**Class Instruction Set Architecture**

This document contains a summary of the Instruction Set Architecture (ISA) for the class project, heretofore known as the Class Instruction Set Architecture (CISA). The CISA will use the same registers and instruction decoding scheme as the Arm Education Core, as defined in the Introduction to Arm Education Core document, with the only difference being a reduced number of instructions to implement.

**Table of Contents**

[**How to Use This Document**](#_heading=h.hvpqva90hqmu) **2**

[**Class Instruction Set Summary**](#_heading=h.qq8jurxbgh16) **4**

[NOP](#_heading=h.i2pt0y5p00na) 4

[MOVZ](#_heading=h.3vki7dwq3w60) 5

[LDUR](#_heading=h.84wxuvf8pjzu) 6

[STUR](#_heading=h.vnpybcghv1ro) 6

[B.cond](#_heading=h.e1prknd93re6) 7

[BR](#_heading=h.u2vfs7g1e6re) 8

[B](#_heading=h.2i7ygqboqmvg) 9

[ADDS (Immediate)](#_heading=h.cxhx3dvn8970) 10

[ADDS (Shifted Register)](#_heading=h.65wwicj5l9sf) 11

[SUBS (Immediate)](#_heading=h.gfwio5ml7ycg) 12

[SUBS (Shifted Register)](#_heading=h.dmd2s93t80dj) 13

[BFM](#_heading=h.xxfw56a5nh8k) 14

[CCMN (Register)](#_heading=h.h75qvypl67) 15

[AND (Immediate) - Optional](#_heading=h.9uzit4kfwjbk) 16

[AND (Shifted Register)](#_heading=h.n0mor1dfk0lz) 19

[ORR (Immediate) - Optional](#_heading=h.mwd8otgnfqa6) 20

[ORR (Shifted Register)](#_heading=h.wgcfgi9dvmjx) 21

[EOR (Immediate) - Optional](#_heading=h.dfdgtuslbhmy) 22

[EOR (Shifted Register)](#_heading=h.1bfvrfsmxhh1) 23

[EON (Shifted Register)](#_heading=h.lw84cm3uwlun) 24

# **How to Use This Document**

What follows in this document is a summary of the required instructions your implementation must support with a short description of what the instruction does, and how it is encoded/decoded. As a refresher, we’ll go through how to interpret the instruction encoding breakdown below.

Just like the A64 ISA, the CISA uses a fixed instruction width of 32 bits. Each instruction is encoded in specific fields that contain the information needed to operate the instruction, as shown below.

| Bit 31 | Bit 30 | Bit 29 | **Bit 28** | **Bit 27** | **Bit 26** | **Bit 25** | … | Bit 2 | Bit 1 | Bit 0 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

Bits [28] to [25] identify the instruction type:

* If **[28:25]** = 100x, then it is a data processing (**immediate**) instruction.
* If **[28:25]** = x101, then it is a data processing (**register**) instruction.
* If **[28:25]** = x1x0, then it is a load/stores instruction.
* If **[28:25]** = 101x, then it is a branches/system instruction.

Lower case x indicates “don’t care” condition, meaning it could be either 1 or 0. As an example, here is the encoding breakdown for the ADD (Immediate) instruction:

| sf[31] | op[30] | S[29] | [28:23] | sh[22] | imm12[21:10] | rn[9:5] | rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- |
| x | 0 | 0 | 100010 | x | xxxxxxxxxxx | 31 is SP | 31 is SP |

Looking at the [instruction description for the ADD (Immediate) instruction](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ADD--immediate---Add--immediate--?lang=en) in the ARM documentation, we can easily deduce what each bit in this instruction represents. Below is the format for the instruction:

ADD [<Wd|Xd|WSP|SP>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ADD--immediate---Add--immediate--?lang=en#sa_wd_wsp), [<Wn|Xn|WSP|SP>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ADD--immediate---Add--immediate--?lang=en#sa_wn_wsp), #[<imm>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ADD--immediate---Add--immediate--?lang=en#sa_imm){, [<shift>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ADD--immediate---Add--immediate--?lang=en#sa_shift)}

First, the **sf[31]** bit indicates 32-bit or 64-bit addressing modes, meaning when **sf[31]** is 0, the source and destination registers will be assumed to be the 32-bit register (Wd/WSP) designated, and when **sf[31]** is 1, they are assumed to be the 64-bit register (Xd/SP) designated. The **op[30]** and **S[29]** bits are used to distinguish between the four different add/subtract immediate instructions. As discussed above, bits **[28:23]** are used to identify the instruction type, in this case an immediate data processing instruction. The **sh[22]** bit is a flag indicating if a shift operand was used. The **imm12[21:10]** bits are an 12 bit immediate unsigned integer (0-4095). Finally, **rn[9:5]** and **rd[4:0]** are the source and destination 32/64 bit registers respectively with 31 or ‘11111’ being the Stack Pointer (SP).

For example, if you were to have the following ADD instruction:

ADD X0,X0,#1

Which simply takes the value currently in the X0 register, adds 1, and places it back into X0 with no shift, then the exact opcode would take the form:

| sf[31] | op[30] | S[29] | [28:23] | sh[22] | imm12[21:10] | rn[9:5] | rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 100010 | 0 | 000000000001 | 00000 | 00000 |

All together as one bit string (byte separated): 10010001 00000000 00000100 00000000

And as a hexadecimal string (byte separated): 91 00 04 00

We can compare this hand-assembled opcode to one generated by the assembler for the ARM Education Core. To do so, simply create file named ‘test.s’ in a working directory anywhere, and paste the following content into it:

.global \_start

.text

\_start:

ADD X0,X0,#1

YIELD

Then, using the next two commands from lab one, we can assemble our ‘test.s’ file and turn it into a ‘.mem’ file to inspect the opcode in hexadecimal format:

aarch64-none-elf-gcc -nostdlib -nodefaultlibs -lgcc -gdwarf-4 -Wa,-march=armv8-a -Wl,-Ttext=0x0 -Wl,-N -o test.elf test.s

aarch64-none-elf-objcopy -O verilog test.elf test.mem

Inspect the ‘test.mem’ file using your favorite text editor, and you’ll see the following content:

@00000000

00 04 00 91 3F 20 03 D5

And as we can see, the hand-assembled instruction from earlier matches the first 32-bit word in the file, the only difference being a flip in the order of bytes. The second 32-bit word, if you’re wondering, is the [YIELD](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/YIELD--YIELD-?lang=en) instruction we added at the end of the file. Using this same method, we can verify that any instruction will be able to run on the Class Instruction Set Architecture, which your CPU will implement!

# Class Instruction Set Summary

Below is a summary for each instruction in our Class Instruction Set for the project. Each entry will include a short description of what the instruction does, the encoding breakdown table, instruction syntax, and a link to more information in the ARM Instruction Set Architecture. There are a couple of optional instructions, which you will not be required to implement, because of their particularly complex nature.

## [NOP](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/NOP--No-Operation-?lang=en)

No Operation. No Operation does nothing, other than advance the value of the program counter by 4. This instruction can be used for instruction alignment purposes.

| No Operation | [31:12] | CRm [11:8] | op2 [7:5] | [4:0] |
| --- | --- | --- | --- | --- |
| NOP | 11010101000000110010 | 0000 | 000 | 11111 |

**Example:**

| [31:12] | CRm [11:8] | op2 [7:5] | [4:0] |
| --- | --- | --- | --- |
| 11010101000000110010 | 0000 | 000 | 11111 |

**Binary -** 11010101 00000011 00100000 00011111

**Hexadecimal -** D5 03 20 1F

**Assembler Output -**

@00000000

1F 20 03 D5 3F 20 03 D5

## 

## [**MOVZ**](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/MOVZ--Move-wide-with-zero-?lang=en)

Move wide with zero. Move wide with zero moves an optionally-shifted 16-bit immediate value to a register.

**Syntax:** MOVZ [<Wd|Xd>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/MOVZ--Move-wide-with-zero-?lang=en#sa_wd), #[<imm>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/MOVZ--Move-wide-with-zero-?lang=en#sa_imm)

Where ***imm*** is the 16-bit unsigned immediate, in the range 0 to 65535.

[**MOVN**](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/MOVN--Move-wide-with-NOT-?lang=en)

Move wide with NOT. Move wide with NOT moves the inverse of a 16-bit immediate value to a register.

**Syntax:** MOVN [<Wd|Xd>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/MOVN--Move-wide-with-NOT-?lang=en#sa_wd), #[<imm>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/MOVN--Move-wide-with-NOT-?lang=en#sa_imm)

Where ***imm*** is the 16-bit unsigned immediate, in the range 0 to 65535.

| Move wide immediate (MOV) | sf [31] | opc [30:29] | [28:23] | hw [22:21] | imm16 [20:5] | Rd [4:0] |
| --- | --- | --- | --- | --- | --- | --- |
| MOVN | x | 00 | 100101 | {0 or sf,x} | x | 31 is RZR |
| MOVZ | x | 10 | 100101 | {0 or sf,x} | x | 31 is RZR |

**Examples:**

| Syntax | sf [31] | opc [30:29] | [28:23] | hw [22:21] | imm16 [20:5] | Rd [4:0] |
| --- | --- | --- | --- | --- | --- | --- |
| MOVZ X0, #25 | 1 | 10 | 100101 | 00 | 0000000000011001 | 00000 |
| MOVN X3, #576 | 1 | 00 | 100101 | 00 | 0000001001000000 | 00011 |

**Binary -**

**MOVZ:** 11010010 10000000 00000011 00100000

**MOVN:** 10010010 10000000 01001000 00000011

**Hexadecimal -**

**MOVZ:** D2 80 03 20

**MOVN:** 92 80 48 03

**Assembler Output -**

@00000000

20 03 80 D2 03 48 80 92 3F 20 03 D5

## [LDUR](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/LDUR--Load-Register--unscaled--?lang=en)

Load Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from a register, [zero-extends](https://en.wikipedia.org/wiki/Sign_extension) it, and writes it to the calculated address.

**Syntax:** LDUR [<Wt|Xt>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/LDUR--Load-Register--unscaled--?lang=en#sa_wt), [[<Xn|SP>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/LDUR--Load-Register--unscaled--?lang=en#sa_xn_sp){, #[<simm>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/LDUR--Load-Register--unscaled--?lang=en#sa_simm)}]

Where ***Wt|Xt*** - Is the 32/64-bit name of the general purpose register to be transferred encoded in the “Rt” field.***Xn|SP*** - Is the 64-bit name of the general purpose base register, encoded in the “Rn” field.***simm*** - Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

## [STUR](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/STUR--Store-Register--unscaled--?lang=en)

Store Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register.

**Syntax:** STUR [<Wt|Xt>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/STUR--Store-Register--unscaled--?lang=en#sa_wt), [[<Xn|SP>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/STUR--Store-Register--unscaled--?lang=en#sa_xn_sp){, #[<simm>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/STUR--Store-Register--unscaled--?lang=en#sa_simm)}]

Where ***Wt|Xt*** - Is the 32/64-bit name of the general purpose register to be transferred, encoded in the “Rt” field. ***Xn|SP*** - Is the 64-bit name of the general purpose base register, encoded in the “Rn” field.***simm*** - Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

| Load/Store Register | size[31:30] | [29:27] | V[26] | [25:24] | opc[23:22] | [21] | Imm9[20:12] | Type[11:10] | Rn[9:5] | Rt[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| STUR/STR | 1x | 111 | 0 | 00 | 00 | 0 | xxxxxxxxx | 0x or 11 | 31 is SP | 31=RZR |
| LDUR/LDR | 1x | 111 | 0 | 00 | 01 | 0 | xxxxxxxxx | 0x or 11 | 31 is SP | 31=RZR |

**Examples:**

| Syntax | size[31:30] | [29:27] | V[26] | [25:24] | opc[23:22] | [21] | Imm9[20:12] | Type[11:10] | Rn[9:5] | Rt[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| LDUR X15, [X2, #0] | 11 | 111 | 0 | 00 | 01 | 0 | 000000000 | 00 | 00010 | 01111 |
| STUR X24, [X8, #8] | 11 | 111 | 0 | 00 | 00 | 0 | 000001000 | 00 | 01000 | 11000 |

**Binary -**

**LDUR:** 11111000 01000000 00000000 01001111

**STUR:** 11111000 00000000 10000001 00011000

**Hexadecimal -**

**LDUR:** F8 40 00 4F

**STUR:** F8 00 81 18

**Assembler Output -**

@00000000

4F 00 40 F8 18 81 00 F8 3F 20 03 D5

## [B.cond](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/B-cond--Branch-conditionally-?lang=en)

Branch conditionally to a label at a [PC-relative offset](https://en.wikipedia.org/wiki/Addressing_mode#:~:text=PC%2Drelative,-%2B%2D%2D%2D%2D&text=This%20offset%20is%20usually%20signed,while%20statements%20are%20reasonably%20short).), with a hint that this is not a subroutine call or return.

**Syntax:** B.[<cond>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/B-cond--Branch-conditionally-?lang=en#sa_cond) [<label>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/B-cond--Branch-conditionally-?lang=en#sa_label)

Where ***cond*** - Is one of the [standard conditions](https://developer.arm.com/documentation/dui0068/b/ARM-Instruction-Reference/Conditional-execution?lang=en), encoded in the "cond" field in the standard way (0-15). ***label*** - Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

| Branch Conditional | [31:24] | Imm19[23:5] | [4] | cond[3:0] |
| --- | --- | --- | --- | --- |
| B.cond | 01010100 | xxxxxxxxxxxxxxxxxxx | 0 | xxxx |

**Example:**

**B.EQ \_start** where **EQ** is the ‘equal to’ conditional flag, and **\_start** is a label defined directly above the instruction (hence the offset is zero).

| [31:24] | Imm19[23:5] | [4] | cond[3:0] |
| --- | --- | --- | --- |
| 01010100 | 0000000000000000000 | 0 | 0000 |

**Binary -** 01010100 00000000 00000000 00000000

**Hexadecimal -** 54 00 00 00

**Assembler Output -**

@00000000

00 00 00 54 3F 20 03 D5

## [BR](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/BR--Branch-to-Register-?lang=en)

Branch to Register branches unconditionally to an address in a register, with a hint that this is not a subroutine return.

**Syntax:** BR [<Xn>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/BR--Branch-to-Register-?lang=en#sa_xn)

Where ***Xn*** is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

| U Branch (reg.) | [31:25] | opc[24:21] | op2[20:16] | op3[15:10] | Rn[9:5] | [4:0] |
| --- | --- | --- | --- | --- | --- | --- |
| BR | 1101011 | 0000 | 11111 | 000000 | 31=RZR | 00000 |

**Example:**

**BR X1**

| [31:25] | opc[24:21] | op2[20:16] | op3[15:10] | Rn[9:5] | [4:0] |
| --- | --- | --- | --- | --- | --- |
| 1101011 | 0000 | 11111 | 000000 | 00001 | 00000 |

**Binary –** 11010110 00011111 00000000 00100000

**Hexadecimal -** D6 1F 00 20

**Assembler Output -**

@00000000

20 00 1F D6 3F 20 03 D5

## [B](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/B--Branch-?lang=en)

Branch (immediate) causes an unconditional branch to a label at a PC-relative offset, with a hint that this is not a subroutine call or return.

**Syntax:** B [<label>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/B--Branch-?lang=en#sa_label)

Where ***label*** is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as "imm26" times 4.

| U Branch (imm.) | op[31] | [30:26] | Imm26[25:0] |
| --- | --- | --- | --- |
| B | 0 | 00101 | xxxxxxxxxxxxxxxxxxxxxxxxxx |

**Example:**

**B \_start** where **\_start** is a label defined directly above the instruction (hence the offset is zero).

| op[31] | [30:26] | Imm26[25:0] |
| --- | --- | --- |
| 0 | 00101 | 00000000000000000000000000 |

**Binary –** 00010100 00000000 00000000 00000000

**Hexadecimal –** 14 00 00 00

**Assembler Output -**

@00000000

00 00 00 14 3F 20 03 D5

## [ADDS (Immediate)](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ADDS--immediate---Add--immediate---setting-flags-?lang=en)

Add (immediate), setting flags, adds a register value and an immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

**Syntax:** ADDS [<Wd|Xd>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ADDS--immediate---Add--immediate---setting-flags-?lang=en#sa_wd), [<Wn|Xn|WSP>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ADDS--immediate---Add--immediate---setting-flags-?lang=en#sa_wn_wsp), #[<imm>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ADDS--immediate---Add--immediate---setting-flags-?lang=en#sa_imm)

Where ***Wd|Xd|WSP (Rd)*** is the destination register, ***Wn|Xn (Rn)*** is the source register, and ***imm*** is the immediate unsigned value ranging from 0-4095 used in the ADD operation.

| Arith Imm | sf[31] | op[30] | S[29] | [28:23] | sh[22] | Imm12[21:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ADDS | x | 0 | 1 | 100010 | 0 | xxxxxxxxxxxx | 31 is SP | 31 is RZR |

**Example:**

**ADDS X0, X5, #128**

| sf[31] | op[30] | S[29] | [28:23] | sh[22] | Imm12[21:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 100010 | 0 | 000010000000 | 00101 | 00000 |

**Binary –** 10110001 00000010 00000000 10100000

**Hexadecimal –** B1 02 00 A0

**Assembler Output -**

@00000000

A0 00 02 B1 3F 20 03 D5

## [ADDS (Shifted Register)](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ADDS--shifted-register---Add--shifted-register---setting-flags-?lang=en)

Add (shifted register), setting flags, adds two source register values and writes the result to the destination register. It updates the condition flags based on the result.

**Syntax:** ADDS [<Wd|Xd>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ADDS--shifted-register---Add--shifted-register---setting-flags-?lang=en#sa_wd), [<Wn|Xn>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ADDS--shifted-register---Add--shifted-register---setting-flags-?lang=en#sa_wn), [<Wm|Xm>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ADDS--shifted-register---Add--shifted-register---setting-flags-?lang=en#sa_wm)

Where ***Wd|Xd (Rd)*** is the destination register, ***Wn|Xn (Rn)*** is the source register, ***Wm|Xm (Rm)*** is the second source register.

| Arith Reg | sf[31] | op[30] | S[29] | [28:24] | [23:22] | [21] | Rm[20:16] | [15:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ADDS | x | 0 | 1 | 01011 | 00 | 0 | xxxxx | 000000 | xxxxx | xxxxx |

**Example:**

**ADDS W0, W1, W2**

| sf[31] | op[30] | S[29] | [28:24] | [23:22] | [21] | Rm[20:16] | [15:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 01011 | 00 | 0 | 00010 | 000000 | 00001 | 00000 |

**Binary –** 00101011 00000010 00000000 00100000

**Hexadecimal –** 2B 02 00 20

**Assembler Output -**

@00000000

20 00 02 2B 3F 20 03 D5

## [SUBS (Immediate)](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/SUBS--immediate---Subtract--immediate---setting-flags-?lang=en)

Subtract (immediate), setting flags, subtracts an immediate value from a source register value, and writes the result to the destination register. It updates the condition flags based on the result.

**Syntax:** SUBS [<Wd|Xd>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ADDS--immediate---Add--immediate---setting-flags-?lang=en#sa_wd), [<Wn|Xn|WSP>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ADDS--immediate---Add--immediate---setting-flags-?lang=en#sa_wn_wsp), #[<imm>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ADDS--immediate---Add--immediate---setting-flags-?lang=en#sa_imm)

Where ***Wd|Xd|WSP (Rd)*** is the destination register, ***Wn|Xn (Rn)*** is the source register, and ***imm*** is the immediate unsigned value ranging from 0-4095 used in the SUB operation.

| Arith Imm | sf[31] | op[30] | S[29] | [28:23] | sh[22] | Imm12[21:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SUBS | x | 1 | 1 | 100010 | 0 | xxxxxxxxxxxx | 31 is SP | 31 is RZR |

**Example:**

**SUBS X0, X1, #128**

| sf[31] | op[30] | S[29] | [28:23] | sh[22] | Imm12[21:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 100010 | 0 | 000010000000 | 00001 | 00000 |

**Binary –** 11110001 00000010 00000000 00100000

**Hexadecimal –** F1 02 00 20

**Assembler Output -**

@00000000

20 00 02 F1 3F 20 03 D5

## [SUBS (Shifted Register)](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/SUBS--shifted-register---Subtract--shifted-register---setting-flags-?lang=en)

Subtract (shifted register), setting flags, subtracts an optionally-shifted register value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

**Syntax:** SUBS [<Wd|Xd>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ADDS--shifted-register---Add--shifted-register---setting-flags-?lang=en#sa_wd), [<Wn|Xn>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ADDS--shifted-register---Add--shifted-register---setting-flags-?lang=en#sa_wn), [<Wm|Xm>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ADDS--shifted-register---Add--shifted-register---setting-flags-?lang=en#sa_wm)

Where ***Wd|Xd (Rd)*** is the destination register, ***Wn|Xn (Rn)*** is the source register, ***Wm|Xm (Rm)*** is the second source register.

| Arith Reg | sf[31] | op[30] | S[29] | [28:24] | [23:22] | [21] | Rm[20:16] | [15:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SUBS | x | 1 | 1 | 01011 | 00 | 0 | xxxxx | 000000 | xxxxx | xxxxx |

**Example:**

**SUBS W0, W1, W2**

| sf[31] | op[30] | S[29] | [28:24] | [23:22] | [21] | Rm[20:16] | [15:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 1 | 01011 | 00 | 0 | 00010 | 000000 | 00001 | 00000 |

**Binary –** 01101011 00000010 00000000 00100000

**Hexadecimal –** 6B 02 00 20

**Assembler Output -**

@00000000

20 00 02 6B 3F 20 03 D5

## [BFM](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/BFM--Bitfield-Move-?lang=en)

Bit Field Move generally copies a specific bit field from a source register to a destination register, however it exhibits different behavior based on the ***<imms>*** and ***<immr>*** parameters:

***<imms> ≥ <immr>*:** This copies a bitfield of (***<imms>***-***<immr>***+1) bits starting from bit position ***<immr>*** in the source register to the least significant bits of the destination register.

***<imms> ≤ <immr>*:** This copies a bitfield of (***<imms>***+1) bits from the least significant bits of the source register to bit position (regsize-***<immr>***) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases the other bits of the destination register remain unchanged.

**Syntax:** BFM [<Wd|Xd>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/BFM--Bitfield-Move-?lang=en#sa_wd), [<Wn|Xn>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/BFM--Bitfield-Move-?lang=en#sa_wn), #[<immr>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/BFM--Bitfield-Move-?lang=en#sa_immr), #[<imms>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/BFM--Bitfield-Move-?lang=en#sa_imms)

Where ***Wd|Xd*** - Is the 32/64-bit name of the general-purpose destination register, encoded in the "Rd" field. ***Wn|Xn*** - Is the 32/64-bit name of the general-purpose source register, encoded in the "Rn" field. ***imms*** - The leftmost bit number to be moved from the source register. 0-31 for 32-bit registers, and 0-63 for 64-bit registers. ***immr*** – The right rotate amount. 0-31 for 32-bit registers, and 0-63 for 64-bit registers.

| Bitfield | sf[31] | opc[30:29] | [28:23] | N[22] | immr[21:16] | imms[15:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| BFM | x | 01 | 100110 | sf | xxxxxx | xxxxxx | xxxxx | xxxxx |

**Example:**

**BFM X0, X1, #8, #7** this instruction puts the first byte of the source register (bits 0-7) into the last byte of the destination register (56-63).

| sf[31] | opc[30:29] | [28:23] | N[22] | immr[21:16] | imms[15:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 01 | 100110 | 1 | 001000 | 000111 | 00001 | 00000 |

**Binary –** 10110011 01001000 00011100 00100000

**Hexadecimal –** B3 48 1C 20

**Assembler Output -**

@00000000

20 1C 48 B3 3F 20 03 D5

## [CCMN (Register)](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/CCMN--immediate---Conditional-Compare-Negative--immediate--?lang=en)

Conditional Compare Negative (immediate) sets the value of the condition flags (NZCV) to the result of the comparison of a register value and a negated immediate value if the condition is TRUE, and an immediate value otherwise.

**Syntax:** CCMN [<Wn|Xn>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/CCMN--register---Conditional-Compare-Negative--register--?lang=en#sa_wn), #[<imm>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/CCMN--immediate---Conditional-Compare-Negative--immediate--?lang=en#sa_imm),#[<nzcv>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/CCMN--register---Conditional-Compare-Negative--register--?lang=en#sa_nzcv), [<cond>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/CCMN--register---Conditional-Compare-Negative--register--?lang=en#sa_cond)

Where ***Wn|Xn*** - Is the 32/64-bit name of the general-purpose source register, encoded in the "Rn'' field. ***imm*** - Is the five bit unsigned (positive) immediate value used in the conditional comparison, encoded in the “imm5” field. ***nzcv*** - Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags (if ***cond*** is FALSE), encoded in the "nzcv" field. ***cond*** - Is one of the [standard conditions](https://developer.arm.com/documentation/dui0068/b/ARM-Instruction-Reference/Conditional-execution?lang=en), encoded in the "cond" field in the standard way (0-15).

| register | sf [31] | op [30] | S [29] | [28:21] | imm5[20:16] | cond[15:12] | [11:10] | Rn[9:5] | [4] | nzcv[3:0] |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| CCMN | x | 0 | 1 | 11010010 | xxxxx | xxxx | 10 | 31=RZR | 0 | xxxx |

**Example:**

**CCMN X0, #5, #0, EQ** this instruction performs a compare operation with register **X0** and immediate value **5 (00101)** when the **EQ** condition is true (Z == 1) and sets the NZCV register to immediate value **0 (0000)** otherwise.

| sf [31] | op [30] | S [29] | [28:21] | imm5[20:16] | cond[15:12] | [11:10] | Rn[9:5] | [4] | nzcv[3:0] |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 11010010 | 00101 | 0000 | 10 | 00000 | 0 | 0000 |

**Binary –** 10111010 01000101 00001000 00000000

**Hexadecimal –** BA 45 08 00

**Assembler Output -**

@00000000

00 08 45 BA 3F 20 03 D5

## [AND (Immediate)](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/AND--immediate---Bitwise-AND--immediate--?lang=en) - Optional

Bitwise AND (immediate) performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register.

**Syntax:** AND [<Wd|Xd|WSP>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/AND--immediate---Bitwise-AND--immediate--?lang=en#sa_wd_wsp), [<Wn|Xn>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/AND--immediate---Bitwise-AND--immediate--?lang=en#sa_wn), #[<imm>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/AND--immediate---Bitwise-AND--immediate--?lang=en#sa_imm)

Where ***Wd|Xd|WSP (Rd)*** is the destination register, ***Wn|Xn (Rn)*** is the source register, and ***imm*** is the immediate bitmask/value used for the comparison.

The encoding of **N**, **immr**, and **imms** in logical immediate instructions is very different from other immediate encodings in the instruction set. Because of the nature of how these immediate values will be used (as bitmasks), a wide range of “repeating” patterns of bits seemed to fit the purpose. This instruction class is the most complicated and non-intuitive one in the ISA. So let’s start by looking into the definition from the ARM Reference Manual:

The logical immediate instructions accept a bitmask immediate bimm32 or bimm64. Such an immediate consists EITHER of a single consecutive sequence with at least one non-zero bit, and at least one zero bit, within an element of 2, 4, 8, 16, 32 or 64 bits; the element then being replicated across the register width, or the bitwise inverse of such a value. The immediate values of all-zero and all-ones may not be encoded as a bitmask immediate, so an assembler must either generate an error for a logical instruction with such an immediate, or a programmer-friendly assembler may transform it into some other instruction which achieves the intended result.

That’s quite a lot of information in such a short paragraph. Let’s try to break it down: Logical immediate instructions have 13-bits for encoding the immediate, it consists of three fields **N** (1 bit), **immr** (6 bits) and **imms** (6 bits). This format does not allow encoding 0 or ~0 (all ones) as an immediate. Although this sounds problematic at first, this isn’t actually a restriction: this format is only used for instructions such as bitwise AND and OR where these constants are not really useful (e.g. x0 | 0 can be optimized to x0 while x0 | ~0 can be optimized to ~0).

The bit pattern of the immediate consists of identical sub-patterns (e.g. repeating bit fields) with 2-, 4-, 8-, 16-, 32- or 64-bits length. Both the sub-pattern size and value is stored in the **N** and **imms** fields. The bit pattern needs to be a consecutive sequence of (at least one) zeroes, followed by a consecutive sequence of (at least one) ones (the [regular expression](https://en.wikipedia.org/wiki/Regular_expression) for that pattern would be 0+1+). To generate the bit pattern, the format really just stores the number of consecutive ones in the element and the size of the element.

The specified element value can be [right-rotated](https://www.geeksforgeeks.org/rotate-bits-of-an-integer/) up to element size (e) minus 1 to move the start of the sequence of ones to any other point in the element. The number of rotations is stored in **immr** which has 6 bits, so it allows up to 63 rotations in the case of an element size of 64 bits. An element size of 2 only allows 0 or 1 rotations, in this case only the least significant bit is considered, the upper 5 bits of **immr** are simply ignored.

The element gets replicated until it reaches 32- or 64-bits. 13-bits could store 8192 different values, but since e.g. the rotation is not always used to its full potential with smaller element sizes it actually allows less different values but a more useful set of bit patterns.

Since **immr** is actually quite boring (just stores the number of rotations), let’s look into how **N** and **imms** can store both the element size and the number of consecutive ones at the same time:

| **N** | **imms** | | | | | | element size |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 1 | 1 | 1 | 0 | x | 2 bits |
| 0 | 1 | 1 | 1 | 0 | x | x | 4 bits |
| 0 | 1 | 1 | 0 | x | x | x | 8 bits |
| 0 | 1 | 0 | x | x | x | x | 16 bits |
| 0 | 0 | x | x | x | x | x | 32 bits |
| 1 | x | x | x | x | x | x | 64 bits |

The upper bits specify the element size, while the lower bits marked with **x** are used to store the consecutive sequence of ones. 0 (0) means there is one 1 in the bit pattern, 1 (01) means there are two 1’s, 2 (10) means there are two 1’s and so on. At the same time it is not allowed to set all **x** to 1, since this would allow to create all ones (Remember: The format doesn’t allow 0 or all ones to be encoded).

Let’s see some examples:

* 0|111100 represents element 01 (2 bits element size, one 1)
* 0|110101 represents element 00111111 (8 bits element size, six 1’s)

A text file with the **imms**, **immr**, and **N** of every possible logical immediate value (aarch64-logical-immediates.txt) can be found in the project files, as well as a C header file (LogicalImmediate.h) containing function definitions for encoding and decoding these immediate values. You may use these two resources to help your understanding of this immediate encoding so that you may implement it in your single cycle computer.

| Logical Im. | sf[31] | opc[30:29] | [28:23] | N[22] | immr[21:16] | imms[15:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| AND | x | 00 | 100100 | sf or x | xxxxxx | xxxxxx | xxxxx | xxxxx |

**Example:**

Say we want to have a 32-bit mask with a 4-bit repeating pattern of: 1001

To encode this in **N**, **immr**, and **imms** we would need to first set the first 4 bits of **imms** to ‘1110’ in order to specify a length of 4 bits. Then, we would set the remaining 2 least-significant bits of **imms** to ‘01’ to specify that we want two 1’s in the pattern. Without any rotations, this would give us the following 4 bits element: ‘0011’.

Next, we would set **immr** to ‘000001’ to specify a right rotation of 1, producing the element: `1001`!

Since this element is less than 64-bits, we then set **N** to 0 to produce the following encoding for the pattern we want:

| N[22] | immr[21:16] | imms[15:10] |
| --- | --- | --- |
| 0 | 000001 | 111001 |

So the bitmask that will be used will be: 10011001 10011001 10011001 10011001

When converted to decimal: 2576980377

And hexadecimal: 0x99999999

**AND W2, W1, #0x99999999**

| sf[31] | opc[30:29] | [28:23] | N[22] | immr[21:16] | imms[15:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 00 | 100100 | 0 | 000001 | 111001 | 00001 | 00010 |

**Binary –** 00010010 00000001 11100100 00100010

**Hexadecimal –** 12 01 E4 22

**Assembler Output -**

@00000000

22 E4 01 12 3F 20 03 D5

## [AND (Shifted Register)](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/AND--shifted-register---Bitwise-AND--shifted-register--?lang=en)

Bitwise AND (shifted register) performs a bitwise AND between two source register values, and writes the result to the destination register.

**Syntax:** AND [<Wd|Xd>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/AND--shifted-register---Bitwise-AND--shifted-register--?lang=en#sa_wd), [<Wn|Xn>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/AND--shifted-register---Bitwise-AND--shifted-register--?lang=en#sa_wn), [<Wm|Xm>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/AND--shifted-register---Bitwise-AND--shifted-register--?lang=en#sa_wm)

Where ***Wd|Xd (Rd)*** is the destination register, ***Wn|Xn (Rn)*** is the source register, ***Wm|Xm (Rm)*** is the second source register.

You may notice that we are leaving the ***shift*** and ***amount*** variables out of our implementation of this instruction, as compared to the AND (Shifted Register) on the Armv8 Instruction Set. This is because the instruction uses an aliasing system to perform a shift instruction (of type ***shift***), which we will not be implementing.

| Logical Re. | sf[31] | opc[30:29] | [28:24] | [23:22] | N[21] | Rm[20:16] | [15:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| AND | x | 00 | 01010 | 00 | 0 | RZR=31 | 000000 | xxxxx | xxxxx |

**Example:**

**AND X0, X1, X2**

| sf[31] | opc[30:29] | [28:24] | [23:22] | N[21] | Rm[20:16] | [15:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 00 | 01010 | 00 | 0 | 00010 | 000000 | 00001 | 00000 |

**Binary –** 10001010 00000010 00000000 00100000

**Hexadecimal –** 8A 02 00 20

**Assembler Output -**

@00000000

20 00 02 8A 3F 20 03 D5

## [ORR (Immediate)](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ORR--immediate---Bitwise-OR--immediate--?lang=en) - Optional

Bitwise OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate register value, and writes the result to the destination register.

**Syntax:** ORR [<Wd|Xd|WSP>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ORR--immediate---Bitwise-OR--immediate--?lang=en#sa_wd_wsp), [<Wn|Xn>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ORR--immediate---Bitwise-OR--immediate--?lang=en#sa_wn), #[<imm>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ORR--immediate---Bitwise-OR--immediate--?lang=en#sa_imm)

Where ***Wd|Xd|WSP (Rd)*** is the destination register, ***Wn|Xn (Rn)*** is the source register, and ***imm*** is the immediate register value to be compared.

| Logical Im. | sf[31] | opc[30:29] | [28:23] | N[22] | immr[21:16] | imms[15:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ORR | x | 01 | 100100 | sf or 0 | xxxxxx | xxxxxx | xxxxx | xxxxx |

**Example:**

For information on how the immediate values are encoded, and which immediate values are valid, please refer to the explanation in [AND (immediate)](#_heading=h.9uzit4kfwjbk).

**ORR W0, W1, #0x99999999**

| sf[31] | opc[30:29] | [28:23] | N[22] | immr[21:16] | imms[15:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 01 | 100100 | 0 | 000001 | 111001 | 00001 | 00000 |

**Binary –** 00110010 00000001 11100100 00100000

**Hexadecimal –** 32 01 E4 20

**Assembler Output -**

@00000000

20 E4 01 32 3F 20 03 D5

## [ORR (Shifted Register)](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ORR--shifted-register---Bitwise-OR--shifted-register--?lang=en)

Bitwise OR (shifted register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

**Syntax:** ORR [<Wd|Xd>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ORR--shifted-register---Bitwise-OR--shifted-register--?lang=en#sa_wd), [<Wn|Xn>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ORR--shifted-register---Bitwise-OR--shifted-register--?lang=en#sa_wn), [<Wm|Xm>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ORR--shifted-register---Bitwise-OR--shifted-register--?lang=en#sa_wm)

Where ***Wd|Xd (Rd)*** is the destination register, ***Wn|Xn (Rn)*** is the source register, ***Wm|Xm (Rm)*** is the second source register.

As with AND (Shifted Register), we are leaving the ***shift*** and ***amount*** parameters out of the class instruction set for simplicity's sake.

| Logical Re. | sf[31] | opc[30:29] | [28:24] | [23:22] | N[21] | Rm[20:16] | [15:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ORR | x | 01 | 01010 | 00 | 0 | RZR=31 | 000000 | xxxxx | xxxxx |

**Example:**

**ORR X0, X1, X2**

| sf[31] | opc[30:29] | [28:24] | [23:22] | N[21] | Rm[20:16] | [15:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 01 | 01010 | 00 | 0 | 00010 | 000000 | 00001 | 00000 |

**Binary –** 10101010 00000010 00000000 00100000

**Hexadecimal –** AA 02 00 20

**Assembler Output -**

@00000000

20 00 02 AA 3F 20 03 D5

## [EOR (Immediate)](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/EOR--immediate---Bitwise-Exclusive-OR--immediate--?lang=en) - Optional

Bitwise Exclusive OR (immediate) performs a bitwise Exclusive OR of a register value and an immediate value, and writes the result to the destination register.

**Syntax:** EOR [<Wd|Xd|WSP>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/EOR--immediate---Bitwise-Exclusive-OR--immediate--?lang=en#sa_wd_wsp), [<Wn|Xn>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/EOR--immediate---Bitwise-Exclusive-OR--immediate--?lang=en#sa_wn), #[<imm>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/EOR--immediate---Bitwise-Exclusive-OR--immediate--?lang=en#sa_imm)

Where ***Wd|Xd|WSP (Rd)*** is the destination register, ***Wn|Xn (Rn)*** is the source register, and ***imm*** is the immediate value to be compared with the source register value.

| Logical Im. | sf[31] | opc[30:29] | [28:23] | N[22] | immr[21:16] | imms[15:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| EOR | x | 10 | 100100 | sf or 0 | xxxxxx | xxxxxx | xxxxx | xxxxx |

**Example:**

For information on how the immediate values are encoded, and which immediate values are valid, please refer to the explanation in [AND (immediate)](#_heading=h.9uzit4kfwjbk).

**EOR W0, W1, #0x99999999**

| sf[31] | opc[30:29] | [28:23] | N[22] | immr[21:16] | imms[15:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 10 | 100100 | 0 | 000001 | 111001 | 00001 | 00000 |

**Binary –** 01010010 00000001 11100100 00100000

**Hexadecimal –** 52 01 E4 20

**Assembler Output -**

@00000000

20 E4 01 52 3F 20 03 D5

## [EOR (Shifted Register)](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/EOR--shifted-register---Bitwise-Exclusive-OR--shifted-register--?lang=en)

Bitwise Exclusive OR (shifted register) performs a bitwise Exclusive OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

**Syntax:** EOR [<Wd|Xd>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ORR--shifted-register---Bitwise-OR--shifted-register--?lang=en#sa_wd), [<Wn|Xn>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ORR--shifted-register---Bitwise-OR--shifted-register--?lang=en#sa_wn), [<Wm|Xm>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ORR--shifted-register---Bitwise-OR--shifted-register--?lang=en#sa_wm)

Where ***Wd|Xd (Rd)*** is the destination register, ***Wn|Xn (Rn)*** is the source register, ***Wm|Xm (Rm)*** is the second source register.

As with AND (Shifted Register) and ORR (Shifted Register), we are leaving the ***shift*** and ***amount*** parameters out of the class instruction set for simplicity's sake.

| Logical Re. | sf[31] | opc[30:29] | [28:24] | [23:22] | N[21] | Rm[20:16] | [15:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| EOR | x | 10 | 01010 | 00 | 0 | RZR=31 | 000000 | xxxxx | xxxxx |

**Example:**

**EOR X0, X1, X2**

| sf[31] | opc[30:29] | [28:24] | [23:22] | N[21] | Rm[20:16] | [15:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 10 | 01010 | 00 | 0 | 00010 | 000000 | 00001 | 00000 |

**Binary –** 11001010000000100000000000100000

**Hexadecimal –** CA 02 00 20

**Assembler Output -**

@00000000

20 00 02 CA 3F 20 03 D5

## [EON (Shifted Register)](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/EON--shifted-register---Bitwise-Exclusive-OR-NOT--shifted-register--?lang=en)

Bitwise Exclusive OR NOT (shifted register) performs a bitwise Exclusive OR NOT of a register value and an optionally-shifted register value, and writes the result to the destination register.

**Syntax:** EON [<Wd|Xd>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ORR--shifted-register---Bitwise-OR--shifted-register--?lang=en#sa_wd), [<Wn|Xn>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ORR--shifted-register---Bitwise-OR--shifted-register--?lang=en#sa_wn), [<Wm|Xm>](https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions/ORR--shifted-register---Bitwise-OR--shifted-register--?lang=en#sa_wm)

Where ***Wd|Xd (Rd)*** is the destination register, ***Wn|Xn (Rn)*** is the source register, ***Wm|Xm (Rm)*** is the second source register.

As with AND (Shifted Register), ORR (Shifted Register), and EOR (Shifted Register) we are leaving the ***shift*** and ***amount*** parameters out of the class instruction set for simplicity's sake.

| Logical Re. | sf[31] | opc[30:29] | [28:24] | [23:22] | N[21] | Rm[20:16] | [15:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| EON | x | 10 | 01010 | 00 | 1 | RZR=31 | 000000 | xxxxx | xxxxx |

**Example:**

**EON X0, X1, X2**

| sf[31] | opc[30:29] | [28:24] | [23:22] | N[21] | Rm[20:16] | [15:10] | Rn[9:5] | Rd[4:0] |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 10 | 01010 | 00 | 1 | 00010 | 000000 | 00001 | 00000 |

**Binary –** 11001010001000100000000000100000

**Hexadecimal –** CA 22 00 20

**Assembler Output -**

@00000000

20 00 22 CA 3F 20 03 D5