Bytes in elm

Diving low level in a high level language

Warming Up

*** **Vocabulary Basis**

Common Types

Endianness

Example Usage

Manipulating Bytes in Elm

Vocabulary Basis

Base-ten = “decimal”	42	**4** * 10 + **2** * 1
Base-two = “binary”	101010 0b101010	**1** * 32 + **0** * 16 + **1** * 8 + **0** * 4 + **1** * 2
Base-sixteen = “hexa”	2A 0x2A	**2** * 16 + **10** * 1

→ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

A byte = 8 bits 42 → 0b00101010

Vocabulary Basis

Base-64 → tricky 2^6 = 64 so we encode blocs of 6 bits

“ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/”

Source	**Text (ASCII)**	**M**							**a**								**n**								
	Octets	77 (0x4d)							97 (0x61)								110 (0x6e)								
Bits		0	1	0	0	1	1	0	1	0	1	1	0	0	0	0	1	0	1	1	0	1	1	1	0
Base64 encoded	**Sextets**	19							22								5								46
	Character	**T**							**W**								**F**								**u**
	Octets	84 (0x54)							87 (0x57)								70 (0x46)								117 (0x75)

Source wikipedia : https://en.wikipedia.org/wiki/Base64

Warming Up

Vocabulary Basis

*** **Common Types**

Endianness

Example Usage

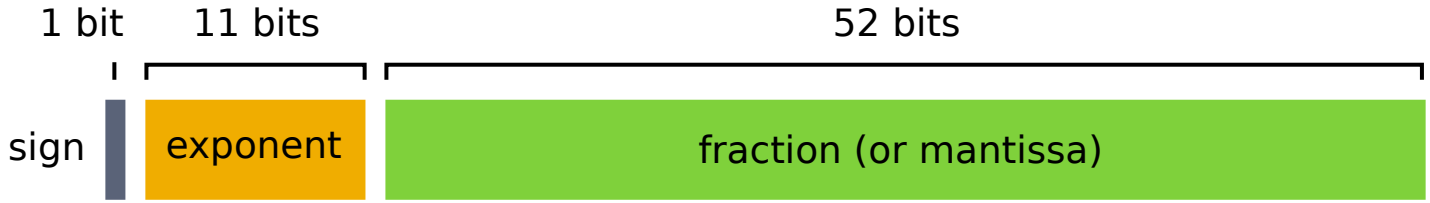
Manipulating Bytes in Elm

Common Types – integers

Boolean	bool	0-1
Unsigned 8-bits integer	u8	0 → 255
Unsigned 16-bits integer	u16	0 → 65535
Unsigned 32-bits integer	u32	0 → 4294967295
Unsigned 64-bits integer	u64	0 → 18446744073709551615
Signed 8-bits integer	i8	-128 → +127
…		
Signed 64-bits integer	i64	a lot ;)

Common Types – floating point numbers

IEEE standard 754 for 64 bits floating point number: (-1)^S * F * 2^E

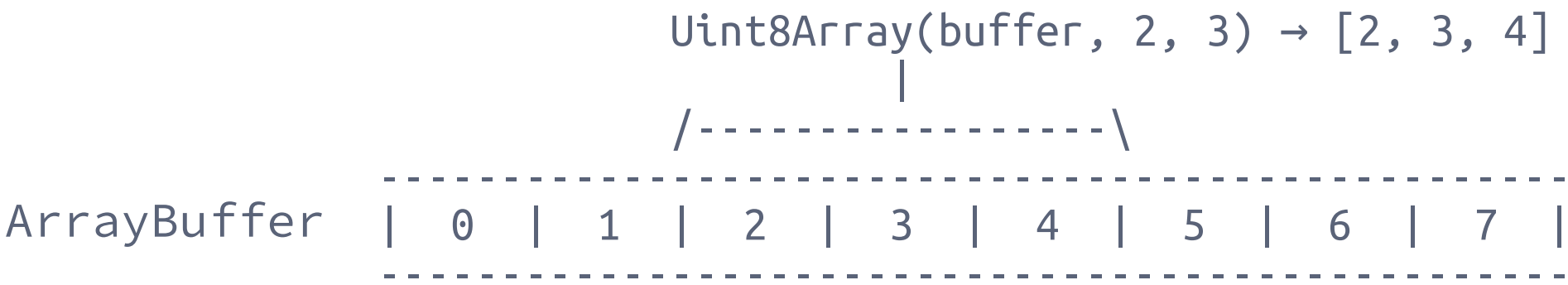


In JavaScript, there is only 1 type: **number*** (backed by f64).
Integer operations are correct until 2^53 – 1.

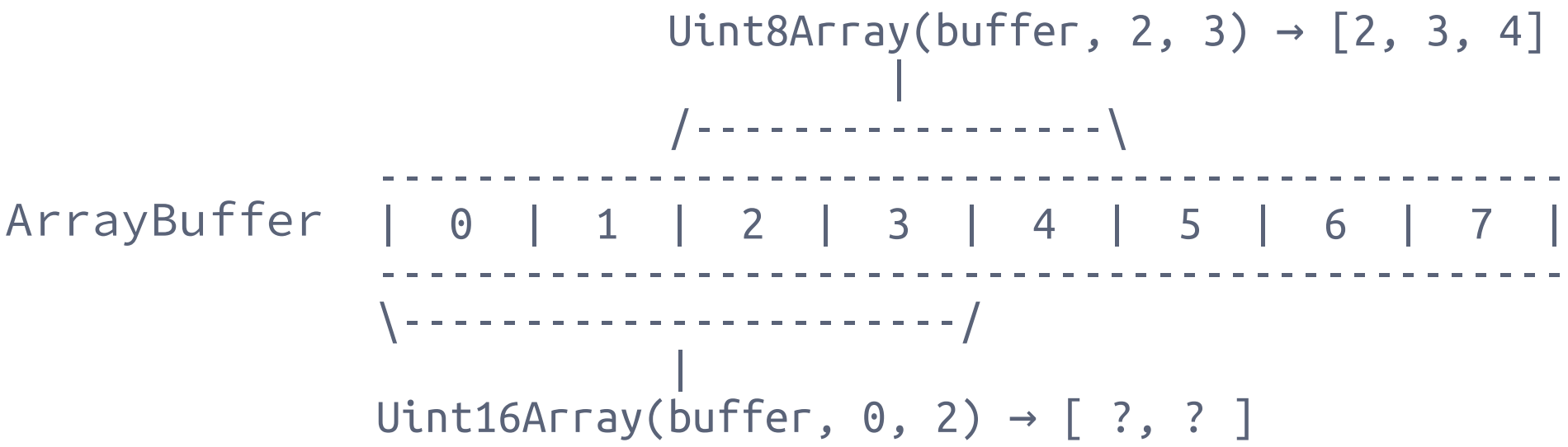
* except for “typed arrays”

Common Types – typed arrays

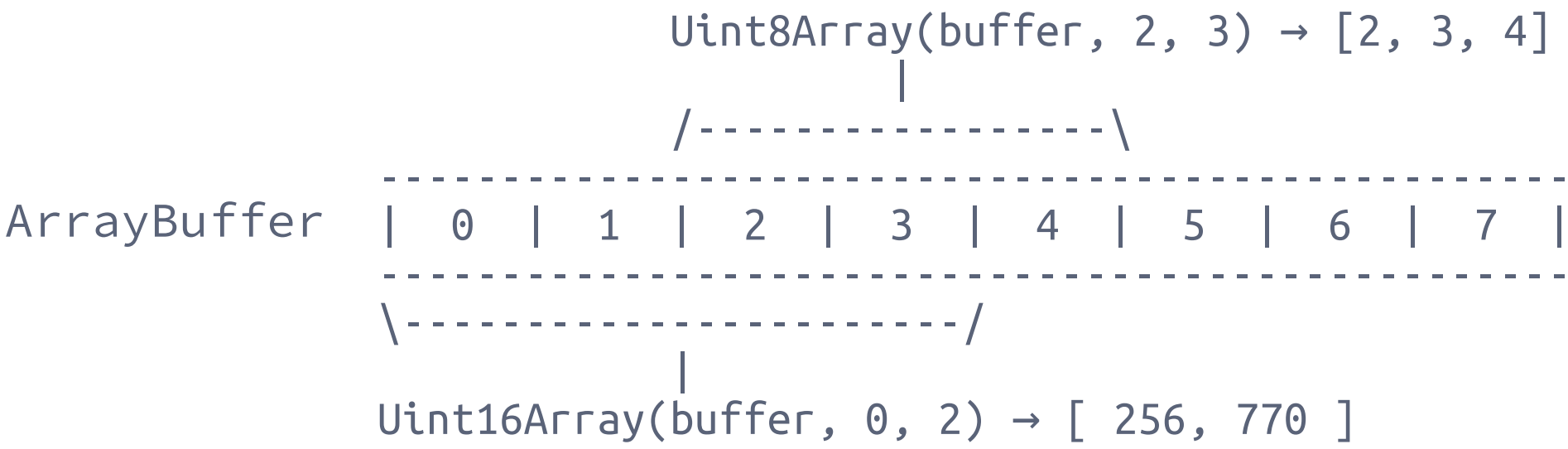
Common Types – typed arrays



Common Types – typed arrays



Common Types – typed arrays



Because of “endianness”

Warming Up

Vocabulary Basis

Common Types

*** **Endianness**

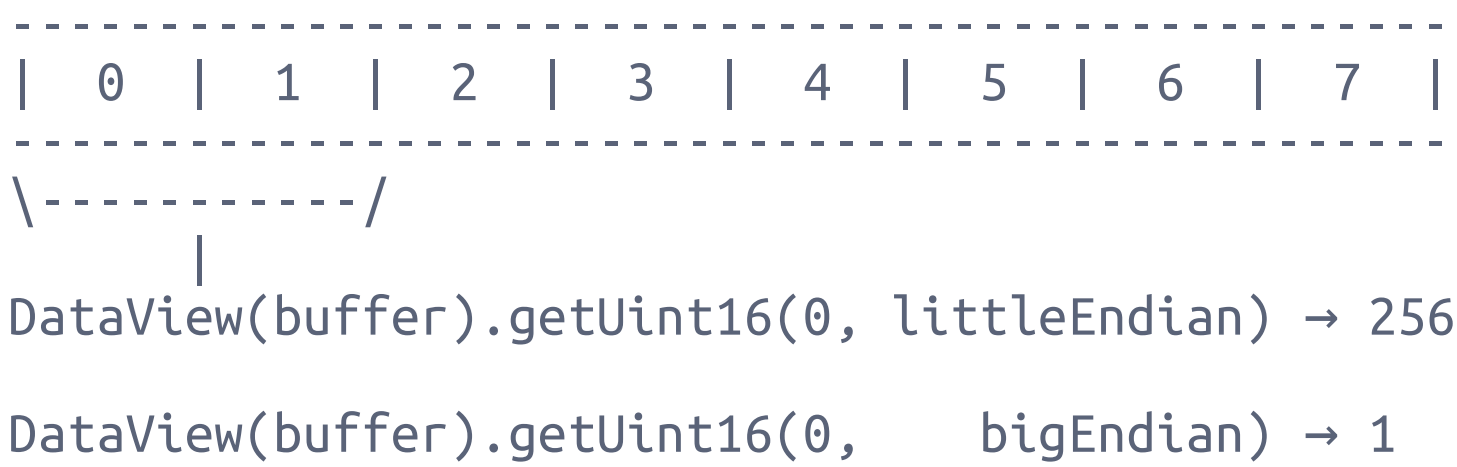
Example Usage

Manipulating Bytes in Elm

Endianness

Big-endian: default **network** byte ordering. Most significant byte first.

Little-endian: default on **most computers** today. Least significant byte first.



Warming Up

Vocabulary Basis

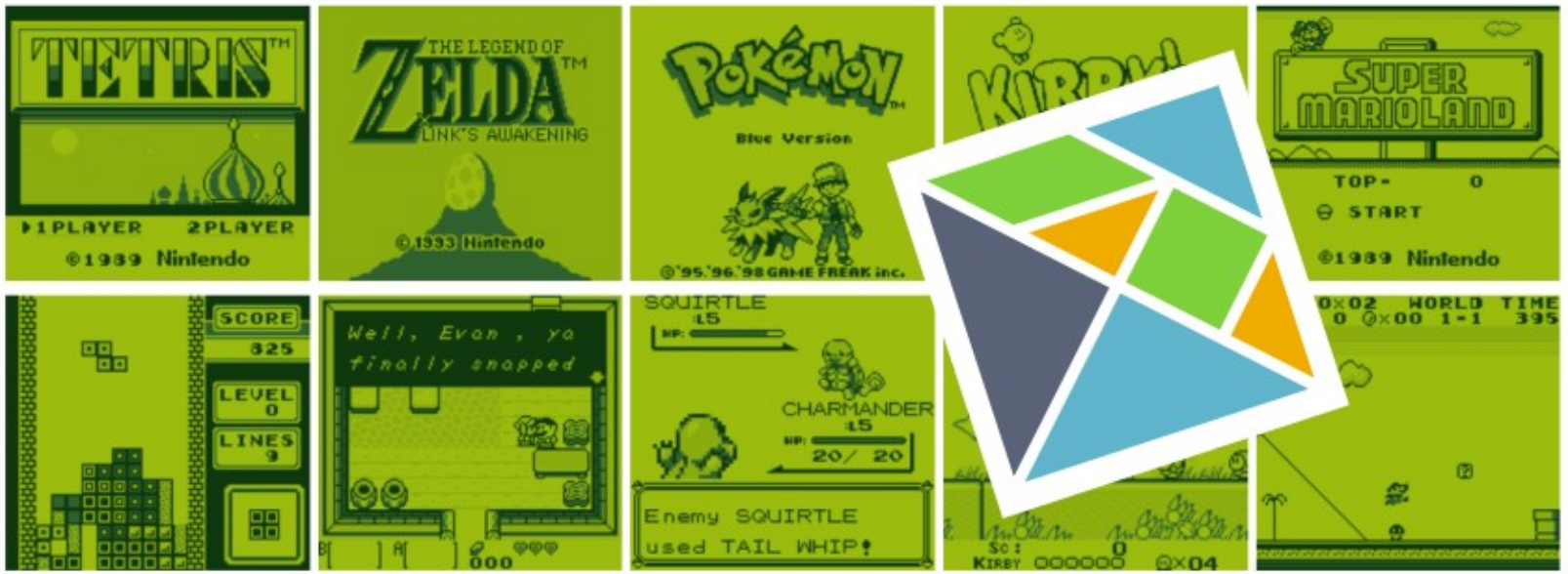
Common Types

Endianness

*** **Example Usage**

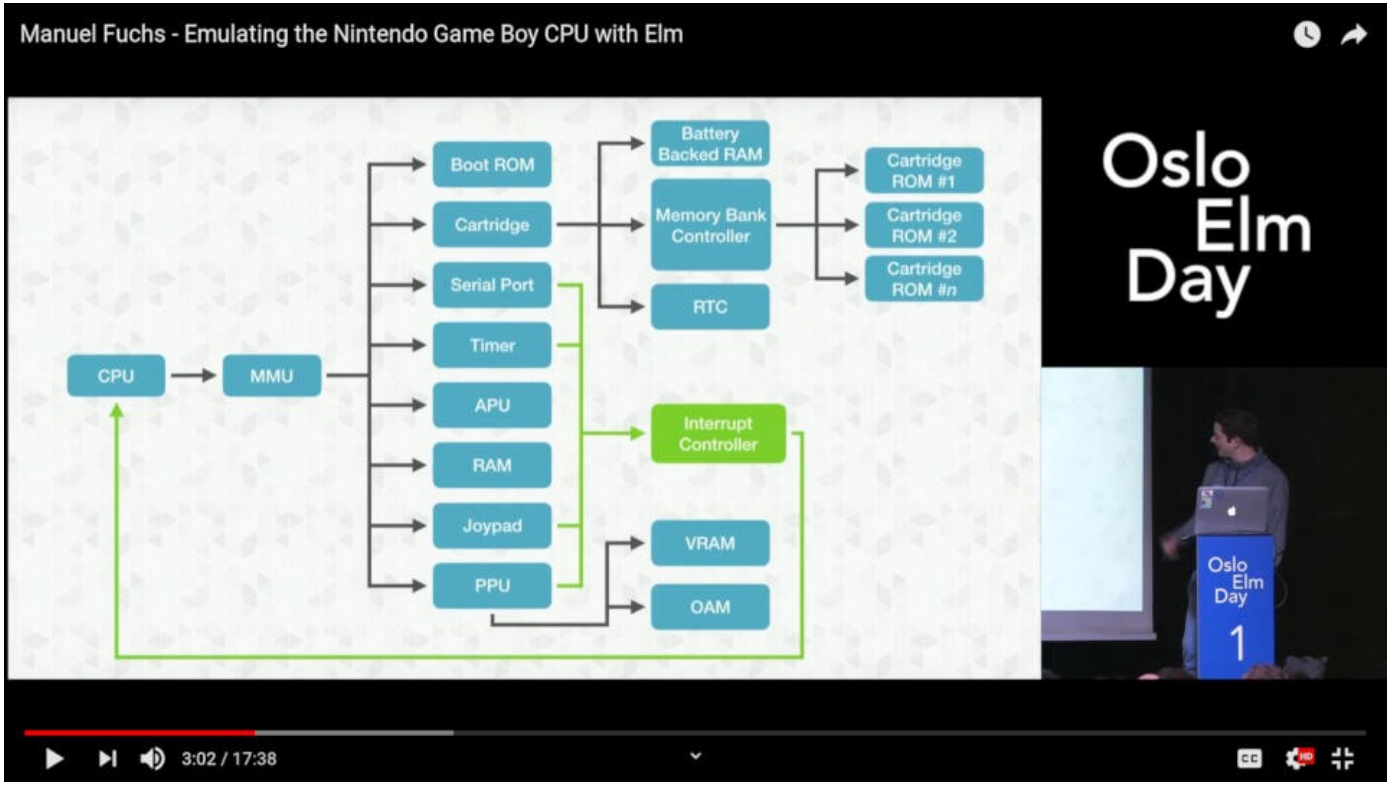
Manipulating Bytes in Elm

Example use case: Gameboy emulator



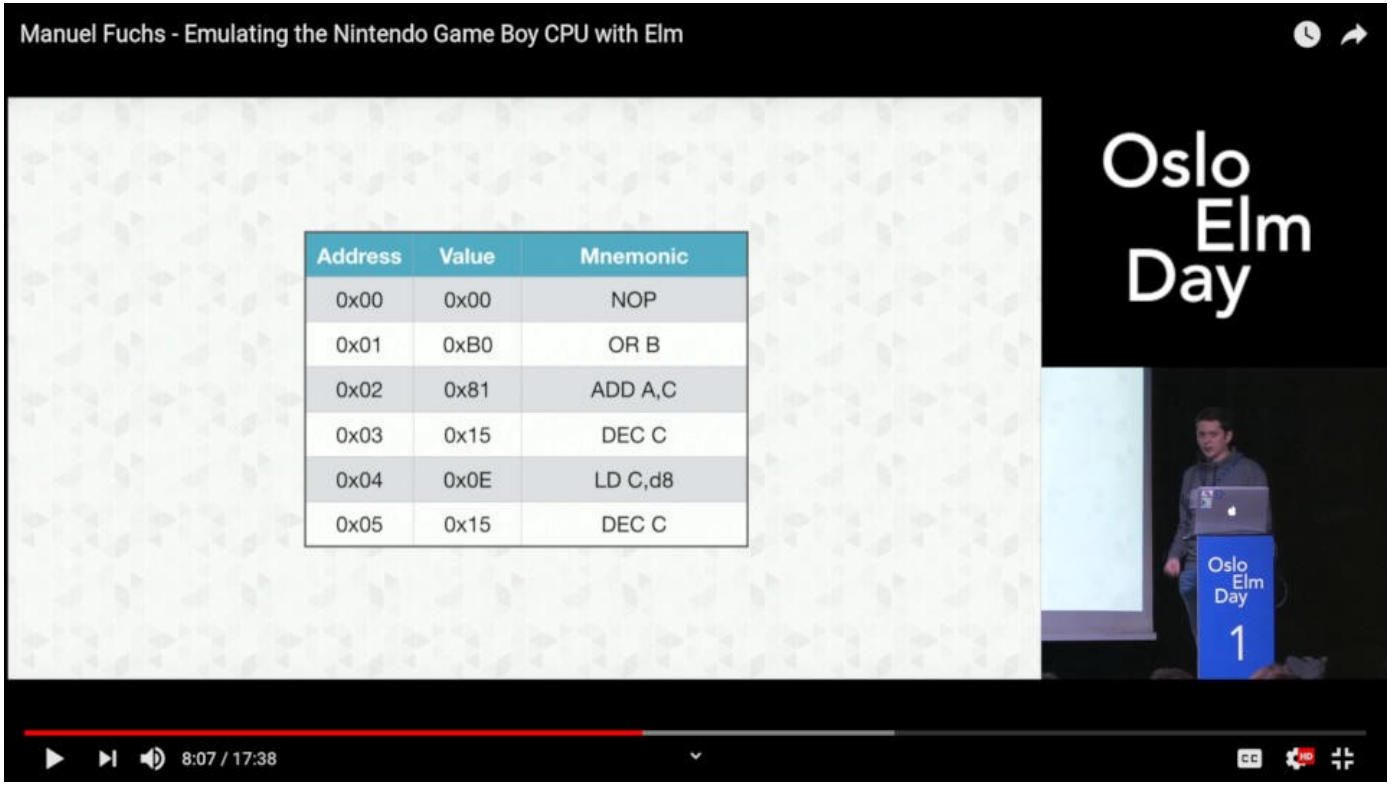
Elmboy: https://github.com/Malax/elmboy

Example use case: Gameboy emulator



https://youtu.be/vl30OvU3QW0?t=169

Example use case: Gameboy emulator



https://youtu.be/vl30OvU3QW0?t=169

Example use case: Gameboy emulator

```
findInstructionHandler : Word8 -> GameBoy -> GameBoy
findInstructionHandler opcode =
    case Word8.toInt opcode of
        0x00 ->
            Instructions.nop

        0x01 ->
            Instructions.ldBCD16

        0x02 ->
            Instructions.ldIndirectBCA
        -- Many more...
        _ ->
            Instructions.nop
```

https://youtu.be/vl30OvU3QW0?t=169

Warming Up

Manipulating Bytes in Elm

*** **Bitwise Operations**

Basics of elm/bytes

Files and Bytes

Http and Bytes

Bitwise Operations

```
Bitwise.and : Int → Int → Int

    Bitwise.and 0b01 0b11 == 0b01

Bitwise.or : Int → Int → Int

    Bitwise.or 0b01 0b10 == 0b11

Bitwise.xor : Int → Int → Int

    Bitwise.xor 0b01 0b11 == 0b10

Bitwise.complement : Int → Int

    Bitwise.complement 0 == -1
```

*

I'm writing binary numbers as "0b01" though it is not valid elm syntax.

We can however write hex numbers like so "0x1"

Bitwise Operations

```
Bitwise.shiftLeftBy : Int → Int → Int

    shiftLeftBy 1 0b0010 == 0b0100
```

= multiply by two

```
Bitwise.shiftRightBy : Int → Int → Int

    shiftRightBy 1 0b1000 == 0b1100
    shiftRightBy 1 0b0100 == 0b0010
```

filling with highest bit
= division by two

```
Bitwise.shiftRightZBy : Int → Int → Int

   shiftRightZBy 1 0b1000 == 0b0100
```

filling with 0

Warming Up

Manipulating Bytes in Elm

- Bitwise Operations
- *** **Basics of elm/bytes**
- Files and Bytes
- Http and Bytes

Basics of elm/bytes

https://package.elm-lang.org/packages/elm/bytes/latest/

```
decode : Decoder a -> Bytes -> Maybe a
```

Turn a sequence of bytes into a nice Elm value.

```
-- decode (unsignedInt16 BE) <0007> == Just 7
-- decode (unsignedInt16 LE) <0700> == Just 7
-- decode (unsignedInt16 BE) <0700> == Just 1792
-- decode (unsignedInt32 BE) <0700> == Nothing
```

The `Decoder` specifies exactly how this should happen. This process may fail if the sequence of bytes is corrupted or unexpected somehow. The examples above show a case where there are not enough bytes.

Basics of elm/bytes

https://package.elm-lang.org/packages/elm/bytes/latest/

```
encode : Encoder -> Bytes
```

Turn an `Encoder` into `Bytes`.

```
encode (unsignedInt8     7) -- <07>
encode (unsignedInt16 BE 7) -- <0007>
encode (unsignedInt16 LE 7) -- <0700>
```

The `encode` function is designed to minimize allocation. It figures out the exact width necessary to fit everything in `Bytes` and then generate that value directly. This is valuable when you are encoding more elaborate data:

Warming Up

Manipulating Bytes in Elm

- Bitwise Operations
- Basics of elm/bytes
- *** **Files and Bytes**
- Http and Bytes

Files and Bytes

https://package.elm-lang.org/packages/elm/file/latest/

```
File.toBytes : File → Task x Bytes
```

```
File.Download.bytes : String → String → Bytes → Cmd msg
```

```
    Download.bytes "frog.png" "image/png" bytes
```

Warming Up

Manipulating Bytes in Elm

- Bitwise Operations
- Basics of elm/bytes
- Files and Bytes
- *** **Http and Bytes**

Http and Bytes

https://package.elm-lang.org/packages/elm/http/latest/

```
Http.bytesBody : String → Bytes → Body

    Http.bytesBody "application/zip" bytes

Http.bytesPart : String -> String -> Bytes -> Part

    Http.bytesPart "photo" "image/png" bytes

-- Custom requests

Http.expectBytes : (Result Error a -> msg) -> Decoder a -> Expect msg

Http.expectBytesResponse :
  (Result x a -> msg) -> (Response Bytes -> Result x a) -> Expect msg

-- Custom tasks

Http.bytesResolver : (Response Bytes -> Result x a) -> Resolver x a
```

Questions ?