# Chapter 4
# ARM Arithmetic and Logic Instructions

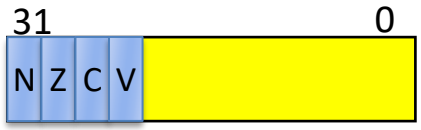
Z. Gu

Fall 2025

Acknowledgement: Lecture slides based on Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C, University of Maine https://web.eece.maine.edu/~zhu/book/

# ARM Programming Model

| R0 |
|---|
| R1 |
| R2 |
| R3 |
| R4 |
| R5 |
| R6 |
| R7 |
| R8 |
| R9 |
| R10 |
| R11 |
| R12 |
| R13: Stack Pointer (SP) |
| R14: Link Register (LR) |
| R15: Program Counter (PC) |




CPSR (Current Program Status Register)

- Every arithmetic, logical, or shifting operation sets xPSR bits:
  - N (negative), Z (zero), C (carry), V (overflow).

# Overview: Arithmetic and Logic Instructions

- **Shift** : **LSL** (logic shift left), **LSR** (logic shift right), **ASR** (arithmetic shift right), **ROR** (rotate right), **RRX** (rotate right with extend)
- **Logic**: **AND** (bitwise and), **ORR** (bitwise or), **EOR** (bitwise exclusive or), **ORN** (bitwise or not), **MVN** (move not)
- **Bit set/clear**: **BFC** (bit field clear), **BFI** (bit field insert), **BIC** (bit clear), **CLZ** (count leading zeroes)
- **Bit/byte reordering**: **RBIT** (reverse bit order in a word), **REV** (reverse byte order in a word), **REV16** (reverse byte order in each half-word independently), **REVSH** (reverse byte order in each half-word independently)
- **Addition**: **ADD**, **ADC** (add with carry)
- **Subtraction**: **SUB**, **RSB** (reverse subtract), **SBC** (subtract with carry)
- **Multiplication**: **MUL** (multiply), **MLA** (multiply-accumulate), **MLS** (multiply-subtract), **SMULL** (signed long multiply-accumulate), **SMLAL** (signed long multiply-accumulate), **UMULL** (unsigned long multiply-subtract), **UMLAL** (unsigned long multiply-subtract)
- **Division**: **SDIV** (signed), **UDIV** (unsigned)
- **Saturation**: **SSAT** (signed), **USAT** (unsigned)
- **Sign extension**: **SXTB** (signed), **SXTH**, **UXTB**, **UXTH**
- **Bit field extract: SBFX** (signed), **UBFX** (unsigned)
- Syntax

```
<Operation>{<cond>}{S} Rd, Rn, Operand2
```

# Example: **Add**

- Unified Assembler Language (UAL) Syntax

```
ADD r1, r2, r3      ; r1 = r2 + r3
ADD r1, r2, #4      ; r1 = r2 + 4
```

- Traditional Thumb Syntax

```
ADD r1, r3          ; r1 = r1 + r3
ADD r1, #15         ; r1 = r1 + 15
```

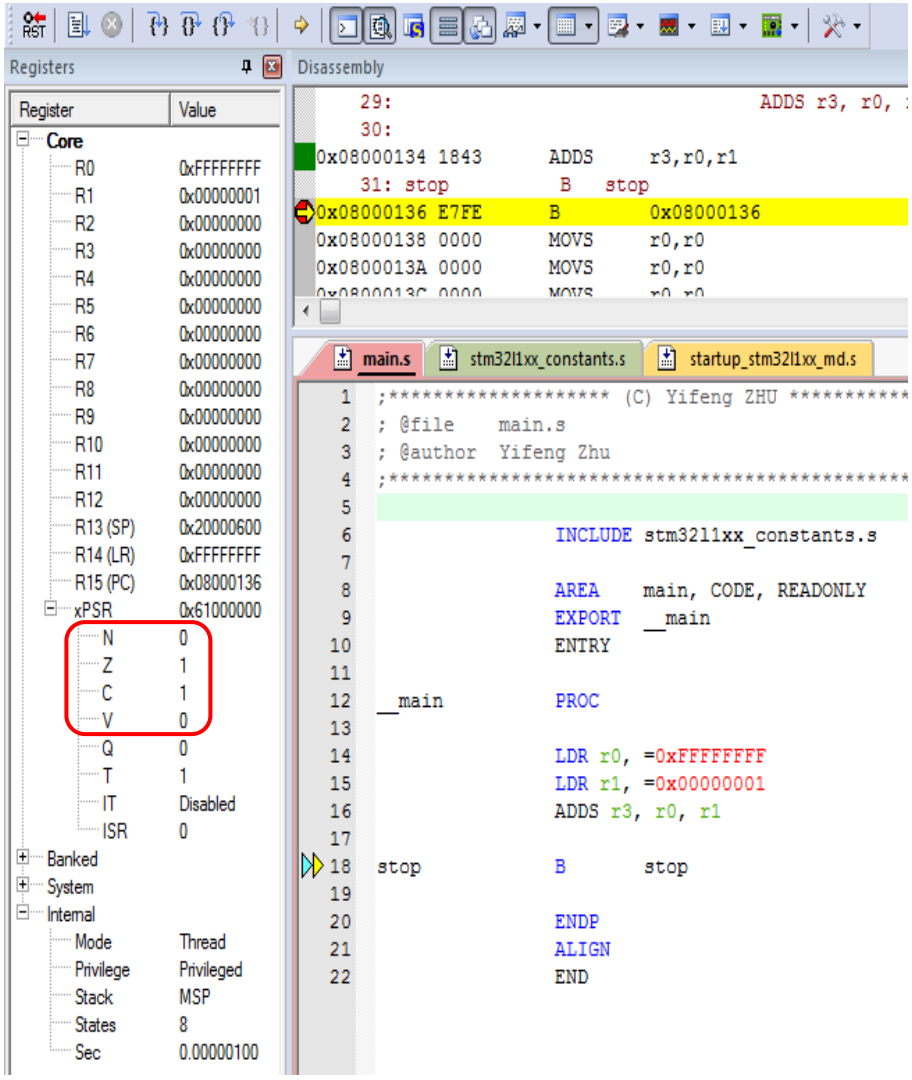
# Commonly Used Arithmetic Operations

| | |
|---|---|
| ADD {Rd,} Rn, Op2 | **Add**<br>Rd ← Rn + Op2 |
| ADC {Rd,} Rn, Op2 | **Add with carry**<br>Rd ← Rn + Op2 + Carry |
| SUB {Rd,} Rn, Op2 | **Subtract**<br>Rd ← Rn - Op2 |
| SBC {Rd,} Rn, Op2 | **Subtract with carry**<br>Rd ← Rn - Op2 + Carry - 1 |
| RSB {Rd,} Rn, Op2 | **Reverse subtract**<br>Rd ← Op2 - Rn |
| MUL {Rd,} Rn, Rm | **Multiply**<br>Rd ← (Rn × Rm)[31:0] |
| MLA Rd, Rn, Rm, Ra | **Multiply with accumulate**<br>Rd ← (Ra + (Rn × Rm))[31:0] |
| MLS Rd, Rn, Rm, Ra | **Multiply and subtract**<br>Rd ← (Ra – (Rn × Rm))[31:0] |
| SDIV {Rd,} Rn, Rm | **Signed divide**<br>Rd ← Rn ÷ Rm |
| UDIV {Rd,} Rn, Rm | **Unsigned divide**<br>Rd ← Rn ÷ Rm |
| SSAT Rd, #n, Rm {,shift #s} | **Signed saturate** |
| USAT Rd, #n, Rm {,shift #s} | **Unsigned saturate** |

# Instruction Suffix:
# S: Set Condition Flags

```
start
        LDR   r0, =0xFFFFFFFF
        LDR   r1, =0x00000001
        ADDS  r0, r0, r1
stop    B   stop
```

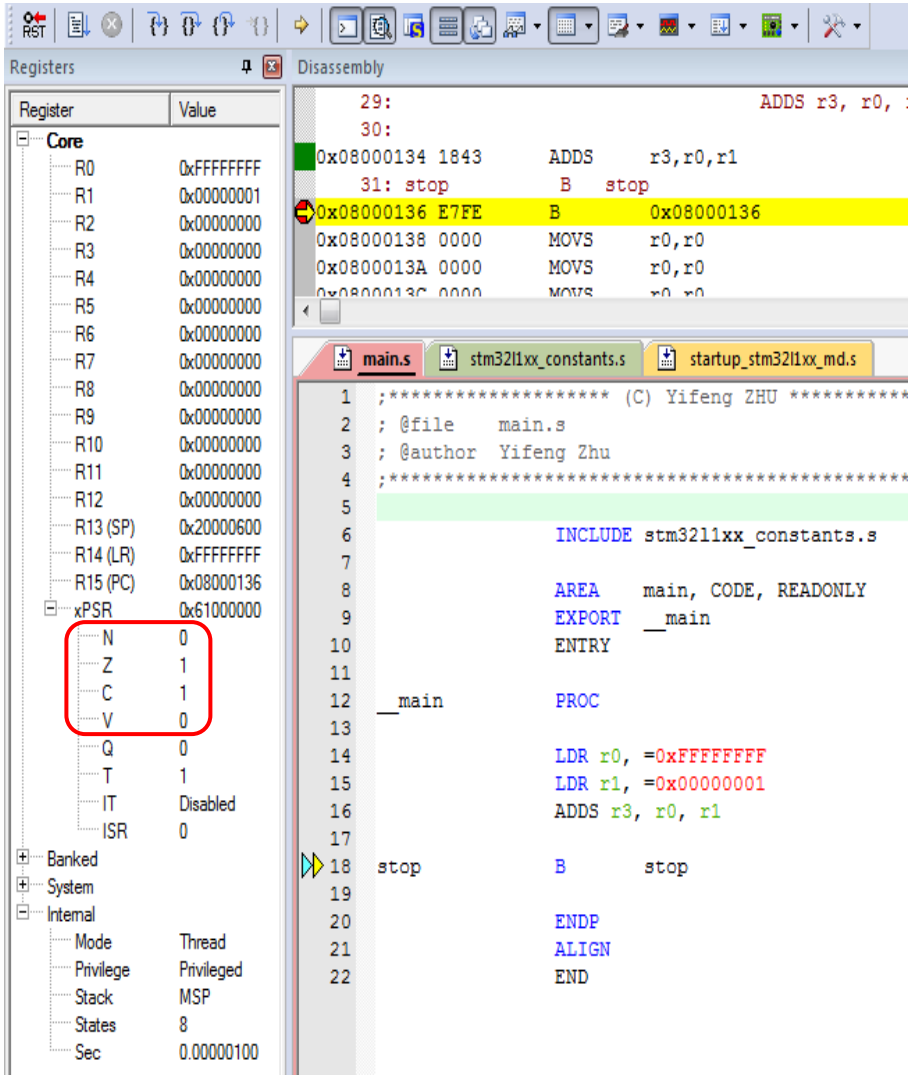
- For most instructions, add a suffix **S** to update N, Z, C,V flags in APSR register.
  - **N** - Negative
  - **Z** - Zero
  - **C** - Carry
  - **V** - oVerflow
- In this example, the Z and C bits are set.



# Instruction Suffix:
# S: Set Condition Flags

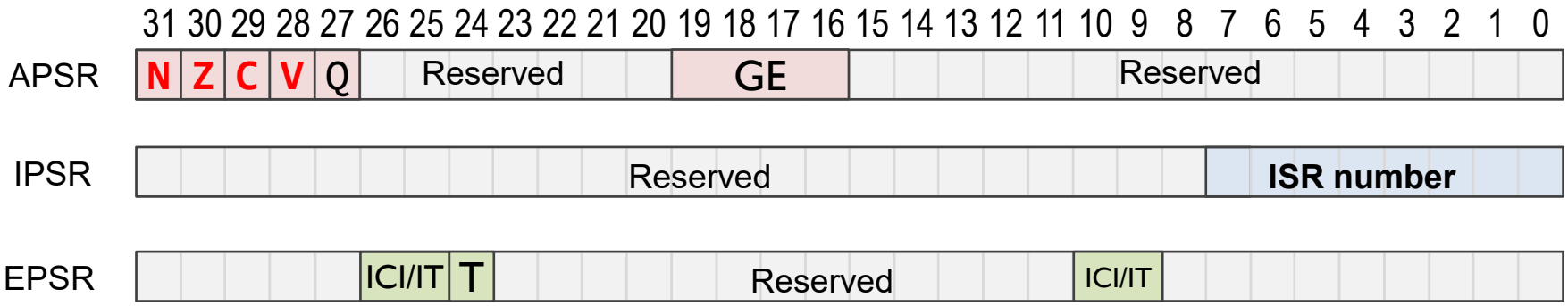
```
start
        LDR   r0, =0xFFFFFFFF
        LDR   r1, =0x00000001
        ADDS  r0, r0, r1
stop    B     stop
```

- R0 = 0x00000000
- **N = 0, Z = 1, C = 1, V = 0**
- CPU sets both C and V:
  - If r0 and r1 are unsigned int: $2^{32}$-1+1, carry flag C=1
  - If r0 and r1 are signed int: -1+1, overflow flag V=0
  - CPU does not know about signed or unsigned integers
  - Software is responsible to use C or V in follow-on code.



# Program Status Register (PSR)

- Application PSR (**APSR**), Interrupt PSR (**IPSR**), Execution PSR (**EPSR**)




Combine them together into one register **PSR = APSR + IPSR + EPSR**


- GE flags are only available on Cortex-M4 and M7
- PSR is named CPSR (Current Program Status Register) in ARM7/ARM9 and ARMv4/ARMv5 processors: named xPSR (Extended Program Status Register) in ARM Cortex-M processors (ARMv7-M)

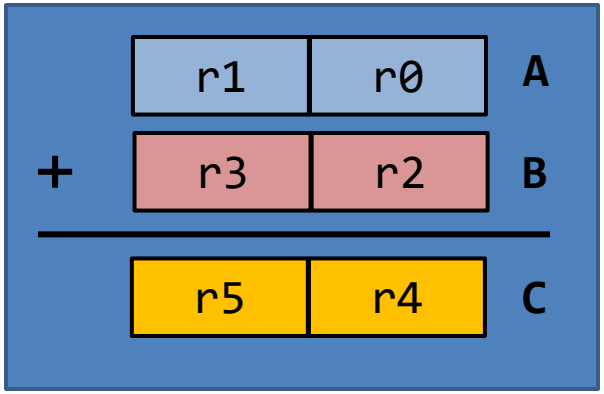
# 64-bit Addition

A register can only store 32 bits

A 64-bit integer needs two registers

Split 64-bit addition into two 32-bit additions

```
start
  ; C = A + B
  ; Two 64-bit integers A (r1,r0) and B (r3,r2).
  ; Result C (r5, r4)
  ; A = 00000002FFFFFFFF
  ; B = 0000000400000001
  LDR   r0, =0xFFFFFFFF   ;  A's lower 32 bits
  LDR   r1, =0x00000002   ;  A's upper 32 bits
  LDR   r2, =0x00000001   ;  B's lower 32 bits
  LDR   r3, =0x00000004   ;  B's upper 32 bits

  ;  Add A to B
  ADDS r4, r2, r0   ; C[31..0] = A[31..0] + B[31..0],  update Carry
  ADC   r5, r3, r1   ; C[64..32] = A[64..32] + B[64..32] + Carry

stop   B   stop
```




ADDS "ADD and Set flags": sets the carry flag C=1 if there is a carry

ADC "ADD with Carry" consumes that carry to complete the high-half addition

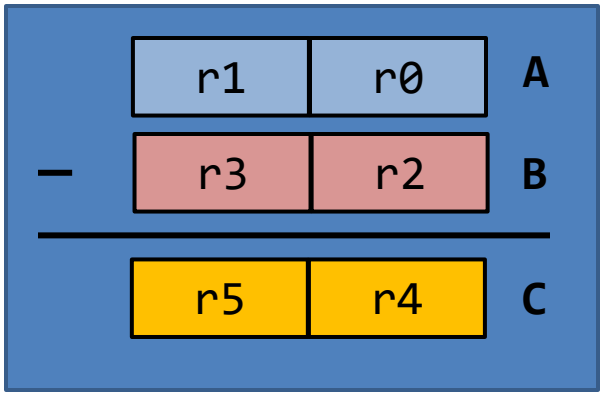
# 64-bit Subtraction

A register can only store 32 bits

A 64-bit integer needs two registers

Split 64-bit subtraction into two 32-bit subtractions

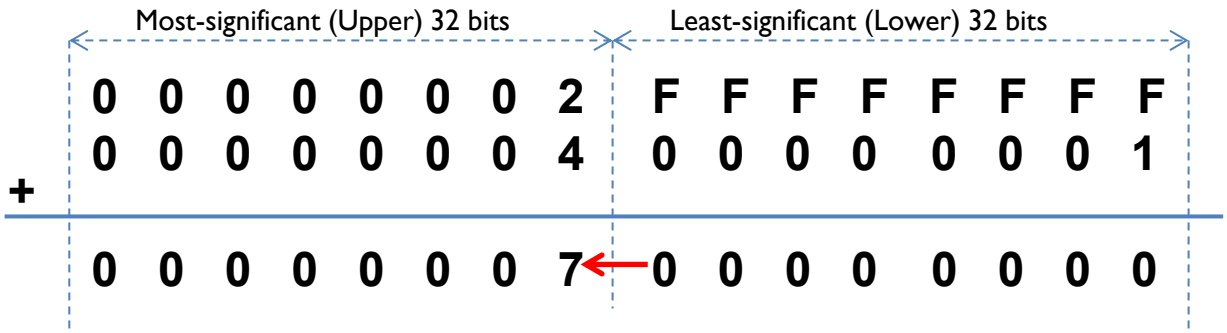
```
start
  ; C = A - B
  ; Two 64-bit integers A (r1,r0) and B (r3,r2).
  ; Result C (r5, r4)
  ; A = 00000002FFFFFFFF
  ; B = 0000000400000001
  LDR  r0, =0xFFFFFFFF  ;  A's lower 32 bits
  LDR  r1, =0x00000002  ;  A's upper 32 bits
  LDR  r2, =0x00000001  ;  B's lower 32 bits
  LDR  r3, =0x00000004  ;  B's upper 32 bits

  ;  Subtract B from A
  SUBS r4, r0, r2  ; C[31..0]= A[31..0] - B[31..0],  update Carry
  SBC  r5, r1, r3  ; C[64..32]= A[64..32] - B[64..32] - Carry

stop  B  stop
```



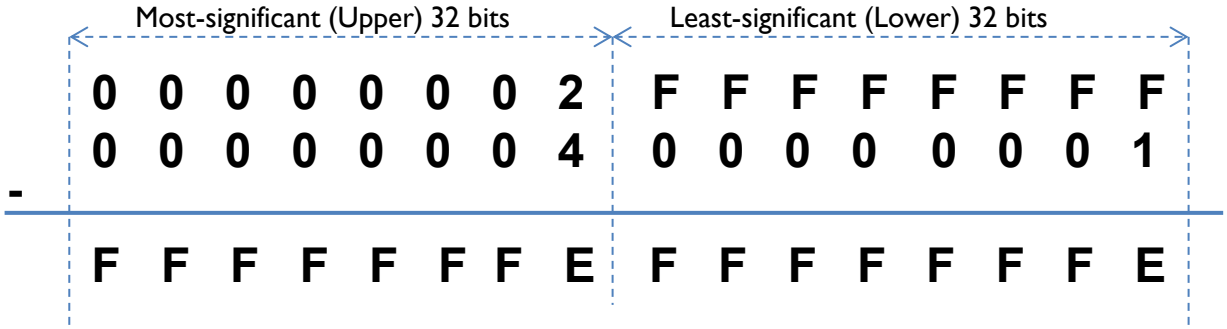

SUBS "SUB and Set flags": sets the carry flag C=0 if there is borrow (borrow flag=1)

SBC "SUB with Carry" consumes that carry to complete the high-half addition

# Example: 64-bit Addition & Subtraction




**64-bit Addition: Carry C=1**




**64-bit Subtraction: Borrow=0**
**(Carry C=1)**

# Example: Short Multiplication and Division

```
MUL: Signed multiply
MUL   r6, r4, r2          ; r6 = LSB32( r4 × r2 )

UMUL: Unsigned multiply
UMUL r6, r4, r2           ; r6 = LSB32( r4 × r2 )

MLA: Multiply with accumulation
MLA   r6, r4, r1, r0      ; r6 = LSB32( r4 × r1 ) + r0

MLS:  Multiply with subtract
MLS   r6, r4, r1, r0      ; r6 = LSB32( r4 × r1 ) - r0
```

**LSB32**: Least significant 32 bits

# Example: Long Multiplication

| | |
|---|---|
| `UMULL RdLo, RdHi, Rn, Rm` | **Unsigned long multiply**<br>RdHi,RdLo ← unsigned(Rn × Rm) |
| `SMULL RdLo, RdHi, Rn, Rm` | **Signed long multiply**<br>RdHi,RdLo ← signed(Rn × Rm) |
| `UMLAL RdLo, RdHi, Rn, Rm` | **Unsigned multiply with accumulate**<br>RdHi,RdLo ← unsigned(RdHi,RdLo + Rn × Rm) |
| `SMLAL RdLo, RdHi, Rn, Rm` | **Signed multiply with accumulate**<br>RdHi,RdLo ← signed(RdHi,RdLo + Rn × Rm) |

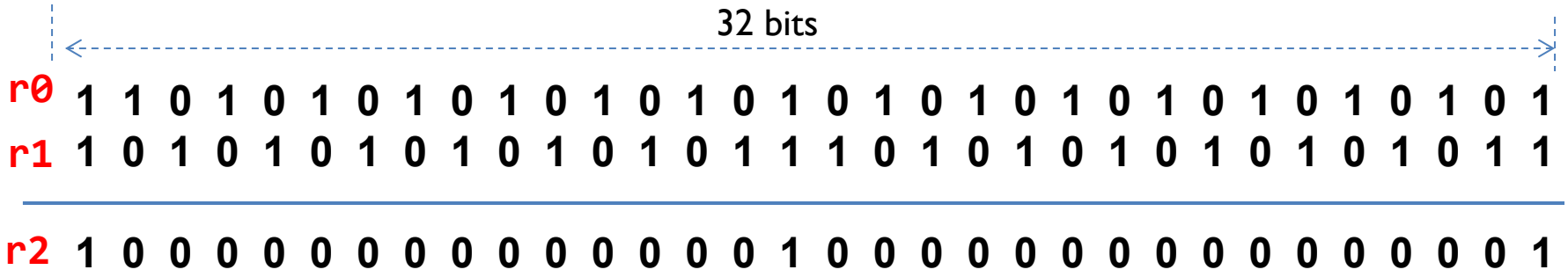
The result has 64 bits, placed in two registers.

```
UMULL r3, r4, r0, r1    ; r4:r3 = r0 × r1, r4 = MSB bits, r3 = LSB bits
SMULL r3, r4, r0, r1    ; r4:r3 = r0 × r1
UMLAL r3, r4, r0, r1    ; r4:r3 = r4:r3 + r0 × r1
SMLAL r3, r4, r0, r1    ; r4:r3 = r4:r3 + r0 × r1
```

# Bitwise Logic

| | |
|---|---|
| **AND** {Rd,} Rn, Op2 | **Bitwise logic AND**<br>Rd ← Rn & operand2 |
| **ORR** {Rd,} Rn, Op2 | **Bitwise logic OR**<br>Rd ← Rn \| operand2 |
| **EOR** {Rd,} Rn, Op2 | **Bitwise logic exclusive OR**<br>Rd ← Rn ^ operand2 |
| **ORN** {Rd,} Rn, Op2 | **Bitwise logic NOT OR**<br>Rd ← Rn \| (NOT operand2) |
| **BIC** {Rd,} Rn, Op2 | **Bit clear**<br>Rd ← Rn & NOT operand2 |
| **BFC** Rd, #lsb, #width | **Bit field clear**<br>Rd[(width+lsb-1):lsb] ← 0 |
| **BFI** Rd, Rn, #lsb, #width | **Bit field insert**<br>Rd[(width+lsb-1):lsb] ← Rn[(width-1):0] |
| **MVN** Rd, Op2 | **Move NOT, logically negate all bits**<br>Rd ← 0xFFFFFFFF EOR Op2 |

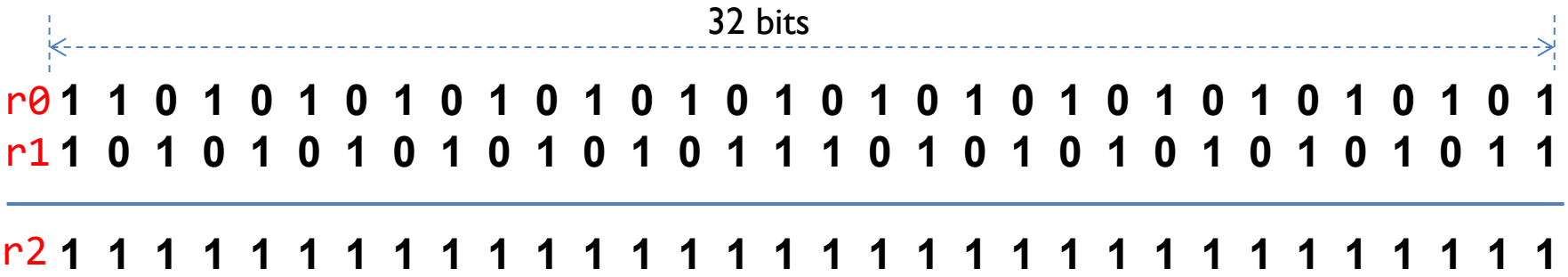
# Example: AND r2, r0, r1




Bit-wise Logic AND

| x | y | x AND y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

# Example: `ORR r2, r0, r1`




Bit-wise Logic OR

| x | y | x OR y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

# Example: `BIC r2, r0, r1`

Bit Clear

**`r2 = r0 & NOT r1`**

Step 1:

```
    r1  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
NOT r1  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0
```

Step 2:

```
    r0  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
NOT r1  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0
-------------------------------------------------------------------------
    r2  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0
```

# Example: BFC and BFI

- Bit Field Clear (**BFC**) and Bit Field Insert (**BFI**).
- Syntax
  - `BFC Rd, #lsb, #width`
  - `BFI Rd, Rn, #lsb, #width`

- Examples:

```
BFC R4, #8, #12
; Clear bit 8 to bit 19 (a total of 12 bits) of R4

BFI R9, R2, #8, #12
; Replace bit 8 to bit 19 (12 bits) of R9
; with bit 0 to bit 11 (12 bits) from R2.
```

# Bit Operators (&, |, ~) *vs* Boolean Operators (&& , | |, !)

| | | | |
|---|---|---|---|
| **A && B** | Boolean and | **A & B** | Bitwise and |
| **A\|\|B** | Boolean or | **A\|B** | Bitwise or |
| **!B** | Boolean not | **~B** | Bitwise not |

- The Boolean operators perform word-wide operations, not bitwise.
- For example,
  - "0x10 & 0x01" = 0x00, but "0x10 && 0x01" = 0x01 (Any nonzero integer is considered true=0x01; false=0x00.)
  - "~0x01" = 0xFFFFFFFE, but "!0x01" = 0x00. (Negation of true=0x01 is false=0x00.)

# Saturating Instruction: SSAT and USAT

- Syntax (for n-bit system):
  - op{cond} Rd, #n, Rm{, shift}
- SSAT saturates a signed value to the signed range $-2^{n-1} \le x \le 2^{n-1} - 1$.

$$SAT(x) = \begin{cases} 2^{n-1} - 1 & if\ x > 2^{n-1} - 1 \\ -2^{n-1} & if\ x < 2^{n-1} \\ x & otherwise \end{cases}$$

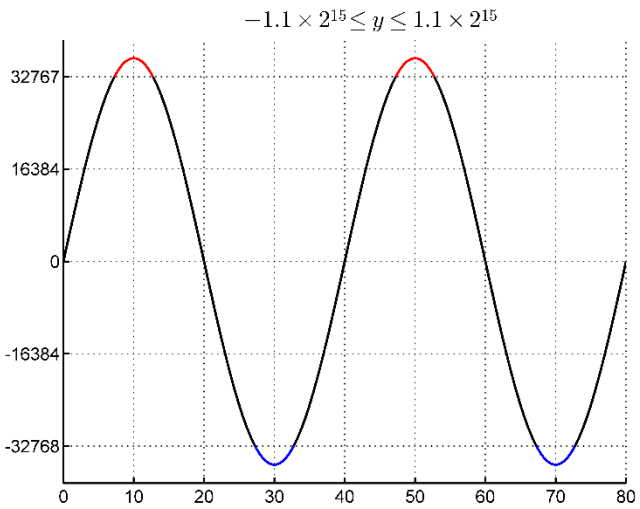
- USAT saturates a signed value to the unsigned range $0 \le x \le 2^n - 1$.

$$USAT(x) = \begin{cases} 2^{n} - 1 & if\ x > 2^{n} - 1 \\ x & otherwise \end{cases}$$

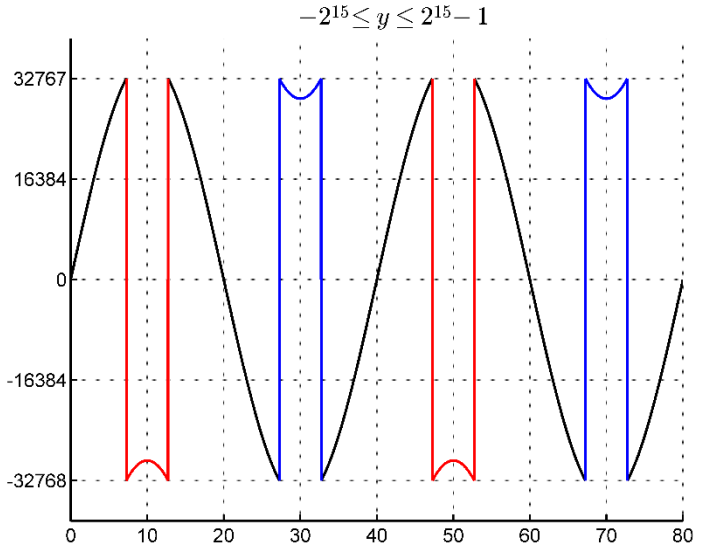
- Examples:
  - `SSAT r2, #11, r1` ; output range: $-2^{10} \le r2 \le 2^{10}$
  - `USAT r2, #11, r3` ; output range: $0 \le r2 \le 2^{11}$

# Example of Saturation

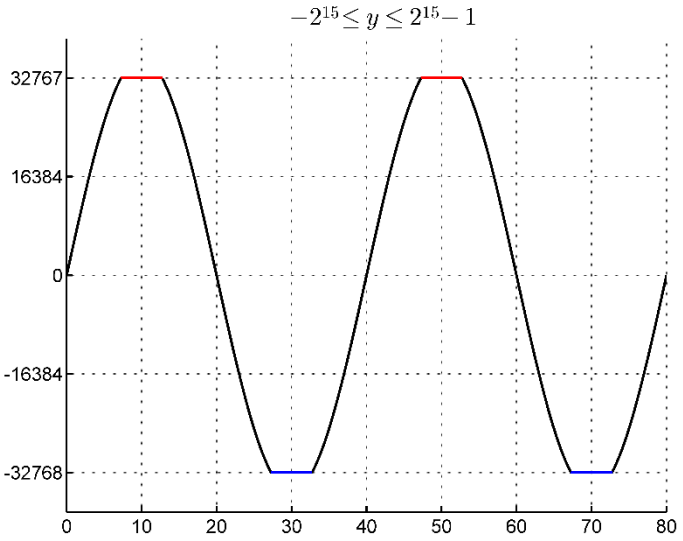
Assume data are limited to **16** bits




Without saturation



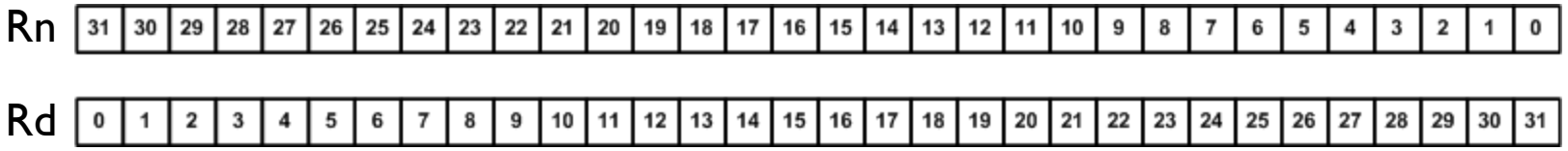

With saturation



# Reverse Order

| | |
|---|---|
| **RBIT** Rd, Rn | **Reverse bit order in a word**<br>for (i = 0; i < 32; i++) Rd[i] ← RN[31– i] |
| **REV** Rd, Rn | **Reverse byte order in a word**<br>Rd[31:24] ← Rn[7:0], Rd[23:16] ← Rn[15:8],<br>Rd[15:8] ← Rn[23:16], Rd[7:0] ← Rn[31:24] |
| **REV16** Rd, Rn | **Reverse byte order in each half-word**<br>Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8],<br>Rd[31:24] ← Rn[23:16], Rd[23:16] ← Rn[31:24] |
| **REVSH** Rd, Rn | **Reverse byte order in bottom half-word and sign extend**<br>Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8],<br>Rd[31:16] ← Rn[7] & 0xFFFF |

**RBIT** Rd, Rn




Example:

```
LDR  r0, =0x12345678  ; r0 = 0x12345678
RBIT r1, r0           ; Reverse bits, r1 = 0x1E6A2C48
```

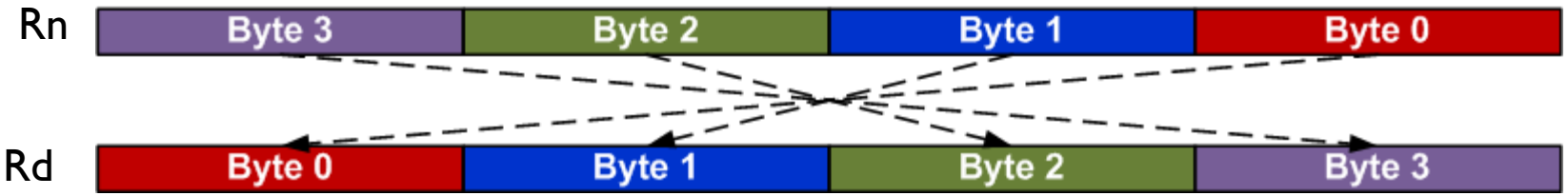
0x12345678 = 0001 0010 0011 0100 0101 0110 0111 1000

0x1E6A2C48 = 0001 1110 0110 1010 0010 1100 0100 1000

# Reverse Order

| | |
|---|---|
| **RBIT** Rd, Rn | **Reverse bit order in a word**<br>for (i = 0; i < 32; i++) Rd[i] ← RN[31– i] |
| **REV** Rd, Rn | **Reverse byte order in a word**<br>Rd[31:24] ← Rn[7:0], Rd[23:16] ← Rn[15:8],<br>Rd[15:8] ← Rn[23:16], Rd[7:0] ← Rn[31:24] |
| REV16 Rd, Rn | Reverse byte order in each half-word<br>Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8],<br>Rd[31:24] ← Rn[23:16], Rd[23:16] ← Rn[31:24] |
| REVSH Rd, Rn | Reverse byte order in bottom half-word and sign extend<br>Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8],<br>Rd[31:16] ← Rn[7] & 0xFFFF |

**REV** Rd, Rn




Example:

```
LDR R0, =0x12345678     ; R0 = 0x12345678
REV R1, R0              ; R1 = 0x78563412
```

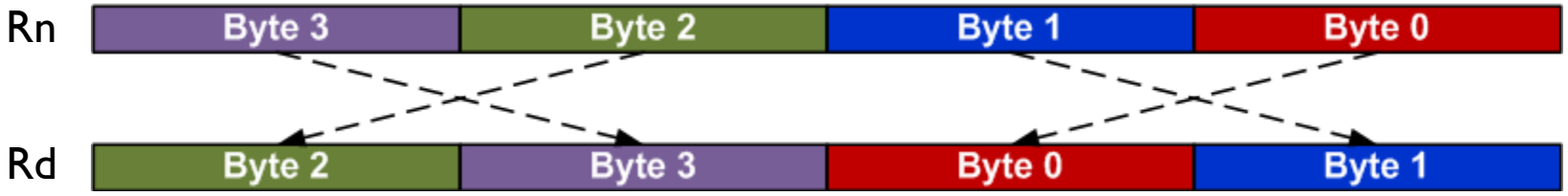
0x12345678 = 0001 0010 0011 0100 0101 0110 0111 1000

0x1E6A2C48 = 0111 1000 0101 0110 0011 0100 0001 0010

# Reverse Order

| | |
|---|---|
| **RBIT** Rd, Rn | **Reverse bit order in a word**<br>for (i = 0; i < 32; i++) Rd[i] ← RN[31- i] |
| **REV** Rd, Rn | **Reverse byte order in a word**<br>Rd[31:24] ← Rn[7:0], Rd[23:16] ← Rn[15:8],<br>Rd[15:8] ← Rn[23:16], Rd[7:0] ← Rn[31:24] |
| **REV16** Rd, Rn | **Reverse byte order in each half-word**<br>Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8],<br>Rd[31:24] ← Rn[23:16], Rd[23:16] ← Rn[31:24] |
| **REVSH** Rd, Rn | **Reverse byte order in bottom half-word and sign extend**<br>Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8],<br>Rd[31:16] ← Rn[7] & 0xFFFF |

**REV16** Rd, Rn




Example:

```
LDR R0, =0x12345678     ; R0 = 0x12345678
REV16 R2, R0            ; R2 = 0x34127856
```

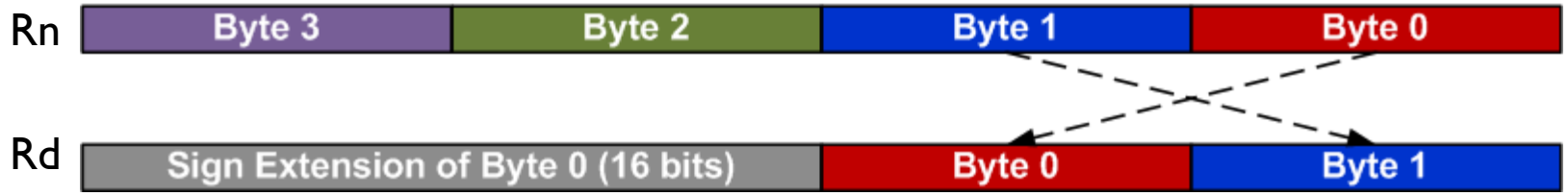
0x12345678 = 0001 0010 0011 0100 0101 0110 0111 1000

0x1E6A2C48 = 0011 0100 0001 0010 0111 1000 0101 0110

# Reverse Order

| | |
|---|---|
| **RBIT** Rd, Rn | **Reverse bit order in a word**<br>for (i = 0; i < 32; i++) Rd[i] ← RN[31– i] |
| **REV** Rd, Rn | **Reverse byte order in a word**<br>Rd[31:24] ← Rn[7:0], Rd[23:16] ← Rn[15:8],<br>Rd[15:8] ← Rn[23:16], Rd[7:0] ← Rn[31:24] |
| **REV16** Rd, Rn | **Reverse byte order in each half-word**<br>Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8],<br>Rd[31:24] ← Rn[23:16], Rd[23:16] ← Rn[31:24] |
| **REVSH** Rd, Rn | **Reverse byte order in bottom half-word and sign extend**<br>Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8],<br>Rd[31:16] ← Rn[7] & 0xFFFF |

**REVSH** Rd, Rn




Example:

```
LDR R0, =0x33448899     ; R0 = 0x33448899
REVSH R1, R0            ; R0 = 0xFFFF9988 %Sign bit of Byte 0 is 1
```

0x12345678 = 0001 0010 0011 0100 0101 0110 0111 1000

0x1E6A2C48 = 1111 1111 1111 1111 1001 1001 1000 1000

# Sign and Zero Extension

```
int8_t  a = -1;    // a signed 8-bit integer,  a = 0xFF
int16_t b = -2;    // a signed 16-bit integer, b = 0xFFFE
int32_t c;         // a signed 32-bit integer

c = a;             // sign extension required, c = 0xFFFFFFFF
c = b;             // sign extension required, c = 0xFFFFFFFE
```

a = 1111 1111

c = a = 1111 1111 1111 1111 1111 1111 1111 1111

b = 1111 1111 1111 1110

c = b = 1111 1111 1111 1111 1111 1111 1111 1110

# Sign and Zero Extension

| | |
|---|---|
| **SXTB** {Rd,} Rm {,ROR #n} | **Sign extend a byte**<br>Rd[31:0] ← Sign Extend((Rm ROR (8 × n))[7:0]) |
| **SXTH** {Rd,} Rm {,ROR #n} | **Sign extend a half-word**<br>Rd[31:0] ← Sign Extend((Rm ROR (8 × n))[15:0]) |
| **UXTB** {Rd,} Rm {,ROR #n} | **Zero extend a byte**<br>Rd[31:0] ← Zero Extend((Rm ROR (8 × n))[7:0]) |
| **UXTH** {Rd,} Rm {,ROR #n} | **Zero extend a half-word**<br>Rd[31:0] ← Zero Extend((Rm ROR (8 × n))[15:0]) |

```
LDR R0, =0x55AA8765
SXTB R1, R0     ; R1 = 0x00000065
SXTH R1, R0     ; R1 = 0xFFFF8765
UXTB R1, R0     ; R1 = 0x00000065
UXTH R1, R0     ; R1 = 0x00008765
```

0x55AA8765 = 0101 0101 1010 1010 1000 0111 0110 0101

0x00000065 = 0000 0000 0000 0000 0000 0000 0110 0101

0xFFFF8765 = 1111 1111 1111 1111 1000 0111 0110 0101

0x00008765 = 0000 0000 0000 0000 1000 0111 0110 0101

# Move Data between Registers

| | |
|---|---|
| **MOV** | Rd ← operand2 |
| **MVN** | Rd ← NOT operand2 |
| **MRS** Rd, spec_reg | Move from special register to general register |
| **MSR** spec_reg, Rm | Move from general register to special register |

```
MOV r4, r5              ; Copy r5 to r4
MVN r4, r5              ; r4 = bitwise logical NOT of r5
MOV r1, r2, LSL #3      ; r1 = r2 << 3
MOV r0, PC              ; Copy PC (r15) to r0
MOV r1, SP              ; Copy SP (r14) to r1
```

# Move Immediate Number to Register

| | |
|---|---|
| `MOVW Rd, #imm16` | **Move Wide**, Rd ← #imm16 |
| `MOVT Rd, #imm16` | **Move Top**, Rd ← #imm16 << 16 |
| `MOV Rd, #const` | **Move**, Rd ← const |

Example: Load a 32-bit number into a register

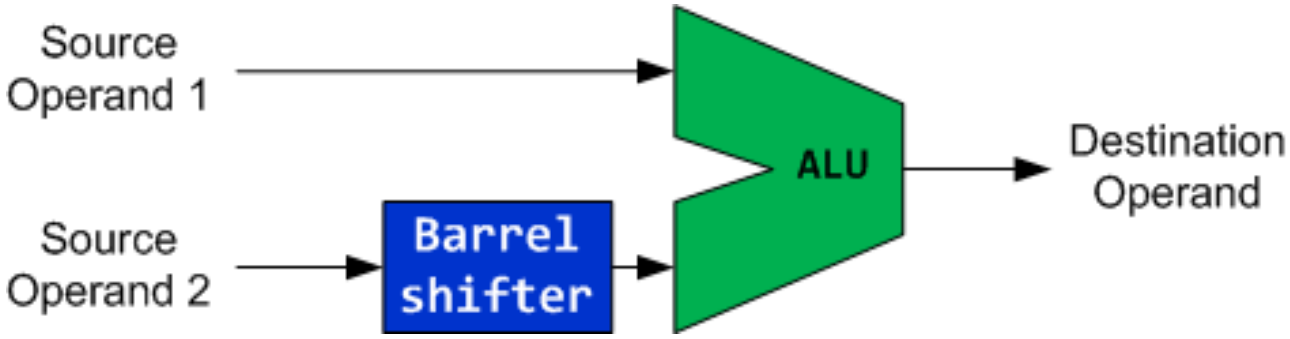
```
MOVW r0, #0x4321     ; r0 = 0x00004321
MOVT r0, #0x8765     ; r0 = 0x87654321
```

Order does matter!

- **MOVW** writes 16-bit immediate into the low half of r0, and zeros upper halfword
- **MOVT** writes 16-bit immediate into the high half of r0 without altering the low halfword

```
MOVT r0, #0x8765     ; r0 = 0x8765xxxx
MOVW r0, #0x4321     ; r0 = 0x00004321
```

# Barrel Shifter




- The second operand of ALU has a special hardware called Barrel shifter
- Five shift and rotation operations:
  - `LSL, LSR, ASR, ROR, RRX`
- Use Barrel shifter to speed up multiplication and division
  - Shifting left 1 bit multiplies a number by 2
- r1, LSL #3 means "r1 shifted left by 3 bits," i.e., r1 << 3, which is the same as multiplying r0 by 8.

```
ADD r3, r2, r1, LSL #3  ; r3 = r2 + r1 << 3
```

# Use shifts to implement multiplication and division

- Examples. r1 = r0 + (r0 << 3) = r0 + 8 × r0 = 9 × r0
  - r1 = 9 × r0

```
ADD r1, r0, r0, LSL #3  <=>  MOV r2, #9      ; r2 = 9
                             MUL r1, r0, r2 ; r1 = r0 * 9
```

MUL instruction takes only registers, not an immediate, so "MUL r1, r0, #9" is invalid syntax

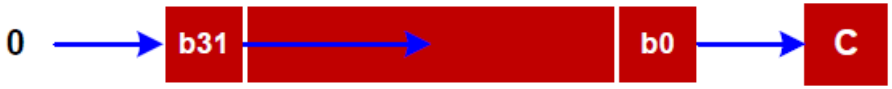
```
ADD r1, r0, r0, LSR #3
; r1 = r0 + r0 >> 3 = r0 + r0/8 (unsigned)

ADD r1, r0, r0, ASR #3
; r1 = r0 + r0 >> 3 = r0 + r0/8 (signed)
```

# Barrel Shifter

Logical Shift Left (**LSL**)

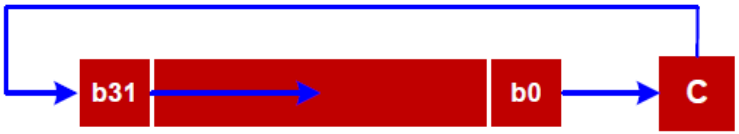

Arithmetic Shift Right (**ASR**)


Logical Shift Right (**LSR**)




Rotate Right (**ROR**)


Rotate Right Extended (**RRX**)




Why is there rotate right but no rotate left?

Rotate left can be replaced by a rotate right with a different rotate offset, e.g., ROL x, n equals ROR x, 32−n for 32-bit values

# Barrel Shifter: Explanations

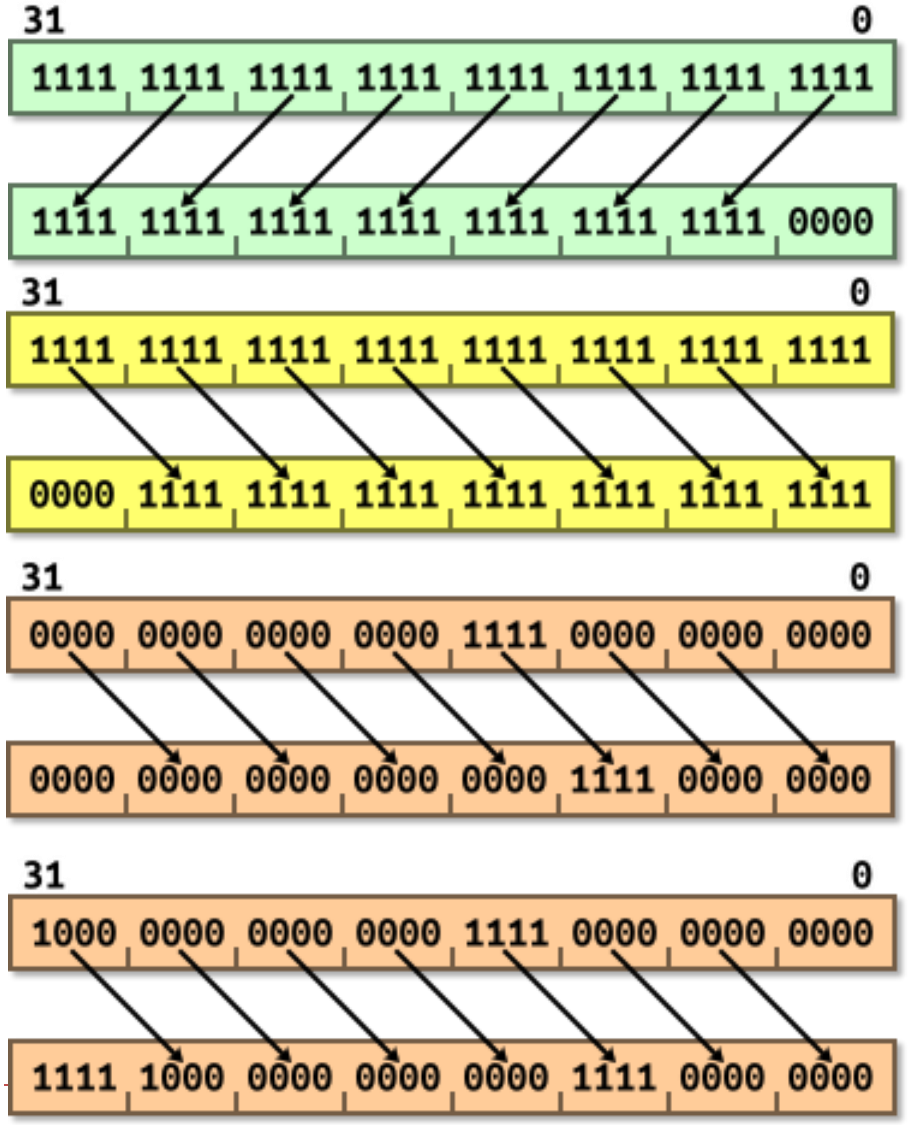
- LSL (logical shift left): shifts left, fills zeros on the right; C gets the last bit shifted out of bit 31. This is multiply by $2^n$ for non-overflowing values.
- LSR (logical shift right): shifts right, fills zeros on the left; C gets the last bit shifted out of bit 0. This is unsigned division by $2^n$.
- ASR (arithmetic shift right): shifts right, fills the sign bit on the left to preserving the sign; C gets the last bit shifted out of bit 0. This is signed division by $2^n$ with sign extension
- ROR (rotate right): rotates bits right with wraparound; bits leaving bit 0 re-enter at bit 31, and C receives the bit that wrapped. This is a pure rotation without data loss.
- RRX (rotate right extended): rotates right by one through the carry flag, treating C as a 33rd bit; new bit 31 comes from old C, and C receives old bit 0.

# Examples: shifting by 4

Logical Shift Left (**LSL**)

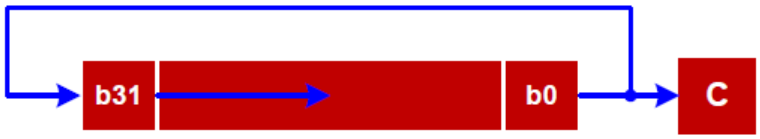

Logical Shift Right (**LSR**)

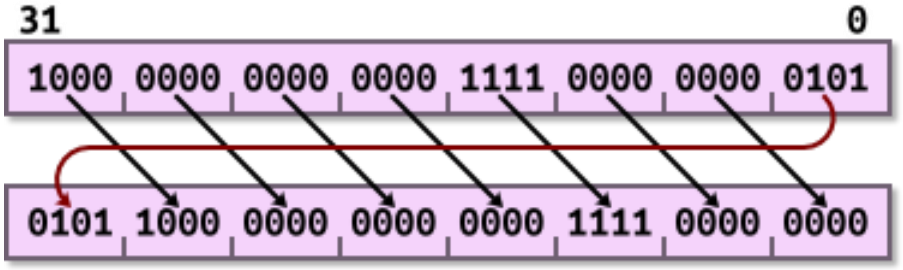

Arithmetic Shift Right (**ASR**)

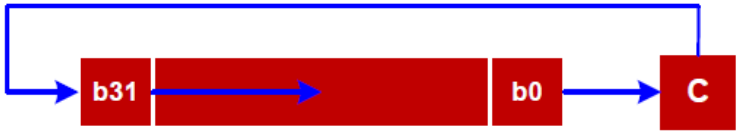
# Examples: shifting by 4

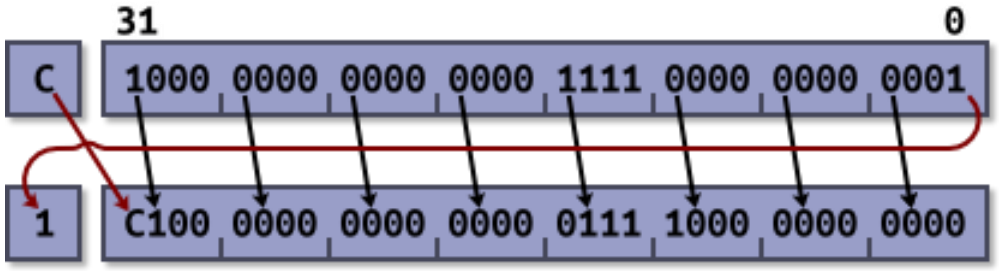
Rotate Right (**ROR**)




Rotate Right Extended (**RRX**)



# Barrel Shifter Examples

- MOV r0, r0, LSL #1
  - Multiply R0 by two.
- MOV r1, r1, LSR #2
  - Divide R1 by four (unsigned).
- MOV r2, r2, ASR #2
  - Divide R2 by four (signed).
- MOV r3, r3, ROR #16
  - Swap the top and bottom halves of R3.
- ADD r4, r4, r4, LSL #4
  - Multiply R4 by 17. (N = N + N * 16)
- RSB r5, r5, r5, LSL #5
  - Multiply R5 by 31. (N = N * 32 - N)
  - Reverse-subtract using barrel shifter on the second operand, giving r5 = (r5 << 5) − r5
- SUB r5, r5, r5, LSR #5
  - Subtract using barrel shifter on the second operand, giving r5 − (r5 >> n)

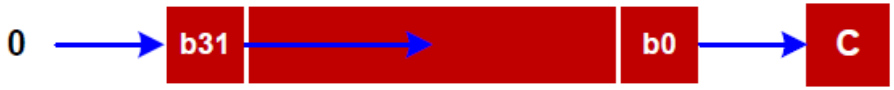
# Updating APSR Flags

- If "S" is present, the instruction update flags. Otherwise, the flags are not updated.
- Let **R** be the final 32-bit result

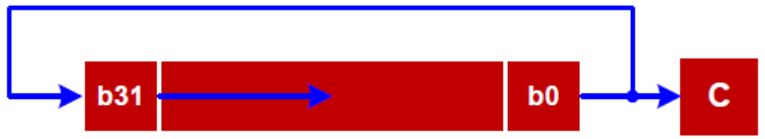
| N | Z | C | V |
|---|---|---|---|
| R<31> | IsZeroBit(R) | carry | unchanged |


LSLS


ASRS




LSRS




RORS

# Carry Flag Update

- The carry flag captures the last bit shifted out of the register during shift operations, when the instruction explicitly updates flags (using S suffix like MOVS, LSLS, ASRS)
  - LSLS (Logical Shift Left): Carry = the last bit shifted out from the MSB position (bit 31)
  - LSRS (Logical Shift Right), ASRS (Arithmetic Shift Right): Carry = the last bit shifted out from the LSB position (bit 0)
  - RORS (Rotate Right): Carry = bit that was rotated from bit 0 to bit 31

# Example 1: ANDS

```
LDR   r0, =0xFFFFFF00
LDR   r1, =0x00000001
ANDS  r2, r1, r0, LSL #1
```

Update carry flag C=1

**N = 0, Z = 1, C = 1, V = 0**

**AND{S}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}**

r0 = 0xFFFFFF00

r1 = 0x00000001

r0, LSL #1 = 0xFFFFFE00

r2 = r1 AND (r0 << 1) = 0x00000001 AND 0xFFFFFE00 = 0x00000000

ANDS sets flags:

Z = 1 (result r2 is zero)

N = 0 (bit 31 of result r2 is 0)

C is unaffected by ANDS. It was set be previous shift to be C=1

V is unaffected by ANDS. Assume it was 0 before.

# Example 2: ADDS

```
LDR   r0, =0xFFFFFF00
LDR   r1, =0x00000001
ADDS  r2, r1, r0, LSL #1
```

Does NOT update carry flag

**N = 1, Z = 0, C = 0, V = 0**

**ADD{S}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}**

r0 = 0xFFFFFF00

r1 = 0x00000001

r0, LSL #1 = 0xFFFFFE00

r2 = r1 + (r0 << 1) = 0x00000001 + 0xFFFFFE00 = 0xFFFFFE01

ADDS sets flags:

Z = 0 (result r2 is non-zero)

N = 1 (bit 31 of result r2 is 1)

C = 0 (there is no carry out from bit 31 for unsigned addition, when adding 0x00000001 and 0xFFFFFE00)

V=0 is (there is no overflow for signed addition, when adding 0x00000001 and 0xFFFFFE00. Recall: adding a positive (1) to a negative (0xFFFFFE00) cannot cause overflow.)

# Set a Bit in C

```
a |= (1 << k)
```

**or**

```
a = a | (1 << k)
```

**Example: k = 5**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **a** | $a_7$ | $a_6$ | **$a_5$** | $a_4$ | $a_3$ | $a_2$ | $a_1$ | $a_0$ |
| **1 << k** | 0 | 0 | **1** | 0 | 0 | 0 | 0 | 0 |
| **a \| (1 << k)** | $a_7$ | $a_6$ | **1** | $a_4$ | $a_3$ | $a_2$ | $a_1$ | $a_0$ |

*The other bits should not be affected.*

# Set a Bit in Assembly

**a |= (1 << 5)**

**Solution 1:**

```
    MOVS r4, #1         ; r4 = 1
    LSLS r4, r4, #5     ; r4 = 1<<5
    ORRS r0, r0, r4     ; r0 = r0 | 1<<5
```

**Solution 2:**

```
    MOVS r4, #1                 ; r4 = 1
    ORRS r0, r0, r4, LSL #5     ; r0 = r0 | 1<<5
```

# Clear a Bit in C

```
a &= ~(1<<k)
```

**Example: k = 5**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **a** | $a_7$ | $a_6$ | $a_5$ | $a_4$ | $a_3$ | $a_2$ | $a_1$ | $a_0$ |
| **~(1 << k)** | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| **a & ~(1<<k)** | $a_7$ | $a_6$ | 0 | $a_4$ | $a_3$ | $a_2$ | $a_1$ | $a_0$ |

*The other bits should not be affected.*

# Clear a Bit in Assembly

**a &= ~(1<<5)**

**Solution 1:**

```
    MOVS r4, #1          ; r4 = 1
    LSLS r4, r4, #5      ; r4 = 1<<5 = 0x20
    MVNS r4, r4          ; r4 = not (1<<5)
    ANDS r0, r0, r4      ; r0 = r0 & not (1<<5)
```

**Solution 2:**

```
    MOVS r4, #1              ; r4 = 1
    MVNS r4, r4, LSL #5      ; r4 = not (1<<5)
    ANDS r0, r0, r4          ; r0 = r0 & not (1<<5)
```

**Solution 3:**

```
    MOVS r4, #1                  ; r4 = 1
    BICS r0, r0, r4, LSL #5      ; r0 = r0 & not (1<<5)
                                 ; BIC (Bit Clear) clears
                                 bit 5 of r0 (from r4=0x20)
```

# Toggle a Bit in C

Without knowing the initial value, a bit can be toggled by XORing it with a "1"

```
a ^= 1<<k
```

**Example:** k = 5

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **a** | $a_7$ | $a_6$ | $a_5$ | $a_4$ | $A_3$ | $a_2$ | $a_1$ | $a_0$ |
| **1 << k** | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| **a ^= 1<<k** | $a_7$ | $a_6$ | NOT($a_5$) | $A_4$ | $a_3$ | $a_2$ | $a_1$ | $a_0$ |

| x | y | x XOR y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| $a_5$ | 1 | $a_5 \oplus 1$ |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 1 | 0 |

Truth table of Exclusive OR with one

# Toggle a Bit in Assembly

**a ^= 1<<5**

**Solution:**

```
    MOVS r4, #1                 ; r4 = 1
    EORS r0, r0, r4, LSL #5     ; r0 = r0 ^ 1<<5
```

# References

- Lecture 25. Arithmetic and Logical Instructions
  - https://www.youtube.com/watch?v=H-vOP2yRUj4&list=PLRJhV4hUhlymmp5CCeIFPyxbknsdcXCc8&index=25
- Lecture 26. Updating NZCV bit flags
  - https://www.youtube.com/watch?v=SGJibM1D2_A&list=PLRJhV4hUhlymmp5CCeIFPyxbknsdcXCc8&index=26