Instruction Set Architecture Design and Implementation

Learning Outcomes

At the end of this lab, you should be able to:

1. understand and modify a simple ISA and its implementation;
2. write a simple assembler

Contents

1 Resources

- Video: https://youtu.be/u__T3TgqjRg.
- Source Codes: https://git.io/JTK0T

2 Discussion

The **processor(CPU)** is composed of the **datapath** and **control**. In the previous labs, you learned that combinational and sequential circuit elements are used as building blocks to create the functional components of the datapath and control. Examples of these functional elements include the **ALU, Register File, Program Counter, and Memory**. You also learned that a **clock** drives the execution forward and control is implemented using a **finite state machines** for the **fetch-decode-execute** cycle. One question that we can answer next is: *How do we program the CPU?*

2.1 Instruction Set Architecture

Instruction Set Architecture (ISA) is an **abstraction** between the hardware and the lowest-level software. It includes anything programmers need to know to make a binary machine language program work correctly. Typically it documents the set of instructions that can be performed by the processor, the number and name of available registers, the memory addressing modes, I/O, interrupt processing, etc [1].

ISA allows computer designers to talk about functions indepedently from the hardware that performs them. This abstact interface enables many implementations (aka **microarchitectures**) of varying costs and performance to run identical software.

Examples of ISA include the **IA-32** and **x86-64**[2] which are commonly used in desktop and laptops. For example, Intel implements these ISA in their Intel Core i5-8250U[3] product as **8th Generation aka Kaby Lake Refresh**[4]. AMD also implements these ISA in the Ryzen 5000 [5] as **4th Gen aka Zen 3**[6]. There are other implementations(aka generations) that vary in their performance characteristics.

For mobile devices, a popular ISA is the **ARMv8 A64**[7]. MediaTek uses ARM's **Cortex-A73** and **Cortex-A53**[8] implementations in their Octa-core Helio P70[9]. Qualcomm also uses the same implementations in their Kryo 240 processor for Snapdragon 460 Mobile Platform[10].

2.2 Application Binary Interface

Application Binary Interface (ABI) is a combination of the basic instruction set and the operating system interface provided for application programmers[1].

For general-purpose use such a desktops and laptops, programming a processor using only the basic instruction set is insufficient. Thus, operating systems perform an important role in the management and efficient use of hardware resources in addition to making it easier for users to use a computer.

ABI describes function-calling conventions, parameter passing, sizes of C data types, executable file formats (ELF, PE). Examples are the IA-32 and **x86-64 System V ABI**[11] which are used in Linux and other Unix-type operating systems. Windows has its own ABI called **x64 ABI**[12]. Android supports different ABIs[13].

2.3 ISA Taxonomy

We can categorize ISAs based on where operands in instructions are stored. **Stack-based** ISAs use a stack(LIFO) where operations are performed on the operands on the top of the stack. In **accumulator-based** ISAs, one register is designated as the *accumulator* and its use in operations is implied. Modern ISAs are **general purpose** where operands are explicitly named in the instruction. In this category, operations can be register-to-register, register-to-memory, or memory-to-register.

2.4 Considerations in ISA Design

- *Types/Class of instructions(Operations in the instruction set)* - arithmetic/logic, data movement, branching/control flow, I/O, etc.
- *Types and sizes of operands (in bits)* - 8, 16, 32, 64, 128, floating point

- *Addressing modes* - register, direct, indirect, immediate, etc.
- *Addressing memory* - byte-addressable, word-addressable, endian-ness
- *Encoding and Instruction Formats* - opcode field, addresses field, mode field
- *Compiler-related issues* - optimization features

2.5 Example Instruction Formats

Figure 1 and Figure 2 are the instruction formats for x86-64 and ARMv8, taken from their documentation manuals. The x86-64 format is more complex than that of the ARMv8 with variable widths in terms of number of bits for the opcode.

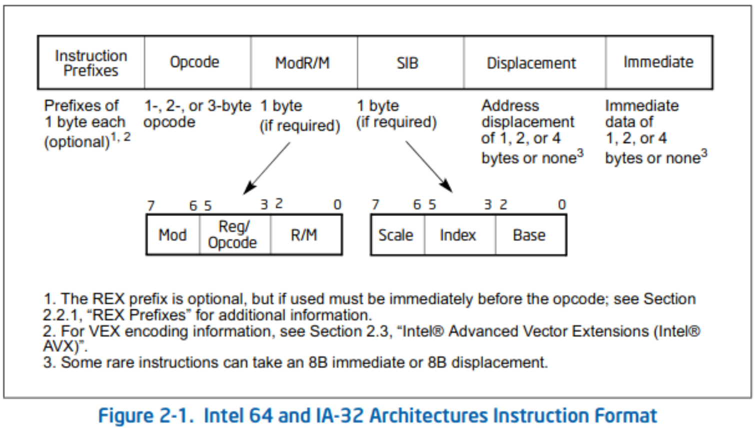

Figure 1: x86-64 Instruction Format (CISC).

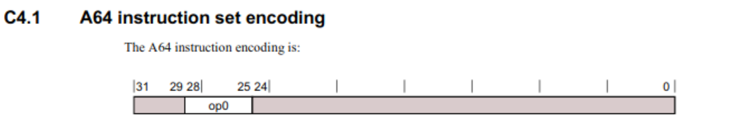

Table C4-1 Main encoding table for the A64 instruction set

Decode fields op0	Decode group or instruction page
0000	*Reserved*
0001	Unallocated.
0010	SVE Instructions. See *The Scalable Vector Extension (SVE)* on page A2-99.
0011	Unallocated.
100x	*Data Processing -- Immediate*
101x	*Branches, Exception Generating and System instructions* on page C4-271
x1x0	*Loads and Stores* on page C4-279
x101	*Data Processing -- Register* on page C4-310
x111	*Data Processing -- Scalar Floating-Point and Advanced SIMD* on page C4-320

Figure 2: ARMv8 Instruction Format (RISC).

2.6 TOMA: The Optimal Machine Architecture

Let us look at the design of a simple ISA which we will call TOMA.

2.6.1 Features

- Four 8-bit registers named `$s0`, `$s1`, `$s2`, `$s3` when used in assembly code
- Instruction memory (IM) is 8 bytes(8x8), address line is 3 bits
- Three-bit Program Counter (PC)
- Single-cycle - completes instruction execution in one clock cycle
- Supports the following instructions: `and`, `add`, `sub`, `addi`
- No data memory, thus has no load and store instructions
- No control transfer instructions

2.6.2 Instruction Format

The size of an instruction in TOMA is 8 bits divided into the configuration shown in Figure 3.

Figure 3: TOMA instruction format.

2.6.3 Supported Instructions

Listing 1: Supported Instructions in TOMA.

```
and : rd <= rs AND rt           (op=00)
add : rd <= rs + rt             (op=01)
sub : rd <= rs -  rt            (op=10)
addi : rs <= rt + immediate     (op=11)
```

2.6.4 Assembly Language

The syntax for the assembly language is shown in Listing 2.

Listing 2: Assembly language syntax.

```
<instruction> <dst>, <src1>, <src2/imm>
```


Listing 3: Example assembly code and corresponding machine code.

```
addi $s0, $s0, 2          ; 11000010b, 0xC2
addi $s1, $s1, 1          ; 11010101b, 0xD5
addi $s2, $s2, 3          ; 11101011b, 0xEB
add $s3, $s0, $s1         ; 01000111b, 0x47
sub $s0, $s2, $s3         ; 10101100b, 0xAC
```

2.7 REDHORSE 500: An implementation of TOMA

2.7.1 Features

- Implements the TOMA ISA in VHDL
- Clocked at 100MHz

2.7.2 Processor

The interface to the processor is shown in Listing 4 which shows that it has one input which is the clock and two outputs. Figure 4 shows the complete wiring of the functional components. We will discuss the operation of the individual components in the remaining subsections.

Listing 4: Interface to the processor.

```
ENTITY Processor IS
    PORT
    (
        clk :  IN  STD_LOGIC;
        current_instruction :  OUT  STD_LOGIC_VECTOR(2 DOWNTO 0);
        value :  OUT  STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
END Processor;
```

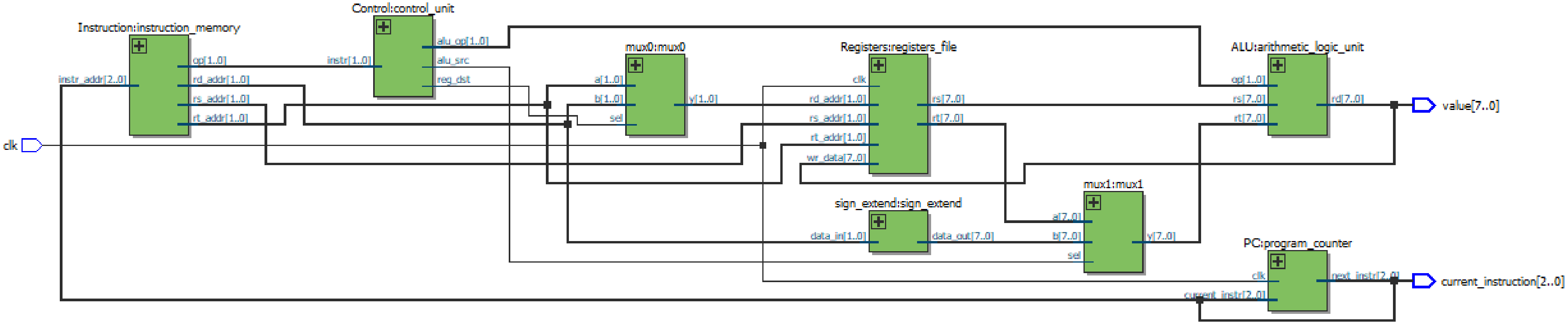

Figure 4: Wiring of the different functional components of REDHORSE 500.

2.7.3 Program Counter

The Program Counter will generate the address of the next instruction. It adds 1 to the value of `current_instr`.

Listing 5: Interface to the Program Counter.

```
entity PC is
  port(
    clk           : in std_logic;
    current_instr : in std_logic_vector(2 downto 0);   -- current instruction
    next_instr    : out std_logic_vector(2 downto 0)   -- next instruction
  );
end PC;
```

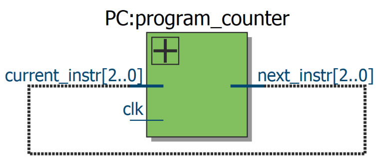

Figure 5: Program Counter.

2.7.4 Instruction Memory

Instruction Memory contains the instructions to be executed in an array of 1-byte cells. It will decode the instruction specified by `instr_addr`. The decode process will extract and output the different parts of the instruction: `op`, `rs_address`, `rt_address`, `rd_address`/`imm`.

Listing 6: Interface to the Instruction Memory.

```
entity Instruction is
  port(
    instr_addr : in std_logic_vector(2 downto 0);  -- instruction address

    op         : out std_logic_vector(1 downto 0); -- operation code
    rs_addr    : out std_logic_vector(1 downto 0); -- source register 1 addr
    rt_addr    : out std_logic_vector(1 downto 0); -- source register 2 addr
    rd_addr    : out std_logic_vector(1 downto 0)  -- dest register addr
  );
end Instruction;
```


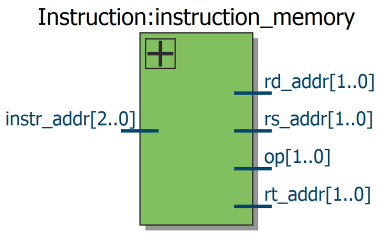

Figure 6: Instruction Memory.

Listing 7: Sample hard-coded instructions in the instruction memory.

```
constant instr : instruction_set := (
  "11000010",  -- addi $s0, $s0, 2
  "11010101",  -- addi $s1, $s1, 1
  "11101011",  -- addi $s2, $s2, 3
  "01000111",  -- add  $s3, $s0, $s1
  "10101100",  -- sub  $s0, $s2, $s3
  "00000000",
  "00000000",
  "00000000"
);
```

2.7.5 Register File

The Register File is composed of four 8-bit registers. The contents of the registers specified by `rs_addr` and `rt_addr` are the outputs `rs` and `rt`. At the falling edge of the clock, the data in `wr_data` is written into the register specified by `rd_addr`. The `rd_addr` input will come from Mux0 in case the instruction is `addi`(See Figure 4).

Listing 8: Interface to the Register File.

```
entity Registers is
  port(
    clk     : in std_logic;

    rs_addr : in std_logic_vector(1 downto 0);    -- source register 1 address
    rt_addr : in std_logic_vector(1 downto 0);    -- source register 2 address
    rd_addr : in std_logic_vector(1 downto 0);    -- destion register address
    wr_data : in std_logic_vector(7 downto 0);    -- write data to dest register

    rs      : out std_logic_vector(7 downto 0);   -- source register 1 value
    rt      : out std_logic_vector(7 downto 0)    -- source register 2 value
  );
end Registers;
```

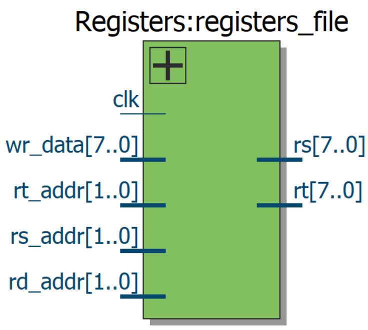

Figure 7: Register File.

Listing 9: Sample hard-coded initial values for the registers.

```
    signal reg: registerFile := (
     "00000001",
     "00000010",
     "00000011",
     "00000100"
);
```

2.7.6 ALU

The ALU performs the supported instructions depending on the opcode in `op`. This is an 8-bit ALU so operations are performed on 8-bit values. The inputs are in `rs` and `rt` and the results in `rd`. `rt` input will come form Mux1 in case the instruction is `addi`(See Figure 4).

Listing 10: Interface to the ALU.

```
entity ALU is
  port(
    op  : in std_logic_vector(1 downto 0);    -- operation code

    rs  : in std_logic_vector(7 downto 0);    -- source register 1
    rt  : in std_logic_vector(7 downto 0);    -- source register 2
    rd  : out std_logic_vector(7 downto 0)    -- destination register
  );
end ALU;
```

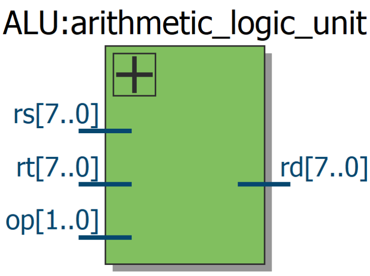

Figure 8: ALU.

2.7.7 Control Unit

The Control Unit activates control lines depending on the opcode in `instr`. It tells the ALU what operation to perform through the `alu_op` line. A special case is the `addi` instruction which activates the `alu_src` and `reg_dst` to adjust the inputs to the ALU and the Register File respectively (See Figure 4).

Listing 11: Interface to the Control Unit.

```
entity Control is
  port(
    instr   : in std_logic_vector(1 downto 0);   -- instruction

    alu_op  : out std_logic_vector(1 downto 0);  -- operation code of AlU
    alu_src : out std_logic;                     -- ALU select ADDi
    reg_dst : out std_logic                      -- select destination address
    register
  );
end Control;
```


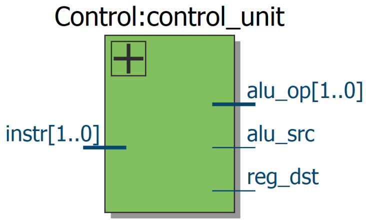

Figure 9: Control Unit.

2.7.8 Mux0

Mux0 will decide whether the `rd_addr` input to the Register File will come from the `rt_addr` output or `rd_addr` output of the Instruction Memory. If the instruction is `addi`, then the Register File `rd_addr` input will come from the `rt_addr` output of the Instruction Memory (See Figure 4).

Listing 12: Interface to Mux0.

```
entity mux0 is
  port(
    sel : in std_logic;                         -- select destination address
    a   : in std_logic_vector(1 downto 0);      -- source register address
    b   : in std_logic_vector(1 downto 0);      -- default destination address
    y   : out std_logic_vector(1 downto 0)      -- destination address
  );
end mux0;
```

2.7.9 Mux1

Mux1 will decide whether the `rt` input to the ALU will come from register `rt_addr` or from the sign-extended immediate value if the instruction is `addi`(See Figure 4).

Listing 13: Interface to Mux1.

```
entity mux1 is
  port(
    sel : in std_logic;                         -- select data
    a   : in std_logic_vector(7 downto 0);      -- default data
    b   : in std_logic_vector(7 downto 0);      -- data from instruction
    y   : out std_logic_vector(7 downto 0)      -- data out
  );
end mux1;
```


2.7.10 Sign Extend

If the instruction is `addi`, then the 2-bit immediate value which is from the `rd_addr` output of the Instruction Memory will be converted to 8 bits before being passed to the ALU which perform operations on 8-bit values.

Listing 14: Interface to Sign Extend.

```
entity sign_extend is
  port(
    data_in  : in std_logic_vector(1 downto 0);
    data_out : out std_logic_vector(7 downto 0)
  );
end sign_extend;
```

2.7.11 Simulation

Figure 10 shows the simulation with the red vertical bar marking the execution of the instruction at line 4 of Listing 7.

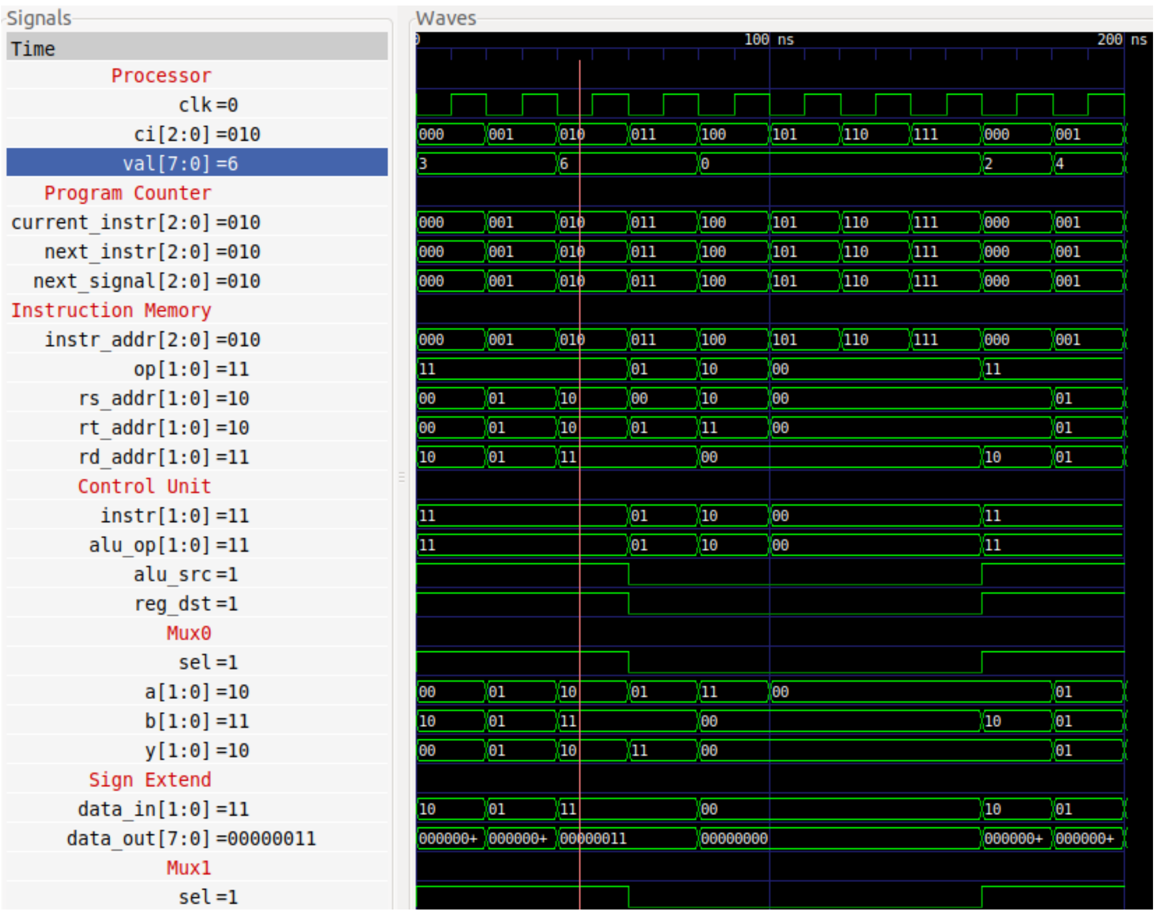

Figure 10: Simulation run of REDHORSE 500.

3 Summary

In this lab, you learned that ISAs enable programmers to program the processor without knowing the low-level hardware implementation details. It also allows multiple vendors to develop different implementations with varying performance characteristics and costs.

You also learned that an ISA combined with an operating system's services is an ABI that allows application programmers to write programs that will run on the specific microarchitecture that implememnts the ISA and supported by that operating system.

Lastly, you were also introduced to the simple TOMA ISA and its implementation, the REDHORSE 500.

4 Self-Assessment Questions

1. What is an ISA?
2. What is an ABI?
3. Is it possible to run linux applications compiled/built for IA-32 on an x86-64?Why or why not?
4. What is purpose of Mux0 and Mux1 in REDHORSE 500?
5. Why is the sign extension component needed in REDHORSE 500?
6. Can you consider TOMA a RISC or a CISC?

5 Learning Activities and Deliverables

Download and study the source codes for this lab. Run the script, `build.sh`, to build everything. To check the operation of REDHORSE 500, there is a file named `redhorse500.gtkw`. Open this file in `gtkwave` and study the simulation.

5.1 Write a testbench

You will observe that there is no testbench in the source codes. Write a testbench and experiment with using different clock frequencies starting with 100MHz. Capture screenshots of the simulations generated using your own testbenches.

Deliverables: `Processor_tb.vhdl`, simulation screenshots

5.2 Write an assembler for TOMA

Write an assember that reads a file of TOMA assembly language code (see Assembly Language) and outputs a file containing binary strings representing the machine code. You can use any programming/scipting language you want.

Deliverables: `tomasm.py` or depending on what language you used. Only submit source code.

5.3 Test your assembler

Using your assembler, assemble the code in Listing 15. Plug in the result to the Intruction Memory. Simulate the execution.

Listing 15: surprise.asm.

```
add  $s0, $s0, $s1
addi $s0, $s3, 6
and  $s2, $s3, $s1
sub  $s0, $s3, $s0
```

Deliverables: Assembled output and screenshots of simulations.

5.4 Discussion

Discuss the changes you have to make to TOMA and REDHORSE 500 if you are to use the ALU you implemented in the Combinational Circuits lab.

Deliverables: Paragraphs describing your approach.

References

[1] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface, ARM Edition.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, arm edition, 2017.

[2] Amd64 architecture programmer's manual volume 1:application programming. `https://www.amd.com/system/files/TechDocs/24592.pdf`. Accessed: 2020-10-26.

[3] Intel core i5-8250u processor (6m cache, up to 3.40 ghz) product specifications. `https://ark.intel.com/content/www/us/en/ark/products/124967/intel-core-i5-8250u-processor-6m-cache-up-to-3-40-ghz.html`. Accessed: 2020-10-26.

[4] Products formerly kaby lake r. `https://ark.intel.com/content/www/us/en/ark/products/codename/126287/kaby-lake-r.html`. Accessed: 2020-10-26.

[5] Ryzen 5000 series desktop processors | fastest in the game | amd. `https://www.amd.com/en/processors/ryzen`. Accessed: 2020-10-26.

[6] Amd zen 3 release date, specs and price: everything we know about amd ryzen 4000 | techradar. `https://www.techradar.com/news/amd-zen-3`. Accessed: 2020-10-26.

[7] Armv8-a instruction set architecture. `https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Learn%20the%20Architecture/Armv8-A%20Instruction%20Set%20Architecture.pdf`. Accessed: 2020-10-26.

[8] Cortex-a73 âĂŞ arm developer. `https://developer.arm.com/ip-products/processors/cortex-a/cortex-a73`. Accessed: 2020-10-26.

[9] Mediatek helio p70. `https://www.mediatek.com/products/smartphones/mediatek-helio-p70`. Accessed: 2020-10-26.

[10] Snapdragon 460 mobile platform | qualcomm. `https://www.qualcomm.com/products/snapdragon-460-mobile-platform`. Accessed: 2020-10-26.

[11] Projects Âů x86 psabis / x86-64 psabi Âů gitlab. `https://gitlab.com/x86-psABIs/x86-64-ABI`. Accessed: 2020-10-26.

[12] x64 software conventions | microsoft docs. https://docs.microsoft.com/en-us/cpp/build/x64-software-conventions?view=vs-2019. Accessed: 2020-10-26.

[13] Android abis | android ndk | android developers. https://developer.android.com/ndk/guides/abis. Accessed: 2020-10-26.

