***Computer Architecture Course***

**Introduction to Arm Education Core**

**Issue 1.0**

**Contents**

[**1**](#_heading=h.30j0zll) **Introduction 1**

[1.1 What is Arm Education Core? 1](#_heading=h.1fob9te)

[1.2 Using Arm Education Core 1](#_heading=h.3znysh7)

[1.2.1 Prerequisite knowledge 1](#_heading=h.2et92p0)

[1.3 Using this document 2](#_heading=h.tyjcwt)

[**2**](#_heading=h.3dy6vkm) **Arm Education Core registers 2**

[2.1 General-Purpose Registers 3](#_heading=h.1t3h5sf)

[2.2 Special registers 3](#_heading=h.4d34og8)

[2.2.1 PSTATE register 4](#_heading=h.17dp8vu)

[**3**](#_heading=h.3rdcrjn) **Instruction handling 5**

[3.1 Data processing instructions 6](#_heading=h.26in1rg)

[3.1.1 Format 6](#_heading=h.lnxbz9)

[3.1.2 Encoding 7](#_heading=h.35nkun2)

[3.2 Load and Store instructions 8](#_heading=h.1ksv4uv)

[3.2.1 Encoding 8](#_heading=h.44sinio)

[3.3 Branch and System instructions 9](#_heading=h.2jxsxqh)

[3.3.1 Encoding 9](#_heading=h.z337ya)

[3.4 Encoding immediate values in A64 instructions 10](#_heading=h.3j2qqm3)

[**4**](#_heading=h.1y810tw) **Functional description 10**

[4.1 Processing stages 10](#_heading=h.4i7ojhp)

[**5**](#_heading=h.2xcytpi) **List of supported instructions 10**

[**6**](#_heading=h.1ci93xb) **Additional references 11**

[**7**](#_heading=h.3whwml4) **Appendix 12**

[7.1 Designing a simple processor from scratch 12](#_heading=h.2bn6wsx)

# Introduction

## What is Arm Education Core?

The Arm Education Core is a processor coredeveloped for the Arm Computer Architecture Education Kit, with the intention of serving as a baseline processor with limited capabilities that students can gradually improve, based on the instructions of the lab manuals provided.

Each lab of the Computer Architecture Education Kit involves modifying Arm Education Core using Verilog so that students can apply concepts such as instruction processing stages, single-cycle instruction processing, pipelined instruction processing, data hazards, stalls, and forwarding paths. Students are first provided a single-cycle version of Arm Education Core.

Arm Education Core is written in Verilog 2005 ([IEEE Standard 1364-2005](https://standards.ieee.org/findstds/standard/1364-2005.html)) Hardware Description Language (HDL) and implements a subset of the Armv8-A A64 Instruction Set Architecture. A subset was chosen to elucidate processor design concepts for educational purposes. Arm Education Core is a 64-bit architecture that accepts instructions that are 32-bit wide.

**Usage of the Arm Education Core is subjected to the** [***Arm Education Introduction to Computer Architecture Education Kit End User License Agreement***](https://www.arm.com/-/media/Files/pdf/education/computer-architecture-education-kit-eula)***.***

## Using Arm Education Core

**Note:**

* **You MUST always only use the subset of Arm Instruction Set provided in this document when using Arm Education Core.** Instructions other than the specified subset are NOT guaranteed to produce desired outcomes.
* Please contact Arm Education to file a bug report or suggest improvements.We appreciate any feedback and contributions to make Arm Education Core better for educators and students.
* **Please read the** [***Arm Education Introduction to Computer Architecture Education Kit End User License Agreement***](https://www.arm.com/-/media/Files/pdf/education/computer-architecture-education-kit-eula) **for terms of usage of Arm Education Core.**

There is also a Getting Started document provided with the Computer Architecture course, which covers an initial setup of tools to compile or synthesize Arm Education Core.

### Prerequisite knowledge

To understand Arm Education Core, you will need prior knowledge or familiarity with:

* Coding in Verilog.
* Basic understanding of Arm assembly codes.

## Using this document

This document introduces the Arm Education Core and contains the relevant information in order for you to understand how to progressively improve Arm Education Core in the Computer Architecture course labs. It is therefore recommended that you read this document before attempting any labs in the Computer Architecture course.

# Arm Education Core registers

Arm Education Core implements General-Purpose Registers and Special Registers, as shown in the following diagram.

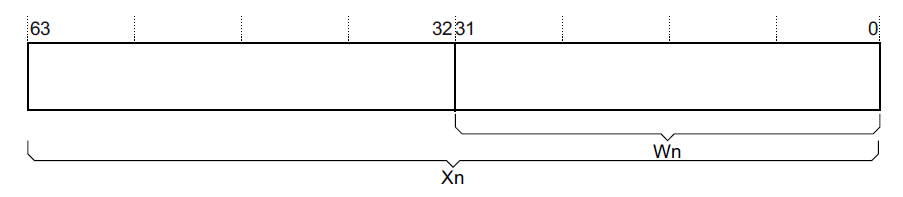
|  |  |  |
| --- | --- | --- |
| General-Purpose Registers  Link Register  (There is no real **X31/W31**, see **Note** in [Special registers](#_heading=h.4d34og8)) | | **X0/W0** |
| **X1/W1** |
| **X2/W2** |
| **X3/W3** |
| **X4/W4** |
| **X5/W5** |
| **X6/W6** |
| **X7/W7** |
| **X8/W8** |
| **X9/W9** |
| **X10/W10** |
| **X11/W11** |
| **X12/W12** |
| **X13/W13** |
| **X14/W14** |
| **X15/W15** |
| **X16/W16** |
| **X17/W17** |
| **X18/W18** |
| **X19/W19** |
| **X20/W20** |
| **X21/W21** |
| **X22/W22** |
| **X23/W23** |
| **X24/W24** |
| **X25/W25** |
| **X26/W26** |
| **X27/W27** |
| **X28/W28** |
| **X29/W29** |
| **X30/W30** |
| **\*SP or \*0** |
| Special Registers | Program Counter | **PC** |
| Stack Pointer | **SP** |
| PSTATE Control Register | **PSTATE** |

*Figure 1: Arm Education Core registers*

## General-Purpose Registers

There are 31 64-bit General-Purpose Registers in Arm Education Core. Similar to the Armv8-A architecture, each register is 64-bit wide and they are generally referred to as registers **X0-X30**. **X30** is also known as the Procedural Link Register, which holds the return address of a subroutine.

Each 64-bit General-Purpose Register (**X0-X30**) also has a 32-bit form (**W0-W30**), as shown in the following diagram. The 32-bit **W** register maps to the lower half of the corresponding **X** register. For example, **W0** maps onto the lower word (32-bits) of **X0**, and **W1** maps onto the lower word of **X1**.



*Figure 2: 64-bit register with W and X access*

## Special registers

In addition to the 31 General-Purpose Registers, Arm Education Core also has several “special” registers (non-general-purpose registers), which are shown in Table 1.

|  |  |
| --- | --- |
| **Special register** | **Description** |
| Program Counter | This register holds the address of the instruction that is to be executed. |
| Stack Pointer | This register holds the address of the stack location of the most recent item put on the stack. A stack is a reserved area in memory that a program may use for internal operations. |
| PSTATE Control Register | See [PSTATE register](#_heading=h.17dp8vu). |

*Table 1: List of Arm Education Core special registers*

**Note:**

* The Armv8-A instruction set uses a 5-bit field to encode which General-Purpose Register is to be accessed. This 5-bit field allows for a total of 32 addresses (2^5=32), which means that the architecture could actually accommodate 32 General-Purpose Registers. In Armv8-A, this “register number 32” isn’t really a General-Purpose Register X31, but some of the Armv8-A instructions that Arm Education Core supports are encoded in a way such that register number 32 represents either the “Zero Register” or the Stack Pointer. More information on this in [Instruction handling](#_heading=h.3rdcrjn).
* Arm Education Core does not implement the “Zero Register” as a physical register where all the bits are set to zero. Instead, if the “Zero Register” is used by a specific instruction, then Arm Education Core sets the values to zero by doing a logical AND in Verilog.

### PSTATE register

PSTATE stands for Processor State. The register fields in a traditional *Current Processor Status Register* (CPSR) in Armv7-A are referred to collectively as the PSTATE in Armv8-A (AArch64).

In Arm Education Core, the PSTATE fields only include the NZCV flags, as shown in the tables below. These flags serve as a simple indication of whether the computed result is negative, zero, a carry out, or overflow.

|  |  |  |  |
| --- | --- | --- | --- |
| **PSTATE Bit 3** | **PSTATE Bit 2** | **PSTATE Bit 1** | **PSTATE Bit 0** |
| N | Z | C | V |

*Table 2: PSTATE bits*

|  |  |
| --- | --- |
| **Flags** | **Description** |
| N | Negative result. This flag is set to 1 when the result is negative. |
| Z | Zero result. This flag is set to 1 when the result is zero. |
| C | Carry out (or Unsigned Overflow). This flag is set to 1 if the result of an unsigned operation overflows. |
| V | Overflow (Signed). This flag is set to 1 if the result for a signed operation overflows. |

*Table 3: NZCV flags in PSTATE*

# Instruction handling

**Arm Education Core only implements a subset of the Arm-v8A A64 Instruction Set Architecture (ISA).** **You must only use the subset of instructions provided in this document.** If you run instructions that are not specified in the subset provided, Arm Education Core is not guaranteed to produce desired outcomes.

Arm Education Core supports the following instruction types:

* Data processing (immediate and register)
* Load/stores
* Branch/system

The A64 ISA 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.

|  |
| --- |
| Note:   * This document contains only the information on the subset of Armv8-A instructions used. * See List of Supported Instructions. * See Excel spreadsheet of encoding * Some instructions that Arm Education Core supports are encoded in a way register number 32 represents either a Stack Pointer (SP) or is set to zero (indicated by RZR—“zero register”). See the *Arm Education Core supported instructions.xlsx* for the full list of supported instruction encodings. For instructions that can deal with the stack, register number 32 is the SP. For all other instructions, it is zero. * In the encoding, fields that change the type of operation tend to be called opN or opcode. |

## Data processing instructions

Data processing instructions are fundamental arithmetic, logical, move, bit manipulation, and extract operations of the processor. There are generally 2 groups of data processing instructions:

* **Data processing (immediate)** operates on **immediate values**.
* **Data processing (register)** operates on values in **registers**.

### Format

Most data processing instructions have 1 destination register and two source operands. There are a few instructions that have a different format (e.g., ADR or EXTR), but most data processing instructions use the following general format:

Instruction <Rd>, <Rn>, <Operand2>

where <Rd> is the destination register,

<Rn> is the first operand (a register),

<Operand2> might be a register, or an **immediate value**.

The <R\*> in the format above indicates that it can either be a X or a W register. If the instruction does not support W registers, then <X\*> will be used in the format.

**Example:**

ADD X0, X1, #1 // X0 = X1 + 1

AND X2, X1, X0 // Logical AND registers X1 and X0, store result in X2

### Encoding

**Data processing (immediate) instructions**

The encoding for data processing (immediate) instructions is as follows:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |
|  |  |  | 100 | | | op1 | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| **op1[25:23]** | **Instruction type** | **Examples** |
| 00x | PC-relative addressing | ADR, ADRP |
| 010 | Add/subtract immediate | ADD, SUB, ADDS, SUBS |
| 100 | Logical immediate | AND, ORR, EOR, ANDS |
| 101 | Move wide immediate | MOVN, MOVZ, MOVK |
| 110 | Bitfield move | BFM, SBFM, UBFM |
| 111 | Extract | EXTR |

For a full list of supported instructions and all its encoding (including grayed-out cells above), see *Arm Education Core supported**instructions.xlsx*.

**Data processing (register) instructions**

The encoding for data processing (register) instructions are as follows:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |
|  | op1 |  | op2 | 101 | | | op3 | | | |  |  |  |  |  | op4 | | | | | |  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **op1[30]** | **op2[28]** | **op3[24:21]** | **op4[15:10]** | **Instruction type** | **Examples** |
| x | 0 | 0xxx | xxxxxx | Logical (shifted register) | AND, BIC, ORR |
| x | 0 | 1xx1 | xxxxxx | Add/subtract (extended register) | ADD, SUB |
| x | 1 | 0000 | 000000 | Add/subtract with carry | ADD, SUB |
| x | 1 | 0010 | xxxx0x | Conditional compare register | CCMN, CCMP |
| x | 1 | 0100 | xxxxxx | Conditional select | CSEL, CSINV |

For a full list of supported instructions and all its encoding (including grayed-out cells above), see *Arm Education Core supported**instructions.xlsx*.

## Load and Store instructions

The Arm architecture is a Load/Store architecture, which means no data processing instruction operates directly on data in memory. The data must be first loaded into registers, modified, and then saved in memory.

Load instructions load data from memory and have the following general format:

LoadInstruction <Rt>, <addr>

where <Rt> is the register to be loaded to,

<addr> might be a register, or an **immediate value**.

The <R\*> in the format above indicates that it can either be a X or a W register. If the instruction does not support W registers, then <X\*> will be used in the format.

Similarly, the store instructions store data to memory and have the following general format:

StoreInstruction <Rn>, <addr>

where <Rn> is the register to be stored,

<addr> might be a register, or an **immediate value**.

**Example:**

LDUR X0, [X9] // Loads X0 with data from memory address in X9

STUR X2, [X1] // Stores data in X2 to memory at address in X1

### Encoding

The encoding for load and store instructions is as follows:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |
| op1 | | | | 1 | op2 | 0 | op3 | |  | op4 | | | | | |  |  |  |  | op5 | |  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **op1[31:28]** | **op2[26]** | **op3[24:23]** | **op4[21:16]** | **op5[11:10]** | **Instruction type** | **Examples** |
| xx01 | x | 0x | xxxxxx | xx | Load register (literal) | LDR X0, <label> |
| xx11 | x | 0x | 0xxxxx | 00 | Load/Store register (unscaled immediate) | STURB W0, [X1, #2] |
| xx11 | x | 0x | 0xxxxx | 01 | Load/Store register (immediate post-indexed) | STRB W0, [X1], #2 |
| xx11 | x | 0x | 0xxxxx | 11 | Load/Store register (immediate pre-indexed) | STRB W0, [X1, #3] |
| xx11 | x | 0x | 1xxxxx | 10 | Load/Store register (register offset) | LDR X0, [X1, X2, LSL #3] |

For a full list of supported instructions and all its encoding (including grayed-out cells above), see *Arm Education Core supported**instructions.xlsx*.

## Branch and System instructions

Branch instructions change the Program Counter (PC) to a different address to begin executing a different instruction sequence.

System instructions are generally instructions that are related to system register access or control, debug, or exception handling (note that Arm Education Core does not currently support exception handling).

There are several different types of branch instructions and system instructions, and their format varies according to the type. For more information, see *Arm Education Core supported**instructions.xlsx* or the Arm Architecture Reference Manual in [Additional references](#_heading=h.1ci93xb).

### Encoding

The encoding for the branch and system instructions is as follows:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |
| op1 | | | 101 | | | op2 | | | | | | | | | | | | | |  |  |  |  |  |  |  | op3 | | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **op1[31:29]** | **op2[25:12]** | **op3[4:0]** | **Instruction type** | **Examples** |
| 010 | 0xxxxxxxxxxxxx | xxxxx | Conditional Branch (immediate) | B.cond |
| 110 | 01000000110010 | 11111 | Hints | NOP, YIELD |
| 110 | 0100x1xxxxxxxx | xxxxx | System register move | MSR, MRS |
| 110 | 1xxxxxxxxxxxxx | xxxxx | Unconditional branch (register) | BR, BLR, RET |
| x00 | xxxxxxxxxxxxxx | xxxxx | Unconditional branch (immediate) | B, BL |

For a full list of supported instructions and all its encoding (including grayed-out cells above), see *Arm Education Core supported**instructions.xlsx*.

## Encoding immediate values in A64 instructions

The Arm A64 ISA is a fixed-width ISA, meaning its fields have a fixed width, as shown in the encoding information of this document. This means that the encoded immediate has a very limited range of values. To accommodate this, the A64 has a way of encoding its immediate values.

More information on this is in one of the lab exercises provided with the Computer Architecture Education Kit.

# Functional description

## Processing stages

Arm Education Core implements 5 stages of processing that are:

* FETCH—Instruction is fetched.
* DECODE—Instruction is decoded.
* EXECUTE—Instruction is executed, and the result is calculated.
* MEMACCESS—Memory is accessed for Data Transfer-type instructions (Load/Store)
* WRITEBACK—Results are written back into the Register File.

More information on this is in one of the lab exercises provided with the Education Kit.

# List of supported instructions

See the *Arm Education Core supported**instructions.xlsx*. Description of each instruction can be found in the Armv8-A Architecture Reference Manual, see [Additional references](#_heading=h.1ci93xb).

# Additional references

**Armv8-A programmer’s guide**

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

**Armv8-A Architecture Reference Manual**

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

* See **A64 Base Instruction Descriptions** chapter
* See **A64 Instruction Set Encoding** chapter

**Armv8-A immediate encoding**

<https://dinfuehr.github.io/blog/encoding-of-immediate-values-on-aarch64/>

<https://gist.github.com/dinfuehr/51a01ac58c0b23e4de9aac313ed6a06a>

# Appendix

## Designing a simple processor from scratch

****