

ARM Instruction Set Architecture (II)

Lecture 6

Yeongpil Cho

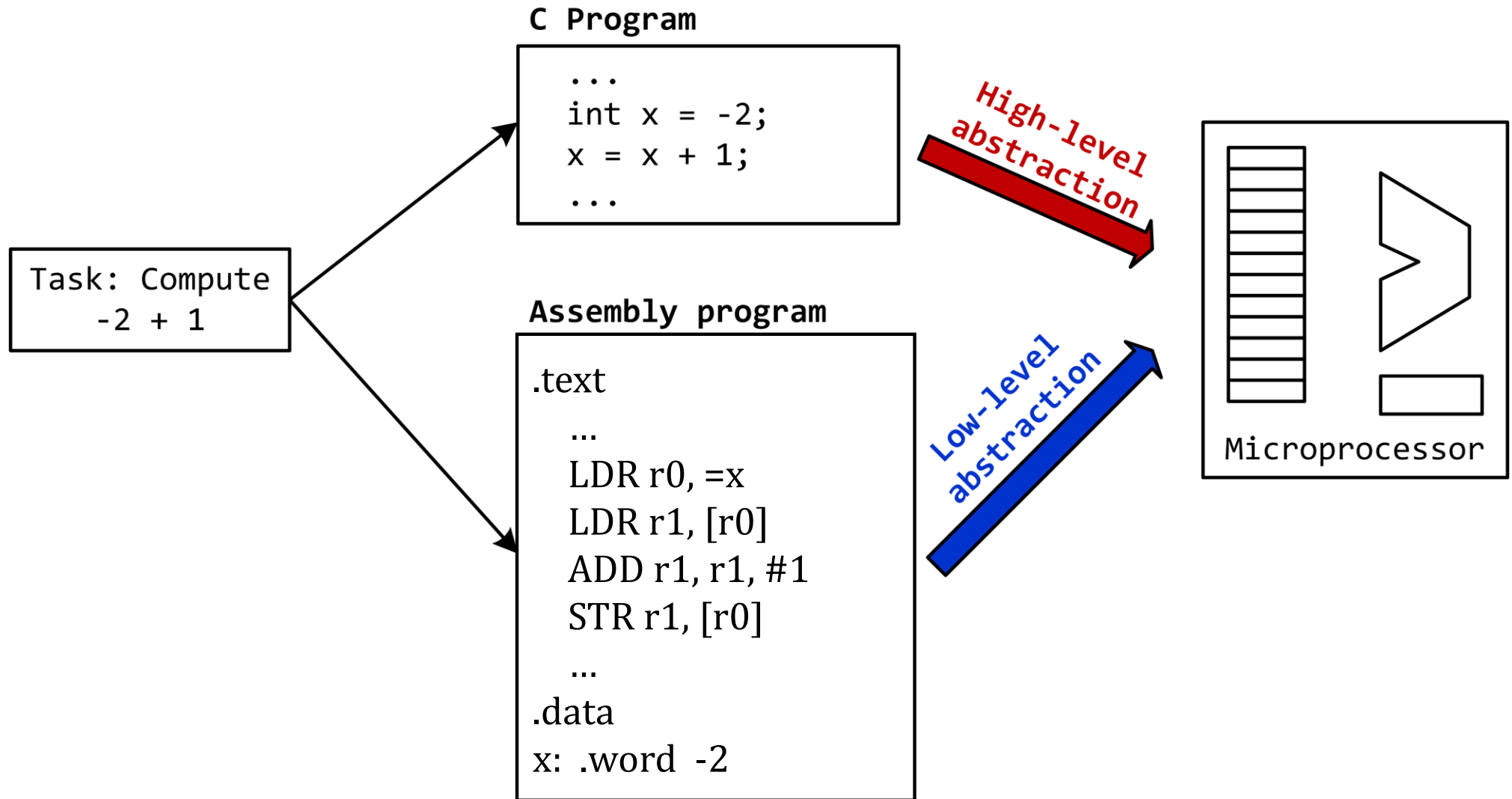
Hanyang University

Topics

- ARM Assembly Instruction *General Syntax*
- ARM Arithmetic and Logic Instructions

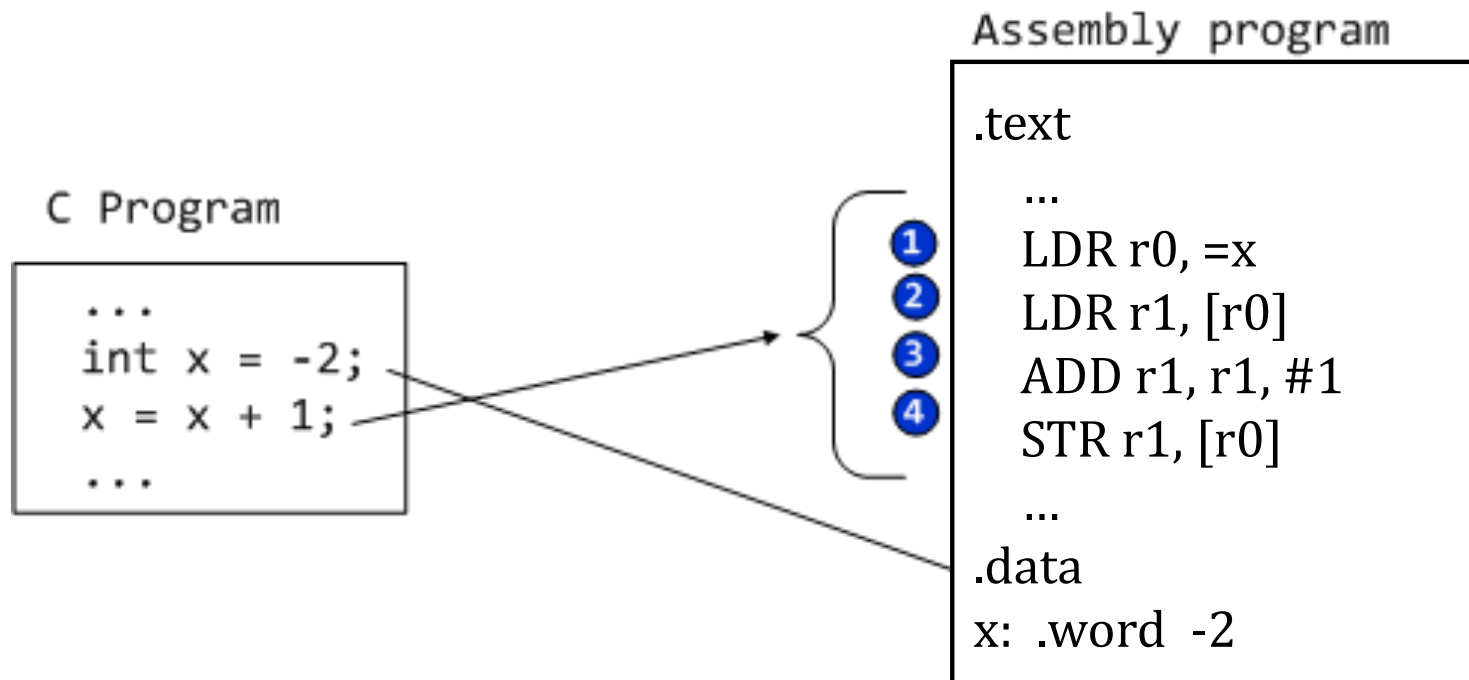
ARM Assembly Instruction

From C to Assembly

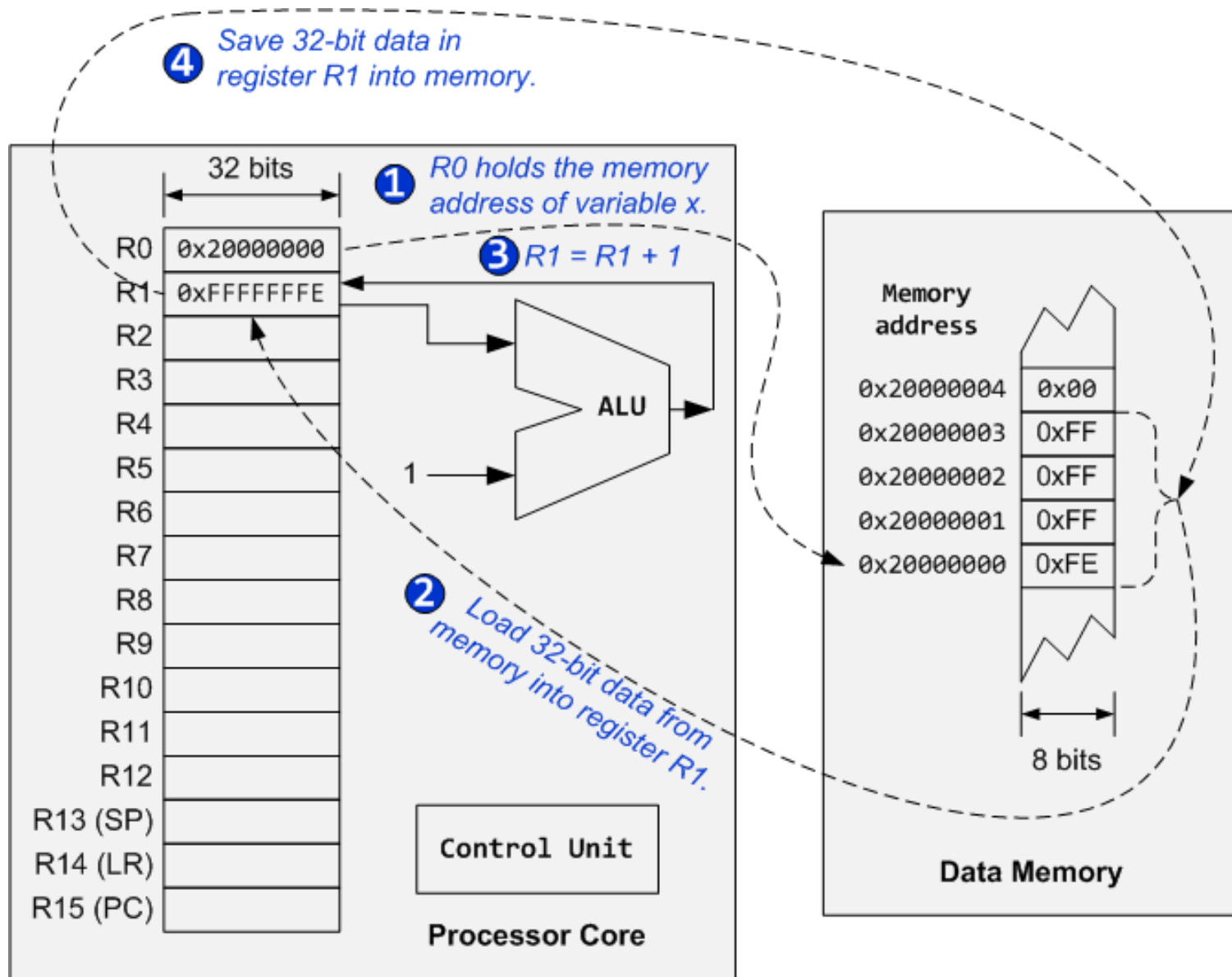


Load-Store Architecture

- Instructions are divided into two categories:
 - memory access operations
 - between memory and registers
 - ALU operations
 - between registers



Load-Store Architecture

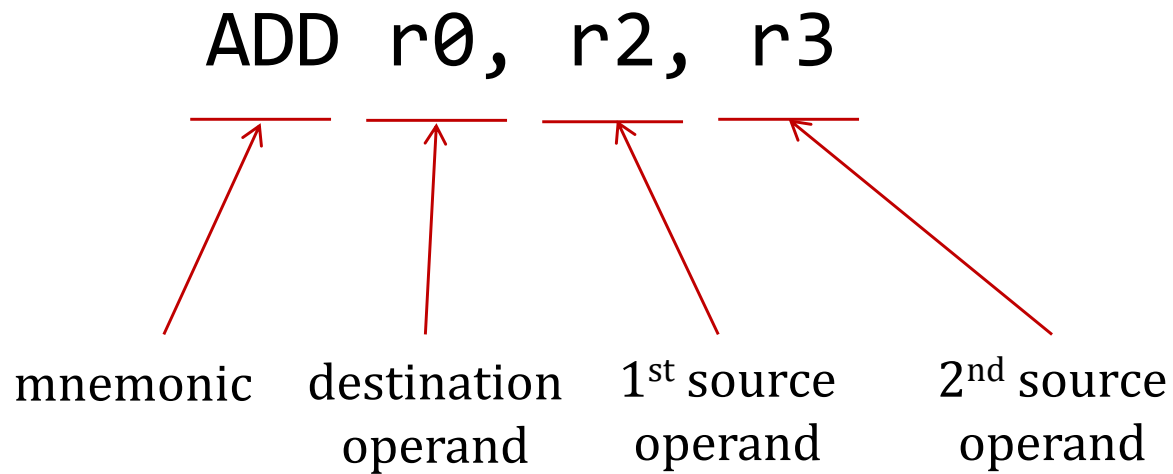


Assembly Instructions Supported

- Arithmetic and logic
 - Add, Subtract, Multiply, Divide, Shift, Rotate
- Data movement
 - Load, Store, Move
- Compare and branch
 - Compare, Test, If-then, Branch, compare and branch on zero
- Miscellaneous
 - Breakpoints, wait for events, interrupt enable/disable, data memory barrier, data synchronization barrier

ARM Instruction Format

mnemonic operand1, operand2, operand3



ARM Instruction Format

mnemonic operand1, operand2, operand3

- Mnemonic represents the operation to be performed.
- The number of operands varies, depending on each specific instruction. Some instructions have no operands at all.
 - Typically, operand1 is the destination register, and operand2 and operand3 are source operands.
 - operand2 is usually a register.
 - operand3 may be a register, an immediate number, a register shifted to a constant amount of bits, or a register plus an offset (used for memory access).

ARM Instruction Format

mnemonic operand1, operand2, operand3

- Examples: Variants of the ADD instruction

ADD r1, r2, r3 ; r1 = r2 + r3

ADD r1, r3 ; r1 = r1 + r3

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

ADD r1, #15 ; r1 = r1 + 15

First Assembly

```
.equ STACK_TOP, 0x20000800      /* Equates symbol to value */
.text                          /* Tells AS to assemble region */
.syntax unified                /* Means language is ARM UAL */
.thumb                        /* Means ARM ISA is Thumb */
.global _start                 /* .global exposes symbol */
                                /* _start label is the beginning */
                                /* ...of the program region */
.type    start, %function      /* Specifies start is a function */
                                /* start label is reset handler */

_start:
    .word    STACK_TOP, start    /* Inserts word 0x20000800 */
                                /* Inserts word (start) */
                                /* start label */

start:
    movs r0, #10
    movs r1, #0
loop:
    adds r1, r0
    subs r0, #1
    bne  loop
deadloop:
    b    deadloop
.end
```

어셈블리 프로그램 시작 부분

trap →

Encoding 16-bit Thumb Instructions

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	minor opcode														Shift, add, subtract, move, & compare
0	1	0	0	0	0	minor opcode				Rm/Rn			Rd/Rn		Data processing	
0	1	0	0	0	1	minor opcode				Rm			Rd/Rn		Special data instructions & branch	
0	1	0	0	1	x	minor opcode										Load from Literal Pool
0	1	0	1	x	x											Load/store single data item
0	1	1	x	x	x											Load/store single data item
1	0	0	x	x	x											Load/store single data item
1	0	1	0	0		Rd		imm8								Generate PC-relative address
1	0	1	0	1		Rd		imm8								Generate SP-relative address
1	0	1	1	minor opcode												Miscellaneous 16-bit instructions
1	1	0	0	0		Rd		register list								Store multiple registers
1	1	0	0	1		Rd		register list								Load multiple registers
1	1	0	1	minor opcode				offset-8								Conditional branch, & supervisor call
1	1	1	0	0	offset-11											Unconditional branch

Encoding: **ORR r1, r0**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	0	0	1	1	0	0	0	0	0	0	0	1	ORR r1, r0
major opcode						minor opcode				Rn			Rd			

0x4301

32-bit Thumb Instructions

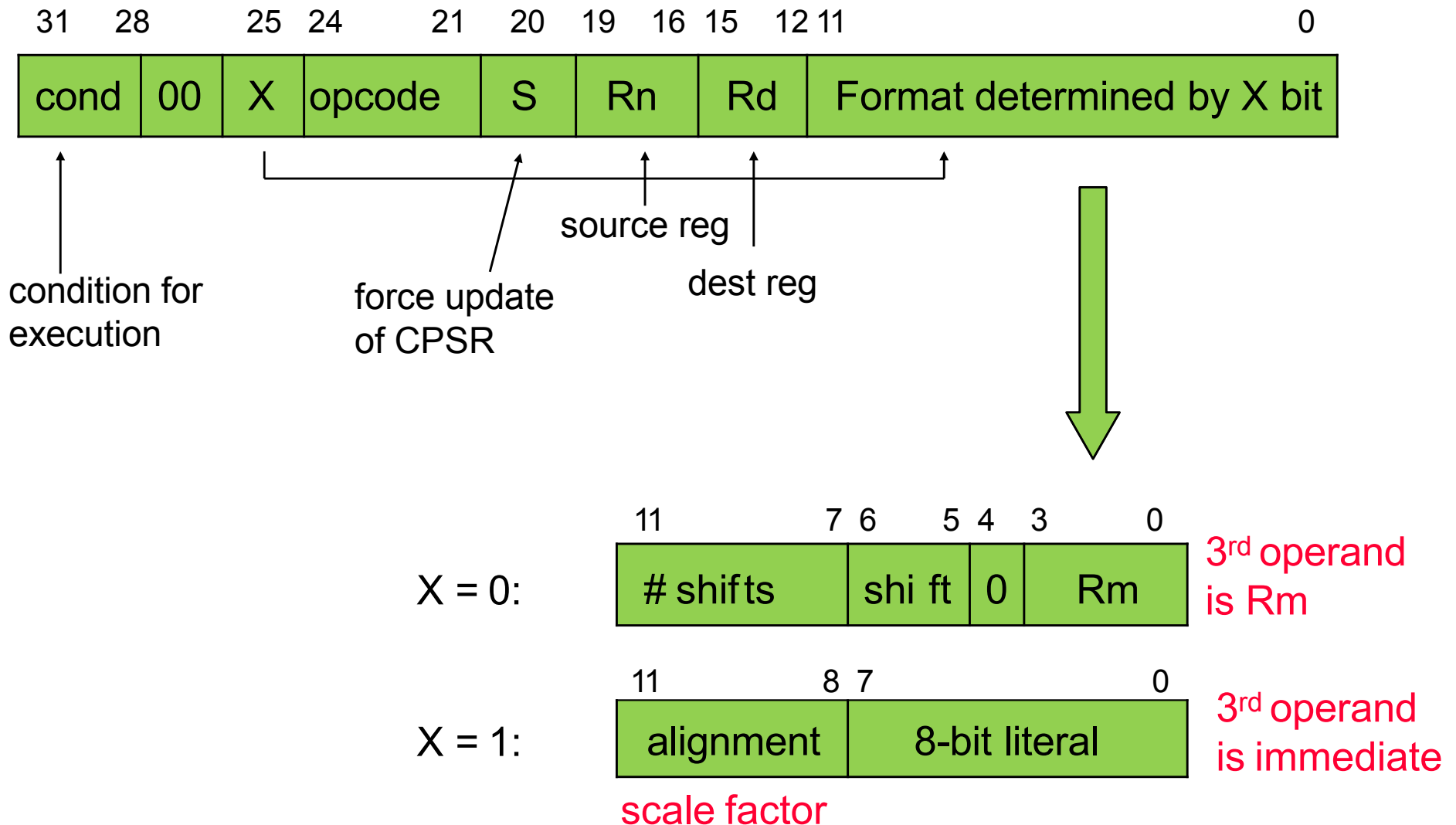
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0							
1	1	1	0	1	0	0	op		0	W	L	Rn				x		0	Register list										Load/store multiple									
1	1	1	0	1	0	0	op1		1	op2		Rn				x							op3						Load/store dual or exclusive, table branch									
1	1	1	0	1	0	1	op				S	Rn				x	imm3			Rd			imm2				Rm			Data processing (shifted register)								
1	1	1	0	1	1	op1										x				coproc					op					Coprocessor instructions								
1	1	1	0	1	x	0	op					Rn				0	imm3			Rd			imm8							Data processing (modified immediate)								
1	1	1	1	0	x	1	op					Rn				0																Data processing (plain binary immediate)						
1	1	1	1	0	op											1	op1																		Branches and miscellaneous control			
1	1	1	1	1	0	0	0	op1			0					x				op2								Store single data item										
1	1	1	1	1	0	0	op1		0	0	1	Rn				Rt			op2													Load byte, memory hints						
1	1	1	1	1	0	0	op1		0	1	1	Rn				Rt			op2												Load halfword, memory hints							
1	1	1	1	1	0	0	op1		1	0	1	Rn				x				op2													Load word					
1	1	1	1	1	0	0	x	x	1	1	1					x																		Undefined				
1	1	1	1	1	0	1	0	op1				Rn				x	1	1	1	1							op2			Rm			Data processing (register)					
1	1	1	1	1	0	1	1	0	op1							x	Ra								0	0	op2		Rm		Multiply, multiply accumulate, and absolute difference							
1	1	1	1	1	0	1	1	1	op1							x													op2			Rm		Long multiply, long multiply accumulate, divide				
1	1	1	1	1	1	op1										x				coproc							op					Coprocessor instructions						

Decoding: 0xF04F, 0x0003

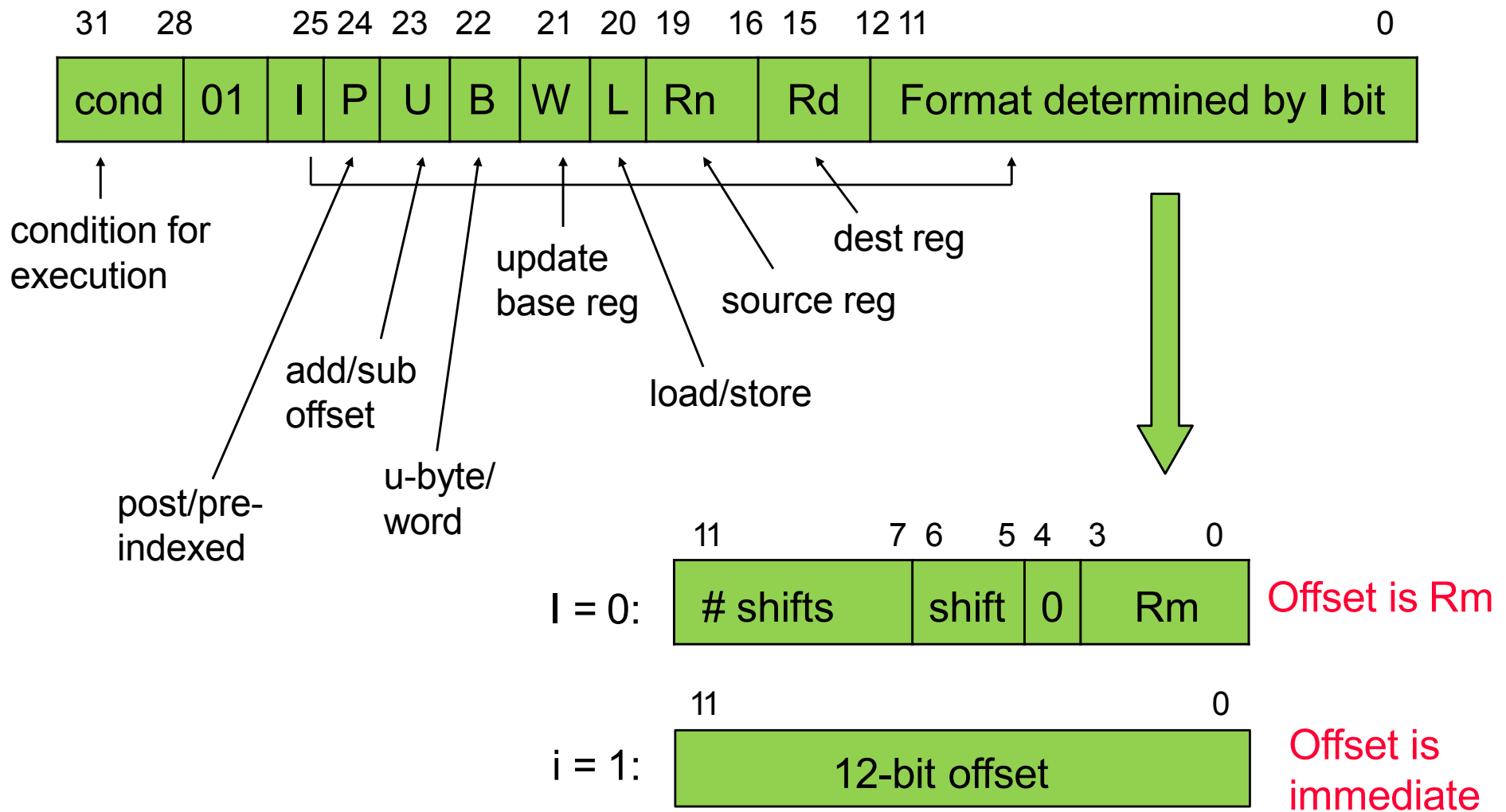
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	1	1	op1		op2						S				
						i										
0xF04F	1	1	1	1	0	0	0	0	0	1	0	0	1	1	1	1
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	op	imm3			Rd				imm8							
0x0003	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	

MOV r0, #3

ARM Instruction Code Format



ARM Load/Store Code Format



ARM Arithmetic and Logic Instructions

Overview:

Arithmetic and Logic Instructions

- Syntax
 - <Operation>{<cond>}{S} Rd, Rn, Operand2
- 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)

Overview:

Arithmetic and Logic Instructions

- Subtraction
 - **SUB**, **RSB** (reverse subtract), **SBC** (subtract with carry)
- Multiplication
 - **MUL** (multiply), **MLA** (multiply-accumulate), **MLS** (multiply-subtract), **SMULL** (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)

Example: **Add**

- Unified Assembler Language (UAL) Syntax

- A common syntax for ARM and Thumb instructions

ADD r1, r2, r3 ; r1 = r2 + r3

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

- Traditional Thumb Syntax

ADD r1, r3 ; r1 = r1 + r3

ADD r1, #15 ; r1 = r1 + 15

Commonly Used Arithmetic Operations

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

S: Set Condition Flags

start:

```
LDR r0, =0xFFFFFFFF
```

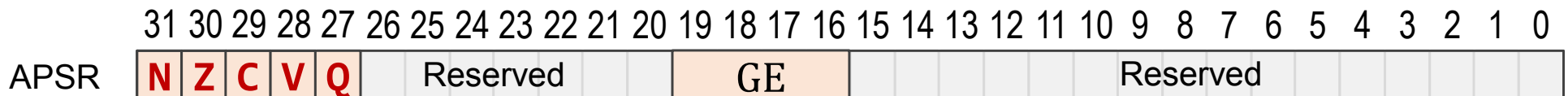
```
LDR r1, =0x00000001
```

```
ADDS r0, r0, r1
```

stop: B stop

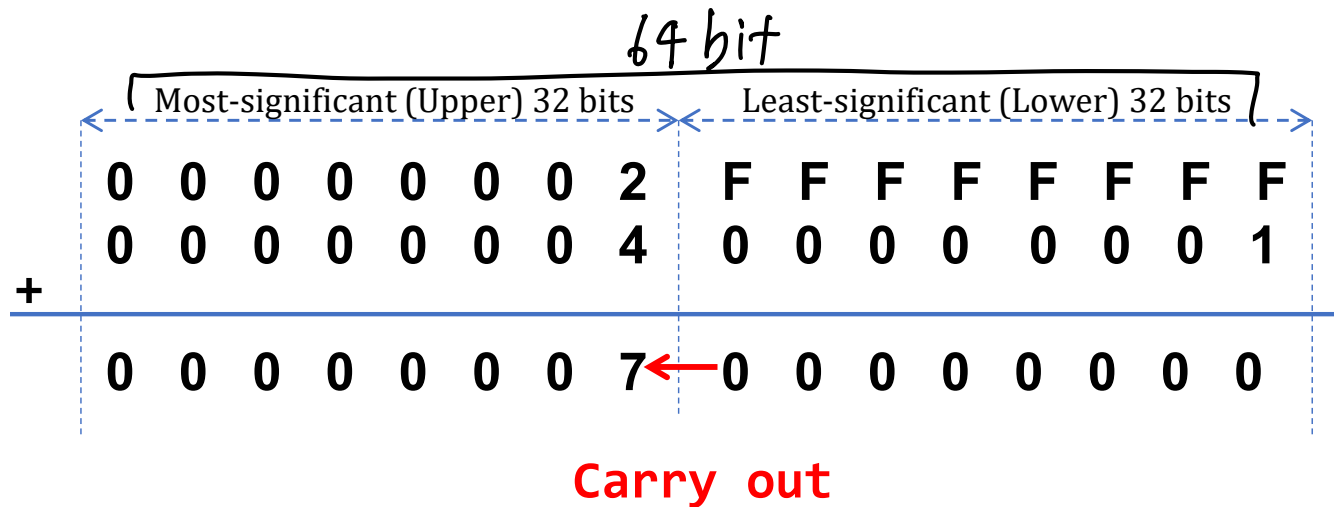
- For most instructions, we can add a suffix **S** to update the N, Z, C, V bit flags of the APSR register.
- In this example, the Z and C bits are set.

The screenshot shows a debugger interface with two main panes. The left pane, titled 'Registers', displays the state of various registers. The 'xPSR' register is highlighted with a red box, showing the following bit flags: N=0, Z=1, C=1, V=0, Q=0, T=1, IT=Disabled, and ISR=0. The right pane, titled 'Disassembly', shows the assembly code for the example. The 'stop' instruction is highlighted in yellow, showing its address (0x08000136) and the instruction code (E7FE). The code includes comments and directives for the main function, such as 'INCLUDE stm32l1xx_constants.s' and 'AREA main, CODE, READONLY'.

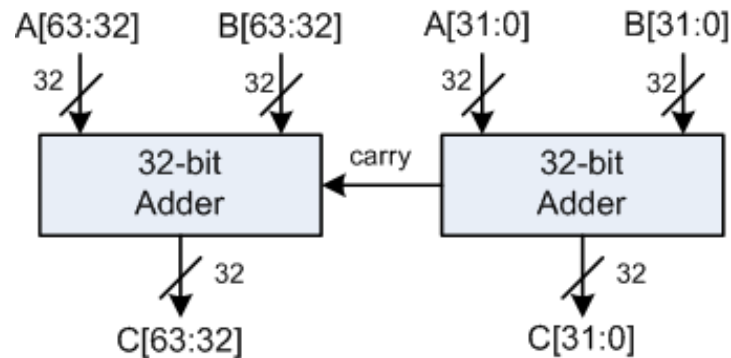


64-bit Addition

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



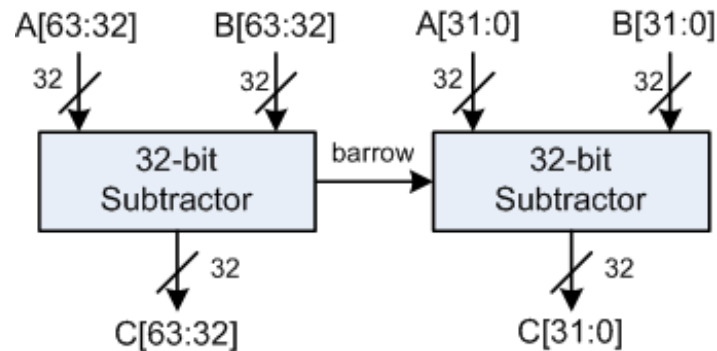
64-bit Addition In Cortex M3 processor



```
; Adding two 64-bit integers A (r1:r0) and B (r3:r2)
; C (r5:r4) = A (r1:r0) + B (r3:r2)
; A = 00002222FFFFFFFF, B = 0000044400000001
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



```
; Subtracting two 64-bit integers A (r1:r0) and B (r3:r2).
```

```
; C (r5:r4) = A (r1:r0) - B (r3:r2)
```

```
; 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 A from B
```

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

```
SBC  r5, r1, r3           ; C[63:32] = A[63:32] - B[63:32] + Carry - 1
```

Short Multiplication

32bit register

; 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

Long Multiplication

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

UMULL **r3, r4**, r0, r1 ; r4:r3 = r0 \times r1, r4 = MSB bits, r3 = LSB bits
SMULL **r3, r4**, r0, r1 ; r4:r3 = r0 \times r1
UMLAL **r3, r4**, r0, r1 ; r4:r3 = r4:r3 + r0 \times r1
SMLAL **r3, r4**, r0, r1 ; r4:r3 = r4:r3 + r0 \times r1

Bitwise Logic

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

Example: AND r2, r0, r1

- Bit-wise Logic **AND**

[illegible]

Example: ORR r2, r0, r1

- Bit-wise Logic **OR**

[illegible]

Example: BIC r2, r0, r1

- Bit Clear
 - $r2 = r0 \& \text{NOT } r1$

Step 1:

[illegible][illegible]

Step 2:

[illegible][illegible]

r2 1 0 0 0 0

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 (12 bits) of R4 to 0

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” = 0xFFFFFFFFE, but “!0x01” = 0x00.

Check a Bit in C

$$\text{bit} = a \ \& \ (1 \ll k)$$

- Example: $k = 5$

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

Set a Bit in C

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

or

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

- Example: $k = 5$

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

Clear a Bit in C

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

- Example: $k = 5$

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

Toggle a Bit in C

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

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

- Example: $k = 5$

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

a_5	1	$a_5 \oplus 1$
0	1	1
1	1	0

Truth table of Exclusive OR with one

Saturating Instruction: SSAT and USAT

- Saturation is commonly used in signal processing—for example, in signal amplification.
- Syntax:
 - `op{cond} Rd, #n, Rm{, shift}`

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

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

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

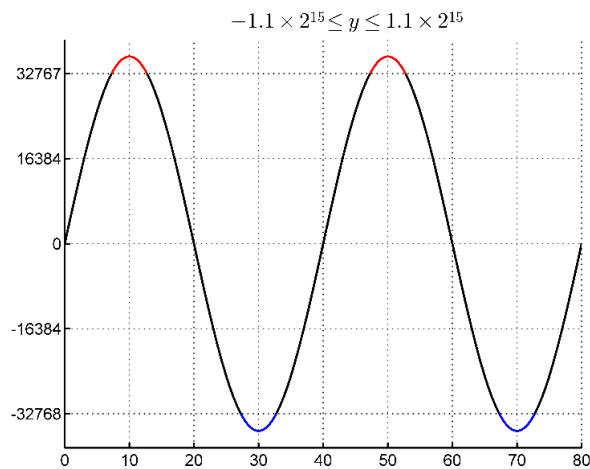
$$USAT(x) = \begin{cases} 2^n - 1 & \text{if } x > 2^n - 1 \\ x & \text{otherwise} \end{cases}$$

- Examples:

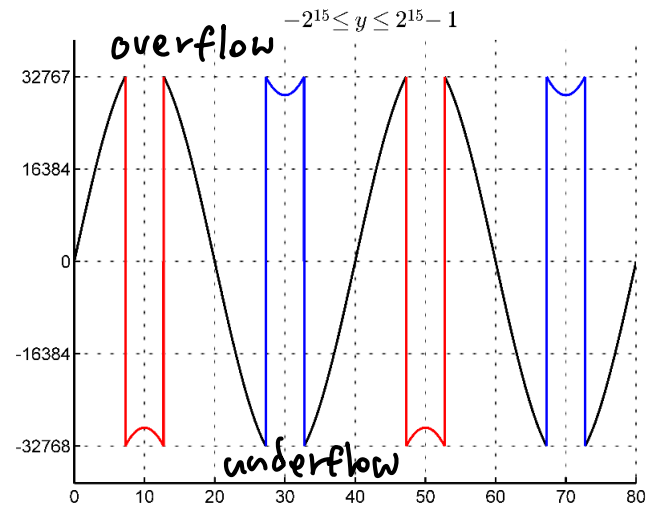
- `SSAT r2, #11, r1` ; output range: $-2^{10} \leq r2 \leq 2^{10}$
- `USAT r2, #11, r3` ; output range: $0 \leq r2 \leq 2^{11}$

Example of Saturation

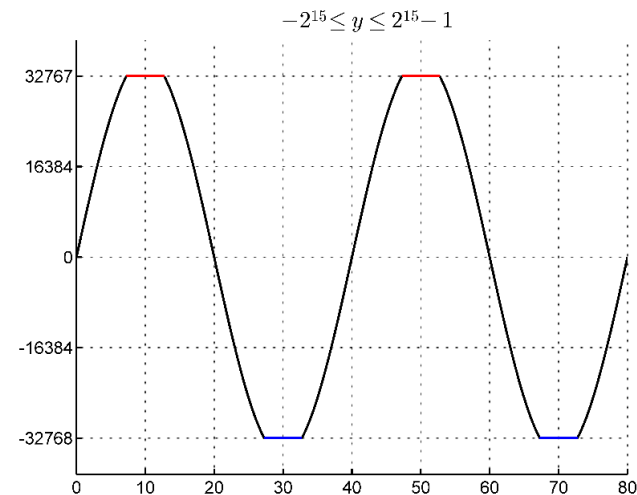
- Assume data are limited to **16** bits



Without
saturation



With
saturation



Reverse Order

- We can use these to change endianness

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

Reverse Order

- **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
----	---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

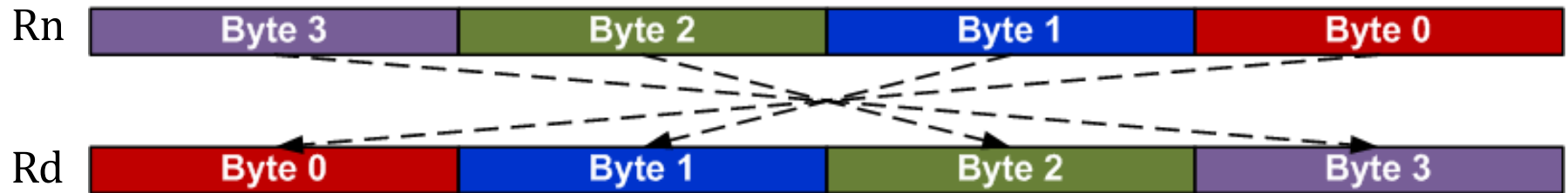
- Example

LDR r0, =0x12345678 ; r0 = 0x12345678

RBIT r1, r0 ; Reverse bits, r1 = 0x1E6A2C48

Reverse Order

- **REV** Rd, Rn



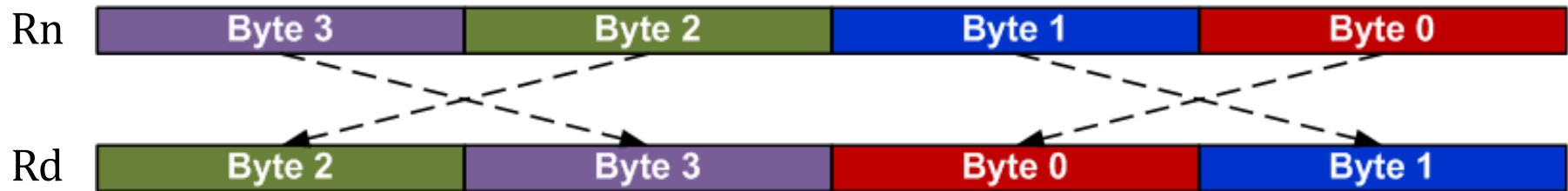
- Example:

LDR R0, =0x12345678 ; R0 = 0x12345678

REV R1, R0 ; R1 = 0x78563412

Reverse Order

- **REV16** Rd, Rn



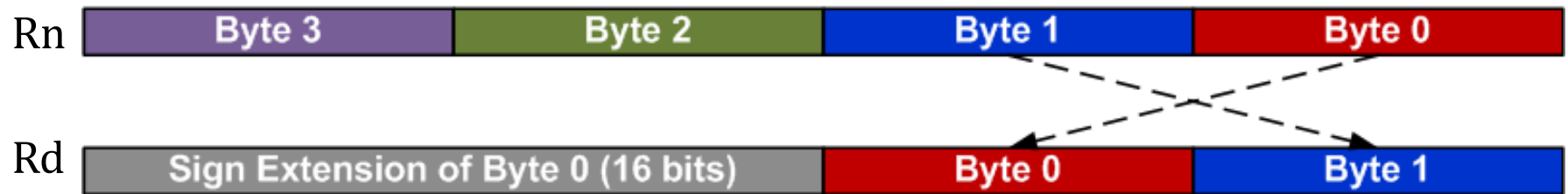
- Example:

LDR R0, =0x12345678 ; R0 = 0x12345678

REV16 R2, R0 ; R2 = 0x34127856

Reverse Order

- **REVSH** Rd, Rn



- Example:

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

Sign and Zero Extension

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

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

Sign and Zero Extension

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

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

Move Data between Registers

MOV Rd, Rn	$Rd \leftarrow \text{operand2}$
MVN Rd, Rn	$Rd \leftarrow \text{NOT operand2}$
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 , $Rd \leftarrow \#imm16$
MOVT Rd, #imm16	Move Top , $Rd \leftarrow \#imm16 \ll 16$
MOV Rd, #const	Move , $Rd \leftarrow \text{const}$

- Example: Load a 32-bit number into a register

```
MOVW r0, #0x4321 ; r0 = 0x00004321
```

```
MOVT r0, #0x8765 ; r0 = 0x87654321
```

half word
destination register

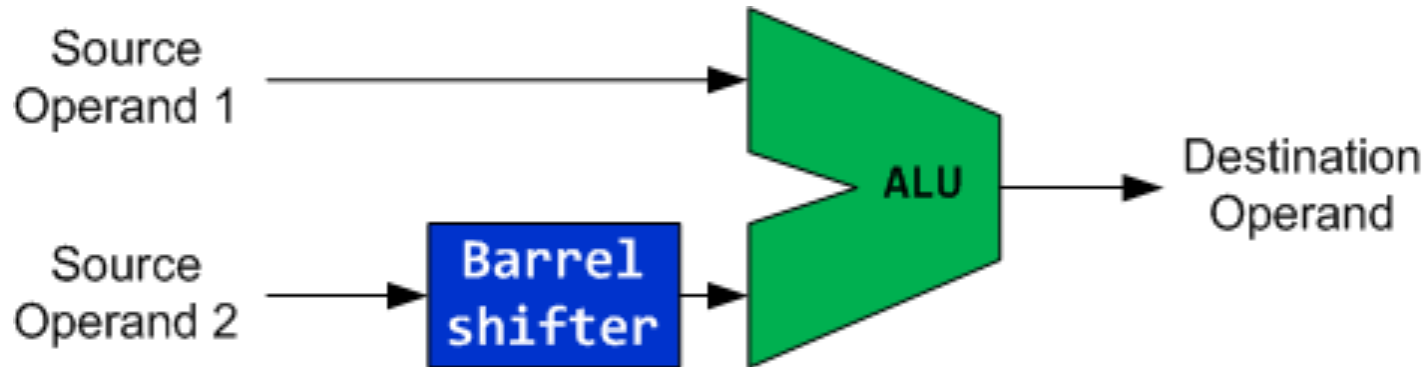
Order does matter!

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

```
MOVT r0, #0x8765 ; r0 = 0x8765xxxx
```

```
MOVW r0, #0x4321 ; r0 = 0x00004321
```

Barrel Shifter



- The second operand of ALU has a special hardware called **Barrel shifter**
- Example:
ADD r1, r0, r0, **LSL** #3 ; $r1 = r0 + r0 \ll 3 = 9 \times r0$
Logical Shift Left

The Barrel Shifter

Logical Shift Left (**LSL**)



Logical Shift Right (**LSR**)



Arithmetic Shift Right (**ASR**)



Rotate Right (**ROR**)



Rotate Right Extended (**RRX**)



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

Barrel Shifter

- Examples:

- **ADD r1, r0, r0, LSL #3**

- ; $r1 = r0 + r0 \ll 3 = r0 + 8 \times r0$

- **ADD r1, r0, r0, LSR #3**

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

- **ADD r1, r0, r0, ASR #3**

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

- Use Barrel shifter to speed up the application

- ADD r1, r0, r0, LSL #3** \Leftrightarrow **MOV r2, #9** ; $r2 = 9$
MUL r1, r0, r2 ; $r1 = r0 * 9$