Shark ISA Specification

Version 0.1

Author: LowByteFox

October 2, 2025

Contents

1 Preamble

2 Introduction

Shark (a short for "**S**imple **H**ardware **Ar**chitecture") is a simple architecture designed by a C programmer for hobbyist and learning purposes.

This document describes the Instruction Set Architecture (ISA) for **Shark**. It covers registers, memory model, instruction formats, and the behavior of each instruction.

The goal is to stay simple, while allowing flexibility for users and staying consistent with encoding. The ISA uses fixed 32-bit encoding per instruction, first 8 most significant bits (31-23) contains opcode header, the first 6 most significant bits are used for opcode while the rest 2 bits are used for opcode "mode bits", which alter the behavior of an instruction decoding and execution.

Most instructions operate on unsigned values, those that operate on signed values have it stated. Each encoded immediate is therefor considered unsigned, those instructions that rely on signed values have immediates encoded using Two's Complement representation.

This ISA supports integers of up most to 32-bits, but support for 64-bit values may become an extension.

3 Execution Model

- Word size: 32-bit
- Endianness: Little-endian
- Addressing: Flat memory model
- Calling convention: [Describe briefly]

4 Registers

Name	Description	Width
R0–R13	General purpose registers	32-bit
BP	Base stack pointer	32-bit
SP	Stack pointer	32-bit

5 Memory Model

- Address space: 4 GiB (0x00000000–0xFFFFFFFF)
- Alignment: word-aligned (4 bytes)
- Supported data types: byte (8-bit), half (16-bit), word (32-bit)

6 Instruction Set

Each instruction is defined with its mnemonic, encoding, description, and examples.

6.1 Arithmetic Instructions

Every instruction (unless stated) is handling integers and treating immediates as unsigned, these instructions support 3 modes:

Bits	Mode	Description
00	R-R	Performs operation using 2 registers
01	R-I	Performs operation using register and immediate
10	I-R	Performs operation using immediate and register
11	I-I	Performs operation using 2 immediates

Encoding is as follows:

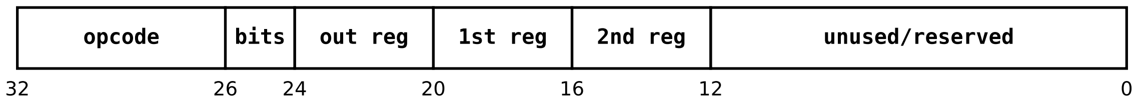

Figure 1: Encoding for R-R

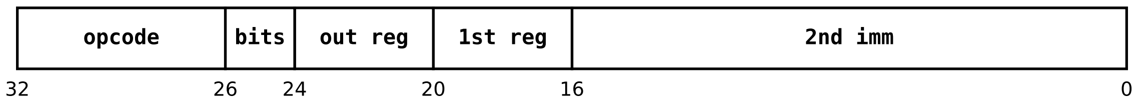

Figure 2: Encoding for R-I

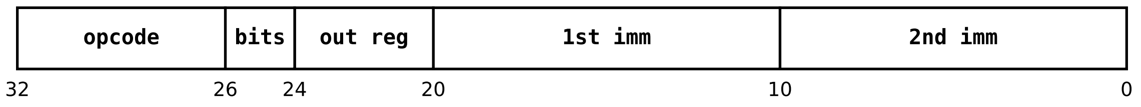

Figure 3: Encoding for I-I

Note:
I-R is encoded the same way as R-I is, however the order of operands is swapped

6.1.1 add

Opcode: 1
Syntax: add DST, SRC1, SRC2
Description:
Adds inputs SRC1 and SRC2, storing the result in DST.
Example:

```
add R0, R1, R2   ; R0 = R1 + R2
add R0, R1, #4   ; R0 = R1 + 4
add R0, #3, #4   ; R0 = 3 + 4
```

6.1.2 sub

Opcode: 2
Syntax: sub DST, SRC1, SRC2
Description:
Subtracts SRC2 from SRC1, storing the result in DST.
Example:

```
sub R0, R1, R2   ; R0 = R1 - R2
sub R0, R1, #4   ; R0 = R1 - 4
sub R0, #7, R2   ; R0 = 7 - R2
sub R0, #7, #4   ; R0 = 7 - 4
```

6.1.3 mul

Opcode: 3
Syntax: mul DST, SRC1, SRC2
Description:
Multiplies SRC1 with SRC2, storing the result in DST.
Example:

```
mul R0, R1, R2   ; R0 = R1 * R2
mul R0, R1, #4   ; R0 = R1 * 4
mul R0, #7, #4   ; R0 = 7 * 4
```

6.1.4 smul

Opcode: 4
Syntax: smul DST, SRC1, SRC2
Description:
Signed multiplication of SRC1 with SRC2, storing the result in DST.
Example:

```
smul R0, R1, R2   ; R0 = R1 * R2
smul R0, R1, #4   ; R0 = R1 * 4
smul R0, #7, #4   ; R0 = 7 * 4
```

6.1.5 div

Opcode: 5
Syntax: div DST, SRC1, SRC2
Description:
Divides SRC1 with SRC2, storing the result in DST.
Example:

```
div R0, R1, R2   ; R0 = R1 / R2
div R0, R1, #4   ; R0 = R1 / 4
div R0, #7, R2   ; R0 = 7 / R2
div R0, #7, #4   ; R0 = 7 / 4
```

6.1.6 sdiv

Opcode: 6
Syntax: sdiv DST, SRC1, SRC2
Description:
Signed division of SRC1 with SRC2, storing the result in DST.
Example:

```
sdiv R0, R1, R2   ; R0 = R1 / R2
sdiv R0, R1, #4   ; R0 = R1 / 4
sdiv R0, #7, R2   ; R0 = 7 / R2
sdiv R0, #7, #4   ; R0 = 7 / 4
```

6.1.7 rem

Opcode: 7
Syntax: rem DST, SRC1, SRC2
Description:
Divides SRC1 with SRC2, storing the remainder in DST.
Example:

```
rem R0, R1, R2   ; R0 = R1 % R2
rem R0, R1, #4   ; R0 = R1 % 4
rem R0, #7, R2   ; R0 = 7 % R2
rem R0, #7, #4   ; R0 = 7 % 4
```

6.1.8 srem

Opcode: 8
Syntax: srem DST, SRC1, SRC2
Description:
Signed division of SRC1 with SRC2, storing the remainder in DST.
Example:

```
srem R0, R1, R2   ; R0 = R1 % R2
srem R0, R1, #4   ; R0 = R1 % 4
srem R0, #7, R2   ; R0 = 7 % R2
srem R0, #7, #4   ; R0 = 7 % 4
```

6.1.9 and

Opcode: 9
Syntax: and DST, SRC1, SRC2
Description:
Bitwise and of SRC1 and SRC2, storing the result in DST.
Example:

```
and R0, R1, R2   ; R0 = R1 & R2
and R0, R1, #4   ; R0 = R1 & 4
and R0, #7, #4   ; R0 = 7 & 4
```

6.1.10 or

Opcode: 10
Syntax: or DST, SRC1, SRC2
Description:
Bitwise or of SRC1 and SRC2, storing the result in DST.
Example:

```
or R0, R1, R2   ; R0 = R1 | R2
or R0, R1, #4   ; R0 = R1 | 4
or R0, #7, #4   ; R0 = 7 | 4
```

6.1.11 xor

Opcode: 11
Syntax: xor DST, SRC1, SRC2
Description:
Bitwise xor of SRC1 and SRC2, storing the result in DST.
Example:

```
xor R0, R1, R2   ; R0 = R1 ^ R2
xor R0, R1, #4   ; R0 = R1 ^ 4
xor R0, #7, #4   ; R0 = 7 ^ 4
```

7 Encoding Specification

Bits	Meaning
15–12	Destination register (Rd)
11–8	Source register (Rs)
7–4	Source register (Rt)
3–0	Opcode

8 ABI Conventions

- R0–R3: argument registers
- R4–R7: callee-saved registers
- SP: grows downward
- Return value: R0

9 Examples

9.1 Hello World Loop

```
foo:                          ; emulator pushne ret addr a BP
  mov BP, SP                  ; SP -> BP
  sub SP, SP, #4              ; SP - 4 -> SP
  add R0, #3, #4              ; 3 + 4 -> R0
  str #[BP - 4], R0           ; R0 -> BP - 4
  ret                         ; BP -> SP, emulator popne BP a ret addr
```