Simple Z80 Emulator

Roman Valiušenko
roman.valiusenko@gmail.com

Abstract

This is a simple program that emulates Z80 microprocessor and hardware interaction such as keyboard input and display output as in ZX Spectrum. The program is written in `F#` programming language. The main objective was to run ZX Spectrum operating system Sinclair BASIC. The undocumented instructions and sound routines were not implemented.

Contents

1 About ZX Spectrum

Sinclair Research Ltd was a British electronics company founded by Sir Clive Sinclair in Cambridge. The company was incorporated in 1973, and in 1980 presented a home computer ZX80, which cost £99.95. Two years later, on 23 April 1982, the legendary ZX Spectrum was launched. It was priced at £125 for the 16KB version, and £175 for the 48KB version[21]. (Later these prices were reduced to £99 and £129 respectively.)

ZX Spectrum was an 8-bit personal home computer that used Zilog Z80 CPU running at 3.5MHZ. It ran Sinclair BASIC operating system, which was a dialect of the BASIC programming language. The operating system was developed by Nine Tiles Networks Ltd (by Steve Vickers[20]).

2 Program layout

We will keep the structure of the program very simple. We will have only two files of the program: One for the types that will be necessary to emulate ZX Spectrum, and the other file that will act a launcher.

All types will be located in `sz80em.fs` file. Everything is going to be in `FSpectrum` namespace.

1a ⟨** 1a*⟩≡
⟨*sz80em.fs* 1b⟩
⟨*sz80emr.fs* 2a⟩

1b ⟨*sz80em.fs* 1b⟩≡ (1a)
```
namespace FSpectrum
⟨References 2b⟩
⟨Types 2c⟩
```

The program launcher is very simple:

2a ⟨*sz80emr.fs* 2a⟩≡ (1a)
```
namespace FSpectrum
open System
open System.Windows.Forms

module Spectrum =
   [<EntryPoint>]
   ⟨Application execution loop 85⟩
```

We are going to need quite a few references, so let's define them straight away:

2b ⟨*References* 2b⟩≡ (1b)
```
open System
open System.IO
open System.Drawing
open System.Drawing.Imaging
open Microsoft.FSharp.NativeInterop
open System.Windows.Forms
open System.Threading
```

IO is needed to read ZX Spectrum ROM image and load it into memory. Drawing is required to emulate screen output. Forms are necessary for the GUI. NativeInterop is required to do efficient screen drawing routines.

3 Keyboard Input

In ZX Spectrum there's only one I/O port, addressed by two bytes, which can be read or written to. In this implementation we will only implement reading from the port, so let's define an abstract port:

2c ⟨*Types* 2c⟩≡ (1b) 3a▷

```
type IPort =
    abstract read: uint16 -> byte
```

That is we can read from port addressed by two bytes, and get a byte as a result. We will need this to implement keyboard routines.

When a program running on ZX Spectrum wants to read keyboard input, it reads port 0xFE. A zero in highest eight address lines selects a row of five keys as defined in the table 1[8].

Row	High Address Lines	Port	Bit 0	Bit 1	Bit 2	Bit 3	Bit 4
0	0x11111110	0xFEFE	SHIFT	Z	X	C	V
1	0x11111101	0xFDFE	A	S	D	F	G
2	0x11111011	0xFBFE	Q	W	E	R	T
3	0x11110111	0xF7FE	1	2	3	4	5
4	0x11101111	0xEFFE	0	9	8	7	6
5	0x11011111	0xDFFE	P	O	I	U	Y
6	0x10111111	0xBFFE	ENTER	L	K	J	H
7	0x01111111	0x7FFE	SPACE	SYM SHIFT	M	N	B

Table 1: Port addresses

The byte as result of port 0xFE read will have zero in one of the five lowest bits if a corresponding key is pressed.

We will implement a physical keyboard device as a class that reacts to keyboard key presses of our GUI application and sets corresponding bits for each of 0xFEFE—0x7FFE ports of which we have 8 in total. Then, programs running on our ZX Spectrum will be able to read ports 0XFEFE—0x7FFE and get these bytes as results. The result bytes will have bits set appropriately depending on keys currently being pressed.

First, let's define an auxiliary type[1] KeyCode as a record:

3a ⟨*Types* 2c⟩+≡ (1b) ◁2c 3b▷

```
type KeyCode = { row: int; bit: uint8; }
```

[1]In this document words "type" and "class" are used interchangeably.

Instances of this type will tell us which row this key correspond to (see Table 1 on page 3), and also the corresponding bit, set as a number, e.g. [0b00000001]. For example if we wanted to define key `A` we would have { row = 1; bit = 0x01uy }.

Before we do the mapping of the keys, let's define an interface of our keyboard as something that has two methods, which are called when a key is pressed, and when a key is released:

3b ⟨*Types* 2c⟩+≡ (1b) ◁3a 3c▷
```
type public IKeyboard =
    abstract KeyPressed: Keys -> unit
    abstract KeyReleased: Keys -> unit
```

Keys is the type that defines the keyboard key in `F#`.

Our keyboard will react to messages, so let's define keyboard message as a discriminated union:

3c ⟨*Types* 2c⟩+≡ (1b) ◁3b 4▷
```
type KeyboardMsg =
    | KeyPressed of Keys
    | KeyReleased of Keys
    | ReadKeys of uint16 * AsyncReplyChannel<byte>
```

Now we are good to define the actual keyboard class, which implements both `IKeyboard` and `IPort` interfaces:

4 ⟨*Types* 2c⟩+≡ (1b) ◁3c 9a▷
```
type Keyboard() = class
    let ports = Array.create 8 0xFFuy
    let keyMap =
      ⟨Key map initialization 5⟩

    let processor =
      ⟨KeyboardMsg event processing logic 6a⟩

    interface IKeyboard with
        member x.KeyPressed e = processor.Post(KeyPressed(e))
        member x.KeyReleased e = processor.Post(KeyReleased(e))

    interface IPort with
        member x.read port = processor.PostAndReply(
                                 (fun reply -> ReadKeys(port, reply)),
                                    timeout = 200)
end
```

The array `ports` will maintain results of corresponding eight keyboard ports.
Using Table 1 on page 3, we can now define key mappings as follows:

5 ⟨*Key map initialization* 5⟩≡ (4)

```
Map.empty.
    Add(Keys.ShiftKey, [ { row = 0; bit = 0x01uy } ]).
    Add(Keys.Z, [ { row = 0; bit = 0x02uy } ]).
    Add(Keys.X, [ { row = 0; bit = 0x04uy } ]).
    Add(Keys.C, [ { row = 0; bit = 0x08uy } ]).
    Add(Keys.V, [ { row = 0; bit = 0x10uy } ]).
    Add(Keys.A, [ { row = 1; bit = 0x01uy } ]).
    Add(Keys.S, [ { row = 1; bit = 0x02uy } ]).
    Add(Keys.D, [ { row = 1; bit = 0x04uy } ]).
    Add(Keys.F, [ { row = 1; bit = 0x08uy } ]).
    Add(Keys.G, [ { row = 1; bit = 0x10uy } ]).
    Add(Keys.Q, [ { row = 2; bit = 0x01uy } ]).
    Add(Keys.W, [ { row = 2; bit = 0x02uy } ]).
    Add(Keys.E, [ { row = 2; bit = 0x04uy } ]).
    Add(Keys.R, [ { row = 2; bit = 0x08uy } ]).
    Add(Keys.T, [ { row = 2; bit = 0x10uy } ]).
    Add(Keys.D1, [ { row = 3; bit = 0x01uy } ]).
    Add(Keys.D2, [ { row = 3; bit = 0x02uy } ]).
    Add(Keys.D3, [ { row = 3; bit = 0x04uy } ]).
    Add(Keys.D4, [ { row = 3; bit = 0x08uy } ]).
    Add(Keys.D5, [ { row = 3; bit = 0x10uy } ]).
    Add(Keys.D0, [ { row = 4; bit = 0x01uy } ]).
    Add(Keys.D9, [ { row = 4; bit = 0x02uy } ]).
    Add(Keys.D8, [ { row = 4; bit = 0x04uy } ]).
    Add(Keys.D7, [ { row = 4; bit = 0x08uy } ]).
    Add(Keys.D6, [ { row = 4; bit = 0x10uy } ]).
    Add(Keys.P, [ { row = 5; bit = 0x01uy } ]).
    Add(Keys.O, [ { row = 5; bit = 0x02uy } ]).
    Add(Keys.I, [ { row = 5; bit = 0x04uy } ]).
    Add(Keys.U, [ { row = 5; bit = 0x08uy } ]).
    Add(Keys.Y, [ { row = 5; bit = 0x10uy } ]).
    Add(Keys.Enter, [ { row = 6; bit = 0x01uy} ]).
    Add(Keys.L, [ { row = 6; bit = 0x02uy } ]).
    Add(Keys.K, [ { row = 6; bit = 0x04uy } ]).
    Add(Keys.J, [ { row = 6; bit = 0x08uy } ]).
    Add(Keys.H, [ { row = 6; bit = 0x10uy } ]).
    Add(Keys.Space, [ { row = 7; bit = 0x01uy } ]).
    Add(Keys.ControlKey, [ { row = 7; bit = 0x02uy } ]).
    Add(Keys.M, [ { row = 7; bit = 0x04uy } ]).
    Add(Keys.N, [ { row = 7; bit = 0x08uy } ]).
    Add(Keys.B, [ { row = 7; bit = 0x10uy } ]).
    Add(Keys.Escape, [ { row = 0; bit = 0x01uy };
                       { row = 7; bit = 0x01uy } ]).
    Add(Keys.Back, [ { row = 0; bit = 0x01uy };
                     { row = 4; bit = 0x01uy } ]).
    Add(Keys.Capital, [ { row = 0; bit = 0x01uy };
                        { row = 3; bit = 0x02uy } ]).
    Add(Keys.Oemplus, [ { row = 7; bit = 0x02uy };
                        { row = 6; bit = 0x02uy } ]) // ctrl + L
```

Note that `keyMap` is a map from `Keys` to an array of `KeyCode`. Some keys, such as `Escape`, `Back` etc. are represented as a combination of `KeyCode` instances.

Now let's write keyboard processor logic. It's going to be a pretty straightforward asynchronous events processor, a pattern matching on message:

6a ⟨*KeyboardMsg event processing logic* 6a⟩≡ (4)

```
MailboxProcessor.Start(fun inbox ->
    let rec nextMsg =
        async {
            let! msg = inbox.Receive()
            match msg with
            | KeyPressed key ->
                ⟨Key pressed 6b⟩
            | KeyReleased key ->
                ⟨Key released 6c⟩
            | ReadKeys (port, reply) ->
                ⟨Read keys 7⟩
        }
    nextMsg
    )
```

When a key is pressed, we need to lookup our map for the key, get `KeyCodes`, and set corresponding bits in the bytes in the `ports` array:

6b ⟨*Key pressed* 6b⟩≡ (6a)

```
if keyMap.ContainsKey(key) then
    for k in keyMap.Item(key) do
       ports.[k.row] <- ports.[k.row] &&& (k.bit ^^^ 0xFFuy)
return! nextMsg
```

Similarly, when a key is released, we do the reverse:

6c ⟨*Key released* 6c⟩≡ (6a)

```
if keyMap.ContainsKey(key) then
    for k in keyMap.Item(key) do
       ports.[k.row] <- ports.[k.row] ||| k.bit
return! nextMsg
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	Y_7	Y_6	Y_2	Y_1	Y_0	Y_5	Y_4	Y_3	X_4	X_3	X_2	X_1	X_0

Table 2: Row/Column Address Scheme

When reading keyboard ports we need to identify which rows are requested to be read. To do that we shift the `port` value by eight bits to the right, because higher eight bits actually identify rows to read, i.e. zero bit in one of eight positions identifies a row (see the table above). Then we create a sequence of shifts to the right by one bit, producing tuples which have index as its first component, and shifted value as its second component. We take eight of such tuples. Next, we iterate on each of those tuples and identify which of those have 0 as their 0th bit in the second tuple's value. If it is 0, we know that row `i` is requested. Finally, we take that row as `ports.[i]` and `AND` all such rows. This is captured in the following code chunk:

7 ⟨*Read keys* 7⟩≡ (6a)
```
let result =
    (byte (port >>> 8))
    |> Seq.unfold (fun i -> Some(i, i >>> 1))
    |> Seq.take 8
    |> Seq.mapi (fun i port -> if port &&& 0x01uy = 0uy then ports.[i] else 0uy)
    |> Seq.filter(fun i -> i > 0uy)
    |> Seq.fold (fun r v -> r &&& v) 0xFFuy
reply.Reply(result)
return! nextMsg
```

7	6	5	4	3	2	1	0
F	B	P_2	P_1	P_0	I_2	I_1	I_0

Table 3: Colour Attribute Scheme

Forward pipe operator `|>` is used to chain operations on sequences above. Its definition is very simple:
`let (|>) x f = f x`,
and its type is
`val (|>) :  'T -> ('T -> 'U) -> 'U`.

4 Display Output

The Spectrum has the memory size of `0xFFFF` bytes. 16K ROM in the lowest part of the address space, and 48K of RAM which occupies the rest of the address space.

The screen memory map starts at `0x4000` and is of 6144 bytes long. 768 bytes that follow right after that, i.e. starting at address `0x6500`, are colour attributes.

The bitmap is made of 192 lines, each of 32 bytes length. Each byte represents 8 pixels, so 32 bytes account for 256 pixels. Therefore, we have a resolution of 256 by 192 pixels.

The interesting thing is that how column and row are addressed. This is encoded using the scheme shown in Table 2. (Also see [5] and [12].)

Basically column and row cal be obtained like this:

```
row = ((address &&& 0b0001100000000000) >>> 8) ||| ((address &&& 0b11100000) >>> 5)
col = (address &&& 0b11111)
```

Colour attribute byte is encoded as shown in Table 3. (See [12].)

Here F is attribute Flash mode, which makes Paper and Ink colours to alternate periodically). B sets Brightness mode (in original ZX Spectrum was generated using the video display's maximum voltage levels for each RGB component). P_2–P_0 is Paper colour, and I_2–I_0 is Ink colour. Paper colour is background colour, so is used where corresponding pixel has 0 bit set, while [Ink colour is used where the bit is set to 1.

Colours are as follows[6]:

Colour #	Binary value	Bright 0	Bright 1	Colour Name
0	000	#000000	#000000	Black
1	001	#0000CD	#0000FF	Blue
2	010	#CD0000	#FF0000	Red
3	011	#CD00CD	#FF00FF	Magenta
4	100	#00CD00	#00FF00	Green
5	101	#00CDCD	#00FFFF	Cyan
6	110	#CDCD00	#FFFF00	Yellow
7	111	#CDCDCD	#FFFFFF	White

Table 4: Colour Table

We have 8 base colours, each of which has an additional bright mode, making 15 colours in total (Black colour doesn't have bright mode.)

Each coulour attribute is applied to 8 by 8 pixels area, which we will references as character area.

OK, before starting with display routines, let's write a helper type to work with bitmaps more efficiently.

`.NET` supports the LockBits method allowing us to manipulate a bitmap's array of pixels in an efficient way.

The implementation of the helper type is very straightforward. The type takes a bitmap as constructor's argument. Then we use `LockBits` method of the `bitmap` to obtain `BitmapData`. We set `ImageLockMode.WriteOnly` and `PixelFormat` from the `bitmap`. Then we define `setPixel` method that takes as argument x and y coordinates and an RGB tuple defining the colour we want to set at (x, y). The method obtains an address by using `data.Scan0` which returns the address of the first line and adding a shift value, calculated by multiplying y coordinate by the stride (got from `data.Stride`), and adding $x * 3$ (three bytes account for RGB components). Then, using `NativePtr` `write` and `set` methods we set corresponding RGB components.

This is captured in the following code chunk:

9a ⟨*Types* 2c⟩+≡ (1b) ◁4 9b▷

```
type DirectBitmapAccess(bitmap: Bitmap) =
     let data = bitmap.LockBits(
                      new Rectangle(0, 0, bitmap.Width, bitmap.Height),
                      ImageLockMode.WriteOnly,
                      bitmap.PixelFormat)

     let setPixel x y (r,g,b) =
        let address =
            NativePtr.add<byte> (NativePtr.ofNativeInt data.Scan0)
                        ((y * data.Stride) + (x * 3))
        NativePtr.write address b
        NativePtr.set address 1 g
        NativePtr.set address 2 r

     member this.SetPixel(x, y, color: Color) =
        setPixel x y (color.R, color.G, color.B)

     interface IDisposable with
        member this.Dispose() =
            bitmap.UnlockBits(data)
```

Now as we have a small helper class that allows us to work with bitmaps in an efficient way, let's write the video processor type. This type will have three methods: `DrawImage`, `GetImage`, and `Flash`. So let's define an interface first:

9b ⟨*Types* 2c⟩+≡ (1b) ◁9a 10a▷

```
type IVideoMemoryProcessor =
    abstract ChangeFlash: unit
    abstract DrawImage: byte[] -> unit
    abstract GetImage: Image
```

And also let's define `VideoMemoryProcessorMsg` type as discriminated union:

10a ⟨*Types* 2c⟩+≡ (1b) ◁9b 10b▷

```
type VideoMemoryProcessorMsg =
    | DrawImage of byte[]
    | GetImage of AsyncReplyChannel<Bitmap>
    | Flash
```

This will be used to send messages to our video processor.

The video memory processor type goes next, which implements `IVideoMemoryProcessor` interface:

10b ⟨*Types* 2c⟩+≡ (1b) ◁10a 14a▷

```
type VideoMemoryProcessor() as this = class
    ⟨VMP Constants 11b⟩
    ⟨VMP Variables 11c⟩

    let processor =
      ⟨VideoMemoryProcessorMsg processor 12a⟩

    let COLORS =
      ⟨Colors table initialization 11a⟩

    member private x.drawFrame(memory: byte[]) =
      ⟨Frame rendering 12b⟩

    interface IVideoMemoryProcessor with
        member x.ChangeFlash = processor.Post(Flash)
        member x.DrawImage bitmap = processor.Post(DrawImage bitmap)
        member x.GetImage =
                   processor.PostAndReply(
                        (fun reply -> GetImage(reply)), timeout = 200) :> Image
end
```

First, according to Table 4 on page 8, let's define colors map:

11a ⟨*Colors table initialization* 11a⟩≡ (10b)
```
Map.empty.
    Add(0b0000, Color.FromArgb(0xFF, 0x00, 0x00, 0x00)).
    Add(0b1110, Color.FromArgb(0xFF, 0xFF, 0xFF, 0x00)).
    Add(0b0001, Color.FromArgb(0xFF, 0x00, 0x00, 0xC0)).
    Add(0b1001, Color.FromArgb(0xFF, 0x00, 0x00, 0xFF)).
    Add(0b0010, Color.FromArgb(0xFF, 0xC0, 0x00, 0x00)).
    Add(0b1010, Color.FromArgb(0xFF, 0xFF, 0x00, 0x00)).
    Add(0b0011, Color.FromArgb(0xFF, 0xC0, 0x00, 0xC0)).
    Add(0b1011, Color.FromArgb(0xFF, 0xFF, 0x00, 0xFF)).
    Add(0b0100, Color.FromArgb(0xFF, 0x00, 0xC0, 0x00)).
    Add(0b1000, Color.FromArgb(0xFF, 0x00, 0x00, 0x00)).
    Add(0b0101, Color.FromArgb(0xFF, 0x00, 0xC0, 0xC0)).
    Add(0b1101, Color.FromArgb(0xFF, 0x00, 0xFF, 0xFF)).
    Add(0b0110, Color.FromArgb(0xFF, 0xC0, 0xC0, 0x00)).
    Add(0b1100, Color.FromArgb(0xFF, 0x00, 0xFF, 0x00)).
    Add(0b0111, Color.FromArgb(0xFF, 0xC0, 0xC0, 0xC0)).
    Add(0b1111, Color.FromArgb(0xFF, 0xFF, 0xFF, 0xFF))
```

We need a few constants, so let's define them next. We define constants such as height and width of the screen, character height (and width; they are the same, as character area is a square) and screen memory size in bytes. All of them will be useful when rendering video frame:

11b ⟨*VMP Constants* 11b⟩≡ (10b)
```
let WIDTH = 256;
let HEIGHT = 192;
let CHAR_WIDTH_HEIGHT = 8;
let WIDTH_IN_CHARS = WIDTH / CHAR_WIDTH_HEIGHT;
let HEIGHT_IN_CHARS = HEIGHT / CHAR_WIDTH_HEIGHT;
let SCREEN_MEMORY_SIZE = WIDTH * HEIGHT / CHAR_WIDTH_HEIGHT;

let CONTENDED_MEMORY_START = 0x4000
```

We will also need a couple of variables. One of those will be an array of two bitmaps. We will alternate bitmaps in such a way that if `GetImage` message comes through, we return a bitmap that is fully rendered, and we render another frame into a different bitmap. Once another bitmap is fully rendered, the other will be selected for rendering, and so on. It's a kind of double-buffer technique.

So let's define an array of two bitmaps, each of `WIDTH` and `HEIGHT` size. We will need a `screen\_index` variable to alternate bitmaps. We also need another variable `flash`, that will be used to implement Flash mode. We set its value to `0b01111111`, that is `0x7F`. Then, we alternate it between `0x7F` and `0xFF` upon each `Flash` message.

11c ⟨*VMP Variables* 11c⟩≡ (10b)
```
let screens =
  Array.init 2 (fun i -> new Bitmap(WIDTH, HEIGHT, PixelFormat.Format24bppRgb))
let mutable screen_index = 0
let mutable flash = 0x7Fuy;
```

So let's write a processor now. In here we implement the alternating bitmaps and the Flash mode. The Flash mode simply changes the value of `flash` variable from `0x7F` to `0xFF` and vice versa by toggling the 8th bit. The `flash` value will be used when rendering the actual frame later.

12a ⟨*VideoMemoryProcessorMsg processor* 12a⟩≡ (10b)

```
MailboxProcessor.Start(fun inbox ->
    let rec nextMsg =
        async {
            let! msg = inbox.Receive()
            match msg with
            | GetImage(reply) ->
                reply.Reply(screens.[(screen_index + 1) % 2])
                return! nextMsg
            | DrawImage bitmap ->
                this.drawFrame bitmap
                screen_index <- (screen_index + 1) % 2
                return! nextMsg
            | Flash ->
                flash <- if flash = 0x7Fuy then 0xFFuy else 0x7Fuy
                return! nextMsg
        }
    nextMsg
    )
```

Now we are ready to start implementing the rendering method. The method takes as argument an array of bytes, which will be the whole ZX Spectrum memory. We will read only portion of it, that is video memory plus colour attributes.

First, we will obtain the lock on the bitmap that is currently used to render the frame. We use `using` construct here to release the lock once we exit the scope:

12b ⟨*Frame rendering* 12b⟩≡ (10b)

```
using (new DirectBitmapAccess(screens.[screen_index])) (fun lockContext ->
    ⟨Render the bitmap 12c⟩
)
```

To render the pixels we iterate on each vide memory byte, starting from address `0x4000` and continuing for `SCREEN_MEMORY_SIZE`:

12c ⟨*Render the bitmap* 12c⟩≡ (12b)

```
for i in CONTENDED_MEMORY_START..(CONTENDED_MEMORY_START + SCREEN_MEMORY_SIZE - 1) do
    ⟨Calculate row, col and line 13a⟩
    ⟨Obtain ink and paper values for this row and col 13b⟩
    ⟨Flash mode: Alternate ink and paper 13c⟩
    ⟨Render pixels 13d⟩
```

Using the encoding scheme described earlier, we can obtain column and row, and line values.

13a ⟨*Calculate row, col and line* 13a⟩≡ (12c)

```
let address = i - CONTENDED_MEMORY_START // 0x4000
let row = ((address &&& 0b0001100000000000) >>> 8) ||| ((address &&& 0b11100000) >>> 5)
let col = (address &&& 0b11111)
let line = ((address &&& 0b11100000000) >>> 8)
```

Colours attributes follow immediately after the video memory segment, so we get the attribute we want by using value of `col` and `row` calculated earlier in a straightforward way:

13b ⟨*Obtain ink and paper values for this row and col* 13b⟩≡ (12c)

```
let attribute = memory.[CONTENDED_MEMORY_START
        + SCREEN_MEMORY_SIZE + row * WIDTH_IN_CHARS + col] &&& flash
let mutable ink = int ((attribute &&& 0b111uy) ||| ((attribute &&& 0b1000000uy) >>> 3))
let mutable paper = int ((attribute &&& 0b1111000uy) >>> 3)
```

Note how we use `flash` variable to set the 8th bit of `attribute`. `ink` and `paper` attributes are calculated using the scheme described earlier.

Now if the 8th bit of the attribute is non-zero, we swap `ink` and `paper` values. This implements Flash mode.

13c ⟨*Flash mode: Alternate ink and paper* 13c⟩≡ (12c)

```
if (attribute &&& 0x80uy) <> 0uy then
   let tmp = ink
   ink <- paper
   paper <- tmp
```

Finally, we are ready to start rendering the actual image. Each byte addressed by `address` encodes 8 pixels. We initialize a `mask` variable and shift it to the right by one, allowing us to inspect each of those 8 pixels in order. If currently inspected bit is 1, we set `ink` colour, otherwise we set `paper` colour. We shift `mask` by one to the right, and move on to next bit. This is implemented in the following chunk:

13d ⟨*Render pixels* 13d⟩≡ (12c)

```
let mutable x = 0
let mutable mask = 0b10000000uy
while mask > 0uy do
   let xx = (col * CHAR_WIDTH_HEIGHT + x);
   let yy = row * CHAR_WIDTH_HEIGHT + line;
   let clr = COLORS.[if (memory.[i] &&& mask) = 0uy then paper else ink]
   lockContext.SetPixel(xx, yy, clr)
   mask <- mask >>> 1
   x <- x + 1
```

This finishes the implementation of our `VideoMemoryProcessor` type.

And the last thing for the Video Output would be the GUI component that will be displaying the image on our GUI form.

Let's define an interface first. It will have one method `draw` taking byte array as argument. This array is the whole memory of our ZX Spectrum machine:

14a ⟨*Types* 2c⟩+≡ (1b) ◁10b 14b▷

```
type IDisplay =
    abstract draw: byte[] -> uint16 -> unit
```

We implement this interface as a class inheriting from `Panel`. We pass in the implementation of `IVideoMemoryProcessor` to its constructor, because the actual rendering is done by the `VideoMemoryProcessor`, while this class only displays what's been rendered. Note how we also issue `ChangeFlash` message every 16th frame.

14b ⟨*Types* 2c⟩+≡ (1b) ◁14a 16▷

```
type Display(videoProcessor: IVideoMemoryProcessor) as this = class
    inherit Panel(Dock = DockStyle.Fill)

    do
        this.DoubleBuffered <- true
        this.Visible <- true

    override x.OnPaint e =
        e.Graphics.DrawImage(videoProcessor.GetImage, 0, 0)

    interface IDisplay with
        member x.draw (memory: byte[]) frame =
            if frame % 16us = 0us then
                videoProcessor.ChangeFlash
            videoProcessor.DrawImage memory
            x.Invalidate()
end
```

We are done with the Video Output.

5 Microprocessor

The Z80 microprocessor is an 8-bit microprocessor by Zilog. It was launched in 1976 and was used in many desktop computers and other systems, such as video games, musical instruments etc[7].

5.1 Registers set

Z80 has 208 bits of read/write memory that are available via the registers. There are eighteen 8-bit registers and four 16-bit registers. All registers are implemented using static RAM. There are six general purpose registers (`B`, `C`, `D`, `E`, `H`, `L`) and an accumulator (`A`) and flags register (`F`). All eight registers have alternate (shadow) register counterparts, which are not directly accessible though. There are also 6 special purpose registers (`I`, `R`, `IX`, `IY`, `SP`, `PC`).

AF formed by two 8 bit registers: accumulator register `A` and flag bits register `F`.

BC formed by two 8 bit registers: `B` and `C`.

DE formed by two 8 bit registers: `D` and `E`.

HL formed by two 8 bit registers: `H` and `L`.

SP 16-bit stack pointer.

PC 16-bit program counter.

IX 16-bit index register.

IY 16-bit index register.

I 8-bit interrupt page address register.

R 8-bit refresh counter.

AF' shadow `AF` register.

BC' shadow `BC` register.

DE' shadow `DE` register.

HL' shadow `HL` register.

`PC` register points to the current instruction being fetched from memory. `PC` is automatically incremented after its contents are transferred to the address lines.

`SP` register points to the location in memory, which indicates top of a stack. The stack can be located anywhere in external system RAM memory. Data can be pushed onto the stack from specific registers or popped off of the stack to specific registers.

`IX` and `IY` registers hold 16-bit base addresses that are used in indexed addressing modes, and additional byte is included in indexed instructions to specify a displacement (as a two's compliment signed integer) from the base.

The Z80 CPU can be operated in a mode in which an indirect call to any memory location can be achieved in response to an interrupt. The `I` register is used for this purpose and stores the high-order eight bits of the indirect address while the interrupting device provides the lower eight bits of the address. This feature allows interrupt routines to be dynamically located anywhere in memory with minimal access time to the routine[2].

The Z80 CPU contains a memory refresh counter, enabling dynamic memories to be used with the same ease as static memories. Seven bits of this 8-bit register are automatically incremented after each instruction fetch. The eighth bit remains as programmed, resulting from an `LD R, A` instruction. The data in the refresh counter is sent out on the lower portion of the address bus along with a refresh control signal while the CPU is decoding and executing the fetched instruction. This mode of refresh is transparent to the programmer and does not slow the CPU operation. The programmer can load the `R` register for testing purposes, but this register is normally not used by the programmer. During refresh, the contents of the I Register are placed on the upper eight bits of the address bus[2].

The flag bits register `F` has the following fields:

7	6	5	4	3	2	1	0
S	Z	-	H	-	P	N	C

Table 5: Flags register

S Sign flag.

Z Zero flag.

H Half-carry flag.

P Parity/overflow flag.

N Add/Subtract flag.

C Carry flag.

5.2 The Main Class

We are going to define `Z80` class, which will take `IPort` as constructor parameter (which is going to be our Keyboard).

We will define an exception that will be thrown when an instruction[2] is not supported.

16 ⟨*Types* 2c⟩+≡ (1b) ◁14b 17a▷

```
exception InstructionNotSupported of string
```

[2]Words "instruction" and "operation" are used interchengeably throughout this program.

17a ⟨*Types* 2c⟩+≡ (1b) ◁16 83▷
```
type Z80(keyboard: IPort) as this = class
    ⟨Registers 17c⟩
    ⟨Registers set 47a⟩
    ⟨Index registers 18a⟩
    ⟨Flags 18b⟩
    ⟨Registers 16-bit 18c⟩
    ⟨Register arrays 17b⟩
    ⟨Memory 21a⟩
    ⟨Precomputed flags 18d⟩
    ⟨Overflow/Half-carry tables 20a⟩
    ⟨Video related contants 22⟩
    ⟨Mutables 23b⟩
    ⟨Constants 29⟩
    ⟨Instruction tables 33a⟩
    ⟨Type initialization 82b⟩
    ⟨Getters/Setters 21b⟩
    ⟨Members 21c⟩
end
```

We will keep all register in an array of 16 bytes, initialized with all the items to 0:

17b ⟨*Register arrays* 17b⟩≡ (17a) 17d▷
```
let register = Array.create 16 0uy
```

Then, we define constants to address registers in the array (e.g. `register.[A]`). Register names that have suffixes `S` are the shadow registers.

17c ⟨*Registers* 17c⟩≡ (17a)
```
let A = 0
let F = 1
let B = 2
let C = 3
let D = 4
let E = 5
let H = 6
let L = 7
let AS = 8
let FS = 9
let BS = 10
let CS = 11
let DS = 12
let ES = 13
let HS = 14
let LS = 15
```

We will also need an array of 2 16-bit registers, namely `IX` and `IY`, so let's define those:

17d ⟨*Register arrays* 17b⟩+≡ (17a) ◁17b
```
let index_register = Array.create 2 0us
```

And also we define constants to address them:

18a ⟨*Index registers* 18a⟩≡ (17a)
```
let IX = 0
let IY = 1
```

Next create a set of constants that will allow us to read/set bits in our flags register `F`. Those are pretty straightforward:

18b ⟨*Flags* 18b⟩≡ (17a)
```
let S_FLAG = 0b10000000uy
let Z_FLAG = 0b01000000uy
let X5_FLAG = 0b00100000uy
let H_FLAG = 0b00010000uy
let X3_FLAG = 0b00001000uy
let P_FLAG = 0b00000100uy
let V_FLAG = 0b00000100uy
let N_FLAG = 0b00000010uy
let C_FLAG = 0b00000001uy
let X53_FLAGS = X5_FLAG ||| X3_FLAG
let SZ_FLAGS = S_FLAG ||| Z_FLAG
let SZP_FLAGS = SZ_FLAGS ||| P_FLAG
let SZHN_FLAGS = SZ_FLAGS ||| H_FLAG ||| N_FLAG
```

Note that `X5_FLAG` and `X3_FLAG` are not officially documented, still Z80 sets them.

We also have a few 16-bit registers along with `IX` and `IY`, namely `SP` and `PC`, and let's put `IR` here too (even though it is formed using two 8-bit registers `I` and `R`):

18c ⟨*Registers 16-bit* 18c⟩≡ (17a)
```
let mutable ir = 0us
let mutable sp = 0us
let mutable pc = 0us
```

We will set various flags, like sign flag or zero flag, so let's precompute these values:

18d ⟨*Precomputed flags* 18d⟩≡ (17a) 19a ▷
```
let precomputed_flags =
      Array.create 256 0uy |> Array.mapi (
                fun i v -> (if i > 0x7F then S_FLAG else 0uy) |||
                           ((byte i) &&& X53_FLAGS) |||
                           (if i = 0 then Z_FLAG else 0uy))
```

Defines:
`precomputed_flags`, used in chunk 19a.

Note that we also set flag 5 and 3 which are officially not documented, although Z80 copies these bits from the operand value. We could have written something like `i &&& S_FLAG` instead of explicit `if`, but it seems to be more readable.

The `N` is used by the `DAA` (Decimal Adjust Accumulator) instruction to distinguish between the `ADD` and `SUB` instructions. For `SUB` instructions, `N` is set to 1, otherwise it's zero. So let's create two sets of precomputed flags, one for `ADD` operations, and the other for `SUB` operations:

19a ⟨*Precomputed flags* 18d⟩+≡ (17a) ◁18d 19b▷

```
let sz53n_add = precomputed_flags
let sz53n_sub = precomputed_flags |> Array.map (fun v -> v ||| N_FLAG)
```

Defines:
`sz53n_add`, used in chunks 19c, 39, 47b, 52, 61, 64, 69, and 72.
`sz53n_sub`, used in chunks 19c, 39, 47b, 52, 60, and 69.
Uses `precomputed_flags` 18d.

Basically, that's the same expect `N` flag is set.

The flag `P`, which is parity/overflow flag, has different uses depending on operations. For arithmetic operations this flag is set if the result is overflown. Such situation can be detected if both operands have the same sign bits and the result has a different sign bit. The overflow doesn't occur if operands have different sign bits.

However, when a logical operations or rotate instructions are executed, this flag indicates parity of the result. That is, if the result parity is even, the flag's set to 1. If the number of 1 bits in a byte is odd, the `P` flag is unset, otherwise it is set.

So let's write a helper function that calculates that:

19b ⟨*Precomputed flags* 18d⟩+≡ (17a) ◁19a 19c▷

```
let powers_of_two = 1 |> Seq.unfold (fun i -> Some(i, i <<< 1))
let parity v len = powers_of_two |>
                   Seq.take len |>
                   Seq.fold (fun e i -> if i &&& v <> 0 then not e else e) true
let parityByte v = parity v 8
```

Now let's have another two arrays with parity bits pre-calculated:

19c ⟨*Precomputed flags* 18d⟩+≡ (17a) ◁19b

```
let sz53pn_add = sz53n_add |> Array.mapi (fun i v -> v ||| (if parityByte i then P_FLAG else 0uy))
let sz53pn_sub = sz53n_sub |> Array.mapi (fun i v -> v ||| (if parityByte i then P_FLAG else 0uy))
```

Defines:
`sz53pn_add`, used in chunks 40a, 52, 59a, 64d, 65, 68, 69, and 71.
`sz53pn_sub`, used in chunk 40a.
Uses `sz53n_add` 19a and `sz53n_sub` 19a.

Like it was mention earlier, arithmetic operations may set overflow flag. In addition to that, they may set half-carry flag.

Let's have a look at half-carry situation first. The flag `H` is set to 1 or 0 depending on the carry/borrow between bits 3 and 4 while executing 8-bit arithmetic operation. The flag is used by the Decimal Adjust Accumulator instruction to correct the result of a packed BCD add or subtract operation[2].

It is set to 1 while adding a carry occurs from bit 3 to bit 4. It is set to 0 while adding and no carry occurs from bit 3 to bit 4. It is set to 1 while subtracting and a borrow from bit 4 occurs. It is set to 0 while subtracting and no borrow occurs from bit 4.

Suppose we are doing `ADD` operation. Let's encode bit 3 of operands and the results like this: $B_1B_2B_3$, where B_1 is bit 3 of the result, B_2 is bit 3 of operand 1, and B_3 is bit 3 of operand 2. If we have $B_1B_2B_3 = 000$, we can be sure that no half-carry occurs. On the contrary, if we have $B_1B_2B_3 = 001$ we know that half-carry occurred, because operand 1 is 0, operand 2 is 1, and the result is 0, means that there was a carry from bit 2 to 3, and therefore from bit 3 to 4. Similarly, for $B_1B_2B_3 = 010$. In case of $B_1B_2B_3 = 011$ obviously there was no carry from 2 to 3, but was from 3 to 4. Following similarly for all other cases we can construct Table 6 (also see [3] and [11]).

Index	$B_1B_2B_3$	Flag status ADD	Flag status SUB
0	`000`	0	0
1	`001`	1	0
2	`010`	1	1
3	`011`	1	0
4	`100`	0	1
5	`101`	0	0
6	`110`	0	1
7	`111`	1	1

Table 6: Half-carry flags

We can translate this table directly to code:

20a ⟨*Overflow/Half-carry tables* 20a⟩≡ (17a) 20b ▷

```
let halfcarry_add_table = [| false; true; true; true; false; false; false; true |]
let halfcarry_sub_table = [| false; false; true; false; true; false; true; true |]
```

For overflow flag in case of `ADD` it's simple: if B_2 and B_3 are the same, but B_1 is different, then we know overflow occurred (that is, if you add two values with the same sign, but you get a result of with a different sign).

In case of `SUB` just note that subtracting a negative is the same as adding a positive, and subtracting a positive is the same as adding a negative[11].

Therefore we get values as defined in Table 7 and we translate that straight into code.

20b ⟨*Overflow/Half-carry tables* 20a⟩+≡ (17a) ◁ 20a

```
let overflow_add_table = [| false; false; false; true; true; false; false; false |]
let overflow_sub_table = [| false; true; false; false; false; false; true; false |]
```

Defines:
`overflow_add_table`, used in chunks 39 and 61.
`overflow_sub_table`, used in chunks 39, 52, and 60.

Index	$B_1B_2B_3$	Flag status ADD	Flag status SUB
0	`000`	0	0
1	`001`	0	1
2	`010`	0	0
3	`011`	1	0
4	`100`	1	0
5	`101`	0	0
6	`110`	0	1
7	`111`	0	0

Table 7: Overflow flags

5.3 ROM and RAM Memory

ZX Spectrum 48K had 16K of ROM memory, located in the lowest part of the address space plus 48K of RAM.

Let us define our main memory as an array of 65536 bytes:

21a ⟨*Memory* 21a⟩≡ (17a)

```
let memory = Array.create (0xFFFF+1) 0xFFuy
```

Let's add a getter:

21b ⟨*Getters/Setters* 21b⟩≡ (17a) 23c ▷

```
member x.Memory
    with get() = memory
```

We will also write a function to load ROM image into memory:

21c ⟨*Members* 21c⟩≡ (17a) 23a ▷

```
member x.loadRom(romFile: String) =
      let rom = File.ReadAllBytes(romFile)
      rom.CopyTo(memory, 0)
```

Defines:
`loadRom`, used in chunk 83.

Cycle	Delay
14335	6
14336	5
14337	4
14338	3
14339	2
14340	1
14341	0
14342	0
14343	6
14343	5
...	...
Until 14463 and then for 96 cycles delay is 0.	
From 14559 the pattern repeats.	
The pattern is valid for all 192 lines of screen data.	

Table 8: Delays pattern

5.4 Contended Memory

Uncommitted Logic Array (ULA) is a chip that is responsible for the display, audio I/O and keyboard.

When ULA is drawing the screen, it reads memory from `0x4000` to `0x5B00` (bitmap and attributes). When CPU and ULA are reading same memory areas at the same time, a contention occurs. In that case ULA has a priority, so execution of a program in contended memory is slowed. The execution is not slowed when the border is being drawn as the ULA supplies the result[8].

An interrupt in ZX Spectrum occurs every 50Hz, and this is synchronized with the video signal generation by the ULA. After an interrupt occurs 14336 cycles pass before the first byte of the screen is displayed[8]. The memory access delay depends on the current cycle which in turn relates to the position in currently rendered display output.

The pattern is defined in Table 8.

Note that from 14335 to 14463 there are 128 cycles, so the pattern repeats every 128 cycles. Starting from 14463 there's no delay for another 96 cycles, because the ULA is updating the border. 24 cycles takes the left border, 24 cycles the right border, and 48 cycles for the retrace. Every half cycle (T state) a pixel is written to CRT[8], so a byte is read in 4 cycles. The border is 48 pixels wide, therefore it takes 24 cycles. We have left and right borders, so it adds up to 48 cycles. And 256 pixels line takes another 128 cycles, thus the whole line takes exactly 224 cycles.

In total it takes $(64 + 192 + 56) * 224 = 69888$ cycles to render a frame, where 64 and 56 account for the borders, and 192 is the number of lines, each taking 224 cycles. Every 16 frames, the Flash effect is produced, where ink and paper of all flashing bytes is swapped[8] (see Section 4 on page 8).

Let us define a few constants that we will be using while calculating cycle delays in the contended memory:

22 ⟨*Video related contants* 22⟩≡ (17a)
```
let SCREEN_CONTENTS = 128
let LEFT_BORDER = 24
let RIGHT_BORDER = 24
let RETRACE = 48
let SCREEN_CONTENTS_AND_BORDER_AND_RETRACE =
        SCREEN_CONTENTS + RIGHT_BORDER + LEFT_BORDER + RETRACE
let TOTAL_LINES = 192
let PATTERN_LENGTH = 8
let TSTATE_START = 14335
let TSTATE_END =
        TSTATE_START + SCREEN_CONTENTS_AND_BORDER_AND_RETRACE * TOTAL_LINES - 1
let CONTENDED_MEMORY_START = 0x4000us
let CONTENDED_MEMORY_END = 0x7FFFus
let FRAME_CYCLES_TOTAL = 69888
```

Now we will write a function that takes cycle as argument and returns delay as described in Table 8. If `cycle` $\notin$ `[TSTATE_START; TSTATE_END]` then the delay is 0. `SCREEN_CONTENTS` constant is the length of the pattern. So by taking `b_m = (tstate - TSTATE_START) % SCREEN_CONTENTS_AND_BORDER_AND_RETRACE` gives us where exactly in one of those 128 lines we are. If we are outside our 128 lines, then the delay is 0, otherwise we need to calculate further. `b_m` < 128, and we know that the pattern is 6, 5, 4, 3, 2, 1, 0, 0, so its length is 8 (`PATTERN_LENGTH`). Simple arithmetic `(PATTERN_LENGTH - (b_m % PATTERN_LENGTH)) - 2` gives us the number, and if it's greater than zero, that's exactly the delay, otherwise just return zero. Here's the chunk:

23a ⟨*Members* 21c⟩+≡ (17a) ◁21c 24a▷

```
member x.get_contention_delay tstate =
      match tstate with
        | tstate when tstate < TSTATE_START || tstate > TSTATE_END -> 0
        | _ ->  let b_m = (tstate - TSTATE_START) % SCREEN_CONTENTS_AND_BORDER_AND_RETRACE
                if b_m >= SCREEN_CONTENTS then 0 else
                    let delay = (PATTERN_LENGTH - (b_m % PATTERN_LENGTH)) - 2
                    if delay > 0 then delay else 0
```

Defines:
`get_contention_delay`, used in chunk 24.

OK, let's define `tstates` and `frames` variables that will keep counting cycles and frames:

23b ⟨*Mutables* 23b⟩≡ (17a) 31a▷

```
let mutable tstates = 0
let mutable frames = 0us
```

And let's write getters for them:

23c ⟨*Getters/Setters* 21b⟩+≡ (17a) ◁21b 31b▷

```
member x.Tstates
   with get() = tstates

member x.Frames
   with get() = frames
```

Now we write a function that adds delays to `tstates` if contended memory access occurs. Remember, the contention occurs when memory region `0x4000..0x7FFF` is accesses and the actual delay depends on current cycle:

24a ⟨*Members* 21c⟩+≡ (17a) ◁23a 24b▷

```
  member x.check_memory_contention (addr: uint16) =
      tstates <- tstates + (if addr >= CONTENDED_MEMORY_START && addr <= CONTENDED_MEMORY_END
                            then x.get_contention_delay tstates
                            else 0)
```

Defines:
`check_memory_contention`, used in chunks 27, 28, 30a, and 32c.
Uses `get_contention_delay` 23a.

We will be using another helper function, similar to `check_memory_contention` but with an additional `times` parameter, which will add delays repeatedly as in the following chunk:

24b ⟨*Members* 21c⟩+≡ (17a) ◁24a 27▷

```
  member x.check_memory_contention_m (addr: uint16) times =
      if addr >= CONTENDED_MEMORY_START && addr <= CONTENDED_MEMORY_END then
          for i = 1 to times do
                tstates <- tstates + 1 + x.get_contention_delay tstates
      else
          tstates <- tstates + times
```

Defines:
`check_memory_contention_m`, used in chunks 27, 28, 36, 42e, 44–46, 49, 50a, 54a, 56, 57, 60, 61, 63–69, and 73–82.
Uses `get_contention_delay` 23a.

Instruction	Delay
NOP LD ⟨A|B|C|D|E|H|L⟩, ⟨A|B|C|D|E|H|L⟩ ⟨ADD|ADC|SUB|SBC|AND|XOR|OR|CP⟩A, ⟨A|B|C|D|E|H|L⟩ INC/DEC ⟨A|B|C|D|E|H|L⟩ EXX EX AF,AF' EX DE,HL DAA CPL CCF SCF DI EI RLA RRA RLCA RRCA JP (HL)	PC:4
NOPD ⟨RLC|RRC|RL|RR|SLA|SRA|SRL|SLL⟩ ⟨A|B|C|D|E|H|L⟩ BIT [0..7],⟨A|B|C|D|E|H|L⟩ SET [0..7],⟨A|B|C|D|E|H|L⟩ RES [0..7],⟨A|B|C|D|E|H|L⟩ NEG IM 0/1/2	PC:4, PC+1:4
LD A,I LD A,R LD I,A LD R,A IM 0/1/2	PC:4, PC+1:4, IR:1
INC/DEC ⟨BC|DE|HL|SP⟩ LD SP,HL	PC:4, IR:1x2
ADD HL,⟨BC|DE|HL|SP⟩	PC:4, IR:1x7
ADC HL,⟨BC|DE|HL|SP⟩ SBC HL,⟨BC|DE|HL|SP⟩	PC:4, PC+1:4, IR:1x7
LD ⟨A|B|C|D|E|H|L⟩,N_8 ⟨ADD|ADC|SUB|SBC|AND|XOR|OR|CP⟩ A,N_8	PC:4,PC+1:3
LD ⟨A|B|C|D|E|H|L⟩,(⟨BC|DE|HL⟩) LD (⟨BC|DE|HL⟩),⟨A|B|C|D|E|H|L⟩	PC:4, ⟨BC|DE|HL⟩:3
⟨ADD|ADC|SUB|SBC|AND|XOR|OR|CP⟩ A,(HL)	PC:4, HL:3
LD ⟨A|B|C|D|E|H|L⟩,(⟨IX|IY⟩+N_8) LD (⟨IX|IY⟩+N_8),⟨A|B|C|D|E|H|L⟩ ⟨ADD|ADC|SUB|SBC|AND|XOR|OR|CP⟩ A,(⟨IX|IY⟩+N_8)	PC:4, PC+1:4, PC+2:3, PC+2:1x5, ⟨IX|IY⟩+N_8:3
BIT [0..7],(HL)	PC:4, PC+1:4, HL:3, HL:1
BIT [0..7],(⟨IX|IY⟩+N_8)	PC:4, PC+1:4, PC+2:3, PC+3:3, PC+3:1x2, ⟨IX|IY⟩+N_8:3, ⟨IX|IY⟩+N_8:1
LD ⟨BC|DE|HL|SP⟩,N_{16} JP ⟨NZ|Z|NC|C|PO|PE|P|M⟩,N_{16}	PC:4, PC+1:3, PC+2:3

Table 9: Instructions and delays 1-2

Instruction	Delay
LD (HL),N_8	PC:4, PC+1:3, HL:3
LD (⟨IX|IY⟩+N_8),N_8	PC:4, PC+1:4, PC+2:3, PC+3:3, PC+3:1x2, ⟨IX|IY⟩+N_8:3
LD A,(N_{16}) LD (N_{16}),A	PC:4, PC+1:3, PC+2:3, N_{16}:3
LD HL,(N_{16}) LD (N_{16}),HL	PC:4, PC+1:3, PC+2:3, N_{16}:3, N_{16}+1:3
LD ⟨BC|DE|HL|SP⟩,(N_{16}) LD (N_{16}),⟨BC|DE|HL|SP⟩	PC:4, PC+1:4, PC+2:3, PC+3:3, N_{16}:3, N_{16}+1:3
INC/DEC (HL)	PC:4, HL:3, HL:1, HL(write):3
SET [0..7], (HL) RES [0..7], (HL) ⟨RLC|RRC|RL|RR|SLA|SRA|SRL|SLL⟩ (HL)	PC:4, PC+1:4, HL:3, HL:1, HL(write):3
INC/DEC (⟨IX|IY⟩+N_8)	PC:4, PC+1:4, PC+2:3, PC+2:1x5, ⟨IX|IY⟩+ N_8:3, ⟨IX|IY⟩+N_8:1, ⟨IX|IY⟩+N_8(write):3
SET [0..7],(⟨IX|IY⟩+N_8) RES [0..7], (⟨IX|IY⟩+N_8) ⟨RLC|RRC|RL|RR|SLA|SRA|SRL|SLL⟩ (⟨IX|IY⟩+N_8)	PC:4, PC+1:4, PC+2:3, PC+3:3, PC+3:1x2, ⟨IX|IY⟩+N_8:3, ⟨IX|IY⟩+N_8:1, ⟨IX|IY⟩+N_8(write):3
POP ⟨BC|DE|HL|SP⟩ RET	PC:4, SP:3, SP+1:3
RETI RETN	PC:4, PC+1:4, SP:3, SP+1:3
RET ⟨NZ|Z|NC|C|PO|PE|P|M⟩	PC:4, IR:1, [SP:3,SP+1:3]
PUSH ⟨BC|DE|HL|SP⟩ RST N_8	PC:4, IR:1, SP-1:3, SP-2:3
CALL N_{16} CALL ⟨NZ|Z|NC|C|PO|PE|P|M⟩, N_{16}	PC:4, PC+1:3, PC+2:3, [PC+2:1, SP-1:3, SP-2:3]
JR N_8 JR ⟨NZ|Z|NC|C|PO|PE|P|M⟩, N_8	PC:4, PC+1:3, [PC+1:1x5]
DJNZ N_8	PC:4, IR:1, PC+1:3, [PC+1:1x5]
RLD RRD	PC:4, PC+1:4, HL:3, HL:1x4, HL(write):3
IN A,(N_8) OUT (N_8),A	PC:4, PC+1:3, I/O
IN ⟨A|B|C|D|E|H|L⟩,(C) OUT (C),⟨A|B|C|D|E|H|L⟩	PC:4, PC+1:4, I/O
EX (SP),HL	PC:4, SP:3, SP+1:3, SP+1:1, SP+1(write):3, SP(write):3, SP(write):1x2
LDI/LDIR LDD/LDDR	PC:4, PC+1:4, HL:3, DE:3, DE:1x2, [DE:1x5]
CPI/CPIR CPD/CPDR	PC:4, PC+1:4, HL:3, HL:1x5, [HL:1x5]
INI/INIR IND/INDR	PC:4, PC+1:4, IR:1, I/O, HL:3, [HL:1x5]
OUTI/OTIR OUTD/OTDR	PC:4, PC+1:4, IR:1, HL:3, I/O, [BC:1x5]

Table 10: Instructions and delays 2-2

Now, to apply a delay when an instruction is executed we need to know in which point within the instruction is I/O operation happening. However, that depends on instruction. Such delays per instruction can be found in Table 9 and Table 10. (Tables are from [14]; also see [8]).

We will be using these tables when implementing instructions. In these tables:

- N_8: An 8-bit value.
- N_{16}: A 16-bit value.
- $\langle X_1|X_2|\dots|X_N\rangle$: Means one of $X_1 \dots X_N$.
- [0..7]: A number from 0 to 7.
- Entries in square brackets e.g. [*smth*] are applied only when the condition is met. If the instruction is not conditional, an entry is always applied.

In the table figures like $R : N$ mean that if $R \in$ `[0x4000..0x7FFF]` then the delay corresponding to the current cycle should be inserted before inserting N.

5.5 Contended I/O

Access to ports are contended too. Ports that have even address (bit 0 is a 0) are ULA ports[15]. In this case the ULA pauses the processor by stopping its clock and if the port being accessed is between `0x4000` and `0x7FFF`, then the contention also occurs because ULA regards this as access to contended memory[16].

In this emulator we don't support writing to ports. Therefore we implement writing to ports as in the following chunk:

27 ⟨*Members* 21c⟩+≡ (17a) ◁24b 28▷

```
member x.write_to_port (port: uint16) (value: byte) =
    x.check_memory_contention port
    tstates <- tstates + 1

    if port &&& 0x0001us <> 0us then
        x.check_memory_contention_m port 3
    else
        x.check_memory_contention port
        tstates <- tstates + 3
```

Defines:
`write_to_port`, used in chunks 43a, 59b, and 69.
Uses `check_memory_contention` 24a and `check_memory_contention_m` 24b.

Reading from the ports is similarly implemented, and the only ports that we will be really reading are keyboard ports.

28 ⟨*Members* 21c⟩+≡ (17a) ◁27 30a▷

```
  member x.read_from_port (port: uint16) =
     x.check_memory_contention port
     tstates <- tstates + 1

     if port &&& 0x0001us <> 0us then
         x.check_memory_contention_m port 3
     else
         x.check_memory_contention port
         tstates <- tstates + 3

     if port &&& 0x00FFus = 0xFEus then
         keyboard.read port
     else
         0xFFuy
```

Uses `check_memory_contention` 24a and `check_memory_contention_m` 24b.

5.6 Interrupts

An interrupt suspends normal program execution and forces the CPU to start another routine. When this routine is completed, the CPU returns to the program execution from which it was interrupted.

The Z80 has two interrupt inputs: maskable interrupt and a non-maskable interrupt. The non-maskable interrupt cannot be disabled programmatically.

There's an interrupt enable flip-flop (`IFF`) that is set/reset when using `EI` and `DI` instructions. When `IFF` is reset, an interrupt cannot be accepted by the CPU. There are two enable flip-flopts: `IFF1` and `IFF2`. `IFF1` is used to inhibit interrupts, and `IFF2` is used as a temporary storage location for `IFF1`. (The purpose of `IFF2` is to save the status of `IFF1` when a nonmaskable interrupt occurs. For more details see [2].)

There are three interrupt modes for the maskable interrupts.

Mode 0 In this mode the interrupting device provides the next instruction to be executed by placing the instruction on the data bus. ZX Spectrum is not using this mode.

Mode 1 In this mode the CPU responds to an interrupt by restarting the CPU at address `0x0038`. In ZX Spectrum this interrupt happens every 50Hz as this is tied to screen refresh.

Mode 2 In this mode a table of interrupt routines is maintained. The table can be located anywhere in the memory. When an interrupt occurs, 16-bit pointer must be formed to obtain the routine address from the table. The uppoer 8-bits of this pointer are formed from the contents of the `I` register, while the lower 8-bits are taken from the data bus, and are supplied by the interrupting device. These lower 8-bits number should be an even number (i.e. bit 0 is 0), because the pointer must receive two adjacent bytes to form 16-bit address. (See [2]).

Let's implement interrupt handler now, but before that, we need a couple of other methods. We need push-to-stack method, as well as reading from memory, and increment memory refresh register method.

Reading and writing memory takes 3 clock cycles, let's define that as constants:

29 ⟨*Constants* 29⟩≡ (17a) 32b ▷

```
let MEMORY_READ_TIME = 3
let MEMORY_WRITE_TIME = 3
```

Implementation is fairly simple:

30a ⟨*Members* 21c⟩+≡ (17a) ◁28 30b▷
```
member x.read_from_memory (addr: uint16) =
    x.check_memory_contention addr
    tstates <- tstates + MEMORY_READ_TIME
    memory.[(int)addr]

member x.write_to_memory (addr: uint16) (data: byte) =
    x.check_memory_contention addr
    tstates <- tstates + MEMORY_WRITE_TIME
    memory.[(int)addr] <- data
```

Defines:
`read_from_memory`, used in chunks 30d, 32a, 36, 37b, 40, 41, 43–45, 48–51, 53, 55, 56a, 58a, 62b, 64–67, 69, 73–75, and 77–80.
`write_to_memory`, used in chunks 30d, 37c, 41, 44a, 46a, 50a, 51b, 62a, 64–66, 68, 73, 75a, and 77–82.
Uses `check_memory_contention` 24a.

OK, let's have a look now at how to increment memory refresh register. One byte instruction increment the register by one. Instructions with `ED`, `CB`, `DD` and `FD` prefixes increment the register by two. `DDCB` and `FDCB` prefixed instructions also increment it by two[17]. 8th bit is never changed by this increment, only the lowest 7 bits are incremented. 8th can be changed though (e.g. using `LD R, A`).

30b ⟨*Members* 21c⟩+≡ (17a) ◁30a 30c▷
```
member x.increment_memory_refresh_register =
    let r = ir &&& 0x00FFus
    let r = (r &&& 0b10000000us) ||| ((r + 1us) &&& 0b1111111us)
    ir <- (ir &&& 0xFF00us) ||| r
```

Let's define another helper method, which takes two bytes and returns 16-bit integer:

30c ⟨*Members* 21c⟩+≡ (17a) ◁30b 30d▷
```
member x.get_compound (r1: byte) (r2: byte) = (uint16 r1 <<< 8) ||| (uint16 r2)
```

Pushing to stack and popping from it is simple too, so let's write the corresponding methods:

30d ⟨*Members* 21c⟩+≡ (17a) ◁30c 32a▷

```
member x.push_to_stack (v: uint16) =
    sp <- sp - 1us
    x.write_to_memory sp (byte (v >>> 8))
    sp <- sp - 1us
    x.write_to_memory sp (byte v)

member x.pop_from_stack =
    let addr1 = x.read_from_memory sp
    sp <- sp + 1us
    let addr2 = x.read_from_memory sp
    sp <- sp + 1us
    x.get_compound addr2 addr1
```

Defines:
`pop_from_stack`, used in chunks 42c, 54, 63b, and 79c.
`push_to_stack`, used in chunks 32a, 42e, 56, 57, and 79d.
Uses `read_from_memory` 30a and `write_to_memory` 30a.

Let's finally define a few mutables such as `IFF1`/`IFF2`, `halt` flag, interrupt mode and interruption variables:

31a ⟨*Mutables* 23b⟩+≡ (17a) ◁23b

```
let mutable im = 0
let mutable ei = false
let mutable iff1 = false
let mutable iff2 = false
let mutable halt = false
let mutable intr = false
```

And some getter/setters:

31b ⟨*Getters/Setters* 21b⟩+≡ (17a) ◁23c

```
member x.R
     with get() = byte ir
     and set(v: byte) = ir <- (ir &&& 0xFF00us) ||| (uint16 v)

member x.I
    with get() = byte (ir >>> 8)
    and set(v: byte) = ir <- (ir &&& 0x00FFus) ||| ((uint16 v) <<< 8)

member x.Iff1
    with get() = iff1
    and set(v) = iff1 <- v

member x.Iff2
    with get() = iff2
    and set(v) = iff2 <- v
```

Now the interrupt method itself. If `halt` flag is set then we reset it and increment `PC` counter. Next, memory refresh register is incremented (accepting an interrupt increases the `R` register by one[17]), flip-flop flags are reset and `PC` is pushed onto stack. Finally, depending on interruption mode, we either set `PC` ← `0x38` or form a 16-bit address from the contents of the `I` register and data on the data bus (which in our case is always `0xFF`).

32a ⟨*Members* 21c⟩+≡ (17a) ◁30d 32c▷
```
member x.interrupt =
    if halt then
        halt <- false
        pc <- pc + 1us

    tstates <- tstates + 7
    x.increment_memory_refresh_register
    iff1 <- false
    iff2 <- false

    x.push_to_stack pc

    if im = 2 then
        let i = (ir &&& 0xFF00us) >>> 8
        let p = i <<< 8 ||| 0xFFus
        let v1 = x.read_from_memory p
        let v2 = x.read_from_memory (p + 1us)
        let v = x.get_compound v2 v1
        pc <- v
    else
        pc <- 0x38us
```

Defines:
  `interrupt`, used in chunk 34a.
Uses `push_to_stack` 30d and `read_from_memory` 30a.

5.7 Instructions

Fetching an instruction from memory takes 4 clock cycles:

32b ⟨*Constants* 29⟩+≡ (17a) ◁29 36e▷
```
let OP_FETCH_TIME = 4
```

Let's write a method that takes next instruction from memory:

32c ⟨*Members* 21c⟩+≡ (17a) ◁32a 33b▷
```
member x.read_next_instruction =
    x.check_memory_contention pc
    tstates <- tstates + OP_FETCH_TIME
    memory.[(int)pc]
```

Defines:
  `read_next_instruction`, used in chunks 33b and 42–45.
Uses `check_memory_contention` 24a.

The method above return a byte from memory. This byte is either an op-code by itself, or a prefix of a larger op-code. There are main instructions, extended instruction, bit instructions and index instructions.

Extended instructions are two-byte instructions, prefixed with `0xED`. Bit instructions are also two-byte instructions prefixed with `0xCB`. `IX` instruction are prefixed with `0xDD`, and `IY` instruction are prefixed with `0xFD`. There are also `IX/IY` bit instructions, prefixed by `0xDDCB/0xFDCB`. Therefore, for `IX/IY` related instruction we will define functions that take `ireg` as argument, selecting the corresponding register (either `IX` or `IY`).

33a ⟨*Instruction tables* 33a⟩≡ (17a)
```
let operations =
    Array.init 256 (fun i ->
        (fun () -> raise (InstructionNotSupported("Unknown Instruction")) ))
let operations_extended =
    Array.init 256 (fun i ->
        (fun () -> raise (InstructionNotSupported("Unknown Extended Instruction")) ))
let operations_bits =
    Array.init 256 (fun i ->
        (fun () -> raise (InstructionNotSupported("Unknown Bit Instruction")) ))
let operations_indexes =
    Array.init 256 (fun i ->
        (fun (ireg: int) -> raise (InstructionNotSupported("Unknown Index Instruction")) ))
let operations_indexes_bits =
    Array.init 256 (fun i -> (fun (address: uint16) (value: byte) ->
            raise (InstructionNotSupported("Unknown Index Bit Instruction")) ))
```

Now executing next instruction is straightforward:

33b ⟨*Members* 21c⟩+≡ (17a) ◁32c 34a▷
```
member public x.execute_next_instruction() =
    x.increment_memory_refresh_register
    let op = x.read_next_instruction
    pc <- pc + 1us
    operations.[(int)op]()
```

Defines:
  `execute_next_instruction`, used in chunk 34a.
Uses `read_next_instruction` 32c.

It was mentioned earlier, that interrupts in ZX Spectrum are executed every 50Hz, which is tied to screen refresh. That is, it happens every frame, which is 69888 clock cycles. Therefore, we better define a method that executes instruction until specified cycle. Basically, it will check current clock cycle, and if it is lower than specified as argument to the method, then continue execution. It also checks whether `intr` flag is set, meaning that interrupts are allowed. Then if it is set, it checks whether `IFF1` is set and no other interrupt is now executing (by checking `not ei`) it interrupts the program. After interrupt, it executes next instruction, and resets `ei` if necessary:

34a ⟨*Members* 21c⟩+≡ (17a) ◁33b 34b▷

```
member x.execute utstates =
    while tstates < utstates do
        if intr && iff1 && (not ei) then
            x.interrupt
        x.execute_next_instruction()
        if ei then ei <- false
```

Defines:
`execute`, used in chunk 34b.
Uses `execute_next_instruction` 33b and `interrupt` 32a.

Having `execute` method, let's write `execute_frame`, which will execute a program for 69888 clock cycles. If it's in the beginning (clock cycle counter is less than 32), then we set a flag allowing interrupts. After that, interrupts are disallowed. Finally, we increment `frames` counter, take modulo of `tstates`, and return `memory`. Returning `memory` will be convenient when rendering video memory to a display.

34b ⟨*Members* 21c⟩+≡ (17a) ◁34a 34c▷

```
member x.execute_frame =
    if tstates < 32 then
        intr <- true
        x.execute 32
    intr <- false
    x.execute(FRAME_CYCLES_TOTAL)
    tstates <- tstates % FRAME_CYCLES_TOTAL
    frames <- frames + 1us
    memory
```

Defines:
`execute_frame`, used in chunk 83.
Uses `execute` 34a.

5.8 Instruction Table Initialization

Now we start adding instructions into our tables.

34c ⟨*Members* 21c⟩+≡ (17a) ◁34b 35c▷

```
member x.initializeOperations =
    ⟨Main Instructions 35a⟩
```

Defines:
`initializeOperations`, used in chunk 82b.

NOP The easiest instruction of all is `NOP`:

35a ⟨*Main Instructions* 35a⟩≡ (34c) 35b ▷
```
  operations.[0x00] <- fun () -> ()
```

RLCA Next operation that we are going to implement is going to be `RLCA`. It's a rotation to the left where 8th bit is copied into carry flag and to 1st bit:

35b ⟨*Main Instructions* 35a⟩+≡ (34c) ◁ 35a 35d ▷
```
  operations.[0x07] <- fun () ->
      let cf = if (register.[A] &&& 0x80uy) <> 0uy then C_FLAG else 0uy
      register.[A] <- (register.[A] <<< 1) ||| cf
      register.[F] <- (register.[F] &&& SZP_FLAGS) ||| (register.[A] &&& X53_FLAGS) ||| cf
```

EX AF, AF' Let's define one helper method, `exchange_register`, which will be helpful in the next operation that we are going to implement and in other places:

35c ⟨*Members* 21c⟩+≡ (17a) ◁ 34c 35f ▷
```
  member x.exchange_registers reg1 reg2 =
      let tmp = register.[reg1]
      register.[reg1] <- register.[reg2]
      register.[reg2] <- tmp
```

Defines:
`exchange_registers`, used in chunks 35d, 43b, and 44c.

Now:

35d ⟨*Main Instructions* 35a⟩+≡ (34c) ◁ 35b 35e ▷
```
  operations.[0x08] <- fun () ->
      x.exchange_registers A AS
      x.exchange_registers F FS
```

Uses `exchange_registers` 35c.

RLCA Similarly to `RLCA`, operation `RRCA` rotates `A` to the right, and 1st bit is copied to carry flag and to 8th bit:

35e ⟨*Main Instructions* 35a⟩+≡ (34c) ◁ 35d 36a ▷
```
  operations.[0x0F] <- fun () ->
      let cf = register.[A] &&& 0x01uy
      register.[A] <- (register.[A] >>> 1) ||| (cf <<< 7)
      register.[F] <- (register.[F] &&& SZP_FLAGS) ||| (register.[A] &&& X53_FLAGS) ||| cf
```

DJNZ * Now we implement the following operation: The register `B` is decremented, and if it's not zero, the signed value of `*` is added to `PC`. Singe we will be required to get signed value of `*` on other commands, let's write a helper:

35f ⟨*Members* 21c⟩+≡ (17a) ◁ 35c 37b ▷
```
  member x.get_signed_byte (v:byte) = int (sbyte v)
```

Defines:
`get_signed_byte`, used in chunks 36, 49b, 77a, 79, and 80d.

OK, the instruction itself is next. Notice how we implement cycle delays in accordance to Table ??:

36a ⟨*Main Instructions* 35a⟩+≡ (34c) ◁35e 36b▷
```
operations.[0x10] <- fun () ->
    x.check_memory_contention_m ir 1
    register.[B] <- (register.[B] - 1uy)
    if register.[B] <> 0uy then
        let offset = x.get_signed_byte(x.read_from_memory pc)
        x.check_memory_contention_m pc 5
        pc <- uint16 (int pc + offset + 1)
    else
        pc <- pc + 1us
```
Uses `check_memory_contention_m` 24b, `get_signed_byte` 35f, and `read_from_memory` 30a.

LD D, * Self-explanatory:

36b ⟨*Main Instructions* 35a⟩+≡ (34c) ◁36a 36c▷
```
operations.[0x16] <- fun () ->
     register.[D] <- x.read_from_memory pc
     pc <- pc + 1us
```
Uses `read_from_memory` 30a.

RLA Rotate register `A` to the left through the carry flag. The previous value of the carry flag is copied to 1st bit. Note that `S`, `Z` and `P` flags are not affected.

36c ⟨*Main Instructions* 35a⟩+≡ (34c) ◁36b 36d▷
```
operations.[0x17] <- fun () ->
    let cf = if (register.[A] &&& 0x80uy) <> 0uy then C_FLAG else 0uy
    register.[A] <- ((register.[A] <<< 1) &&& 0xFEuy) ||| (register.[F] &&& C_FLAG)
    register.[F] <- (register.[F] &&& SZP_FLAGS) ||| (register.[A] &&& X53_FLAGS) ||| cf
```

JR * Unconditional jump. The value of `*` is added to `PC` and the next instruction is fetched from the location pointing by `PC`.

36d ⟨*Main Instructions* 35a⟩+≡ (34c) ◁36c 37a▷
```
operations.[0x18] <- fun () ->
    let offset = x.get_signed_byte(x.read_from_memory pc)
    x.check_memory_contention_m pc 5
    pc <- uint16 (int pc + offset + 1)
```
Uses `check_memory_contention_m` 24b, `get_signed_byte` 35f, and `read_from_memory` 30a.

RRA Rotate register `A` to the right through the carry flag. The previous value of the carry flag is copied to 8th bit. Note that `S`, `Z` and `P` flags are not affected.

Let's define a useful constant:

36e ⟨*Constants* 29⟩+≡ (17a) ◁32b
```
let MASK_8 = 0xFFuy
```

The instruction:

37a ⟨*Main Instructions* 35a⟩+≡ (34c) ◁36d 37c▷

```
operations.[0x1F] <- fun () ->
    let cf = if register.[A] &&& 0x01uy <> 0uy then C_FLAG else 0uy
    register.[A] <- ((register.[A] >>> 1) &&& MASK_8) ||| ((register.[F] &&& C_FLAG) <<< 7)
    register.[F] <- (register.[F] &&& SZP_FLAGS) ||| (register.[A] &&& X53_FLAGS) ||| cf
```

LD (**), HL Store `HL` into memory location pointed by `PC`. Another helper method will be useful:

37b ⟨*Members* 21c⟩+≡ (17a) ◁35f 38▷

```
member x.read_address_pointing_by_pc =
    let offset1 = x.read_from_memory pc
    pc <- pc + 1us
    let offset2 = x.read_from_memory pc
    pc <- pc + 1us
    x.get_compound offset2 offset1
```

Defines:
`read_address_pointing_by_pc`, used in chunks 37c, 40–42, 62, 77, and 78.
Uses `read_from_memory` 30a.

Now the instruction's implementation:

37c ⟨*Main Instructions* 35a⟩+≡ (34c) ◁37a 40a▷

```
operations.[0x22] <- fun () ->
    let offset = x.read_address_pointing_by_pc
    x.write_to_memory offset register.[L]
    x.write_to_memory (offset + 1us) register.[H]
```

Uses `read_address_pointing_by_pc` 37b and `write_to_memory` 30a.

DAA Conditionally adjusts the Accumulator for binary-coded decimals addition and subtraction operations. In BCD any decimal digit can be represented by a four bit pattern. In natural binary-coded decimals, decimal digits are represented by their corresponding four-bit binary values (also called 8421 encoding[18]), which is the case with Z80. An advantage is that this encoding simplifies input/output of the values, and can still be used with normal binary additions and subtraction. However, after an addition or subtraction, the result needs to be corrected in some cases, as the result may involve values that are outside the range of possible values (e.g. `0x0A`). We will call half of a byte a nibble[19]. Correction needs to be done specifically in the following cases: In case of addition, each time when a half-carry happens or if the value is outside of the range, we add a correction value of 0110 (`0x06`) to the nibble from which the bit was carried or the nibble is outside the range. In case of subtraction, each time a bit is borrowed from the higher nibble, the value of 0110 needs to be subtracted from the nibble. (See [2] for more details.)

Let's start off with some helper methods:

38 ⟨*Members* 21c⟩+≡ (17a) ◁37b 39▷

```
member x.sub8_carry a b =
    let res = (int a - int b)
    (byte res, res < 0)

member x.add8_carry a b =
    let res = (int a + int b)
    (byte res, res > 0xFF)
```

Defines:
`add8_carry`, used in chunk 39.
`sub8_carry`, used in chunks 39 and 52.

These methods simply add/subtract two 8bit values, and return the result and the carry flag, if that happened (e.g. for subtraction, if the result is less than zero, and for addition, if it more than `0xFF`).

Now we will write a method that takes two arguments, the register index and the value, and adds them. Notice how we use `halfcarry_add_table` and `overflow_add_table` tables:

39 ⟨*Members* 21c⟩+≡ (17a) ◁38 41b▷

```
member x.add_to_register reg v =
    let r = register.[reg]
    let res, carry = x.add8_carry r v
    let cf = if carry then C_FLAG else 0uy
    let lookup = ((r &&& 0x88uy) >>> 3) ||| ((v &&& 0x88uy) >>> 2) ||| ((res &&& 0x88uy) >>> 1)
    register.[F] <- sz53n_add.[(int)res] ||| cf
    register.[F] <- register.[F] |||
        if halfcarry_add_table.[(int)(lookup &&& 0x07uy)] then H_FLAG else 0uy
    register.[F] <- register.[F] |||
        if overflow_add_table.[(int)(lookup >>> 4)] then V_FLAG else 0uy
    register.[reg] <- res

member x.sub_from_register reg v =
    let r = register.[reg]
    let res, carry = x.sub8_carry r v
    let cf = if carry then C_FLAG else 0uy
    register.[F] <- sz53n_sub.[int res] ||| cf
    let lookup = ((r &&& 0x88uy) >>> 3) ||| ((v &&& 0x88uy) >>> 2) ||| ((res &&& 0x88uy) >>> 1)
    register.[F] <- register.[F] |||
        if halfcarry_sub_table.[int (lookup &&& 0x07uy)] then H_FLAG else 0uy
    register.[F] <- register.[F] |||
        if overflow_sub_table.[int (lookup >>> 4)] then V_FLAG else 0uy
    register.[reg] <- res
```

Defines:
`add_to_register`, used in chunks 40a, 52, 53, 58a, and 77b.
`sub_from_register`, used in chunks 40a, 52, 53, 58a, 63a, and 77b.

Uses `add8_carry` 38, `overflow_add_table` 20b, `overflow_sub_table` 20b, `sub8_carry` 38, `sz53n_add` 19a, and `sz53n_sub` 19a.

Now we simply implement the required instruction as follows:

40a ⟨*Main Instructions* 35a⟩+≡ (34c) ◁37c 40b▷
```
  operations.[0x27] <- fun () ->
      let cf = register.[F] &&& C_FLAG
      let correction = if (register.[F] &&& C_FLAG <> 0uy) || (register.[A] > 0x99uy)
                          then 0x60uy else 0x00uy
      let correction = correction |||
                          (if ((register.[F] &&& H_FLAG <> 0uy) ||
                                  ((register.[A] &&& 0x0Fuy) > 0x09uy))
                                      then 0x06uy else 0x00uy)
      let cf = if register.[A] > 0x99uy then C_FLAG else cf
      if register.[F] &&& N_FLAG <> 0uy then
          x.sub_from_register A correction
          register.[F] <- sz53pn_sub.[int register.[A]] ||| (register.[F] &&& H_FLAG) ||| cf
      else
          x.add_to_register A correction
          register.[F] <- sz53pn_add.[int register.[A]] ||| (register.[F] &&& H_FLAG) ||| cf
```

Uses `add_to_register` 39, `sub_from_register` 39, `sz53pn_add` 19c, and `sz53pn_sub` 19c.

LD HL,(**) Loads the value pointed by `(**)` to `HL`:

40b ⟨*Main Instructions* 35a⟩+≡ (34c) ◁40a 40c▷
```
  operations.[0x2A] <- fun () ->
      let offset = x.read_address_pointing_by_pc
      register.[L] <- x.read_from_memory offset
      register.[H] <- x.read_from_memory (offset + 1us)
```

Uses `read_address_pointing_by_pc` 37b and `read_from_memory` 30a.

CPL Contents of `A` are inverted. Note that flags `N` and `H` are set, others are unaffected:

40c ⟨*Main Instructions* 35a⟩+≡ (34c) ◁40b 40d▷
```
  operations.[0x2F] <- fun () ->
      register.[A] <- register.[A] ^^^ MASK_8
      register.[F] <- (register.[F] &&& SZP_FLAGS) |||
              H_FLAG |||
              (register.[A] &&& X53_FLAGS) |||
              N_FLAG |||
              (register.[F] &&& C_FLAG)
```

LD SP, ** `SP` ← `**`.

40d ⟨*Main Instructions* 35a⟩+≡ (34c) ◁40c 41a▷
```
  operations.[0x31] <- fun () ->
      sp <- x.read_address_pointing_by_pc
```

Uses `read_address_pointing_by_pc` 37b.

LD (**), A Stores A into the memory location pointed by (**)

41a ⟨*Main Instructions* 35a⟩+≡ (34c) ◁40d 41c▷
```
  operations.[0x32] <- fun () ->
      x.write_to_memory x.read_address_pointing_by_pc register.[A]
```
Uses read_address_pointing_by_pc 37b and write_to_memory 30a.

LD (HL),* Loads * into (HL) We need a helper method:

41b ⟨*Members* 21c⟩+≡ (17a) ◁39 47b▷
```
  member x.get_register16 a b = x.get_compound register.[a] register.[b]
```
Defines:
get_register16, used in chunks 41c, 44–46, 48–51, 53, 56b, 59–62, 64–69, and 73–76.

Then,

41c ⟨*Main Instructions* 35a⟩+≡ (34c) ◁41a 41d▷
```
  operations.[0x36] <- fun () ->
      x.write_to_memory (x.get_register16 H L) (x.read_from_memory pc)
      pc <- pc + 1us
```
Uses get_register16 41b, read_from_memory 30a, and write_to_memory 30a.

SCF Sets the carry flag.

41d ⟨*Main Instructions* 35a⟩+≡ (34c) ◁41c 41e▷
```
  operations.[0x37] <- fun () ->
      register.[F] <- (register.[F] &&& SZP_FLAGS) |||
                      (register.[A] &&& X53_FLAGS) ||| C_FLAG
```

LD A,(**) Loads the value pointed by ** to A.

41e ⟨*Main Instructions* 35a⟩+≡ (34c) ◁41d 41f▷
```
  operations.[0x3A] <- fun () ->
      let offset = x.read_address_pointing_by_pc
      register.[A] <- x.read_from_memory offset
```
Uses read_address_pointing_by_pc 37b and read_from_memory 30a.

CCF Inverts the carry flag.

41f ⟨*Main Instructions* 35a⟩+≡ (34c) ◁41e 42a▷
```
  operations.[0x3F] <- fun () ->
      register.[F] <- (register.[F] &&& SZP_FLAGS) |||
                      (register.[A] &&& X53_FLAGS) |||
                      (if (register.[F] &&& C_FLAG) <> 0uy then H_FLAG else C_FLAG)
```

HALT Suspends CPU until interrupt or reset.

42a ⟨*Main Instructions* 35a⟩+≡ (34c) ◁41f 42b▷
```
  operations.[0x76] <- fun () ->
      halt <- true
      pc <- pc - 1us
```

JP ** Jumps to `**` by setting `PC` to the value pointing by `PC`.

42b ⟨*Main Instructions* 35a⟩+≡ (34c) ◁42a 42c▷
```
  operations.[0xC3] <- fun () ->
      pc <- x.read_address_pointing_by_pc
```

Uses `read_address_pointing_by_pc` 37b.

RET The top stack entry is popped into `PC`.

42c ⟨*Main Instructions* 35a⟩+≡ (34c) ◁42b 42d▷
```
  operations.[0xC9] <- fun () ->
      pc <- x.pop_from_stack
```

Uses `pop_from_stack` 30d.

0xCB prefixed instructions This is a lookup into another table, which we will implemented a bit later. But we can set up a function anyway:

42d ⟨*Main Instructions* 35a⟩+≡ (34c) ◁42c 42e▷
```
  operations.[0xCB] <- fun () ->
      x.increment_memory_refresh_register
      let op = int x.read_next_instruction
      pc <- pc + 1us
      operations_bits.[op]()
```

Uses `read_next_instruction` 32c.

CALL ** Pushes `PC` onto stack and sets `PC` to `**`.

42e ⟨*Main Instructions* 35a⟩+≡ (34c) ◁42d 43a▷
```
  operations.[0xCD] <- fun () ->
      let offset = x.read_address_pointing_by_pc
      x.check_memory_contention_m (pc - 1us) 1
      x.push_to_stack pc
      pc <- offset
```

Uses `check_memory_contention_m` 24b, `push_to_stack` 30d, and `read_address_pointing_by_pc` 37b.

OUT (*), A Writes `A` to port `*`.

43a ⟨*Main Instructions* 35a⟩+≡ (34c) ◃42e 43b▹
```
  operations.[0xD3] <- fun () ->
      let value = x.read_from_memory pc
      x.write_to_port (x.get_compound register.[A] value) register.[A]
      pc <- pc + 1us
```

Uses `read_from_memory` 30a and `write_to_port` 27.

EXX Exchanges `BC`, `DE`, `HL` with their shadow counterparts.

43b ⟨*Main Instructions* 35a⟩+≡ (34c) ◃43a 43c▹
```
  operations.[0xD9] <- fun () ->
      x.exchange_registers B BS
      x.exchange_registers C CS
      x.exchange_registers D DS
      x.exchange_registers E ES
      x.exchange_registers H HS
      x.exchange_registers L LS
```

Uses `exchange_registers` 35c.

IN A, (*) Read from port `*` and write to `A`.

43c ⟨*Main Instructions* 35a⟩+≡ (34c) ◃43b 43d▹
```
  operations.[0xDB] <- fun () ->
      register.[A] <- x.read_from_port (x.get_compound register.[A] (x.read_from_memory pc))
      pc <- pc + 1us
```

Uses `read_from_memory` 30a.

0xDD prefixed instructions This is a lookup into `0xDD` prefixed table.

43d ⟨*Main Instructions* 35a⟩+≡ (34c) ◃43c 44a▹
```
  operations.[0xDD] <- fun () ->
      x.increment_memory_refresh_register
      let op = int x.read_next_instruction
      pc <- pc + 1us
      operations_indexes.[op] IX
```

Uses `read_next_instruction` 32c.

EX (SP), HL `(SP)` ↔ `L`, `(SP+1)` ↔ `H`.

44a ⟨*Main Instructions* 35a⟩+≡ (34c) ◁43d 44b▷
```
  operations.[0xE3] <- fun () ->
      let curr_h = register.[H]
      let curr_l = register.[L]
      register.[L] <- x.read_from_memory sp
      register.[H] <- x.read_from_memory (sp + 1us)
      x.check_memory_contention_m (sp + 1us) 1
      x.write_to_memory sp curr_l
      x.write_to_memory (sp + 1us) curr_h
      x.check_memory_contention_m sp 2
```

Uses `check_memory_contention_m` 24b, `read_from_memory` 30a, and `write_to_memory` 30a.

JP (HL) `PC` ← `(HL)`.

44b ⟨*Main Instructions* 35a⟩+≡ (34c) ◁44a 44c▷
```
  operations.[0xE9] <- fun () ->
      pc <- x.get_register16 H L
```

Uses `get_register16` 41b.

EX DE, HL `DE` ↔ `HL`.

44c ⟨*Main Instructions* 35a⟩+≡ (34c) ◁44b 44d▷
```
  operations.[0xEB] <- fun () ->
      x.exchange_registers D H
      x.exchange_registers E L
```

Uses `exchange_registers` 35c.

0xED prefixed instructions This is a lookup into `0xDD` prefixed table.

44d ⟨*Main Instructions* 35a⟩+≡ (34c) ◁44c 44e▷
```
  operations.[0xED] <- fun () ->
      x.increment_memory_refresh_register
      let op = int x.read_next_instruction
      pc <- pc + 1us
      operations_extended.[op]()
```

Uses `read_next_instruction` 32c.

DI Resets flip-flop flags.

44e ⟨*Main Instructions* 35a⟩+≡ (34c) ◁44d 45a▷
```
  operations.[0xF3] <- fun () ->
      iff1 <- false
      iff2 <- false
```

LD SP, HL `SP` ← `HL`.

45a ⟨*Main Instructions* 35a⟩+≡ (34c) ◁44e 45b▷

```
  operations.[0xF9] <- fun () ->
      x.check_memory_contention_m ir 2
      sp <- x.get_register16 H L
```

Uses `check_memory_contention_m` 24b and `get_register16` 41b.

EI Sets both flip-flop flags.

45b ⟨*Main Instructions* 35a⟩+≡ (34c) ◁45a 45c▷

```
  operations.[0xFB] <- fun () ->
      ei <- true
      iff1 <- true
      iff2 <- true
```

0xFD prefixed instructions This is a lookup into `0xFD` prefixed table.

45c ⟨*Main Instructions* 35a⟩+≡ (34c) ◁45b 45d▷

```
  operations.[0xFD] <- fun () ->
      x.increment_memory_refresh_register
      let op = int x.read_next_instruction
      pc <- pc + 1us
      operations_indexes.[op] IY
```

Uses `read_next_instruction` 32c.

LD ⟨BC|DE|HL⟩, ** Loads `**` into a corresponding register. We will implement these instructions initialization in one go. First, we will create an array of tuples where the first component is an instruction byte and the second is an array of 8-bit registers. Then we iterate on the tuples, read memory pointing by `PC`, set to the 8-bit register, and move onto next byte to set another half of the register.

45d ⟨*Main Instructions* 35a⟩+≡ (34c) ◁45c 46a▷

```
  [(0x01, [B; C]); (0x11, [D; E]); (0x21, [H; L])] |> List.iter
      (fun c ->
          match c with
          | (op, reg) ->
              operations.[op] <- fun () ->
                  reg |> List.rev |> List.iter (fun reg_half ->
                      register.[reg_half] <- x.read_from_memory pc
                      pc <- pc + 1us))
```

Uses `read_from_memory` 30a.

We will use this technique in many other places.

LD ⟨(BC)|(DE)⟩, A The idea of implementation is similar to the previous one, although here we write `A` into memory location pointed by the 16-bit register.

46a ⟨*Main Instructions* 35a⟩+≡ (34c) ◁45d 46b▷

```
[(0x02, [B; C]); (0x12, [D; E])] |> List.iter
    (fun c ->
        match c with
        | (op, [reg1; reg2]) ->
            operations.[op] <- fun () ->
                x.write_to_memory (x.get_register16 reg1 reg2) register.[A]
        | (_, _) -> ()
    )
```

Uses `get_register16` 41b and `write_to_memory` 30a.

INC/DEC SP Increments/decrements `SP`.

46b ⟨*Main Instructions* 35a⟩+≡ (34c) ◁46a 46c▷

```
[(0x33, +1); (0x3B, -1)] |> List.iter (fun c ->
    match c with
    | (op, offset) -> operations.[op] <- fun () ->
                    x.check_memory_contention_m ir 2
                    sp <- uint16 (int sp + offset))
```

Uses `check_memory_contention_m` 24b.

INC/DEC ⟨BC|DE|HL⟩ Increments/decrements corresponding registers.

46c ⟨*Main Instructions* 35a⟩+≡ (34c) ◁46b 48a▷

```
[
    (0x03, [B;C], +1)
    (0x13, [D;E], +1)
    (0x23, [H;L], +1)
    (0x0B, [B;C], -1)
    (0x1B, [D;E], -1)
    (0x2B, [H;L], -1)
] |> List.iter (fun c ->
    match c with
    | (op, [reg1; reg2], offset) ->
            operations.[op] <- fun () ->
                x.check_memory_contention_m ir 2
                let r = uint16 (int (x.get_register16 reg1 reg2) + offset)
                register.[reg1] <- byte (r >>> 8)
                register.[reg2] <- byte r
    | (_, _, _) -> ()
)
```

Uses `check_memory_contention_m` 24b and `get_register16` 41b.

Before implementing next instruction set, let's define a couple of helper method and useful constants.

First, an array of indexes of registers:

47a ⟨*Registers set* 47a⟩≡ (17a)
```
  let regs = [B; C; D; E; H; L; A]
```

Defines:
`regs`, used in chunks 48, 51b, and 59.

Then, a couple of helpers:

47b ⟨*Members* 21c⟩+≡ (17a) ◁41b 48c▷
```
  member x.inc_with_flags8 value =
      let value = value + 1uy
      register.[F] <- sz53n_add.[int value] ||| (register.[F] &&& C_FLAG)
      register.[F] <- register.[F] ||| if (value &&& 0x0Fuy) = 0uy then H_FLAG else 0uy
      register.[F] <- register.[F] ||| if (value = 0x80uy) then V_FLAG else 0uy
      value

  member x.dec_with_flags8 value =
      let value = value - 1uy
      register.[F] <- sz53n_sub.[int value] ||| (register.[F] &&& C_FLAG)
      register.[F] <- register.[F] ||| if ((value &&& 0x0Fuy) = 0x0Fuy) then H_FLAG else 0uy
      register.[F] <- register.[F] ||| if (value = 0x7Fuy) then V_FLAG else 0uy
      value

  member x.inc_reg_8 reg =
      let value = register.[reg]
      let value = x.inc_with_flags8 value
      register.[reg] <- value

  member x.dec_reg_8 reg =
      let value = register.[reg]
      let value = x.dec_with_flags8 value
      register.[reg] <- value
```

Defines:
`dec_reg_8`, used in chunk 48a.
`dec_with_flags8`, used in chunks 50a and 79a.
`inc_reg_8`, used in chunk 48a.
`inc_with_flags8`, used in chunks 50a and 79a.
Uses `sz53n_add` 19a and `sz53n_sub` 19a.

INC/DEC ⟨B|C|D|E|H|L⟩ Increments/decrements corresponding registers.

48a ⟨*Main Instructions* 35a⟩+≡ (34c) ◁46c 48b▷
```
  let get_ops_with_regs ops regs = List.zip ops regs
  let get_ops ops regs f = (get_ops_with_regs ops regs) |> List.map (fun (a, b) -> (a, b, f))
  let ops1 = get_ops (List.append [0x04 .. 0x08 .. 0x2C] [0x3C]) regs x.inc_reg_8
  let ops2 = get_ops (List.append [0x05 .. 0x08 .. 0x2D] [0x3D]) regs x.dec_reg_8
  List.append ops1 ops2 |> List.iter (fun c ->
      match c with
      | (op, reg, f) ->
          operations.[op] <- fun () -> f reg
  )
```

Uses `dec_reg_8` 47b, `inc_reg_8` 47b, and `regs` 47a.

LD ⟨B|C|D|E|H|L|A⟩, * Loads ∗ into corresponding register.

48b ⟨*Main Instructions* 35a⟩+≡ (34c) ◁48a 49a▷
```
  get_ops_with_regs (List.append [0x06 .. 0x08 .. 0x2E] [0x3E]) regs |> List.iter (fun c ->
      match c with
      | (op, reg) ->
          operations.[op] <- fun () ->
              register.[reg] <- x.read_from_memory pc
              pc <- pc + 1us
  )
```

Uses `read_from_memory` 30a and `regs` 47a.

ADD HL, ⟨BC|DE|HL|SP⟩, * Loads ∗ into corresponding register.

Let's have a couple of helper methods:

48c ⟨*Members* 21c⟩+≡ (17a) ◁47b 52▷
```
  member x.add_to_register16 reg1 reg2 v =
      let a = x.get_register16 reg1 reg2
      let sum = x.add_to_register16_val a v
      register.[reg1] <- byte (sum >>> 8)
      register.[reg2] <- byte sum

  member x.add_to_register16_val a v =
      let value = int a + int v
      register.[F] <- (register.[F] &&& SZP_FLAGS) |||
                      ((byte (value >>> 8)) &&& X53_FLAGS) |||
                      (if value > 0xFFFF then C_FLAG else 0uy)
      let value = uint16 value;
      if (value &&& 0x0FFFus) < (a &&& 0x0FFFus) then
          register.[F] <- register.[F] ||| H_FLAG
      value
```

Defines:
`add_to_register16`, used in chunk 49a.
`add_to_register16_val`, used in chunk 76.
Uses `get_register16` 41b.

49a ⟨*Main Instructions* 35a⟩+≡ (34c) ◁48b 49b▷

```
  [(0x09, [B;C]); (0x19, [D;E]); (0x29, [H;L]); (0x39, [])] |> List.iter (fun c ->
      match c with
      | (op, [reg1; reg2]) ->
          operations.[op] <- fun () ->
              x.check_memory_contention_m ir 7
              x.add_to_register16 H L (x.get_register16 reg1 reg2)
      | (op, _) ->
          operations.[op] <- fun () ->
              x.check_memory_contention_m ir 7
              x.add_to_register16 H L sp
  )
```

Uses `add_to_register16` 48c, `check_memory_contention_m` 24b, and `get_register16` 41b.

JR ⟨NZ|NC|Z|C⟩, * If condition is true, then signed value of `*` is added to `PC`.

49b ⟨*Main Instructions* 35a⟩+≡ (34c) ◁49a 50a▷

```
  [
      (0x20, (Z_FLAG, false))
      (0x30, (C_FLAG, false))
      (0x28, (Z_FLAG, true))
      (0x38, (C_FLAG, true))
  ] |> List.iter (fun c->
      match c with
      | (op, (flag, exp_flag_value)) ->
          operations.[op] <- fun () ->
              let offset = x.get_signed_byte (x.read_from_memory pc)
              let flag_value = register.[F] &&& flag > 0uy
              if flag_value = exp_flag_value then
                  x.check_memory_contention_m pc 5
                  pc <- uint16 (int pc + offset + 1)
              else
                  pc <- pc + 1us
  )
```

Uses `check_memory_contention_m` 24b, `get_signed_byte` 35f, and `read_from_memory` 30a.

INC/DEC (HL) Increments/decrements (`HL`).

50a ⟨*Main Instructions* 35a⟩+≡ (34c) ◁49b 50b▷
```
[
    (0x34, x.inc_with_flags8)
    (0x35, x.dec_with_flags8)
] |> List.iter (fun c ->
    match c with
    | (op, f) ->
        operations.[op] <- fun () ->
            let hl = x.get_register16 H L
            let value = f (x.read_from_memory hl)
            x.check_memory_contention_m hl 1
            x.write_to_memory hl value
)
```

Uses `check_memory_contention_m` 24b, `dec_with_flags8` 47b, `get_register16` 41b, `inc_with_flags8` 47b, `read_from_memory` 30a, and `write_to_memory` 30a.

LD A, ⟨BC|DE⟩ Loads the value pointed by the corresponding register into `A`.

50b ⟨*Main Instructions* 35a⟩+≡ (34c) ◁50a 51a▷
```
[
    (0x0A, [B;C])
    (0x1A, [D;E])
] |> List.iter (fun c ->
    match c with
    |(op, [reg1; reg2]) ->
        operations.[op] <- fun () ->
            register.[A] <- x.read_from_memory (x.get_register16 reg1 reg2)
    |(_, _) -> ()
)
```

Uses `get_register16` 41b and `read_from_memory` 30a.

LD ⟨B|C|D|E|H|L|A⟩, ⟨B|C|D|E|H|L|(HL)|A⟩ Load from corresponding register to register.

51a ⟨*Main Instructions* 35a⟩+≡ (34c) ◁50b 51b▷

```
[
    ([0x40..0x47], B)
    ([0x48..0x4F], C)
    ([0x50..0x57], D)
    ([0x58..0x5F], E)
    ([0x60..0x67], H)
    ([0x68..0x6F], L)
    ([0x78..0x7F], A)
] |> List.iter (fun c ->
    match c with
    | (ops, dest_reg) ->
        List.zip ops [B;C;D;E;H;L;-1;A] |> List.iter (fun y ->
            match y with
            | (op, src_reg) ->
                operations.[op] <-
                 if src_reg <> -1 then
                    fun () -> register.[dest_reg] <- register.[src_reg]
                 else
                    fun () -> register.[dest_reg] <- x.read_from_memory (x.get_register16 H L)
        )
)
```

Uses `get_register16` 41b and `read_from_memory` 30a.

LD (HL), ⟨B|C|D|E|H|L|A⟩ Loads contents of the corresponding registers to (HL).

51b ⟨*Main Instructions* 35a⟩+≡ (34c) ◁51a 53▷

```
List.zip (List.append [0x70..0x75] [0x77]) regs |> List.iter (fun c ->
    match c with
    |(op, src_reg) ->
        operations.[op] <- fun () ->
            x.write_to_memory (x.get_register16 H L) register.[src_reg]
)
```

Uses `get_register16` 41b, `regs` 47a, and `write_to_memory` 30a.

Arithmetic and logical operations This is going to be the largest cluster of instructions:

- ADD A, $\langle B|C|D|E|H|L|(HL)|A\rangle$
- ADC A, $\langle B|C|D|E|H|L|(HL)|A\rangle$
- SUB $\langle B|C|D|E|H|L|(HL)|A\rangle$
- SBC A, $\langle B|C|D|E|H|L|(HL)|A\rangle$
- AND $\langle B|C|D|E|H|L|(HL)|A\rangle$
- XOR $\langle B|C|D|E|H|L|(HL)|A\rangle$
- OR $\langle B|C|D|E|H|L|(HL)|A\rangle$
- CP $\langle B|C|D|E|H|L|(HL)|A\rangle$

First, some helpers:

52 ⟨*Members* 21c⟩+≡ (17a) ◁48c 58b▷

```
member x.add_to_register_with_carry reg v =
    x.add_to_register reg (v + (if register.[F] &&& C_FLAG <> 0uy then 1uy else 0uy))

member x.sub_from_register_with_carry reg v =
    x.sub_from_register reg (v + (if register.[F] &&& C_FLAG <> 0uy then 1uy else 0uy))

member x.xor_register reg v =
    register.[reg] <- register.[reg] ^^^ v
    register.[F] <- sz53pn_add.[int register.[reg]]

member x.or_register reg v =
    register.[reg] <- register.[reg] ||| v
    register.[F] <- sz53pn_add.[int register.[reg]]

member x.and_register reg v =
    register.[reg] <- register.[reg] &&& v
    register.[F] <- sz53pn_add.[int register.[reg]] ||| H_FLAG

member x.cp_to_register reg v =
    let r = register.[reg]
    let res, carry = x.sub8_carry r v
    let cf = if carry then C_FLAG else 0uy
    register.[F] <- (sz53n_sub.[int res] &&& SZHN_FLAGS) |||
                    (sz53n_add.[int v] &&& X53_FLAGS) |||
                    cf
    let lookup = ((r &&& 0x88uy) >>> 3) |||
                  ((v &&& 0x88uy) >>> 2) |||
                  ((res &&& 0x88uy) >>> 1)
    register.[F] <- register.[F] |||
                    (if halfcarry_sub_table.[int (lookup &&& 0x07uy)] then H_FLAG else 0uy)
    register.[F] <- register.[F] |||
                    (if overflow_sub_table.[int (lookup >>> 4)] then V_FLAG else 0uy)
```

Defines:
`add_to_register_with_carry`, used in chunks 53, 58a, and 77b.

`and_register`, used in chunks 53, 58a, and 77b.
`cp_to_register`, used in chunks 53, 58a, 67, and 77b.
`or_register`, used in chunks 53, 58a, and 77b.
`sub_from_register_with_carry`, used in chunks 53, 58a, and 77b.

Uses `add_to_register` 39, `overflow_sub_table` 20b, `sub8_carry` 38, `sub_from_register` 39, `sz53n_add` 19a, `sz53n_sub` 19a, and `sz53pn_add` 19c.

Now the instructions:

53 ⟨*Main Instructions* 35a⟩+≡ (34c) ◁51b 54a▷

```
[
    ([0x80..0x87], x.add_to_register)
    ([0x88..0x8F], x.add_to_register_with_carry)
    ([0x90..0x97], x.sub_from_register)
    ([0x98..0x9F], x.sub_from_register_with_carry)
    ([0xa0..0xA7], x.and_register)
    ([0xa8..0xAF], x.xor_register)
    ([0xb0..0xB7], x.or_register)
    ([0xb8..0xBF], x.cp_to_register)
] |> List.iter (fun c ->
    match c with
    | (ops, f) ->
       List.zip ops [B;C;D;E;H;L;-1;A] |> List.iter (fun y ->
            match y with
            | (op, src_reg) ->
                operations.[op] <-
                 if src_reg <> -1 then
                    fun () -> f A register.[src_reg]
                 else
                    fun () -> f A (x.read_from_memory (x.get_register16 H L))
       )
)
```

Uses `add_to_register` 39, `add_to_register_with_carry` 52, `and_register` 52, `cp_to_register` 52, `get_register16` 41b, `or_register` 52, `read_from_memory` 30a, `sub_from_register` 39, and `sub_from_register_with_carry` 52.

RET ⟨NZ|NC|NP|NS|Z|C|P|S⟩ If condition is true, pops top of the stack entry into `PC`.

54a ⟨*Main Instructions* 35a⟩+≡ (34c) ◁53 54b▷

```
[
    (0xC0, (Z_FLAG, false))
    (0xD0, (C_FLAG, false))
    (0xE0, (P_FLAG, false))
    (0xF0, (S_FLAG, false))
    (0xC8, (Z_FLAG, true))
    (0xD8, (C_FLAG, true))
    (0xE8, (P_FLAG, true))
    (0xF8, (S_FLAG, true))
] |> List.iter (fun c ->
    match c with
    | (op, (flag, exp_flag_value)) ->
        operations.[op] <- fun () ->
            x.check_memory_contention_m ir 1
            let flag_value = register.[F] &&& flag > 0uy
            if flag_value = exp_flag_value then
                pc <- x.pop_from_stack
)
```

Uses `check_memory_contention_m` 24b and `pop_from_stack` 30d.

POP ⟨BC|DE|HL|AF⟩ The value is popped from the stack and stored into a corresponding register.

54b ⟨*Main Instructions* 35a⟩+≡ (34c) ◁54a 55▷

```
[
    (0xC1,[B;C])
    (0xD1,[D;E])
    (0xE1,[H;L])
    (0xF1,[A;F])
] |> List.iter (fun c ->
    match c with
    |(op, [reg1;reg2]) ->
        operations.[op] <- fun () ->
            let v = x.pop_from_stack
            register.[reg1] <- byte (v >>> 8)
            register.[reg2] <- byte v
    | (_, _) -> ()
)
```

Uses `pop_from_stack` 30d.

JP ⟨NZ|NC|NP|NS|Z|C|P|S⟩ If condition is true, `**` is copied into `PC`.

55 ⟨*Main Instructions* 35a⟩+≡ (34c) ◁54b 56a▷

```
[
    (0xC2,(Z_FLAG, false))
    (0xD2,(C_FLAG, false))
    (0xE2,(P_FLAG, false))
    (0xF2,(S_FLAG, false))
    (0xCA,(Z_FLAG, true))
    (0xDA,(C_FLAG, true))
    (0xEA,(P_FLAG, true))
    (0xFA,(S_FLAG, true))
] |> List.iter (fun c ->
    match c with
    | (op, (flag, exp_flag_value)) ->
        operations.[op] <- fun () ->
            let offset1 = x.read_from_memory pc
            pc <- pc + 1us
            let offset2 = x.read_from_memory pc
            pc <- pc + 1us
            let offset = x.get_compound offset2 offset1
            let flag_value = register.[F] &&& flag > 0uy
            if flag_value = exp_flag_value then
                pc <- offset
)
```

Uses `read_from_memory` 30a.

CALL ⟨NZ|NC|NP|NS|Z|C|P|S⟩, ** If condition is true, `**` is current `PC` value plus 3 is pushed onto the stack, then `PC` is set with `**`.

56a ⟨*Main Instructions* 35a⟩+≡ (34c) ◁55 56b▷

```
  [
    (0xC4,(Z_FLAG, false))
    (0xD4,(C_FLAG, false))
    (0xE4,(P_FLAG, false))
    (0xF4,(S_FLAG, false))
    (0xCC,(Z_FLAG, true))
    (0xDC,(C_FLAG, true))
    (0xEC,(P_FLAG, true))
    (0xFC,(S_FLAG, true))
  ] |> List.iter (fun c ->
      match c with
      | (op, (flag, exp_flag_value)) ->
          operations.[op] <- fun () ->
              let offset1 = x.read_from_memory pc
              let offset2 = x.read_from_memory (pc + 1us)
              let offset = x.get_compound offset2 offset1
              let flag_value = register.[F] &&& flag > 0uy
              if flag_value = exp_flag_value then
                  pc <- pc + 1us
                  x.check_memory_contention_m pc 1
                  pc <- pc + 1us
                  x.push_to_stack pc
                  pc <- offset
              else
                  pc <- pc + 2us
  )
```

Uses `check_memory_contention_m` 24b, `push_to_stack` 30d, and `read_from_memory` 30a.

PUSH ⟨BC|DE|HL|AF⟩ Pushes the value of corresponding register onto the stack.

56b ⟨*Main Instructions* 35a⟩+≡ (34c) ◁56a 57▷

```
  [
      (0xC5, [B;C])
      (0xD5, [D;E])
      (0xE5, [H;L])
      (0xF5, [A;F])
  ] |> List.iter (fun c ->
      match c with
      | (op, [reg1;reg2]) ->
          operations.[op] <- fun () ->
              x.check_memory_contention_m ir 1
              x.push_to_stack (x.get_register16 reg1 reg2)
      | (_, _) -> ()
  )
```

Uses `check_memory_contention_m` 24b, `get_register16` 41b, and `push_to_stack` 30d.

RST ⟨0x00|0x10|0x20|0x30|0x08|0x18|0x28|0x38⟩ Pushes current `PC` onto the stack, and set `PC` to the corresponding value.

57 ⟨*Main Instructions* 35a⟩+≡ (34c) ◁56b 58a▷

```
  [
      (0xC7,0x00us)
      (0xD7,0x10us)
      (0xE7,0x20us)
      (0xF7,0x30us)
      (0xCF,0x08us)
      (0xDF,0x18us)
      (0xEF,0x28us)
      (0xFF,0x38us)
  ] |> List.iter (fun c->
      match c with
      | (op, addr) ->
          operations.[op] <- fun () ->
              x.check_memory_contention_m ir 1
              x.push_to_stack pc
              pc <- addr
  )
```

Uses `check_memory_contention_m` 24b and `push_to_stack` 30d.

Arithmetic and logical operations Performs logical and arithmetic operation with `*` and register `A`.

- ADD A, *
- ADC A, *
- SUB *
- SBC A, *
- AND *
- XOR *
- OR *
- CP *

58a ⟨*Main Instructions* 35a⟩+≡ (34c) ◁57

```
[
    (0xC6, x.add_to_register)
    (0xD6, x.sub_from_register)
    (0xE6, x.and_register)
    (0xF6, x.or_register)
    (0xCE, x.add_to_register_with_carry)
    (0xDE, x.sub_from_register_with_carry)
    (0xEE, x.xor_register)
    (0xFE, x.cp_to_register)
] |> List.iter (fun c->
    match c with
    | (op, f) ->
        operations.[op] <- fun () ->
            f A (x.read_from_memory pc)
            pc <- pc + 1us
)
```

Uses `add_to_register` 39, `add_to_register_with_carry` 52, `and_register` 52, `cp_to_register` 52, `or_register` 52, `read_from_memory` 30a, `sub_from_register` 39, and `sub_from_register_with_carry` 52.

This completes main instruction table.

5.9 Extended Instructions

Now we will initialize extended instructions table.

58b ⟨*Members* 21c⟩+≡ (17a) ◁52 70▷

```
member x.initializeExtendedOperations =
    ⟨Extended Instructions 59a⟩
```

Defines:
`initializeExtendedOperations`, used in chunk 82b.

IN ⟨B|C|D|E|H|L⟩, (C) `A` byte from port `C` is written to a corresponding register.

59a ⟨*Extended Instructions* 59a⟩≡ (58b) 59b ▷

```
List.zip [0x40; 0x48; 0x50; 0x58; 0x60; 0x68; 0x78] regs |> List.iter (fun c ->
    match c with
    | (op, reg) ->
        operations_extended.[op] <- fun () ->
            register.[reg] <- x.read_from_port (x.get_register16 B C)
            register.[F] <- sz53pn_add.[int register.[B]] ||| (register.[F] &&& C_FLAG)
)
```

Uses `get_register16` 41b, `regs` 47a, and `sz53pn_add` 19c.

OUT (C), ⟨B|C|D|E|H|L⟩ `A` value in `C` is written to a port specified in a corresponding register.

59b ⟨*Extended Instructions* 59a⟩+≡ (58b) ◁ 59a 60 ▷

```
List.zip [0x41; 0x49; 0x51; 0x59; 0x61; 0x69; 0x79] regs |> List.iter (fun c ->
    match c with
    | (op, reg) ->
        operations_extended.[op] <- fun () ->
            x.write_to_port (x.get_register16 B C) register.[reg]
)
```

Uses `get_register16` 41b, `regs` 47a, and `write_to_port` 27.

SBC HL, ⟨BC|DE|HL|SP⟩ Subtracts a corresponding register and the carry flag from HL.

60 ⟨*Extended Instructions* 59a⟩+≡ (58b) ◁59b 61▷

```
  [
      (0x42,[B;C])
      (0x52,[D;E])
      (0x62,[H;L])
      (0x72,[-1;-1])
  ] |> List.iter (fun c ->
      match c with
      |(op, [reg1;reg2]) ->
          operations_extended.[op] <- fun () ->
              x.check_memory_contention_m ir 7
              let hl_value = x.get_register16 H L
              let reg_value = if reg1=(-1) then sp else x.get_register16 reg1 reg2
              let res = (int hl_value - int reg_value - int (register.[F] &&& C_FLAG))
              let cf = if res < 0 then C_FLAG else 0uy
              let res = uint16 res
              register.[H] <- byte (res >>> 8)
              register.[L] <- byte res
              register.[F] <- sz53n_sub.[int register.[H]] ||| cf
              if res <> 0us then
                  register.[F] <- register.[F] &&& (Z_FLAG ^^^ 0xFFuy)

              let lookup = ((hl_value &&& 0x8800us) >>> 11 ) |||
                           (((reg_value) &&& 0x8800us) >>> 10 ) |||
                           ((res &&& 0x8800us) >>>  9 )

              register.[F] <- register.[F] |||
                  (if halfcarry_sub_table.[int (lookup &&& 0x07us)] then H_FLAG else 0uy)
              register.[F] <- register.[F] |||
                  (if overflow_sub_table.[int (lookup >>> 4)] then V_FLAG else 0uy)
      |(_, _) -> ()
  )
```

Uses check_memory_contention_m 24b, get_register16 41b, overflow_sub_table 20b, and sz53n_sub 19a.

ADC HL, ⟨BC|DE|HL|SP⟩ Adds a corresponding register and the carry flag from `HL`.

61 ⟨*Extended Instructions* 59a⟩+≡ (58b) ◁60 62a▷

```
  [
      (0x4A,[B;C])
      (0x5A,[D;E])
      (0x6A,[H;L])
      (0x7A,[-1;-1])
  ] |> List.iter (fun c ->
      match c with
      |(op, [reg1;reg2]) ->
          operations_extended.[op] <- fun () ->
              x.check_memory_contention_m ir 7
              let hl_value = x.get_register16 H L
              let reg_value = if reg1=(-1) then sp else x.get_register16 reg1 reg2
              let res = (int hl_value + int reg_value + int (register.[F] &&& C_FLAG))
              let cf = if res > 0xFFFF then C_FLAG else 0uy
              let res = uint16 res
              register.[H] <- byte (res >>> 8)
              register.[L] <- byte res
              register.[F] <- sz53n_add.[int register.[H]] ||| cf
              if res <> 0us then
                  register.[F] <- register.[F] &&& (Z_FLAG ^^^ 0xFFuy)

              let lookup = ((hl_value &&& 0x8800us) >>> 11 ) |||
                            (((reg_value) &&& 0x8800us) >>> 10 ) |||
                            ((res &&& 0x8800us) >>>  9 )

              register.[F] <- register.[F] |||
                  (if halfcarry_add_table.[int (lookup &&& 0x07us)] then H_FLAG else 0uy)
              register.[F] <- register.[F] |||
                  (if overflow_add_table.[int (lookup >>> 4)] then V_FLAG else 0uy)
      |(_, _) -> ()
  )
```

Uses `check_memory_contention_m` 24b, `get_register16` 41b, `overflow_add_table` 20b, and `sz53n_add` 19a.

LD (**), ⟨BC|DE|HL|SP⟩ Stores a corresponding register into the memory location pointed by `**`.

62a ⟨*Extended Instructions* 59a⟩+≡ (58b) ◁61 62b▷

```
[
    (0x43,[B;C])
    (0x53,[D;E])
    (0x63,[H;L])
    (0x73,[-1;-1])
] |> List.iter (fun c ->
    match c with
    |(op, [reg1;reg2]) ->
        operations_extended.[op] <- fun () ->
            let offset = x.read_address_pointing_by_pc
            let value = if reg1 = (-1) then sp else x.get_register16 reg1 reg2
            x.write_to_memory offset (byte value)
            x.write_to_memory (offset + 1us) (byte (value >>> 8))
    |(_,_) -> ()
)
```

Uses `get_register16` 41b, `read_address_pointing_by_pc` 37b, and `write_to_memory` 30a.

LD ⟨BC|DE|HL|SP⟩, (**) Stores the value pointed to by `**` into a corresponding register.

62b ⟨*Extended Instructions* 59a⟩+≡ (58b) ◁62a 63a▷

```
[
    (0x4B,[B;C])
    (0x5B,[D;E])
    (0x6B,[H;L])
    (0x7B,[-1;-1])
] |> List.iter (fun c ->
    match c with
    |(op, [reg1;reg2]) ->
        operations_extended.[op] <- fun () ->
            let offset = x.read_address_pointing_by_pc
            let val1 = x.read_from_memory offset
            let val2 = x.read_from_memory (offset + 1us)
            if reg1 = (-1) then
                sp <- x.get_compound val2 val1
            else
                register.[reg1] <- val2
                register.[reg2] <- val1
    |(_,_) -> ()
)
```

Uses `read_address_pointing_by_pc` 37b and `read_from_memory` 30a.

NEG Contents of `A` are negated.

63a ⟨*Extended Instructions* 59a⟩+≡ (58b) ◁62b 63b▷
```
  operations_extended.[0x44] <- fun () ->
      let v = register.[A]
      register.[A] <- 0uy
      x.sub_from_register A v
```

Uses `sub_from_register` 39.

RETN/RETI Copies `IFF2` into `IFF1`, and pops the entry from the stack and assigns it to `PC`.

63b ⟨*Extended Instructions* 59a⟩+≡ (58b) ◁63a 63c▷
```
  [0x45; 0x55; 0x65; 0x75; 0x4D; 0x5D; 0x6D; 0x7D] |> List.iter (fun op ->
      operations_extended.[op] <- fun () ->
          iff1 <- iff2
          pc <- x.pop_from_stack
  )
```

Uses `pop_from_stack` 30d.

IM ⟨0|1|2⟩ Sets corresponding interrupt mode.

63c ⟨*Extended Instructions* 59a⟩+≡ (58b) ◁63b 63d▷
```
  [
      (0x46,0)
      (0x56,1)
      (0x66,0)
      (0x76,1)
      (0x5E,2)
      (0x7E,2)
  ] |> List.iter (fun c ->
      match c with
      | (op, v) ->
          operations_extended.[op] <- fun () ->
              im <- v
  )
```

LD I, A Stores `A` into `I`.

63d ⟨*Extended Instructions* 59a⟩+≡ (58b) ◁63c 64a▷
```
  operations_extended.[0x47] <- fun () ->
      x.check_memory_contention_m ir 1
      ir <- ((uint16 register.[A]) <<< 8) ||| (ir &&& 0x00FFus)
```

Uses `check_memory_contention_m` 24b.

LD R, A Stores `A` into `R`.

64a ⟨*Extended Instructions* 59a⟩+≡ (58b) ◁63d 64b▷

```
  operations_extended.[0x4F] <- fun () ->
      x.check_memory_contention_m ir 1
      ir <- (ir &&& 0xFF00us) ||| ((uint16 register.[A]) &&& 0x00FFus)
```

Uses `check_memory_contention_m` 24b.

LD A, I Stores `I` into `A`.

64b ⟨*Extended Instructions* 59a⟩+≡ (58b) ◁64a 64c▷

```
  operations_extended.[0x57] <- fun () ->
      x.check_memory_contention_m ir 1
      register.[A] <- byte ((ir &&& 0xFF00us) >>> 8)
      register.[F] <- sz53n_add.[int register.[A]] ||| (register.[F] &&& C_FLAG)
      if iff2 then
          register.[F] <- register.[F] ||| P_FLAG
```

Uses `check_memory_contention_m` 24b and `sz53n_add` 19a.

LD A, R Stores `R` into `A`.

64c ⟨*Extended Instructions* 59a⟩+≡ (58b) ◁64b 64d▷

```
  operations_extended.[0x5F] <- fun () ->
      x.check_memory_contention_m ir 1
      register.[A] <- byte ir
      register.[F] <- sz53n_add.[int register.[A]] ||| (register.[F] &&& C_FLAG)
      if iff2 then
          register.[F] <- register.[F] ||| P_FLAG
```

Uses `check_memory_contention_m` 24b and `sz53n_add` 19a.

RRD According to [2]: The contents of the low-order four bits (bits 3, 2, 1, and 0) of memory location (`HL`) are copied to the low-order four bits of the Accumulator (Register `A`). The previous contents of the low-order four bits of the Accumulator are copied to the high-order four bits (7, 6, 5, and 4) of location (`HL`); and the previous contents of the high-order four bits of (`HL`) are copied to the low-order four bits of (`HL`). The contents of the high-order bits of the Accumulator are unaffected.

64d ⟨*Extended Instructions* 59a⟩+≡ (58b) ◁64c 65▷

```
  operations_extended.[0x67] <- fun () ->
      let a4 = register.[A] &&& 0x0Fuy
      let hl = x.get_register16 H L
      let hl8 = x.read_from_memory hl
      let hl4 = hl8 &&& 0x0Fuy
      register.[A] <- (register.[A] &&& 0xF0uy) ||| hl4
      x.check_memory_contention_m hl 4
      x.write_to_memory hl ((hl8 >>> 4) ||| (a4 <<< 4))
      register.[F] <- sz53pn_add.[int register.[A]] ||| (register.[F] &&& C_FLAG)
```

Uses `check_memory_contention_m` 24b, `get_register16` 41b, `read_from_memory` 30a, `sz53pn_add` 19c, and `write_to_memory` 30a.

RLD According to [2]: The contents of the low-order four bits (bits 3, 2, 1, and 0) of the memory location (`HL`) are copied to the high-order four bits (7, 6, 5, and 4) of that same memory location; the previous contents of those high-order four bits are copied to the low-order four bits of the Accumulator (Register `A`); and the previous contents of the low-order four bits of the Accumulator are copied to the low-order four bits of memory location (`HL`). The contents of the high-order bits of the Accumulator are unaffected.

65 ⟨*Extended Instructions* 59a⟩+≡ (58b) ◁64d 66▷

```
operations_extended.[0x6F] <- fun () ->
    let a4 = register.[A] &&& 0x0Fuy
    let hl = x.get_register16 H L
    let hl8 = x.read_from_memory hl
    let hl4 = (hl8 &&& 0xF0uy) >>> 4
    register.[A] <- (register.[A] &&& 0xF0uy) ||| hl4
    x.check_memory_contention_m hl 4
    x.write_to_memory hl (((hl8 <<< 4) &&& MASK_8) ||| a4)
    register.[F] <- sz53pn_add.[int register.[A]] ||| (register.[F] &&& C_FLAG)
```

Uses `check_memory_contention_m` 24b, `get_register16` 41b, `read_from_memory` 30a, `sz53pn_add` 19c, and `write_to_memory` 30a.

LDI/LDIR/LDD/LDDR See [2] for description.

66 ⟨*Extended Instructions* 59a⟩+≡ (58b) ◁65 67▷
```
[
    (0xA0, 1, 0)
    (0xB0, 1, -1)
    (0xA8, -1, 0)
    (0xB8, -1, 1)
] |> List.iter (fun c ->
    match c with
    |(op, incr, p) ->
        let f = fun () ->
            let hl = x.get_register16 H L
            let value = x.read_from_memory hl
            let de = x.get_register16 D E
            x.write_to_memory de value
            x.check_memory_contention_m de 2
            let hl = uint16 (int hl + incr)
            register.[H] <- byte (hl >>> 8)
            register.[L] <- byte hl

            let de = uint16 (int de + incr)
            register.[D] <- byte (de >>> 8)
            register.[E] <- byte de

            let bc = x.get_register16 B C
            let bc = bc - 1us
            register.[B] <- byte (bc >>> 8)
            register.[C] <- byte bc

            register.[F] <- (register.[F] &&& SZ_FLAGS) |||
                    (value &&& X3_FLAG) |||
                    (register.[F] &&& C_FLAG)
            if value &&& N_FLAG <> 0uy then
                register.[F] <- register.[F] ||| X5_FLAG
            if bc <> 0us then
                register.[F] <- register.[F] ||| P_FLAG
        operations_extended.[op] <-
            if p <> 0 then
                fun () ->
                    f()
                    if (register.[F] &&& P_FLAG <> 0uy) then
                        pc <- (pc - 2us)
                        let de = x.get_register16 D E
                        x.check_memory_contention_m (uint16 (int de + p)) 5
            else
                fun () -> f()
)
```

Uses `check_memory_contention_m` 24b, `get_register16` 41b, `read_from_memory` 30a, and `write_to_memory` 30a.

CPI/CPIR/CPD/CPDR See [2] for description.

67 ⟨*Extended Instructions* 59a⟩+≡ (58b) ◁66 68▷

```
[
    (0xA1, 1, 0)
    (0xB1, 1, -1)
    (0xA9, -1, 0)
    (0xB9, -1, 1)
] |> List.iter (fun c ->
    match c with
    |(op, incr, p) ->
        let f = fun () ->
            let hl = x.get_register16 H L
            let value = x.read_from_memory hl
            let cf = register.[F] &&& C_FLAG
            x.cp_to_register A value
            x.check_memory_contention_m hl 5
            let hl = uint16 (int hl + incr)
            register.[H] <- byte (hl >>> 8)
            register.[L] <- byte hl

            let bc = x.get_register16 B C
            let bc = bc - 1us
            register.[B] <- byte (bc >>> 8)
            register.[C] <- byte bc

            let value = byte (int register.[A] - int value -
                            int (if register.[F] &&& H_FLAG <> 0uy
                                 then 1uy else 0uy))
            register.[F] <- (register.[F] &&& SZHN_FLAGS) |||
                            (value &&& X3_FLAG) ||| cf
            if value &&& N_FLAG <> 0uy then
                register.[F] <- register.[F] ||| X5_FLAG
            if bc <> 0us then
                register.[F] <- register.[F] ||| P_FLAG
        operations_extended.[op] <-
            if p <> 0 then
                fun () ->
                    f()
                    if (register.[F] &&& P_FLAG <> 0uy) &&
                       (register.[F] &&& Z_FLAG = 0uy) then
                         pc <- (pc - 2us)
                         let hl = x.get_register16 H L
                         x.check_memory_contention_m (uint16 (int hl + p)) 5
            else
                fun () -> f()
)
```

Uses `check_memory_contention_m` 24b, `cp_to_register` 52, `get_register16` 41b, and `read_from_memory` 30a.

INI/INIR/IND/INDR See [2] for description.

68 ⟨*Extended Instructions* 59a⟩+≡ (58b) ◁67 69▷
```
  [
      (0xA2, 1, 0)
      (0xB2, 1, -1)
      (0xAA, -1, 0)
      (0xBA, -1, 1)
  ] |> List.iter (fun c ->
      match c with
      |(op, incr, p) ->
          let f = fun () ->
              x.check_memory_contention_m ir 1
              let bc = x.get_register16 B C
              let value = x.read_from_port bc
              let hl = x.get_register16 H L
              x.write_to_memory hl value
              register.[B] <- (register.[B] - 1uy)
              let hl = uint16 (int hl + incr)
              register.[H] <- byte (hl >>> 8)
              register.[L] <- byte hl

              register.[F] <- sz53pn_add.[int register.[B]]
              if (value > 0x7Fuy) then
                  register.[F] <- register.[F] ||| N_FLAG
              let value = value + (register.[C] + 1uy)
              if (value > 0xFFuy) then
                  register.[F] <- register.[F] ||| H_FLAG ||| C_FLAG

              if ((sz53pn_add.[int ((value &&& 0x07uy) ^^^ register.[B])] &&& P_FLAG) <> 0uy) then
                  register.[F] <- register.[F] ||| P_FLAG
              else
                  register.[F] <- register.[F] &&& (P_FLAG ^^^ 0xFFuy)
          operations_extended.[op] <-
              if p <> 0 then
                  fun () ->
                      f()
                      if register.[B] <> 0uy then
                          pc <- (pc - 2us)
                          let hl = x.get_register16 H L
                          x.check_memory_contention_m (uint16 (int hl + p)) 5
              else
                  fun () -> f()
  )
```

Uses `check_memory_contention_m` 24b, `get_register16` 41b, `sz53pn_add` 19c, and `write_to_memory` 30a.

OUTI/OTIR/OUTD/OTDR See [2] for description.

69 ⟨*Extended Instructions* 59a⟩+≡ (58b) ◁68

```
[
    (0xA3, 1, 0)
    (0xB3, 1, 1)
    (0xAB, -1, 0)
    (0xBB, -1, 1)
] |> List.iter (fun c ->
    match c with
    |(op, incr, p) ->
        let f = fun () ->
            x.check_memory_contention_m ir 1
            register.[B] <- register.[B] - 1uy
            let hl = x.get_register16 H L
            let value = x.read_from_memory hl
            let bc = x.get_register16 B C
            x.write_to_port bc value
            let hl = uint16 (int hl + incr)
            register.[H] <- byte (hl >>> 8)
            register.[L] <- byte hl

            register.[F] <- if value > 0x7Fuy then
                                sz53n_sub.[int register.[B]]
                                else sz53n_add.[int register.[B]]
            if (register.[L] + value) > 0xFFuy then
                register.[F] <- register.[F] ||| H_FLAG ||| C_FLAG
            if (sz53pn_add.[int (((register.[L] + value) &&& 0x07uy) ^^^
                               register.[B])] &&& P_FLAG) <> 0uy then
                    register.[F] <- register.[F] ||| P_FLAG

        operations_extended.[op] <-
            if p <> 0 then
                fun () ->
                    f()
                    if register.[B] <> 0uy then
                        pc <- (pc - 2us)
                        let bc = x.get_register16 B C
                        x.check_memory_contention_m bc 5
            else
                fun () -> f()

)
```

Uses `check_memory_contention_m` 24b, `get_register16` 41b, `read_from_memory` 30a, `sz53n_add` 19a, `sz53n_sub` 19a, `sz53pn_add` 19c, and `write_to_port` 27.

This completes extended instructions table.

5.10 Bit Instructions Table

Now we are going to implement bit operations.

70 ⟨*Members* 21c⟩+≡ (17a) ◁58b 71▷

```
  member x.initializeBitOperations =
        ⟨Bit Instructions 73⟩
```

Defines:
  `initializeBitOperations`, used in chunk 82b.

We will need a few helper methods, such as rotations and shifts to the left and right with and without carry flags.

So let's get started.

71 ⟨*Members* 21c⟩+≡ (17a) ◁70 72▷

```
  member x.rotate_left_carry value =
      let cf = (value >>> 7)
      let value = (value <<< 1) ||| cf
      register.[F] <- sz53pn_add.[int value] ||| cf
      value

  member x.rotate_right_carry value =
      let cf = value &&& 0x01uy
      let value = (value >>> 1) ||| (cf <<< 7)
      register.[F] <- sz53pn_add.[int value] ||| cf
      value

  member x.rotate_left value =
      let cf = value >>> 7
      let value = (value <<< 1) ||| (register.[F] &&& C_FLAG)
      register.[F] <- sz53pn_add.[int value] ||| cf
      value

  member x.rotate_right value =
      let cf = value &&& 0x01uy
      let value = (value >>> 1) ||| ((register.[F] &&& C_FLAG) <<< 7)
      register.[F] <- sz53pn_add.[int value] ||| cf
      value

  member x.shift_left value =
      let cf = value >>> 7
      let value = (value <<< 1)
      register.[F] <- sz53pn_add.[int value] ||| cf
      value

  member x.shift_right value =
      let cf = value &&& 0x01uy
      let value = (value &&& 0x80uy) ||| (value >>> 1)
      register.[F] <- sz53pn_add.[int value] ||| cf
      value

  member x.shift_right_logical value =
      let cf = value &&& 0x01uy
      let value = value >>> 1
      register.[F] <- sz53pn_add.[int value] ||| cf
      value
```

Defines:
- `rotate_left`, used in chunks 73 and 81a.
- `rotate_left_carry`, used in chunks 73 and 81a.
- `rotate_right`, used in chunks 73 and 81a.
- `rotate_right_carry`, used in chunks 73 and 81a.
- `shift_left`, used in chunks 73 and 81a.
- `shift_right`, used in chunks 73 and 81a.

`shift_right_logical`, used in chunks 73 and 81a.
Uses `sz53pn_add` 19c.

Also, let's write some helpers for testing, setting and resetting bits.

72 ⟨*Members* 21c⟩+≡ (17a) ◁71 75b▷

```
member x.test_bit (mask: byte) value =
    let zf = (mask &&& value) = 0uy
    register.[F] <- (sz53n_add.[int value] &&& (SZP_FLAGS ^^^ 0xFFuy)) |||
                        H_FLAG ||| (register.[F] &&& C_FLAG)
    if zf then
        register.[F] <- register.[F] ||| P_FLAG ||| Z_FLAG
    if (mask = 0x80uy) && (not zf) then
        register.[F] <- register.[F] ||| S_FLAG

member x.reset_bit b value = value &&& b
member x.set_bit b value = value ||| b
```

Defines:
`reset_bit`, used in chunks 75a and 82a.
`set_bit`, used in chunks 75a and 82a.
`test_bit`, used in chunks 74 and 81b.
Uses `sz53n_add` 19a.

Rotations and Shifts This is a large cluster of instructions: Rotations with and without the carry flag, shifts to the right and left, plus right shift logical.

73 ⟨*Bit Instructions* 73⟩≡ (70) 74▷

```
[
    ([0x00..0x07], x.rotate_left_carry)
    ([0x08..0x0F], x.rotate_right_carry)
    ([0x10..0x17], x.rotate_left)
    ([0x18..0x1F], x.rotate_right)
    ([0x20..0x27], x.shift_left)
    ([0x28..0x2F], x.shift_right)
    ([0x38..0x3F], x.shift_right_logical)
] |> List.iter (fun c ->
    match c with
    | (ops, f) ->
       List.zip ops [B;C;D;E;H;L;-1;A] |> List.iter (fun y ->
            match y with
            | (op, reg) ->
                operations_bits.[op] <-
                 if reg <> -1 then
                    fun () -> register.[reg] <- f register.[reg]
                 else
                    fun () ->
                        let hl_value = x.get_register16 H L
                        let res = f (x.read_from_memory hl_value)
                        x.check_memory_contention_m hl_value 1
                        x.write_to_memory hl_value res
       )
)
```

Uses `check_memory_contention_m` 24b, `get_register16` 41b, `read_from_memory` 30a, `rotate_left` 71, `rotate_left_carry` 71, `rotate_right` 71, `rotate_right_carry` 71, `shift_left` 71, `shift_right` 71, `shift_right_logical` 71, and `write_to_memory` 30a.

Bit Testing This another cluster of instruction: Testing for bits.

74 ⟨*Bit Instructions* 73⟩+≡ (70) ◁73 75a▷
```
  [
      ([0x40..0x47], x.test_bit, 0b00000001uy)
      ([0x48..0x4F], x.test_bit, 0b00000010uy)
      ([0x50..0x57], x.test_bit, 0b00000100uy)
      ([0x58..0x5F], x.test_bit, 0b00001000uy)
      ([0x60..0x67], x.test_bit, 0b00010000uy)
      ([0x68..0x6F], x.test_bit, 0b00100000uy)
      ([0x70..0x77], x.test_bit, 0b01000000uy)
      ([0x78..0x7F], x.test_bit, 0b10000000uy)
  ] |> List.iter (fun c ->
      match c with
      | (ops, f, mask) ->
          List.zip ops [B;C;D;E;H;L;-1;A] |> List.iter (fun y ->
              match y with
              | (op, reg) ->
                 operations_bits.[op] <-
                   if reg <> -1 then
                      fun () -> f mask register.[reg]
                   else
                      fun () ->
                          let hl_value = x.get_register16 H L
                          f mask (x.read_from_memory hl_value)
                          x.check_memory_contention_m hl_value 1
          )
  )
```

Uses `check_memory_contention_m` 24b, `get_register16` 41b, `read_from_memory` 30a, and `test_bit` 72.

Bit Setting and Resetting This cluster of instructions is for setting and resetting bits.

75a ⟨*Bit Instructions* 73⟩+≡ (70) ◁74

```
[
    ([0x80..0x87], x.reset_bit, 0b11111110uy)
    ([0x88..0x8F], x.reset_bit, 0b11111101uy)
    ([0x90..0x97], x.reset_bit, 0b11111011uy)
    ([0x98..0x9F], x.reset_bit, 0b11110111uy)
    ([0xA0..0xA7], x.reset_bit, 0b11101111uy)
    ([0xA8..0xAF], x.reset_bit, 0b11011111uy)
    ([0xB0..0xB7], x.reset_bit, 0b10111111uy)
    ([0xB8..0xBF], x.reset_bit, 0b01111111uy)
    ([0xC0..0xC7], x.set_bit, 0b00000001uy)
    ([0xC8..0xCF], x.set_bit, 0b00000010uy)
    ([0xD0..0xD7], x.set_bit, 0b00000100uy)
    ([0xD8..0xDF], x.set_bit, 0b00001000uy)
    ([0xE0..0xE7], x.set_bit, 0b00010000uy)
    ([0xE8..0xEF], x.set_bit, 0b00100000uy)
    ([0xF0..0xF7], x.set_bit, 0b01000000uy)
    ([0xF8..0xFF], x.set_bit, 0b10000000uy)
] |> List.iter (fun c ->
    match c with
    | (ops, f, mask) ->
        List.zip ops [B;C;D;E;H;L;-1;A] |> List.iter (fun y ->
            match y with
            | (op, reg) ->
               operations_bits.[op] <-
                 if reg <> -1 then
                    fun () -> register.[reg] <- f mask register.[reg]
                 else
                    fun () ->
                        let hl_value = x.get_register16 H L
                        let res = f mask (x.read_from_memory hl_value)
                        x.check_memory_contention_m hl_value 1
                        x.write_to_memory hl_value res
        )
)
```

Uses `check_memory_contention_m` 24b, `get_register16` 41b, `read_from_memory` 30a, `reset_bit` 72, `set_bit` 72, and `write_to_memory` 30a.

OK, that's it for bit operations. Let's move on to index instructions.

5.11 Index Instructions

Now to the index instruction.

75b ⟨*Members* 21c⟩+≡ (17a) ◁72 80e▷

```
member x.initializeIndexOperations =
    ⟨Index Instructions 76a⟩
```

Defines:
`initializeIndexOperations`, used in chunk 82b.

ADD ⟨IX|IY⟩, ⟨IX|IY⟩

76a ⟨*Index Instructions* 76a⟩≡ (75b) 76b ▷

```
  operations_indexes.[0x29] <- fun (ireg) ->
      x.check_memory_contention_m ir 7
      index_register.[ireg] <- x.add_to_register16_val index_register.[ireg] index_register.[ireg]
```

Uses add_to_register16_val 48c and check_memory_contention_m 24b.

ADD ⟨IX|IY⟩, SP

76b ⟨*Index Instructions* 76a⟩+≡ (75b) ◁ 76a 76c ▷

```
  operations_indexes.[0x39] <- fun (ireg) ->
      x.check_memory_contention_m ir 7
      index_register.[ireg] <- x.add_to_register16_val index_register.[ireg] sp
```

Uses add_to_register16_val 48c and check_memory_contention_m 24b.

ADD ⟨IX|IY⟩, ⟨BC|DE⟩

76c ⟨*Index Instructions* 76a⟩+≡ (75b) ◁ 76b 77a ▷

```
  [
      (0x09, [B;C])
      (0x19, [D;E])
  ] |> List.iter (fun c ->
      match c with
      | (op, [reg1;reg2]) ->
          operations_indexes.[op] <- fun (ireg) ->
              x.check_memory_contention_m ir 7
              index_register.[ireg] <-
                  x.add_to_register16_val index_register.[ireg] (x.get_register16 reg1 reg2)
      | (_, _) -> ()
  )
```

Uses add_to_register16_val 48c, check_memory_contention_m 24b, and get_register16 41b.

Load to/from register The following cluster of instructions would either load into a register value pointed by index register plus offset, or vice versa.

77a ⟨*Index Instructions* 76a⟩+≡ (75b) ◁76c 77b▷

```
let f = fun ireg g ->
    let offset = x.get_signed_byte (x.read_from_memory pc)
    x.check_memory_contention_m pc 5
    let address = uint16 (int index_register.[ireg] + offset)
    g address
    pc <- pc + 1us

[
    (0x46, B, 0);(0x4E, C, 0);(0x56, D, 0);(0x5E, E, 0);(0x66, H, 0);(0x6E, L, 0);(0x7E, A, 0);
    (0x70, B, 1);(0x71, C, 1);(0x72, D, 1);(0x73, E, 1);(0x74, H, 1);(0x75, L, 1);(0x77, A, 1);
] |> List.iter (fun c ->
    match c with
    | (op, reg, p) ->
        operations_indexes.[op] <-
            if p = 0 then
                fun (ireg) -> f ireg (fun (address) -> register.[reg] <- x.read_from_memory address)
            else
                fun (ireg) -> f ireg (fun (address) -> x.write_to_memory address register.[reg])
)
```

Uses check_memory_contention_m 24b, get_signed_byte 35f, read_from_memory 30a, and write_to_memory 30a.

Arithmetic and logical instructions

77b ⟨*Index Instructions* 76a⟩+≡ (75b) ◁77a 77c▷

```
[
    (0x86, x.add_to_register)
    (0x96, x.sub_from_register)
    (0xA6, x.and_register)
    (0xB6, x.or_register)
    (0x8E, x.add_to_register_with_carry)
    (0x9E, x.sub_from_register_with_carry)
    (0xAE, x.xor_register)
    (0xBE, x.cp_to_register)
] |> List.iter (fun c ->
    match c with
    | (op, g) ->
        operations_indexes.[op] <- fun (ireg) ->
           f ireg (fun address -> g A (x.read_from_memory address))
)
```

Uses add_to_register 39, add_to_register_with_carry 52, and_register 52, cp_to_register 52, or_register 52, read_from_memory 30a, sub_from_register 39, and sub_from_register_with_carry 52.

LD ⟨IX|IY⟩, **

77c ⟨*Index Instructions* 76a⟩+≡ (75b) ◁77b 78a▷

```
operations_indexes.[0x21] <- fun ireg -> index_register.[ireg] <- x.read_address_pointing_by_pc
```

Uses read_address_pointing_by_pc 37b.

LD (**), ⟨IX|IY⟩

78a ⟨*Index Instructions* 76a⟩+≡ (75b) ◁77c 78b▷

```
operations_indexes.[0x22] <- fun ireg ->
    let address = x.read_address_pointing_by_pc
    x.write_to_memory address (byte index_register.[ireg])
    x.write_to_memory (address + 1us) (byte (index_register.[ireg] >>> 8))
```

Uses read_address_pointing_by_pc 37b and write_to_memory 30a.

INC/DEC ⟨IX|IY⟩

78b ⟨*Index Instructions* 76a⟩+≡ (75b) ◁78a 78c▷

```
[
    (0x23, 1)
    (0x2B, -1)
] |> List.iter (fun c ->
    match c with
    | (op, p) ->
        operations_indexes.[op] <- fun (ireg) ->
           x.check_memory_contention_m ir 2
           index_register.[ireg] <- uint16 (int index_register.[ireg] + p)
)
```

Uses check_memory_contention_m 24b.

LD ⟨IX|IY⟩, (**)

78c ⟨*Index Instructions* 76a⟩+≡ (75b) ◁78b 79a▷

```
operations_indexes.[0x2A] <- fun ireg ->
    let address = x.read_address_pointing_by_pc
    let v1 = x.read_from_memory address
    let v2 = x.read_from_memory (address + 1us)
    index_register.[ireg] <- x.get_compound v2 v1
```

Uses read_address_pointing_by_pc 37b and read_from_memory 30a.

INC (⟨IX|IY⟩+*)

79a ⟨*Index Instructions* 76a⟩+≡ (75b) ◁78c 79b▷

```
  [
      (0x34, x.inc_with_flags8)
      (0x35, x.dec_with_flags8)
  ] |> List.iter (fun c ->
      match c with
      | (op, f) ->
          operations_indexes.[op] <- fun (ireg) ->
              let offset = x.get_signed_byte (x.read_from_memory pc)
              x.check_memory_contention_m pc 5
              let address = uint16 (int index_register.[ireg] + offset)
              let value = x.read_from_memory address
              x.check_memory_contention_m address 1
              x.write_to_memory address (f value)
              pc <- pc + 1us
  )
```

Uses check_memory_contention_m 24b, dec_with_flags8 47b, get_signed_byte 35f, inc_with_flags8 47b, read_from_memory 30a, and write_to_memory 30a.

LD (⟨IX|IY⟩+*), *

79b ⟨*Index Instructions* 76a⟩+≡ (75b) ◁79a 79c▷

```
  operations_indexes.[0x36] <- fun ireg ->
      let offset = x.get_signed_byte (x.read_from_memory pc)
      let address = uint16 (int index_register.[ireg] + offset)
      pc <- pc + 1us
      let value = x.read_from_memory pc
      x.check_memory_contention_m pc 2
      x.write_to_memory address value
      pc <- pc + 1us
```

Uses check_memory_contention_m 24b, get_signed_byte 35f, read_from_memory 30a, and write_to_memory 30a.

POP ⟨IX|IY⟩

79c ⟨*Index Instructions* 76a⟩+≡ (75b) ◁79b 79d▷

```
  operations_indexes.[0xE1] <- fun ireg ->
      index_register.[ireg] <- x.pop_from_stack
```

Uses pop_from_stack 30d.

PUSH ⟨IX|IY⟩

79d ⟨*Index Instructions* 76a⟩+≡ (75b) ◁79c 80a▷

```
  operations_indexes.[0xE5] <- fun ireg ->
      x.check_memory_contention_m ir 1
      x.push_to_stack index_register.[ireg]
```

Uses check_memory_contention_m 24b and push_to_stack 30d.

JP (⟨IX|IY⟩)

80a ⟨*Index Instructions* 76a⟩+≡ (75b) ◁79d 80b▷
```
  operations_indexes.[0xE9] <- fun ireg ->
      pc <- index_register.[ireg]
```

LD SP, ⟨IX|IY⟩

80b ⟨*Index Instructions* 76a⟩+≡ (75b) ◁80a 80c▷
```
  operations_indexes.[0xF9] <- fun ireg ->
      x.check_memory_contention_m ir 2
      sp <- index_register.[ireg]
```

Uses `check_memory_contention_m` 24b.

EX (SP), ⟨IX|IY⟩

80c ⟨*Index Instructions* 76a⟩+≡ (75b) ◁80b 80d▷
```
  operations_indexes.[0xE3] <- fun ireg ->
      let value = index_register.[ireg]
      let v1 = x.read_from_memory sp
      let v2 = x.read_from_memory (sp + 1us)
      index_register.[ireg] <- x.get_compound v2 v1
      x.check_memory_contention_m (sp + 1us) 1
      x.write_to_memory (sp + 1us) (byte (value >>> 8))
      x.write_to_memory sp (byte value)
      x.check_memory_contention_m sp 2
```

Uses `check_memory_contention_m` 24b, `read_from_memory` 30a, and `write_to_memory` 30a.

0xCB prefixed operations

80d ⟨*Index Instructions* 76a⟩+≡ (75b) ◁80c
```
  operations_indexes.[0xCB] <- fun ireg ->
      let offset = x.get_signed_byte (x.read_from_memory pc)
      let address = uint16 (int index_register.[ireg] + offset)
      pc <- pc + 1us
      let next_op = x.read_from_memory pc
      x.check_memory_contention_m pc 2
      pc <- pc + 1us
      operations_indexes_bits.[int next_op] address (x.read_from_memory address)
```

Uses `check_memory_contention_m` 24b, `get_signed_byte` 35f, and `read_from_memory` 30a.

Finally, all we need to implement now is index bit operations. Those are straightforward.

5.12 Index Bit Instructions

80e ⟨*Members* 21c⟩+≡ (17a) ◁75b
```
  member x.initializeIndexBitOperations =
      ⟨Index Bit Instructions 81a⟩
```

Defines:
  `initializeIndexBitOperations`, used in chunk 82b.

Rotations and shifts

81a ⟨*Index Bit Instructions* 81a⟩≡ (80e) 81b ▷

```
[
    (0x06, x.rotate_left_carry)
    (0x0E, x.rotate_right_carry)
    (0x16, x.rotate_left)
    (0x1E, x.rotate_right)
    (0x26, x.shift_left)
    (0x2E, x.shift_right)
    (0x3E, x.shift_right_logical)
] |> List.iter (fun c ->
    match c with
    | (op, f) ->
        operations_indexes_bits.[op] <- fun address value ->
            let res = f value
            x.check_memory_contention_m address 1
            x.write_to_memory address res
)
```

Uses `check_memory_contention_m` 24b, `rotate_left` 71, `rotate_left_carry` 71, `rotate_right` 71, `rotate_right_carry` 71, `shift_left` 71, `shift_right` 71, `shift_right_logical` 71, and `write_to_memory` 30a.

BIT [0..7], (⟨IX|IY⟩ + *)

81b ⟨*Index Bit Instructions* 81a⟩+≡ (80e) ◁ 81a 82a ▷

```
[
    (0x46,0b00000001uy)
    (0x4E,0b00000010uy)
    (0x56,0b00000100uy)
    (0x5E,0b00001000uy)
    (0x66,0b00010000uy)
    (0x6E,0b00100000uy)
    (0x76,0b01000000uy)
    (0x7E,0b10000000uy)
] |> List.iter (fun c ->
    match c with
    | (op, mask) ->
        operations_indexes_bits.[op] <- fun address value ->
            x.test_bit mask value
            x.check_memory_contention_m address 1
)
```

Uses `check_memory_contention_m` 24b and `test_bit` 72.

SET/RES [0..7], (⟨IX|IY⟩ + *)

82a ⟨*Index Bit Instructions* 81a⟩+≡ (80e) ◁81b

```
[
    (0x86, x.reset_bit,0b11111110uy)
    (0x8E, x.reset_bit,0b11111101uy)
    (0x96, x.reset_bit,0b11111011uy)
    (0x9E, x.reset_bit,0b11110111uy)
    (0xA6, x.reset_bit,0b11101111uy)
    (0xAE, x.reset_bit,0b11011111uy)
    (0xB6, x.reset_bit,0b10111111uy)
    (0xBE, x.reset_bit,0b01111111uy)
    (0xC6, x.set_bit,0b00000001uy)
    (0xCE, x.set_bit,0b00000010uy)
    (0xD6, x.set_bit,0b00000100uy)
    (0xDE, x.set_bit,0b00001000uy)
    (0xE6, x.set_bit,0b00010000uy)
    (0xEE, x.set_bit,0b00100000uy)
    (0xF6, x.set_bit,0b01000000uy)
    (0xFE, x.set_bit,0b10000000uy)
] |> List.iter (fun c ->
    match c with
    | (op, f, mask) ->
        operations_indexes_bits.[op] <- fun address value ->
            let v = f mask value
            x.check_memory_contention_m address 1
            x.write_to_memory address v
)
```

Uses `check_memory_contention_m` 24b, `reset_bit` 72, `set_bit` 72, and `write_to_memory` 30a.

5.13 Initialization

Now as we have written all the initialization methods, we can initialize our type:

82b ⟨*Type initialization* 82b⟩≡ (17a)

```
do
    this.initializeOperations
    this.initializeExtendedOperations
    this.initializeBitOperations
    this.initializeIndexOperations
    this.initializeIndexBitOperations
```

Uses `initializeBitOperations` 70, `initializeExtendedOperations` 58b, `initializeIndexBitOperations` 80e, `initializeIndexOperations` 75b, and `initializeOperations` 34c.

6 The Machine

Finally, we can put everything together. The machine is made of Z80 microprocessor and an input device keyboard. As parameters to the constructor we take a display and a keyboard. The original machine executed each frame in approximately 20ms. So, if the emulator executes it quicker, then it will sleep.

83 ⟨*Types* 2c⟩+≡ (1b) ◁17a 84▷

```
type Machine(display: IDisplay, keyboard: IPort) as this = class
    let mutable processor = new Z80(keyboard)

    let measure_execution_time f =
        let timer = new System.Diagnostics.Stopwatch()
        timer.Start()
        f()
        timer.Stop()
        timer.ElapsedMilliseconds

    let machine = async {
            processor <- new Z80(keyboard)
            processor.loadRom(
                    if (Environment.GetCommandLineArgs().Length > 1) then
                        Environment.GetCommandLineArgs().[1]
                    else
                        "48k.rom"
                )
            let! token = Async.CancellationToken
            while not token.IsCancellationRequested do
                let execution_time =
                    measure_execution_time
                        (fun () -> display.draw processor.execute_frame processor.Frames)
                let difference = 20L - execution_time
                if difference > 0L then
                    Thread.Sleep(int difference)
        }
    let mutable cancellationToken = new System.Threading.CancellationTokenSource()

    do
        this.Reset

    member x.Reset =
        cancellationToken.Cancel()
        cancellationToken <- new System.Threading.CancellationTokenSource()
        Async.StartAsTask(machine, cancellationToken=cancellationToken.Token) |> ignore

    member x.Stop =
       cancellationToken.Cancel()
end
```

Defines:
  `Machine`, used in chunk 84.
Uses `execute_frame` 34b and `loadRom` 21c.

And now the GUI form. We create a keyboard, a display and then initialize the machine with them.

84 ⟨*Types* 2c⟩+≡ (1b) ◁83

```
type Z80GUI() as this = class
    inherit Form(Width = 256 + 70, Height = 192 + 120, Text = "Z80")

    let formSize = new Size(Width = 256 + 70, Height = 192 + 120)
    let mainMenu = new MainMenu()
    let mnuFile = new MenuItem()
    let mnuSystem = new MenuItem()
    let mnuFileClose = new MenuItem()
    let mnuSystemReset = new MenuItem()

    let keyboard = new Keyboard()
    let display = new Display(new VideoMemoryProcessor())
    let machine = new Machine(display, keyboard)

    do
        this.MaximumSize <- formSize
        this.MinimumSize <- formSize

        this.BackColor <- Color.FromArgb(0xFF, 0xC0, 0xC0, 0xC0)
        mnuFile.Text <- "&File"
        mnuSystem.Text <- "&System"
        mnuFileClose.Text <- "&Close"
        mnuSystemReset.Text <- "&Reset"
        mnuSystemReset.Click.Add(fun e ->
            let r = MessageBox.Show(
                        text="Are you sure you want to reset the machine?",
                        caption="Reset",
                        buttons=MessageBoxButtons.YesNo)
            if r = DialogResult.Yes then machine.Reset)
        mnuFileClose.Click.Add(fun e -> machine.Stop; this.Close())

        mnuFile.MenuItems.AddRange([| mnuFileClose; |])
        mnuSystem.MenuItems.AddRange([| mnuSystemReset; |])

        mainMenu.MenuItems.AddRange([| mnuFile; mnuSystem |])
        this.Menu <- mainMenu

        this.KeyDown.Add(fun e -> (keyboard :> IKeyboard).KeyPressed e.KeyCode)
        this.KeyUp.Add(fun e -> (keyboard :> IKeyboard).KeyReleased e.KeyCode)
        display.Location <- new Point(25, 25)
        display.Dock <- DockStyle.None
        display.Size <- new Size(256, 192)
        this.Controls.Add(display)
end
```

Defines:
`Z80GUI`, used in chunk 85.
Uses `Machine` 83.

Application execution loop is now trivial:

85 ⟨*Application execution loop* 85⟩≡ (2a)
```
  do Application.Run(new Z80GUI())
```

Uses `Z80GUI` 84.

7 Notes

I used [3] and [4] as reference implementations when writing this program. A great source of help were [8], [5], [17]. I have also used tests from [3] to test my implementation but that's a different story. For someone who wants to write their own accurate emulator [17] is essential, also [8]. For how the Sinclair BASIC operating system is written see [10]. For introduction to instruction set of Z80 see [22]. For how to program in Sinclair BASIC see [23]. For those who are interested in literate programming, of course see [1].

8 License

Copyright©2016 Roman Valiušenko

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

References

[1] Donald E. Knuth, Literate Programming, The Computer Journal, 1984, pp 97–111

[2] Z80 Microprocessors, Z80 CPU User Manual, 2014

[3] FUSE emulator source code, http://fuse-emulator.sourceforge.net/

[4] ZX-Dart emulator source code, https://github.com/jcmellado/zx-dart. Retrieved 2016/03/28

[5] http://wordpress.animatez.co.uk/computers/zx-spectrum/screen-memory-layout/. Retrieved 2016/03/28

[6] https://en.wikipedia.org/wiki/ZX_Spectrum_graphic_modes. Retrieved 2016/03/28

[7] https://en.wikipedia.org/wiki/Zilog_Z80. Retrieved 2016/03/28

[8] http://www.worldofspectrum.org/faq/reference/48kreference.htm

[9] Expert F# 3.0, Don Syme, Adam Granicz, Antonio Cisternino, 2012

[10] The Complete SPECTRUM ROM Disassembly, Dr Ian Logan, Dr Frank O'Hara, 1983

[11] The CARRY flag and OVERFLOW flag in binary arithmetic, Ian! D. Allen

[12] http://www.zxdesign.info/memoryToScreen.shtml. Retrieved 2016/03/28

[13] http://faqwiki.zxnet.co.uk/wiki/Main_Page. Retrieved 2016/03/28

[14] http://faqwiki.zxnet.co.uk/wiki/Contended_memory#Instruction_breakdown. Retrieved 2016/03/28

[15] https://en.wikipedia.org/wiki/ZX_Spectrum_Contended_Memory. Retrieved 2016/03/28

[16] http://faqwiki.zxnet.co.uk/wiki/Contended_I/O, Retrieved on 2016/04/24

[17] The Undocumented Z80 Documented (Version 0.91, 18th September, 2005), Sean Young

[18] https://en.wikipedia.org/wiki/Binary-coded_decimal

[19] https://en.wikipedia.org/wiki/Nibble

[20] https://en.wikipedia.org/wiki/Sinclair_BASIC. Retrieved 2016/05/05

[21] https://en.wikipedia.org/wiki/ZX_Spectrum. Retrieved 2016/05/05

[22] http://www.z80.info/z80code.htm

[23] Sinclair ZX Spectrum BASIC Programming, Sinclair Research, Steven Vickers, 1982

Definitions

⟨*VMP Variables* 11c⟩ 10b, 11c

Index