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;
```

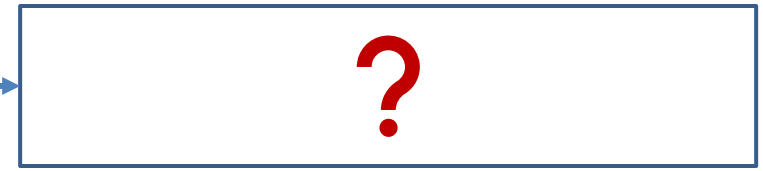
If values are in registers

- 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

```
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

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

Destination

Source Operand 2

Source Operand 1

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

```
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

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

ADD works for both signed and unsigned add operations.

Adding Two Integers

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

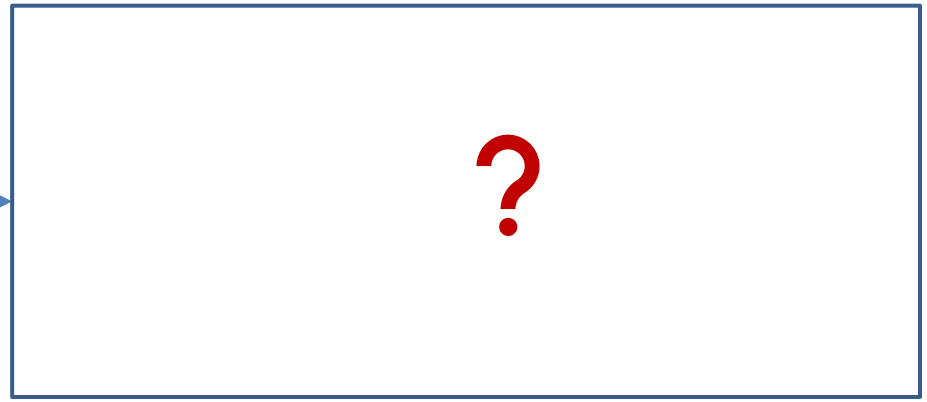
If addresses are in registers

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

C Statement

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

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
```

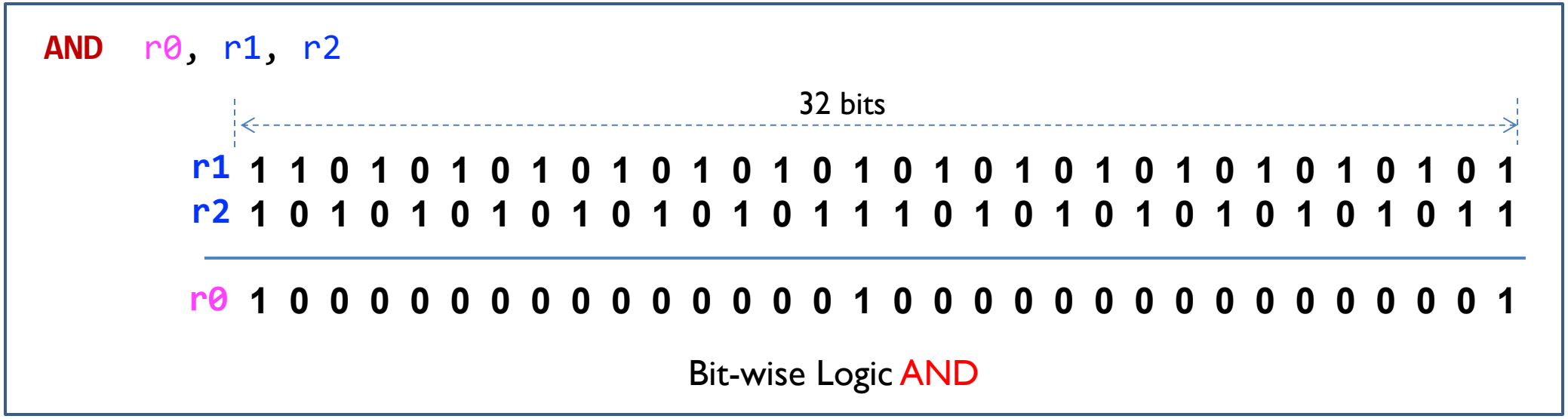
Load, modify, and store

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

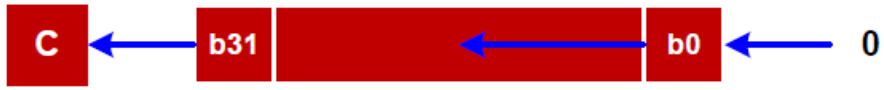
- `AND  r0, r1, r2 ; Bitwise AND, r0 = r1 AND r2`
- `ORR  r0, r1, r2 ; Bitwise OR, r0 = r1 OR r2`
- `EOR  r0, r1, r2 ; Bitwise Exclusive OR, r0 = r1 EOR r2`
- `ORN  r0, r1, r2 ; Bitwise OR NOT, r0 = r1 ORN r2`
- `BIC  r0, r1, r2 ; Bit clear, r0 = r1 & ~r2`



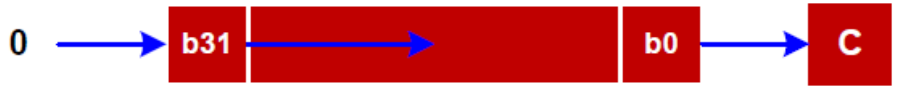
Example Shift & Rotate Instructions

- `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`

Logical Shift Left (**LSL**)



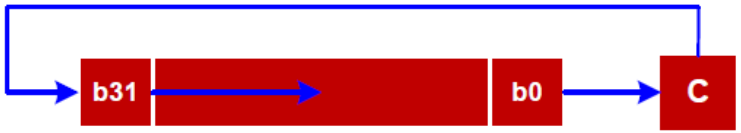
Logical Shift Right (**LSR**)



Arithmetic Shift Right (**ASR**)

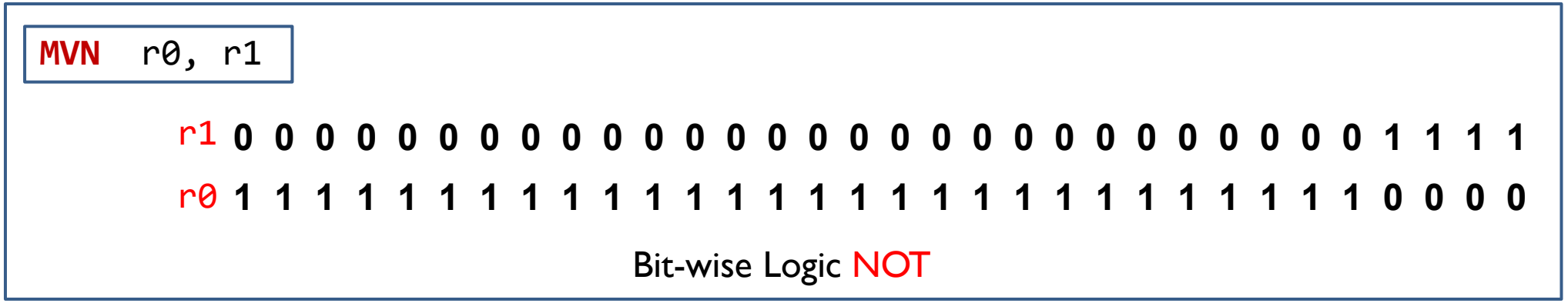
Rotate Right (**ROR**)

Rotate Right Extended (**RRX**)



Example Data Transfer Instructions

- `MOV  r0, r1 ; Move, r0 = r1`
- `MVN  r0, r1 ; Move NOT, r0 = bitwise NOT r1`



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

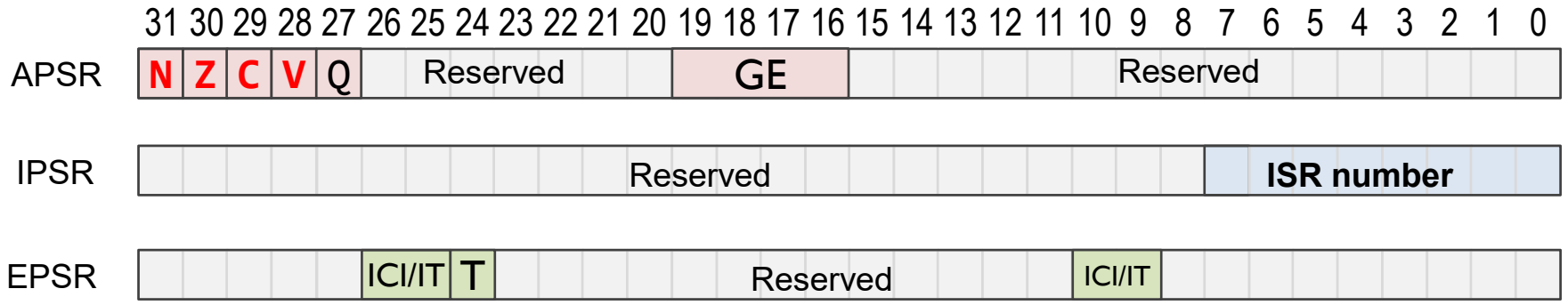
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13: Stack Pointer (SP)
R14: Link Register (LR)
R15: Program Counter (PC)

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**)

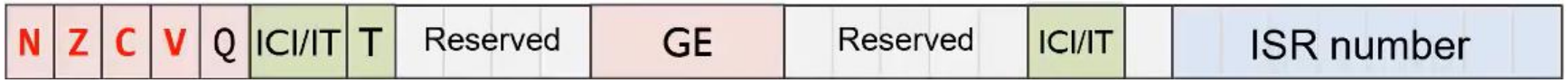


Combine them together into one register (**PSR**)

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
```

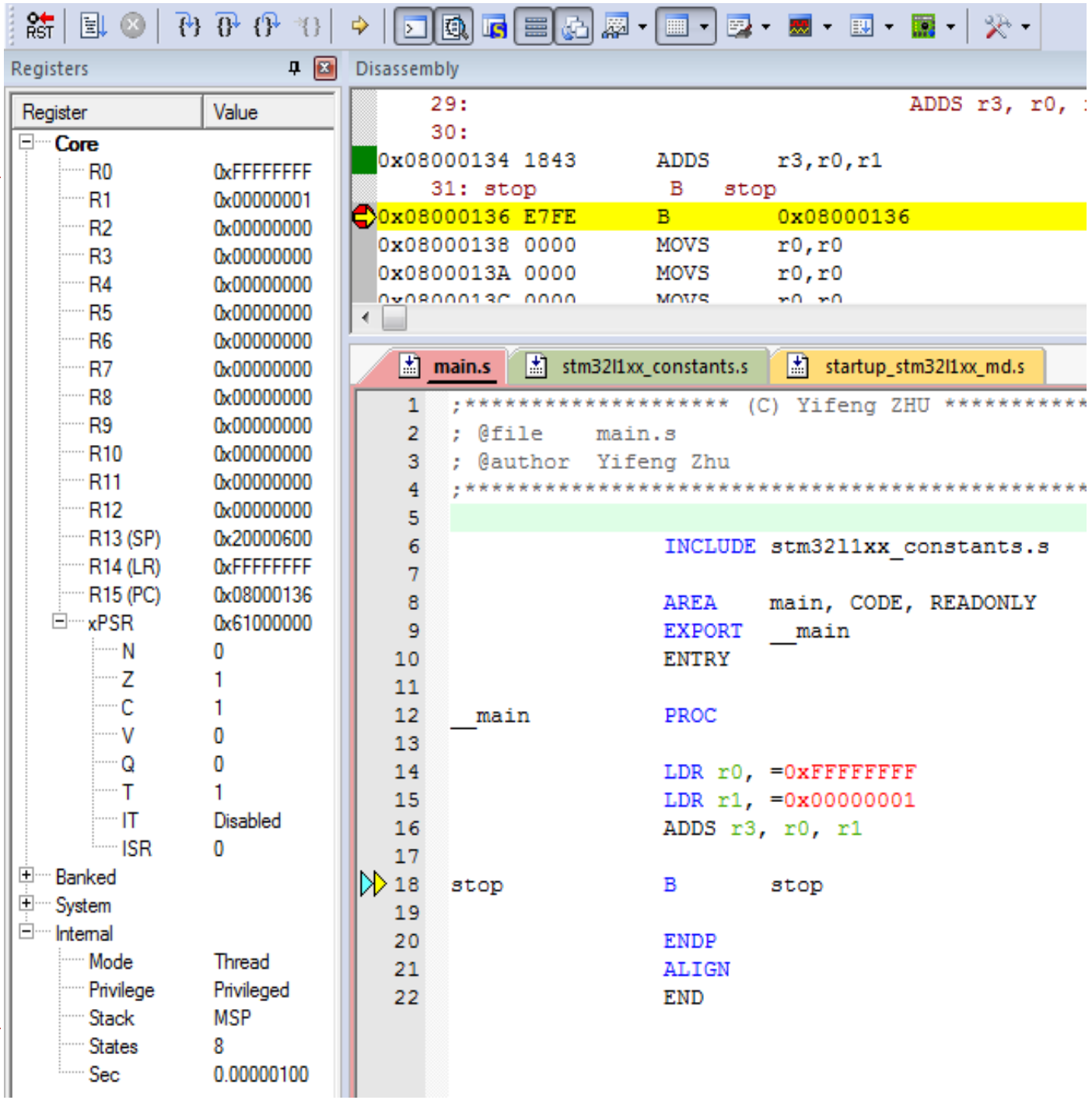
- ADD does not update flags
- ADDS updates flags
  - xPSR.**N** = bit 31 of result
  - xPSR.**Z** = IsZero(result)
  - xPSR.**C** = carry, assuming r1 and r2 representing unsigned integers
  - xPSR.**V** = overflow, assuming r1 and r2 representing signed integers

Suffix S: Update Flags

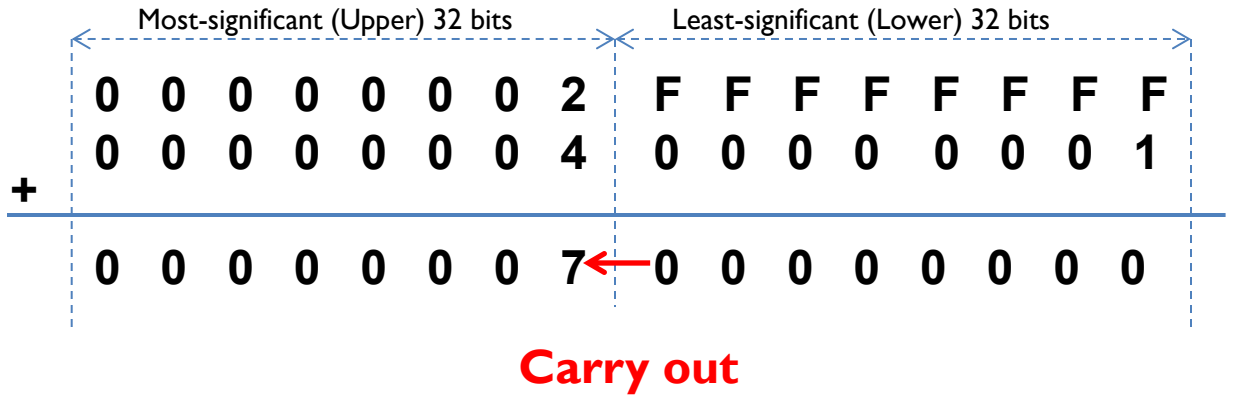
```
LDR  r0, =0xFFFFFFFF
LDR  r1, =0x00000001
ADDS r0, r0, r1
```

```
   0xFFFFFFFF   r0
 + 0x00000001   r1
-----------------
   0x00000000   sum
```

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



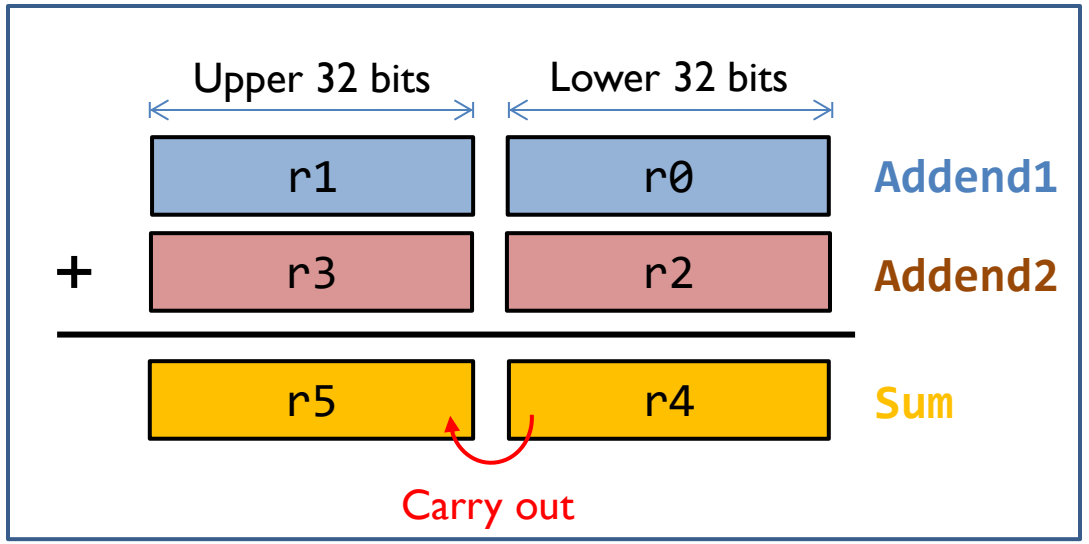
Example: 64-bit Addition



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

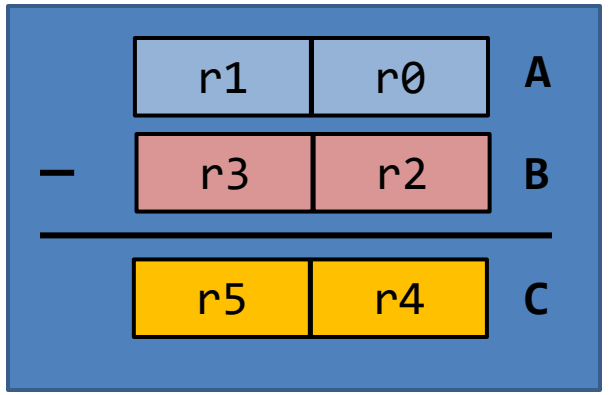
Example: 64-bit Addition

```
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
```

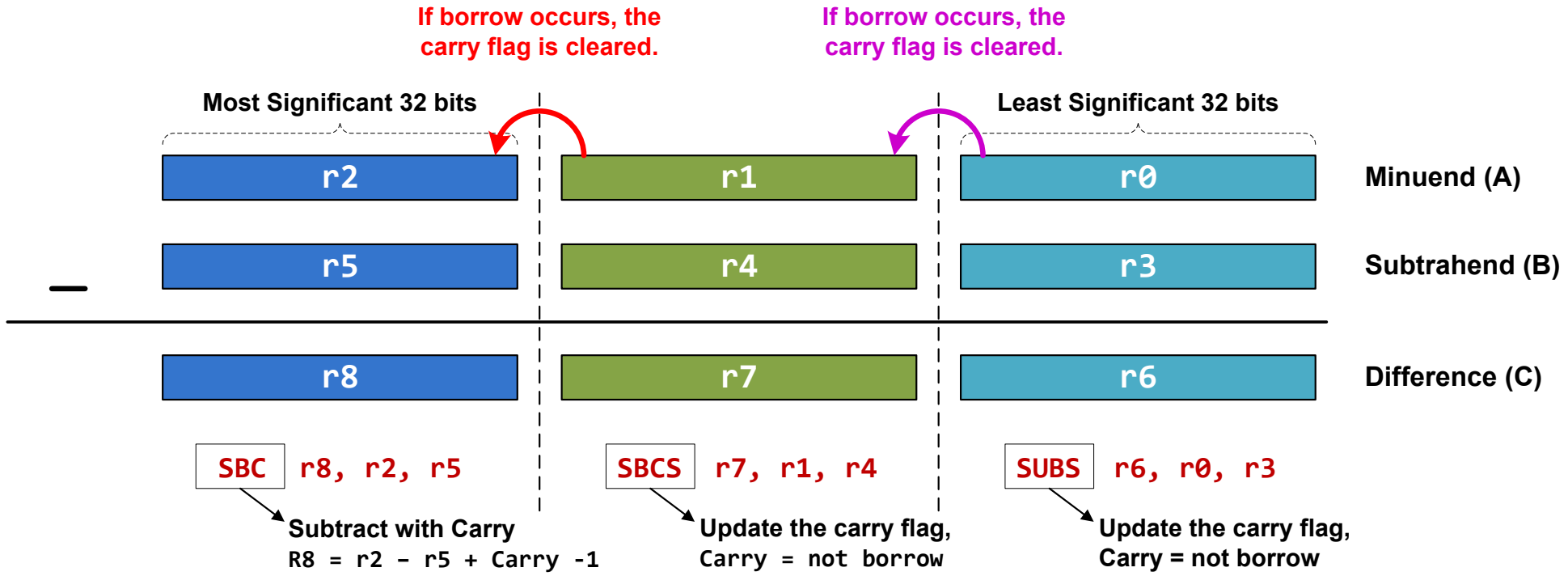


Example: 64-bit Subtraction

```
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)

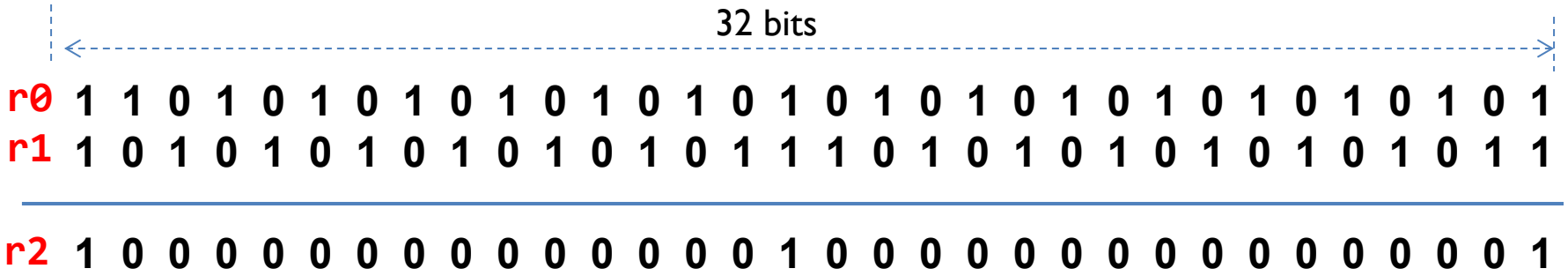
The result has 64 bits, placed in two registers.

```
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

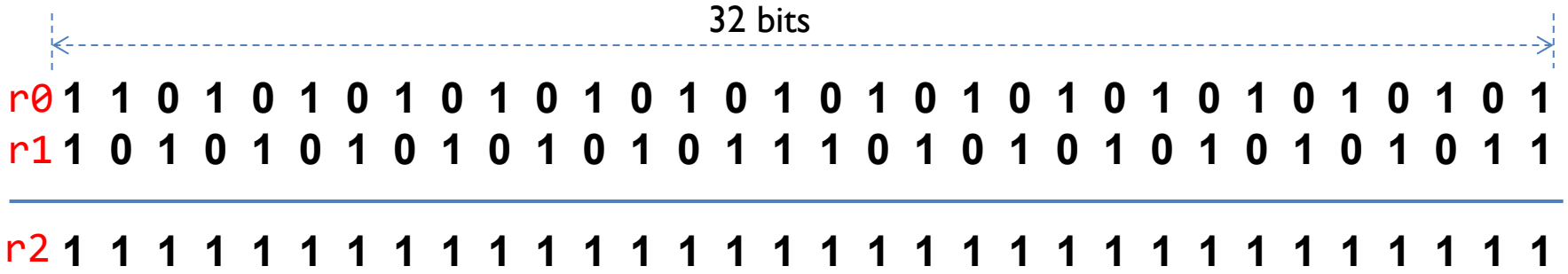
AND {Rd,} Rn, Op2	**Bitwise logic AND** Rd ← Rn & operand2
ORR {Rd,} Rn, Op2	**Bitwise logic OR** Rd ← Rn | operand2
EOR {Rd,} Rn, Op2	**Bitwise logic exclusive OR** Rd ← Rn ^ operand2
ORN {Rd,} Rn, Op2	**Bitwise logic NOT OR** Rd ← Rn | (NOT operand2)
BIC {Rd,} Rn, Op2	**Bit clear** Rd ← Rn & NOT operand2
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



Bit-wise Logic AND

Example: `ORR r2, r0, r1`



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.`
  - `"~0x01" = 0xFFFFFFFE,  but  "!0x01" = 0x00.`

Saturating Instruction: SSAT and USAT

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

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

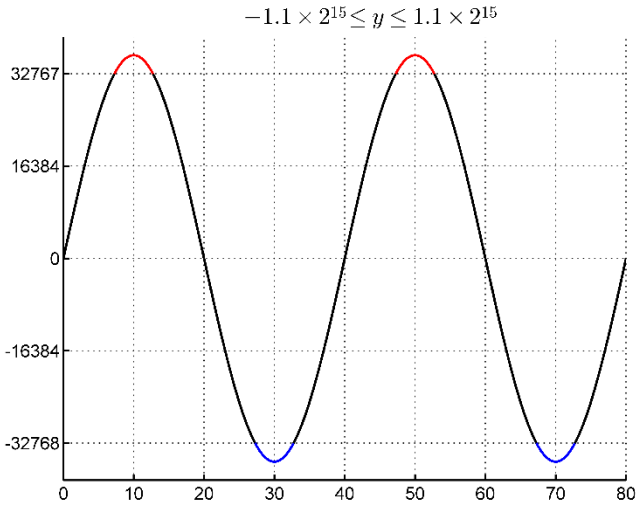
- USAT saturates a signed value to the unsigned range $0 \le x \le 2^n - 1$.

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

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

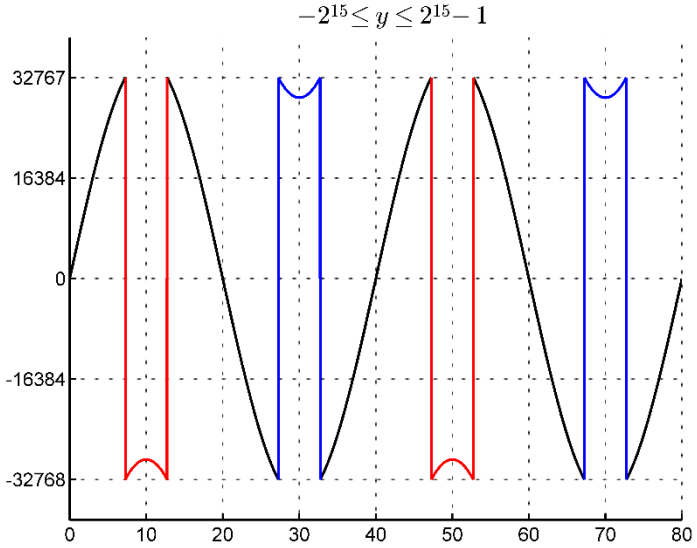
Example of Saturation

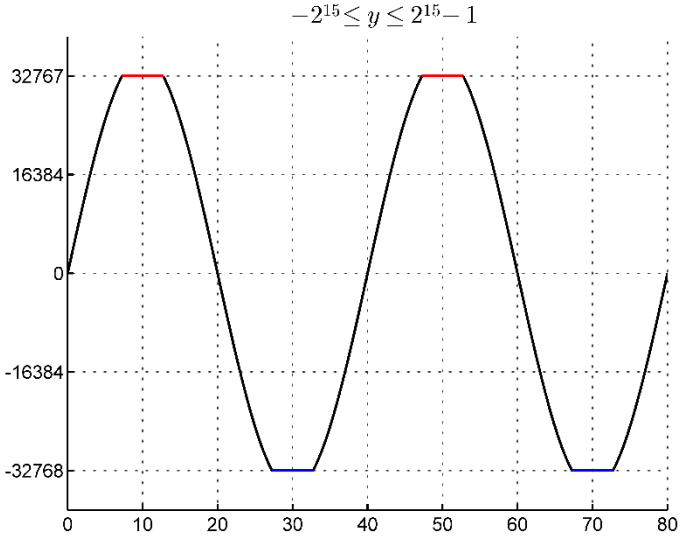
Assume data are limited to **16** bits



Without saturation

With saturation

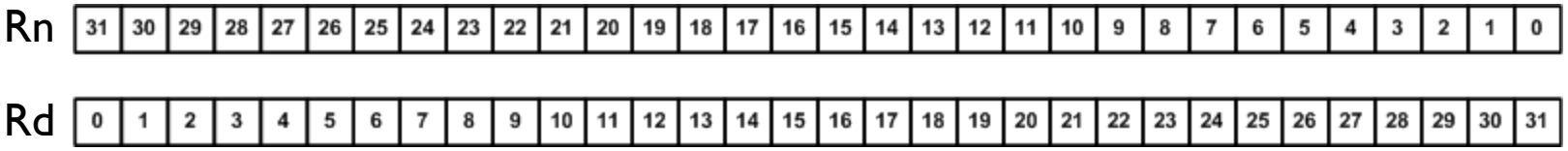




Reverse Order

RBIT Rd, Rn	**Reverse bit order in a word** 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



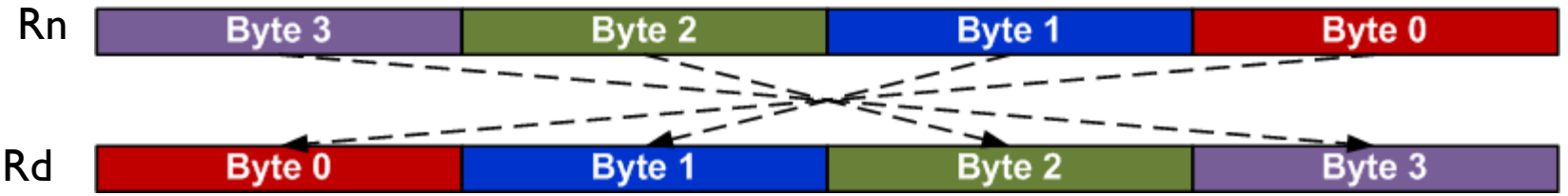
Example:

```
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



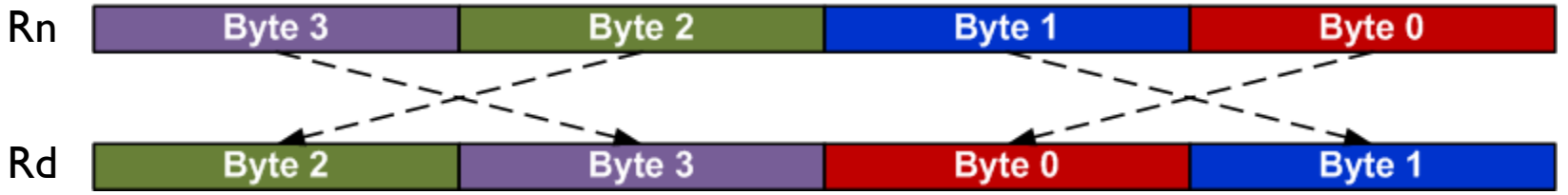
Example:

```
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



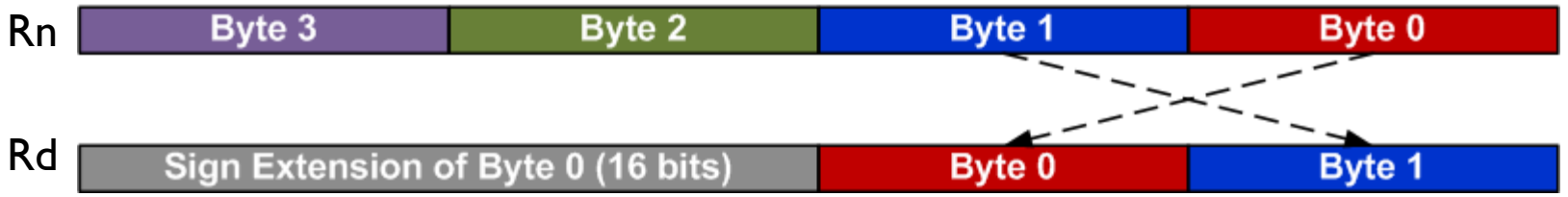
Example:

```
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

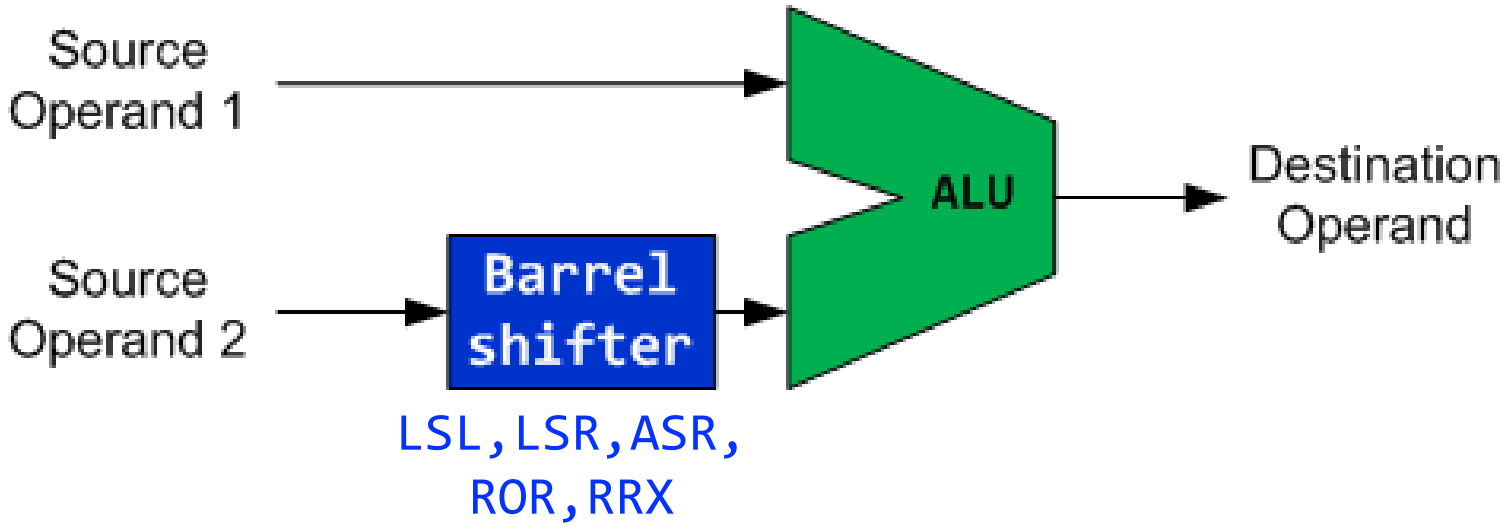
```
MOVW r0, #0x4321     ; r0 = 0x00004321
MOVT r0, #0x8765     ; r0 = 0x87654321
```

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
```

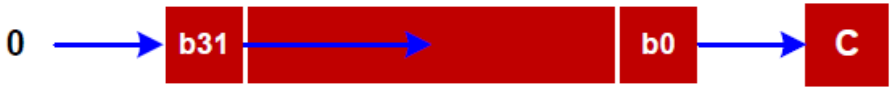
MUL instruction takes only registers, not an immediate, so "MUL r1, r0, #9" is invalid syntax

```
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)
```

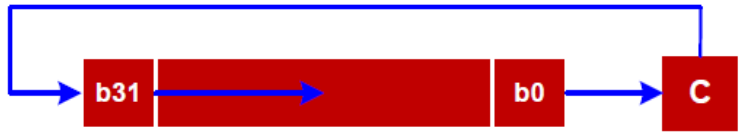
Barrel Shifter

Logical Shift Left (**LSL**)

Logical Shift Right (**LSR**)



Rotate Right Extended (**RRX**)



Arithmetic Shift Right (**ASR**)

Rotate Right (**ROR**)

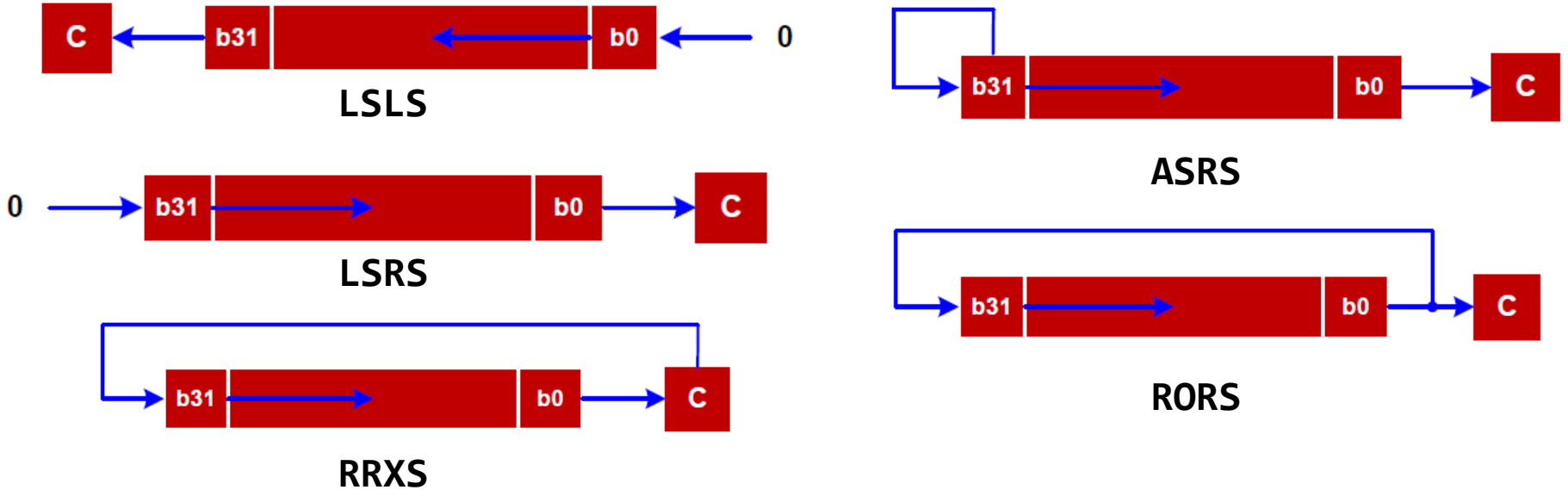
Why is there rotate right but no rotate left?

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



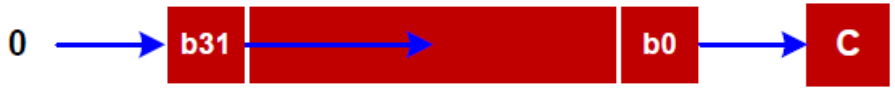
Barrel Shifter: Explanations

- 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.

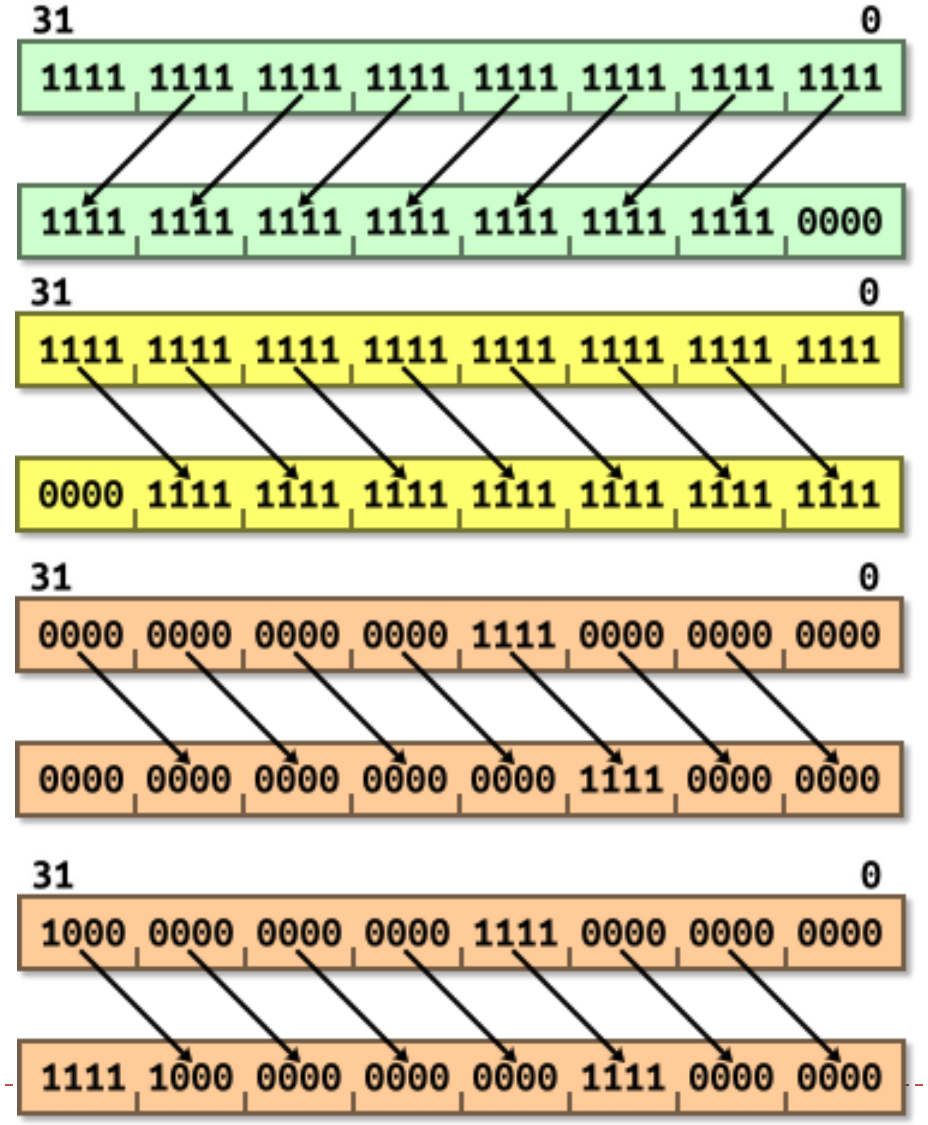
Examples (shifting by 4)

Logical Shift Left (**LSL**)

Logical Shift Right (**LSR**)

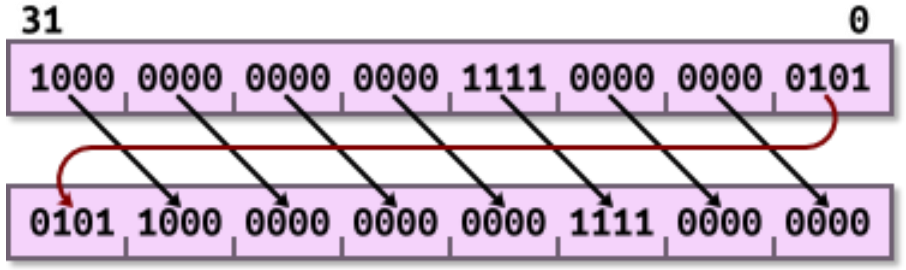


Arithmetic Shift Right (**ASR**)



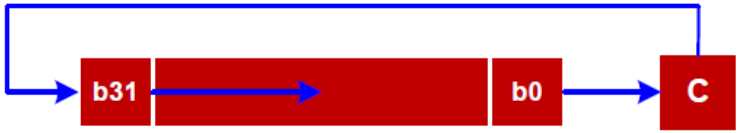
Examples (rotate)

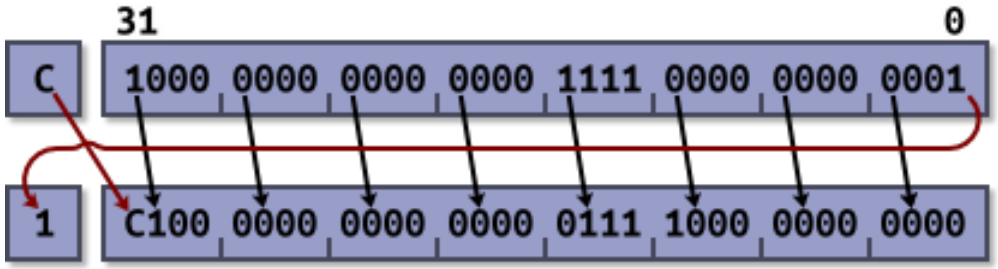
Rotate Right (**ROR**) (rotate by 4)



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
```

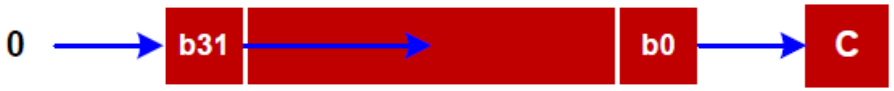
Example 3

```
; 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 !

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

$$1 \le n \le 32$$

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
; r1 = 0x0000_000E
RORS r3, r2, r1
; r3 = 0xC007_C3C3
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 = 0
```

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 flags

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. It was set be previous shift "r0, LSL #1" to be C=1
V is unaffected by ANDS or shift (left unchanged from its prior value)

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.)

Set a Bit in C

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

or

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

Example: k = 5

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

The other bits should not be affected.

Set a Bit in Assembly

a |= (1 << 5)

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

```
Solution 2:
    MOVS r4, #1                 ; r4 = 1
    ORRS r0, r0, r4, LSL #5     ; r0 = r0 | 1<<5
```

Clear a Bit in C

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

Example: k = 5

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

The other bits should not be affected.

Clear a Bit in Assembly

a &= ~(1<<5)

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

```
Solution 2:
    MOVS r4, #1             ; r4 = 1
    MVNS r4, r4, LSL #5     ; r4 = not (1<<5)
    ANDS r0, r0, r4         ; r0 = r0 & not (1<<5)
```

```
Solution 3:
    MOVS r4, #1                 ; r4 = 1
    BICS r0, r0, r4, LSL #5     ; r0 = r0 & not (1<<5)
```

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**

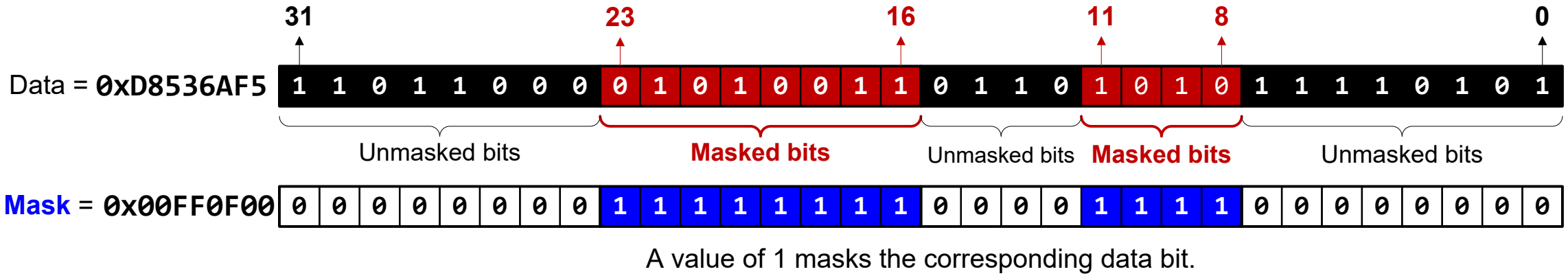
m	n	m⊕n
0	0	0
0	1	1
1	0	1
1	1	0

Toggle a Bit in Assembly

a ^= 1<<5

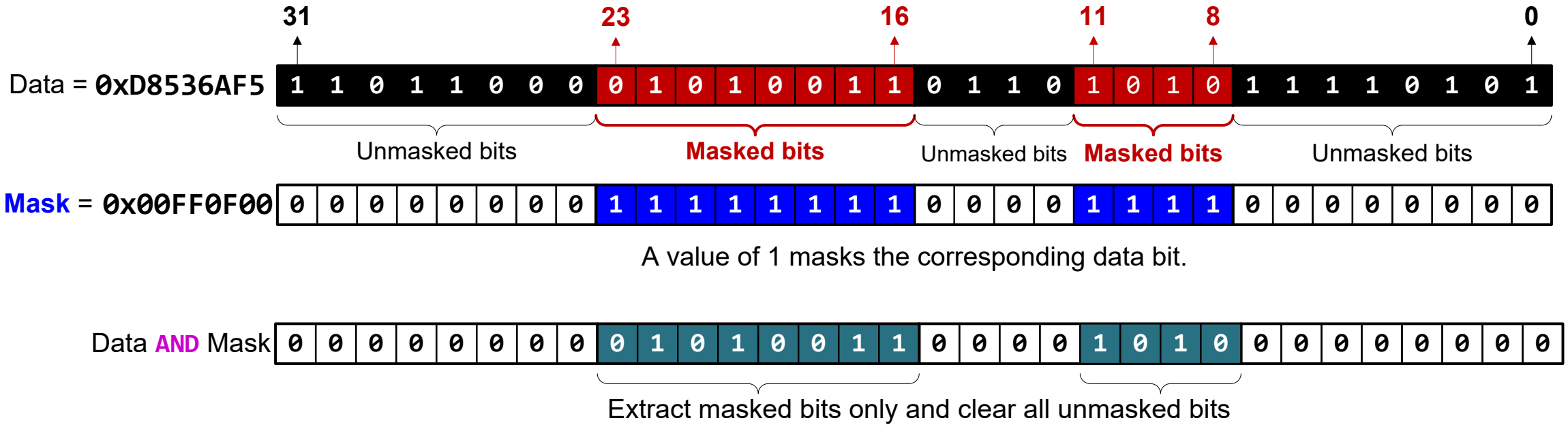
```
Solution:
    MOVS r4, #1                 ; r4 = 1
    EORS r0, r0, r4, LSL #5     ; r0 = r0 ^ 1<<5
```

Mask



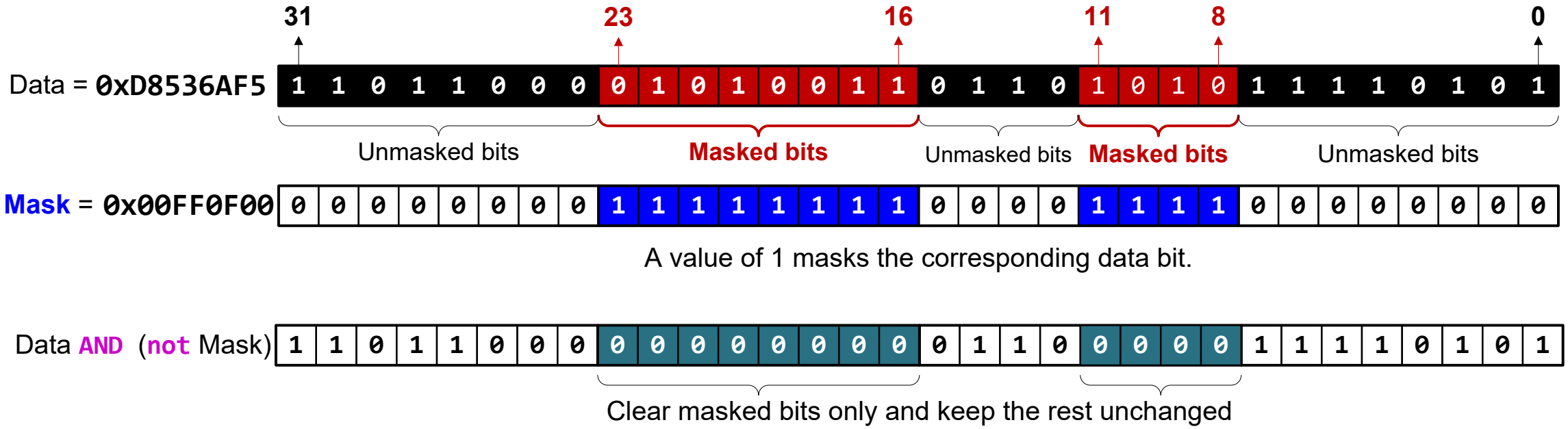
- Bits 8-11 and bits 16-23 are **masked**.
- All the rest bits are **unmasked**

Clear all unmasked bits



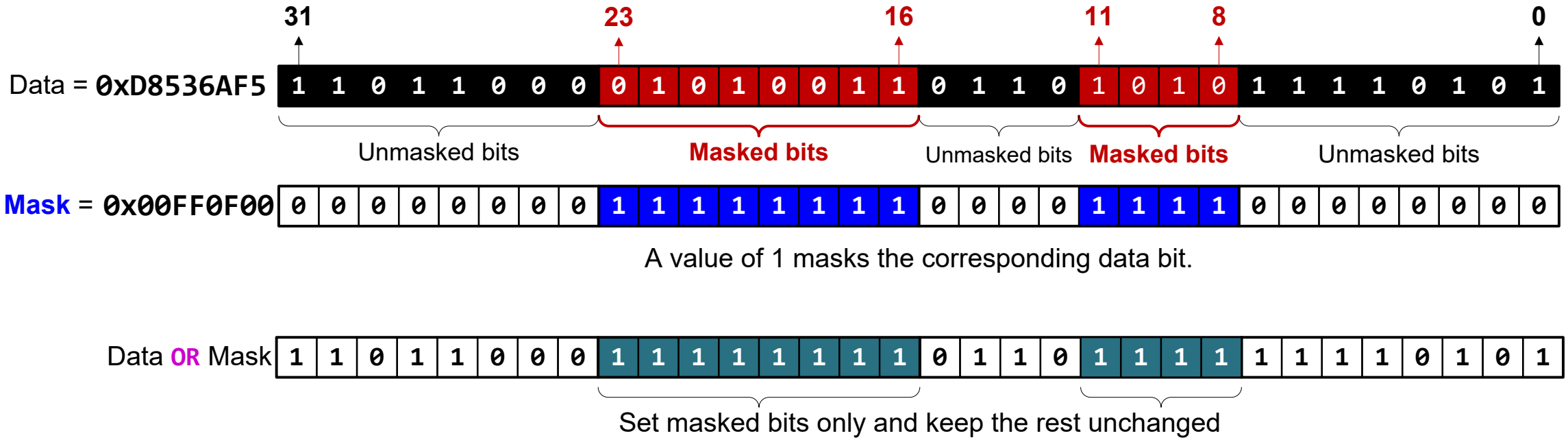

```
Data &= Mask;
```

Clear all masked bits



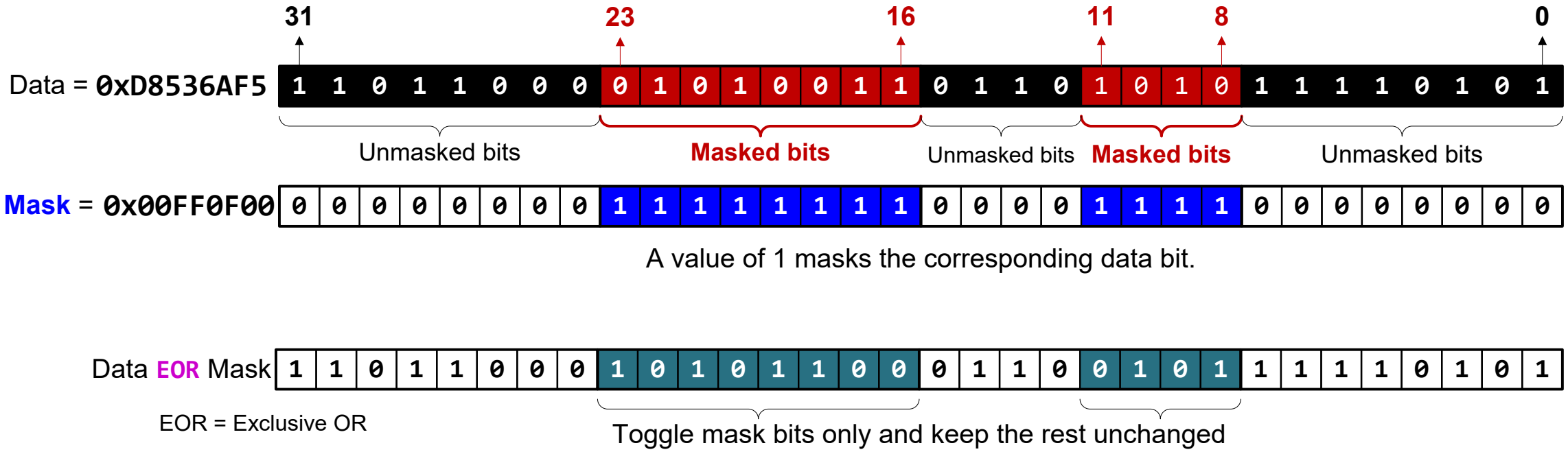

```
Data &= ~Mask;
```

Set all masked bits




```
Data |= Mask;
```

Toggle all tasked bits




```
Data ^= Mask;
```

References

- 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

References

- 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