

Chapter 4

ARM Arithmetic and Logic Instructions

Z. Gu

Fall 2025

Adding Two Integers

```
int x = 1;  
int y = 2;  
int z;
```

C Statement

```
z = x + y;
```

If values are in registers

- ▶ Value of *x* in *r0*
- ▶ Value of *y* in *r1*
- ▶ Value of *z* in *r2*

Assembly Statement

?

Adding Two Integers

```
int x = 1;  
int y = 2;  
int z;
```

C Statement

```
z = x + y;
```

If values are in registers

- ▶ Value of **x** in **r0**
- ▶ Value of **y** in **r1**
- ▶ Value of **z** in **r2**

Assembly Statement

```
ADD r2, r1, r0
```

Destination

Source Operand 2

Source Operand 1

Adding Two Integers

```
uint x = 1;  
uint y = 2;  
uint z;
```

C Statement

```
z = x + y;
```

If values are in registers

- ▶ Value of **x** in **r0**
- ▶ Value of **y** in **r1**
- ▶ Value of **z** in **r2**

Assembly Statement

?

Adding Two Integers

```
uint x = 1;  
uint y = 2;  
uint z;
```

C Statement

```
z = x + y;
```

If values are in registers

- ▶ Value of **x** in **r0**
- ▶ Value of **y** in **r1**
- ▶ Value of **z** in **r2**

Assembly Statement

```
ADD r2, r1, r0
```

ADD works for both signed and unsigned add operations.

Adding Two Integers

```
int x = 1;  
int y = 2;  
int z;
```

C Statement

```
z = x + y;
```

If addresses are in registers

- ▶ Address of x in r0
- ▶ Address of y in r1
- ▶ Address of z in r2

Assembly Statements

?

Adding Two Integers

```
int x = 1;  
int y = 2;  
int z;
```

C Statement

```
z = x + y;
```

If addresses are in registers

- ▶ Address of x in r0
- ▶ Address of y in r1
- ▶ Address of z in r2

Assembly Statements

```
LDR r3, [r0] ; Read x  
LDR r4, [r1] ; Read y  
ADD r5, r3, r4  
STR r5, [r2] ; Write z
```

Load, modify, and store

Example Arithmetic Instructions

- ▶ **ADD** `r0, r1, r2` ; $r0 = r1 + r2$
- ▶ **ADC** `r0, r1, r2` ; Add with carry, $r0 = r1 + r2 + \text{carry}$
- ▶ **SUB** `r0, r1, r2` ; $r0 = r1 - r2$
- ▶ **SBC** `r0, r1, r2` ; Subtract with borrow, $r0 = r1 - r2 - (1 - \text{carry})$
- ▶ **MUL** `r0, r1, r2` ; $r0 = r1 * r2$, product limited to 32 bits
- ▶ **UDIV** `r0, r1, r2` ; Unsigned divide, $r0 = r1 / r2$
- ▶ **SDIV** `r0, r1, r2` ; Signed divide, $r0 = r1 / r2$
- ▶ **SMULL** `r0, r1, r2, r3` ; Signed multiply (64-bit product), $r1:r0 = r2 * r3$
- ▶ **UMULL** `r0, r1, r2, r3` ; Unsigned multiply (64-bit product), $r1:r0 = r2 * r3$

Example Logical Instructions

- ▶ **AND** r0, r1, r2 ; Bitwise AND, $r0 = r1 \text{ AND } r2$
- ▶ **ORR** r0, r1, r2 ; Bitwise OR, $r0 = r1 \text{ OR } r2$
- ▶ **EOR** r0, r1, r2 ; Bitwise Exclusive OR, $r0 = r1 \text{ EOR } r2$
- ▶ **ORN** r0, r1, r2 ; Bitwise OR NOT, $r0 = r1 \text{ ORN } r2$
- ▶ **BIC** r0, r1, r2 ; Bit clear, $r0 = r1 \& \sim r2$

AND r0, r1, r2

32 bits

r1	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
r2	1	0	1	0	1	0	1	0	1	0	1	0	1	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1
r0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Bit-wise Logic **AND**

Example Shift & Rotate Instructions

- ▶ **LSL** `r0, r1, r2` ; Logical shift left,
 $r0 = r1 \ll r2$
- ▶ **LSR** `r0, r1, r2` ; Logical shift right,
 $r0 = r1 \gg r2$
- ▶ **ASR** `r0, r1, r2` ; Arithmetic shift right,
 $r0 = r1 \ggg r2$
- ▶ **ROR** `r0, r1, r2` ; Rotate right,
 $r0 = r1 \text{ rotate by } r2 \text{ bits}$
- ▶ **RRX** `r0, r1, r2` ; Extended rotate right,
 $\{C, r0\} = \{C, r1\} \text{ rotate by } r2 \text{ bits}$

Logical Shift Left (**LSL**)



Logical Shift Right (**LSR**)



Arithmetic Shift Right (**ASR**)



Rotate Right (**ROR**)



Rotate Right Extended (**RRX**)



Example Data Transfer Instructions

- ▶ **MOV** r0, r1 ; Move, r0 = r1
- ▶ **MVN** r0, r1 ; Move NOT, r0 = bitwise NOT r1

MVN r0, r1

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	1	1	1	1
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	0	0	0	0

Bit-wise Logic **NOT**

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 $Rd \leftarrow Rn + Op2$
ADC {Rd,} Rn, Op2	Add with carry $Rd \leftarrow Rn + Op2 + Carry$
SUB {Rd,} Rn, Op2	Subtract $Rd \leftarrow Rn - Op2$
SBC {Rd,} Rn, Op2	Subtract with carry $Rd \leftarrow Rn - Op2 + Carry - 1$
RSB {Rd,} Rn, Op2	Reverse subtract $Rd \leftarrow Op2 - Rn$
MUL {Rd,} Rn, Rm	Multiply $Rd \leftarrow (Rn \times Rm)[31:0]$
MLA Rd, Rn, Rm, Ra	Multiply with accumulate $Rd \leftarrow (Ra + (Rn \times Rm))[31:0]$
MLS Rd, Rn, Rm, Ra	Multiply and subtract $Rd \leftarrow (Ra - (Rn \times Rm))[31:0]$
SDIV {Rd,} Rn, Rm	Signed divide $Rd \leftarrow Rn \div Rm$
UDIV {Rd,} Rn, Rm	Unsigned divide $Rd \leftarrow Rn \div Rm$
SSAT Rd, #n, Rm {,shift #s}	Signed saturate
USAT Rd, #n, Rm {,shift #s}	Unsigned saturate



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)

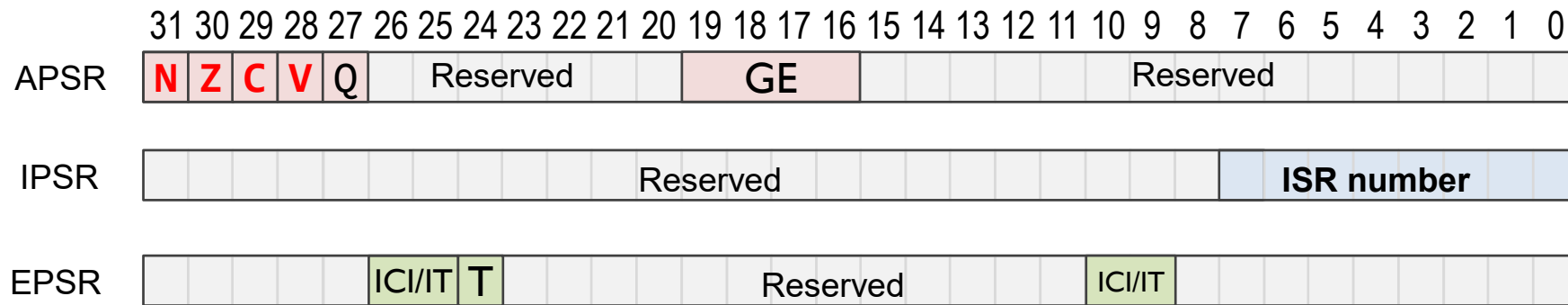


CPSR (Current Program Status Register)

- Four flag bits:
 - N (negative), Z (zero), C (carry), V (overflow).

Program Status Register (PSR)

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



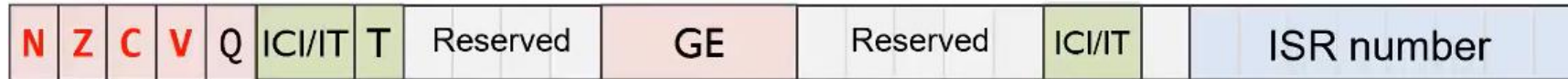
Combine them together into one register (**PSR**)



Note:

- GE flags are only available on Cortex-M4 and M7
- Use PSR in code

NZCV Flags in xPSR



- ▶ **N**: I/O = Result from ALU is **N**egative/positive
- ▶ **Z**: I/O = Result from ALU is **Z**ero/non-zero
- ▶ **C**: Three cases (for unsigned arithmetic) :
 - ▶ I/O = ALU addition **C**arry out/no carry out
 - ▶ I/O = ALU subtraction no borrow/borrow
 - ▶ I/O = Bit shifted/rotated out
- ▶ **V**: I/O = ALU o**V**erflowed/no overflow (for signed arithmetic)

Borrow and carry share the same flag bit.
For unsigned subtract,
Borrow = NOT Carry

ALU performs the same addition whether the numbers are signed or unsigned. It just adds bits.
Afterward: C tells you about unsigned overflow. V tells you about signed overflow. Same hardware. Different interpretation.

Updating NZCV flags in PSR

Flags not changed		Flags updated
ADD	→	ADD S
SUB	→	SUB S
MUL	→	MUL S
UDIV	→	UDIV S
AND	→	AND S
ORR	→	ORR S
LSL	→	LSL S
MOV	→	MOV S

`CMP r1, r2` vs `SUBS r0, r1, r2`

Some instructions update NZCV flags even if no S is specified.

- **CMP**: Compare, like SUBS but without destination register
- **CMN**: Compare Negative, like ADDS but without destination register
- **TST**: Test, like ANDS but without destination register
- **TEQ**: Test equivalence, like EORS but without destination register

*Most instructions update NZCV flags
only if S suffix is present*

ADD *vs* ADDS

ADD r0, r1, r2 ; r0 = r1 + r2, NZCV flags unchanged

ADDS r0, r1, r2 ; r0 = r1 + r2, NZCV flags updated

- ▶ ADD does not update flags
- ▶ ADDS updates flags
 - ▶ xPSR.**N** = bit 31 of result (N = 1 indicates result is negative, vice versa.)
 - ▶ xPSR.**Z** = IsZero(result)
 - ▶ xPSR.**C** = carry, assuming r1 and r2 representing unsigned integers
 - ▶ xPSR.**V** = overflow, assuming r1 and r2 representing signed integers

Suffix S: Update Flags

```
LDR  r0, =0xFFFFFFFF
LDR  r1, =0x00000001
ADDS r0, r0, r1
```

0xFFFFFFFF	r0
+ 0x00000001	r1
0x00000000	sum

N (Negative) = 0
Z (Zero) = 1
C (Carry) = 1
V (oVerflow) = 0

The screenshot shows a disassembler interface with two main panes. The left pane, titled 'Registers', displays the state of various registers. The right pane, titled 'Disassembly', shows the assembly code being analyzed.

Registers Pane:

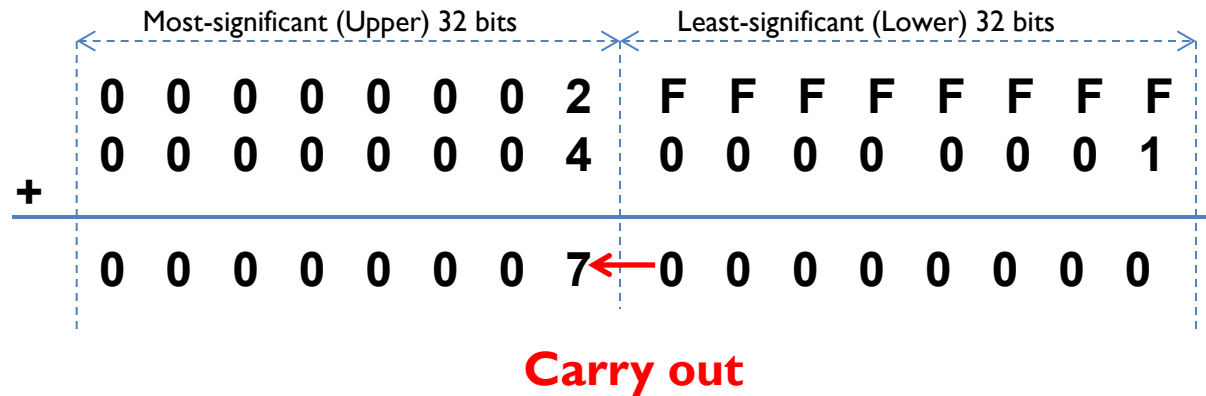
Register	Value
Core	
R0	0xFFFFFFFF
R1	0x00000001
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000600
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x08000136
xPSR	
N	0
Z	1
C	1
V	0
Q	0
T	1
IT	Disabled
ISR	0
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	8
Sec	0.00000100

Disassembly Pane:

```
29:                                ADDS r3, r0, r1
30:
0x08000134 1843      ADDS      r3,r0,r1
31: stop                      B      stop
0x08000136 E7FE      B      0x08000136
0x08000138 0000      MOVS      r0,r0
0x0800013A 0000      MOVS      r0,r0
0x0800013C 0000      MOVS      r0,r0

main.s  stm32l1xx_constants.s  startup_stm32l1xx_md.s
1  ;***** (C) Yifeng ZHU *****
2  ; @file    main.s
3  ; @author  Yifeng Zhu
4  ;*****
5
6          INCLUDE stm32l1xx_constants.s
7
8          AREA    main, CODE, READONLY
9          EXPORT  __main
10         ENTRY
11
12         __main      PROC
13
14         LDR  r0, =0xFFFFFFFF
15         LDR  r1, =0x00000001
16         ADDS r3, r0, r1
17
18         stop      B      stop
19
20         ENDP
21         ALIGN
22         END
```

Example: 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

Example: 64-bit Addition

start

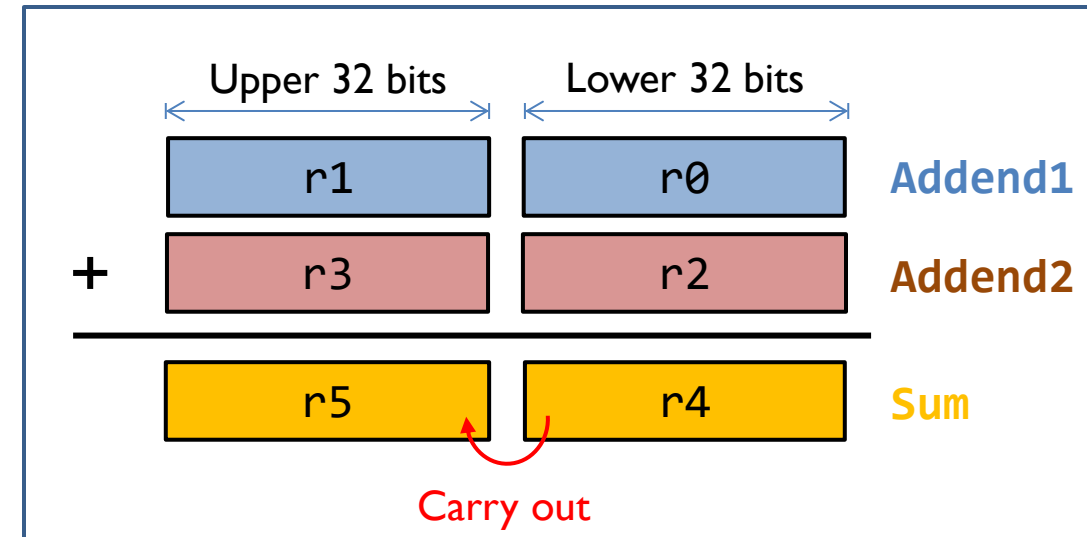
```
; C = A + B
; Two 64-bit integers A (r1,r0) and B (r3,r2).
; Result C (r5, r4)
; A = 00000002FFFFFFFF
; B = 0000000400000001
LDR  r0, =0xFFFFFFFF ; A's lower 32 bits
LDR  r1, =0x00000002   ; A's upper 32 bits
LDR  r2, =0x00000001   ; B's lower 32 bits
LDR  r3, =0x00000004   ; B's upper 32 bits
```

```
; 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



Example: 64-bit Subtraction

start

```
; C = A - B
; Two 64-bit integers A (r1,r0) and B (r3,r2).
; Result C (r5, r4)
; A = 00000002FFFFFFFF
; B = 0000000400000001
```

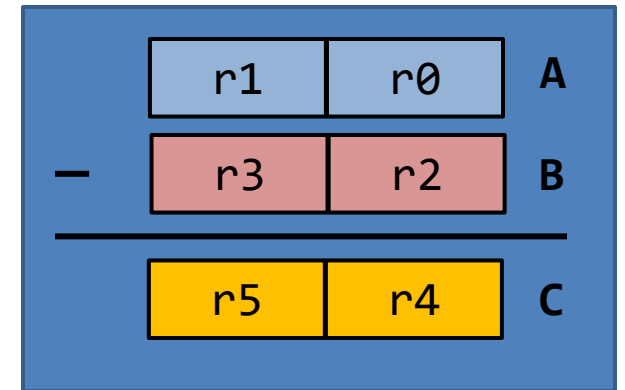
```
LDR  r0, =0xFFFFFFFF ; A's lower 32 bits
LDR  r1, =0x00000002  ; A's upper 32 bits
LDR  r2, =0x00000001  ; B's lower 32 bits
LDR  r3, =0x00000004  ; B's upper 32 bits
```

```
; Subtract B from A
```

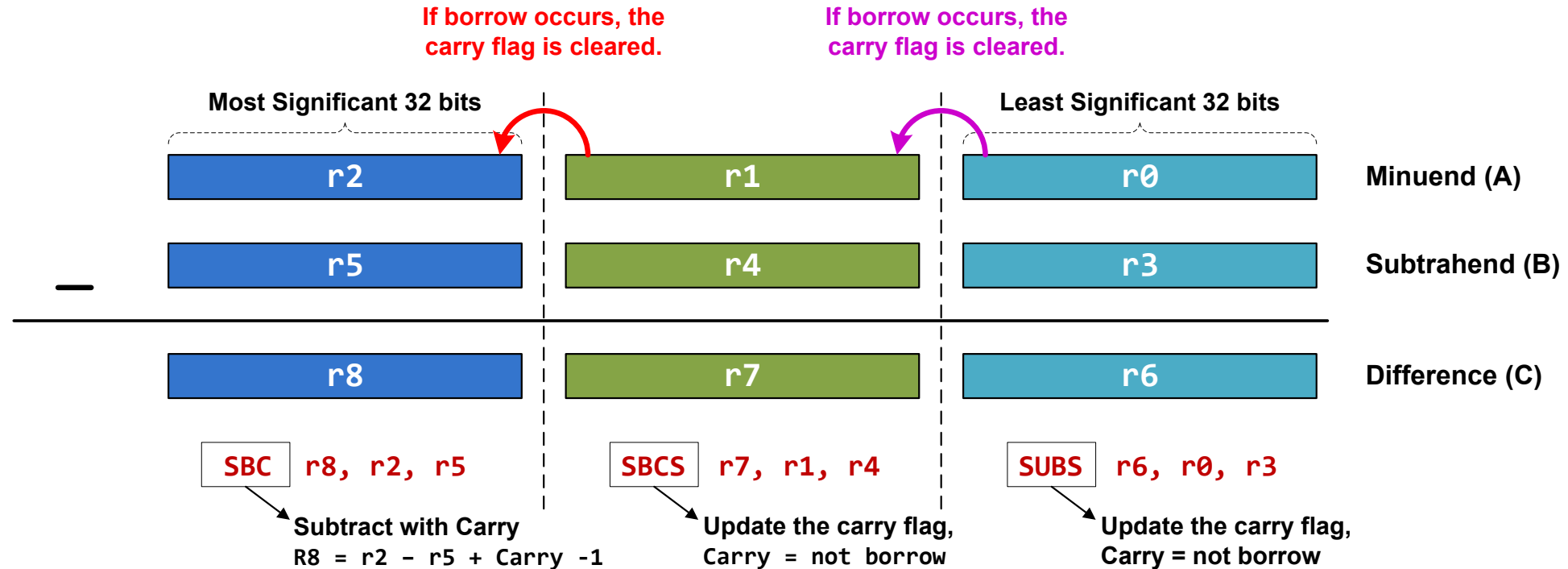
```
SUBS r4, r0, r2 ; C[31..0] = A[31..0] - B[31..0], update Carry (= 1 - Borrow)
```

```
SBC  r5, r1, r3 ; C[64..32] = A[64..32] - B[64..32] - (1 - Carry)
```

stop B stop



Example: 96-bit Subtraction



SUBS r6, r0, r3 (Sets flag Borrow = 1 - Carry)

SBCS r7, r1, r4 (Sets flag Borrow = 1 - Carry)

SBC r8, r2, r5 (SBCS r8, r2, r5 is also OK, but the Carry flag is not used by any other instruction)

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 $\text{RdHi}, \text{RdLo} \leftarrow \text{unsigned}(\text{Rn} \times \text{Rm})$
SMULL RdLo, RdHi, Rn, Rm	Signed long multiply $\text{RdHi}, \text{RdLo} \leftarrow \text{signed}(\text{Rn} \times \text{Rm})$
UMLAL RdLo, RdHi, Rn, Rm	Unsigned multiply with accumulate $\text{RdHi}, \text{RdLo} \leftarrow \text{unsigned}(\text{RdHi}, \text{RdLo} + \text{Rn} \times \text{Rm})$
SMLAL RdLo, RdHi, Rn, Rm	Signed multiply with accumulate $\text{RdHi}, \text{RdLo} \leftarrow \text{signed}(\text{RdHi}, \text{RdLo} + \text{Rn} \times \text{Rm})$

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 $Rd \leftarrow Rn \& \text{operand2}$
ORR {Rd,} Rn, Op2	Bitwise logic OR $Rd \leftarrow Rn \mid \text{operand2}$
EOR {Rd,} Rn, Op2	Bitwise logic exclusive OR $Rd \leftarrow Rn \wedge \text{operand2}$
ORN {Rd,} Rn, Op2	Bitwise logic NOT OR $Rd \leftarrow Rn \mid (\text{NOT } \text{operand2})$
BIC {Rd,} Rn, Op2	Bit clear $Rd \leftarrow Rn \& \text{NOT } \text{operand2}$
BFC Rd, #lsb, #width	Bit field clear $Rd[(\text{width}+\text{lsb}-1):\text{lsb}] \leftarrow 0$
BFI Rd, Rn, #lsb, #width	Bit field insert $Rd[(\text{width}+\text{lsb}-1):\text{lsb}] \leftarrow Rn[(\text{width}-1):0]$
MVN Rd, Op2	Move NOT, logically negate all bits $Rd \leftarrow 0xFFFFFFFF \text{ EOR } \text{Op2}$

Example: AND r2, r0, r1

32 bits

r0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
r1	1	0	1	0	1	0	1	0	1	0	1	0	1	1	1	0	1	0	1	0	1	0	1	0	1	0	1	1
r2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1

Bit-wise Logic AND

Example: ORR r2, r0, r1

32 bits

r0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
r1	1	0	1	0	1	0	1	0	1	0	1	0	1	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1
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	1	

Bit-wise Logic OR

Example: BIC r2, r0, r1

Bit Clear

$$r2 = r0 \& \text{NOT } r1$$

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	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	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	
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	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	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 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. (true && true = true, any non-zero value is logical true, e.g., 0x10 and 0x01 are both true)
 - ▶ “~0x01” = 0xFFFFFFFF, but “!0x01” = 0x00. (!true = false)
 - ▶ 0x01 as 32-bit register is actually 0x00000001, so ~0x01 = 0xFFFFFFFF

Saturating Instruction: **SSAT** and **USAT**

- ▶ Syntax:

- ▶ `op{cond} Rd, #n, Rm{, shift}`

- ▶ **SSAT** saturates a signed value to the signed range $-2^{n-1} \leq x \leq 2^{n-1} - 1$.

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

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

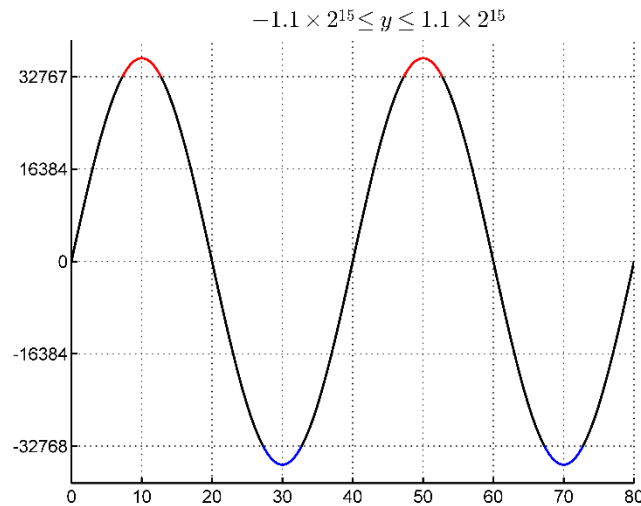
$$USAT(x) = \begin{cases} 2^n - 1 & \text{if } x > 2^n - 1 \\ x & \text{otherwise} \end{cases}$$

- ▶ Examples:

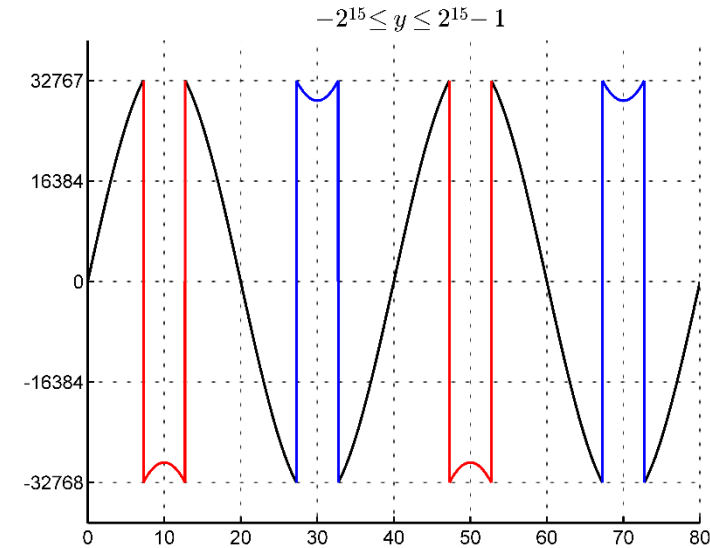
- ▶ `SSAT r2, #11, r1` ; output range: $-2^{10} \leq r2 \leq 2^{10}$
 - ▶ `USAT r2, #11, r3` ; output range: $0 \leq r2 \leq 2^{11}$

Example of Saturation

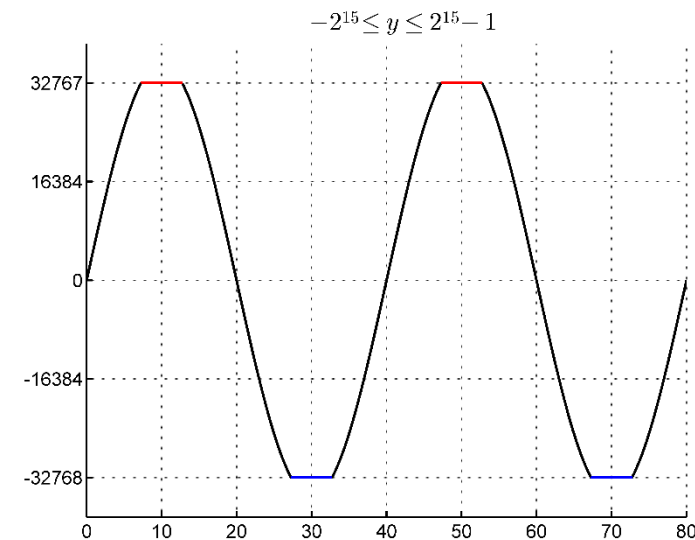
Assume data are limited to **16** bits



Without
saturation



With
saturation



Reverse Order

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

RBIT Rd, Rn

Rn

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	---	---	---	---	---	---	---	---	---	---

Rd

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Example:

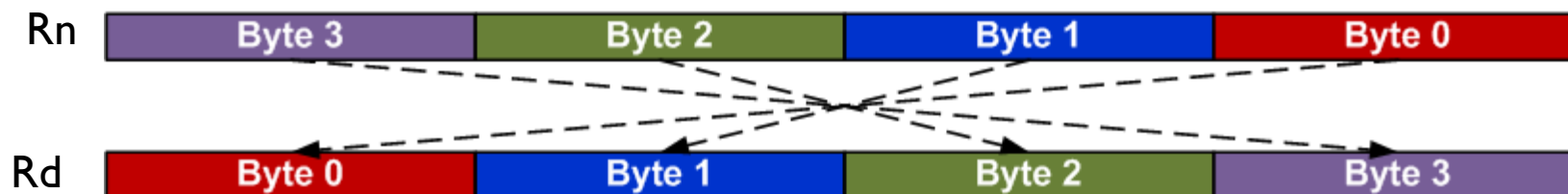
```
LDR  r0, =0x12345678 ; r0 = 0x12345678
RBIT r1, r0           ; Reverse bits, r1 = 0x1E6A2C48
                        ; Note r1 != 0x87654321
```



Reverse Order

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

REV Rd, Rn



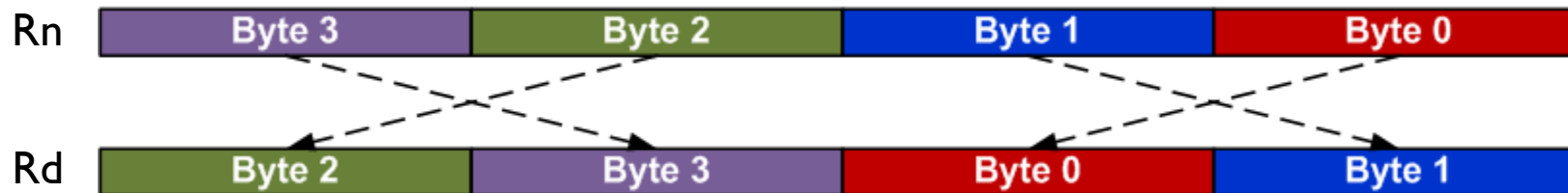
Example:

```
LDR R0, =0x12345678    ; R0 = 0x12345678
REV R1, R0               ; R1 = 0x78563412
                        ; Note r1 != 0x87654321
```

Reverse Order

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

REV16 Rd, Rn



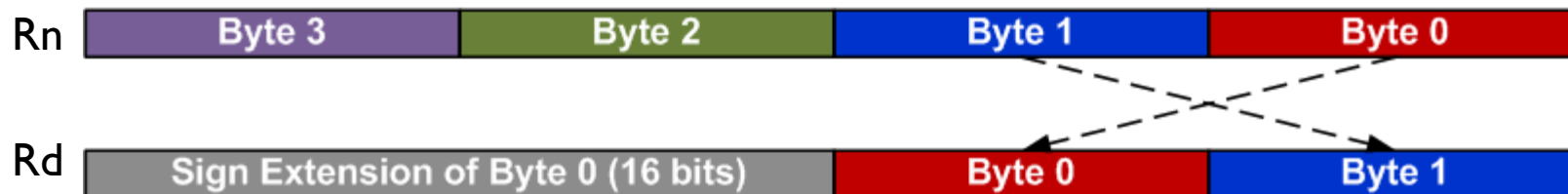
Example:

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

Reverse Order

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

REVSH Rd, Rn



Example:

```
LDR R0, =0x33448899    ; R0 = 0x33448899
REVSH R1, R0            ; R0 = 0xFFFF9988
```

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 = 0xFFFFFFFFE
```


Sign and Zero Extension

SXTB {Rd,} Rm {,ROR #n}	Sign extend a byte $Rd[31:0] \leftarrow \text{Sign Extend}((Rm \text{ ROR } (8 \times n))[7:0])$
SXTH {Rd,} Rm {,ROR #n}	Sign extend a half-word $Rd[31:0] \leftarrow \text{Sign Extend}((Rm \text{ ROR } (8 \times n))[15:0])$
UXTB {Rd,} Rm {,ROR #n}	Zero extend a byte $Rd[31:0] \leftarrow \text{Zero Extend}((Rm \text{ ROR } (8 \times n))[7:0])$
UXTH {Rd,} Rm {,ROR #n}	Zero extend a half-word $Rd[31:0] \leftarrow \text{Zero Extend}((Rm \text{ ROR } (8 \times n))[15:0])$

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

Sign extension (SX) preserves register value for signed int
Zero extension (UX) preserves register value for unsigned int
e.g., 0xFFFF8765 = 0x8765 as signed int, but 0xFFFF8765 != 0x8765 as unsigned int
0x00008765 = 0x8765 as unsigned int, but 0x00008765 != 0x8765 as signed int

Move Data between Registers

MOV	$Rd \leftarrow \text{operand2}$
MVN	$Rd \leftarrow \text{NOT operand2}$
MRS $Rd, \text{spec_reg}$	Move from special register to general register
MSR $\text{spec_reg}, Rm$	Move from general register to special register

MOV $r4, r5$	; Copy $r5$ to $r4$
MVN $r4, r5$	; $r4 = \text{bitwise logical NOT of } r5$
MOV $r1, r2, \text{LSL } \#3$	; $r1 = r2 \ll 3$
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, $Rd \leftarrow \#imm16$
MOVT Rd, #imm16	Move Top, $Rd \leftarrow \#imm16 \ll 16$
MOV Rd, #const	Move, $Rd \leftarrow \text{const}$

Example: Load a 32-bit number into a register

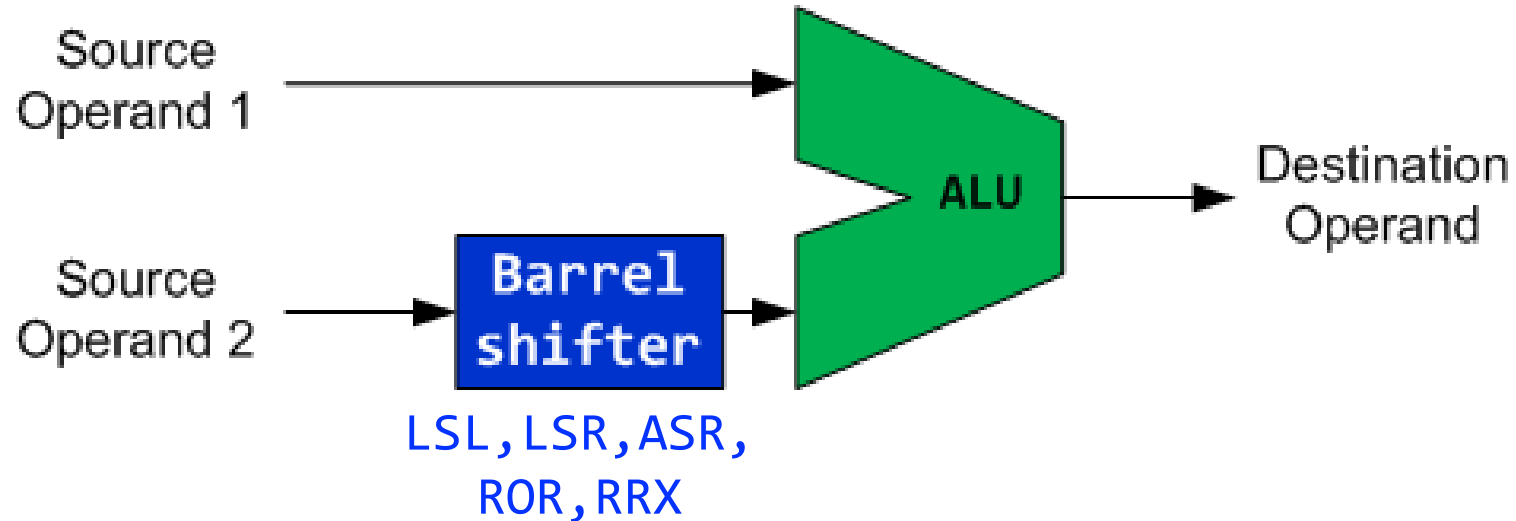
```
MOVW r0, #0x4321    ; r0 = 0x00004321
MOVT r0, #0x8765     ; r0 = 0x87654321
```

Order does matter!

- **MOVW** will zero the upper halfword
- **MOVT** won't zero the lower halfword

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

Flexible 2nd Source Operand



ADD r0, r1, Operand2

- ▶ Add r0, r1, **r2** ; $r0 = r1 + r2$
- ▶ Add r0, r1, **#1** ; $r0 = r1 + 1$
- ▶ Add r0, r1, **r2 LSL #2** ; $r0 = r1 + r2 \ll 2$

Use Shifts To Implement Multiplication And Division

- ▶ Use Barrel shifter to speed up multiplication and division

- ▶ Shifting left 1 bit $\Leftrightarrow$ multiplication by 2

- ▶ Examples:

- ▶ $r1 = 9 \times r0 = r0 + 8 \times r0$

`ADD r1, r0, r0, LSL #3` $\Leftrightarrow$ `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`

; $r1 = r0 + r0 \gg 3 = r0 + r0/8$ (unsigned)

`ADD r1, r0, r0, ASR #3`

; $r1 = r0 + r0 \gg 3 = r0 + r0/8$ (signed)

Barrel Shifter

Logical Shift Left (**LSL**)



Logical Shift Right (**LSR**)



Rotate Right Extended (**RRX**)



Arithmetic Shift Right (**ASR**)



Rotate Right (**ROR**)



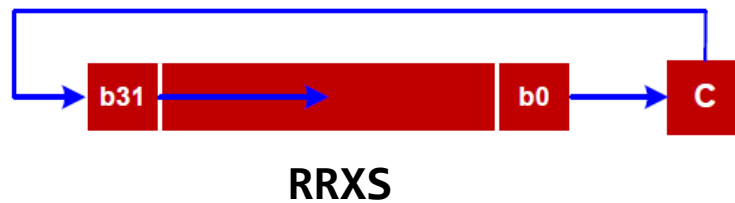
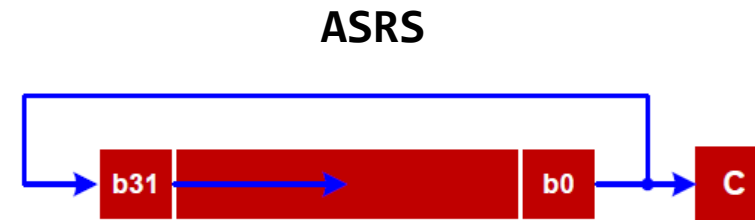
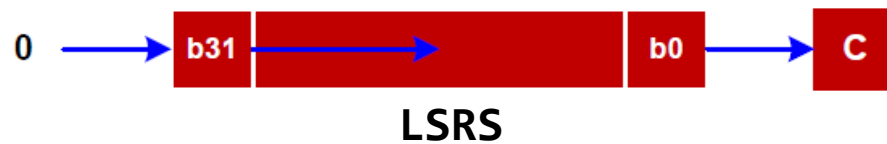
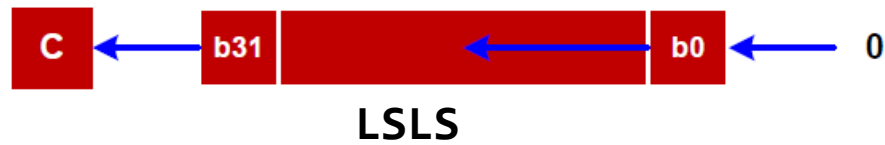
Why is there rotate right but no rotate left?

Rotate left (by n bits) can be implemented by a rotate right with by $(32 - n)$ bits).

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



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 .
 - ▶ There is no “ASL” (arithmetic shift left) instruction, since shifting left does not involve filling in the sign bit, which is shifted out
- ▶ 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 gets the bit that was rotated from bit 0 to bit 31. 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)

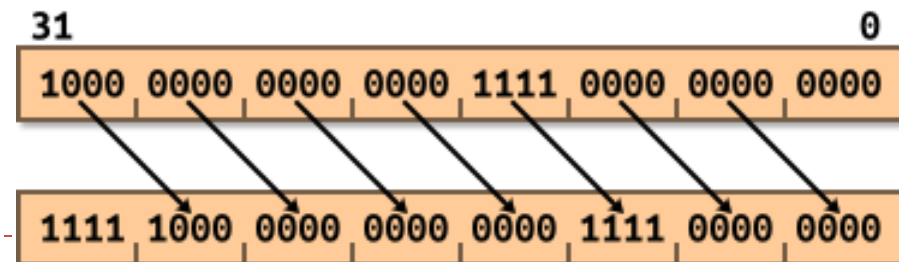
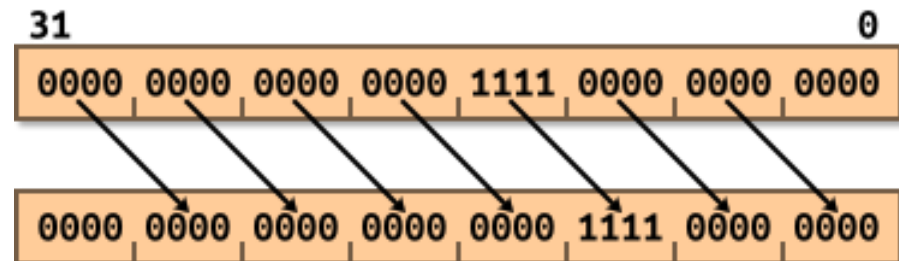
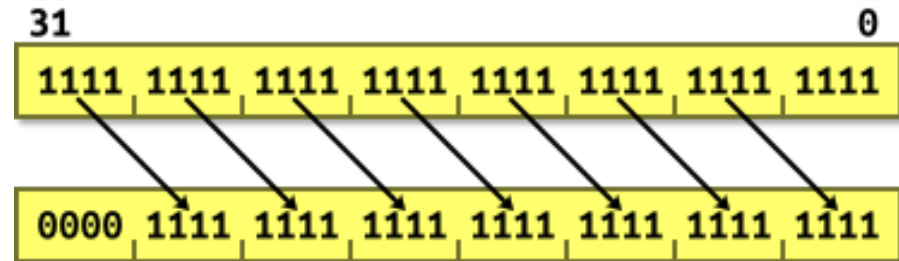
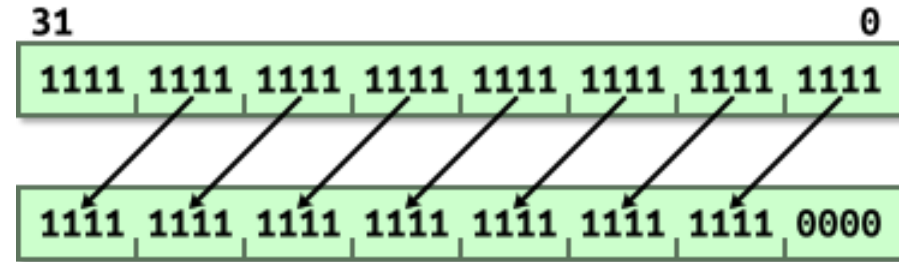
Logical Shift Left (**LSL**) (by 4 bits)



Logical Shift Right (**LSR**) (by 4 bits)

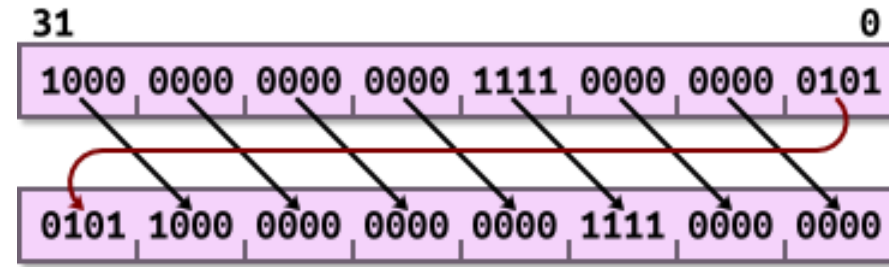


Arithmetic Shift Right (**ASR**) (by 4 bits)

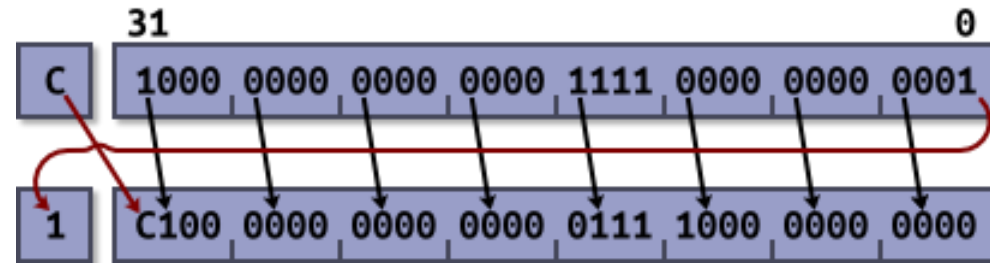


Examples (rotate)

Rotate Right (**ROR**) (rotate by 4)



Rotate Right Extended (**RRX**)
(can only rotate by 1)



Shift Operations (showing signed ints in decimal)

Logical Shift Left (LSL)



LSL {S} Rd, Rn, <shift>

moves all the bits of a register by n positions to the **left** and inserts n zeros in the right end

$$0 \leq n \leq 31$$

Example 1

```
; r2 = 0x0000_0001 (#1)
```

```
LSL r3, r2, #3
```

```
; r3 = 0x0000_0008 (#8)
```

```
; 8 = 23 * 1
```

For LSL, we can interpret the numbers as either unsigned or signed ints. They are the same when the sign bit (MSB) is 0, but different when the sign bit is 1.

Example 2

```
; r2 = 0x0000_0003 (#3)
```

```
LSL r3, r2, #2
```

```
; r3 = 0x0000_000C (#12)
```

```
; 12 = 22 * 3
```

Example 3

```
; r3 = 0xFFFF_0000 (#-65536)
```

```
LSLS r2, r3, #1
```

```
; r2 = 0xFFFE_0000 (#-131072)
```

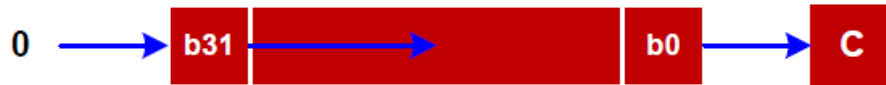
```
; -131072 = 21 * -65536
```

C=1, N=1, Z=0, V=not updated

If S is used: C ← last bit shifted out; N ← result[31]; Z ← (result==0); V unchanged.

Shift Operations (showing unsigned ints in decimal)

Logical Shift Right (LSR)



LSR{S} Rd, Rn, <shift>

moves all the bits of a register by n positions to the right and inserts n zeros in the left end

$$1 \leq n \leq 32$$

Example 1

```
; r2 = 0x0000_0010 (#16)
```

```
LSR r1, r2, #3
```

```
; r1 = 0x0000_0002 (#2)
```

```
; 2 = 16/23
```

Example 2

```
; r2 = 0x8000_0000 (#2,147,483,648)
```

```
LSR r2, r2, #2
```

```
; r2 = 0x2000_0000 (#536,870,912)
```

```
; 536,870,912 = 2,147,483,648/22
```

Example 3

```
; r2 = 0x0000_0001 (#1)
```

```
LSRS r3, r2, #1
```

```
; r3 = 0x0000_0000 (#0)
```

```
; 0 = 1/21
```

C=1, N=0, Z=1, V=not updated

If S is used: C ← last bit shifted out; N ← result[31]; Z ← (result==0); V unchanged.

For LSR, we can only interpret the numbers as unsigned ints.

Shift Operations (showing signed ints in decimal)

Arithmetic Shift Right (ASR)



ASR{S} Rd, Rn, <shift>

moves all the bits of a register by n positions to the **right** and inserts n copies of the sign bit in the left end

$$1 \leq n \leq 32$$

Example 1

```
; r0 = 0xFFF8_0000 (-524288)
```

```
ASR r1, r0, #3
```

```
; r1 = 0xFFFF_0000 (-65536)
```

```
; -65536 = -524288/23
```

Example 2

```
; r2 = 0x8000_0000 (-2,147,483,648)
```

```
ASR r2, r2, #2
```

```
; r2 = 0xE000_0000 (# -536,870,912)
```

```
; -536,870,912 = -2,147,483,648/22
```

Example 3

```
; r2 = 0xFFFF_F001 (#-4095)
```

```
ASRS r3, r2, #1
```

```
; r3 = 0xFFFF_F800 (#-2048)
```

```
; -2048 = -4096/21
```

C=1, N=1, Z=0, V=not updated

If S is used: C ← last bit shifted out; N ← result[31]; Z ← (result==0); V unchanged.

For ASR, we can only interpret the numbers as signed ints.

Rotate Operations

Rotate Right (ROR)



ROR{S} Rd, Rn, <shift>

Circular shifts of all the bits of a register by n positions to the **right** as if the right end of the register is joined with its left end. The last shifted bit updates the carry bit

$$1 \leq n \leq 31$$

Example 1

```
; r2 = 0x0008_0000
```

```
ROR r2, r2, #10
```

```
; r2 = 0x0000_0200
```

Example 2: rotate **left** by 12 bits

```
; r0 = 0xF000_0000
```

```
ROR r2, r0, #20
```

```
; r2 = 0x0000_0F00
```

Rotate left by m (12) bits is equivalent to rotate right

ROR by $32 - m = 20$ bits

Example 3

```
; r2 = 0xF0F0_F001 (binary: 1111 0000 1111  
0000 1111 0000 0000 0001)
```

```
; r1 = 0x0000_000E (rotate right by 14 bits)
```

```
RORS r3, r2, r1
```

```
; r3 = 0xC007_C3C3 (binary: 1100 0000 0000  
0111 1100 0011 1100 0011)
```

C=1, N=1, Z=0, V=not updated

If S is used: $C \leftarrow$ last bit shifted out; $N \leftarrow$ result[31]; $Z \leftarrow$ (result==0); V unchanged.

Rotate Operations

Rotate Right Extended (RRX)



RRX{S} Rd, Rn

This is a one-bit rotate instruction.

Example 1

```
; r2 = 0x0008_0003, c = 1
```

RRX r2, r2

```
; r2 = 0x8004_0001, c = 1
```

Example 2:

```
; r2 = 0xF000_0001, c = 0
```

RRX r1, r2

```
; r1 = 0x7800_0000, c = 1
```

Example 3

```
; r2 = 0xF0F0_F001, c = 0
```

RRXS r3, r2

```
; r3 = 0x7878_7800, c = 1
```

C=1, N=0, Z=0, V= not updated

If S is used: $C \leftarrow$ last bit shifted out; $N \leftarrow$ result[31]; $Z \leftarrow$ (result==0); V unchanged.

Barrel Shifter More Examples

- ▶ MOV r0, r0, LSL #1
 - ▶ $r0 = r0 * 2$
- ▶ MOV r1, r1, LSR #2
 - ▶ $r1 = r1 / 4$ (unsigned)
- ▶ MOV r2, r2, ASR #2
 - ▶ $r2 = r2 / 4$ (signed)
- ▶ MOV r3, r3, ROR #16
 - ▶ Swap the top and bottom halves of r3.
- ▶ ADD r4, r4, r4, LSL #4
 - ▶ $r4 = r4 * 17 (= r4 + r4 * 16)$
- ▶ RSB r5, r5, r5, LSL #5
 - ▶ $r5 = r5 * 31 (= r5 * 32 - r5)$
 - ▶ Reverse-subtract using barrel shifter on 2nd operand
- ▶ SUB r5, r5, r5, LSR #5
 - ▶ $r5 = r5 - (r5 / 32)$ (signed)
- ▶ LDR r9, [r12, r8, LSL #2]
 - ▶ Load a 32-bit word into r9 from the memory address computed as $r12 + (r8 * 4)$

SUB vs. RSB

- ▶ SUB instruction: SUB Rd, Rn, Operand2 performs $Rd = Rn - \text{Operand2}$
- ▶ RSB instruction: RSB Rd, Rn, Operand2 performs $Rd = \text{Operand2} - Rn$
- ▶ There are equivalent:
 - ▶ SUB R5, R3, #10 @ R5 = R3 - 10
 - ▶ RSB R5, R3, #10 @ R5 = 10 - R3
- ▶ When to use RSB?
- ▶ Subtracting from constants, since constants can only appear as Operand2 in ARM instructions. For example:
 - ▶ RSB R2, R4, #1 means $R2 = 1 - R4$
 - ▶ This cannot be done with SUB without first loading the constant into a register
- ▶ Negation Operations by subtracting from zero:
 - ▶ RSB R0, R0, #0 effectively computes $R0 = 0 - R0 = -R0$
- ▶ Complex Operand2 Operations
 - ▶ RSB is valuable when you want to perform operations on Operand2 before subtraction, such as shifting :
 - ▶ RSB R1, R2, R3, LSL #1 computes $R1 = (R3 \ll 1) - R2$
 - ▶ This allows you to shift a value and then subtract from it in a single instruction

Integer Array Access with LSL

- ▶ To calculate the address of element `array[i]` of 32-bit integers, we calculate (base address of array) + $i*4$ for an array of words (1 word = 4 Bytes). For example:
 - ▶ `ADR r3,ARRAY` @ load base address of `ARRAY` into `r3` (`ARRAY` contains 4-byte ints)
 - ▶ `MOV r2,#6` @ Suppose we want to access `ARRAY[6]`
 - ▶ `MOV r4,r2,LSL #2` @ logical shift `i`'s value in `r2` by 2 to multiply its value by 4
 - ▶ `ADD r5,r3,r4` @ finish calculation of the address of element `array[i]` in `r5`
 - ▶ `LDR r6,[r5]` @ load value of `array[i]` into `r6` using the address in `r5`
- ▶ Alternatively, we can perform this same address calculation with a single `ADD` instead of 2 instructions `MOV` and `ADD` shown above:
 - ▶ `MOV r4,r2,LSL #2` and `ADD r5,r3,r4` can be replaced by one instruction:
 - ▶ `ADD r5,r3,r2,LSL #2` @ calculate address of `array[i]` in `r5` with single `ADD`
- ▶ Alternatively, ARM has some nice addressing modes to speedup array item access:
 - ▶ `LDR r6,[r3,r2,LSL #2]`

Notes on Shifts and Flags

- ▶ A standalone logical shift instruction without the **S** suffix (e.g. LSL R0, R0, #1) does **not** update the condition flags **NZCV**. The **S** suffix (e.g. LSLS R0, R0, #1) makes the instruction update.
- ▶ When a shift appears as part of a data-processing instruction that ends with **S** (e.g., ADDS), the processor first computes the shifted Operand2. During that computation, the shift logic sets the **carry (C)** flag to the *last bit shifted out*. After that, the data-processing instruction may itself update NZCV based on its arithmetic or logical result, potentially overwriting the C flag.
- ▶ LSL can appear as a *shift operator* within another instruction, but LSLS cannot.
- ▶ **Examples:**
 - ▶ LSL R0, R0, #1 ; standalone LSL does NOT update flags
 - ▶ LSLS R0, R0, #1 ; standalone LSLS updates NZCV (S suffix)
 - ▶ ANDS R2, R1, R0, LSL #1 ; valid. LSL forms part of operand, ANDS updates NZCV
 - ▶ ANDS R2, R1, R0, LSLS #1 ; invalid. Cannot embed 'LSLS' inside operand

In Detail: ANDS R2, R1, R0, LSL #1

- ▶ The barrel shifter always runs first. If the main arithmetic instruction does not redefine C (like ANDS), shift's C stays. If the main arithmetic instruction defines C (like ADDS), it overwrites shift's C.
- ▶ Step 1: Compute shifted Operand2
 - ▶ R0 is shifted left by 1. During this shift, the barrel shifter sets C = the last bit shifted out (bit 31 of original R0).
- ▶ Step 2: Perform AND
 - ▶ $R2 = R1 \text{ AND } (R0 \ll 1)$
- ▶ Step 3: Update flags (because of S suffix in ANDS)
 - ▶ N = bit 31 of result
 - ▶ Z = result == 0
 - ▶ V = unchanged (logical ops don't touch V)
 - ▶ C = the carry generated by the shift

Example 1: ANDS

```
LDR  r0, =0xFFFFFFFF00
LDR  r1, =0x00000001
ANDS r2, r1, r0, LSL #1
```

N = 0, Z = 1, C = 1, V = not updated

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

r0 = 0xFFFFFFFF00

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 = 1 is set by shift “r0, LSL #1” to be bit 31 of r0. (LSL updates the C flag, since ANDS does not update C.)

V is unaffected by either ANDS or shift (left unchanged from its previous value)

Example 2: ADDS

```
LDR    r0, =0xFFFFFFFF00
LDR    r1, =0x00000001
ADDS   r2, r1, r0, LSL #1
```

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

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

r0 = 0xFFFFFFFF00

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). (C = 1 is set by shift “r0, LSL #1”, but the C flag is overwritten by ADDS to be C=0.)

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



Set a Bit in C

$$a \mid= (1 \ll k)$$

or

$$a = a \mid (1 \ll k)$$

Example: $k = 5$

a	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
$1 \ll k$	0	0	1	0	0	0	0	0
$a \mid (1 \ll k)$	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:

```
MOV r4, #1          ; r4 = 1
LSL r4, r4, #5       ; r4 = 1<<5
ORR r0, r0, r4      ; r0 = r0 | 1<<5
```

Solution 2:

```
MOV r4, #1          ; r4 = 1
ORR r0, r0, r4, LSL #5 ; r0 = r0 & not (1<<5)
```

Solution 3:

```
ORR r0, r0, # (1 << 5) ; r0 = r0 & not (1<<5)
```


Clear a Bit in C

$$a \ \&= \ \sim(1 \ll k)$$

Example: $k = 5$

a	a ₇	a ₆	a ₅	a ₄	a ₃	a ₂	a ₁	a ₀
$\sim(1 \ll k)$	1	1	0	1	1	1	1	1
$a \ \& \ \sim(1 \ll k)$	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

Recall:

BIC r0, r1, r2 ; Bit clear, $r0 = r1 \& \sim r2$

a &= ~(1<<5)

Solution 1:

```
MOV r4, #1           ; r4 = 1
LSL r4, r4, #5        ; r4 = 1<<5
MVN r4, r4            ; r4 = not (1<<5)
AND r0, r0, r4       ; r0 = r0 & not (1<<5)
```

Solution 2:

```
MOV r4, #1           ; r4 = 1
MVN r4, r4, LSL #5 ; r4 = not (1<<5)
AND r0, r0, r4       ; r0 = r0 & not (1<<5)
```

Solution 3:

```
MOV r4, #1           ; r4 = 1
BIC r0, r0, r4, LSL #5 ; r0 = r0 & not (1<<5)
```

Solution 4:

```
BIC r0, r0, # (1 << 5) ; r0 = r0 & not (1<<5)
```



Toggle a Bit in C

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

$$a \wedge= 1 \ll k$$

Example: $k = 5$

a	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
$1 \ll k$	0	0	1	0	0	0	0	0
$a \wedge (1 \ll k)$	a_7	a_6	NOT(a_5)	A_4	a_3	a_2	a_1	a_0

Truth table of
Exclusive OR

m	n	$m \oplus n$
0	0	0
0	1	1
1	0	1
1	1	0

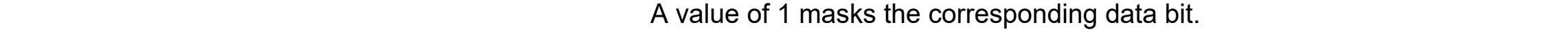
Toggle a Bit in Assembly

$a \oplus 1 \ll 5$

Solution:

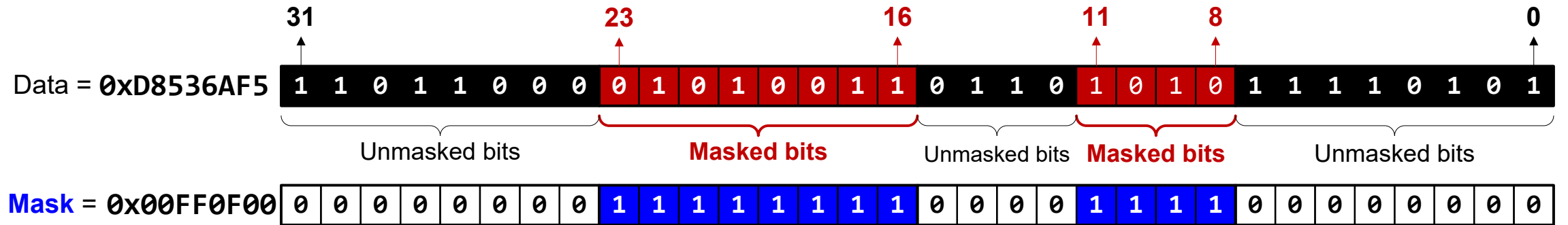
```
MOV r4, #1           ; r4 = 1
EOR r0, r0, r4, LSL #5 ; r0 = r0 ^ 1<<5
```

Here we can use MOVS and EORS instead of MOV and EOR, if the flags are used by later instructions.

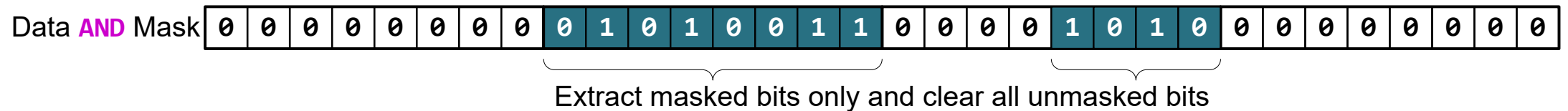


- 69

Clear all unmasked bits

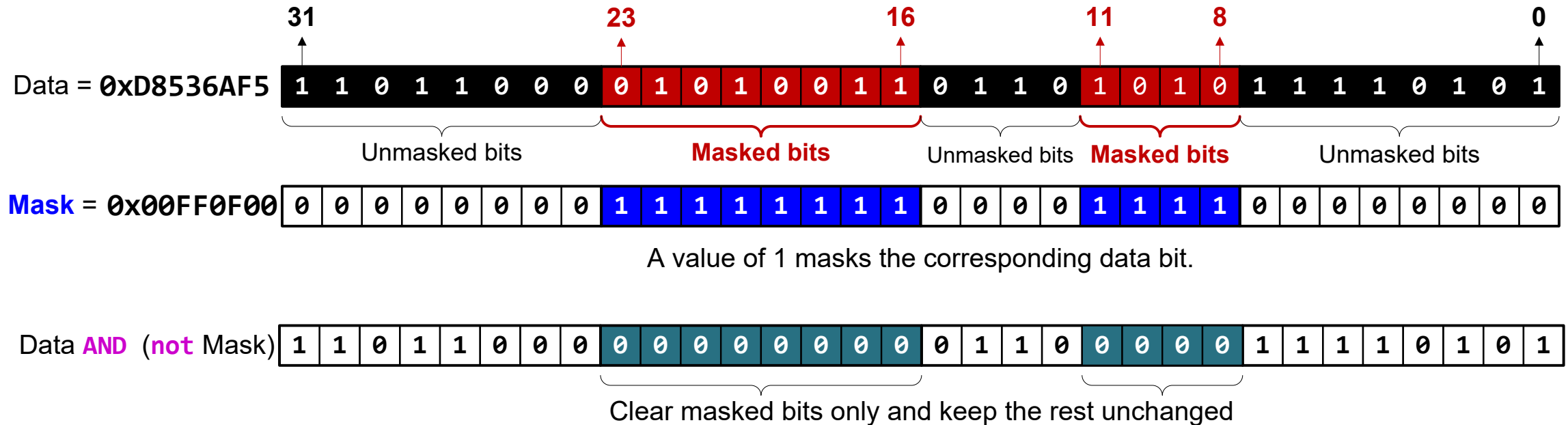


A value of 1 masks the corresponding data bit.



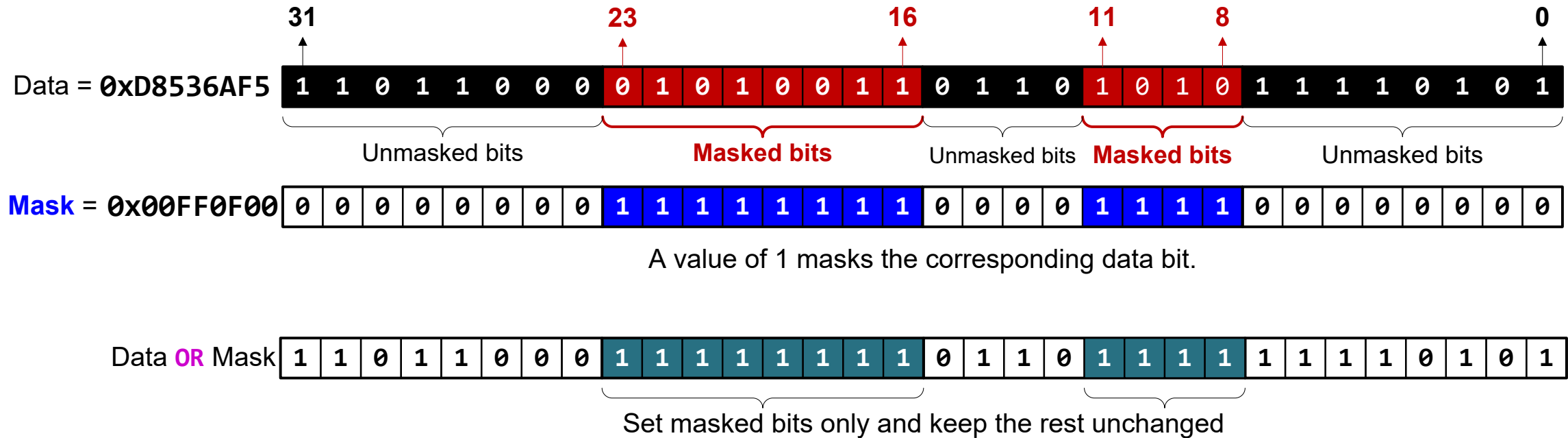
Data &= Mask;

Clear all masked bits

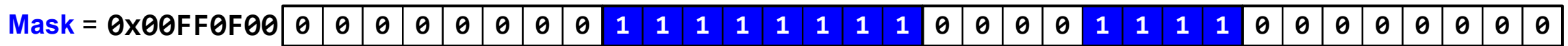


Data &= ~Mask;

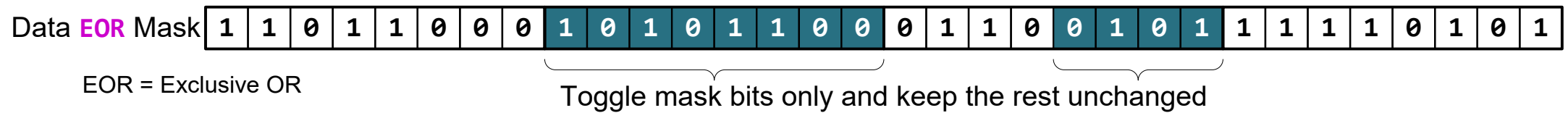
Set all masked bits



Data |= Mask;



A value of 1 masks the corresponding data bit.



Data ^= Mask;

Carry and Overflow Flags w/ Arithmetic Instructions

Carry flag $C = 1$ (Borrow flag $= 0$) upon an **unsigned** addition if the answer is wrong (true result $> 2^n - 1$)

Carry flag $C = 0$ (Borrow flag $= 1$) upon an **unsigned** subtraction if the answer is wrong (true result < 0)

Overflow flag $V = 1$ upon a **signed** addition or subtraction if the answer is wrong (true result $> 2^{n-1} - 1$ or true result $< -2^{n-1}$)

Overflow may occur when adding 2 operands with the same sign, or subtracting 2 operands with different signs; $V = 1$ if Positive + Positive $\rightarrow$ Negative, or Negative + Negative $\rightarrow$ Positive.

Overflow cannot occur when adding 2 operands with different signs or when subtracting 2 operands with the same sign.

Tip: Convert subtraction to addition with Two's complement.

	Unsigned Addition	Unsigned Subtraction	Signed Addition or Subtraction
Carry flag	true result $> 2^n - 1 \rightarrow$ Carry flag $= 1$ Borrow flag $= 0$ (Result incorrect)	true result $< 0 \rightarrow$ Carry flag $= 0$ Borrow flag $= 1$ (Result incorrect)	N/A
Overflow flag	N/A	N/A	true result $> 2^{n-1} - 1$ or true result $< -2^{n-1}$ $\rightarrow$ Overflow flag $= 1$ (Result incorrect)

Example

- ▶ For an 8-bit system, calculate $0x35 + 0x19$, setting C and V flags
 - ▶ Unsigned int range $[0, 2^8-1] = [0, 255]$, signed int range $[-2^7, 2^7-1] = [-128, 127]$
- ▶ ANS:
 - ▶ Convert to binary and perform addition as in table
 - ▶ $C = 0$ since there is no carry-out from MSB b7
 - ▶ $V = 0$ since Op1, Op2 and result are all positive (sign bit = 0)
 - ▶ In decimal (not needed for exam):
 - ▶ Unsigned addition: $53 + 25 = 78$ (result correct)
 - ▶ Signed addition: $53 + 25 = 78$ (result correct)

	0	1	1	0	0	0	1		Carry
	0	0	1	1	0	1	0	1	Op1: 0x35
	0	0	0	1	1	0	0	1	Op2: 0x19
C = 0	0	1	0	0	1	1	1	0	Result: 0x4E
V = 0									

Example

- ▶ For an 8-bit system, calculate $0x35 + 0x5B$, setting C and V flags
 - ▶ Unsigned int range $[0, 2^8-1] = [0, 255]$, signed int range $[-2^7, 2^7-1] = [-128, 127]$
- ▶ ANS:
 - ▶ Convert to binary and perform addition as in table
 - ▶ $C = 0$ since there is no carry-out from MSB b7
 - ▶ $V = 1$ since Op1, Op2 are positive, result is negative
 - ▶ In decimal:
 - ▶ Unsigned addition: $53 + 91 = 144$ (result correct)
 - ▶ Signed addition: true sum = $53 + 91 = 144 \rightarrow \text{result} = -112$ (result incorrect, $V=1$)

	1	1	1	1	1	1		Carry
	0	0	1	1	0	1	1	Op1: 0x35
	0	1	0	1	1	0	1	Op2: 0x5B
C = 0 V = 1	1	0	0	1	0	0	0	Result: 0x90

Example: Subtraction

- ▶ For an 8-bit system, calculate $0x35 - 0x2D$, setting C and V flags
- ▶ ANS:
 - ▶ Convert to binary and perform addition as in table (another way is to perform subtraction in binary, but we do not cover it here)
 - ▶ $0x2D = 00101101$, its negation $TC(00101101) = 11010011 = 0xD3$
 - ▶ $C = 1$ since there is carry-out from MSB b7
 - ▶ $V = 0$ since Op1 is positive, Op2 is negative, result is positive
 - ▶ Overflow cannot occur when adding 2 operands with different signs
 - ▶ In decimal:
 - ▶ Unsigned subtraction: $53 - 45 = 8$ (result correct, $C=1$, Borrow Flag=0)
 - ▶ Signed: $53 - 45 = 8$ (result correct, $V=0$)

$C = 1$ $V = 0$	1	1	1	1	0	1	1	1	Carry
	0	0	1	1	0	1	0	1	Op1: 0x35
	1	1	0	1	0	0	1	1	Op2: 0xD3
	0	0	0	0	1	0	0	0	Result: 0x08 (drop 1 in 0x108)

Example: Subtraction More Details

- ▶ Signed arithmetic $0x35 - 0x2D = 0x35 + TC(0x2D) = 0x35 + 0xD3$, as $TC(0x2D)$ is negation of $0x2D$
 - ▶ $C=1$, Borrow Flag=0
 - ▶ Signed subtraction: $53 - 45 = 8$ (result correct, $V=0$)
 - ▶ Signed addition: $53 + (-45) = 8$ (result correct, $V=0$)
- ▶ TC is only valid for signed ints, so $0x35 - 0x2D \neq 0x35 + TC(0x2D)$ for signed arithmetic, as $TC(0x2D)=0xD3$ is a totally different unrelated value 211
 - ▶ Unsigned subtraction: $53 - 45 = 8$ (result correct, $C=1$)
 - ▶ Unsigned addition: $53 + 211 = 8$ (true result 264) (result wrong, $C=1$)

$C = 1$ $V = 0$	I	1	1	1	0	1	1	1	
		0	0	1	1	0	1	0	1
		1	1	0	1	0	0	1	1
		0	0	0	0	1	0	0	0

Example

- ▶ For an 8-bit system, calculate $0x9E - 0x2D$, setting C and V flags
- ▶ ANS:
 - ▶ Convert to binary and perform addition as in table
 - ▶ $0x2D = 00101101$, its negation $TC(00101101) = 11010011 = 0xD3$
 - ▶ $C = 1$ since there is carry-out from MSB b7
 - ▶ $V = 1$ since Op1, Op2 are both negative, result is positive
 - ▶ In decimal:
 - ▶ Unsigned subtraction: $158 - 45 = 113$ (result correct, $C=1$, Borrow Flag=0)
 - ▶ Signed subtraction: true sum = $-98 - 45 = -143 \rightarrow$ result = $+113$ (wrong, $V=1$)

C = 1 V = 1	1	0	0	1	1	1	1	0		Carry
		1	0	0	1	1	1	1	0	Op1: 0x9E
		1	1	0	1	0	0	1	1	Op2: 0xD3
		0	1	1	1	0	0	0	1	Result: 0x71 (drop 1 in 0x171)

Dec	Hex	Binary
53	35	0 1 1 0 0 0 1
+25	+19	0 0 1 1 0 1 0 1
<u>78</u>	<u>4E</u>	0 0 0 1 1 0 0 1
		0 1 0 0 1 1 1 0
		C = 0
		V = 0

Note no carry from bit 6 to bit 7 and no carry from bit 7 to C.

Dec	Hex	Binary
53	35	1 1 1 0 1 1 1
-45	+D3	0 0 1 1 0 1 0 1
<u>8</u>	<u>108</u>	1 1 0 1 0 0 1 1
		0 0 0 0 1 0 0 0
		C = 1
		V = 0

Ignore carry

Note carry from bit 6 to bit 7 and carry from bit 7 to C.

Thinking SIGNED we added a positive number to a negative number and got the correct positive answer. Therefore, the OVERFLOW bit, V, is cleared to 0. Correct answer (8) is inside the range -128 to +127.

Dec	Hex	Binary
53	35	1 1 1 1 1 1 1
+91	+5B	0 0 1 1 0 1 0 1
<u>144</u>	<u>90</u>	0 1 0 1 1 0 1 1
		1 0 0 1 0 0 0 0
		C = 0
		V = 1

Note carry from bit 6 to bit 7 but no carry from bit 7 to C.

Thinking SIGNED we added two positive numbers and got a negative result. This can't be correct! Therefore, the OVERFLOW bit, V, is set to 1. Correct answer (144) is outside the range -128 to +127.

Dec	Hex	Binary
-98	9E	0 0 1 1 1 1 0
-45	+D3	1 0 0 1 1 1 1 0
<u>-143</u>	<u>171</u>	1 1 0 1 0 0 1 1
		1 1 1 0 0 0 1
		C = 1
		V = 1

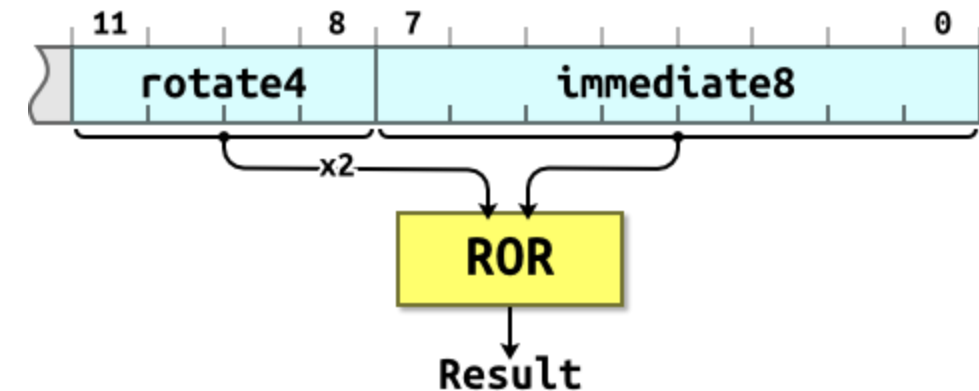
Ignore carry

Note no carry from bit 6 to bit 7 but there is a carry from bit 7 to C.

Thinking SIGNED we added two negative numbers and got a positive answer. This must be wrong! Therefore, the OVERFLOW bit, V, is set to 1. Correct answer (-143) is outside the range -128 to +127.

ARM Immediate Values

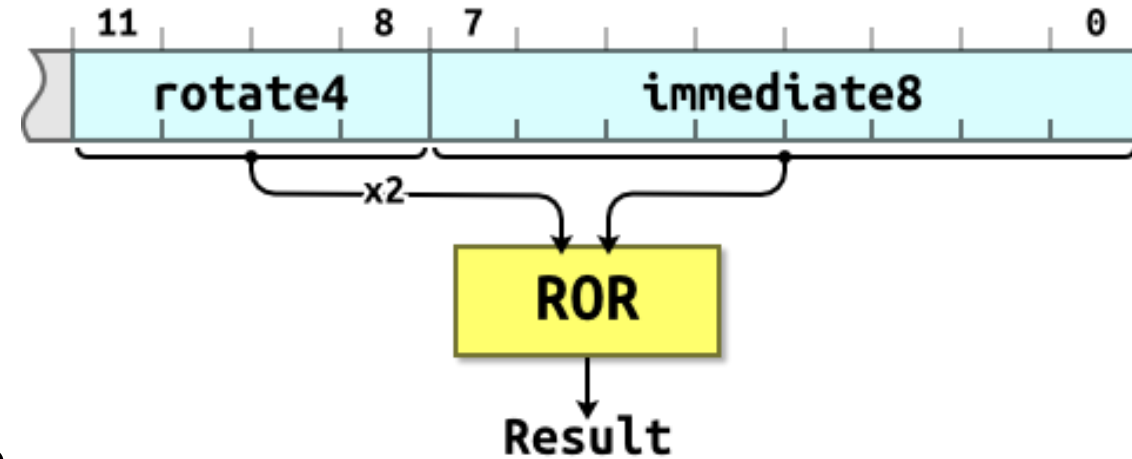
- ▶ You can't fit an arbitrary 32-bit value into a 32-bit instruction word. ARM data processing instructions have 12 bits of space for values in one 32-bit instruction word. This is arranged as a 4 rotate value and an 8 immediate value. The real immediate = **ROR(immediate8, rotate4 × 2)**.
 - ▶ The 4-bit rotate value stored in bits 11-8 is multiplied by two giving a range of 0-30 in steps of two.
- ▶ Using this scheme we can express immediate constants such as:
 - ▶ 0x000000FF
 - ▶ 0x0000FF0 = ROR(0x000000FF, 28) ; rotate4 = 14
 - ▶ Rotate left by n=4 bits can be implemented by a rotate right with by (32 - n = 28) bits.
 - ▶ 0xFF000000 = ROR(0x000000FF, 8) ; rotate4 = 4
 - ▶ 0xF000000F = ROR(0x000000FF, 4) ; rotate4 = 2
- ▶ But these immediate constants are not possible:
 - ▶ 0x000001FE
 - ▶ 0xF000F000
 - ▶ 0x55550000
 - ▶ 0xFFFFFFFF
- ▶ An assembler will convert big values to the rotated form. Impossible values will cause an error. For example, this instruction is invalid:
 - ▶ AND R2, R0, #0xFFFFFFFF8F



12-bit immediate value

Encoding #4080 as Immediate

- ▶ `ADD r1, r2, #4080`
 - ▶ `4080 = 0xFF0` in binary
- ▶ You need to set values for `rotate4` and `immediate8` to encode `#4080`. The encoding is:
 - ▶ `immediate8 = 0x000000FF` (11111111 in binary)
 - ▶ `rotate4 = 1110` ($14 * 2 = 28$)
- ▶ `ROR(0x000000FF, 28) = 0x000000FF0` (4080 in decimal)
 - ▶ rotate left by 4 = rotate right by 28
- ▶ Values such as `#4079`, `#4081`, `#4082`... cannot be encoded exactly, since no matter how you set `immediate8`, you only have 8 bits and you must lose some 1's in the original number. You can use `#4080` as an approximation for them
 - ▶ `#4079 = 111111101111`
 - ▶ `#4081 = 111111110001`
 - ▶ `#4082 = 111111110010`



Loading Wide Values

- ▶ You can form constants wider than those available in a single instruction by using a sequence of instructions to build up the constant. For example:
 - ▶ `MOV r2, #0x55` ; R2 = 0x00000055
 - ▶ `ORR r2, r2, r2, LSL #8` ; R2 = 0x00005555
 - ▶ `ORR r2, r2, r2, LSL #16` ; R2 = 0x55555555
- ▶ Or load the value from memory with pseudo-instruction `LDR Rx,=const`
 - ▶ `LDR r2, =0x55555555`
- ▶ Or use `MVN` instead of `MOV`:
 - ▶ The invalid instruction `MOV r0, #0xFFFFFFFF` can be implemented as `MVN r0, #0`

Pseudo instruction

- ▶ The ARMv7 pseudo instruction `LDR r2, =0x55555555` is implemented by the assembler to load a 32-bit immediate value large constants beyond the range of the immediate field of a `MOV/MVN` instruction. It is translated into a PC-relative load instruction that fetches the constant from a literal pool embedded in the code:
 - ▶ The assembler first tries to generate a `MOV` or `MVN` instruction if the immediate value can be encoded directly by those instructions.
 - ▶ Since `0x55555555` cannot be encoded directly in a `MOV` or `MVN`, the assembler places this value in a literal pool, which is a section of memory embedded in the code to hold constant values.
 - ▶ Then, the assembler generates a PC-relative `LDR` instruction that loads the value from the literal pool address into the specified register (`r2` in this case).
 - ▶ The actual machine instruction looks like `LDR r2, [pc, #offset]`, where the offset points to the location of the `0x55555555` constant in the literal pool.
 - ▶ This makes register value assignment flexible, but at the cost of incurring a memory access
- ▶ (In exam questions like Q3 in the midterm, you are not allowed to use the `LDR` pseudo-instruction)

References

- ▶ Lesson 45b - Adders Carry and Overflow, LBEbooks
 - ▶ https://www.youtube.com/watch?v=9cXe_T99nL4
- ▶ Lecture 25. Arithmetic and Logical Instructions
 - ▶ <https://www.youtube.com/watch?v=H-vOP2yRUj4&list=PLRjhV4hUhlymmp5CCelFPyxbknsdcXCc8&index=25>
- ▶ Lecture 26. Updating NZCV bit flags
 - ▶ https://www.youtube.com/watch?v=SGjibMID2_A&list=PLRjhV4hUhlymmp5CCelFPyxbknsdcXCc8&index=26