***Computer Architecture Course***

**LAB 2**

**Armv8-A Instruction Encoding**

**Issue 1.0**

**Contents**

[**1**](#_heading=h.1fob9te) **Introduction 1**

[1.1](#_heading=h.3znysh7) Lab overview 1

[**2**](#_heading=h.2et92p0) **Requirements 1**

[**3**](#_heading=h.tyjcwt) **Instruction encodings 2**

[3.1](#_heading=h.3dy6vkm) Task: Obtaining encodings 2

[3.2](#_heading=h.4d34og8) Exercise: Interpreting the encodings 4

[**4**](#_heading=h.17dp8vu) **Encoding immediate values in Armv8-A 8**

[4.1](#_heading=h.3rdcrjn) Exercise: Move instruction 8

[4.2](#_heading=h.26in1rg) Exercise: ADD/SUB instructions 8

[4.3](#_heading=h.lnxbz9) Logical and bitfield instructions immediate values 9

[4.3.1](#_heading=h.44sinio) Exercise 10

[**5**](#_heading=h.2jxsxqh) **Instruction aliases in Armv8-A 12**

[5.1](#_heading=h.z337ya) Exercise 12

[**6**](#_heading=h.3j2qqm3) **Summary 14**

[**7**](#_heading=h.4i7ojhp) **Additional references 15**

# Introduction

|  |
| --- |
| Lab overview At the end of this lab, you will be able to:   * Use GNU Toolchain to obtain instruction encodings in a human-readable format. * Categorize the Armv8-A AArch64 instruction encodings according to respective bit fields. * Identify what some of the AArch64 instruction encoding fields mean. * Demonstrate how the Armv8-A instructions encode immediate values. * Compare Armv8-A instruction alias with their respective base instructions. |

# Requirements

Before attempting this lab, ensure that you have already completed the installation instructions in the *Getting Started Guide* provided with this course.

The prerequisites for this lab are:

* Familiarity with Arm assembly
* Verilog

This lab requires files generated from Lab 1 and the use of Arm Education Core from Educore-SingleCycle.zip. (Lab 2 comes with Educore-SingleCycle.zip as well, which is similar to the one used in Lab 1).

# Instruction encodings

The Armv8-A instructions have a fixed width of 32 bits. Each instruction is encoded in terms of 32 bits of 1s and 0s. A processor running these instructions will have to decode these encodings.

## Task: Obtaining encodings

In Lab 1, we used t**he objcopy command to obtain these encodings in a Verilog Memory Model hexadecimal format.** These encodings are then read into the “instruction memory” in the Arm Education Core testbench file. The processor then decodes these encodings and executes them accordingly.

To obtain the instruction encodings in a more human-readable format, we shall use the objdump tool that is provided with the GNU Toolchain you downloaded in the *Getting Started Guide*. Follow these steps:

1. Create a folder in your working directory called Lab\_2 (C:\Workspace\Lab\_2).
2. Copy and extract Educore-SingleCycle.zip into yocdur Lab\_2 folder.
3. Copy the final ELF file and corresponding .mem file generated in Lab 1 into the Lab\_2 folder. Make sure you retain the same name of the ELF and mem files (test\_STRCPY.elf and test\_STRCPY.mem).

e

1. Change directory into the Lab\_2 folder and run the GNU objdump tool by using the following command:

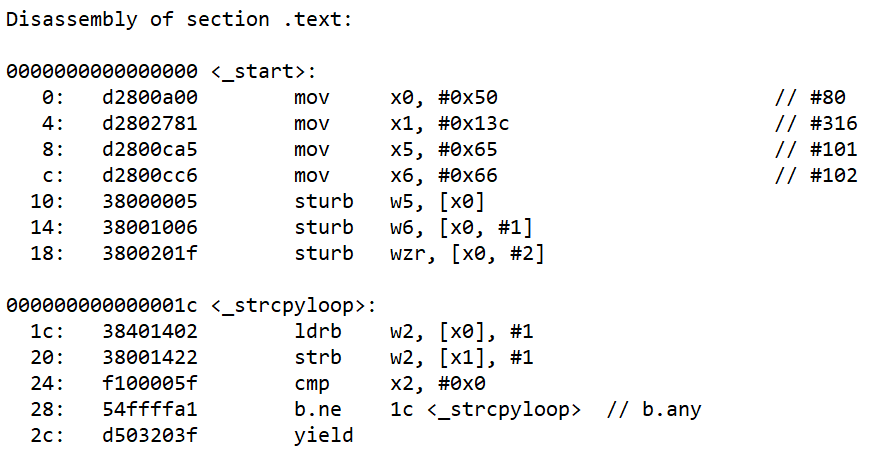
cd Lab\_2

c

**Note:**

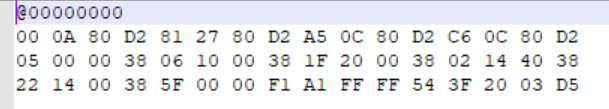
|  |  |
| --- | --- |
| Command/switch | Description |
| aarch64-none-elf-objdump | aarch64-none-elf-objdump <option> file |
| -d | Display assembler contents of executable sections |
| > | This is a Windows operator to shift the displayed contents into a specified file (e.g., log file) |

1. Open the generated test\_STRCPY\_disassembly.log file; the file should have the following content:



*Figure 1: Snapshot of disassembly of test\_STRPCY.elf*

1. Compare the encodings in the test\_STRCPY.mem file you used, as shown below:



**Observations**:

* @00000000 corresponds to -Ttext=0x0 switch during compilation, which specifies starting address (0x0) for the output file.
* The Least Significant Byte of the encoding is stored first.

|  |  |
| --- | --- |
|  | **First instruction encoding** |
| Least Significant Byte | 00 |
|  | 0A |
|  | 80 |
| Most Significant Byte | D2 |

## Exercise: Interpreting the encodings

The following shows the final code used in Lab 1’s test\_STRCPY.s:

.global \_start

.text

\_start:

//place move instructions here

MOVZ X0, #0x0050

MOVZ X1, #0x013C

MOVZ X5, #0x65

MOVZ X6, #0x66

// store values in memory

STURB W5, [X0]

STURB W6, [X0, #1]

STURB WZR, [X0, #2]

//strcpy operation

\_strcpyloop:

// Load byte into W2 from memory pointed to by X0 (\*src). X0 is incremented.

LDRB W2, [X0], #1

// Store byte in W2 into memory pointed to by W2 (\*dst). X1 is incremented.

STRB W2, [X1], #1

CMP X2, #0 // Was the byte 0?

BNE \_strcpyloop // If not, repeat the \_strcpyloop

YIELD

In this exercise, we will decode some of the Armv8-A instructions used in test\_STRCPY.s, specifically these instructions:

* MOVZ
* LDRB
* STRB
* BNE

Based on the encodings obtained in [Task: Obtaining encodings](#_heading=h.3dy6vkm), answer the following questions:

1. For the instruction MOVZ X0, #0x0050, the encoding (as obtained from test\_STRCPY.mem) is:



where 0xD2 is the most significant byte and 0x00 is the least significant byte.

The full syntax of an Armv8-A MOVZ instruction is as follows:

MOVZ <Xd>, #<imm>{, LSL #shift}

where {, LSL #shift} is an optional shift that can be implemented on the immediate value.

The following table shows the instruction encoding format for the MOVZ instruction. Based on the encoding obtained for MOVZ X0, #0x0050, complete the table below:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |
| **Sf** | **1** | **0** | **1** | **0** | **0** | **1** | **0** | **1** | **hw** | | **imm16** | | | | | | | | | | | | | | | | **Rd** | | | | |
|  |  |  |  |  |  |  |  |  |  | |  | | | | | | | | | | | | | | | |  | | | | |

For instruction MOVZ X0, #0x0050, fill up the corresponding encoding values below:

|  |  |  |
| --- | --- | --- |
| **Register field** | **Description** | **Value** |
| sf | If 1, 64-bit variant. (MOVZ <Xd>, …)  If 0, 32-bit variant. (MOVZ <Wd>, …) | b1 |
| hw | Encodes the <shift>.  Usage example: MOVZ X0, #1 , lsl #16 For 64-bit variant: 00 – default 01 – 16 10 – 32  11 – 48 | b00 (no shift specified) |
| imm16 | 16-bit unsigned immediate (0 to 65535) | b0000000001010000 |
| Rd | General-purpose destination registers are encoded here. | b00000 |

1. For the instruction ldrb w2, [x0], #1, the encoding (as obtained from test\_STRCPY.mem) is:



where 0x38 is the most significant byte and 0x02 is the least significant byte.

Full syntax: LDRB <Wt>, [<Xn|SP>], #<simm>

Instruction encoding (post-index version):

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |
| **0** | **0** | **1** | **1** | **1** | **0** | **0** | **0** | **0** | **1** | **0** | **imm9** | | | | | | | | | **0** | **1** | **Rn** | | | | | **Rt** | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  | | | | | | | | |  |  |  | | | | |  | | | | |

For instruction LDRB X2, [X0], #1 , fill up the corresponding encoding values below:

|  |  |  |
| --- | --- | --- |
| **Register field** | **Description** | **Value** |
| imm9 | <simm>  9-bit signed immediate (-256 to 255) | b000000000 |
| Bits[11:10] | 01 – post index.  If pre-index, this encoding would be 11. | b01 |
| Rn | 64-bit general-purpose base register is encoded here, or stack pointer. | b100000 |
| Rt | 32-bit general-purpose destination to be transferred to is encoded here. | b00010 |

Full = b**0**011\_1000\_0100\_0000\_0001\_0100\_0000\_0010

1. For the instruction STRB W2, [X1], #1, the encoding (as obtained from test\_STRCPY.mem) is:



where 0x38 is the most significant byte and 0x22 is the least significant byte.

Syntax: STRB <Wt>, [<Xn|SP>], #<simm>

Instruction encoding (post-index version):

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |
| **0** | **0** | **1** | **1** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **imm9** | | | | | | | | | **0** | **1** | **Rn** | | | | | **Rt** | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  | | | | | | | | |  |  |  | | | | |  | | | | |

For instruction STRB X2, [X1], #1 , fill up the corresponding encoding values below:

|  |  |  |
| --- | --- | --- |
| **Register field** | **Description** | **Value** |
| imm9 | <simm> 9-bit signed immediate (-256 to 255) | b000000001 |
| Bits[11:10] | 01 – post index.  If pre-index, this encoding would be 11. | 01 |
| Rn | 64-bit general-purpose base register is encoded here, or stack pointer. | b00001 |
| Rt | 32-bit general-purpose destination to be transferred from is encoded here. | b00010 |

Full = b0011\_1000\_0000\_0000\_0001\_0100\_0010\_0010

1. For the instruction BNE \_strcpyloop, the encoding (as obtained from test\_STRCPY.mem) is:



where 0x54 is the most significant byte and 0xa1 is the least significant byte.

Syntax: B.cond <label>

Instruction encoding:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |
| **0** | **1** | **0** | **1** | **0** | **1** | **0** | **0** | **imm19** | | | | | | | | | | | | | | | | | | | **0** | **cond** | | | |
|  |  |  |  |  |  |  |  |  | | | | | | | | | | | | | | | | | | |  |  | | | |

For instruction BNE \_strcpyloop , fill up the corresponding encoding values below:

|  |  |  |
| --- | --- | --- |
| **Register field** | **Description** | **Value** |
| cond | Standard Arm conditions | b0001 (cond = ne, not equal) |
| labelm | Program label. Its offset from the address of this instruction, in the range +/−1MB, is encoded as “imm19” times 4. | Imm19= b1111\_1111\_1111\_1111\_1010 << 2  shift by 2 (multiply by 4 = 2^2)  **label = b1111\_1111\_1111\_1110\_1000** |

Full = b0101\_0100\_1111\_1111\_1111\_1111\_1010\_0001

* 1. Notice that the label in the encoding supports a range of positive and negative numbers. What is the integer value of the two’s complement of **imm19**? Does this integer value match the offset between the BNE instruction and the label?

**2’s complement of Imm19:**

1. Invert b1111\_1111\_1111\_1111\_1010 🡪 b0000\_0000\_0000\_0000\_0101
2. Add 1 to b0000\_0000\_0000\_0000\_0101 🡪 b0000\_0000\_0000\_0000\_0100
3. **Integer val:** 4

* **Yes,** the integer value and offset value are both 4 in base 10.

**Note:** The Armv8-A Architecture Reference Manual documents the encodings for all supported instructions.

# Encoding immediate values in Armv8-A

The Armv8-A AArch64 instructions are 32 bits wide. This means that encoding immediate values (constant numbers embedded into the instruction itself) **are constrained by limited space within the instruction encoding.** The AArch64 has clever ways to encode immediate values.

## Exercise: Move instruction

The move instructions (**MOVZ, MOVK,** and **MOVN**) have a 16-bit field for encoding the immediate values (0-65,536 values). The immediate values can be optionally shifted by 0, 16, 32, or 48 bits. In this way, the instruction can accommodate more immediate values.

Answer the following questions:

1. What is the result of X1 in the following instruction? You may use the Arm Education Core to simulate the answer:

MOVZ X1, #0xff, LSL #48

Simulated Output:

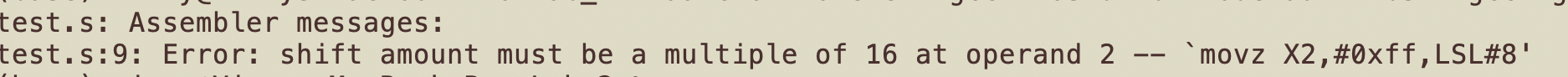
A screen shot of a computer

Description automatically generated

1. Why will you get a compilation error if you executed the following instruction? What alternative instruction can you use instead that would produce the same result?

MOVZ X2, #0xff, LSL #8

Compiling this will result in the following error:



* Because the shift value given is not a multiple of 16, the instruction will not work.
* Alternatively, ….can be used.

## Exercise: ADD/SUB instructions

The **ADD/SUB instructions (including ADDS and SUBS)** have a 12-bit field for encoding immediate values (0 – 4096). It can be optionally shifted by 0 or 12.

Answer the following questions:

1. What is the result of X3 in the following instruction? X1 should still hold the result from [Exercise: Move instruction](#_heading=h.3rdcrjn). You may use the Arm Education Core to simulate the answer:

ADD X3, X1, #0xdd, LSL #12

Output of Testbench:A screenshot of a video game

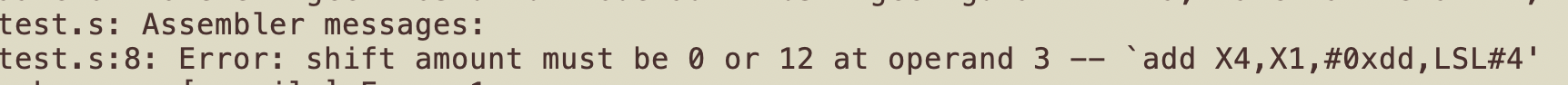
Description automatically generated

* Value for X3 = 00FF0000000DD000

1. Why will you get a compilation error if you executed the following instruction? What alternative instruction can you use instead that would produce the same result?

ADD X4, X1, #0xdd, LSL #4

Error Message:



* Compilation error results from not using 0 or 12 as the shift amount.N

## Logical and bitfield instructions immediate values

The logical instructions (such as **AND, ORR, EOR,** and **ANDS**)and bitfield instructions (such as **SBFM**, **BFM**, and **UBFM**)encode their immediate values with only 13 bits in the instruction encoding. These instructions use a *bitmask* method whereby an immediate value is made up of elements. An element is a sub-pattern that can be 2, 4, 8, 16, 32, or 64-bits in length. The element is then replicated across the register width. **It is not allowed for the element to have either all 1s or all 0s.** Table 1 shows an example of an 8-bit element with pattern “00000011” being replicated across the 64-bit register width.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 00000011 | 00000011 | 00000011 | 00000011 | 00000011 | 00000011 | 00000011 | 00000011 |

*Table 1: 8-bit element replicated across the register width. The element contains pattern 00000011 (imms: 110001, immr=0, N=0)*

The following table shows the instruction encoding format for logical instruction, and in this case, the AND (immediate) instruction.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |
| **sf** | **0** | **0** | **1** | **0** | **0** | **1** | **0** | **0** | **N** | **immr** | | | | | | **imms** | | | | | | **Rn** | | | | | | **Rd** | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  | | | | | | | | |  |  |  | | | | | |  | | | | |

*Table 2: Encoding of the AND (immediate) instruction*

The encoding of the immediate values of logical instructions is done using 13 bits spread across three fields, which are the N-bit (1 bit), imms bits (6 bit) and immr bits (6 bits). This encoding method does not encode for all 2^13 (8192) values. However, it encodes 5334 possible 64-bit numbers, which allows for a useful set of bit patterns, as you will see in the explanation below.

The element pattern and size are encoded using the N-bit and imms in this way:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **N** | **imms[5:0]** | | | | | | **Element size (bits)** |
| 0 | 1 | 1 | 1 | 1 | 0 | X | 2 |
| 0 | 1 | 1 | 1 | 0 | X | X | 4 |
| 0 | 1 | 1 | 0 | X | X | X | 8 |
| 0 | 1 | 0 | X | X | X | X | 16 |
| 0 | 0 | X | X | X | X | X | 32 |
| 1 | X | X | X | X | X | X | 64 |

*Table 3: N and imms encoding*

* The N bit and upper bits of imms (shaded in gray in Table 3) encodes the size of an element.
* The lower imms bits (Xs) encode the number of consecutive 1s in the pattern of the element.
* You would notice in Table 3 that the first most significant “0” *separates* the “upper bits” and “lower bits” of the imms encoding.
* The immr bits just specify the number of right-rotations for the element; it can support up to 63 rotations (for 64-bit element).

For example:

|  |  |  |
| --- | --- | --- |
| **Encoding** | **Meaning** | **Resulting element** |
| N=0, imms=11110**0** immr=0 | 2-bit element. **0** indicates that there is one 1 in the element. Do recall that an element cannot contain all 0s or all 1s. | 0b01 |
| N=0, imms=11110**1**  immr=0 | 2-bit element. **1** indicates that there are two 1s in the element. | Not valid. An element is not allowed to have all 1s or 0s. |
| N=1, imms=**111101**  immr=0 | 64-bit element. **111101** (decimal=61) indicates there are 62 1s in the element. | 0x3fff\_ffff\_ffff\_ffff |

### Exercise

Answer the following question:

1. Complete the following table.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **N** | **imms** | **immr** | **Element size** | **Number of 1s** | **Number of right rotation** | **Resulting element** | **Resulting 64-bit value** |
| 0 | 11110**0** | 000000 | 2 | 1 | 0 | 0b01 | 0x5555\_5555\_5555  \_5555 (i.e. repeat b0101 = x5, 4 times) |
| 0 | 11110**1** | 000000 | 2 | 2 | 0 | Not valid. An element is not allowed to have all 1s or 0s. | N/A |
| 1 | **111101** | 000000 | 64 | 62 | 0 | 0x3fff\_ffff\_ffff  \_ffff | 0x3fff\_ffff\_ffff  \_ffff |
| 0 | 10**0010** | 000000 | 16 | 3 | 0 | 0x0007  or  0b0000\_0000\_0000\_0111 | 0x0007\_0007\_0007\_0007 |
| 0 | 0**01011** | 011110 | Not Valid | 12 | 29 or 30? | Not valid because element size is not encoded properly. | N/A |

1. Create a new Assembly file in your Lab\_2 workspace called test\_Lab2.s:

.global \_start

.text

\_start:

MOV X5, #0xff

AND X6, X5, #0x00003ffc00003f**fc**

ORR X7, X5, #0x00003ffc00003ffc

YIELD

**AND**

0b1111\_1100 [last 8 bits]: 0x00003ffc00003ffc

X5: 0b0000…0000\_1111\_1111: 0x00000000000000ff

------------------------------------------------------------------------

X6 [last 8 bits] = 0b1111\_1100: 0xfc

**ORR**

#0x00003ffc00003f**ff**

1. Simulate the following instructions in Arm Education Core (obtained from Educore-SingleCycle.zip)

cd Lab\_2

aarch64-none-elf-gcc -nostdlib -nodefaultlibs -lgcc -gdwarf-4 -Wa,-march=armv8-a -Wl,-Ttext=0x0 -Wl,-N -o test\_Lab2.elf test\_Lab2.s

aarch64-none-elf-objcopy -O verilog test\_Lab2.elf test\_Lab2.mem

cd Educore-SingleCycle

iverilog -Wall -Wno-timescale -Wno-implicit-dimensions -I head/ -t vvp -y src/ -s test\_Educore src/\* tests/\* -o test\_Educore.vvp

vvp test\_Educore.vvp -lx2 +TEST\_CASE=../test\_Lab2.mem

gtkwave dump.lx2

* 1. What is the result of register X6?
* Result of X6 after ANDing with given immediate value and X5.



* 1. What is the result of register X7?
* Result of X7 after ORing with immediate value and X5.



* 1. Show the waveform for the corresponding N, imms, and immr signals from the Immediate Decoder module for the above AND and ORR instruction. Are the values in the register X6, X7 as expected? Are the values in signals N, imms, and immr as expected?

A screenshot of a computer

Description automatically generated

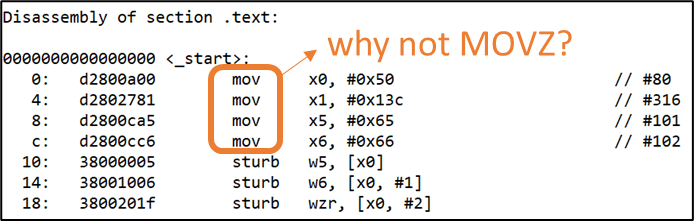
* Yes, the results of the AND and inclusive OR operations are as expected in the registers X6 and X7 respectively.
* Yes, the

# Instruction aliases in Armv8-A

The Armv8-A architecture supports a long list of 32-bit instructions, as documented in the Armv8-A Architecture Reference Manual. A few of these instructions are aliases of their base instructions, where some constants are assigned to the encoded fields of the alias instruction. Doing so accommodates fairly common usage or more naturally readable instruction. The exercise in this section will demonstrate some examples of instruction aliases.

## Exercise

In Figurecd 1, you may have noticed that in the disassembly log file, the MOVZ instruction is being disassembled as a MOV instruction instead. In this exercise, we will investigate why this is the case.



1. Make a copy of the Lab 1’s final test\_STRCPY.s in Lab\_2 folder and name the copied file test\_ALIAS.s.
2. In test\_ALIAS.s, replace the first MOVZ instruction with MOV, and the CMP instruction with SUBS, such as shown in the following code snippet:

\_start:

// address values

**MOV** X0, #0x0050

MOVZ X1, #0x013C

\_strcpyloop:

LDRB W2, [X0], #1

STRB W2, [X1], #1

**SUBS XZR, X2, #0** // Was the byte 0?

BNE \_strcpyloop

YIELD

1. Assemble and run the objdump command to obtain the encodings.

cd Lab\_2

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

aarch64-none-elf-objdump -d test\_ALIAS.elf > test\_ALIAS\_disassembly.log

Answer the following questions:

1. For instruction SUBS XZR, X2, #0 , what is **XZR**? (Hint: Read about Zero Register in [Additional references](#_heading=h.4i7ojhp))

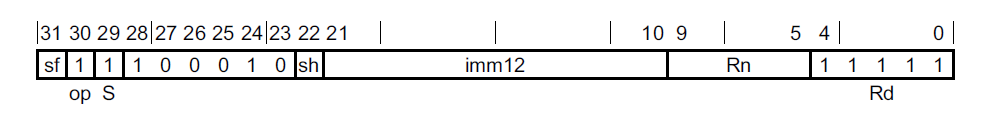
* XZR: This is the designated Zero register, therefore when ….

1. According to the Armv8-A Architecture Reference Manual, MOV (wide immediate) is an alias of MOVZ. MOV (wide immediate) is preferred when the immediate value is not zero and has no optional shift.

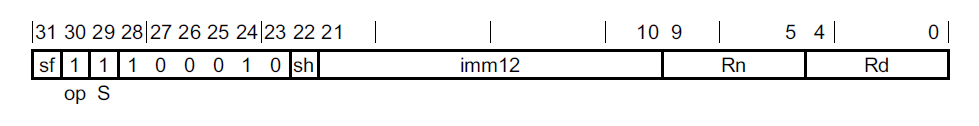
What are your observations for the encodings of MOV X0, #0x0050 and MOVZ X0, #0x0050 instructions?

1. What are your observations for the encodings of SUBS XZR, X2, #0 and CMP X2, #0? Use the diagrams below to explain why your observation is so.

**CMP encoding (alias instruction):**



**SUBS encoding (base instruction):**



# Summary

In this lab, we have learned how encodings are interpreted. We also learned that there are instruction aliases in the Armv8-A ISA. In the next lab, we will take a closer look at single-cycle Arm Education Core’s microarchitecture.

# Additional references

**Armv8-A Architecture Reference Manual**

<https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile>

**Arm Cortex-A Series Programmer’s Guide for Armv8-A (read Armv8 Register chapter)**

<https://developer.arm.com/docs/den0024/a/preface>

**Article on AArch64 immediate values encoding**

* <https://dinfuehr.github.io/blog/encoding-of-immediate-values-on-aarch64/>
* <https://gist.github.com/dinfuehr/51a01ac58c0b23e4de9aac313ed6a06a>

**A64 Base Instructions**

* <https://developer.arm.com/documentation/ddi0596/2021-03/Base-Instructions>

**Zero register**

* <https://developer.arm.com/documentation/den0024/a/ARMv8-Registers/AArch64-special-registers/Zero-register>
* <https://developer.arm.com/documentation/den0024/a/An-Introduction-to-the-ARMv8-Instruction-Sets/The-ARMv8-instruction-sets/Registers>
* <https://developer.arm.com/documentation/den0024/a/ARMv8-Registers>