SPRING 2016 CMPE 364

Microprocessor Based Design

Dr. Mohamed Al-Meer

Introduction to the ARM Instruction Set

DATA PROCESSING INSTRUCTION

Lecture Objectives

- Expected to achieve:
  - Know about its Instruction format
  - Know about MOV Instruction
  - Know about Barrel Shifter
  - Know about Arithmetic Instructions
  - Know about Logic Instructions
  - Know about Shift and Rotate Instructions

Introduction

- Different ARM architecture revisions support different instructions. However, new revisions usually add instructions and remain backwardly compatible.
- Will focus on **ARMv5E** core instructions
- Table 3.1 provides a complete list of ARM instructions available in the ARMv5E instruction set architecture (ISA)

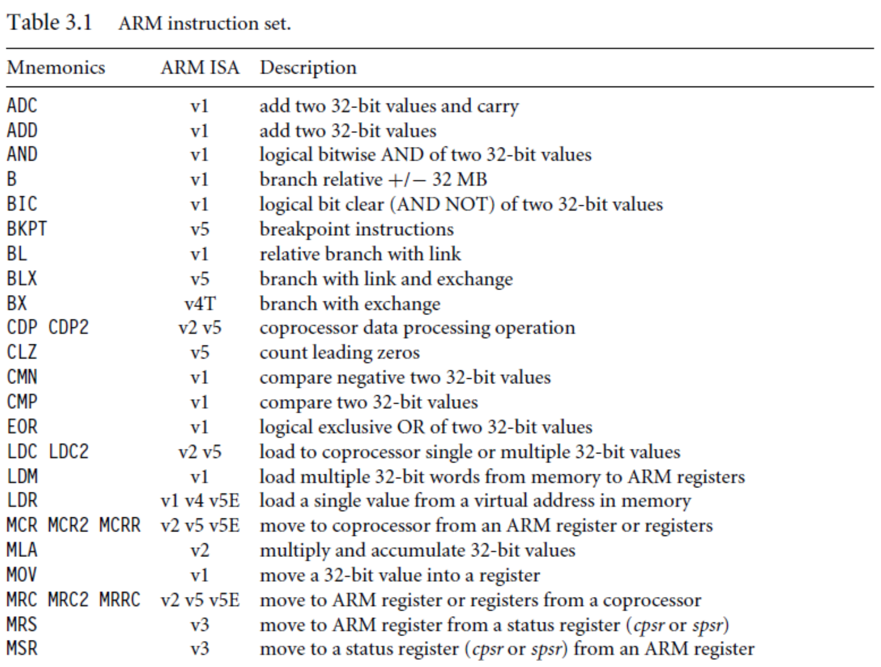
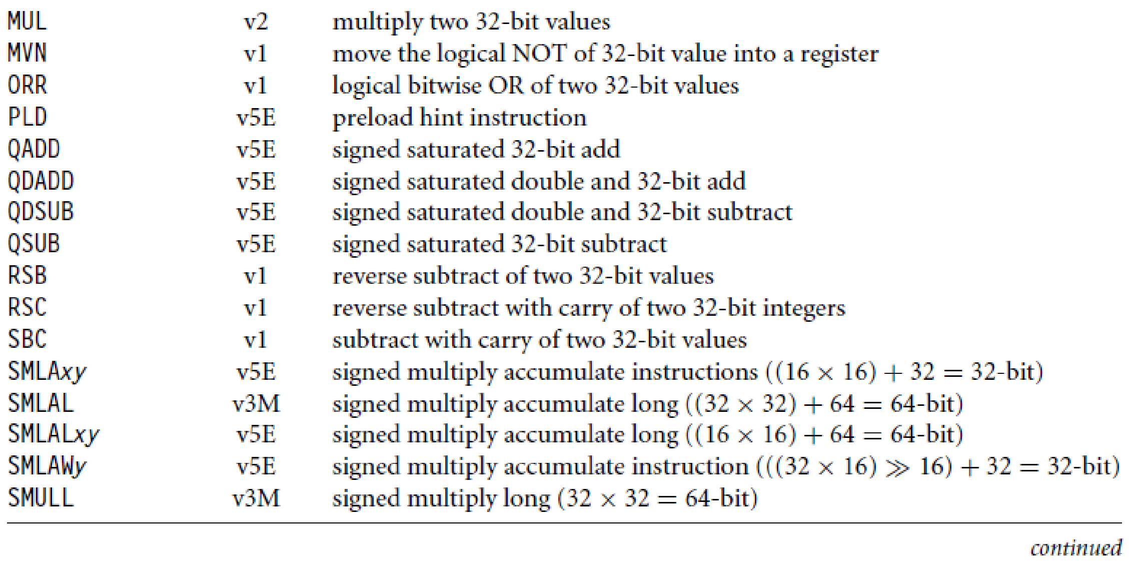


Table 3.1 ARM instruction set.

Mnemonics	ARM ISA	Description
ADC	v1	add two 32-bit values and carry
ADD	v1	add two 32-bit values
AND	v1	logical bitwise AND of two 32-bit values
B	v1	branch relative +/− 32 MB
BIC	v1	logical bit clear (AND NOT) of two 32-bit values
BKPT	v5	breakpoint instructions
BL	v1	relative branch with link
BLX	v5	branch with link and exchange
BX	v4T	branch with exchange
CDP CDP2	v2 v5	coprocessor data processing operation
CLZ	v5	count leading zeros
CMN	v1	compare negative two 32-bit values
CMP	v1	compare two 32-bit values
EOR	v1	logical exclusive OR of two 32-bit values
LDC LDC2	v2 v5	load to coprocessor single or multiple 32-bit values
LDM	v1	load multiple 32-bit words from memory to ARM registers
LDR	v1 v4 v5E	load a single value from a virtual address in memory
MCR MCR2 MCRR	v2 v5 v5E	move to coprocessor from an ARM register or registers
MLA	v2	multiply and accumulate 32-bit values
MOV	v1	move a 32-bit value into a register
MRC MRC2 MRRC	v2 v5 v5E	move to ARM register or registers from a coprocessor
MRS	v3	move to ARM register from a status register (*cpsr* or *spsr*)
MSR	v3	move to a status register (*cpsr* or *spsr*) from an ARM register



MUL	v2	multiply two 32-bit values
MVN	v1	move the logical NOT of 32-bit value into a register
ORR	v1	logical bitwise OR of two 32-bit values
PLD	v5E	preload hint instruction
QADD	v5E	signed saturated 32-bit add
QDADD	v5E	signed saturated double and 32-bit add
QDSUB	v5E	signed saturated double and 32-bit subtract
QSUB	v5E	signed saturated 32-bit subtract
RSB	v1	reverse subtract of two 32-bit values
RSC	v1	reverse subtract with carry of two 32-bit integers
SBC	v1	subtract with carry of two 32-bit values
SMLA*xy*	v5E	signed multiply accumulate instructions $((16 \times 16) + 32 = 32\text{-bit})$
SMLAL	v3M	signed multiply accumulate long $((32 \times 32) + 64 = 64\text{-bit})$
SMLAL*xy*	v5E	signed multiply accumulate long $((16 \times 16) + 64 = 64\text{-bit})$
SMLAW*y*	v5E	signed multiply accumulate instruction $(((32 \times 16) \gg 16) + 32 = 32\text{-bit})$
SMULL	v3M	signed multiply long $(32 \times 32 = 64\text{-bit})$

continued

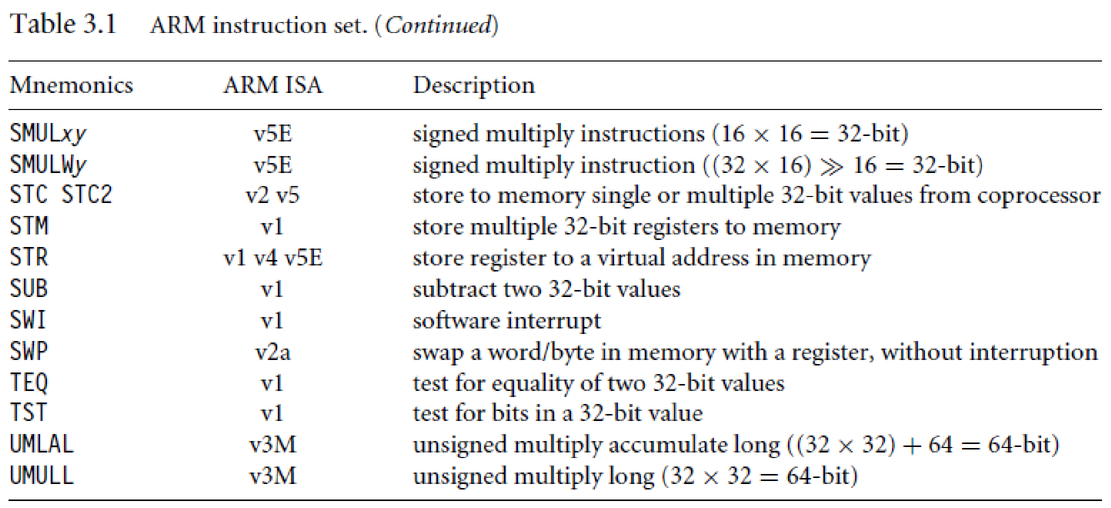


Table 3.1 ARM instruction set. (*Continued*)

Mnemonics	ARM ISA	Description
SMUL*xy*	v5E	signed multiply instructions (16 × 16 = 32-bit)
SMULW*y*	v5E	signed multiply instruction ((32 × 16) ≫ 16 = 32-bit)
STC STC2	v2 v5	store to memory single or multiple 32-bit values from coprocessor
STM	v1	store multiple 32-bit registers to memory
STR	v1 v4 v5E	store register to a virtual address in memory
SUB	v1	subtract two 32-bit values
SWI	v1	software interrupt
SWP	v2a	swap a word/byte in memory with a register, without interruption
TEQ	v1	test for equality of two 32-bit values
TST	v1	test for bits in a 32-bit value
UMLAL	v3M	unsigned multiply accumulate long ((32 × 32) + 64 = 64-bit)
UMULL	v3M	unsigned multiply long (32 × 32 = 64-bit)

Data Processing Instruction Format

- Most Data Processing Instructions follows:

  MNEMONIC DST, SRC1, SRC2
- Mnemonic is a short name for the operation
- Like ADD, SUB, UMUL, EOR
- Destination is first register listed (must GP Register = R0 – R15)
- will represent hexadecimal numbers with the prefix **0x** and binary numbers with the prefix **0b**.
- Memory will denoted as **mem<data_size>[address]**
- Where data size bits of memory starting at the given byte address

Data Processing Instruction Format

- Most data processing instructions can process one of their operands using the **barrel shifter**.
- If you use the **S suffix** on a data processing instruction, then it updates the flags in the **CPSR**.
- Move and logical operations update the carry flag *C*, negative flag *N*, and zero flag *Z.*
- The carry flag is set from the result of the barrel shift as the last bit shifted out.
- The *N* flag is set to bit 31 of the result. The *Z* flag is set if the result is zero.

MOV Instruction

- It copies *N* into a destination register *Rd*, where *N* is a register or immediate value.
- useful for setting initial values and transferring data between registers

MOV DST, SRC

```
Syntax: <instruction>{<cond>}{S} Rd, N
```

`MOV`	Move a 32-bit value into a register	$Rd = N$
`MVN`	move the NOT of the 32-bit value into a register	$Rd = \sim N$

MOV Instruction

- Operand N is usually it is a register *Rm* or a constant preceded by #.
- Example-1
  given the instruction shown below, get R3 and R9 after execution. Assume R3 = 0xFEEA082C and R9 = 0x8000AC40?

  MOV R3, R9

- Solution
  R3 = 0x8000AC40 and R9 remains the same.

MOV Instruction

- Example 2
- Determine R1 after execution on next instruction assuming it contains = 0x2E059401?

  MOV R1, #0x0000009C

- Solution
  R1 = 0x0000009C

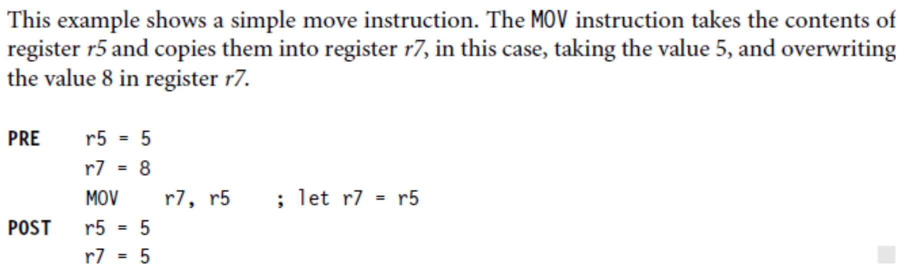
MOV Instruction

- Example 3
- Determine R10 after execution on next instruction assuming it contains = 0x2E059401?

  MOV R10, #384
- Solution

  R10 = 0x00000180

MOV Instruction



This example shows a simple move instruction. The MOV instruction takes the contents of register *r5* and copies them into register *r7*, in this case, taking the value 5, and overwriting the value 8 in register *r7*.

```
PRE    r5 = 5
       r7 = 8
       MOV    r7, r5    ; let r7 = r5
POST   r5 = 5
       r7 = 5
```

Barrel Shifter

- N could be a constant or a register that cab be processed by a shifter before it can affected by any instruction (MOV).
- See next Figure to see how it works.
- A unique and powerful feature of the ARM processor is the ability to shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before it enters the ALU.
- **Pre-processing** or **shift** occurs within the cycle time of the instruction
- useful for **loading constants** into a register and achieving **fast multiplies** or **division** by a power of **2**

Barrel Shifter

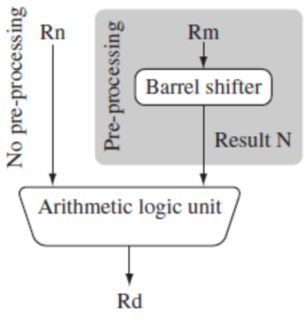
- Example 3

Assume R5 = 5, R7 = 8. what is the value R7 after the execution of next instruction?

MOV R7, R5, LSL #2 ; let R7 = R5*4 = (R5 << 2)

R5 = 5

R7 = 20



Barrel Shifter

- The five different shift operations that you can use within the barrel shifter are summarized in next Table.

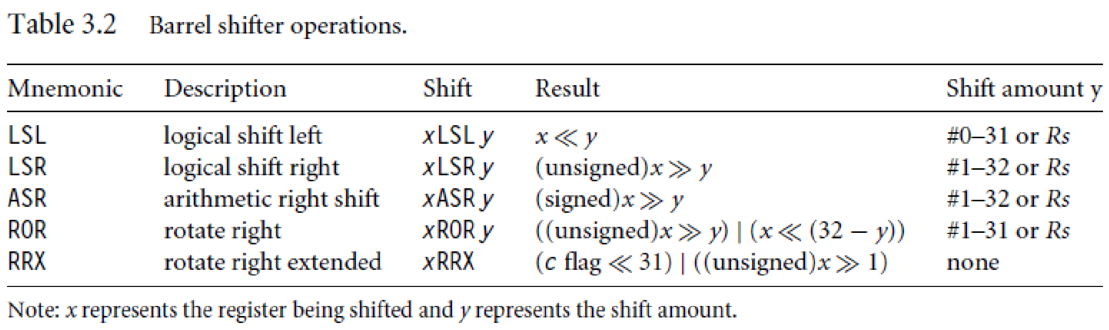


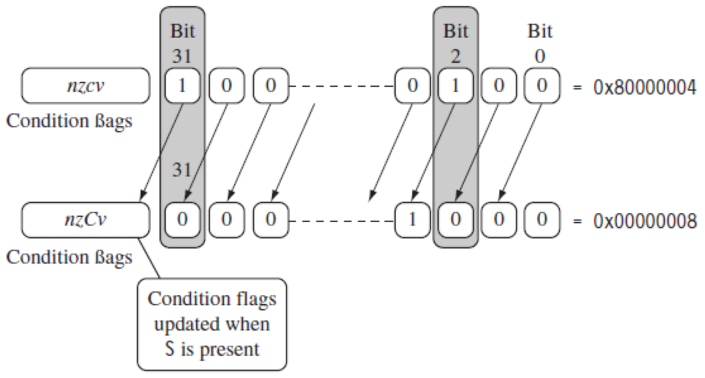
Table 3.2 Barrel shifter operations.

Mnemonic	Description	Shift	Result	Shift amount y
LSL	logical shift left	*x*LSL *y*	$x \ll y$	#0–31 or *Rs*
LSR	logical shift right	*x*LSR *y*	(unsigned)$x \gg y$	#1–32 or *Rs*
ASR	arithmetic right shift	*x*ASR *y*	(signed)$x \gg y$	#1–32 or *Rs*
ROR	rotate right	*x*ROR *y*	((unsigned)$x \gg y$) | ($x \ll (32 - y)$)	#1–31 or *Rs*
RRX	rotate right extended	*x*RRX	(*c* flag $\ll$ 31) | ((unsigned)$x \gg 1$)	none

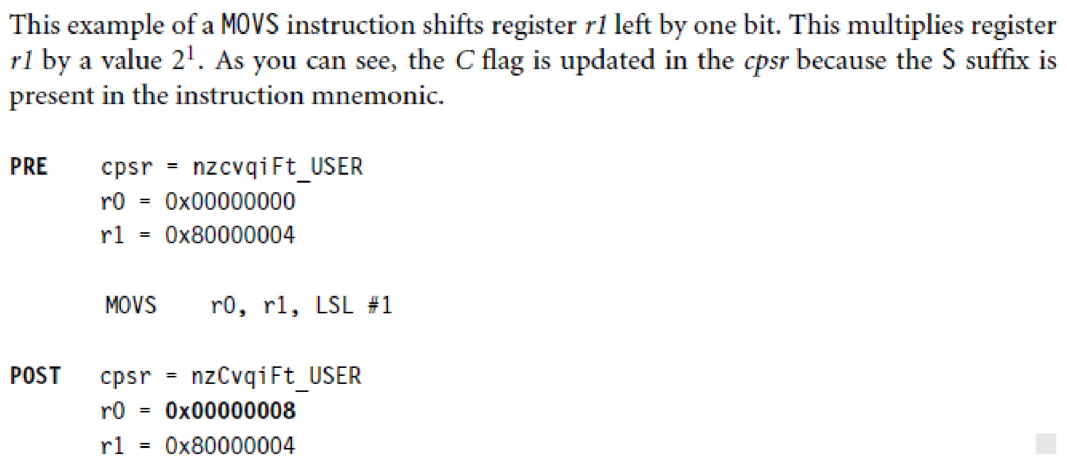
Note: *x* represents the register being shifted and *y* represents the shift amount.

Barrel Shifter

- Next Figure illustrates a logical shift left by one.
- For example, the contents of bit 0 are shifted to bit 1. Bit 0 is cleared. The *C* flag is updated with the last bit shifted out of the register



Barrel Shifter, Another Example with Status



This example of a MOVS instruction shifts register *r1* left by one bit. This multiplies register *r1* by a value 2^1. As you can see, the *C* flag is updated in the *cpsr* because the S suffix is present in the instruction mnemonic.

```
PRE    cpsr = nzcvqiFt_USER
       r0 = 0x00000000
       r1 = 0x80000004

        MOVS    r0, r1, LSL #1

POST   cpsr = nzCvqiFt_USER
       r0 = 0x00000008
       r1 = 0x80000004
```

MVN Instruction

- Move Negative Instruction.
- Has same effect of MOV instruction but copies **the one's complement** of the source to destination.
- Source: general purpose register or immediate

MVN DST, SRC

Example

MVN Instruction

- Example: Determine the content of R2 and R5 after the execution of the following instruction. Assume R2 = 0xA6E9F004, R5 = 0xCE00A824?

  MVN R2, R5

- Solution:

  R2 = 0x31FF57DB, R5 = SAME.

MVN Instruction

- Example: Determine the content of R4 after the execution of the following instruction?

  MVN R4, #24

- Solution:

BEFORE: #24 = $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001\ 1000_2$.

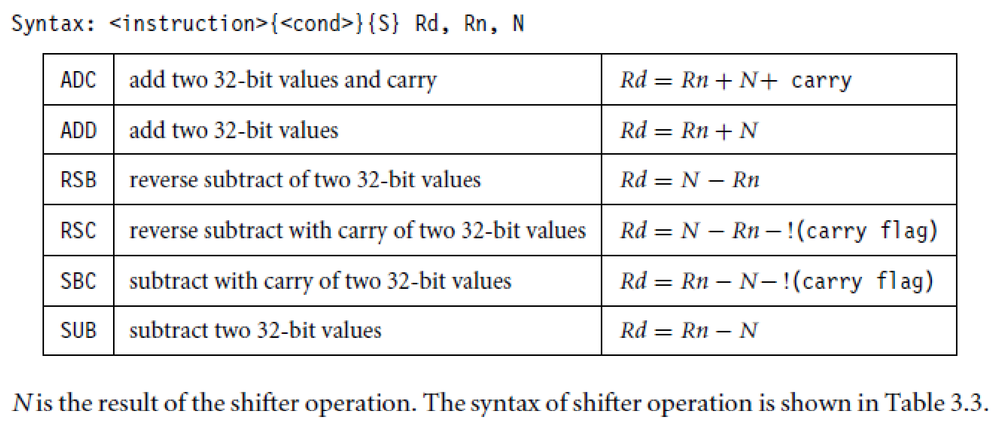
#24 = **0x00000018**.

AFTER: R4 = $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0111_2$.

R4 = **0xFFFFFFE7**.

Arithmetic Instructions

- The arithmetic instructions implement addition and subtraction of 32-bit signed and unsigned values.



Syntax: `<instruction>{<cond>}{S} Rd, Rn, N`

ADC	add two 32-bit values and carry	$Rd = Rn + N +$ `carry`
ADD	add two 32-bit values	$Rd = Rn + N$
RSB	reverse subtract of two 32-bit values	$Rd = N - Rn$
RSC	reverse subtract with carry of two 32-bit values	$Rd = N - Rn -$ `!(carry flag)`
SBC	subtract with carry of two 32-bit values	$Rd = Rn - N -$ `!(carry flag)`
SUB	subtract two 32-bit values	$Rd = Rn - N$

N is the result of the shifter operation. The syntax of shifter operation is shown in Table 3.3.

ADD Instruction

- ADD instruction adds 2 source operands SRC1 and SRC2, and stores the result in destination register DST.
- The first operand must be general purpose register, while the other can be a register or an immediate operand.

ADD DST, SRC1, SRC2

ADD Examples

- Example:

  Get R0, R3, and R9 after the next instruction, if R0 =0x00014009, and R3 = 0x00326018?

  ADD R9, R0, R3

Answer:

R9 = **0x0033A021**

R0 and R3 No Change

ADD Examples

- Example:

  Get R1and R6 after the next instruction, IF:

  R6 = 0xFFFFFFFE (-2?)

  ADD R1, R6, #24

Answer:

R1 = **0x00000016 = 22**.

SUB Instruction

- The SUB instruction subtracts the 2nd Source from the 1st Operand and replaces the result in destination register.

  SUB DST, SRC1, SRC2

- EXAMPLE: Get R4 if R2 = 0x000006A0, R1 = 0x000003C4

  SUB R4, R2, R1

- SOLUTION: R4 = **0x000002DC**

SUB Instruction

- EXAMPLE: Get R4 if R2 = 0x000008E0?

  SUB R4, R2, #0xFFFFFFFE (-1)

- SOLUTION:

  R4 = 0x000008E0 – 0xFFFFFFFE = **0x000008E2**

RSB Instruction

- RSB is Reverse Subtract Instruction.
- Subtracts SRC1 from SRC2

RSB DST, SRC1, SRC2

- EXAMPLE: Show how the subtracts occur? If R3 = 0x000006BE, R9 = 0x000009AC?

RSB R8, R3, R9

- SOLUTION: R8 = R9 – R3 = **0x000002EE**

RSB Instruction

- EXAMPLE:
  - Write Assembly instruction to subtract R7 from 1000 and replace result in R7?
- SOLUTION:

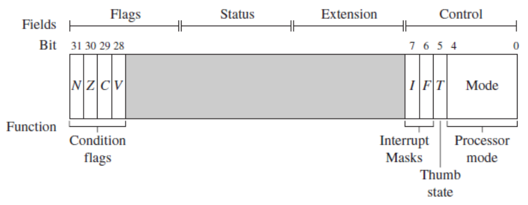
RSB R7, R7, #1000

- EXAMPLE
  - Write a single code to convert the sign of R1 register without knowing its content?
- SOLUTION:

RSB R1, R1, #0

ADC Instruction: Add with Carry

- Add with Carry bit



ADC DST, SRC1, SRC2

- Adds the SRC1 with SRC2 with Carry bit in CPSR register and places result in DST.
- SRC1 should be a register, while SRC2 can be a register or a immediate operand.

ADC Instruction: Add with Carry

- EXAMPLE: Get R6 after the execute of next instruction if R11 = 0x000E5FA9and R10 = 0x00005204 and if CF = 1?

ADC R6, R11, R10

- SOLUTION: R11 = R10 + R11 + C = **0x000EB1AE**

ADC Instruction: Add with Carry

- EXAMPLE: Get R4 after the execution of next instruction, if R2 = 0x80040608 and **CPSR** = 0x**50000010**?

```
ADC R4, R2, #134      ;(0x86)
```

- SOLUTION:R4 = R2 + 134 + **0** = **0x8004068E**

SBC Instruction

- SBC is subtract with carry.
- Subtracts the **SRC2** and **complement of carry bit** in CPSR register **from** SRC1 and places the result in DST
- DST = SRC1 – SRC2 – Not(C)

```
SBC DST, SRC1, SRC2
```

- EXAMPLE: Get R4 if R2 = 0x000006A0 and R1 = 0x000003C4, and CPSR = 0x00000010?

```
SBC R4, R2, R1
```

SOLUTION: 000006A0 – 000003C4 – 1 = **000002DB**

RSC Instruction

- **Reverse Subtract with Carry**
- Subtracts **SRC1** And **complement** of **Carry from SRC2** and places the result in **DST** register.
- DST = SRC2 – SRC1 – Not C

  RSC DST, SRC1, SRC2
- EXAMPLE: Get R3, if R0,and R2and CPSR = 0x08009420, 0x014520C0, and 0x0000010.
- SOLUTION: ????

Using the Barrel Shifter with Arithmetic Instructions

- As known The wide range of second operand shifts is very powerful feature of the ARM instruction set.
- The next example is a use of the inline barrel shifter with an arithmetic instruction.
- All conditions regarding barrel shifter implemented on MOV instruction applies here as well.

Example

Register *r1* is first shifted one location to the left to give the value of twice *r1*. The ADD instruction then adds the result of the barrel shift operation to register *r1*. The final result transferred into register *r0* is equal to three times the value stored in register *r1*.

```
PRE     r0 = 0x00000000
        r1 = 0x00000005

        ADD     r0, r1, r1, LSL #1

POST    r0 = 0x0000000f
        r1 = 0x00000005
```

Logical Instructions

- Logical instructions perform bitwise logical operations on the two source registers.
- Allow individual bit manipulation: at bit level

`Syntax: <instruction>{<cond>}{S} Rd, Rn, N`

AND	logical bitwise AND of two 32-bit values	$Rd = Rn \& N$
ORR	logical bitwise OR of two 32-bit values	$Rd = Rn \mid N$
EOR	logical exclusive OR of two 32-bit values	$Rd = Rn \wedge N$
BIC	logical bit clear (AND NOT)	$Rd = Rn \& \sim N$

AND Instruction

- Performs bitwise AND of two source operands, SRC1 and SRC2, and stores the result in DST.

  AND DST, SRC1, SRC2
- SRC1 is a GP register, while SRC2 could be GP register or an immediate operand. DS must be a GP register.
- EXAMPLE: R1=0x1A89F045, R12=0x96231E8D

  AND R4, R1, R12

Get R4 = ?, show bit by bit solution

AND Instruction

- EXAMPLE: write an AND instruction to clear every bit except 16, 17, and 19 of register R4?
- SOLUTION:

  mask = **0x000B0000**

  = 0000 0000 0000 **1011** 0000 0000 0000 0000

  AND R4, R4, 0x000B0000

Show in bit by bit.

ORR Instruction

- Performs bitwise Inclusive OR of SRC1 and SRC2 and places the result in DST.
- SRC1 is a register while SRC2 could be a register or an immediate operand.

  ORR DST, SRC1, SRC2

- EXAMPLE: R6 = 0xEF486EE9, R1=0x80182ACF
- SOLUTION: ??? (**0xEF586EEF**)

ORR Instruction

- EXAMPLE

Set bit positions 7 and 9 in register R0 to 1 without touching other 30 bits in the same register?

- SOLUTION:

Will use an ORR instruction, Oring R1 register with a 32-bit Mask.

Mask will be = 0x --------?

ORR R0, R0, #0x00000280

EOR (Exclusive OR) Instruction

- Exclusive OR which performs Exclusive OR of 2 sources, SRC1 And SRC2, and places result in DST.

  EOR DST, SRC1, SRC2

- SRC1: GP register, SRC2: GP register or immediate operand, DST must be GP register.
- Very useful instruction for **comparison** to see which bit position they differ. Also used in **Encryption and Decryption with Keys** (**our coming Project ??**)

EOR Instruction

- EXAMPLE: Get R9? if R5 = 0x7C5D6B21, R8 = 0x9624EC30?

  EOR R9, R5, R8
- ANSWER:

  R9 = 0xEA798711.
- EXAMPLE: **Invert bit 30** of R0 using a single instruction?
- SOLUTION:

  MASK = **0100** 0000 0000 0000 0000 0000 0000 0000 =
  = 0x40000000

  EOR R0, R0, 0x40000000

BIC (Bit Clear) Instruction

- Bit Clear Instruction clears 1 bit or more in the first source register SRC1, and stores the result in DST register
- The **bit positions** needed for c**le**ar should be placed in **SRC2** register by using bit-1 and bit-0 means do not invert

BIC DST, SRC1, SRC2

SRC1 AND !SRC2

BIC (Bit Clear) Instruction

- Example: using a single instruction clear bit-5 I register R0?
- Solution:

Mask = **0000 0000 0000 0000 0000 0000 0010 0000** =

= **0x20**

BIC R0, R0, #0x20

Shift & Rotation

- Moves bits to left or right in a register

Logical Shifting

  - Shifting in 0's
  - Logical Left shift and Logical Right shift

- Logical Shift Left
  - All bits shifted to left
  - 0 is fed to b0
  - **B31 will be copied to C flag**.

Logical Shift Left

- Example

  Shift the 32-bit value 0x89C21E46 **5 positions to left**?

- Solution

Before	1000 1001 1100 0010 0001 1110 0100 0110
After	0011 1000 0100 0011 1100 1000 110**0 0000**
	0x3843C4C0

Logical Shifting a number to left 1 position is equivalent to multiply by 2.

Logical Shift Left

- EXAMPLE: show logical left shift of 180,156 six positions is equal to multiply by 64?
- ANSWER:

180,156 = $\underline{000000}00000000101011111110111100$ x 64 =

= $00000000101011111110111100\underline{000000}$

= 11,529,984

Logical Shift Right

- Shift all bits to the right.
  - All bits shifted to right
  - 0 is fed to b31
  - **b0 will be copied to C flag.**
- EXAMPLE: logically shift 32-bit the value 0x89C21E46 fourteen positions?
- SOLUTION:

1000 1001 1100 0010 00$\underline{01\ 1110\ 0100\ 0110}$

0000 0000 0000 0010 0010 0111 0000 1000

Logical Shift Right

- Logical Shifting a number to right1 position is equivalent to divide by 2.
- EXAMPLE: use logical shift right to divide unsigned number (0x00008105) BY 256?
- SOLUTION:

  0000 0000 0000 0000 1000 0001 0000 0101

  0000 0000 0000 0000 0000 0000 1000 0001

Arithmetic Shifting

- A drawback of logical shifting is that signed numbers cannot be taken care of (results may lead to incorrect sign bit).
- Arithmetic shifting will consider the sign bit after the shifting
- **Arithmetic Right Shift preserves the sign bit**
- EXAMPLE: perform ASR (arithmetic shift right) on 32-bit number value 0x8621A93C **19 positions**?
- SOLUTION:

  1000 0110 0010 0001 1010 1001 0011 1100

  1111 1111 1111 1111 1111 0000 1100 0100

  = 0xFFFFF0C4

Rotation

- Rotation is similar to shifting bit differ in the sense **that bits shifted out of one side enters into the other end**.
- Rotate Left

  Moving all the bits to the left. It is rotated around at the ends.
- EXAMPLE: rotate **6 position to the left** for the value 0xA7250149?
- SOLUTION

  1010 0111 0010 0101 0000 0001 0100 1001

  1100 1001 0100 0000 0101 0010 0110 1001

Rotation

- EXAMPLE: rotate 0xE79212DB 12 positions to the right?
- SOLUTION:

  1110 0111 1001 0010 0001 0010 1101 1011

  0010 1101 1011 1110 0111 1001 0010 0001

  = 0x2DBE7921