; 0000 → beq; 0000 0010 → Soffset 8 → 2 positions

When we take the branch, the target address will be the address of the instruction, plus 4, plus twice the offset ⇒ initial address + 4 + 2*2

0x 3801: 0011 1000 0000 0001

0011 → subs. imm.; 1000 → subs, r0=dest.; 0000 0001 → offset 8 = #1

0x 1852: 0001 1000 0101 0010

0001 → add with reg. operand; 1000 → reg. add, r1; 0101 → r1; 0010 → r2=source, r2=dest.

0x e7fa: 1110 0111 1111 1001

1110 → unconditional branch = b; 0111 1111 1001 → offset 11 → -6 positions

We proceed the same way: initial address + 4 − 2*6.

0x 0010: 0000 0000 0001 0000

0000 → lsls; 0000 0001 0000 → offset 5 = #0, r2=source, r0=dest.

equivalent to movs r0, r2

0x 4770: 0100 0111 0111 0000

0100 0111 → bx; 0111 0000 → lr

Question: What is a displacement more precisely?

The number of positions we move in the program when we have branches?

③

```
foo:
    movs  r2, #0
    subs  r2, r2, r0
    adds  r1, r1, #1
loop:
    adds  r2, r2, r0
    subs  r1, r1, #1
    bne   loop
done:
    movs  r0, r2
    bx    lr
```

@ This is the reverse of a loop as we can only perform the operation from the loop only if y = r1 > 0

@ We go back in the loop as long as y ≠ 0. bne replaces the cmp and beq done.

In total, in our loop we have 3 instructions and 5 cycles (3 for bne, 1 for adds, 1 for subs).

④ beq, as all conditional branches have just 8 bits as offset, whereas b, the unconditional branch has 11 bits as offset. To be able to simulate conditional branches with a bigger range, we can:

```
foo:
    cmp  r0, r1    @ Do a comparison to set the NZVC flags
    bne  jump      @ if not equal, we branch to jump (jumping over the unconditional branch)
    b    next
jump:
    .....
    .
    .
    .
next:
```

Here, we basically created a beq with a bigger range. This can be done with all conditional branches that have complements (like beq and bne or blt and bge).

Instead of only 2 cycles (for cmp and beq), here we need 5 (3 for b).

The branch-and-link instruction bl uses the lr register, so in order to use it in a similar way to b, we need to store the value of lr somewhere else in order to be able to come back to the correct address after we are done with the sub-routine. ?

⑤ We take for example the 2 vectors of bits:

1011 1000 < = −72, for signed
= 184, for unsigned

and

0110 0100 = 100, both for signed and unsigned

Then, we need to subtract 100 from −72 with ⊖, so we'll calculate:

$-100 = 1001\ 1011 + 1$ (negate its bits)

$1001\ 1100$

Therefore, let's see what happens for each conditional branch:

$$\begin{array}{r} 1011\ 1000 + \\ 1001\ 1100 \\ \hline 1\ 0101\ 0100 \end{array}$$

Here, C=1, V=1 as the result appears to be positive, but $-72 < 0 < 100$ and because the result appears to be positive, N=0.

Then, blo r0, r1, where r0 = −72, r1 = 100 will be !C, meaning 0, or false, and this is true because for unsigned, −72 = 184, which is not less than 100.

Additionally, blt r0,r1 will be N!=V, which is true, because -72 <100, indeed.
I am not sure I got the calculation of the V flag properly, can we cover that at the tutorial?

⑥ The code for this problem is in C++, where I defined the function division, as:

```
unsigned int division (unsigned int x, unsigned int y);
{
   unsigned int result;
   result = 0;
   while (x>=y)                  // As long as I can subtract an y from x, we continue
   {
     x=x-y;                      // We do the subtraction
     result ++;                  // And increment result
   }
   return result;                // When we stop, x<y, so we return the result
}
```

After each loop, we have the invariant $x + result * b = a$, where a and b are the initial arguments, and we apply division (a,b). So, when $x < y$ (or $x<b$) we get that result is the quotient and x, the remainder

Now, let's code it in assembly:

```
foo:
    movs  r2, #0    @ result is in r2, a in r0 and b in r1
loop:
    cmp   r0, r1
    blo   done      @ comparing for unsigned
    subs  r0, r0, r1
    adds  r2, #1
    b     loop
done:
    movs  r0, r2
    bx    lr
```

Considering that this division is for unsigned numbers, if we try it for signed, we might get into a very long loop and we can get the wrong result. (At the lecture, Mike Spivey said that (-7) Div 4 = (-2) is mathematically correct and not (-7) Div 4 =(-1). What do you think?) The possibility of overflow does not affect the program, as blo only considers the C flag, so V here is irrelevant. ?

⑦ We can use clz this way:

1) if r0 < r1 then we stop

2) We compute clz r0 and clz r1. We notice that r0 is at least $r1 * 2^{clz\,r0 - clz\,r1 - 1}$ because if we left shift r1 by clz r0 - clz r1 - 1 bits, we still get a smaller number, because its clz is bigger with 1 than clz 0.

70 = 0100 0110
7 = 0000 0110 $\Rightarrow$ clz 70 - clz 7 = 4 $\Rightarrow$ $70 \geq 7 \cdot 2^{4-1} = 7 \cdot 2^3 = 7 \cdot 8 = 56$ true, as clz 70 = clz 56 - 1

So, we know that now we can subtract lsls r1, #(clz r0 - clz r1 - 1) (of course, we can't write it this way) from r0 and add to the result $2^{clz\,r0 - clz\,r1 - 1}$

! Notice that if clz r0 = clz r1 and r0 ≥ r1, we know that r0 div r1 = 1, so we will treat it as a seppanate case.

The program in assembly would look like:

```
foo:
    movs r2,#0      @The result is kept in r2
loop:
    cmp  r0,r1      @We compare a and b
    blo  done       @if a<b, we return result,which will be #0.
    clz  r3,r0      @We keep the number of leading zeros for a and b in r3 and r4,
    clz  r4,r1      @respectively
    cmp  r3,r4      @We compare the two results (we are ensured that clz a ≥ clz b now)
    beq  predone    @if they are equal, we return the result which is #1.
    subs r3,r3,r4
    subs r3,#1      @Now, r3 = clz a - clz b - 1 ≥0
    lsls r1,r1,r3   @We left-shift r1 by r3 positions
    movs r4,#1      @We want to add to r2 (the result) 2^r3, so r4 becomes 2^0=1, and
    lsls r4,r4,r3   @then we left-shift it by r3 positions
    adds r2,r2,r4   @We add 2^r3 to our result
    subs r0,r0,r1   @We subtract from a, 2^r3.b
    lsrs r1,r1,r3   @And we get b back to its original form, in r1
    b    loop       @We repeat this until we get to "done" or "predone"
predone:
    adds r2,#1      @It's exactly like "done", but we can subtract b from a one more time,
done:               @so we increase result by 1.
    movs r0,r2      @Now the result is in r0
    bx   lr         @We finish by returning r0
```