

Embedded Systems with ARM Cortex-M3 Microcontrollers in Assembly Language and C

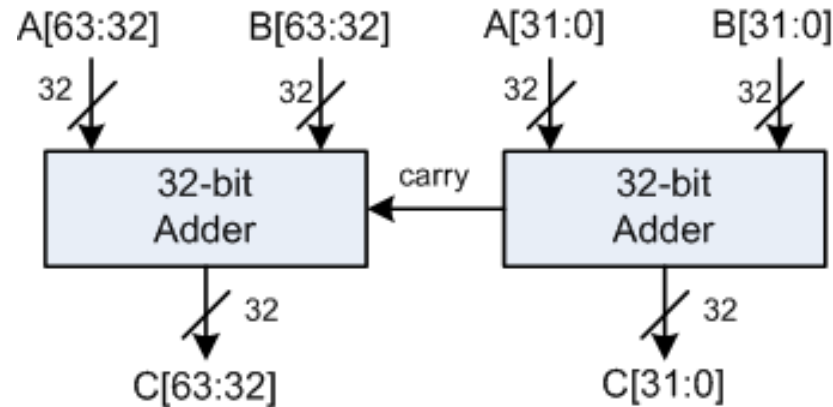
Chapter 9 64-bit Data Processing

Z. Gu

Fall 2025

64-bit Addition

It uses ADDS (add with carry update) and ADC (add with carry) instructions to compute the 64-bit result across two 32-bit parts with carry propagation.

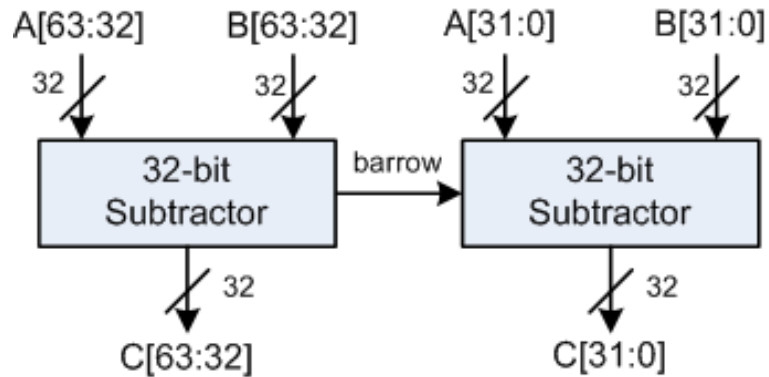


```
; Adding two 64-bit integers A (r1:r0) and B (r3:r2)
; C (r5:r4) = A (r1:r0) + B (r3:r2)
; A = 00002222FFFFFFFF, B = 0000444000000001
LDR  r0, =0xFFFFFFFF    ; A's lower 32 bits
LDR  r1, =0x00002222    ; A's upper 32 bits
LDR  r2, =0x00000001    ; B's lower 32 bits
LDR  r3, =0x00000444    ; B's upper 32 bits

; Add A and B
ADDS r4, r2, r0           ; C[31:0] = A[31:0] + B[31:0], update Carry
ADC  r5, r3, r1           ; C[63:32] = A[63:32] + B[63:32] + Carry
```

64-bit Subtraction

It uses SUBS (subtract with carry update) and SBC (subtract with carry) instructions with borrow (= not Carry) to handle the two-part subtraction.



```
; Subtracting two 64-bit integers A (r1:r0) and B (r3:r2).  
; C (r5:r4) = A (r1:r0) - B (r3:r2)  
; A = 00000002FFFFFFFFF, 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 A from B
```

```
SUBS r4, r0, r2 ; C[31:0] = A[31:0] - B[31:0], update Carry
```

```
SBC r5, r1, r3 ; C[64:32] = A[64:32] - B[64:32] + Carry - 1
```

64-bit Counting Leading Zeroes

```
; 64-bit input data = (r1:r0), r1 = upper word, r0 = lower word  
; r2 = # of leading zero bits in the 64-bit data
```

```
; Counting # of leading zeroes in upper word  
CLZ  r2, r1          ; CLZ = Count leading zeroes
```

```
; Counting # of leading zeroes in lower word  
CMP   r2, #32  
CLZEQ r3, r0          ; if r2 == 32, then count leading zero  
                        ; bits of the lower word
```

```
ADDEQ  r2, r2, r3      ; if all bits of the upper word are zero,  
                        ; add the leading zeroes of the lower word
```

64-bit Counting Leading Zeroes: Explanations

- ▶ Run CLZ on the upper 32 bits: `CLZ r2, r1`; if $r1 \neq 0$, $r2$ already equals the 64-bit leading-zero count since leading zeros must lie entirely in the upper half for a nonzero upper word.
- ▶ Compare $r2$ with 32: `CMP r2, #32`; if equal, the upper word is zero, so count lower: `CLZEQ r3, r0`; then `ADDEQ r2, r2, r3` to add the lower's leading zeros for the total over 64 bits.
- ▶ Edge cases:
 - ▶ If $r1 \neq 0$, the lower word is ignored because the first 1-bit lies in the upper word, making $r2$ the final answer directly from `CLZ r1`.
 - ▶ If $r1 = 0$ and $r0 \neq 0$, the total is $32 + \text{CLZ}(r0)$, as there are 32 leading zeros from the upper word plus those from the lower word until its first 1-bit.
 - ▶ If $r1 = 0$ and $r0 = 0$, the total is 64; the slide's conditional path computes $r2 = 32 + 32 = 64$ via `CLZ` on both halves.

64-bit Sign Extension

```
; r0 = Lower word of 64-bit data  
; r1 = Upper word of 64-bit data
```

```
TST r0, 0x80000000      ; Check the sign bit  
LDRNE r1, =0xFFFFFFFF  ; If MSB is 1, duplicate 1 in upper word  
LDREQ r1, =0x00000000  ; If MSB is 0, duplicate 0 in upper word
```

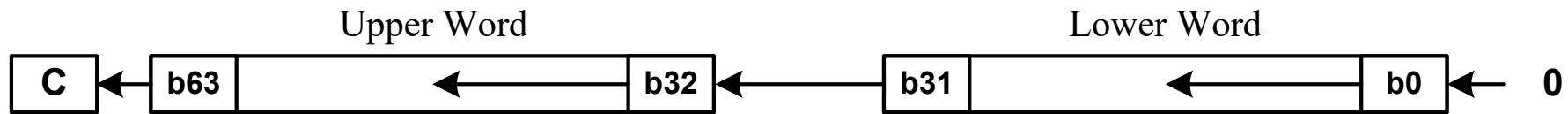
TST (test) performs a bitwise AND between operands to test bits, updates condition flags (notably Z and N), and discards the result. This is effectively “ANDS without write-back,” used to check whether specific bits are set or clear before a conditional operation.

TST r0, 0x80000000 checks the sign bit of the low 32 bits by ANDing r0 with the mask and sets condition flags without storing a result. Z=0 implies MSB=1 (negative), Z=1 implies MSB=0 (non-negative).

LDRNE r1, =0xFFFFFFFF ; if Z=0, sign bit set MSB=1 (result non-zero → NE), set upper 32 = all 1s

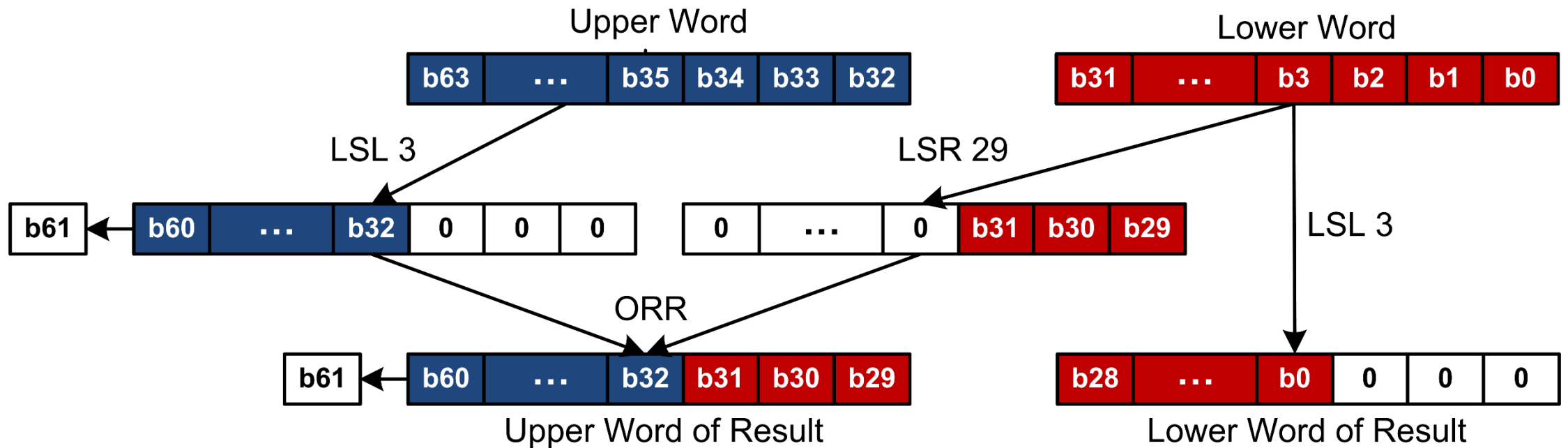
LDREQ r1, =0x00000000 ; if Z=1, sign bit clear MSB=0 (result zero → EQ), set upper 32 = all 0s

64-bit Logic Shift Left

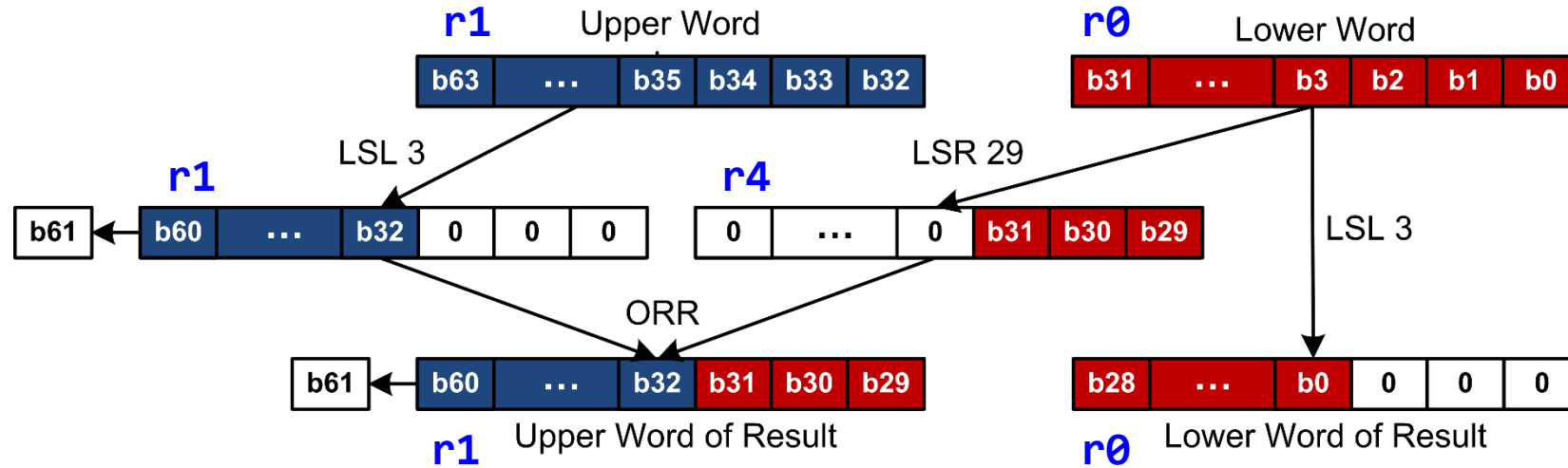


64-bit Logic Shift Left

Example of Logic Shift Left by 3

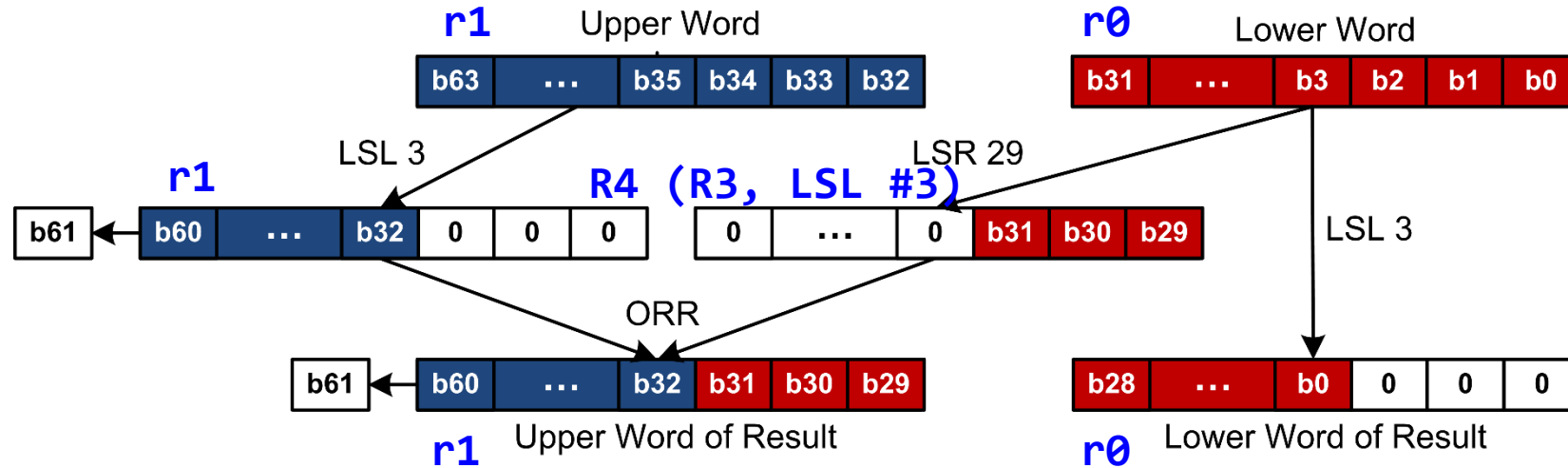


64-bit Logic Shift Left



- Write the assembly code

64-bit Logic Shift Left

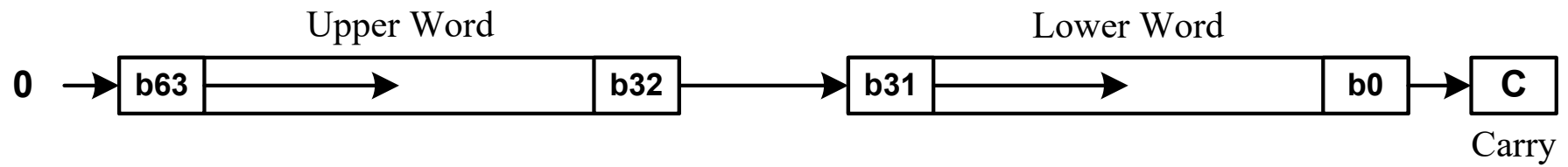


- Write the assembly code
 - ; r0 = Lower word of 64-bit data
 - ; r1 = Upper word of 64-bit data

```
MOV  r3, r0           ; Backup the lower word
MOVS r1, r1, LSL #3    ; Shift left the upper word
MOV  r0, r0, LSL #3    ; Shift left the lower word

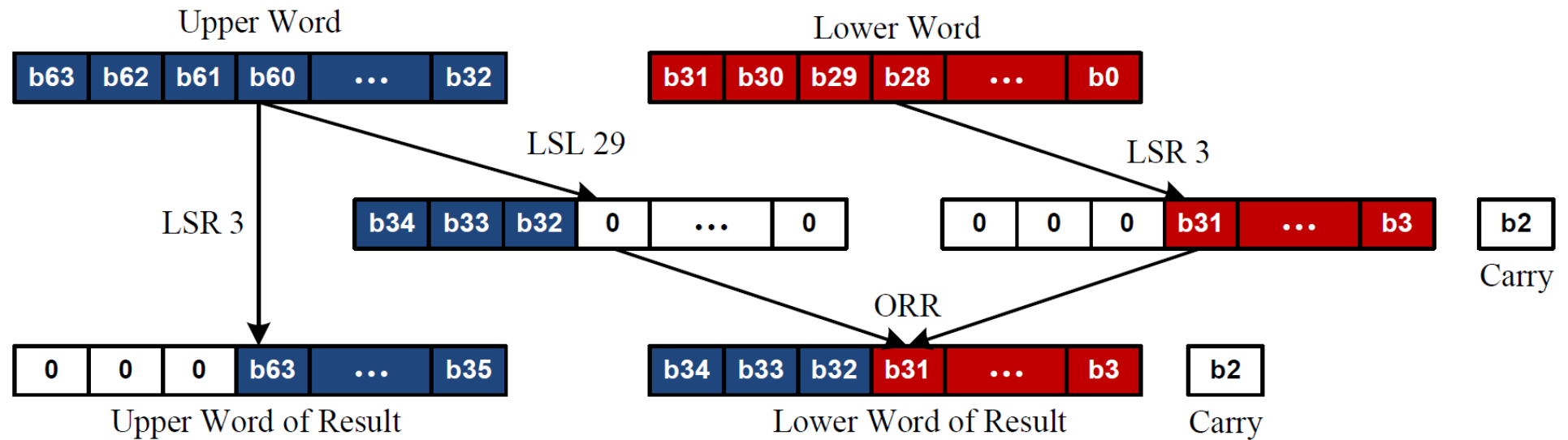
ORR  r1, r1, r3, LSR #29 ; upper |= lower >> (32 - 3)
```

64-bit Logic Shift Right

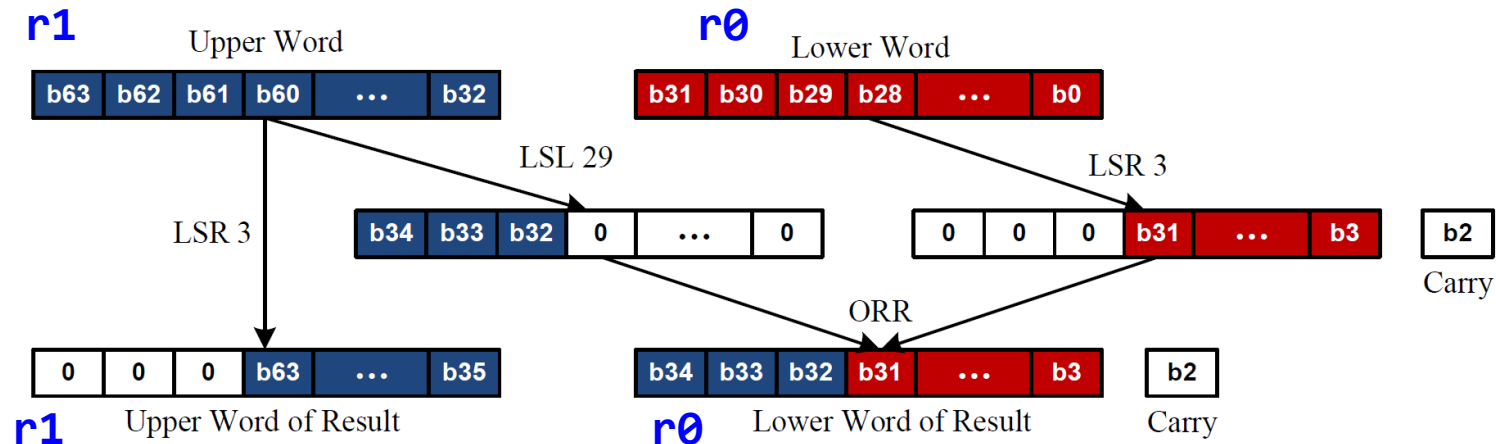


64-bit Logic Shift Right

Example of Logic Shift Right by 3



64-bit Logic Shift Right



► Write the assembly code

; r0 = Lower word of 64-bit data

; r1 = Upper word of 64-bit data

MOV r3, r1 ; Backup the upper word

MOV r1, r1, LSR #3 ; Shift right upper word

MOV r0, r0, LSR #3 ; Shift right lower word

ORR r0, r0, r3, LSL 29 ; lower |= upper << (32 - 3)

64-bit Multiplication

```
; product (r5:r4) = multiplier (r1:r0) × multiplicand (r3:r2)
; (r5:r4) = r0 × r2 + 232 × (r1 × r2 + r0 × r3) + 264 × r1 × r3
; The last item exceeds 64 bits and thus it is ignored.
```

```
UMULL r4, r5, r0, r2    ; (r5:r4) = r0 * r2
MLA    r5, r1, r2, r5    ; r5 = r5 + r1 * r2
MLA    r5, r0, r3, r5    ; r5 = r5 + r0 * r3
```

UMULL r4, r5, r0, r2 computes the 64-bit product of the low words and places the low 32 bits in r4 and the high 32 bits in r5, establishing the initial 64-bit accumulator.

MLA Multiply Accumulate: MLA Rd, Rn, Rm, Ra computes $Rd = (Ra + (Rn \times Rm)) \bmod 2^{32}$. Only the least-significant 32 bits of the sum are written to Rd; any higher bits are discarded.

MLA r5, r1, r2 adds the cross term $r1 \cdot r2$ (which conceptually belongs at bit position 32) into the high half r5

MLA r5, r0, r3, r5 similarly adds the other cross term $r0 \cdot r3$ into r5, completing the contribution of both cross terms at the correct alignment; overflow beyond 32 bits of r5 is discarded, ignoring the $2^{64} \cdot (r1 \cdot r3)$ part.