SFU SIMON FRASER UNIVERSITY
ENGAGING THE WORLD

ENSC254 – Data-processing and multiplication Instructions

Ensc254 – Updated June 2021

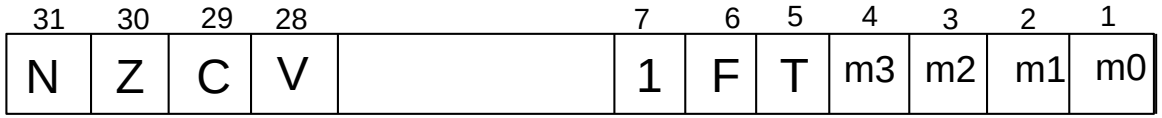
School of Engineering Science
Simon Fraser University
Burnaby, BC, Canada

Data-processing Operations

- We have previously discussed Memory Access operations
- Now we are moving into the heart of the processor, the ALU (and more generally the Datapath)
  - ***Datapath:*** *Part of the processor where computation is performed – can include ALU, Shifter, Multiplier, maybe a few additional ALUs, etc ….*
- Here, we can gain a very large part of our competitive advantage, because a large part of the instructions being completed by a processor are Data-processing operations.

ARM has a very peculiar Datapath, developed with power efficiency and instruction density in mind

Specificity of the ARM Architecture



- As is discussed in this course, ARM is a very peculiar architecture.
  - While it is RISC in using load/store, it has a very specific usage of the register file
  - **Many operations that would Write-back on a register in other RISC architectures will write on FLAGS in ARM**

1. This makes Compilation more complex (but most don't care, as it is done offline and ARM and others can afford to build good compilers for us)
2. This also makes the hardware a little more complex and Un-RISC-like, but it also makes Instruction memory consumption much smaller, and this is a great advantage

For this reason, we need to carefully review the Status register and the meaning of each Flag in it!

ARM versus Generic RISC architecture

SRA is Shift Right Arithmetic

http://programmedlessons.org/AssemblyTutorial/Chapter-14/ass14_14.html

C Code

```
if (a>b)
   { /* code */ };
/* moreCode */
```

RISC Asm (e.g. MIPS)

```
        SUB r1,r2,r3
        SRA  r1,r1,#31
        BNZ   r1, more
        ;@ ... code
more:   ;@ ... moreCode
```

ARM Asm

```
        CMP r2,r3
        BLE  more
        ;@ ... code
more:   ;@ ... moreCode
```

The ARM Assembly is a little more cryptic but

i. Uses a smaller number of registers (in fact ARM has 16 rather than 32 regs)

ii. Uses fewer instructions for the same task

- The difference is that ARM stores the CMP result in FLAGS inside the status register

Altering program status register

- Many operations MAY alter the program status register

BUT

Depending on their usage we do not necessarily want them to. If a given ARM instruction should alter the status flags we add an S after its mnemonic

ADD -> ADDS

MUL -> MULS

….

There are a few exceptions such as CMP, which always update the flags.

Wake-Up Call

What are the ARM program status register flags?

A. Registers of the register file where we are saving our temporary results

B. The Program Counters

C. Specific bits where the programming mode is saved (USER / SYSTEM / SUPERVISOR / ….)

D. A set of flags that are explicitly addressed during computation by ALU operations

E. A set of flags that may be implicitly addressed during computation by ALU operations

Wake-Up Call

What are the ARM program status register flags?

A. Registers of the register file where we are saving our temporary results

B. The Program Counters

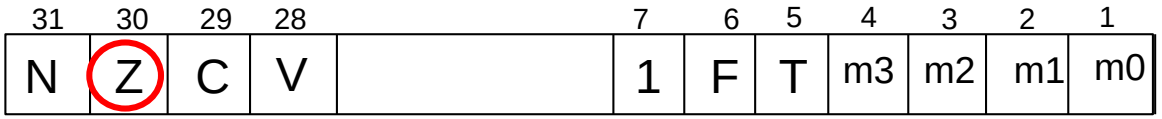
C. Specific bits where the programming mode is saved (USER / SYSTEM / SUPERVISOR / ….)

D. A set of flags that are explicitly addressed during computation by ALU operations

E. A set of flags that may be implicitly addressed during computation by ALU operations

ARM PROGRAM STATUS FLAGS

Z Flag : Zero Result



Flag Z is set when the result of a given operation is Zero

The reason ARM architects did this is to easily implement comparisons: there is no need below to waste an extra ALU operation to verify if R2 is actually zero. We can check that information during, for example, a subtraction and then annotate that with the Z flag.

C Code

```
if (--a == 0)
   { /* code */ };
/* moreCode */
```

ARM Asm

```
SUBS R2,R2,#1
BNE moreCode
;@ code here ...
moreCode:   ;@ more code here
```

CARRY OUT

Let's briefly review what a CARRY OUT is:

- Remember the standard method for Binary Additions: we call carry the temporary result of each bit addition that is «Carried out» to the following bit column
- We define Carry Out of the ALU the carry output of the most-significant column bit
  - CO(i) is typically calculated as CI(i)+A(i)+B(i)

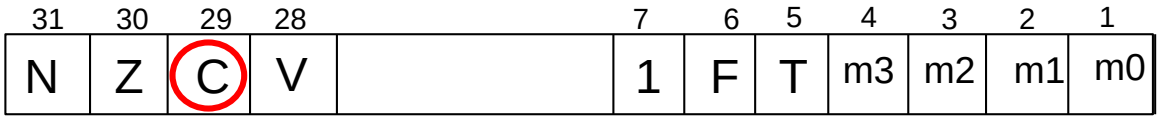
```
0101 +
0011=
--------
111
1000
```

Carry out of each column

```
0101 +
1110=
-------
1100
0011
```

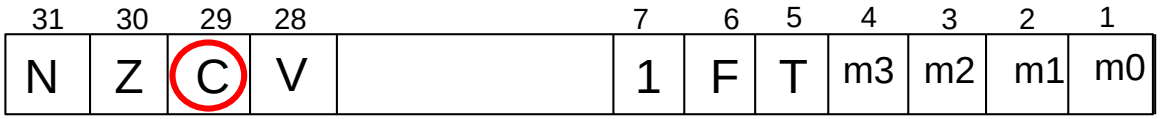
Carry Out of ALU

C Flag: Carry Out



Flag C is set to the Carry Out value of any ALU arithmetical instruction if the instruction is specified to update the program status register. The carry out of the instruction is the carry out value of the operation at bit 31

C Flag: Carry Out

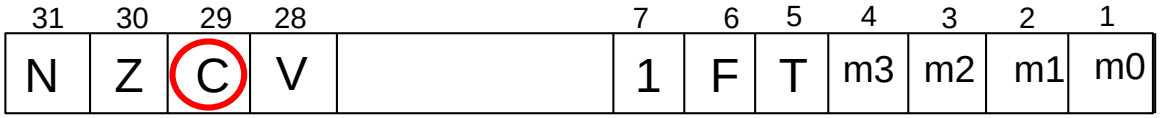


Flag C can be set to the Carry Out value of any ALU arithmetical operation

The carry out can be used for different purposes:

One relevant use is to perform 64-bit arithmetic on a 32 bit processor: the carry out of the first part of the operation can be used as Carry In for the second part of the operation

C Flag: Carry Out



Flag C for the Carry Out value of any ALU arithmetical operation

One relevant use is to perform 64-bit arithmetic on a 32 bit processor: the carry out of the first part of the operation can be used as Carry In for the second part of the operation;

ADDS = Add and set Carry flag

ADC = Add with Carry, uses the carry flag as *carry in* *(ci is added to the sum)*

C Code

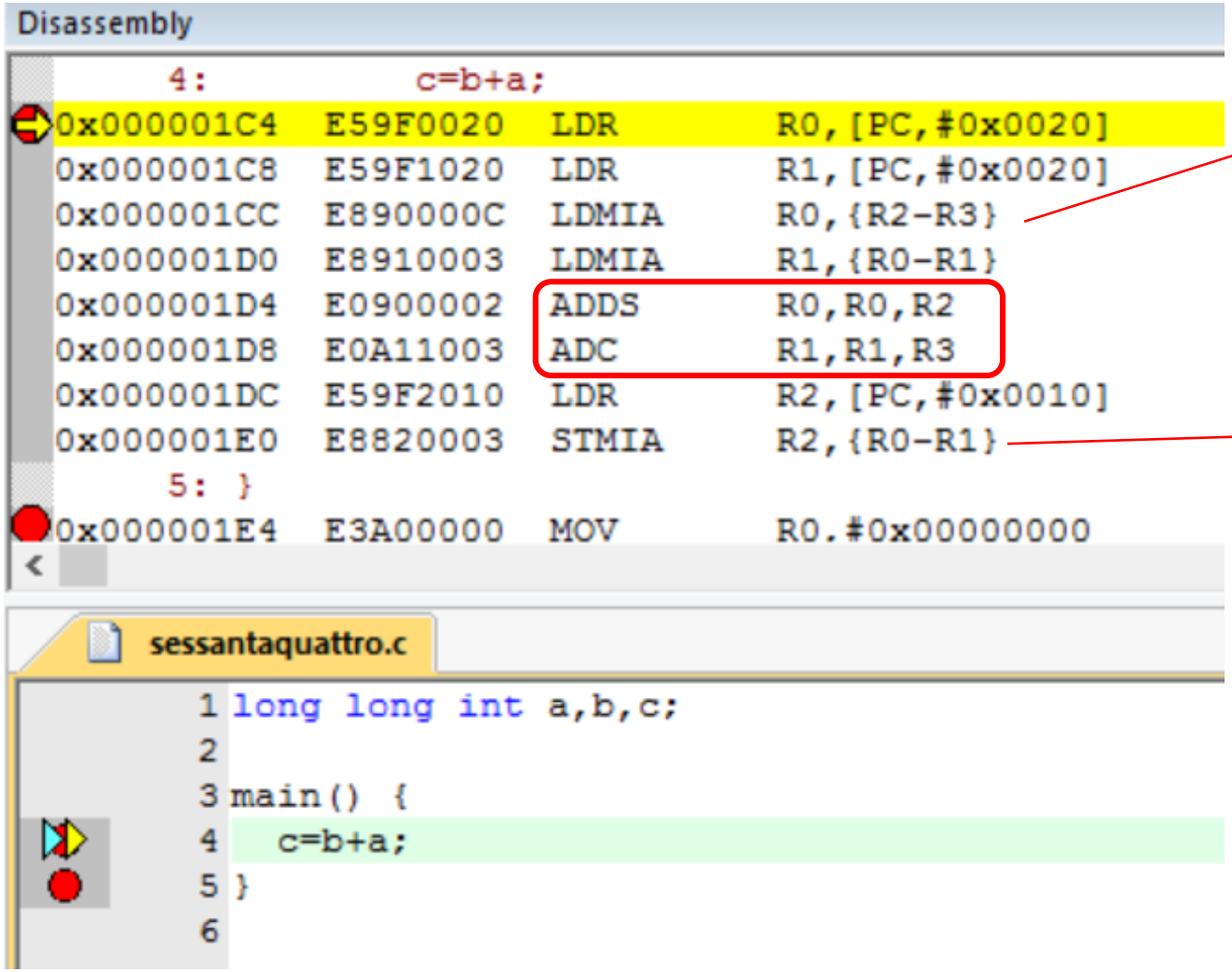
```
long long int a,b,c;
main() {
   c=a+b;} // a+=b;
```

We assume the mapping
a on r0,r1,
b on r2,r3
c on r0,r1

ARM Asm

```
ADDS r0,r0,r2
ADC  r1,r1,r3
```

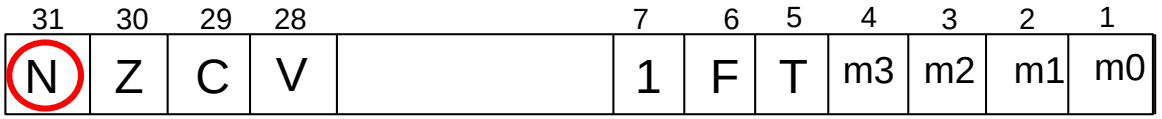
Example:



Load Multiple Registers

Store Multiple Registers

N Flag: Negative numbers



Flag N for when the result of a given operation is Negative,

Which means that the result of a SIGNED operation has its most significant bit (Bit 31) at 1. The reason ARM architects did this is to easily implement comparisons: there is no need to waste a target register for one bit, we can make it so that all conditions implicitly target a flag or two in the status register

C Code

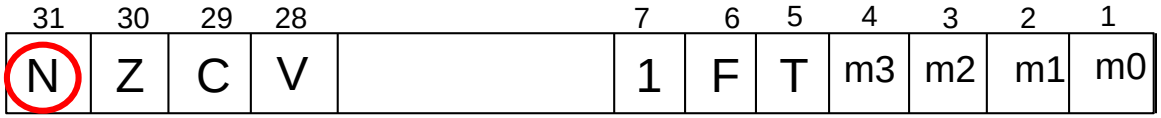
```
if (a >= b)
    { /* code */ };
/* moreCode */
```

moarCode

ARM Asm

```
CMP r2,r3
 BLT moreCode
;@ code here ...
;@ more code here
```

N Flag: Negative numbers



Flag N for when the result of a given operation is Negative,

Which means that the result of an operation has its most significant bit (Bit 31) at 1. The N flag is usually only relevant when a SIGNED operation was intended.

Please Note that in case of overflow, the N flag does not make any sense, and could assume unpredictable values

OVERFLOW: SILLY EXAMPLE

Suppose you are baby sitting your neighbor's kid. As it is raining (Nah! Around Vancouver? Impossible!) you try to kill time teaching him/her how to do additions counting with his/her hands

- Soon he/she gets the hang of it … he/she's so excited! *Congratulations, you have just created an engineer*

- Now he/she wants to count his friends at day care, but there are 7 boys and 5 girls. He runs out of fingers …..
  - He/She just discovered the concept of overflow!

Overflow and Negative Numbers

- Math in Computers has the fundamental feature of being limited to a given DATA_WIDTH (=32 in the case of the traditional ARM datapath)

  - An operation between two **«Legal»** numbers (valid 32-bit numbers), may generate an **«illegal»** number: Suppose you have the following operation between SIGNED numbers

7B00 0000 +

3000 0000 =

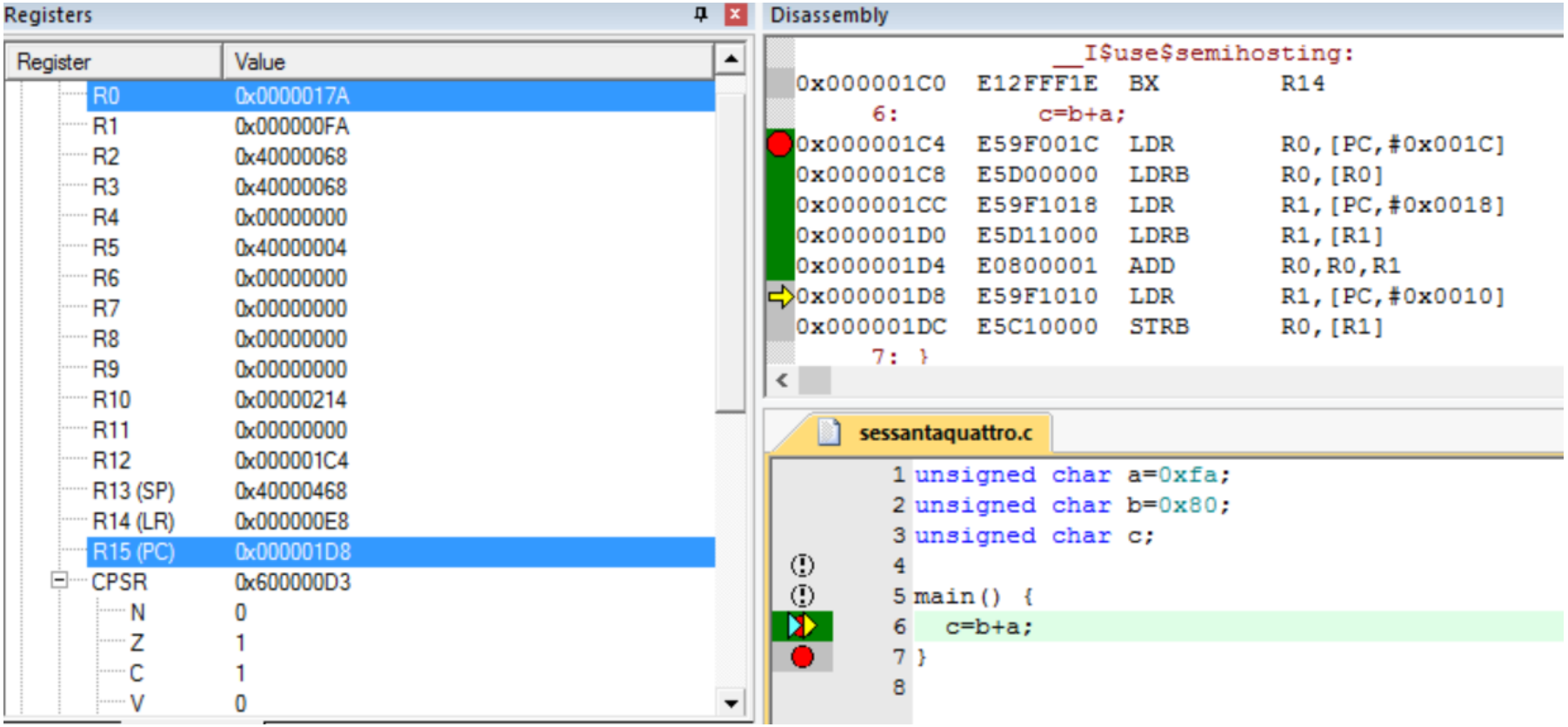
AB00 0000

NOTE : The result as a 32-bit signed number appears to be NEGATIVE. The sum of two positive numbers can not be a negative number. This occurrence is CALLED **OVERFLOW:** ***it essentially means that the result of the operation can not be represented in the number of bits that is available in the processor***

Quiz:

- Will the ARM core set the carry flag in case of overflow when operating on two 8-bit unsigned numbers?

Example:



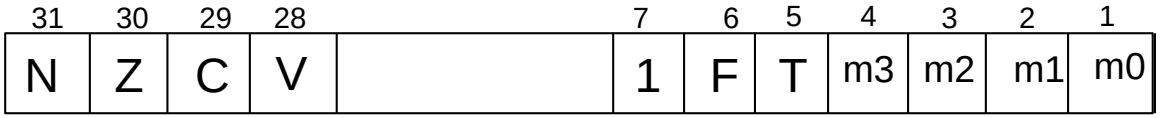
The ARM datapath after the loads DOES NOT KNOW that the numbers were bytes. In the register file, all data are 32-bit.
If we go on and store the result as a byte we will simply lose information!
Question: why is the carry flag shown as 1?

Note: Questions you might get during quizzes

1. Write an operation operating on 8-bit binary numbers that generates an overflow condition
   a. Between UNSIGNED numbers
   b. Between SIGNED numbers

2. Write an operation operating on 32-bit numbers in hexadecimal notation that generates an overflow condition
   a. Between UNSIGNED numbers
   b. Between SIGNED numbers

V Flag : oVerflow for SIGNED numbers



Flag V can be set when the result of a given ALU operation leads to oVerflow for SIGNED numbers.

Overflow conditions are determined by the following:

Unsigned Overflow: CARRY_OUT(31) (this is the C flag)

Signed Overflow: CARRY_OUT(30) XOR CARRY_OUT(31) (this is the V flag)

Note: **XOR** (exclusive or) is 1 when the two inputs are different:

0 **XOR** 0 = 0

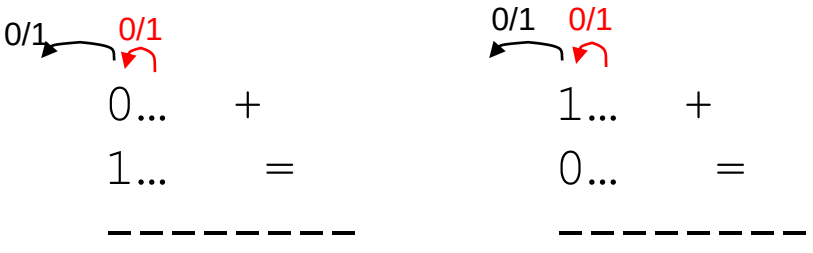
0 **XOR** 1 = 1

1 **XOR** 1 = 0

Overflow

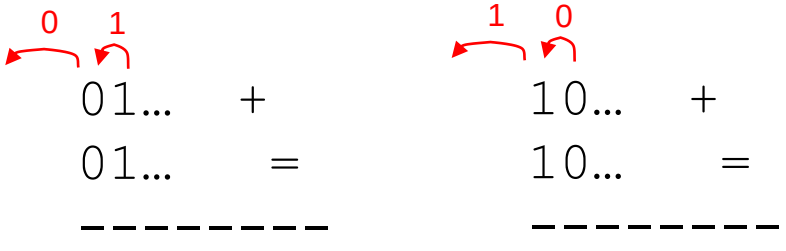
oVerflow: CARRY_OUT(30) XOR CARRY_OUT(31) (this is the V flag)

Note1: If the two numbers have different Sign, there can not be overflow on addition. *Indeed, if they have different Sign carry out 31 is 0 if carry out 30==0 and 1 if carry out 30==1 so CO[31] XOR CO[30] =0*



Positive + negative will always produce a number smaller in magnitude than one of the operands

Note2: If the two numbers have same Sign, there can be overflow. We will have overflow if the bit 30 is the same in the two numbers and different from bit 31



Positive + Positive or negative + negative will produce a number larger in magnitude than the two operands, so it may overflow

MOVE TO GP REGISTER (MRS)

- The main drawback of the «Program Status Register» strategy to processor design, is that ANY ALU operation with S bit set will in general destroy the status set by the previous such operation.

- If we need the status information set by an instruction, for example for a Conditional Branch, we can save it on a generic register (this may be a result of Feature Creep) with the instruction MRS (Move to general purpose Register from program Status register)

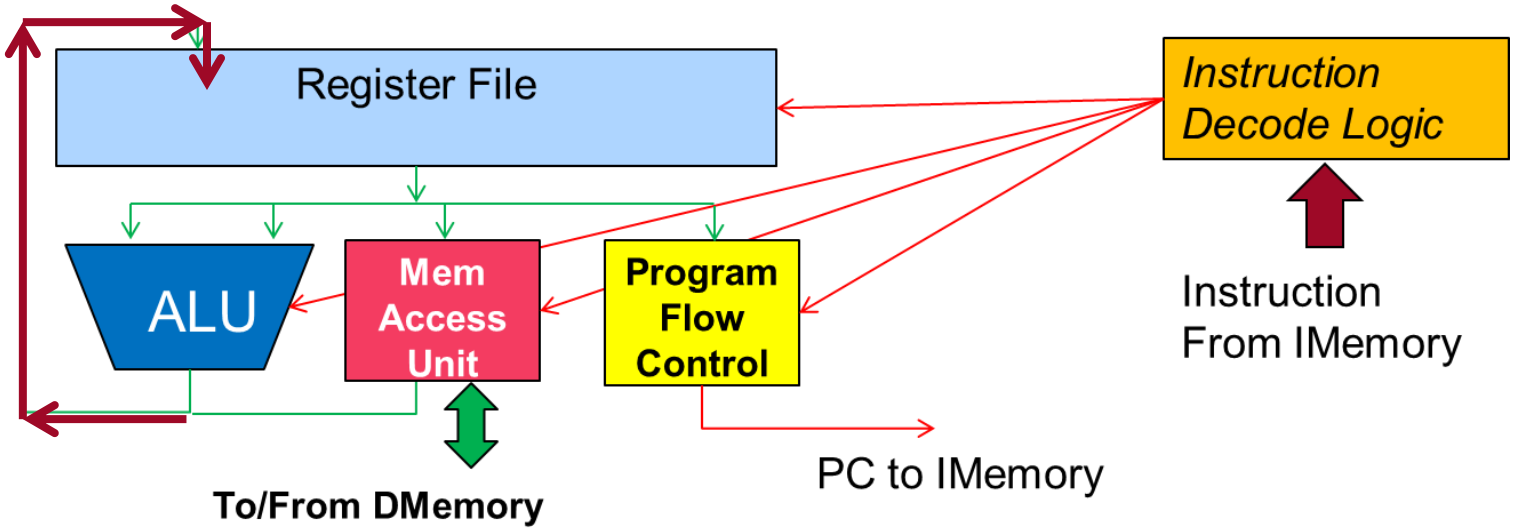
```
MRS r0, CPSR        ;@ R0 = CPSR
MRS r1, SPSR        ;@ R1 = SPSR ;@ we'll learn about SPSR later
```

Wake-up call

What is a ***Write-back*** operation?

A. Writing on the same register as the operation before, hence causing an error (you lose the information just saved)
B. Writing on the register file the output of any instruction
C. A store operation writing data into memory
D. The saving of the return address in case of branch and link operations
E. The store operation performed when writing on a cache memory

WRITEBACK



We define write-back the operation of writing a result into the register file.
Write-back is performed by

- Any Arithmetical-Logical operation
- Branch and Link operations
- **Load Operations**

DATA PROCESSING INSTRUCTIONS

Comparison Operations

As we just saw, we can use standard arithmetic operations to set flags. But in that case, a destination register is updated and our setting of flags is only a side effect of the operation

In some cases we want an instruction expressly for making a comparison, and SETTING ONLY FLAGS AS THE RESULT

Mnemonic	Name	Description
CMP rs1, operand	Compare	Calculate rs1 – operand and only set flags as result
CMN rs1, operand	Compare Negative	Calculate rs1 – –operand (equals rs1 + operand) and only set flags as result
TST rs1, operand	Test	Bitwise* AND rs1 with operand (same as ANDS but no write-back)
TEQ rs1, operand	Test Equivalence	Bitwise* XOR rs1 with operand (same as EORS but no write-back)

*Architectural Reference Manual gives a poor definition

Boolean BITWISE Operations

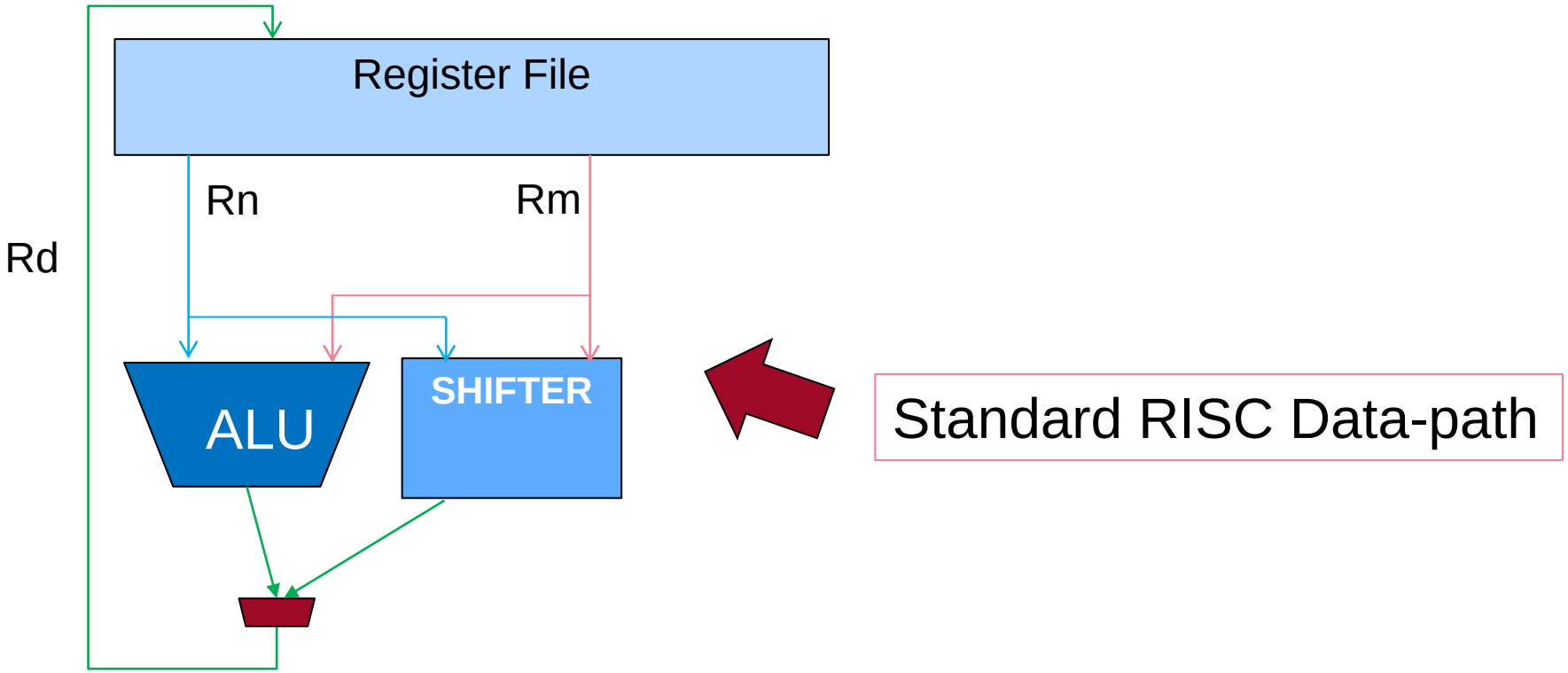
Boolean operations are amongst the simplest instructions implemented by the ARM data-path. Except for MVN, they are based on the standard RRR or RRI format

Mnemonic	Name
AND Rd, Rn, operand	Bitwise AND
ORR Rd, Rn, operand	Bitwise OR
EOR Rd, Rn, operand	Bitwise XOR
(MVN Rd, operand)	Bitwise NOT
BIC Rd, Rn, operand	Bit Clear (AND NOT)

'operand' is derived from a Rm source register or an IMMEDIATE constant

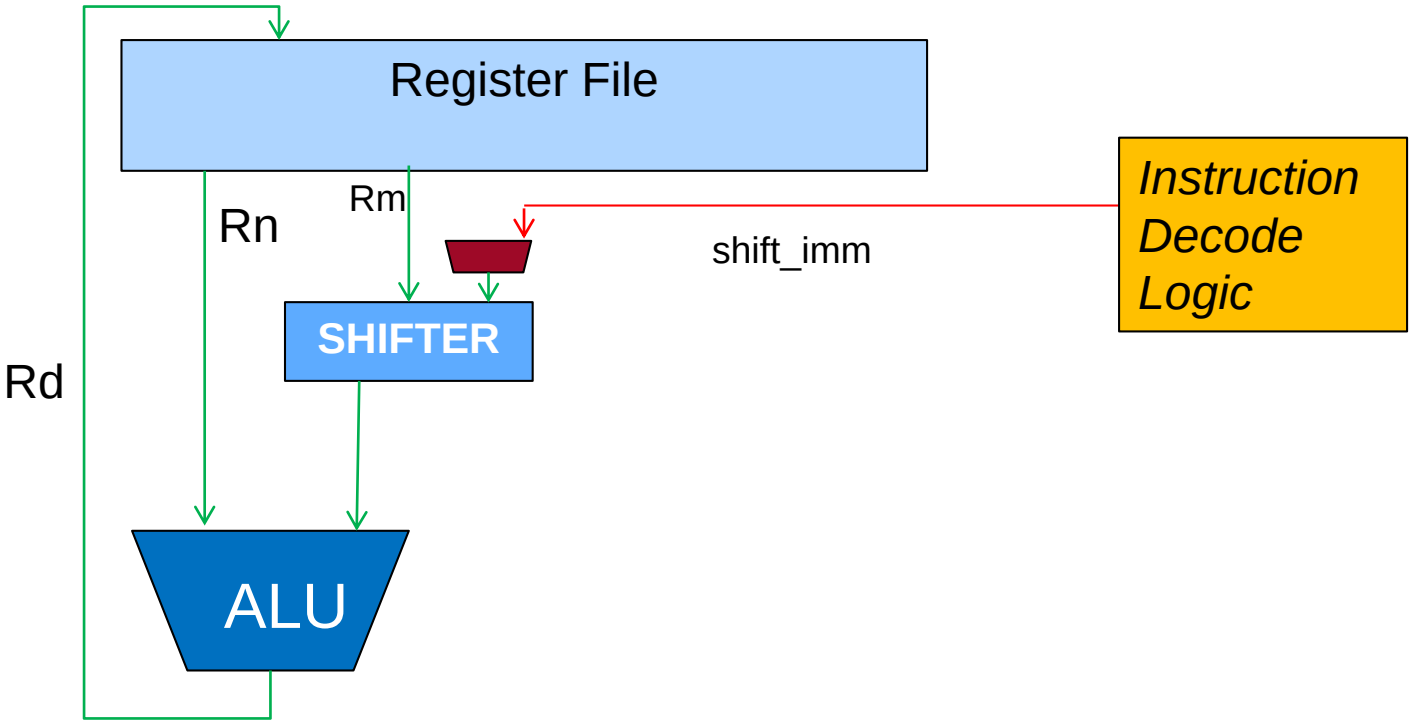
Bit Clear is used to selectively ensure only specific bits in a register are cleared

SHIFT Operations



In standard RISC datapath the shifter is a function unit that can be used “alternatively” to the ALU

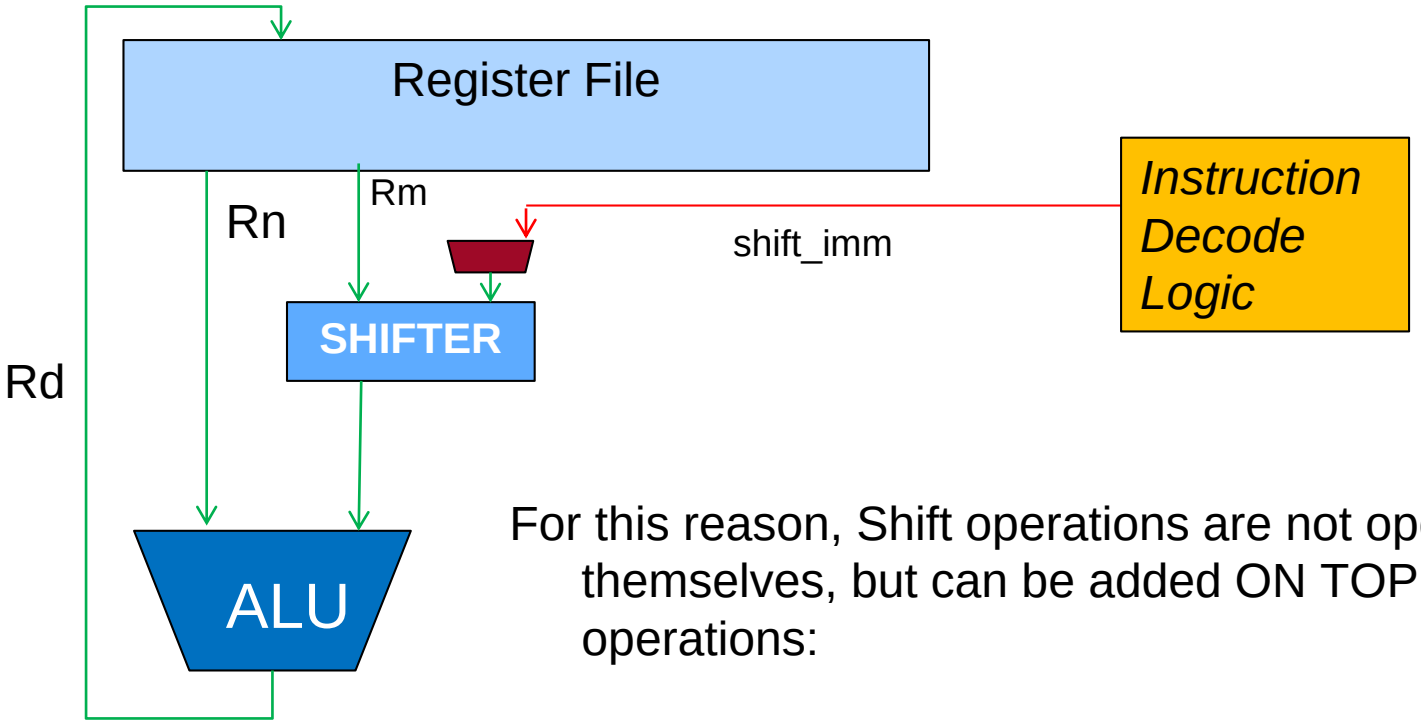
SHIFT Operations



The ARM data-path has a very peculiar unique feature: The shifter is not part of the ALU, or another functional unit parallel to the ALU: it is actually moved to be used BEFORE the ALU.

This would make the processor slower, AND the instruction format more complex, but will allow for much more compact program size

SHIFT OPERATIONS



For this reason, Shift operations are not operations in themselves, but can be added ON TOP OF existing operations:

MOV Rd, Rm, LSL #4 ; Rd=Rm << 4 bits

ADD Rd, Rn, Rm, LSL #8 ; Rd=Rn+(Rm << 8 bits)

SHIFT Operations by Constant

We can implement the following types of shift operations by Constant:

MOV Rd, Rm, LSL #n (Shift Left N – Multiply by 2^N – insert “0” on lower bits)

MOV Rd, Rm, LSR #n (Shift Right N – Divide by 2^N – Insert “0” on higher bits)

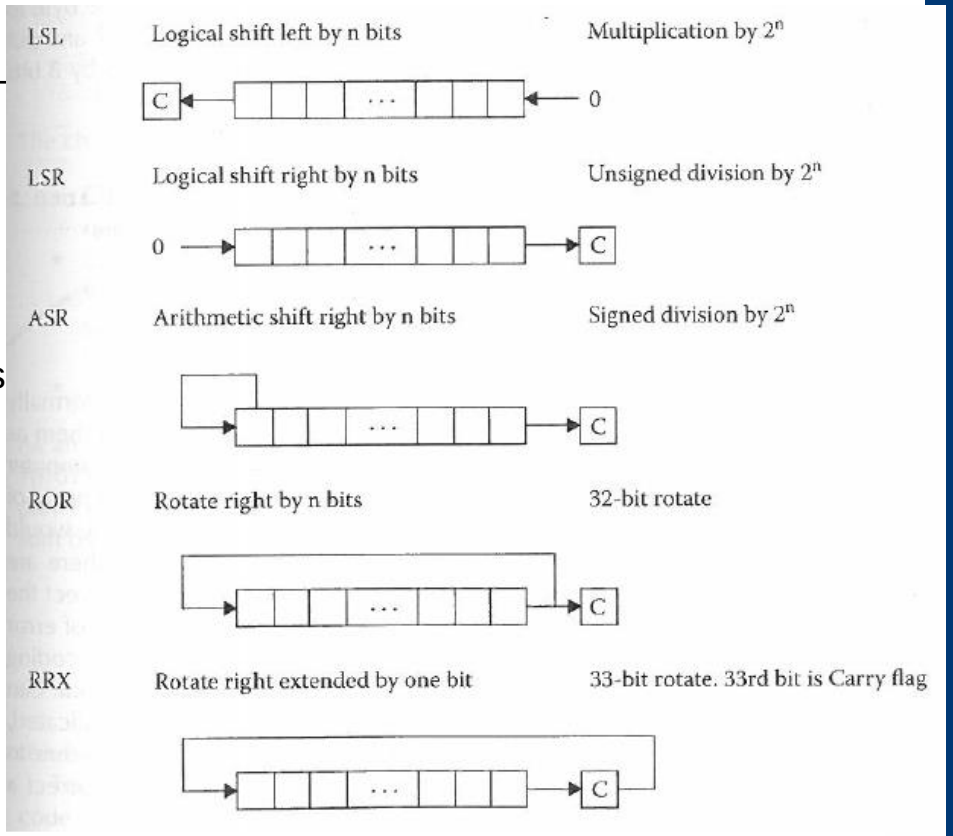
MOV Rd, Rm, ASR #n (Shift Right Arithmetical)

In this last case, the division is signed so in order not to lose the sign bit, the bit 31 is inserted on higher bits

MOV Rd, Rm, ROR #n (Rotate Right N – Bits disappearing on lower bits are inserted on higher bits)

MOV Rd, Rm, RRX (Rotate Right extended by one bit)

Note: Rotate Left (ROL or RLX) is not necessary. ROL can be obtained by rotating right by #(32-N)



All operations above can be performed on Top of the Rm of any Arithmetical/Logical instruction

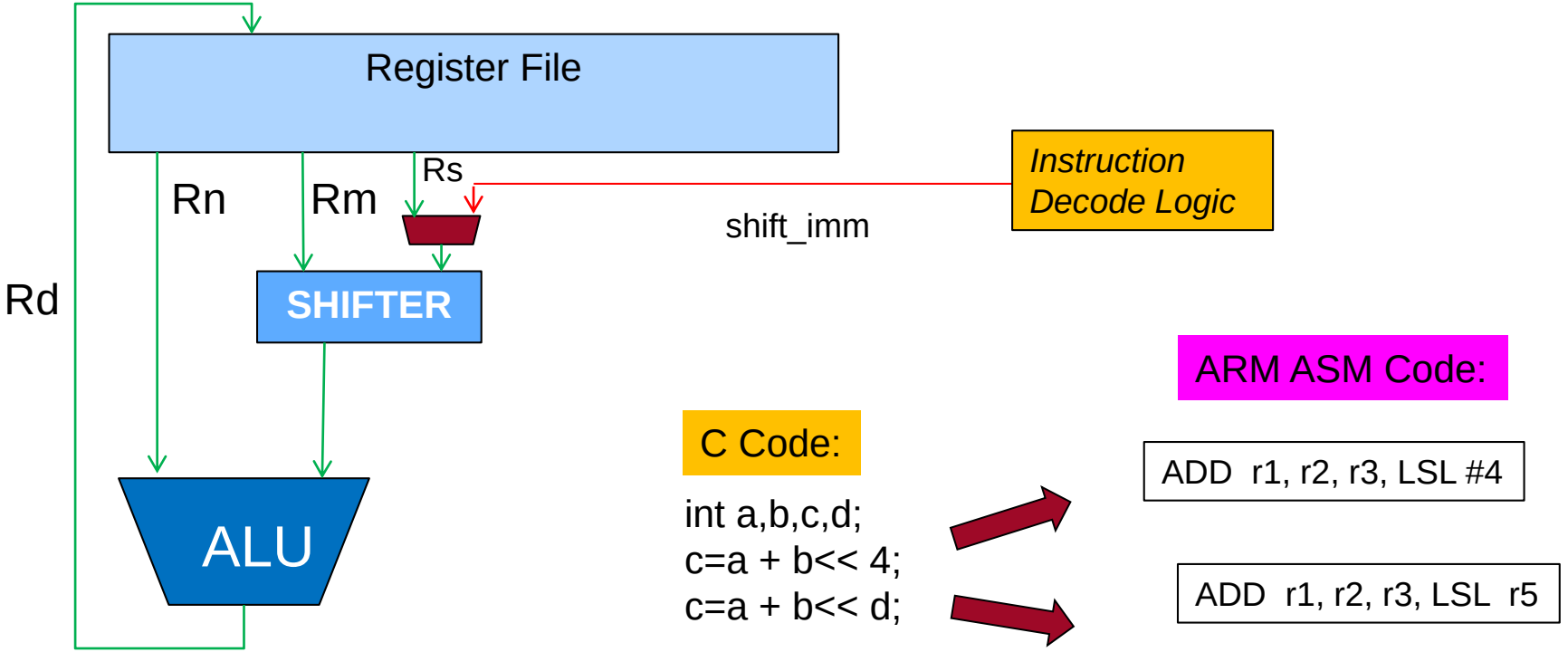
The following instruction produces a single-bit Rotate Left with Extend operation (33-bit rotate through the Carry flag) on Rx:

ADCS Rx, Rx, Rx

Shift Operations by Variable

The shifts described in the previous slide need the programmer to know the shift value. In some case, this may not be possible.

ARM allows one to implement the shift amount as a variable for data-processing instructions. This is not available for scaled register offsets in load/store instructions.



ADDITION / SUBTRACTION

- Addition/subtraction in ARM is implemented much like in any other RISC processor
- They can be performed either on RRR or RRI Format
- Remember that if you need to change the status register you need to force that with a S towards end of the mnemonic

Mnemonic	Name
ADD Rd, Rn, operand	Addition Rd = Rn + operand
SUB Rd, Rn, operand	Subtraction Rd = Rn - operand
RSB Rd, Rn, operand	Reverse Subtraction Rd=operand - Rn

SUB R4, R3, R2
SUB R4, R2, R3

SUB R4, R3, R2, LSL #4
RSB R4, R3, R2, LSL #4

Wake up call

Why in the world would one want to use a RSB instruction ?

SUB Rd=Rn - operand

RSB Rd=operand - Rn

RSB is useful since the two operands are not symmetric: suppose you need to perform

c=8-b;
OR
c=(a<<4)-b;

Since the first operand does not support shift or immediate, we would have to develop a macro and use another instruction as well.

Quiz

Write down a macro that realizes the operation *c = (a<<4) - b;*
Suppose c=r4, a=r2, b=r3

a) In a single line with RSB

a) Without RSB

Quiz

Write down a macro that realize the operation *c=(a<<4)-b;*
Suppose r2 holds a, r3 holds b, and r4 will hold c,

a) In a single line with RSB

RSB r4, r3, r2, LSL #4

b) Without RSB

MOV r4, r2, LSL #4
SUB r4, r4, r3 (probably better)

Or

MOV r2, r2, LSL #4
SUB r4, r2, r3 (maybe not as good)

or

Were You able to solve the quiz?

A. I was able to solve the Quiz
B. I was nearly there
C. I took the wrong route, sorry
D. I did not have a clue
E. I just did not bother

Addition with Carry

We mentioned previously that ARM has a native 32-bit data-path.

Differently from some processor architectures, ARM has a native support for multi-word operations:

Mnemonic	Name
ADCS rd, rn, operand	Addition rd=rn + operand + carry_flag
SBCS rd, rn, operand	Subtraction rd=rn – operand – NOT carry_flag
RSCS rd, rn, operand	Reverse Subtraction rd=operand – rn – NOT carry_flag

Addition / Subtraction with Carry

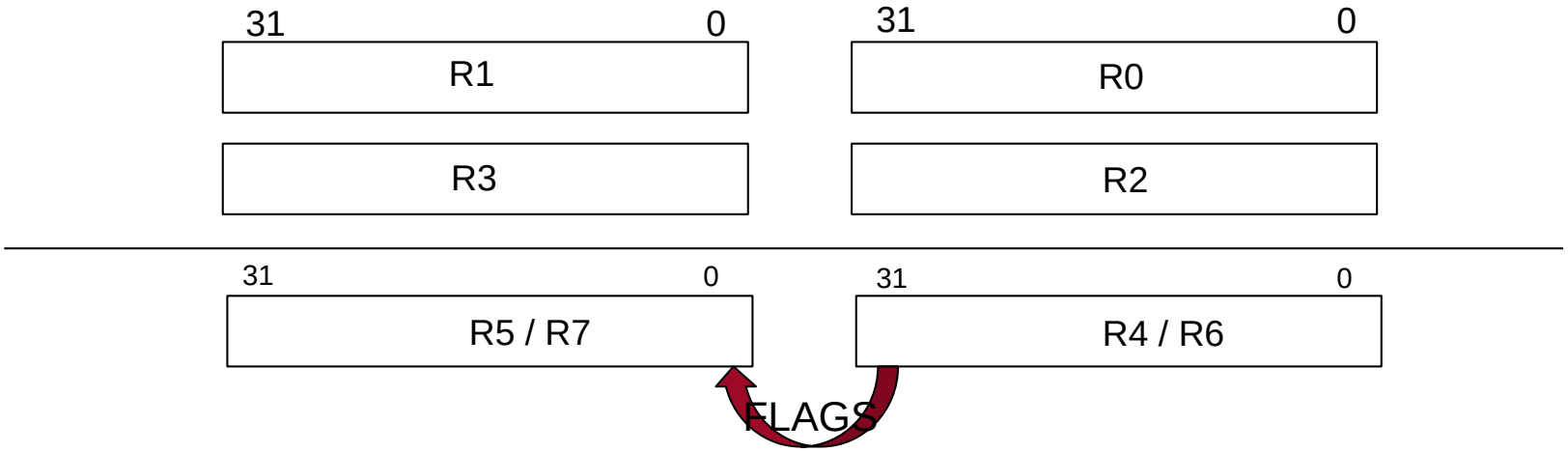
C Code

```
long long int a,b,c,d;
{ c=a+b;
  d=a-b;}
```

We assume the map
a on r0,r1,
b on r2,r3
c on r4,r5
d on r6,r7

ARM Asm

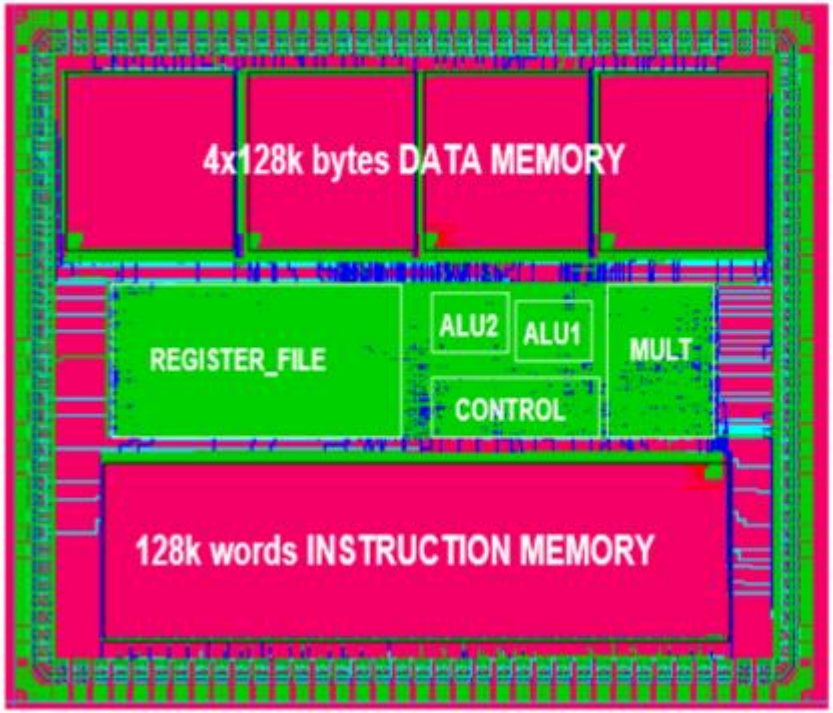
```
ADDS r4,r0,r2
ADC  r5,r1,r3
SUBS r6,r0,r2
SBC  r7,r1,r3
```



Multiplication

- Multiplication is an essential operation for almost any signal processing application: Image and Audio processing, telecommunication, encryption make large use of multiplications

- Hardware multiplication is very, very costly
  - Until the advent of VLSI, Multiplication was based on SW, Now HW multipliers are the rule, but occupy almost half the area of an entire processor! The energy consumption is proportional
  - The use of the multiplier must be carefully weighted and limited whenever possible



Multiplication

- From Boolean algebra we know that a given sum between 2 numbers of N bits can produce a result of N+1 bits
- Multiplying a N bit number by a second N bit number, can produce a 2*N bit number

$$(2^N) * (2^N) = \ 2^{2N}$$

- ***So in the ARM datapath, we have 64-bit results***
- If we don't want to lose information, we will need 2 registers to store the result of a multiplication
- In the ARM ISA, we have the option of choosing a full 64-bit or a reduced 32-bit result

Mnemonic	Name
MUL Rd, Rm, Rs	Rd= bits(31 to 0) of Rm*Rs
(S|U)MULL RdLo,RdHi,Rm,Rs	RdLo = bits(31 to 0) of Rm*Rs RdHi = bits(63 to 32) of Rm*Rs

Please note that the MUL overflow can be much bigger than 1 bit (it is 32-bits), so the V/C Flags are never set. If you use MUL*S* only N and Z will be updated

Multiply-Accumulation

Very many signal processing applications, use widely the formula

$$Acc = \sum_{i=0}^{N-1} a_i * b_i$$

That can be rendered in C with

char a[N], b[N]; int Acc = 0; for(i=0; i<N; i++) {Acc += A[i] * b[i]}

A possible rendition of the code could be on the left. ***Please note, that the loop part will be computed N times***, so that is the part that should be optimized first.

Defining the operation MLA: Rd=Rm*Rs+Rn we can gain a performance boost

```
        Mov r2,#0
        Mov r0,#N
        ;@ load r4 and r6
Loop:
        Ldrb r3,[r4], #1
        Ldrb r5,[r6], #1
        Mul r3,r3,r5
        Add r2,r2,r3
        Subs r0,r0,#1
        Bne Loop
```

```
        Mov r2,#0
        Mov r0,#N
        ;@ load r4 and r6
Loop:
        Ldrb r3,[r4], #1
        Ldrb r5,[r6], #1
        Mla r2,r3,r5,r2
        Subs r0,r0,#1
        Bne Loop
```

Multiply-Accumulation

Similarly to MUL, MLA can be performed on 32-bits losing information, or on 64 with full information:

Note that MLA is richer, as it allows to add a different register than RD. This is not possible in MLAL as we do not have any more space in the 32 instruction-encoding bits to encode more registers

Mnemonic	Name
MLA Rd, Rm, Rs, Rn	Rd= bits(31 to 0) of Rm*Rs + Rn
(S|U)MLAL ... ... RdLo, RdHi, Rm, Rs	RdLo= bits(31 to 0) of Rm*Rs+RdHi<<32+RdLo RdHi=bits(63 to 32) of Rm*Rs+RdHi<<32+RdLo

Note: The MLS (Multiply-Subtract) operation is not available in the version of the ARM ISA we are referring to at the moment, but is present in later versions

Multiplication by Constant

Note that the HW multiplier is a very costly device, and we would be very happy if we could avoid using it.

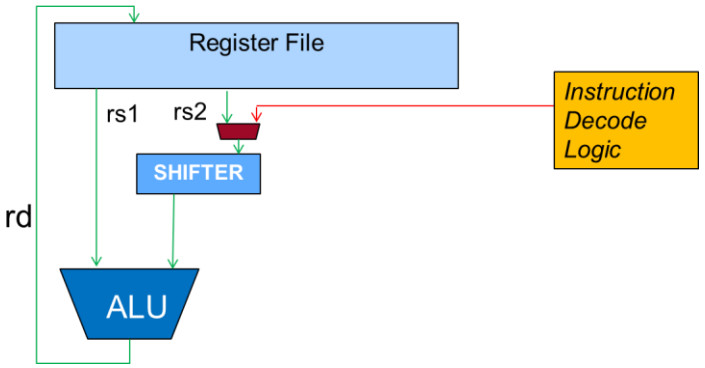
- WHEN THE ASSEMBLER KNOWS IN ADVANCE ONE OPERAND OF THE MULTIPLICATION (that is, we are multiplying by a constant) the operation can be done with less expense using sums and shifts.
  - This is a reason why ARM does not support multiplications with an immediate

Examples:

- b=2*a -> b=a<<1 -> *Mov r0, r1, LSL#1*
- b=(2^N)*a -> b=a<<(N) -> *Mov r0, r1, LSL #N*
- b=5*a -> b= a + a<<2 -> Add r0, r1, r1, LSL #2
- b=-3a -> b= a - a<<2 -> Sub r0, r1, r1, LSL #2
- b=(2^N +/- 1)*a -> b= (a<<N) +/- a -> Add/Rsb r0, r1, r1, LSL #N
- b=-(2^N - 1)*a -> b= a - (a<<N) -> Sub r0, r1, r1, LSL #N

Multiplication by Constant

- Remember that ARM, unlike any other processor, supports shift and Sum in the same instruction, so that all commands in the previous slide can be performed in one single instruction.
- We can multiply in a single instruction by any constant (2^N +/- 1) or -(2^N - 1)
- Combining different sum/shift operations we can describe any other constant:



Examples:

- b=13*a -> b=16*a – 3*a ->
  *Add r0,r1,r1, LSL #1 ; r0=r1+ 2*r1*
  *Rsb r0,r0,r1, LSL#4 ; r0=16*r1-r0*
- b=11*a -> b=7*a + 4*a ->
  *Rsb r0,r1,r1, LSL #3 ; r0= 8*r1 - r1*
  *Add r0,r0,r1, LSL #2 ; r0= 4*r1+ r0*

Note: Please review these arithmetic tricks for quizzes!!!

Division

- Binary division in hardware is very complex and utilizes a lot of area/power, while being used very infrequently
- For this reason it is rarely implemented in processors, and more often it is realized by specific software routines

- There is no division instruction in ARM Assembly
- For the C/C++ developer, the issue is completely transparent, as the compiler will utilize an efficient software routine from Arithmetic libraries

- Of course, if you need to divide by constant 2^N a signed/unsigned number you can use
  - *Mov r0, r1, (A or L)SR #N* (Arithmethic/Logical Shift Right)

- We will look at dividing by other constants later in the course.

Quiz

- Suppose you have a specific algorithm that needs a lot of divisions, so you decide to add a Hardware divider to your system
- Since ARM does not have an assembly operation for division, how can you persuade the processor to use your hardware divider?

C Code

```
z=x/y;
```

ARM asm Code

```
?
```

Quiz

- Suppose you have a specific algorithm that needs a lot of division, so you decide to add a Hardware divider to your system

- Since ARM does not have an assembly operation for division, how can you persuade the processor to use your hardware divider?
- In an ARM system, peripherals are ALWAYS MEMORY MAPPED. If a given Hardware entity is not part of the ISA, ***the only way to access it is to use it as a peripheral***

C Code

We assume the map
x on r0,
y on r1,
z on r2

ARM asm Code

z=x/y;

```
LDR   R4,=base_addr_divider
STR   R0, [r4]
STR   R1, [r4,#4]
;@ possibly do other task...
LDR   R2, [r4,#8]
```