28 oct, 2022
Lec 12

# ARM7 Instruction set Part III

## Arithmetic Instruction

① ADD R0, R1, R2

R0 ← R1 + R2

* R0, R1, R2 all are 32 bit registers. The sum of R1 and R2 stores in R0.

~~the~~ the sum of two 32 bit number can be 32 bit or 33 bit

```
   99   ff
   99   ff
  ---  ---
  198  1fe
```

the extra bit is the carry.

if we want to get this carry flag value. we have to write

ADDS R0, R1, R2
 └ status

Now we will get all the 33 bit value.

② ADC R0, R1, R2

R0 ← R1 + R2 + CF

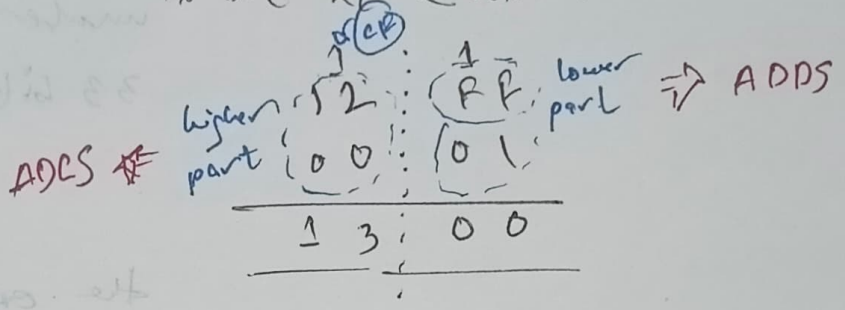
* ADC ~~mean~~ stands for add with carry.

sometimes we do 64 bit addition.

but ARM can operate 32 bit at a time. So we divide the user number by two part.

The summation of lower part can produce a carry. this carry is will be added with the higher 32 bit of number.

for example. a processor can add 16 bit at a time.




# By this way we can also add 64 bit, 128 bit, 256 bit... numbers. The first 32 bit will be added using ADD. The remaining bits will be added with using ADC.

(3) SUB R0, R1, R2

R0 ← R1 - R2

# For affecting the flags we have to write

SUBS R0, R1, R2.

there will be borrow when R1 is less than R2.

In ARM if there is borrow carry flag becomes 0. if there is no borrow, carry flag become 1.

It works like $\overline{CF}$.

* carry flag contains inverted borrow.

④ SBC R0, R1, R2

R0 ← R1 − R2 − $\overline{CF}$

#SBC stands for subtract with borrow.

Like addition. we can subtract 64bit, 128 bit.. by using SBC. The first 32 bit subtractions occurs using SUBS. The remaining bits will be subtracted using SBCS. 's' for affecting the flags.

# If we want subtract too large number. first we subtract the lower one than we subtract the higher one. If the lower one part is higher than it borrow it from the higher part.

for example (decimal number)

```
   1
   4  7
 - 1  9
 -------
   2  8
```

done first sub → SUB

second sub → ~~SUB~~ SBC

if there is borrow carry flag becomes 0.

$\therefore \overline{CF} = 1$.

if there is no borrow carry flag become 0.

$\overline{CF} = 0$.

~~SUB~~ SBC R0, R1, R2

R0 ← R1 − R2 − $\overline{CF}$ .

⑤ RSB R0, R1, R2

R0 ← R2 − R1

# RSB stands for ~~RSB~~ "reverse subtraction"

⑥ RSC R0, R1, R2

R0 ← R2 − R1 − $\overline{CF}$

# RSC stands for "reverse subtraction with carry.".

# Explain the instruction. RSCEQS R0, R1, R2

Ans: ~~RSCE~~ RSCEQS R0, R1, R2.

if the result of the previous instruction is equal. (that means zero flag is '1')

then RSC will execute (EQ Effect)

and subtract the result will affect the flags. (S effect)

# Multiply Instaction

# There is no instruction for division in ARM.

$\frac{7}{2} = 3.5$ (1) or quiont = 3, remainder = 1. (2)

# 8051 support division. it gives the answer in (2) way. But in modern day we want answer in (1) way (floting value)

# ARM dose not support the floting values. So there is no instr for division. when we need divin oper. operation we use an additional step with ARM to get division value with fractions.

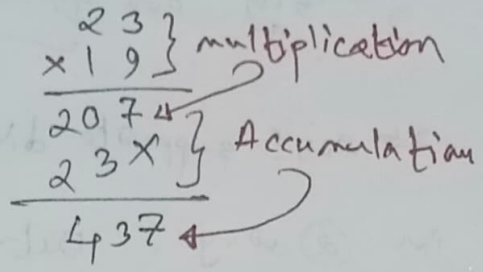
① MUL R0, R1, R2
R0 ← R1 × R2

# MUL stands for Multiply.

② MLA R0, R1, R2, R3
R0 ← (R1 * R2) + R3

# MLA stands for multiply and Accumulate
first it multiply R1 and R2 and adds the value with R3. and store it in R0.
# Normally R0 and R3 use the same registers.
MLA R0, R1, R2, R0
R0 ← (R1 × R2) + R0.

when normally we multiply several values and finally add all of them.




LONG MULTIPLY

* 32×32 ⇒ can generate 32 bit value.

32×32 ⇒ max size is 64 bit

The normal multiply works on

32×32 ⇒ 32.

The long multiply works on 32×32 ⇒ 64.

The long multiply actually do multiplication of 32 bit values.

⇒ so question arises why the normal multiplication is required.

Sometimes we works on 8 bit or 16 bit.

16×16 max generate 32 bit value.

for 8 bit or 16 bit multiplication normal multiplication can be used.

* Also normal multiply instr works only on unsigned numbers.

:) UMULL R0, R1, R2, R3

(R1, R0) ← R2 x R3

# 'UMULL' stands for unsigned multiplication. This instr is used to multiply 32 bit unsigned numbers.

:) SMULL R0, R1, R2, R3

# SMULL stands for signed multiplication. It is used to multiply 32 bit signed numbers.

# The processor follows two different approaches for unsigned and signed number multiplication

# UMULL → stands for Long.

:) UMLAL R0, R1, R2, R3

(R1. R0) ← (R2 x R3) + (R1. R0)

# UMLAL → stands for unsigned, ML stands for multiplication, A for accumulate. and the last L stands for Long.

first R2 x R3

second (R2 x R3) + (R1. R0)

finally the result will be stored in (R1 and R0).

:) SMLAL R0, R1, R2, R3

# SMLAL stands for signed multiplication and accumulate. → Long

# SMLALS x, y, z, w explain

first  zxw

second  zxw + y, x

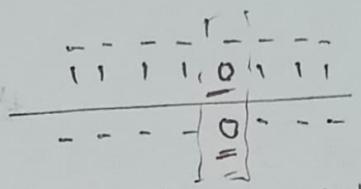
third  y, x ← zxw + y, x

fourth  affects the flags after operation.

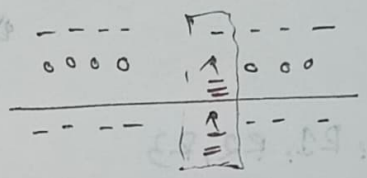
## Logic Instr

1. AND R0, R1, R2
   R0 ← R1 ∧ R2
2. OR R0, R1, R2
   R0 ← R1 ∨ R2
3. EOR R0, R1, R2
   R0 ← R1 ⊕ R2
4. BIC R0, R1, R2
   R0 ← R1 ∧ $\overline{R2}$

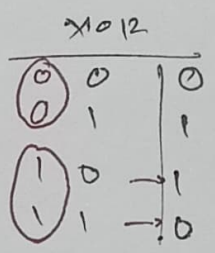
S needed to affect flags



# anything AND with 0 the bit becomes 0.



# anything OR with 1 the bit becomes 1.




# anything XOR with 1 the bit becomes complemented.

anything XOR with 0 the bit becomes remains the same.

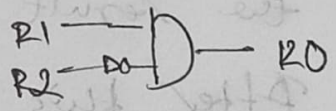
# Suppose we want to perform any specific operations on a ~~sigma~~ single bit without affecting the other bits of of number we can apply AND, OR, XOR ~~operation~~.

The operations are

- → make the bit 0.
- → make the bit 1
- → make the bit complement

# BIC is in AND operation but it ands with complement value of second register with first register.

BIC R0, R1, R2




R0 ← (R1 ∧ $\overline{R2}$)

(5) CMP R0, R1
(R0 − R1)

(6) CMN R0, R1
(R0 + R1)

(7) TST R0, R1
(R0 ∧ R1)

(8) TEQ R0, R1
(R0 ⊕ R1)

will not store result. But will affect flags. 's' not needed

# CMP compare is a subtraction operation. and affects the zero and negative flags.

# CMN (compare with neg negative value) negative means 2's complement.

2's complement of R1 is (−R1)

so, R0 − (−R1)

R0 + R1

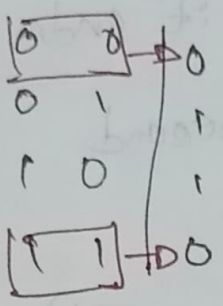
what is semaforce

* TST means Test.

this instruction the test whether the two values are semaforce or not.

It is an AND operation - It does not store the value but affects the flags.

XOR




* TEQ stands for "Test if Equal."

TEQ R0, R1
(R0 ⊕ R1)

If the two numbers are equal the result becomes zero.

After this instr we check zero flag. If zero flag is 1. The numbers are equal. otherwise the numbers are not equal.