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/

Adding Two Integers

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

If values are in registers

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

C Statement

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

Assembly Statement

?

Adding Two Integers

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

If values are in registers

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

C Statement

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

Assembly Statement

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

- r2: Destination
- r1: Source Operand 1
- r0: Source Operand 2

Adding Two Integers

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

If values are in registers

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

C Statement

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

Assembly Statement

?

Adding Two Integers

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

If values are in registers

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

C Statement

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

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 + carry

SUB  r0, r1, r2  ; r0 = r1 - r2
SBC  r0, r1, r2  ; Subtract with borrow, r0 = r1 - r2 – (1 – 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

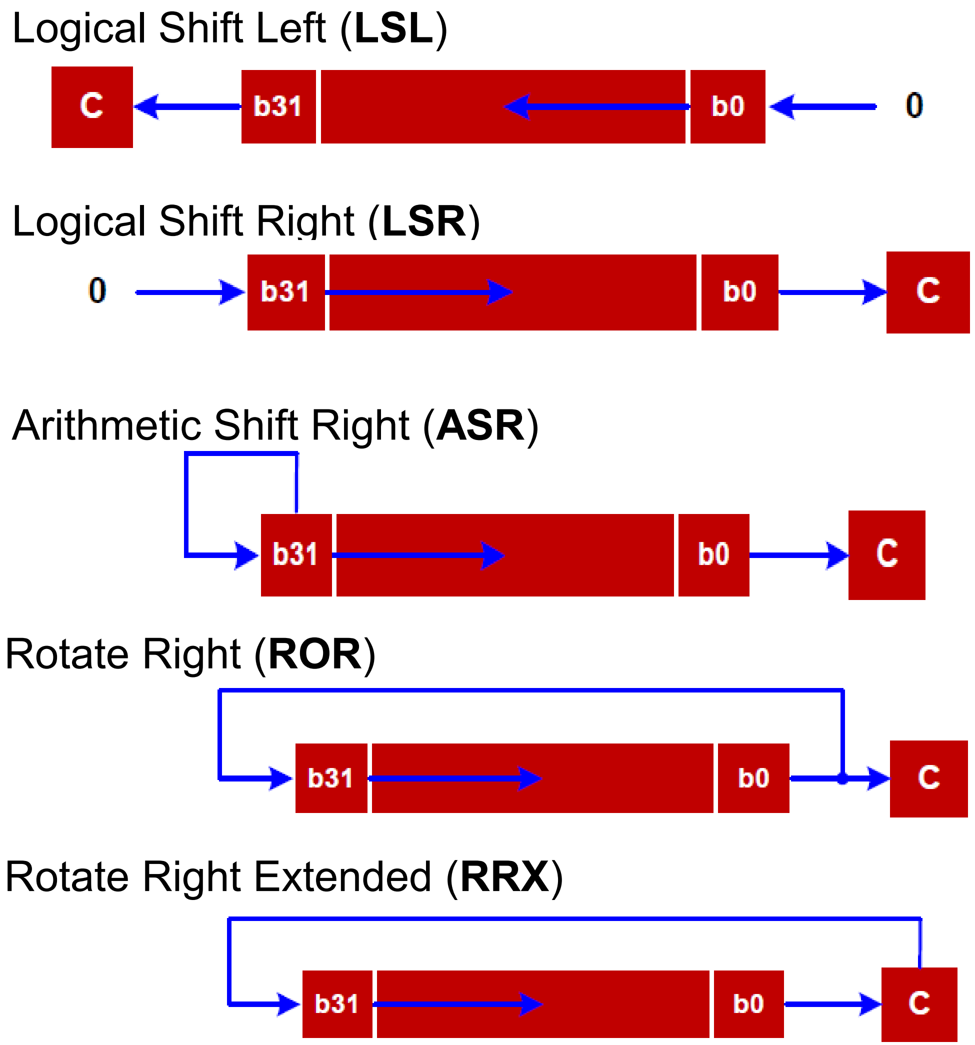
- **AND** r0, r1, r2 ; Bitwise AND, r0 = r1 AND r2
- **ORR** r0, r1, r2 ; Bitwise OR, r0 = r1 OR r2
- **EOR** r0, r1, r2 ; Bitwise Exclusive OR, r0 = r1 EOR r2
- **ORN** r0, r1, r2 ; Bitwise OR NOT, r0 = r1 ORN r2
- **BIC** r0, r1, r2 ; Bit clear, r0 = r1 & ~r2

```
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 0 1
r2 1 0 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 0 0 1
```

Bit-wise Logic AND

Example Shift & Rotate Instructions

- **LSL** r0, r1, r2 ; Logical shift left,
  r0 = r1 << r2
- **LSR** r0, r1, r2 ; Logical shift right,
  r0 = r1 >> r2
- **ASR** r0, r1, r2 ; Arithmetic shift right,
  r0 = r1 >> r2
- **ROR** r0, r1, r2 ; Rotate right,
  r0 = r1 rotate by r2 bits
- **RRX** r0, r1, r2 ; Extended rotate right,
  {C, r0} = {C, r1} rotate by r2 bits



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 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 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

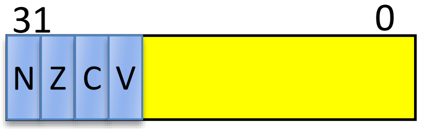
```
ADD r1, r3          ; r1 = r1 + r3
ADD r1, #15         ; r1 = r1 + 15
```

Commonly Used Arithmetic Operations

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

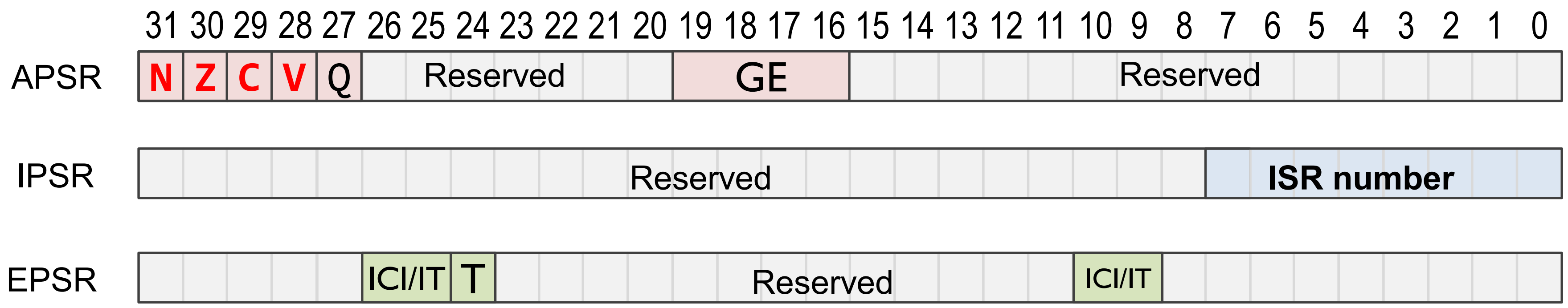


CPSR (Current Program Status Register)

- 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: 1/0 = Result from ALU is Negative/positive
- Z: 1/0 = Result from ALU is Zero/non-zero
- C: Three cases:
  - 1/0 = ALU addition Carry out/no carry out
  - 1/0 = ALU subtraction no borrow/borrow
  - 1/0 = Bit shifted/rotated out
- V: 1/0 = ALU oVerflowed/no overflow

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

Updating NZCV flags in PSR

Flags not changed		Flags updated
ADD	⟶	ADDS
SUB	⟶	SUBS
MUL	⟶	MULS
UDIV	⟶	UDIVS
AND	⟶	ANDS
ORR	⟶	ORRS
LSL	⟶	LSLS
MOV	⟶	MOVS

Most instructions update NZCV flags only if S suffix is present

`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

ADD *vs* ADDS

```
ADD  r0, r1, r2  ; r0 = r1 + r2, NZCV flags unchanged
ADDS r0, r1, r2  ; r0 = r1 + r2, NZCV flags updated
```

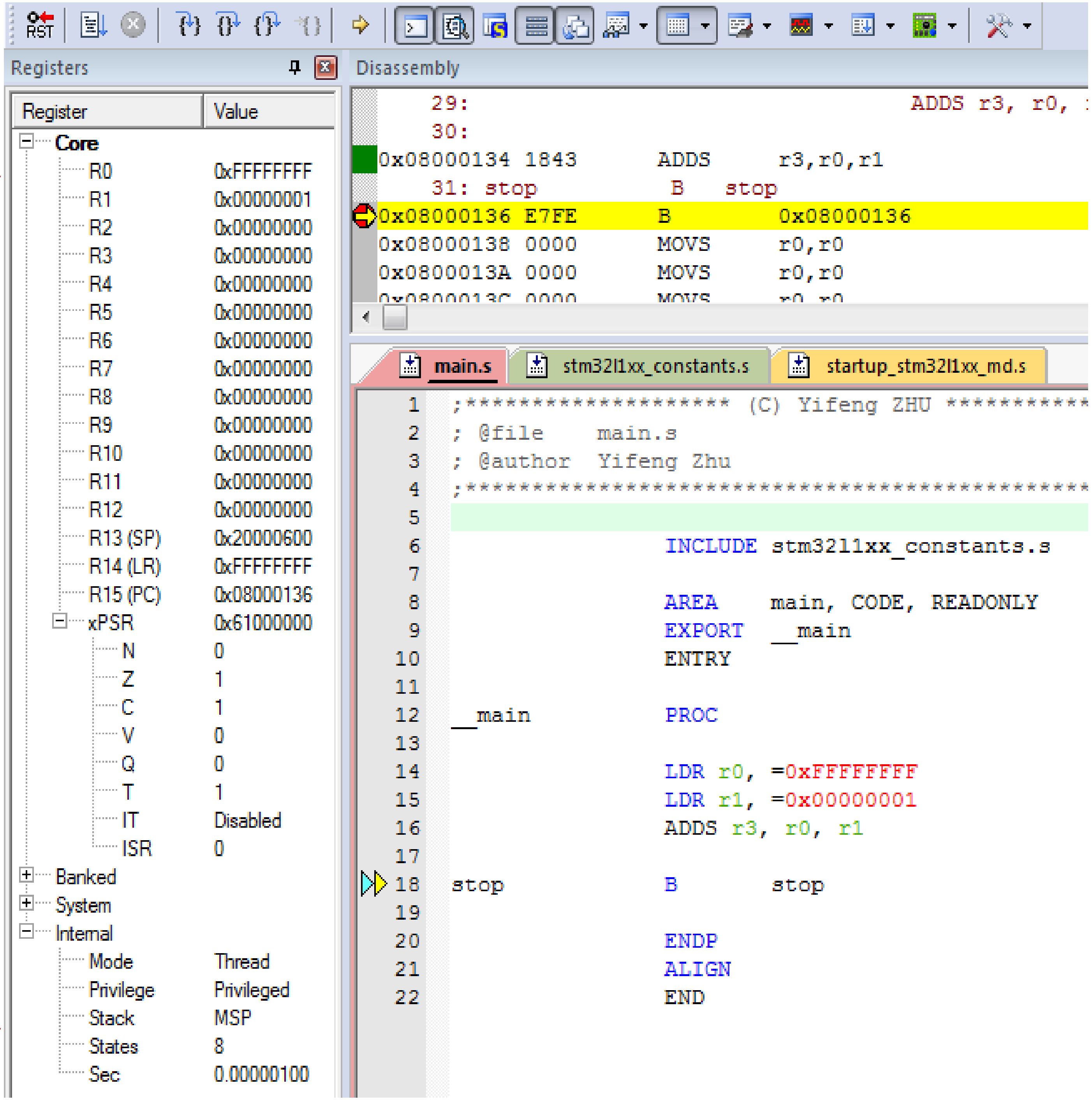
- ADD does not update flags
- ADDS updates flags
  - `xPSR.`**N** `= bit 31 of result`
  - `xPSR.`**Z** `= IsZero(result)`
  - `xPSR.`**C** `= carry, assuming r1 and r2 representing unsigned integers`
  - `xPSR.`**V** `= overflow, assuming r1 and r2 representing signed integers`

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



Example: 64-bit Addition

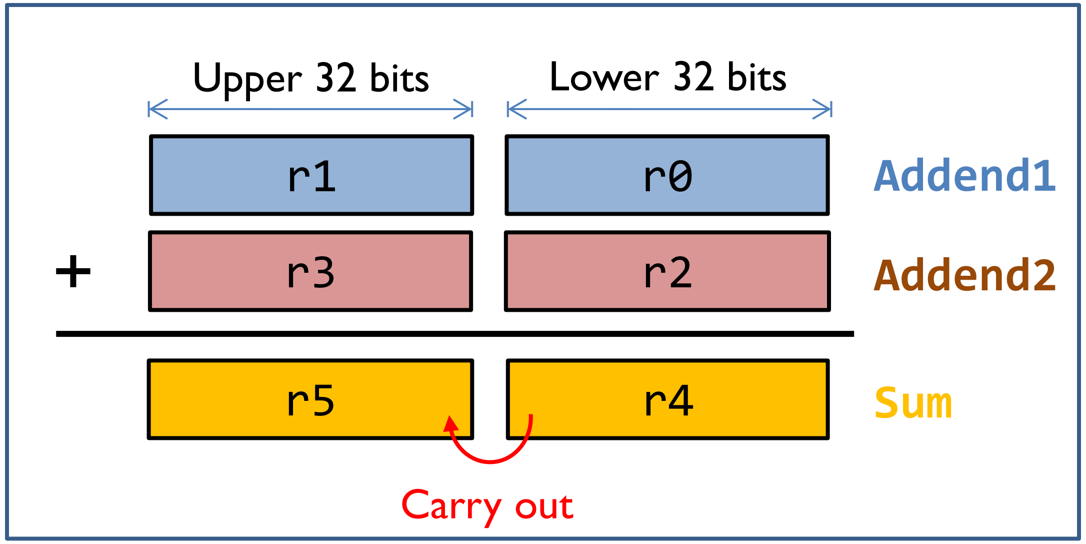
	Most-significant (Upper) 32 bits	Least-significant (Lower) 32 bits
	0 0 0 0 0 0 0 2	F F F F F F F F
+	0 0 0 0 0 0 0 4	0 0 0 0 0 0 0 1
	0 0 0 0 0 0 0 7 ←	0 0 0 0 0 0 0 0

Carry out

- A register can only store 32 bits
- A 64-bit integer needs two registers
- Split 64-bit addition into two 32-bit additions

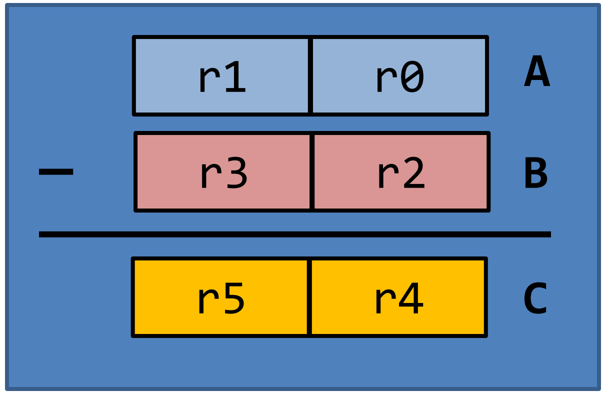
Example: 64-bit Addition

```
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
```

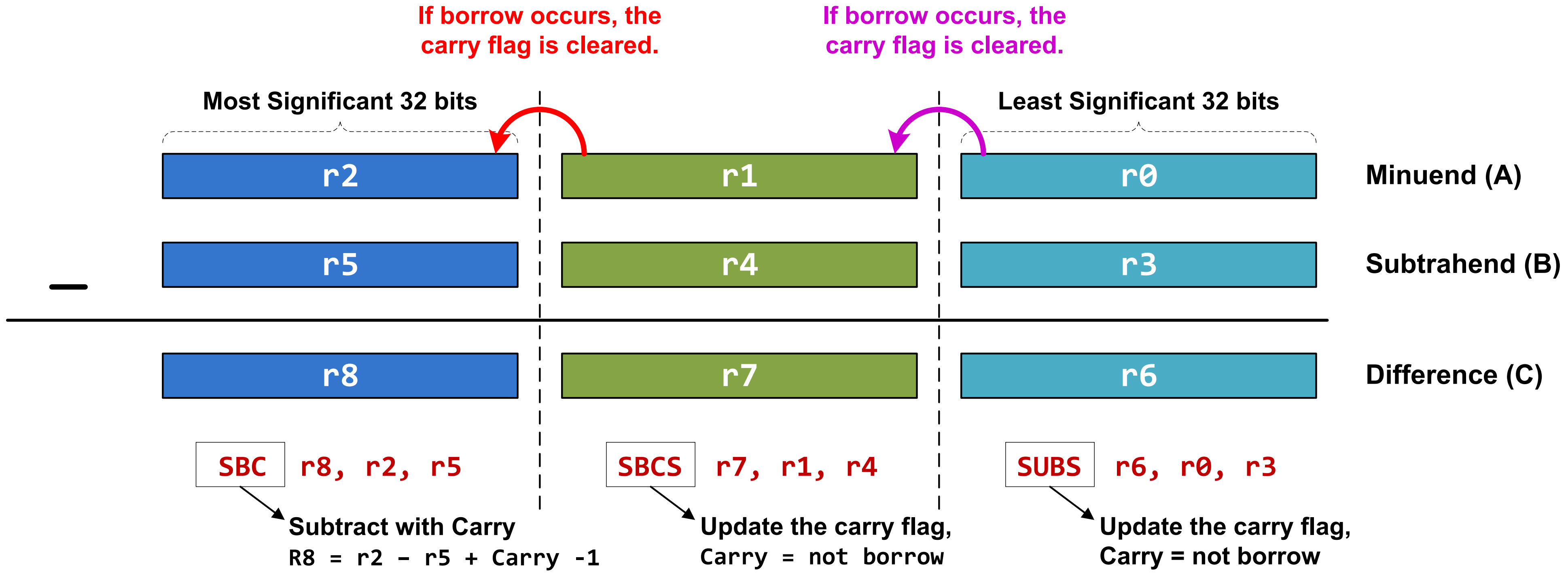


Example: 64-bit Subtraction

```
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
  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
SBCS r7, r1, r4
SBC  r8, r2, r5
```

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** RdHi,RdLo ← unsigned(Rn × Rm)
`SMULL RdLo, RdHi, Rn, Rm`	**Signed long multiply** RdHi,RdLo ← signed(Rn × Rm)
`UMLAL RdLo, RdHi, Rn, Rm`	**Unsigned multiply with accumulate** RdHi,RdLo ← unsigned(RdHi,RdLo + Rn × Rm)
`SMLAL RdLo, RdHi, Rn, Rm`	**Signed multiply with accumulate** 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** Rd ← Rn & operand2
ORR {Rd,} Rn, Op2	**Bitwise logic OR** Rd ← Rn | operand2
EOR {Rd,} Rn, Op2	**Bitwise logic exclusive OR** Rd ← Rn ^ operand2
ORN {Rd,} Rn, Op2	**Bitwise logic NOT OR** Rd ← Rn | (NOT operand2)
BIC {Rd,} Rn, Op2	**Bit clear** Rd ← Rn & NOT operand2
BFC Rd, #lsb, #width	**Bit field clear** Rd[(width+lsb-1):lsb] ← 0
BFI Rd, Rn, #lsb, #width	**Bit field insert** Rd[(width+lsb-1):lsb] ← Rn[(width-1):0]
MVN Rd, Op2	**Move NOT, logically negate all bits** Rd ← 0xFFFFFFFF EOR 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 0 1 0 1
r1 1 0 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 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 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 0 1
r1 1 0 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 1 1 1

Bit-wise Logic OR

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 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)
  - "~0x01" = 0xFFFFFFFE, but "!0x01" = 0x00. (!true = false)

Saturating Instruction: SSAT and USAT

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

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

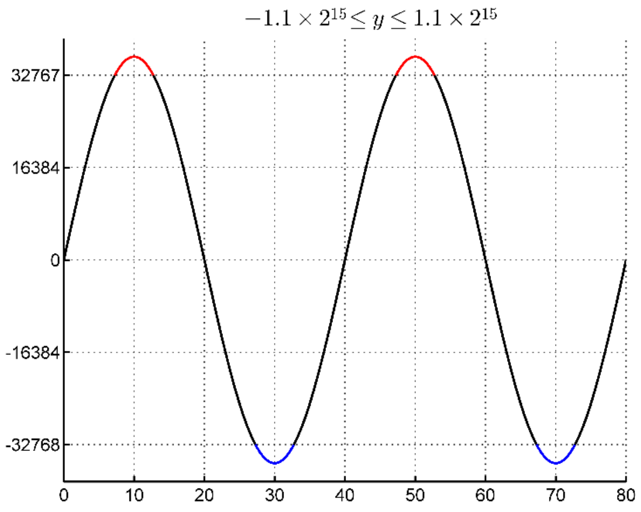
- USAT saturates a signed value to the unsigned range $0 \leq x \leq 2^n - 1$.

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

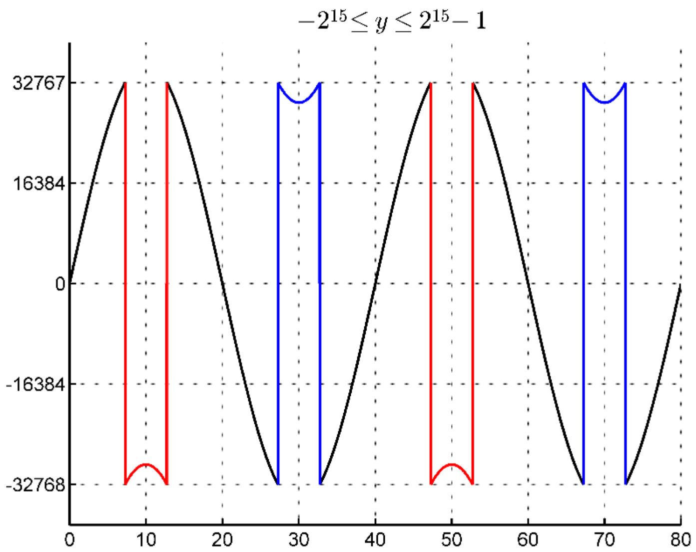
- Examples:
  - `SSAT r2, #11, r1     ; output range:` $-2^{10} \leq r2 \leq 2^{10}$
  - `USAT r2, #11, r3     ; output range:` $0 \leq r2 \leq 2^{11}$

Example of Saturation

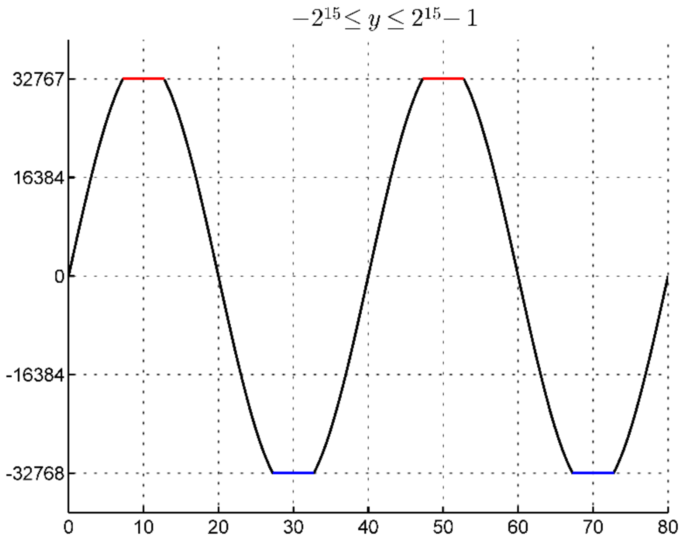
Assume data are limited to **16** bits



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

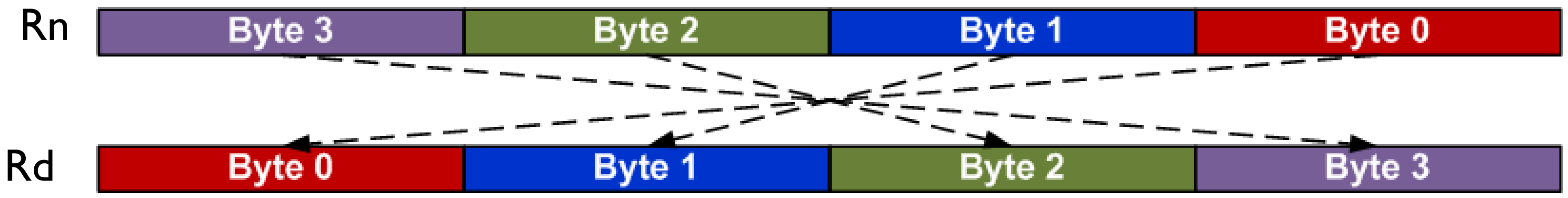
Example:

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

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



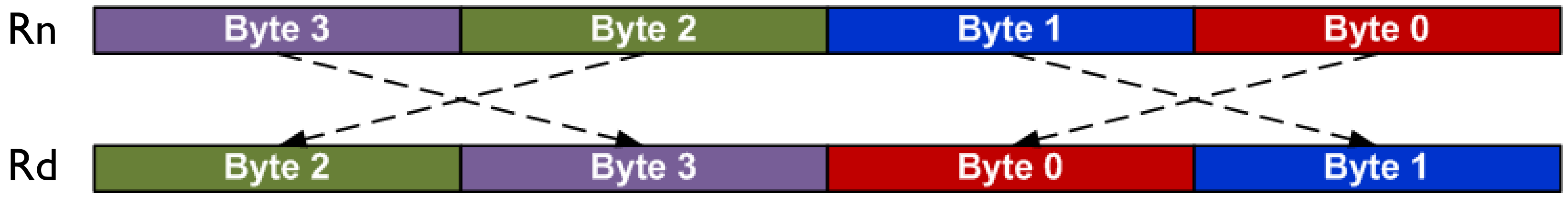
Example:

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

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 = 0xFFFFFFFE
```

Sign and Zero Extension

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

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

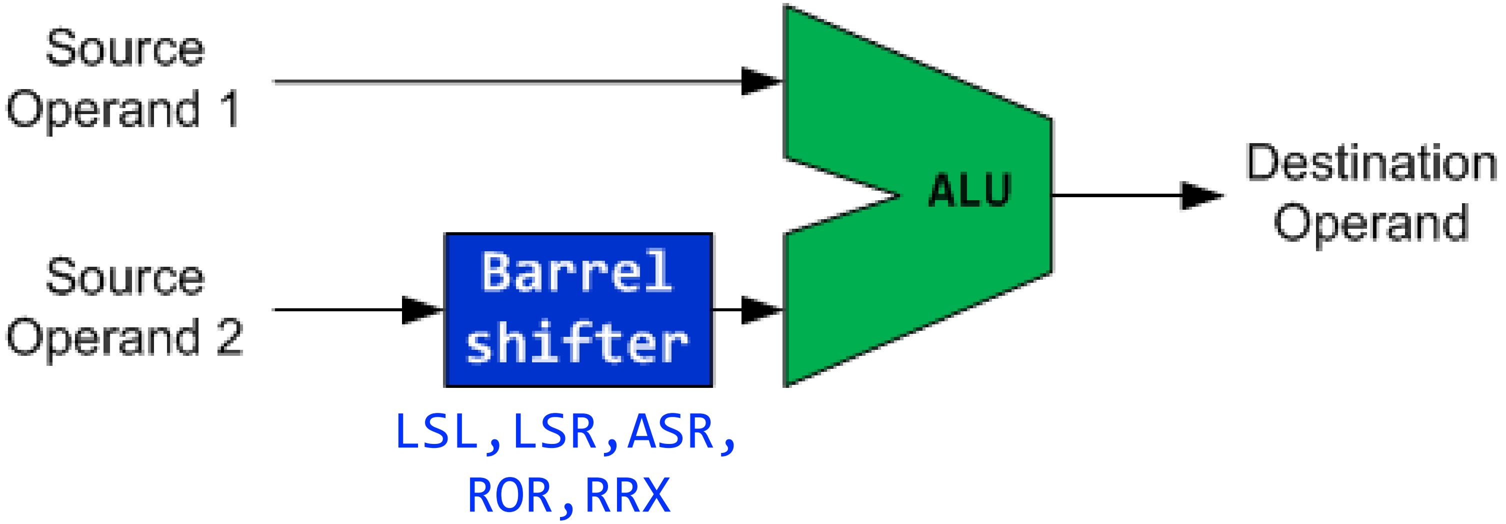
```
MOVW r0, #0x4321      ; r0 = 0x00004321
MOVT r0, #0x8765      ; r0 = 0x87654321
```

Order does matter!

- **MOVW** 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 << 2

Use Shifts To Implement Multiplication And Division

- Use Barrel shifter to speed up multiplication and division
  - Shifting left 1 bit <=> multiplication by 2
- Examples:
  - r1 = 9 × r0 = r0 + 8 × r0

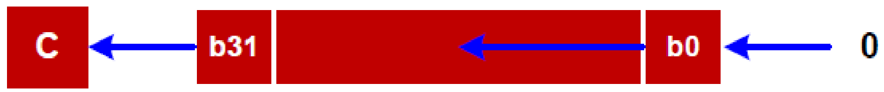
```
ADD r1, r0, r0, LSL #3  <=>  MOV r2, #9      ; r2 = 9
                              MUL r1, r0, r2 ; r1 = r0 * 9
```

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

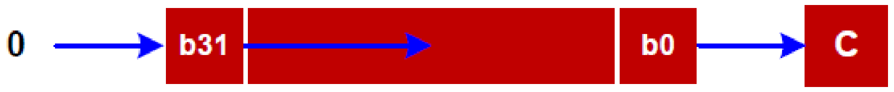
```
ADD r1, r0, r0, LSR #3
; r1 = r0 + r0 >> 3 = r0 + r0/8 (unsigned)

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

Barrel Shifter

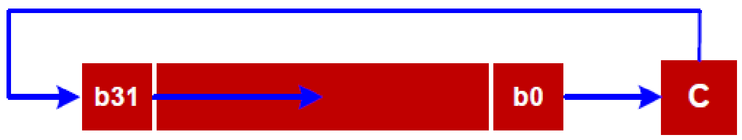
Logical Shift Left (**LSL**)



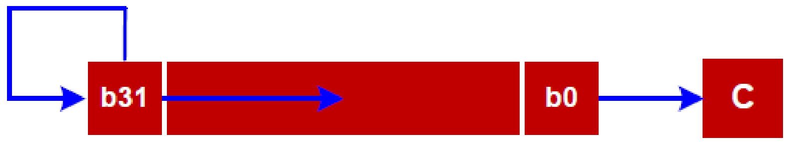
Logical Shift Right (**LSR**)



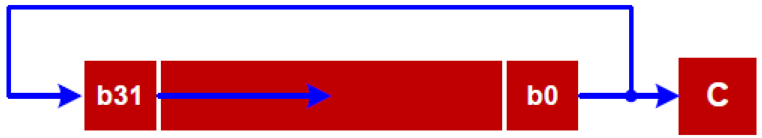
Rotate Right Extended (**RRX**)



Arithmetic Shift Right (**ASR**)



Rotate Right (**ROR**)



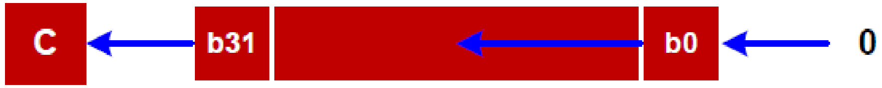
Why is there rotate right but no rotate left?

Rotate left can be replaced by a rotate right with a different rotate offset.

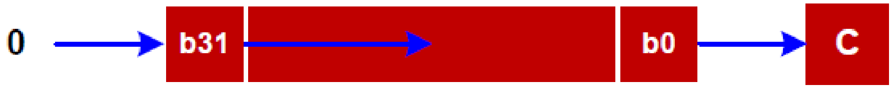
Updating APSR Flags

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

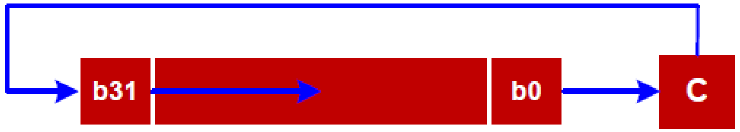
N	Z	C	V
R<31>	IsZeroBit(R)	carry	unchanged



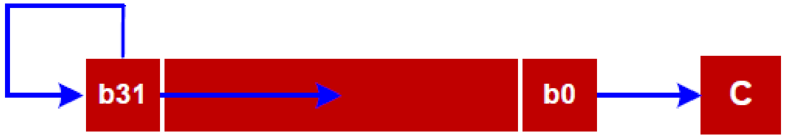
LSLS



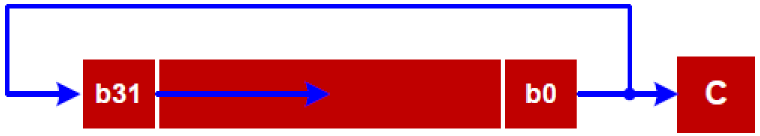
LSRS



RRXS



ASRS



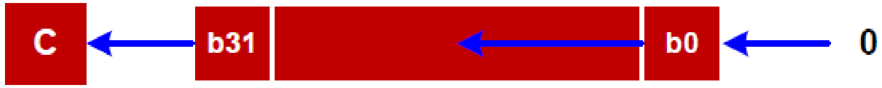
RORS

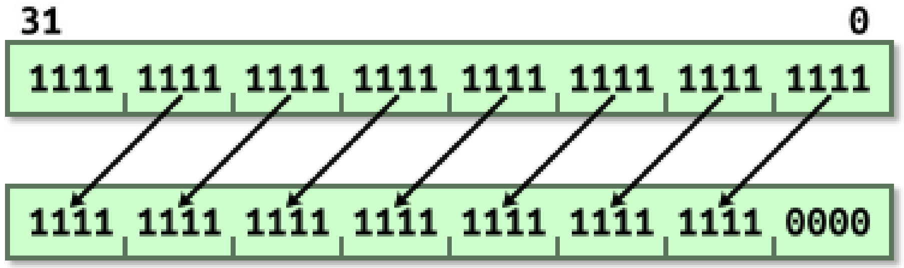
Barrel Shifter: Explanations

- 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.
- 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 hat 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.

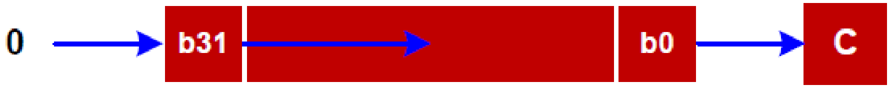
Examples (shifting by 4)

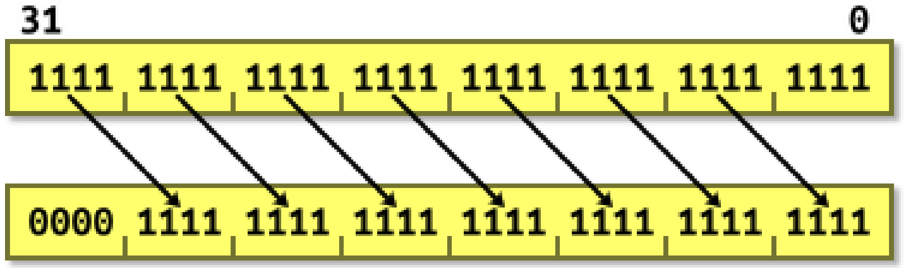
Logical Shift Left (**LSL**)

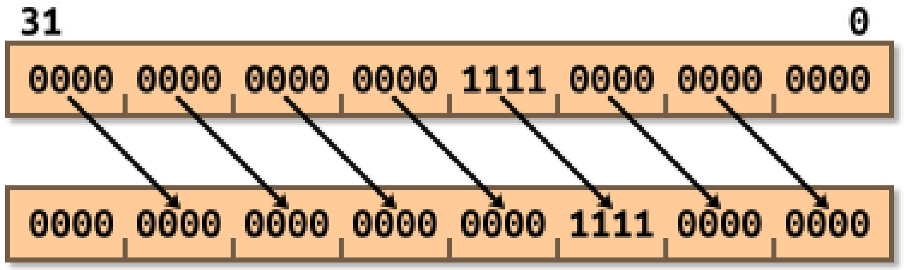




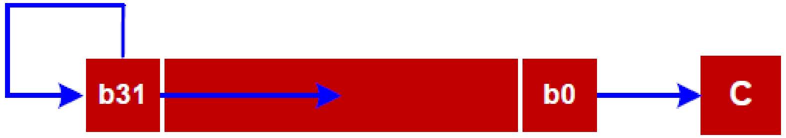
Logical Shift Right (**LSR**)

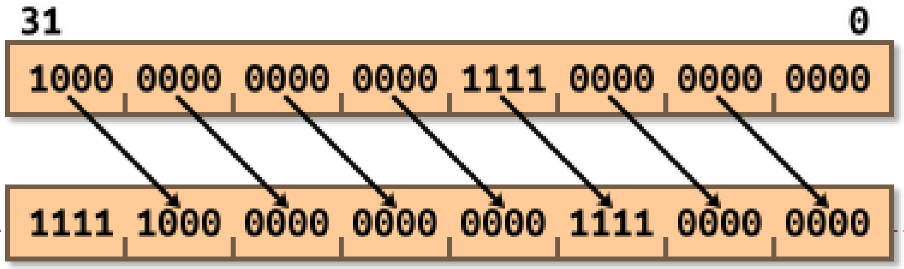






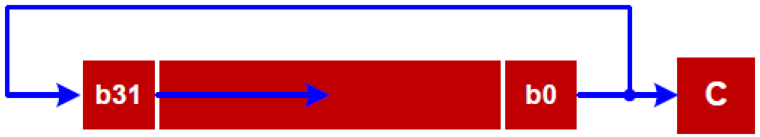
Arithmetic Shift Right (**ASR**)

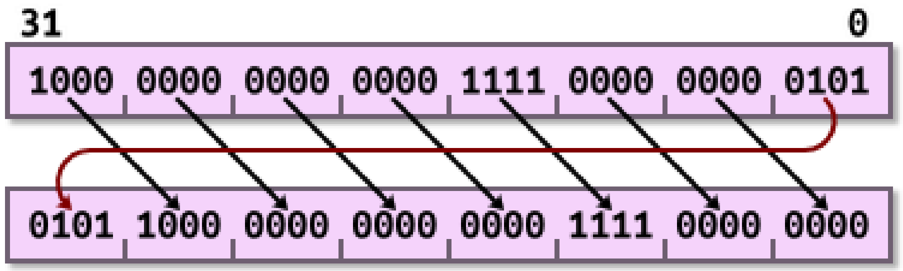




Examples (rotate)

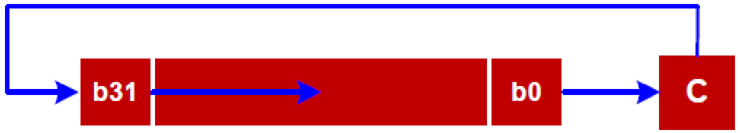
Rotate Right (**ROR**) (rotate by 4)

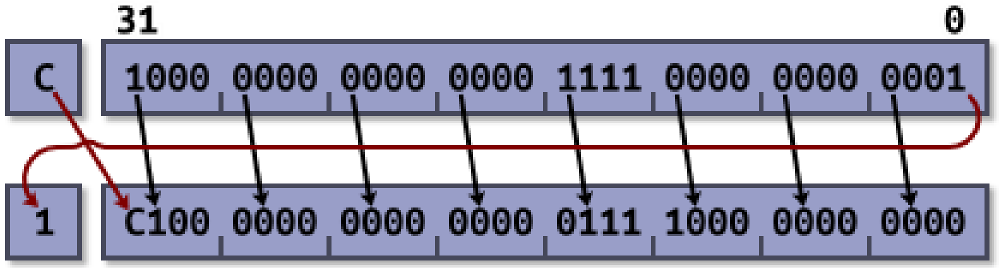




Rotate Right Extended (**RRX**)

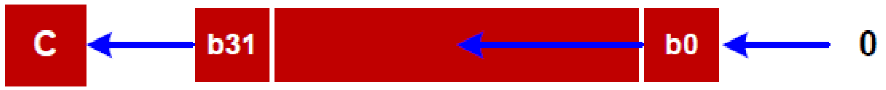
(can only rotate by 1)





Shift Operations

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

$$0 \le n \le 31$$

Example 1

; r2 = 0x0000_0001 (#1)

LSL r3, r2, #3

; r3 = 0x0000_0008 (#8)

; 8 = $2^3 * 1$

Example 2

; r2 = 0x0000_0003 (#3)

LSL r3, r2, #2

; r3 = 0x0000_000C (#12)

; 12 = $2^2 * 3$

Example 3

; r3 = 0xFFFF_0000 (#-65536)

LSLS r2, r3, #1

; r2 = 0xFFFE_0000 (#-131072)

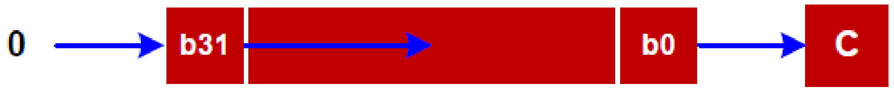
; -131072 = $2^1 * -65536$

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

Note: If the suffix S is used, the carry flag is updated to the value of the last shifted bit.

Shift Operations

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

$1 \le n \le 32$

Example 1

```
; r2 = 0x0000_0010 (#16)
LSR r1, r2, #3
; r1 =0x0000_0002 (#2)
```

; $2 = 16/2^3$

Example 2

```
; r2 = 0x8000_0000 (# -2,147,483,648)
LSR r2, r2, #2
; r2 = 0x2000_0000 (# 536,870,912)
```

; $536,870,912 = -2,147,483,648/2^2$

→ with LSR sign bit is lost (if r2 is a signed integer). So do not use logical shifts for signed integers!

Example 3

```
; r2 = 0x0000_0001 (#1)
LSRS r3, r2, #1
; r3 = 0x0000_0000 (#0)
```

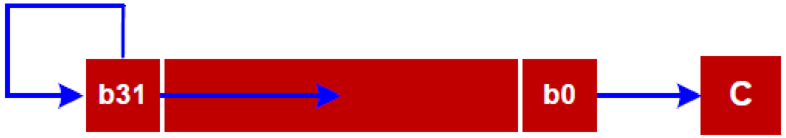
; $0 = 1/2^1$

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

Note: If the suffix S is used, the carry flag is updated to the value of the last shifted bit.

Shift Operations

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= −524288/2^3
```

ASR is equivalent to signed integer division

Example 2

```
; r2 = 0x8000_0000 (-2,147,483,648)
ASR r2, r2, #2
; r2 = 0xE000_0000 (# -536,870,912)
; -536,870,912= -2,147,483,648/2^2
```

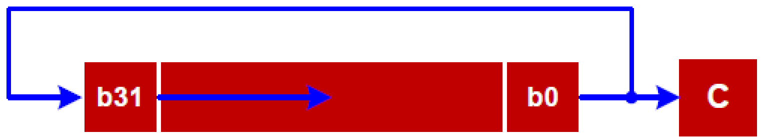
Example 3

```
; r2 = 0xFFFF_F001 (#-4095)
ASRS r3, r2, #1
; r3 = 0xFFFF_F800 (#-2048)
; -2048 = −4096/2^1
C=1, N=1, Z=0, V=not updated
```

Note: If the suffix S is used, the carry flag is updated to the value of the last shifted bit.

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

$$1 \le n \le 31$$

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 bits is equivalent to rotate right ROR by 32-m bits

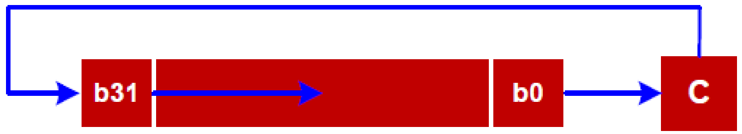
Example 3

```
;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
```

Note: If the suffix S is used, the carry flag is updated to the value of the last shifted bit.

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
```

Note: the carry flag is updated by b0 only if the suffix S is used, otherwise it keeps its original value

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)
- 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 - Operand2
- RSB instruction: RSB Rd, Rn, Operand2 performs Rd = 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 << 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. 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:
  - ADD r5, r3, r2, LSL #2 @ calculate address of array[i] in r5 with single ADD
  - LDR r6, [r5] @ load value of array[i] into r4 using the address in r5
- Alternatively, ARM has some nice addressing modes to speedup array item access:
  - LDR r6, [r3, r2, LSL #2]

Example 1: ANDS

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

Updates carry flag,
since ANDS does not update carry flag

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

```
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, since logical operations don't affect overflow. It was set by previous shift "r0, LSL #1" to be C=1
V is unaffected by either ANDS or shift (left unchanged from its previous value)
Note: LSL updates the C flag when it is used within the ANDS instruction, since ANDS does not update C.

Example 2: ADDS

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

Does NOT update carry flag, since ADDS updates flags

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

Note: LSL updates the C flag when it is used within the ADDS instruction. However, its update of the C flag is overwritten by ADDS, or equivalently, we say that LSL does not update the C flag.

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. The **S** suffix (e.g. LSLS R0, R0, #1) makes the instruction update **NZCV**.
- When a shift appears as part of a data-processing instruction that ends with **S**, the processor first computes the shifted operand. 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

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:

```
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 &= ~(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:
    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 ^= 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

Truth table of **Exclusive OR**

m	n	m⊕n
0	0	0
0	1	1
1	0	1
1	1	0

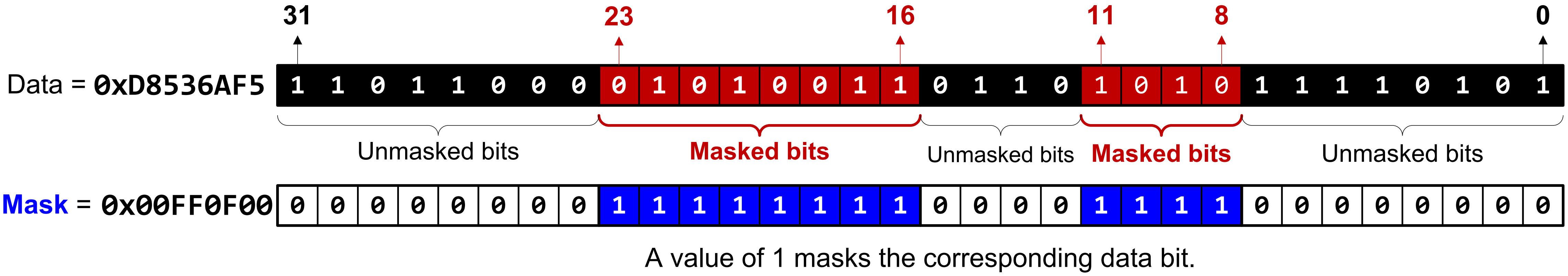
Toggle a Bit in Assembly

```
a ^= 1<<5
```

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

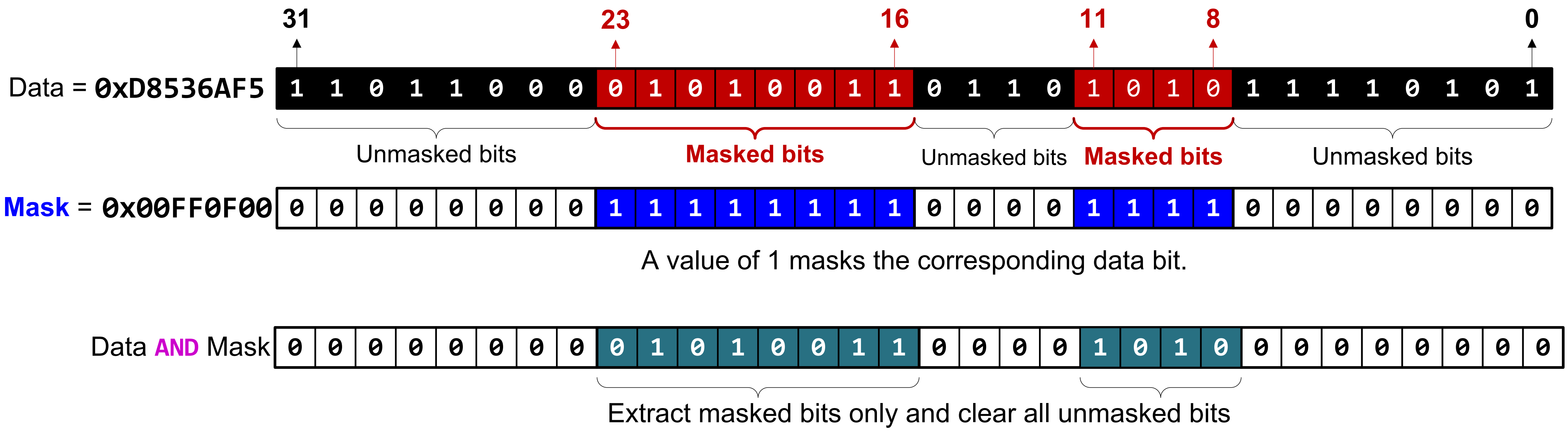
Here we can use MOVS and EORS instead of MOV and EOR, if the flags are used by later instructions.

Mask



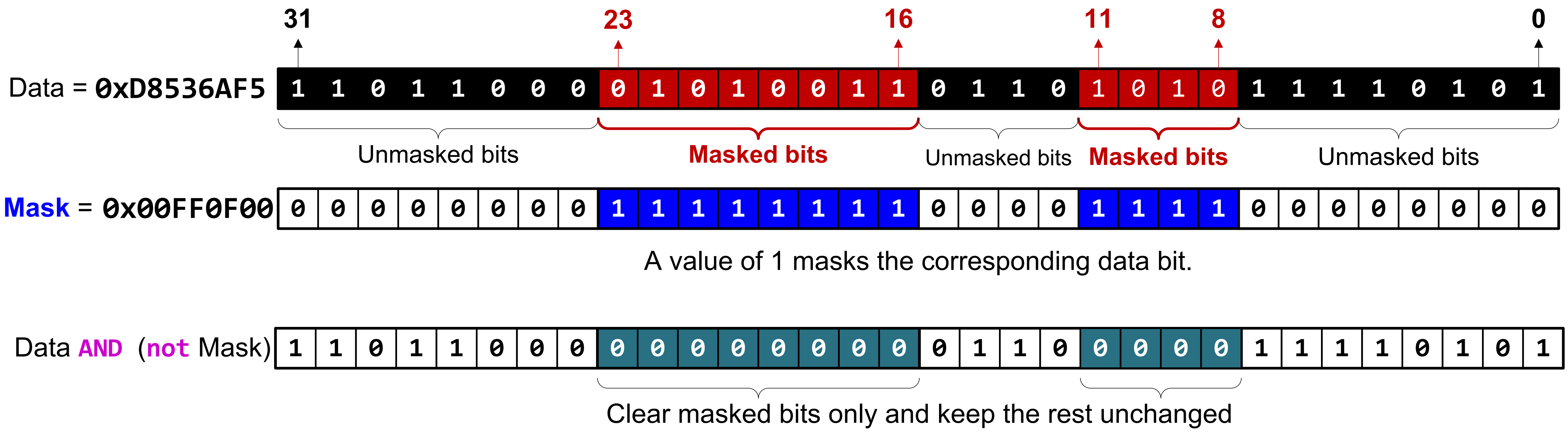
- Bits 8-11 and bits 16-23 are **masked**.
- All the rest bits are **unmasked**

Clear all unmasked bits



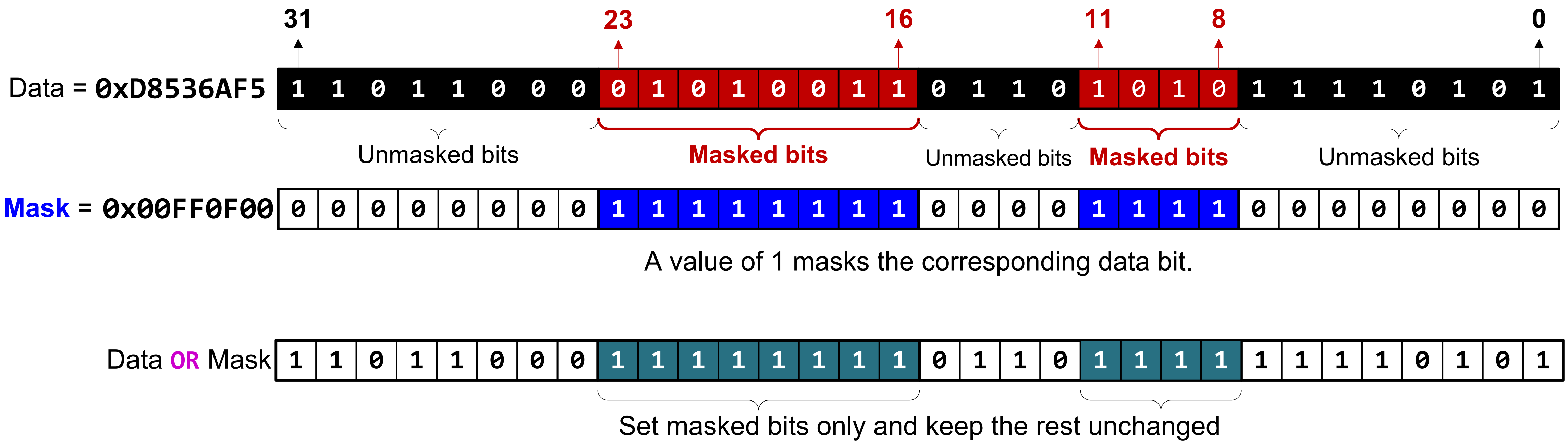

```
Data &= Mask;
```

Clear all masked bits



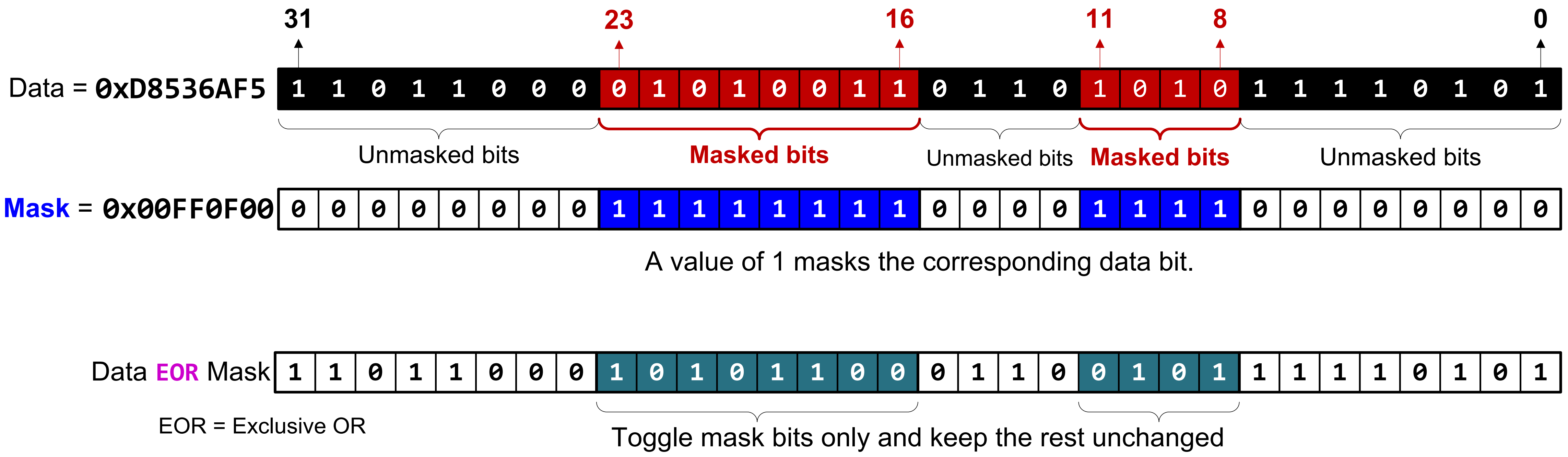

```
Data &= ~Mask;
```

Set all masked bits




```
Data |= Mask;
```

Toggle all tasked bits




```
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; Overflow cannot occur when adding 2 operands with different signs or when subtracting 2 operands with the same sign.
If two operands have same sign, and the result has opposite sign, then V = 1; else V = 0
Tip: Convert subtraction to addition with Two's complement.

	Unsigned Addition	Unsigned Subtraction	Signed Addition or Subtraction
Carry flag	true result > 2^n-1 ➔ Carry flag=1 Borrow flag=0 (Result incorrect)	true result < 0 ➔ 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} ➔ Overflow flag=1 (Result incorrect)

Example

- For an 8-bit system, calculate 0x35 + 0x19, setting C and V flags
- 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
C = 0	0	0	0	1	1	0	0	1	Op2: 0x19
V = 0	0	1	0	0	1	1	1	0	Result: 0x4E

Example

- For an 8-bit system, calculate 0x35 + 0x5B, setting C and V flags
- 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 → result = -112 (result incorrect, V=1)

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

Example

- 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 negative
    - 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)

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

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 → result = +113 (wrong, V=1)

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

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

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

Dec	Hex									
			1	1	1	1	1	1	1	
53	35		0	0	1	1	0	1	0	1
+91	+5B		0	1	0	1	1	0	1	1
144	90	C = 0	1	0	0	1	0	0	0	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									
			1	1	1	0	1	1	1	
53	35		0	0	1	1	0	1	0	1
- 45	+D3		1	1	0	1	0	0	1	1
8	108	C = 1	0	0	0	0	1	0	0	0
		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									
			0	0	1	1	1	1	0	
- 98	9E		1	0	0	1	1	1	1	0
- 45	+D3		1	1	0	1	0	0	1	1
- 143	171	C = 1	0	1	1	1	0	0	0	1
		V = 1								

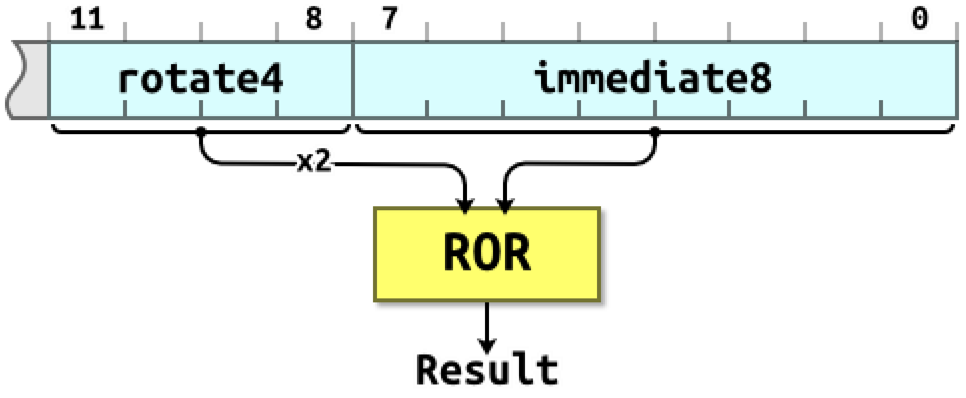
Ignore carry

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

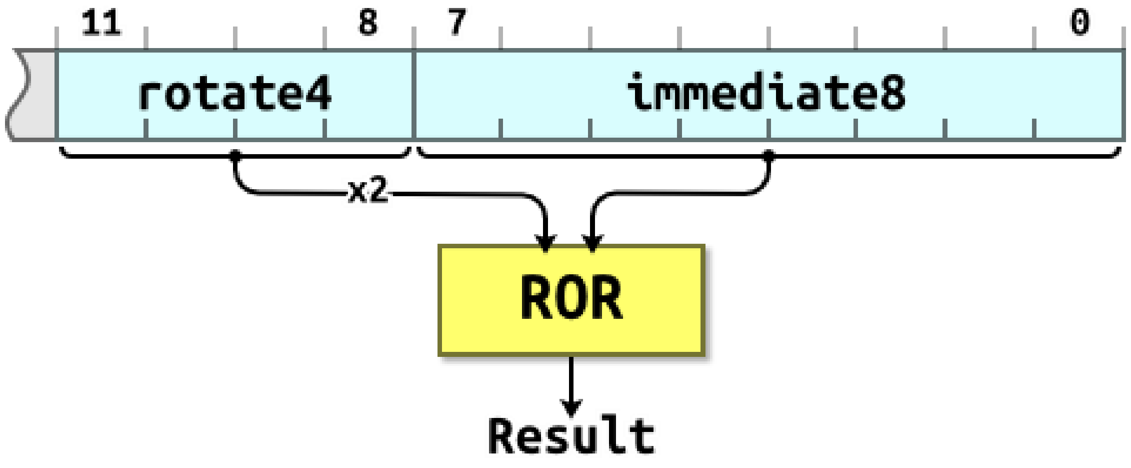
- You can't fit an arbitrary 32-bit value into a 32-bit instruction word. ARM data processing instructions have 12 bits of space for values in one 32-bit instruction word. This is arranged as a 4 rotate value and an 8 immediate value. The real immediate = **ROR(immediate8, rotate4 × 2)**.
  - The 4-bit 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
  - 0x00000FF0
  - 0xFF000000
  - 0xF000000F
- 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, #0xFFFFFF8F



12-bit immediate value

Encoding #4080 as Immediate

- ADD r1, r2, #4080
  - 4080 = 111111110000 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) = 0x00000FF0 (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=PLRJhV4hUhIymmp5CCeIFPyxbknsdcXCc8&index=25
- Lecture 26. Updating NZCV bit flags
  - https://www.youtube.com/watch?v=SGJibM1D2_A&list=PLRJhV4hUhIymmp5CCeIFPyxbknsdcXCc8&index=26