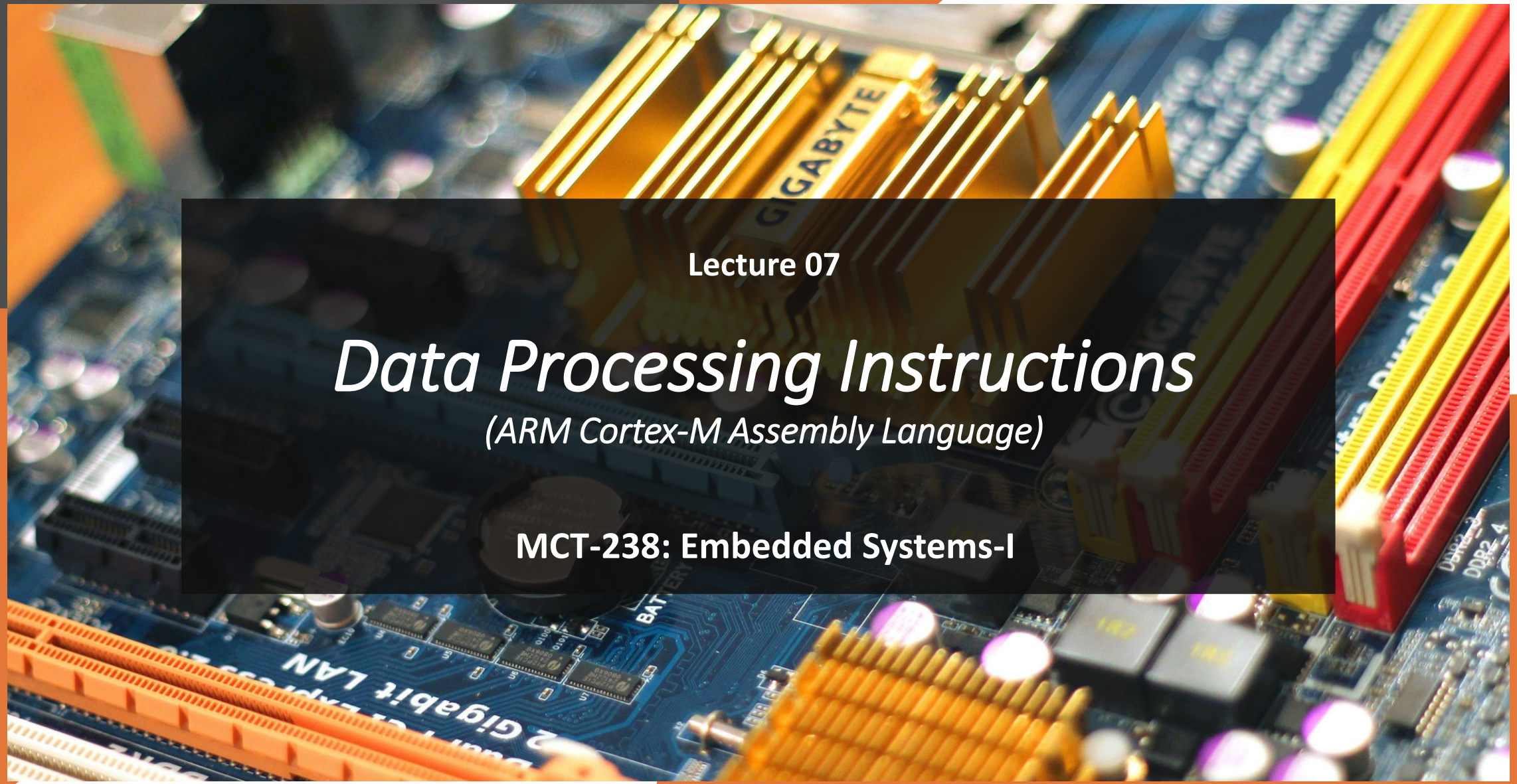
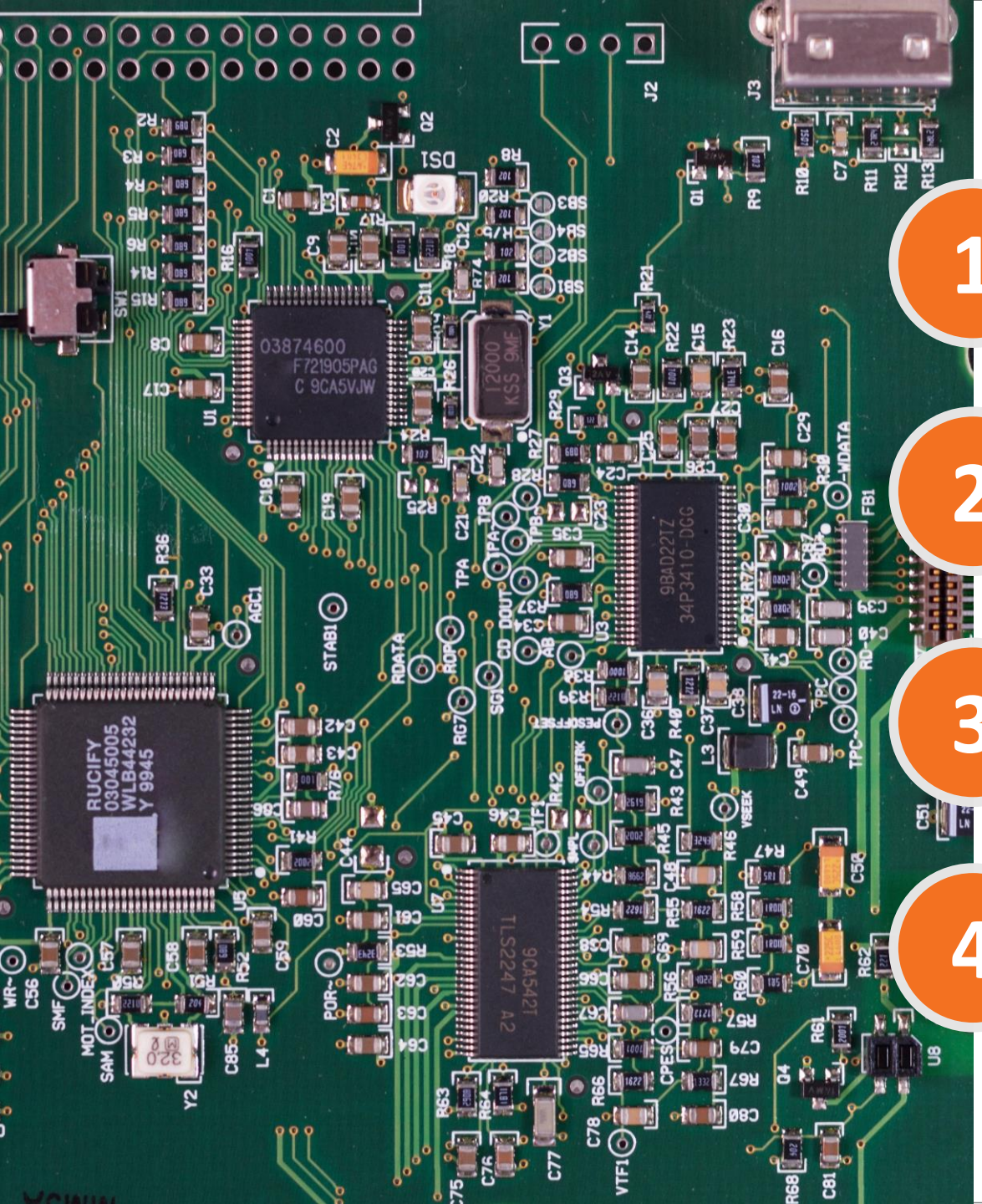


Lecture 07

Data Processing Instructions

(ARM Cortex-M Assembly Language)

MCT-238: Embedded Systems-I



1. SHIFT AND ROTATE INSTRUCTIONS
2. LOGICAL INSTRUCTIONS
3. ARITHMETIC INSTRUCTIONS
4. DATA MOVEMENT INSTRUCTIONS

SHIFT AND ROTATE INSTRUCTIONS

ARM Cortex-M4 Assembly Language

Shift and Rotate Instructions

- Shift operations move the bits in a register left or right by a specified number of bits, termed the **shift length**.
- Register shift operation can be performed directly by using the appropriate shift instructions and the result is written to a destination register.
- Another usage of shift operation is by some of the assembly instructions during the evaluation of their second operand in the form of a register with shift.
- The shift lengths that are permitted depend not only on the type of shift operation but also on the specific instruction being used.
- When the specified shift length is 0 then no shift occurs.
- Register shift operations can optionally update the flag(s)
- In case of rotate instruction, the bits in a register are circulated in the specified direction by the given number of bit locations.

Instruction Encoding and Flags

SHIFT AND ROTATE INSTRUCTIONS

Mnemonic	Brief description	Encoding	Flags
ASR	Arithmetic shift right	16 or 32 bit	N,Z,C
LSR	Logical shift right	16 or 32 bit	N,Z,C
LSL	Logical shift left	16 or 32 bit	N,Z,C
ROR	Rotate right	16 or 32 bit	N,Z,C
RRX	Rotate right with extend	32 bit	N,Z,C

Instruction Encoding and Flags

SHIFT AND ROTATE INSTRUCTIONS

$$
\begin{array}{llll}
\text{LSR}\{\text{S}\}\{cond\}\{.size\} & \{Rd,\} & Rm, & Rs \text{ or } \#n \\
\text{ASR}\{\text{S}\}\{cond\}\{.size\} & \{Rd,\} & Rm, & Rs \text{ or } \#n \\
\text{LSL}\{\text{S}\}\{cond\}\{.size\} & \{Rd,\} & Rm, & Rs \text{ or } \#n \\
\text{ROR}\{\text{S}\}\{cond\}\{.size\} & \{Rd,\} & Rm, & Rs \text{ or } \#n \\
\text{RRX}\{\text{S}\}\{cond\} & \{Rd,\} & Rm &
\end{array}
$$

- **Rd** Specifies the optional destination register.
- **Rm** Specifies the register containing the value to be shifted.
- **Rs** Specifies the register holding the shift length to apply to the value in Rm. Only the least significant byte is used.
  - RRX moves the bits in register Rm to the right by 1 bit, therefor we don't specify Rs for RRX instruction
- **n** Specifies the shift length using an immediate value. The range of shift length depends on the instruction.

Instruction Encoding and Flags

SHIFT AND ROTATE INSTRUCTIONS

$$
\begin{array}{llll}
\text{LSR}\{\text{S}\}\{cond\}\{.size\} & \{Rd,\} & Rm, & Rs \text{ or } \#n \\
\text{ASR}\{\text{S}\}\{cond\}\{.size\} & \{Rd,\} & Rm, & Rs \text{ or } \#n \\
\text{LSL}\{\text{S}\}\{cond\}\{.size\} & \{Rd,\} & Rm, & Rs \text{ or } \#n \\
\text{ROR}\{\text{S}\}\{cond\}\{.size\} & \{Rd,\} & Rm, & Rs \text{ or } \#n \\
\text{RRX}\{\text{S}\}\{cond\} & \{Rd,\} & Rm &
\end{array}
$$

- **{}** Represents the optional field.
- If **S**, optional suffix, is specified, the condition code flags of APSR are updated on the result of the operation. Only N, Z, and C flags are updated by these instructions.
- ***cond*** If this optional condition code suffix is specified, the instruction will conditionally executed based on the condition flags set by the preceding instruction(s)
- ***.size*** The optional suffix to enforce 16-bit (**.N**) or 32-bit (**.W**) instruction size encoding and is formerly called instruction size qualifier.
  - If the instruction size qualifier is not specified, the assembler or complier chooses appropriate instruction size which is more suitable option

Application Program Status Register

ARM Assembly Language

- ARM v6/v7 maintains a APSR register that holds five status bits, **negative** (N), **zero** (Z), **carry** (C), **overflow** (V), and **saturation** (Q)
- APSR set automatically according to most recently executed ALU Data Processing Instruction that includes “S” suffix. e.g., ADDS will modify status bits but ADD will not
- The top four bits of the APSR hold the following useful information about the results of that arithmetic/logical operation:
  - The negative (N) bit is set when the result is negative in two's-complement arithmetic
  - The zero (Z) bit is set when every bit of the result is zero
  - The carry (C) bit is set when there is a carry out of the operation
  - The overflow (V) bit is set when an arithmetic operation results in an overflow
  - The saturation (Q) bit is set when a saturation in the value of register happens
- APSR bits must be checked right after desired statement execution because the next operation may change its values

Condition Code Suffixes

ARM Assembly Language

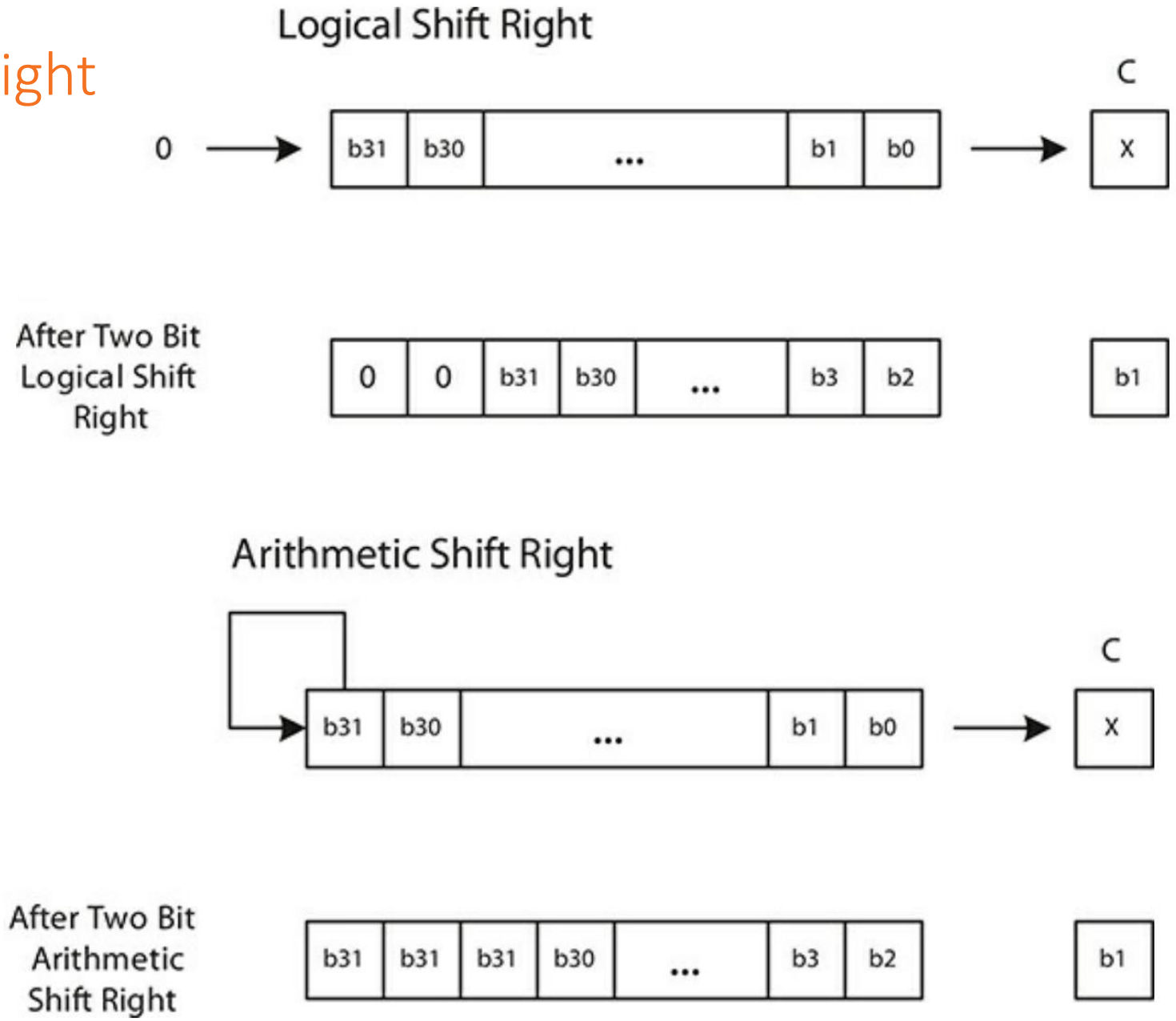
- Nearly, all ARM instructions can include an optional condition code that determines if the instruction will be executed or skipped over
- In other words, an instruction whose condition code is evaluated to false will not change the state of the processor
- For example, the **LSREQ** (where **LSR** is the instruction and **EQ** is condition code suffix for equality) instruction will only execute if the **Z**-bit in the CPSR is set as a result of most recent computational instruction
- When combining the condition code and the "S" suffix, the condition code comes first.

Suffix	Flags	Meaning
EQ	**Z** set	Equal
NE	**Z** clear	Not equal
CS or **HS**	**C** set	Unsigned >=
CC or **LO**	**C** clear	Unsigned <
MI	**N** set	Negative
PL	**N** clear	Positive or zero
VS	**V** set	Overflow
VC	**V** clear	No overflow
HI	**C** set and **Z** clear	Unsigned >
LS	**C** clear or **Z** set	Unsigned <=
GE	**N** and **V** the same	Signed >=
LT	**N** and **V** differ	Signed <
GT	**Z** clear, **N** and **V** the same	Signed >
LE	**Z** set, **N** and **V** differ	Signed <=

Logical and Arithmetic Shift Right

SHIFT AND ROTATE INSTRUCTIONS

- When LSRS and ASRS instructions shift the register position by multiple bits, the value of the carry flag C will be the last bit that shifts out of the register from right side
- The range of shift length is 1 to 32 for this instruction



Logical and Arithmetic Shift Right

SHIFT AND ROTATE INSTRUCTIONS

```
LSR     R4, R3, #3          ; R4 = R3 >> 3
ASR     R5, R3, #3          ; R5 = R3 >> 3
```

For Unsigned Number:

- Consider R3 = 64 (0x0000 0040), **above instructions will produce same result i.e.,** R4 = R5 = 8 (0x0000 0008)
- **But for** R3 = 2,147,483,648 (0x8000 0000), **above instructions will produce different results i.e.** 0x1000 0000 = 268,435,456**, while** R5 = 0xF000 0000 = 4,026,531,840

- We can conclude that for unsigned values the LSR **gives the correct result, however,** ASR **doesn't work properly**

Logical and Arithmetic Shift Right

SHIFT AND ROTATE INSTRUCTIONS

```
LSR     R4, R3, #3          ; R4 = R3 >> 3
ASR     R5, R3, #3          ; R5 = R3 >> 3
```

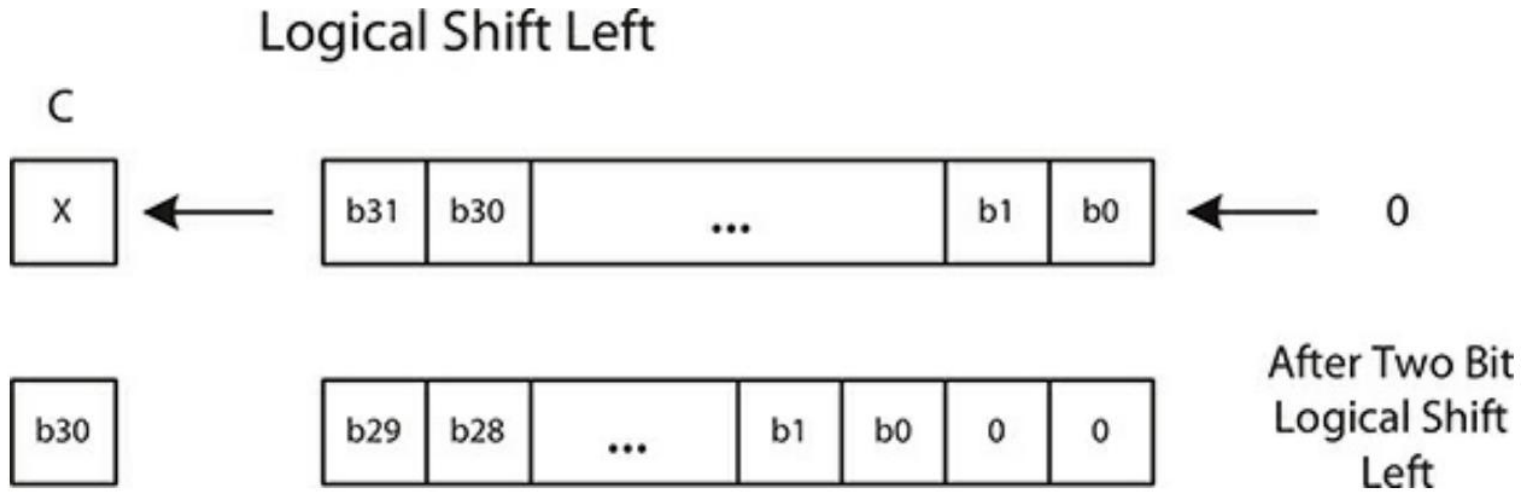
For Signed Number:

- Consider R3 = -64 (0xFFFF FFC0) **performing the** LSR **and** ASR **operations by 3 bits leads to** R4 = 0x1FFFFFF8 = 536870904, **while** R5 = 0xFFFFFFF8 = -8

- We can conclude that for signed values the ASR gives the correct result, however, LSR doesn't work properly
- Therefore, when dealing with unsigned numbers, LSR must be used and for signed numbers ASR will be used to get correct results
- Shift right by one bit position leads to division by 2

Logical Shift Left

SHIFT AND ROTATE INSTRUCTIONS



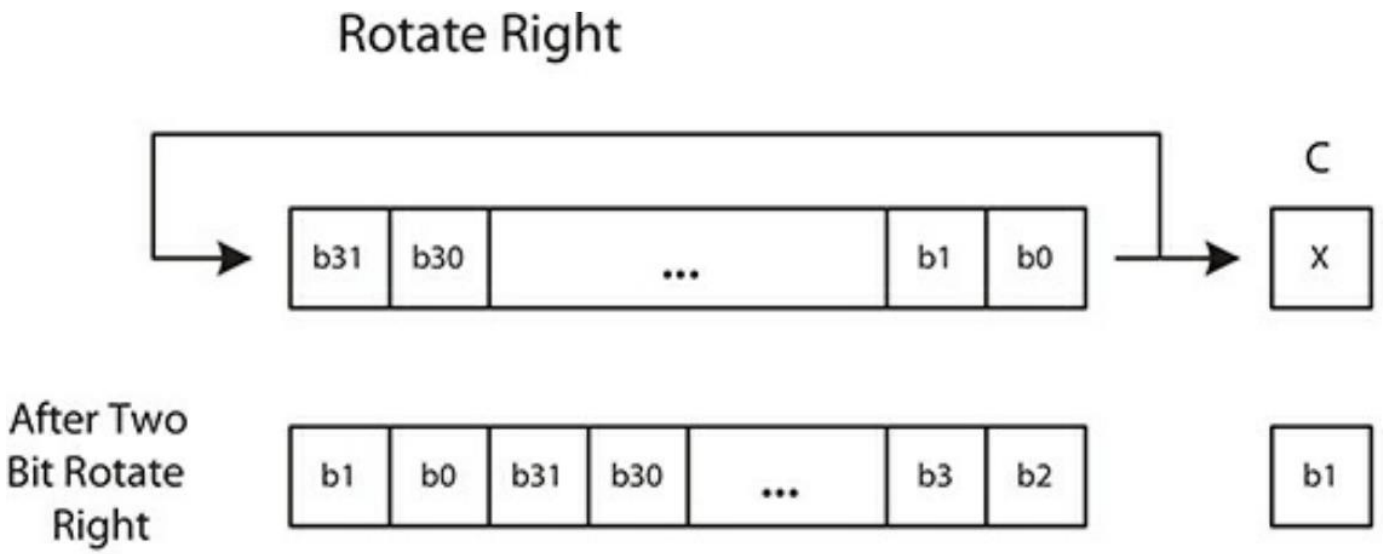
- Shift left by one bit position leads to multiplication by 2.
- The shift length can take values between 0 and 31 for this instruction.
- In case of LSLS, the bits leaving the register (from most significant bit [MSB] side) go to the carry flag bit.

```
LSL    R2, R1, #10          ; R2 = R1 << 10
LSLS   R3, R1, R5           ; R3 = R1 << R5
```

- In the first instruction the contents of register R1 are shifted left by 10 bit positions. This is equivalent to multiplication by 1024.
- In the second instruction, register R5 specifies the number of bits by which the contents of register R1 will be shifted. In this instruction, the flags do get updated after the instruction execution.

Rotate Right

SHIFT AND ROTATE INSTRUCTIONS



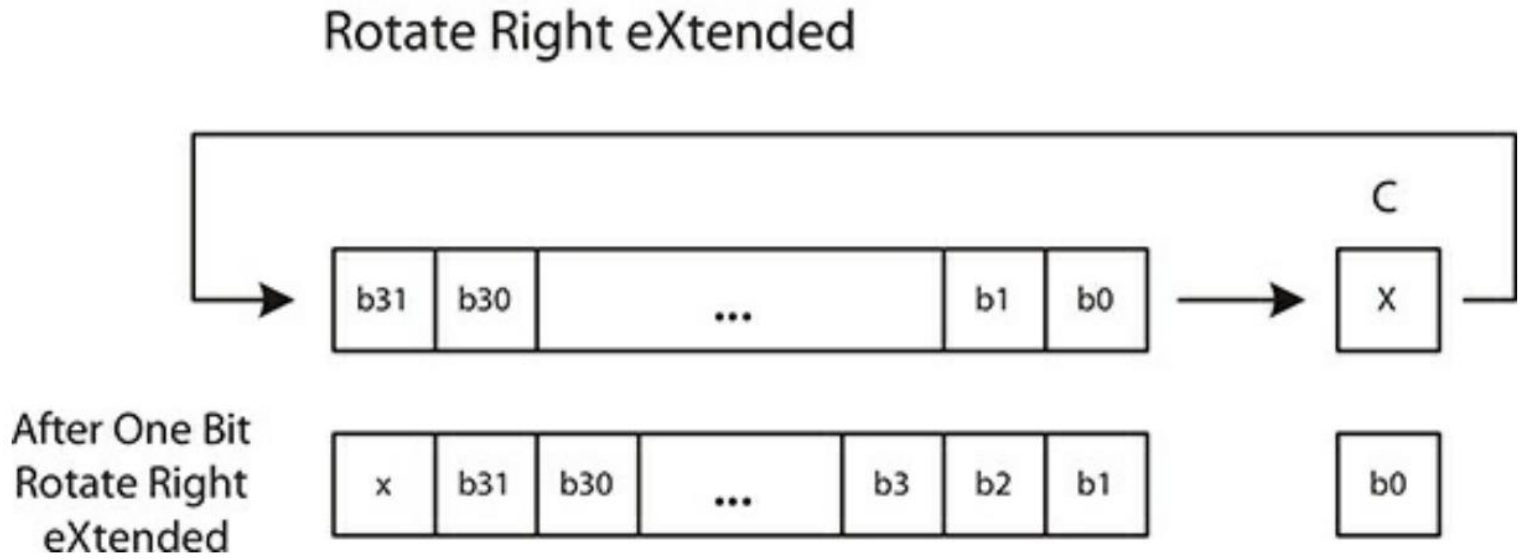
- ROR rotates the contents by a specified value without making **C** flag part of the rotation
- The range of shift length is 1 to 31
- The rotate left can be implemented by using rotate right operation with a different rotate value, e.g.,

```
ROR    R2, R2, #29      ; Equivalent to rotate left by 3 bit
                        ; R2 = 64 after execution
```

- It is important to realize that due to the barrel shifter the rotate right operation for 29 bit positions takes one processor clock cycle and no issue regarding execution performance inefficiency arise.

Rotate Right Extended

SHIFT AND ROTATE INSTRUCTIONS



- The special rotate right extended (RRX) instruction has a slightly different behavior from the usual rotate operations.
- One key difference is that no count is specified and this instruction always rotates by one bit.
- Second, 1 bit rotation in RRX involves the C flag as part of the rotation. The LSB of the register moves to C flag and the original value of C flag enters the MSB of the register

Rotate Right Extended

SHIFT AND ROTATE INSTRUCTIONS

```
LSRS   R2, R2, #1      ; R2 = R2 << 1 and C = LSB of R2
RRX    R1, R1          ; R1 = ((MSB = C) + R1 << 1)
```

- Consider the scenario where you have a 64 bit number and we are interested in performing logical shift right by 1 bit.
- Assume that R2 contains high 32 bits and R1 contains low 32 bits of the 64 bit number.
- Now applying LSRS operation to R2 will shift the LSB of R2 to C flag.
- Next using RRX with R1 register will move the value of carry flag (which contains the LSB of R2) to the MSB of R1 and shift right the original value of R1 by 1 bit to achieve the desired result.

Instruction Encoding

SHIFT AND ROTATE INSTRUCTIONS

- It is important to notice that different shift operations have different shift ranges.
- The amount by which the register will be shifted is specified either using a 5-bit field in the instruction or it is specified by the bottom byte of a register.
- However, using 5-bit field in the instruction code has no extra overhead, while using a register requires an extra cycle. When we use 5-bit field for specifying shift range, it allows 32 different values.
- Since we need to have a 0 value when we don’t want to apply any shifts the possible values of LSL range from 0-31.
- On the other hand, LSR and ASR have a range of 1-32 because LSR #0 or ASR #0 are not required since they are equivalent to LSL#0 and can be replaced by LSL#0 (this aspect is tool dependent and is available only if supported by the tools).
- The LSR #32 or ASR #32 can be useful if we want to set the register value either to zeroes or ones.

LOGICAL INSTRUCTIONS

ARM Cortex-M4 Assembly Language

Logical Instructions

The logical instructions supported by Cortex-M3/M4 are given below

1. **AND Instruction**: Bitwise logical AND operation
2. **OR (ORR) Instruction**: Bitwise logical OR operation
3. **Exclusive OR (EOR) Instruction**: Bitwise logical XOR operation
4. **OR negative (ORN) Instruction**: Bitwise logical OR operation with bitwise complement of second operand
5. **Bit clear (BIC) Instruction**: Bitwise logical AND operation with bitwise complement of second operand

Instruction Encoding and Flags

LOGICAL INSTRUCTIONS

Mnemonic	Brief description	Encoding	Flags
AND	Logical AND operation	16 or 32 bit	N,Z,C
ORR	Logical OR	16 or 32 bit	N,Z,C
EOR	Exclusive OR	16 or 32 bit	N,Z,C
ORN	Logical OR NOT	32 bit	N,Z,C
BIC	Bit clear	16 or 32 bit	N,Z,C

Instruction Encoding and Flags

LOGICAL INSTRUCTIONS

$$
\begin{array}{llll}
\text{AND}\{\text{S}\}\{cond\}\{.size\} & \{Rd,\} & Rn, & Operand2 \\
\text{BIC}\{\text{S}\}\{cond\}\{.size\} & \{Rd,\} & Rn, & Operand2 \\
\text{EOR}\{\text{S}\}\{cond\}\{.size\} & \{Rd,\} & Rn, & Operand2 \\
\text{ORN}\{\text{S}\}\{cond\} & \{Rd,\} & Rn, & Operand2 \\
\text{ORR}\{\text{S}\}\{cond\}\{.size\} & \{Rd,\} & Rn, & Operand2
\end{array}
$$

- **Rd** Specifies the optional destination register.
- **Rn** Specifies the register containing the value as 1st operand.

Flexible Second Operand

LOGICAL INSTRUCTIONS

Operand2 Flexible second operand, with following options

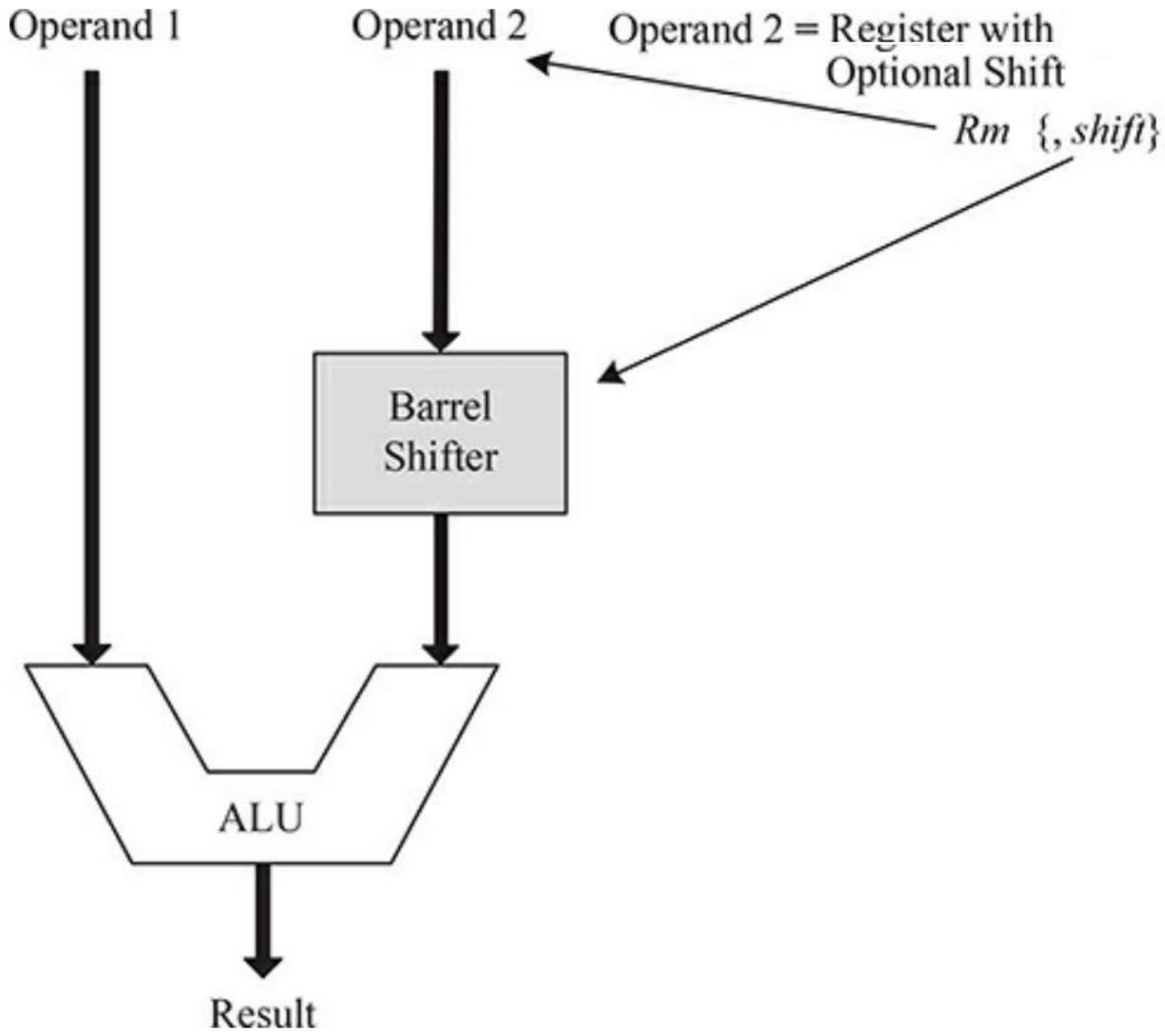
- **Constant**: specified as *#constant* where *constant* can be any constant that can be produced by shifting an 8-bit value left by any number of bits within a 32-bit word.
- **Register**: *Rm* specifies the register holding the data
- **Register with Optional Shift**: *Rm* {, shift}, where *shift* is an optional shift to be applied to Rm. It can be one of the following options.
  - When Operand2 is specified as register with optional shift, the specified value of the shift is applied to the contents of register Rm and the resulting 32-bit value is used by the instruction and the contents in the register Rm remain unchanged.

ASR #*n*	Arithmetic shift right *n* bits, $1 < n < 32$.
LSL #*n*	Logical shift left *n* bits, $0 < n < 31$.
LSR #*n*	Logical shift right *n* bits, $1 < n < 32$.
ROR #*n*	Rotate right *n* bits, $1 < n < 31$.
RRX	Rotate right one bit, with extend. If omitted, no shift occurs, equivalent to LSL#0.

Flexible Second Operand

LOGICAL INSTRUCTIONS

- When Operand2 is a register with optional shift the shift operation is executed by the barrel shifter.
- The barrel shifter has the capability of performing a shift operation by any arbitrary number of bit positions (up to 32 bits in the case of Cortex-M processors) in one instruction cycle of the processor.
- E.g., the shift operation by 3 bits or 30 bits takes the same amount of execution time when using a barrel shifter.



Examples

LOGICAL INSTRUCTIONS

```
AND    R2, R1, #0xFF          ; R2 = R1 & 0xFF
ORR    R3, R1, R5             ; R3 = R1 | R5
BIC    R5, R1                 ; R5 = R5 & (~R1)
EORS   R4, R1, #0x55          ; R4 = R1 ^ 0x55
```

- The first two instructions simply illustrate the use of AND and OR operations. The comment field shows the C language equivalent representation of the logical operations.
- In the BIC instruction above, we have used only two operands and have omitted the optional destination register.
- In the last instruction, the flags in APSR are affected based on the result of the operation.

Applications

LOGICAL INSTRUCTIONS

- Let's discuss some useful functionalities by using shift and rotate bit operations combined with logical operations that can be considered as fundamental building blocks for many different embedded system applications.

Testing n^{th} bit of a variable

- Let R0 contain the value of variable x. We are interested in finding whether the n^{th} bit of variable x is equal to 1 or 0.
- Method 1: Check the zero flag in the APSR to find out whether the n^{th} bit is 1 or 0.

```
LSR    R0, #7          ; R0 = R0 << 7
ANDS   R0, #1          ; R0 = R0 & 1 and flags are update
```

- The original value of R0 is changed and must be preserved before performing bit testing
- Method 2: Used a free register R1 and set its n^{th} bit to 1 to perform bit test operation.

```
MOV    R1, #1
LSL    R1, #7          ; R1 = 1 << 7
ANDS   R1, R1, R0      ; R1 = R0 & R1 and flags are update
```

Applications

LOGICAL INSTRUCTIONS

Bit Masking

- Setting or clearing specific bit or bits of a variable or register
- One of the most fundamental operations used in embedded programming
- **Set a bit**: setting the fourth bit (this is bit 3 when LSB is labeled as bit 0) of register R0

```
MOV    R1, #1
LSL    R1, #4              ; R1 = 1 << 4
ORR    R0, R0, R1          ; R0 = R0 | R1
```

- **Clear a bit**: clearing the fourth bit of register R0

```
MOV   R1, #1
LSL   R1, #4               ; R1 = 1 << 4
BIC   R0, R0, R1           ; R0 = R0 & ~R1
```

ARIHMETIC INSTRUCTIONS

ARM Cortex-M4 Assembly Language

Instruction Encoding and Flags

Basic Arithmetic Instructions

Mnemonic	Brief description	Encoding	Flags
ADC	Add with carry	16 or 32 bit	N,Z,C,V
ADD	Addition	16 or 32 bit	N,Z,C,V
ADDW	Addition (32-bit encoded)	32 bit	N,Z,C,V
RSB	Reverse subtract	16 or 32 bit	N,Z,C,V
SBC	Subtract with carry	16 or 32 bit	N,Z,C,V
SUB	Subtract	16 or 32 bit	N,Z,C,V
SUBW	Subtract (32-bit encoded)	32 bit	N,Z,C,V
MUL	Multiply (32-bit result)	16 or 32 bit	N,Z
MLA	Multiply with accumulate (32-bit result)	32 bit	-
MLS	Multiply with subtract (32-bit result)	32 bit	-
SMLAL	Signed multiply accumulate (64-bit result)	32 bit	-
SMULL	Signed multiply (64-bit result)	32 bit	-
UMLAL	Unsigned multiply accumulate (64-bit result)	32 bit	-
UMULL	Unsigned multiply (64-bit result)	32 bit	-
UDIV	Unsigned divide	32 bit	-
SDIV	Signed divide	32 bit	-

Addition and Subtraction Instructions

Basic Arithmetic Instructions

- **ADD Instruction**: Simply adds two numbers and can be encoded using either 16-bit or 32-bit formats
- **ADDW Instruction**: Always 32-bit encoded. The difference between ADD and ADDW is not only in the instruction encoding size but also in the instruction syntax
- **ADC Instruction**: Implements add with carry and adds the value of carry flag (at the time of instruction execution) to the operands of addition operation
- **SUB Instruction**: similar to ADD except that the operation performed is subtraction
- **SUBW Instruction**: similar to ADDW except that the operation performed is subtraction
- **SBC Instruction**: the current value of the carry flag is first inverted and then it is subtracted from the result of subtraction
- **RSB Instruction**: the order of operands is reversed

Instruction Encoding and Flags

Addition and Subtraction Instructions

ADD{S}{*cond*}{*.size*}	{*Rd*,}	*Rn*,	*Operand2*
ADC{S}{*cond*}{*.size*}	{*Rd*,}	*Rn*,	*Operand2*
SUB{S}{*cond*}{*.size*}	{*Rd*,}	*Rn*,	*Operand2*
SBC{S}{*cond*}{*.size*}	{*Rd*,}	*Rn*,	*Operand2*
RSB{S}{*cond*}{*.size*}	{*Rd*,}	*Rn*,	*Operand2*
ADDW{*cond*}	{*Rd*,}	*Rn*,	#*imm*12
SUBW{*cond*}	{*Rd*,}	*Rn*,	#*imm*12

- Most of the fields are the same as described previously for shift and logical instructions
- **Operand2**: Flexible second operand
- **#imm12:** represents an arbitrary integer value in the range 0-4095
- Both ADDW and SUBW cannot affect the status flags because of the absence of ‘S’ field

Examples – Simple

Addition and Subtraction Instructions

```
ADD   R2, R1, R3      ; R2 = R1 + R3
ADC   R2, R1, R3      ; R2 = R1 + R3 + C, here C represent
                      ; value of carry flag
SUBS R8, R6, #240     ; R8 = R6 - 240, sets the flags on
                      ; the result
RSB   R4, R4, #1280   ; R4 = 1280 - R4,
```

Example – 64-bit addition

Addition and Subtraction Instructions

Addition of two 64-bit numbers is accomplished using two instructions,

- **ADDS** performs addition of least significant 32-bit value and updates the flags based on the result
- **ADC** instruction is used to add the most significant 32-bits along with the updated value of C flag

- The following two instructions add a 64-bit integer contained in R2 and R3 to another 64-bit integer contained in R0 and R1.
- The result of addition is stored in registers R4 and R5.

```
ADDS   R4, R0, R2      ; adding lower 32-bit words
ADC    R5, R1, R3      ; adding higher 32-bit words along with
                       ; carry flag from lower word addition
```

Example – Flexible Second Operand

Addition and Subtraction Instructions

- This example illustrates the use of flexible second operand for an efficient implementation of the linear scaling with scaling coefficient represented as 2^m for $1 < m < 31$
- Consider an expression $y = 32x + c$ and assume that R3 = y, R2 = x, and R1 = c
- One possible implementation is given by the following assembly code

```
LSL   R2, R2, #5      ; Multiplying R2 with 32
ADD   R3, R2, R1      ; Now perform the addition operation
```

- An efficient implementation of the above expression is achieved by using the flexible second operand.

```
ADD R3, R1, R2, LSL #5 ; Shift left (logical) R2 by five
                       ; bits, add to R1 and finally store
                       ; result in R3. This operation is
                       ; equivalent to R3 = R1 + (R2 << 5)
```

- An efficient implementation of linear scaling will be presented using multiplication instruction that allows arbitrary choice of scaling coefficients instead of using specific 2^m.

Multiply and Divide Instructions

Basic Arithmetic Instructions

Two different types of multiplication instructions are supported.

- One set of multiplication instructions uses 32-bit operands, and the result of multiplication is also 32-bit.
- The other set of multiplication instructions produces a 64- bit result as a consequence of 32-bit multiplication.

Multiplication Instructions with 32-bit Result

Multiply and Divide Instructions

- Multiply (**MUL**), multiply with accumulate (**MLA**), and multiply with subtract (**MLS**) instructions use 32-bit operands and produce a 32-bit result.
- Since both the operands are 32-bit, it is possible that the result of multiplication can be as large as 64 bits.
- However, these multiplication instructions are limited in the sense that they only write the least significant 32 bits of the result to the destination register
- These instructions do not differentiate whether the operands are signed or unsigned integers.

Instruction Encoding and Flags

Multiplication Instructions with 32-bit Result

$$
\begin{array}{llllll}
\mathrm{MUL}\{S\}\{cond\} & \{Rd,\} & Rn, & Rm & & \\
\mathrm{MLA}\{cond\} & Rd, & Rn, & Rm, & Ra & \\
\mathrm{MLS}\{cond\} & Rd, & Rn, & Rm, & Ra &
\end{array}
$$

- **Rd**: Destination register that holds the least significant 32-bits of the result of operation.
- **Rn**, **Rm**: Registers containing the values to be multiplied.
- **Ra**: Register holding the value to be added to or subtracted from

- The MUL instruction multiplies the values contained by the registers Rn and Rm and writes the least significant 32 bits of the result in the destination register Rd.
- The MLA instruction multiplies the two operand values from Rn and Rm, adds the result of multiplication to the contents of Ra and finally writes the least significant 32 bits of the result to Rd.
- The MLS instruction first multiplies the contents of Rn and Rm and then subtracts the least significant 32 bits of the product from the value of register Ra and finally writes the least significant 32 bits of the result to register Rd

Examples

Multiplication Instructions with 32-bit Result

```
MUL    R1, R2, R5        ; Multiply, R1 = R2 x R5
MLA    R1, R2, R3, R5    ; Multiply with accumulate,
                         ; R1=(R2 x R3)+R5
MULS   R1, R2, R2        ; Multiply with flag update,
                         ; R1=R2 x R2
MLS    R1, R5, R6, R7    ; Multiply with subtract,
                         ; R1=R7-(R5 x R6).
```

- It is important to remember that only the least significant 32 bits of the result are written to a destination register

Example – Arithmetic Sum of a Series

Multiplication Instructions with 32-bit Result

- The arithmetic sum of the first n integers is given by $S = n(n + 1)/2 = (n^2 + n)/2$.
- Assuming register R2 = n, the last expression is implemented using the following code

```
MLA   R1, R2, R2, R2   ; Multiply with accumulate,
                       ; R1 = (R2xR2)+R2
LSR   R1, R1, #1       ; R1 = R1/2
```

- The equivalent C code for the above assembly program is given below.

```
R1 = (R2*(R2 + 1)) << 1;
```

Multiplication Instructions with 64-bit Result

Multiply and Divide Instructions

- The Cortex-M3/M4 processors also support 32-bit multiplication instructions with or without accumulate and produce 64-bit results.
- These instructions support both signed as well as unsigned operand values and produce correspondingly signed or unsigned results.
- Following are these instructions:
  - Unsigned multiply long (**UMULL**)
  - Unsigned multiply with accumulate long (**UMLAL**)
  - Signed multiply long (**SMULL**)
  - Signed multiply with accumulate long (**SMLAL**)

Instruction Encoding and Flags

Multiplication Instructions with 64-bit Result

$$\begin{array}{llllll}
\text{UMULL}\{cond\} & RdLow, & RdHigh, & Rn, & Rm \\
\text{UMLAL}\{cond\} & RdLow, & RdHigh, & Rn, & Rm \\
\text{SMULL}\{cond\} & RdLow, & RdHigh, & Rn, & Rm \\
\text{SMLAL}\{cond\} & RdLow, & RdHigh, & Rn, & Rm
\end{array}$$

- **Rn**, **Rm**: Registers containing the operands to be multiplied.
- **RdHigh**, **RdLow** : These are the destination registers to hold the 64-bit result of multiplication. For UMLAL and SMLAL instructions these registers also hold the value to be accumulated.

Instruction Encoding and Flags

Multiplication Instructions with 64-bit Result

UMULL instruction

- The values from the registers Rn and Rm are treated as unsigned integers.
- It multiplies the 32-bit integer operands and writes the least significant 32 bits of the result in RdLow while the most significant 32 bits of the result are written to RdHigh.

UMLAL instruction

- Uses the values from the registers Rn and Rm as unsigned numbers.
- It multiplies these integers, adds the 64-bit result to the 64-bit unsigned integer contained in RdHigh and RdLow, and finally writes the result back to RdHigh and RdLow registers.
- The least significant 32-bits of the multiplication result are added to RdLow, while the most significant 32-bits of the multiplication result are added to the RdHigh register.

Instruction Encoding and Flags

Multiplication Instructions with 64-bit Result

SMULL instruction

- treats the values from the registers Rn and Rm as 2's complement signed numbers.
- It multiplies these operands and places the least significant 32-bits of the result in RdLow while the most significant 32-bits of the result are stored in RdHigh.

SMLAL instruction

- treats the values from Rn and Rm as signed integers in 2's complement form.
- This instruction first multiplies these operands, adds the 64-bit result of multiplication to the 64-bit signed value contained in RdHigh and RdLow, and finally writes the result back to RdHigh and RdLow registers.

The registers RdHigh and RdLow used in these instructions must be different from each other. In addition, these instructions do not affect the condition code flags.

Examples

Multiplication Instructions with 64-bit Result

- Multiplication instructions with 32-bit operands and 64-bit result

```
UMULL   R0, R4, R5, R6 ; Unsigned [R4 R0] = R5 x R6
SMLAL   R4, R5, R3, R8 ; Signed [R5 R4] = [R5 R4] + R3 x R8
```

- The notation [x y] represents that y is the lower 32-bits and x is the upper 32-bits of a 64-bit variable

Divide Instructions

Multiply and Divide Instructions

- Unsigned division (UDIV) and signed division (SDIV) are the two integer divide instructions that are supported by the Cortex-M3/M4 architecture.
- It should be noted that in case of integer division the fractional part (i.e., the remainder) is discarded. The syntax for these two instruction is given below.

$$\begin{array}{llll} \text{SDIV}\{cond\} & Rd, & Rn, & Rm \\ \text{UDIV}\{cond\} & Rd, & Rn, & Rm \end{array}$$

- **Rd**: Destination register, contains the result (non-fractional part) of division operation
- **Rn**: register containing the 32-bit value of dividend
- **Rm**: register containing the 32-bit value of divisor
- These instructions do not affect the condition code flags in the APSR

```
UDIV   R1, R8, R2    ; Unsigned divide, R1 = R8/R2
SDIV   R1, R2, R4    ; Signed divide, R1 = R2/R4
```

Example – Implementation of remainder operation

Divide Instructions

- In C and other high level languages remainder operation is quite useful for performing certain tasks. i.e., for variables x and y the remainder is evaluated as x%y.
- In x86 processor architecture, the divide instruction produces both quotient and remainder therefore reminder operation is quite straightforward
- However, in ARM Cortex-M processor architecture, the divide instruction only produces the quotient. In this case, we need to develop an algorithm to evaluate the remainder.
- Since the divide instruction in ARM Cortex-M produces the quotient, we can use the expression x — (x/y)y to evaluate the remainder indirectly.
- Assuming R0 = x and R1 = y the following assembly code evaluates the reminder.

```
SDIV   R2, R0, R1          ; Signed divide, R2 = R0/R1
MLS    R3, R1, R2, R0      ; R3 = R0 - (R2 x R1)
```

DATA MOVEMENT INSTRUCTIONS

ARM Cortex-M4 Assembly Language

Data Movement Instructions

Data movement inside the processor can be performed using following instructions:

- **MOV Instruction**: Move instruction, moves immediate value or data from a processor register to another register
- **MOVW Instruction**: Move word instruction, moves 16-bit immediate value to a processor register specified as destination
- **MVN Instruction**: Move negative instruction, moves negative of the specified value to the processor register specified as destination
- **MOVT Instruction**: Move top instruction, moves 16-bit immediate value to the top half-word of the processor register specified as destination

Instruction Encoding and Flags

Data Movement Instructions

$$
\begin{array}{lll}
\text{MOV}\{\text{S}\}\{cond\} & Rd, & Operand2 \\
\text{MOV}\{cond\} & Rd, & \#imm16 \\
\text{MVN}\{\text{S}\}\{cond\} & Rd, & Operand2 \\
\text{MOVW}\{cond\} & Rd, & \#imm16 \\
\text{MOVT}\{cond\} & Rd, & \#imm16
\end{array}
$$

- Instructions specified with {S} can only update ‘Z’ and ‘N’ flags of APSR
- However, there is a possibility of getting the ‘C’ flag updated because of evaluation of Operand2 and should not be considered as a flag update due to move operation.

MOV Instruction with Optional Shift

Data Movement Instructions

- The MOV instruction copies the value from Operand2 to the register Rd.
- When Operand2 in a MOV instruction is a register with an optional shift and the shift operation is other than LSL#0, then it preferred that the MOV instruction should be replaced with the corresponding shift instruction.
- For instance, the instruction

$$\mathrm{MOV\{S\}\{}cond\mathrm{\}}\quad Rd,\quad Rm,\quad ASR\#n.$$

- should be replaced by the corresponding shift instruction, which is and is the preferred syntax for this operation.

$$\mathrm{ASR\{S\}\{}cond\mathrm{\}}\quad Rd,\quad Rm,\quad \#n$$

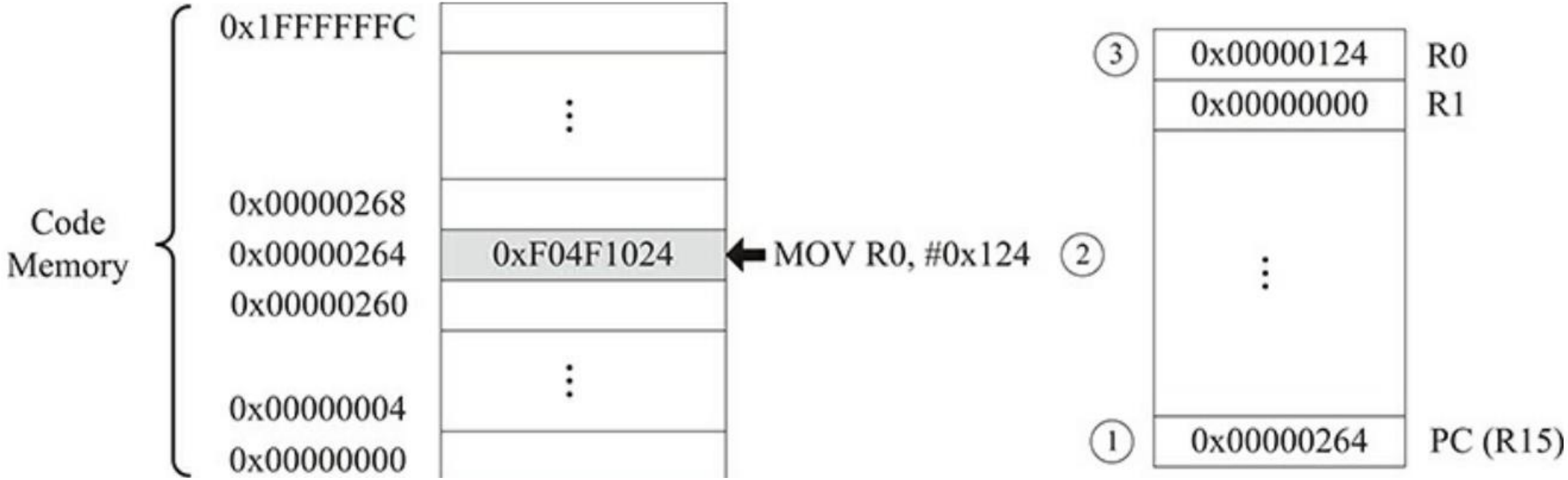
Examples

Data Movement Instructions

```
MOVS   R11, #0x000B ; Write 0x000B to R11, flags are updated
MOV    R1,  #0xA05  ; Write 0xA05 to R1, do not update flags
MOVS   R10, R12     ; Move R12 to R10, flags are updated
MOV    R3,  #23     ; Write value of 23 to R3
MOV    R8,  SP      ; Write stack pointer to R8
MVNS   R2,  #0xF    ; Write 0xFFFFFFF0 (bitwise inverse
                    ; of 0xF) to R2 and update flags.
MOVT   R3,  #0xF123 ; Write 0xF123 to upper halfword of R3,
                    ; lower halfword and APSR are unchanged
MOVW R2, #0x4466    ; move immediate value 0x4466 to lower
                    ; halfword of R2
MOVT R2, #0x5533    ; move immediate value 0x5533 to upper
                    ; halfword of R2
```

MOV Instruction Execution

Data Movement Instructions



1. PC is holding the address of MOV R0, 0x124 instruction in ROM, which is 0x264. And the contents of register R0 are equal to zero.
2. Move instruction is decoded and executed by processor, the PC will be incremented to 0x268
3. The content of R0 is updated as a result of MOV instruction execution

THANK YOU

Any Questions???