



**Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)**

ФАКУЛЬТЕТ Информатика и системы управления (ИУ)

КАФЕДРА «Информационная безопасность» (ИУ8)

ПРОГРАММНЫЙ СИМУЛЯТОР RDP-11

Текст программы

А.В.00001-01 12 01

листов 55

Исполнитель, студент группы ИУ8-71

_____ Тимощук А.А.
«___» _____ 20__ г.

Исполнитель, студент группы ИУ8-71

_____ Шаповалов М. Е,
«___» _____ 20__ г.

Исполнитель, студент группы ИУ8-71

_____ Штырков В. С.
«___» _____ 20__ г.

Руководитель курсового проекта,
преподаватель кафедры ИУ8

_____ Рафиков А. Г.
«___» _____ 20__ г.

Заведующий кафедрой ИУ8

_____ Басараб М. А.
«___» _____ 20__ г.

Аннотация

В данном программном документе приведены исходные коды программы “Программный симулятор PDP-11”.

Содержание

Аннотация.....	2
Основная часть.....	4
1 Текст общих моделей.....	4
1.1 Текст модели проекта.....	4
2 Текст модуля Ассемблера.....	4
2.1 Текст класса Compiler.....	4
2.2 Текст класса Parser.....	5
2.3 Текст класса TokenBuilder.....	6
2.4 Текст класса CommandLine.....	16
2.5 Текст класса Token.....	16
3 Текст модуля Исполнителя.....	19
3.1 Текст класса Executor.....	19
3.2 Текст класса CommandParser.....	23
3.3 Текст классов аргументов.....	24
3.4 Текст классов команд.....	27
4 Текст модуля Внешних устройств.....	27
4.1 Текст класса DevicesManager.....	27
4.2 Текст класса DeviceProvider.....	28
4.3 Текст класса DeviceValidator.....	29
4.4 Текст класса DeviceContext.....	30
4.5 Текст интерфейса IDevice.....	30
5 Текст модуля Графического интерфейса.....	31
5.1 Текст класса MainWindowViewModel.....	31
5.2 Текст класса SettingsViewModel.....	40
5.3 Текст класса FileTabViewModel.....	42
5.4 Текст класса ExecutorWindowViewModel.....	43
5.5 Текст класса ProjectManager.....	47
5.6 Текст класса FileManager.....	49
5.7 Текст класса FileModel.....	51
5.8 Текст класса SettingsManager.....	52
5.9 Текст класса TabManager.....	52
Лист регистрации изменений.....	55

Основная часть

1 Текст общих моделей

1.1 Текст модели проекта

```
public class Project : IProject
{
    public string Executable { get; set; } = string.Empty;
    public IList<string> Files { get; init; } = new List<string>();
    public IList<string> Devices { get; init; } = new List<string>();
    public ushort StackAddress { get; set; } = 512;
    public ushort ProgramAddress { get; set; } = 512;
    public string ProjectFile { get; init; } = string.Empty;
    public string ProjectDirectory => PathHelper.GetDirectoryName(ProjectFile);
    public string ProjectName => PathHelper.GetFileName(ProjectFile);
    public string ProjectBinary => PathHelper.Combine(ProjectDirectory,
$"{ProjectName}.pdp11bin");
}
```

2 Текст модуля Ассемблера

2.1 Текст класса Compiler

```
public class Compiler
{
    private readonly Parser _parser;
    private readonly TokenBuilder _tokenBuilder;

    public Compiler()
    {
        _parser = new Parser();
        _tokenBuilder = new TokenBuilder();
    }

    public async Task Compile(IProject project)
    {
        var mainFile = project.Executable;
        var mainCommandLines = await _parser.Parse(mainFile);

        var tokens = new List<IToken>();
        var marks = new Dictionary<string, int>();
        var currentAddr = 0;
        foreach (var cmdLine in mainCommandLines)
        {
            foreach (var mark in cmdLine.Marks)
            {
                if (!marks.ContainsKey(mark))
                {
                    marks.Add(mark, currentAddr);
                }
                else
                {
                    throw new Exception($"The mark '{mark}' has been used several
times");
                }
            }

            var cmdTokens = _tokenBuilder.Build(cmdLine);
```

```

        tokens.AddRange(cmdTokens);
        currentAddr += cmdTokens.Count() * 2;
    }

    currentAddr = 0;
    var codes = new List<string>();
    foreach (var token in tokens)
    {
        codes.AddRange(token.Translate(marks, currentAddr));
        currentAddr += 2;
    }

    await File.WriteAllLinesAsync(project.ProjectBinary, codes);
}
}

```

2.2 Текст класса Parser

```

internal class Parser
{
    private static readonly char[] BadSymbols = { ' ', '\t', ',', ':' };

    private readonly Regex _regexMaskCommandLine =
        new(@"^\s*([^\s,;]+\s*)?(\S+)\s*([^\s,;\s*]{0,}$",
        RegexOptions.IgnoreCase | RegexOptions.Singleline);

    private readonly Regex _regexMaskRemovingComment =
        new(@"^[^;]+(?:=|?)", RegexOptions.IgnoreCase | RegexOptions.Singleline);

    private readonly Regex _regexMaskMarkExistence =
        new(@"^\s*^[^;]*:", RegexOptions.IgnoreCase | RegexOptions.Singleline);

    private readonly Regex _regexMaskMarkValidation =
        new(@"^\s*[a-zA-Z]+[a-zA-Z0-9_]*([^;:])(\w)*(?:=|)", RegexOptions.IgnoreCase |
        RegexOptions.Singleline);

    public async Task<List<CommandLine>> Parse(string filePath)
    {
        var res = new List<CommandLine>();
        string line;

        using var reader = new StreamReader(filePath);
        var marksSet = new HashSet<string>();

        while ((line = await reader.ReadLineAsync()) != null)
        {
            line = line.Split('; ', StringSplitOptions.TrimEntries)[0];
            if (string.IsNullOrEmpty(line))
            {
                continue;
            }

            var markExistence = _regexMaskMarkExistence.Match(line).Groups[0].Value;
            if (markExistence != "")
            {
                var markValid = _regexMaskMarkValidation.Match(line).Groups[0].Value;
                if (markValid == "")
                {
                    throw new Exception($"Invalid mark: {markExistence}.");
                }
            }
        }
    }
}

```

```

        var match = _regexMaskCommandLine.Match(line);

        var mark = match.Groups[1].Value.Trim().Trim(BadSymbols).ToLower();
        marksSet.Add(mark);
        var instruction = match.Groups[2].Value.Trim(BadSymbols).ToLower();
        if (string.IsNullOrEmpty(instruction))
        {
            continue;
        }

        var arguments = match.Groups[3].Captures.Select(c =>
c.Value.Trim(BadSymbols).ToLower());

        var command = new CommandLine(marksSet, instruction, arguments);
        command.ThrowIfInvalid();
        res.Add(command);
        marksSet.Clear();
    }

    return res;
}
}

```

2.3 Текст класса TokenBuilder

```

internal class TokenBuilder
{
    private const string RegexPatternAddrType0 = @"^r([0-7])$";
    private const string RegexPatternAddrType1 = @"^@r([0-7])$";
    private const string RegexPatternAddrType2 = @"^\\(r([0-7])\\)\\";
    private const string RegexPatternAddrType3 = @"^@\\(r([0-7])\\)\\";
    private const string RegexPatternAddrType4 = @"^\\(r([0-7])\\)$";
    private const string RegexPatternAddrType5 = @"^@\\(r([0-7])\\)$";
    private const string RegexPatternAddrType6 = @"^([0-1]*[0-7]{1,5})\\(r([0-7])\\)$";
    private const string RegexPatternAddrType6Mark = @"^([a-z]+[_a-z0-9]*)(\\[+-])
([0-1]*[0-7]{1,5}))?\\(r([0-7])\\)$";
    private const string RegexPatternAddrType7 = @"^@([0-1]*[0-7]{1,5})\\(r([0-7])\\)
$";
    private const string RegexPatternAddrType7Mark = @"^@([a-z]+[_a-z0-9]*)(\\[+-])
([0-1]*[0-7]{1,5}))?\\(r([0-7])\\)$";
    private const string RegexPatternAddrType21 = @"^#([0-1]*[0-7]{1,5})$";
    private const string RegexPatternAddrType21Mark = @"^#([a-z]+[_a-z0-9]*)$";
    private const string RegexPatternAddrType31 = @"^@#([0-1]*[0-7]{1,5})$";
    private const string RegexPatternAddrType31Mark = @"^@#([a-z]+[_a-z0-9]*)$";
    private const string RegexPatternAddrType61 = @"^([a-z]+[_a-z0-9]*)$";
    private const string RegexPatternAddrType71 = @"^@([a-z]+[_a-z0-9]*)$";
    private const string RegexPatternArgNN = @"^([0-7]{1,2})$";
    private const string RegexPatternArgWORD = @"^([-]?[0-9]+)([.]?)$";
    private const string RegexPatternArgBLKW = @"^([0-9]+)$";

    private readonly Regex _regexMaskAddrType0;
    private readonly Regex _regexMaskAddrType1;
    private readonly Regex _regexMaskAddrType2;
    private readonly Regex _regexMaskAddrType3;
    private readonly Regex _regexMaskAddrType4;
    private readonly Regex _regexMaskAddrType5;
    private readonly Regex _regexMaskAddrType6;
    private readonly Regex _regexMaskAddrType6Mark;
    private readonly Regex _regexMaskAddrType7;
    private readonly Regex _regexMaskAddrType7Mark;

```

```

private readonly Regex _regexMaskAddrType21;
private readonly Regex _regexMaskAddrType21Mark;
private readonly Regex _regexMaskAddrType31;
private readonly Regex _regexMaskAddrType31Mark;
private readonly Regex _regexMaskAddrType61;
private readonly Regex _regexMaskAddrType71;
private readonly Regex _regexMaskArgNN;
private readonly Regex _regexMaskArgWORD;
private readonly Regex _regexMaskArgBLKW;

private readonly Dictionary<string, Func<CommandLine, List<IToken>>>
_instructions;

private int ArgumentHandler(string arg, List<IToken> extraTokens)
{
    int instArgCode;

    if (_regexMaskAddrType0.IsMatch(arg))
    {
        instArgCode = 0b000_000;
        instArgCode = instArgCode |
int.Parse(_regexMaskAddrType0.Match(arg).Groups[1].Value);
    }
    else if (_regexMaskAddrType1.IsMatch(arg))
    {
        instArgCode = 0b001_000;
        instArgCode = instArgCode |
int.Parse(_regexMaskAddrType1.Match(arg).Groups[1].Value);
    }
    else if (_regexMaskAddrType2.IsMatch(arg))
    {
        instArgCode = 0b010_000;
        instArgCode = instArgCode |
int.Parse(_regexMaskAddrType2.Match(arg).Groups[1].Value);
    }
    else if (_regexMaskAddrType3.IsMatch(arg))
    {
        instArgCode = 0b011_000;
        instArgCode = instArgCode |
int.Parse(_regexMaskAddrType3.Match(arg).Groups[1].Value);
    }
    else if (_regexMaskAddrType4.IsMatch(arg))
    {
        instArgCode = 0b100_000;
        instArgCode = instArgCode |
int.Parse(_regexMaskAddrType4.Match(arg).Groups[1].Value);
    }
    else if (_regexMaskAddrType5.IsMatch(arg))
    {
        instArgCode = 0b101_000;
        instArgCode = instArgCode |
int.Parse(_regexMaskAddrType5.Match(arg).Groups[1].Value);
    }
    else if (_regexMaskAddrType6.IsMatch(arg))
    {
        instArgCode = 0b110_000;
        instArgCode = instArgCode |
int.Parse(_regexMaskAddrType6.Match(arg).Groups[2].Value);
        var extraWordCode =
Convert.ToInt32(_regexMaskAddrType6.Match(arg).Groups[1].Value, 8);
        extraTokens.Add(new RawToken(extraWordCode));
    }
}

```

```

    }
    else if (_regexMaskAddrType6Mark.IsMatch(arg))
    {
        instArgCode = 0b110_000;
        instArgCode = instArgCode |
int.Parse(_regexMaskAddrType6Mark.Match(arg).Groups[5].Value);
        var mark = _regexMaskAddrType6Mark.Match(arg).Groups[1].Value;
        var parseValue = _regexMaskAddrType6Mark.Match(arg).Groups[4].Value;
        var num = string.IsNullOrEmpty(parseValue) ? 0 :
Convert.ToInt32(parseValue, 8);
        var opSign = _regexMaskAddrType6Mark.Match(arg).Groups[3].Value;
        extraTokens.Add(new MarkRelocationToken(mark, num, opSign == "+" ? true :
false));
    }
    else if (_regexMaskAddrType7.IsMatch(arg))
    {
        instArgCode = 0b111_000;
        instArgCode = instArgCode |
int.Parse(_regexMaskAddrType7.Match(arg).Groups[2].Value);
        var extraWordCode =
Convert.ToInt32(_regexMaskAddrType7.Match(arg).Groups[1].Value, 8);
        extraTokens.Add(new RawToken(extraWordCode));
    }
    else if (_regexMaskAddrType7Mark.IsMatch(arg))
    {
        instArgCode = 0b111_000;
        instArgCode = instArgCode |
int.Parse(_regexMaskAddrType7Mark.Match(arg).Groups[5].Value);
        var mark = _regexMaskAddrType7Mark.Match(arg).Groups[1].Value;
        var parseValue = _regexMaskAddrType7Mark.Match(arg).Groups[4].Value;
        var num = string.IsNullOrEmpty(parseValue) ? 0 :
Convert.ToInt32(parseValue, 8);
        var opSign = _regexMaskAddrType7Mark.Match(arg).Groups[3].Value;
        extraTokens.Add(new MarkRelocationToken(mark, num, opSign == "+" ? true :
false));
    }
    else if (_regexMaskAddrType21.IsMatch(arg))
    {
        instArgCode = 0b010_111;
        var extraWordCode =
Convert.ToInt32(_regexMaskAddrType21.Match(arg).Groups[1].Value, 8);
        extraTokens.Add(new RawToken(extraWordCode));
    }
    else if (_regexMaskAddrType21Mark.IsMatch(arg))
    {
        instArgCode = 0b010_111;
        var mark = _regexMaskAddrType21Mark.Match(arg).Groups[1].Value;
        extraTokens.Add(new MarkRelocationToken(mark, 0, true));
    }
    else if (_regexMaskAddrType31.IsMatch(arg))
    {
        instArgCode = 0b011_111;
        var extraWordCode =
Convert.ToInt32(_regexMaskAddrType31.Match(arg).Groups[1].Value, 8);
        extraTokens.Add(new RawToken(extraWordCode));
    }
    else if (_regexMaskAddrType31Mark.IsMatch(arg))
    {
        instArgCode = 0b011_111;
        var mark = _regexMaskAddrType31Mark.Match(arg).Groups[1].Value;
        extraTokens.Add(new MarkRelocationToken(mark, 0, true));
    }

```



```

    }
    else if (_regexMaskAddrType61.IsMatch(arg))
    {
        instArgCode = 0b110_111;
        extraTokens.Add(new
MarkRelatedToken(_regexMaskAddrType61.Match(arg).Groups[1].Value));
    }
    else if (_regexMaskAddrType71.IsMatch(arg))
    {
        instArgCode = 0b111_111;
        extraTokens.Add(new
MarkRelatedToken(_regexMaskAddrType61.Match(arg).Groups[1].Value));
    }
    else
    {
        throw new ArgumentException($"Incorrect argument: {arg}.");
    }

    return instArgCode;
}

private List<IToken> InstructionArgsNull(CommandLine cmdLine)
{
    return new List<IToken>
    {
        new
OperationToken(Instruction.Instructions[cmdLine.InstructionMnemonics].Code, cmdLine)
    };
}

private List<IToken> InstructionArgsDD(CommandLine cmdLine)
{
    var resultTokens = new List<IToken>();
    var extraTokens = new List<IToken>();

    var instArgCode = ArgumentHandler(cmdLine.Arguments[0], extraTokens);

    resultTokens.Add(new
OperationToken(Instruction.Instructions[cmdLine.InstructionMnemonics].Code |
instArgCode, cmdLine));
    resultTokens.AddRange(extraTokens);

    return resultTokens;
}

private List<IToken> InstructionArgsSSDD(CommandLine cmdLine)
{
    var resultTokens = new List<IToken>();
    var extraTokens = new List<IToken>();

    var instArgCode = ArgumentHandler(cmdLine.Arguments[0], extraTokens);
    instArgCode = instArgCode << 6;
    instArgCode = instArgCode | ArgumentHandler(cmdLine.Arguments[1],
extraTokens);

    resultTokens.Add(new
OperationToken(Instruction.Instructions[cmdLine.InstructionMnemonics].Code |
instArgCode, cmdLine));
    resultTokens.AddRange(extraTokens);

    return resultTokens;
}

```

```

    }

    private List<IToken> InstructionArgsR(CommandLine cmdLine)
    {
        var resultTokens = new List<IToken>();
        int instArgCode = 0;

        if (_regexMaskAddrType0.IsMatch(cmdLine.Arguments[0]))
        {
            instArgCode =
            Convert.ToInt32(_regexMaskAddrType0.Match(cmdLine.Arguments[0]).Groups[1].Value, 8);
        }
        else
        {
            throw new ArgumentException($"Incorrect argument:
            {cmdLine.Arguments[0]}.");
        }

        resultTokens.Add(new
        OperationToken(Instruction.Instructions[cmdLine.InstructionMnemonics].Code |
        instArgCode, cmdLine));
        return resultTokens;
    }

    private List<IToken> InstructionArgsRDD(CommandLine cmdLine)
    {
        var resultTokens = new List<IToken>();
        var extraTokens = new List<IToken>();
        int instArgCode = 0;

        if (_regexMaskAddrType0.IsMatch(cmdLine.Arguments[0]))
        {
            instArgCode =
            Convert.ToInt32(_regexMaskAddrType0.Match(cmdLine.Arguments[0]).Groups[1].Value, 8);
            instArgCode = instArgCode << 6;
        }
        else
        {
            throw new ArgumentException($"Incorrect argument:
            {cmdLine.Arguments[0]}.");
        }

        instArgCode = instArgCode | ArgumentHandler(cmdLine.Arguments[1],
        extraTokens);

        resultTokens.Add(new
        OperationToken(Instruction.Instructions[cmdLine.InstructionMnemonics].Code |
        instArgCode, cmdLine));
        resultTokens.AddRange(extraTokens);

        return resultTokens;
    }

    private List<IToken> InstructionArgsNN(CommandLine cmdLine)
    {
        var resultTokens = new List<IToken>();
        int instArgCode = 0;

        if (_regexMaskArgNN.IsMatch(cmdLine.Arguments[0]))
        {
            instArgCode =

```

```

Convert.ToInt32(_regexMaskArgNN.Match(cmdLine.Arguments[0]).Groups[1].Value, 8);
    }
    else
    {
        throw new ArgumentException($"Incorrect argument:
{cmdLine.Arguments[0]}.");
    }

    resultTokens.Add(new
OperationToken(Instruction.Instructions[cmdLine.InstructionMnemonics].Code |
instArgCode, cmdLine));
    return resultTokens;
}

private List<IToken> InstructionArgsRNN(CommandLine cmdLine)
{
    var resultTokens = new List<IToken>();
    int instArgCode = 0;

    if (_regexMaskAddrType0.IsMatch(cmdLine.Arguments[0]))
    {
        instArgCode =
Convert.ToInt32(_regexMaskAddrType0.Match(cmdLine.Arguments[0]).Groups[1].Value, 8);
        instArgCode = instArgCode << 6;
    }
    else
    {
        throw new ArgumentException($"Incorrect argument:
{cmdLine.Arguments[0]}.");
    }

    if (_regexMaskAddrType61.IsMatch(cmdLine.Arguments[1]))
    {
        resultTokens.Add(new ShiftBackOperationToken(
            Instruction.Instructions[cmdLine.InstructionMnemonics].Code |
instArgCode,
            cmdLine.Arguments[1],
            0b111_111,
            cmdLine));
    }
    else
    {
        throw new ArgumentException($"Incorrect argument:
{cmdLine.Arguments[1]}.");
    }

    return resultTokens;
}

private List<IToken> InstructionArgsShift(CommandLine cmdLine)
{
    var resultTokens = new List<IToken>();

    var arg = cmdLine.Arguments[0];
    if (_regexMaskAddrType61.IsMatch(arg))
    {
        resultTokens.Add(new ShiftOperationToken(
            Instruction.Instructions[cmdLine.InstructionMnemonics].Code,
            cmdLine.Arguments[0],
            0b1111_1111,
            cmdLine)

```

```

        );
    }
    else
    {
        throw new ArgumentException($"Incorrect argument:
{cmdLine.Arguments[0]}.");
    }

    return resultTokens;
}

private List<IToken> PseudoInstructionWORD(CommandLine cmdLine)
{
    var resultTokens = new List<IToken>();

    foreach (var arg in cmdLine.Arguments)
    {
        if (_regexMaskArgWORD.IsMatch(arg))
        {
            var value = _regexMaskArgWORD.Match(arg).Groups[1].Value;

            int valueDec;
            if
(string.IsNullOrEmpty(_regexMaskArgWORD.Match(arg).Groups[2].Value))
            {
                var isNegative = value.StartsWith('-');
                valueDec = (isNegative ? -1 : 1) * Convert.ToInt32(isNegative ?
value[1..] : value, 8);
            }
            else
            {
                valueDec = Convert.ToInt32(value);
            }

            if (valueDec is > short.MaxValue or < short.MinValue)
            {
                throw new ArgumentException($"Incorrect argument: {arg}.");
            }

            valueDec &= 0xFFFF;

            resultTokens.Add(new RawToken(valueDec));
        }
        else
        {
            throw new ArgumentException($"Incorrect argument: {arg}.");
        }
    }

    return resultTokens;
}

private List<IToken> PseudoInstructionBLKW(CommandLine cmdLine)
{
    var resultTokens = new List<IToken>();

    if (_regexMaskArgBLKW.IsMatch(cmdLine.Arguments[0]))
    {
        var valueDec =
Convert.ToInt32(_regexMaskArgBLKW.Match(cmdLine.Arguments[0]).Groups[1].Value, 8);
        for (var i = 0; i < valueDec; i++)

```

```

        {
            resultTokens.Add(new RawToken(0));
        }
    }
    else
    {
        throw new ArgumentException($"Incorrect argument:
{cmdLine.Arguments[0]}.");
    }

    return resultTokens;
}

private List<IToken> PseudoInstructionEND(CommandLine cmdLine)
{
    var resultTokens = new List<IToken>();

    if (_regexMaskAddrType61.IsMatch(cmdLine.Arguments[0]))
    {
        resultTokens.Add(new
MarkRelocationToken(_regexMaskAddrType61.Match(cmdLine.Arguments[0]).Groups[1].Value,
0, true));
    }
    else
    {
        throw new ArgumentException($"Incorrect argument:
{cmdLine.Arguments[0]}.");
    }

    return resultTokens;
}

public TokenBuilder()
{
    _instructions = new Dictionary<string, Func<CommandLine, List<IToken>>>
    {
        { "clr", InstructionArgsDD },
        { "clrb", InstructionArgsDD },
        { "com", InstructionArgsDD },
        { "comb", InstructionArgsDD },
        { "inc", InstructionArgsDD },
        { "incb", InstructionArgsDD },
        { "dec", InstructionArgsDD },
        { "decb", InstructionArgsDD },
        { "neg", InstructionArgsDD },
        { "negb", InstructionArgsDD },
        { "tst", InstructionArgsDD },
        { "tstb", InstructionArgsDD },
        { "asr", InstructionArgsDD },
        { "asrb", InstructionArgsDD },
        { "asl", InstructionArgsDD },
        { "aslb", InstructionArgsDD },
        { "ror", InstructionArgsDD },
        { "rorb", InstructionArgsDD },
        { "rol", InstructionArgsDD },
        { "rolb", InstructionArgsDD },
        { "swab", InstructionArgsDD },
        { "adc", InstructionArgsDD },
        { "adcb", InstructionArgsDD },
        { "sbc", InstructionArgsDD },
        { "sbc b", InstructionArgsDD },
    }
}

```

```

{ "sxt", InstructionArgsDD },
{ "mfps", InstructionArgsDD },
{ "mtps", InstructionArgsDD },
{ "mov", InstructionArgsSSDD },
{ "movb", InstructionArgsSSDD },
{ "cmp", InstructionArgsSSDD },
{ "cmpb", InstructionArgsSSDD },
{ "add", InstructionArgsSSDD },
{ "sub", InstructionArgsSSDD },
{ "bit", InstructionArgsSSDD },
{ "bitb", InstructionArgsSSDD },
{ "bic", InstructionArgsSSDD },
{ "bicb", InstructionArgsSSDD },
{ "bis", InstructionArgsSSDD },
{ "bisb", InstructionArgsSSDD },
{ "mul", InstructionArgsRDD },
{ "div", InstructionArgsRDD },
{ "ash", InstructionArgsRDD },
{ "ashc", InstructionArgsRDD },
{ "xor", InstructionArgsRDD },
{ "br", InstructionArgsShift },
{ "bne", InstructionArgsShift },
{ "beq", InstructionArgsShift },
{ "bpl", InstructionArgsShift },
{ "bmi", InstructionArgsShift },
{ "bvc", InstructionArgsShift },
{ "bvs", InstructionArgsShift },
{ "bcc", InstructionArgsShift },
{ "bcs", InstructionArgsShift },
{ "bge", InstructionArgsShift },
{ "blt", InstructionArgsShift },
{ "bgt", InstructionArgsShift },
{ "ble", InstructionArgsShift },
{ "bhi", InstructionArgsShift },
{ "blos", InstructionArgsShift },
{ "bhis", InstructionArgsShift },
{ "blo", InstructionArgsShift },
{ "jmp", InstructionArgsDD },
{ "jsr", InstructionArgsRDD },
{ "rts", InstructionArgsR },
{ "fmul", InstructionArgsR },
{ "fdiv", InstructionArgsR },
{ "fadd", InstructionArgsR },
{ "fsub", InstructionArgsR },
{ "mark", InstructionArgsNn },
{ "sob", InstructionArgsRnn },
{ "bpt", InstructionArgsNull },
{ "iot", InstructionArgsNull },
{ "rti", InstructionArgsNull },
{ "rtt", InstructionArgsNull },
{ "halt", InstructionArgsNull },
{ "wait", InstructionArgsNull },
{ "reset", InstructionArgsNull },
{ "clc", InstructionArgsNull },
{ "clv", InstructionArgsNull },
{ "clz", InstructionArgsNull },
{ "cln", InstructionArgsNull },
{ "sec", InstructionArgsNull },
{ "sev", InstructionArgsNull },
{ "sez", InstructionArgsNull },
{ "sen", InstructionArgsNull },

```

```

        { "scc", InstructionArgsNull },
        { "ccc", InstructionArgsNull },
        { "nop", InstructionArgsNull },
        { ".word", PseudoInstructionWORD },
        { ".blkw", PseudoInstructionBLKW },
        { ".end", PseudoInstructionEND }
    };

    _regexMaskAddrType0 = new Regex(RegexPatternAddrType0,
    RegexOptions.IgnoreCase | RegexOptions.Singleline);
    _regexMaskAddrType1 = new Regex(RegexPatternAddrType1,
    RegexOptions.IgnoreCase | RegexOptions.Singleline);
    _regexMaskAddrType2 = new Regex(RegexPatternAddrType2,
    RegexOptions.IgnoreCase | RegexOptions.Singleline);
    _regexMaskAddrType3 = new Regex(RegexPatternAddrType3,
    RegexOptions.IgnoreCase | RegexOptions.Singleline);
    _regexMaskAddrType4 = new Regex(RegexPatternAddrType4,
    RegexOptions.IgnoreCase | RegexOptions.Singleline);
    _regexMaskAddrType5 = new Regex(RegexPatternAddrType5,
    RegexOptions.IgnoreCase | RegexOptions.Singleline);
    _regexMaskAddrType6 = new Regex(RegexPatternAddrType6,
    RegexOptions.IgnoreCase | RegexOptions.Singleline);
    _regexMaskAddrType6Mark =
        new Regex(RegexPatternAddrType6Mark, RegexOptions.IgnoreCase |
    RegexOptions.Singleline);
    _regexMaskAddrType7 = new Regex(RegexPatternAddrType7,
    RegexOptions.IgnoreCase | RegexOptions.Singleline);
    _regexMaskAddrType7Mark =
        new Regex(RegexPatternAddrType7Mark, RegexOptions.IgnoreCase |
    RegexOptions.Singleline);
    _regexMaskAddrType21 = new Regex(RegexPatternAddrType21,
    RegexOptions.IgnoreCase | RegexOptions.Singleline);
    _regexMaskAddrType21Mark =
        new Regex(RegexPatternAddrType21Mark, RegexOptions.IgnoreCase |
    RegexOptions.Singleline);
    _regexMaskAddrType31 = new Regex(RegexPatternAddrType31,
    RegexOptions.IgnoreCase | RegexOptions.Singleline);
    _regexMaskAddrType31Mark =
        new Regex(RegexPatternAddrType31Mark, RegexOptions.IgnoreCase |
    RegexOptions.Singleline);
    _regexMaskAddrType61 = new Regex(RegexPatternAddrType61,
    RegexOptions.IgnoreCase | RegexOptions.Singleline);
    _regexMaskAddrType71 = new Regex(RegexPatternAddrType71,
    RegexOptions.IgnoreCase | RegexOptions.Singleline);
    _regexMaskArgNN = new Regex(RegexPatternArgNN, RegexOptions.IgnoreCase |
    RegexOptions.Singleline);

    _regexMaskArgWORD = new Regex(RegexPatternArgWORD, RegexOptions.IgnoreCase |
    RegexOptions.Singleline);
    _regexMaskArgBLKW = new Regex(RegexPatternArgBLKW, RegexOptions.IgnoreCase |
    RegexOptions.Singleline);
    }

    public IEnumerable<IToken> Build(CommandLine cmdLine)
    {
        var resultTokens = new List<IToken>();
        resultTokens.AddRange(_instructions[cmdLine.InstructionMnemonics](cmdLine));
        return resultTokens;
    }
}

```

2.4 Текст класса CommandLine

```
internal record CommandLine
{
    private const string RegexPatternMarkValidation = @"^\s*[a-zA-Z]+(?:\s*[:;]\w)*(?:\s*=:)*";

    public CommandLine(IEnumerable<string> marks, string instructionMnemonics,
        IEnumerable<string> args)
    {
        Marks = marks.ToHashSet();
        InstructionMnemonics = instructionMnemonics;
        Arguments = args.ToList();
    }

    public void ThrowIfInvalid()
    {
        if (string.IsNullOrEmpty(InstructionMnemonics))
        {
            return;
        }

        if (!Instruction.Instructions.ContainsKey(InstructionMnemonics))
        {
            throw new System.Exception($"Unexisting instruction:
{InstructionMnemonics}.");
        }

        if ((Arguments.Count !=
            Instruction.Instructions[InstructionMnemonics].ArgumentsCount) &
            (Instruction.Instructions[InstructionMnemonics].ArgumentsCount != -1))
        {
            throw new System.Exception(
                $"Incorrect number of arguments: {InstructionMnemonics}. " +
                $"Must be
{Instruction.Instructions[InstructionMnemonics].ArgumentsCount}, " +
                $"but was: {Arguments.Count}.");
        }
    }

    public string GetSymbol()
    {
        return $"{string.Join(',', Marks)}: {InstructionMnemonics} {string.Join(',',
Arguments)}";
    }

    public IEnumerable<string> Marks { get; }
    public string InstructionMnemonics { get; }
    public List<string> Arguments { get; }
}
```

2.5 Текст класса Token

```
internal class MarkRelatedToken : IToken
{
    private readonly string _mark;

    public MarkRelatedToken(string mark)
    {
        _mark = mark;
    }
}
```



```

    public IEnumerable<string> Translate(Dictionary<string, int> marksDict, int
currentAddr)
    {
        if (!marksDict.ContainsKey(_mark))
        {
            throw new Exception($"The mark ({_mark}) is not determined.");
        }

        var delta = marksDict[_mark] - currentAddr;
        if (Math.Abs(delta) > 65535)
        {
            throw new Exception($"The distance to the mark ({_mark}) is too large.
{delta}");
        }

        var relDist = Convert.ToString(Convert.ToInt16(delta - 2), 8).PadLeft(6,
'0');
        return new[] { relDist };
    }
}

internal class MarkRelocationToken : IToken
{
    private readonly string _mark;

    private readonly int _addValue;

    private readonly bool _opSign;

    public MarkRelocationToken(string mark, int addValue, bool opSign)
    {
        _mark = mark;
        _addValue = addValue;
        _opSign = opSign;
    }

    public IEnumerable<string> Translate(Dictionary<string, int> marksDict, int
currentAddr)
    {
        if (!marksDict.ContainsKey(_mark))
        {
            throw new Exception($"The mark ({_mark}) is not determined.");
        }

        var word = Convert.ToString(marksDict[_mark] + (_opSign ? 1 : -1) *
_addValue, 8).PadLeft(6, '0') + "";
        return new[] { word };
    }
}

internal class OperationToken : IToken
{
    private readonly int _machineCode;

    private readonly CommandLine _originCmdLine;

    public OperationToken(int machineCode, CommandLine originCmdLine)
    {
        _machineCode = machineCode;
        _originCmdLine = originCmdLine;
    }
}

```

```

        public IEnumerable<string> Translate(Dictionary<string, int> marksDict, int
currentAddr)
        {
            return new[] { Convert.ToString(_machineCode, 8).PadLeft(6, '0') + $"
{_originCmdLine.GetSymbol()}" };
        }
    }

internal class RawToken : IToken
{
    private readonly int _machineCode;

    public RawToken(int machineCode)
    {
        _machineCode = machineCode;
    }

    public IEnumerable<string> Translate(Dictionary<string, int> marksDict, int
currentAddr)
    {
        return new[] { Convert.ToString(_machineCode, 8).PadLeft(6, '0') };
    }
}

internal class ShiftOperationToken : IToken
{
    protected readonly int _machineCode;
    protected readonly string _mark;
    protected readonly CommandLine _originCmdLine;
    protected readonly int _shiftMask;

    public ShiftOperationToken(int machineCode, string mark, int shiftMask,
CommandLine originCmdLine)
    {
        _machineCode = machineCode;
        _mark = mark;
        _originCmdLine = originCmdLine;
        _shiftMask = shiftMask;
    }

    public virtual IEnumerable<string> Translate(Dictionary<string, int> marksDict,
int currentAddr)
    {
        int delta = 0;
        if (marksDict.TryGetValue(_mark, out var markAddress))
        {
            delta = markAddress - currentAddr;
        }
        else
        {
            throw new Exception($"The mark ({_mark}) is not determined.");
        }

        if (delta > _shiftMask)
        {
            throw new Exception($"The distance to the mark ({_mark}) is too large.
{delta}");
        }

        var shiftValue = (delta / 2 - 1) & _shiftMask;
    }
}

```

```

        return new List<string> { Convert.ToString(_machineCode | shiftValue,
8).PadLeft(6, '0') + $";{_originCmdLine.GetSymbol()}" };
    }
}

internal class ShiftBackOperationToken : ShiftOperationToken
{
    public ShiftBackOperationToken(int machineCode, string mark, int shiftMask,
CommandLine originCmdLine) :
        base(machineCode, mark, shiftMask, originCmdLine)
    {
    }

    public override IEnumerable<string> Translate(Dictionary<string, int> marksDict,
int currentAddr)
    {
        int delta = 0;
        if (marksDict.TryGetValue(_mark, out var markAddress))
        {
            if (markAddress >= currentAddr)
            {
                throw new Exception($"The instruction
({_originCmdLine.InstructionMnemonics}) can't uses forward marks ({_mark}).");
            }
            delta = currentAddr - markAddress;
        }
        else
        {
            throw new Exception($"The mark ({_mark}) is not determined.");
        }

        if (delta > _shiftMask)
        {
            throw new Exception($"The distance to the mark ({_mark}) is too large.
{delta}");
        }

        var shiftValue = (delta / 2 + 1) & _shiftMask;

        return new List<string> { Convert.ToString(_machineCode | shiftValue,
8).PadLeft(6, '0') + $";{_originCmdLine.GetSymbol()}" };
    }
}

```

3 Текст модуля Исполнителя

3.1 Текст класса Executor

```

public class Executor
{
    private bool _initialized;
    private ICommand _lastCommand;

    private readonly Stack<string> _trapStack = new();

    private readonly HashSet<string> _trapsToHalt = new()
    {
        nameof(BusException),
        nameof(OddAddressException),
    }
}

```

```

        nameof(EMT),
        nameof(TRAP),
        nameof(IOT),
        nameof(BPT),
        "Trace"
    };

    private readonly IState _state;
    private readonly IStorage _memory;
    private readonly IDeviceValidator _deviceValidator;
    private readonly IDevicesManager _devicesManager;
    private readonly Bus _bus;

    private readonly CommandParser _commandParser;
    private readonly Dictionary<ushort, string> _symbols = new();
    private readonly HashSet<ushort> _breakpoints = new();

    public ushort ProcessorStateWord => _state.ProcessorStateWord;

    public IReadOnlyCollection<ushort> Registers => _state.Registers;

    public IReadOnlyStorage Memory => _memory;

    public IReadOnlyCollection<IDevice> Devices => _devicesManager.Devices;

    public IReadOnlyDictionary<ushort, string> Symbols => _symbols;

    public IReadOnlySet<ushort> Breakpoints => _breakpoints;

    public IProject Project { get; private set; }

    public Executor()
    {
        _state = new State();
        _memory = new Memory();
        var provider = new DeviceProvider();
        _devicesManager = new DevicesManager(provider);
        _deviceValidator = new DeviceValidator(provider);
        _bus = new Bus(_memory, _devicesManager);
        _commandParser = new CommandParser(_bus, _state);
    }

    public void Init()
    {
        if (_initialized)
        {
            return;
        }

        _initialized = true;

        _bus.Init();
    }

    public async Task<bool> ExecuteAsync(Cancellation_token cancellation_token)
    {
        Init();

        var res = true;

        while (!cancellation_token.IsCancellationRequested && res)

```

```

    {
        if (_breakpoints.Contains(_state.Registers[7]))
        {
            break;
        }

        res = await ExecuteNextInstructionAsync();
        await Task.Yield();
    }

    return res;
}

public bool ExecuteNextInstruction()
{
    Init();

    if (_state.T && _lastCommand is not RTT and not TrapInstruction and not WAIT)
    {
        HandleInterrupt("Trace", 12); // 0o14
    }

    var interruptedDevice = _bus.GetInterrupt(_state.Priority);
    if (interruptedDevice != null)
    {
        interruptedDevice.AcceptInterrupt();
        HandleInterrupt(interruptedDevice.GetType().Name,
interruptedDevice.InterruptVectorAddress);
    }
    else if (_lastCommand is WAIT)
    {
        return true;
    }

    try
    {
        var word = _memory.GetWord(_state.Registers[7]);
        _state.Registers[7] += 2;

        _lastCommand = _commandParser.GetCommand(word);
        _lastCommand.Execute(_lastCommand.GetArguments(word));

        if (_lastCommand is TrapInstruction)
        {
            _trapStack.Push(_lastCommand.GetType().Name);
        }
        else if (_lastCommand is TrapReturn)
        {
            _trapStack.Pop();
        }
    }
    catch (HaltException e) when (e.IsExpected)
    {
        return false;
    }
    catch (Exception e)
    {
        HandleHardwareTrap(e);
    }

    return true;
}

```

```

    }

    public Task<bool> ExecuteNextInstructionAsync() =>
    Task.Run(ExecuteNextInstruction);

    public Task LoadProgram(IProject project)
    {
        if (project.ProgramAddress % 2 == 1)
        {
            throw new InvalidOperationException("Start program address cannot be
odd");
        }

        if (project.StackAddress % 2 == 1)
        {
            throw new InvalidOperationException("Start stack address cannot be odd");
        }

        Project = project;

        return Reload();
    }

    public async Task Reload()
    {
        _initialized = false;
        _devicesManager.Clear();
        Array.Fill<ushort>(_state.Registers, 0);

        _state.Registers[6] = Project.StackAddress;
        _state.Registers[7] = Project.ProgramAddress;

        using var reader = new StreamReader(Project.ProjectBinary);

        var address = Project.ProgramAddress;
        while (await reader.ReadLineAsync() is { } line)
        {
            var tokens = line.Split('; ', StringSplitOptions.TrimEntries);

            var code = tokens[0];
            var isRelocatable = code.EndsWith('\ ');

            var word = isRelocatable
                ? Convert.ToUInt16(code[..6], 8) + Project.ProgramAddress
                : Convert.ToUInt16(code, 8);

            if (word > ushort.MaxValue)
            {
                throw new OutOfMemoryException("Program is too large");
            }

            _memory.SetWord(address, (ushort)word);

            var symbol = tokens.ElementAtOrDefault(1);
            _symbols.Add(address, symbol);

            address += 2;
        }

        foreach (var device in project.Devices) {
            AddDevice(device);
        }
    }

```

```

    }
}

public void AddBreakpoint(ushort address) => _breakpoints.Add(address);

public void RemoveBreakpoint(ushort address) => _breakpoints.Remove(address);

private void AddDevice(string path)
{
    _deviceValidator.ThrowIfInvalid(path);
    _devicesManager.Add(path);
}

private void HandleHardwareTrap(Exception e)
{
    ushort address;

    if (e is BusException or OddAddressException)
    {
        if (_trapStack.Any(t => _trapsToHalt.Any(m => m == t)))
        {
            throw new HaltException(false,
                $"Get bus error while already in trap. Trap stack: {string.Join("->", _trapStack)}");
        }

        address = 4;
    }
    else if (e is InvalidInstructionException)
    {
        address = 4;
    }
    else if (e is ReservedInstructionException)
    {
        address = 8;
    }
    else
    {
        throw new Exception($"Unknown error '{e.GetType()}', '{e.Message}'");
    }

    HandleInterrupt(e.GetType().Name, address);
}

private void HandleInterrupt(string name, ushort address)
{
    TrapInstruction.HandleInterrupt(_bus, _state, address);
    _trapStack.Push(name);
}
}

```

3.2 Текст класса CommandParser

```

public class CommandParser
{
    private readonly ushort[] _masks =
    {
        //FEDC_BA98_7654_3210
        0b1111_1111_1111_1111, // halt, wait, reset, rtt, rti, iot, bpt
        0b1111_1111_1111_1000, // rts
        0b1111_1111_1110_0000, // flag instruction
    }
}

```

```

        0b1111_1111_1100_0000, // one operand, mark
        0b1111_1111_0000_0000, // branch, trap, emt
        0b1111_1110_0000_0000, // jsr, sob, mul, div, ash
        0b1111_0000_0000_0000, // two operand
    };

    private readonly Dictionary<ushort, ICommand> _opcodesDictionary;

    public CommandParser(IStorage storage, IState state)
    {
        _opcodesDictionary = Assembly.GetExecutingAssembly().GetTypes()
            .Where(type => typeof(ICommand).IsAssignableFrom(type) && !
type.IsAbstract)
            .Select(commandType => Activator.CreateInstance(commandType, storage,
state) as ICommand)
            .ToDictionary(command => command!.OperationCode);
    }

    public ICommand GetCommand(ushort word)
    {
        foreach (var mask in _masks)
        {
            var opcode = (ushort)(word & mask);

            if (_opcodesDictionary.TryGetValue(opcode, out var command))
            {
                return command;
            }
        }

        throw new ReservedInstructionException(word);
    }
}

```

3.3 Текст классов аргументов

```

public abstract class BaseRegisterArgument<TValue> : IRegisterArgument<TValue>
{
    private readonly Lazy<ushort?> _address;

    protected BaseRegisterArgument(IStorage storage, IState state, ushort mode,
ushort register)
    {
        Storage = storage;
        State = state;
        Mode = mode;
        Register = register;
        _address = new Lazy<ushort?>(InitAddress);
    }

    public object GetValue() => Value;
    public void SetValue(object obj) => Value = (TValue)obj;

    public ushort Register { get; }
    public ushort Mode { get; }
    public abstract TValue Value { get; set; }
    public ushort? Address => _address.Value;
    protected abstract ushort Delta { get; }

    protected IStorage Storage { get; }
    protected IState State { get; }
}

```



```

private ushort? InitAddress()
{
    ushort offset;
    ushort address;

    switch (Mode)
    {
        case 0:
            return null;
        case 1:
            return State.Registers[Register];
        case 2:
            address = State.Registers[Register];
            State.Registers[Register] += Delta;
            return address;
        case 3:
            address = Storage.GetWord(State.Registers[Register]);
            State.Registers[Register] += 2;
            return address;
        case 4:
            State.Registers[Register] -= Delta;
            return State.Registers[Register];
        case 5:
            State.Registers[Register] -= 2;
            return Storage.GetWord(State.Registers[Register]);
        case 6:
            offset = Storage.GetWord(State.Registers[7]);
            State.Registers[7] += 2;
            return (ushort)(State.Registers[Register] + offset);
        case 7:
            offset = Storage.GetWord(State.Registers[7]);
            State.Registers[7] += 2;
            return Storage.GetWord((ushort)(State.Registers[Register] + offset));
        default:
            throw new InvalidOperationException("Invalid addressing mode");
    }
}

}

public class FlagArgument : IArgument
{
    public FlagArgument(ushort word)
    {
        C = (word & 1) != 0;
        V = (word & 2) != 0;
        Z = (word & 4) != 0;
        N = (word & 8) != 0;
        ToSet = (word & 16) != 0;
    }

    public object GetValue() => (ToSet, N, Z, V, C);
    public void SetValue(object obj) => throw new
ReadOnlyArgumentException(GetType());

    public bool ToSet { get; }
    public bool C { get; }
    public bool V { get; }
    public bool Z { get; }
    public bool N { get; }
}

```

```

public class MarkArgument : IArgument
{
    public MarkArgument(ushort number)
    {
        Number = number;
    }

    public object GetValue() => Number;
    public void SetValue(object value) => throw new
ReadOnlyArgumentException(GetType());

    public ushort Number { get; }
}

public class OffsetArgument : IOffsetArgument
{
    public object GetValue() => Offset;
    public void SetValue(object obj) => throw new
ReadOnlyArgumentException(typeof(OffsetArgument));

    public sbyte Offset { get; }

    public OffsetArgument(sbyte offset)
    {
        Offset = offset;
    }
}

public class RegisterWordArgument : BaseRegisterArgument<ushort>
{
    public RegisterWordArgument(IStorage storage, IState state, ushort mode, ushort
register)
        : base(storage, state, mode, register)
    {
    }

    public override ushort Value
    {
        get => !Address.HasValue ? State.Registers[Register] :
Storage.GetWord(Address.Value);
        set
        {
            if (!Address.HasValue)
            {
                State.Registers[Register] = value;
                return;
            }

            Storage.SetWord(Address!.Value, value);
        }
    }

    protected override ushort Delta => 2;
}

public class RegisterByteArgument : BaseRegisterArgument<byte>
{
    public RegisterByteArgument(IStorage storage, IState state, ushort mode, ushort
register)
        : base(storage, state, mode, register)

```

```

    {
    }

    public override byte Value
    {
        get => !Address.HasValue ? (byte)(State.Registers[Register] & 0xFF) :
Storage.GetByte(Address.Value);
        set
        {
            if (!Address.HasValue)
            {
                State.Registers[Register] = (ushort)((State.Registers[Register] &
0xFF00) | value);
                return;
            }

            Storage.SetByte(Address!.Value, value);
        }
    }

    protected override ushort Delta => (ushort)(Register < 6 ? 1 : 2);
}

public class SobArgument : IArgument
{
    public SobArgument(ushort register, byte offset)
    {
        Register = register;
        Offset = offset;
    }

    public object GetValue() => (Register, Offset);
    public void SetValue(object word) => throw new
ReadOnlyArgumentException(typeof(SobArgument));

    public ushort Register { get; }
    public byte Offset { get; }
}

```

3.4 Текст классов команд

4 Текст модуля Внешних устройств

4.1 Текст класса DevicesManager

```

public sealed class DevicesManager : IDevicesManager
{
    private List<IDeviceContext> _contexts = new();
    private readonly IDeviceProvider _provider;

    private List<IDeviceContext> SafeContexts => _contexts ?? throw new
ObjectDisposedException("Manager is disposed");

    public DevicesManager(IDeviceProvider provider)
    {
        _provider = provider;
    }

    public IReadOnlyCollection<IDevice> Devices => SafeContexts.SelectMany(d =>

```

```

d.Devices).ToList();

public void Add(string devicePath)
{
    if (SafeContexts.SingleOrDefault(d => d.AssemblyPath == devicePath) != null)
    {
        return;
    }

    var device = _provider.Load(devicePath);

    SafeContexts.Add(device);
}

public void Remove(string devicePath)
{
    var model = SafeContexts.SingleOrDefault(d => d.AssemblyPath == devicePath);

    if (model == null)
    {
        return;
    }

    model.Dispose();
    SafeContexts.Remove(model);
}

public void Clear()
{
    SafeContexts.ForEach(d => d.Dispose());
    SafeContexts.Clear();
}

public void Dispose()
{
    if (_contexts == null)
    {
        return;
    }

    Clear();
    _contexts = null;
}
}

```

4.2 Текст класу DeviceProvider

```

public class DeviceProvider : IDeviceProvider
{
    private static TType CreateInstance<TType>(Type type, out Exception error) where
TType : class
    {
        try
        {
            var res = Activator.CreateInstance(type) as TType;
            error = null;
            return res;
        }
        catch (Exception e)
        {
            error = e;
        }
    }
}

```

```

        return null;
    }
}

public IDeviceContext Load(string assemblyFilePath)
{
    var context = new AssemblyContext(assemblyFilePath);
    var assembly = context.Load(assemblyFilePath);

    var types = assembly
        .GetExportedTypes()
        .Where(t =>
            t.IsClass && t.GetInterfaces().Any(i => i.FullName ==
typeof(IDevice).FullName))
        .ToList();

    if (!types.Any())
    {
        throw new InvalidOperationException("Cannot find devices");
    }

    var devices = types
        .Select(
            t => CreateInstance<IDevice>(t, out var err)
                ?? throw new InvalidOperationException($"Cannot create instance
of device '{t.FullName}', err));

    return new DeviceContext(context, devices);
}

public bool TryLoad(string assemblyFilePath, out IDeviceContext device)
{
    try
    {
        device = Load(assemblyFilePath);
        return true;
    }
    catch
    {
        device = null;
        return false;
    }
}
}

```

4.3 Текст класса DeviceValidator

```

public class DeviceValidator : IDeviceValidator
{
    private readonly IDeviceProvider _provider;

    public DeviceValidator(IDeviceProvider provider)
    {
        _provider = provider;
    }

    public bool Validate(string path, out string errorMessage)
    {
        try
        {
            _provider.Load(path);

```

```

        errorMessage = null;
        return true;
    }
    catch (Exception e)
    {
        errorMessage = e.Message;
        return false;
    }
}

public void ThrowIfInvalid(string path)
{
    try
    {
        _provider.Load(path);
    }
    catch (Exception e)
    {
        throw new ValidationException($"Device [{path}] is invalid. Error:
{e.Message}", e);
    }
}
}

```

4.4 Текст класса DeviceContext

```

public sealed class DeviceContext : IDeviceContext
{
    private AssemblyContext _context;

    private List<IDevice> _devices;

    public DeviceContext(AssemblyContext context, IEnumerable<IDevice> devices)
    {
        _context = context;
        _devices = devices.ToList();
    }

    public string AssemblyPath =>
        _context?.Assembly.Location ?? throw new ObjectDisposedException("Device is
disposed");

    public IReadOnlyCollection<IDevice> Devices =>
        _devices ?? throw new ObjectDisposedException("Device is disposed");

    public void Dispose()
    {
        _devices?.ForEach(d => d.Dispose());
        _context?.Dispose();
        _devices = null;
        _context = null;
    }
}

```

4.5 Текст интерфейса IDevice

```

public interface IDevice : IDisposable
{
    string Name { get; }

    ushort BufferRegisterAddress { get; }
}

```

```

ushort ControlRegisterAddress { get; }

ushort InterruptVectorAddress { get; }
bool HasInterrupt { get; }

ushort BufferRegisterValue { get; set; }
ushort ControlRegisterValue { get; set; }

int Init();

void AcceptInterrupt();
}

```

5 Текст модуля Графического интерфейса

5.1 Текст класса MainWindowViewModel

```

public class MainWindowViewModel : WindowViewModel<MainWindow>, IMainWindowViewModel
{
    private const string DefaultWindowTitle = "PDP-11 Simulator";
    private const string MainFileName = "main.asm";

    private readonly IFileManager _fileManager;
    private readonly IMessageBoxManager _messageBoxManager;
    private readonly IWindowProvider _windowProvider;
    private readonly ITabManager _tabManager;
    private readonly IProjectManager _projectManager;

    public MainWindowViewModel(MainWindow window, ITabManager tabManager,
        IProjectManager projectManager,
        IFileManager fileManager, IMessageBoxManager messageBoxManager,
        IWindowProvider windowProvider) : base(window)
    {
        CreateFileCommand = ReactiveCommand.CreateFromTask(CreateFileAsync);
        OpenFileCommand = ReactiveCommand.CreateFromTask(OpenFileAsync);
        SaveFileCommand = ReactiveCommand.CreateFromTask<bool>(
            async saveAs => await SaveFileAndUpdateTab(_tabManager!.Tab, saveAs));
        SaveAllFilesCommand = ReactiveCommand.CreateFromTask(SaveAllFilesAsync);
        DeleteFileCommand = ReactiveCommand.CreateFromTask(DeleteFileAsync);
        CreateProjectCommand = ReactiveCommand.CreateFromTask(async () => { await
        CreateProjectAsync(); });
        OpenProjectCommand = ReactiveCommand.CreateFromTask(async () => { await
        OpenProjectAsync(); });
        OpenSettingsWindowCommand =
        ReactiveCommand.CreateFromTask(OpenSettingsWindowAsync);
        OpenExecutorWindowCommand =
        ReactiveCommand.CreateFromTask(OpenExecutorWindowAsync);
        BuildProjectCommand = ReactiveCommand.CreateFromTask(BuildProjectAsync);

        _fileManager = fileManager;
        _messageBoxManager = messageBoxManager;
        _windowProvider = windowProvider;

        _projectManager = projectManager;
        _projectManager.PropertyChanged += (_, args) =>
        {
            if (args.PropertyName == nameof(_projectManager.Project))
            {
                this.RaisePropertyChanged(nameof(WindowTitle));
                OnProjectUpdated();
            }
        }
    }
}

```

```

    };

    _tabManager = tabManager;
    _tabManager.Tabs.CollectionChanged += (_, _) =>
    { this.RaisePropertyChanged(nameof(Tabs)); };
    _tabManager.PropertyChanged += (_, args) =>
    {
        if (args.PropertyName == nameof(_tabManager.Tab))
        {
            this.RaisePropertyChanged(nameof(FileContent));
        }
    };

    window.Closing += OnClosingWindow;
    window.Opened += async (_, _) =>
    {
        if (!await InitProjectAsync())
        {
            View.Close();
        }
    };

    SettingsManager.Instance.PropertyChanged += (_, args) =>
    this.RaisePropertyChanged(args.PropertyName);

    InitContext();
}

public ReactiveCommand<Unit, Unit> CreateFileCommand { get; }
public ReactiveCommand<Unit, Unit> OpenFileCommand { get; }
public ReactiveCommand<bool, Unit> SaveFileCommand { get; }
public ReactiveCommand<Unit, Unit> SaveAllFilesCommand { get; }
public ReactiveCommand<Unit, Unit> DeleteFileCommand { get; }
public ReactiveCommand<Unit, Unit> CreateProjectCommand { get; }
public ReactiveCommand<Unit, Unit> OpenProjectCommand { get; }
public ReactiveCommand<Unit, Unit> OpenSettingsWindowCommand { get; }
public ReactiveCommand<Unit, Unit> OpenExecutorWindowCommand { get; }
public ReactiveCommand<Unit, Unit> BuildProjectCommand { get; }

public string WindowTitle => _projectManager?.IsOpened == true
    ? $"{DefaultWindowTitle} - {_projectManager.Project.ProjectName}"
    : DefaultWindowTitle;

public ObservableCollection<FileTab> Tabs => _tabManager.Tabs.Select(t =>
t.View).ToObservableCollection();

public string FileContent
{
    get => File.Text;
    set
    {
        File.Text = value;
        File.IsNeedSave = true;
        _tabManager.UpdateForeground(_tabManager.Tab);
        this.RaisePropertyChanged();
    }
}

private FileModel File => _tabManager.Tab.File;

private async Task CreateTabForFiles(IEnumerable<FileModel> files)

```



```

{
    IFileTabViewModel tab = null;

    foreach (var file in files)
    {
        try
        {
            tab = _tabManager.CreateTab(file, t =>
            {
                _tabManager.SelectTab(t);
                return Task.CompletedTask;
            }, t => CloseTabAsync(t, true));
        }
        catch (TabExistsException e)
        {
            tab = e.Tab;

            var res = await
            _messageBoxManager.ShowCustomMessageBoxAsync("Warning",
                $"File '{file.FileName}' is already open", Icon.Warning, View,
                Buttons.ReopenButton,
                Buttons.SkipButton);

            if (res == Buttons.ReopenButton.Name)
            {
                e.Tab.File.Text = file.Text;
                if (ReferenceEquals(e.Tab, _tabManager.Tab))
                {
                    this.RaisePropertyChanged(nameof(FileContent));
                }
            }
        }
    }

    if (tab != null)
    {
        _tabManager.SelectTab(tab);
    }
}

private async Task CreateFileAsync()
{
    var file = await _fileManager.CreateFile(View.StorageProvider,
        _projectManager.IsOpened ? _projectManager.Project.ProjectDirectory :
        null, null);

    if (file != null)
    {
        await CreateTabForFiles(new[] { file });
        _projectManager.AddFileToProject(file.FilePath);
        await _projectManager.SaveProjectAsync();
    }
}

private async Task OpenFileAsync()
{
    var files = await _fileManager.OpenFilesAsync(View.StorageProvider);
    await CreateTabForFiles(files);
}

private async Task<bool> SaveFileAsAsync(FileModel file)

```

```

{
    var paths = _tabManager.Tabs
        .Where(t => t.File.FilePath != file.FilePath)
        .Select(t => t.File.FilePath)
        .ToHashSet();

    var options = new FilePickerSaveOptions
    {
        Title = "Save file as...",
        ShowOverwritePrompt = true,
        SuggestedFileName = file.FileName
    };

    do
    {
        var filePath = await _fileManager.GetFilesAsync(View.StorageProvider,
options);

        if (filePath == null)
        {
            return false;
        }

        if (!paths.Contains(filePath))
        {
            file.FilePath = filePath;
            await _fileManager.WriteFileAsync(file);
            return true;
        }

        await _messageBoxManager.ShowErrorMessageBox("That file already opened",
View);
    } while (true);
}

private async Task<bool> SaveProjectFile(FileModel file)
{
    var error = await JsonHelper.ValidateJsonAsync<ProjectDto>(file.Text);

    if (error == null)
    {
        await _fileManager.WriteFileAsync(file);
        await _projectManager.ReloadProjectAsync();
        return true;
    }

    await _messageBoxManager.ShowErrorMessageBox(error, View);

    return false;
}

private bool IsProjectTab(IFileTabViewModel tab) => IsProjectFile(tab.File);
private bool IsProjectFile(FileModel file) =>
    _projectManager.IsOpened && file.FilePath ==
_projectManager.Project.ProjectFile;

private async Task<bool> SaveFileAsync(FileModel file, bool saveAs)
{
    if (IsProjectFile(file))
    {
        if (!saveAs)

```

```

        {
            return await SaveProjectFile(file);
        }

        await _messageBoxManager.ShowErrorMessageBox("This feature is not
available for project file", View);
        return false;
    }

    if (saveAs)
    {
        return await SaveFileAsAsync(file);
    }

    await _fileManager.WriteFileAsync(file);
    return true;
}

private async Task SaveAllFilesAsync()
{
    foreach (var tab in _tabManager.Tabs)
    {
        await SaveFileAndUpdateTab(tab, false);
    }
}

private async Task SaveFileAndUpdateTab(IFileTabViewModel tab, bool saveAs)
{
    if (await SaveFileAsync(tab.File, saveAs))
    {
        _tabManager.UpdateForeground(tab);
        _tabManager.UpdateHeader(tab);
    }
}

private async Task DeleteFileAsync()
{
    if (IsProjectTab(_tabManager.Tab))
    {
        await _messageBoxManager.ShowErrorMessageBox("Cannot delete project
file", View);
        return;
    }

    var res = await _messageBoxManager.ShowMessageBoxAsync("Confirmation",
        $"Are you sure you want to delete the file '{File.FileName}'?",
        ButtonEnum.YesNo, Icon.Question, View);

    if (res == ButtonResult.Yes)
    {
        _projectManager.RemoveFileFromProject(File.FilePath);
        await _projectManager.SaveProjectAsync();
        await _fileManager.DeleteAsync(File);
        _tabManager.DeleteTab(_tabManager.Tab);
    }
}

private async Task CloseTabAsync(IFileTabViewModel tab, bool isUi)
{
    if (IsProjectTab(tab) && isUi)
    {

```

```

        await _messageBoxManager.ShowErrorMessageBox("Cannot close project file",
View);
        return;
    }

    if (tab.File.IsNeedSave)
    {
        var res = await _messageBoxManager.ShowMessageBoxAsync("Confirmation",
            $"Do you want to save the file '{File.FileName}'?", ButtonEnum.YesNo,
Icon.Question, View);

        if (res == ButtonResult.Yes)
        {
            await SaveFileAsync(tab.File, false);
        }
    }

    _tabManager.DeleteTab(tab);
}

private async Task CloseAllTabs()
{
    var tabs = _tabManager.Tabs.ToList();

    foreach (var tab in tabs)
    {
        await CloseTabAsync(tab, false);
    }
}

private async Task<bool> InitProjectAsync()
{
    if (SettingsManager.Instance.CommandLineOptions?.Project != null &&
        await
OpenProjectAsync(SettingsManager.Instance.CommandLineOptions.Project))
    {
        return true;
    }

    while (true)
    {
        var boxRes = await _messageBoxManager.ShowCustomMessageBoxAsync("Init",
"Create or open project", Icon.Info,
            View, Buttons.CreateButton, Buttons.OpenButton, Buttons.CancelButton
        );

        if (boxRes == Buttons.CreateButton.Name && await CreateProjectAsync()
            || boxRes == Buttons.OpenButton.Name && await OpenProjectAsync())
        {
            return true;
        }

        if (boxRes == Buttons.CancelButton.Name || boxRes == null)
        {
            return false;
        }
    }
}

private async Task<bool> NewProjectValidation()
{

```

```

        if (!Tabs.Any())
        {
            return true;
        }

        var res = await _messageBoxManager
            .ShowMessageBoxAsync("Warning", "This action closes current project and
all tabs",
            ButtonEnum.OkAbort, Icon.Warning, View);

        return res == ButtonResult.Ok;
    }

    private async Task OpenProjectFilesAsync()
    {
        await CloseAllTabs();

        var projectFile = await
            _fileManager.OpenFileAsync(_projectManager.Project.ProjectFile);

        var files = new List<FileModel> { projectFile };

        foreach (var filePath in _projectManager.Project.Files)
        {
            try
            {
                var file = await _fileManager.OpenFileAsync(filePath);
                files.Add(file);
            }
            catch (FileNotFoundException e)
            {
                await _messageBoxManager.ShowErrorMessageBox($"{e.Message} Skipping
it.", View);
            }
        }

        await CreateTabForFiles(files);
    }

    private async Task<bool> CreateProjectAsync()
    {
        if (!await NewProjectValidation())
        {
            return false;
        }

        bool successCreation;
        while (true)
        {
            var (res, projectName) = await
                _messageBoxManager.ShowInputMessageBoxAsync("Create project",
                    "Enter project name", ButtonEnum.OkCancel, Icon.Setting, View,
                    "Project name");

            if (res == ButtonResult.Cancel)
            {
                return false;
            }

            try
            {

```

```

        successCreation = await
        _projectManager.CreateProjectAsync(View.StorageProvider, projectName.Trim());
    }
    catch (ArgumentException e)
    {
        await _messageBoxManager.ShowErrorMessageBox(e.Message, View);
        continue;
    }

    break;
}

if (!successCreation)
{
    return false;
}

var mainFile = new FileModel
{
    FilePath = PathHelper.Combine(_projectManager.Project.ProjectDirectory,
MainFileName)
};
await _fileManager.WriteFileAsync(mainFile);
_projectManager.AddFileToProject(mainFile.FilePath);
_projectManager.SetExecutableFile(mainFile.FilePath);
await _projectManager.SaveProjectAsync();

await OpenProjectFilesAsync();
return true;
}

private async Task<bool> OpenProjectAsync(string projectPath = null)
{
    if (!await NewProjectValidation())
    {
        return false;
    }

    try
    {
        if (projectPath != null)
        {
            try
            {
                await _projectManager.LoadProjectAsync(projectPath);
                await OpenProjectFilesAsync();
                return true;
            }
            catch (Exception e)
            {
                await _messageBoxManager.ShowErrorMessageBox(e.Message, View);
            }
        }

        if (await _projectManager.OpenProjectAsync(View.StorageProvider))
        {
            await OpenProjectFilesAsync();
            return true;
        }
    }
    catch (Exception e)

```

```

        {
            await _messageBoxManager.ShowErrorMessageBox(e.Message, View);
            return false;
        }

        return false;
    }

    private async Task OpenSettingsWindowAsync()
    {
        var viewModel = _windowProvider.CreateWindow<SettingsWindow,
SettingsViewModel>(_projectManager, _fileManager,
        new DeviceValidator(new DeviceProvider()), _messageBoxManager);
        await viewModel.ShowDialog(View);
    }

    private async Task OpenExecutorWindowAsync()
    {
        var executor = new Executor.Executor();
        await executor.LoadProgram( _projectManager.Project);

        var viewModel = _windowProvider.CreateWindow<ExecutorWindow,
ExecutorViewModel>(_messageBoxManager, executor);
        await viewModel.ShowDialog(View);
    }

    private async void OnClosingWindow(object sender, WindowClosingEventArgs args)
    {
        args.Cancel = true;

        if (_tabManager.Tabs.Any(t => t.File.IsNeedSave))
        {
            var res = await _messageBoxManager.ShowMessageBoxAsync("Warning",
                "You have unsaved files. Save all of them?", ButtonEnum.YesNoCancel,
Icon.Warning, View);

            if (res == ButtonResult.Cancel)
            {
                return;
            }

            if (res == ButtonResult.Yes)
            {
                await SaveAllFilesAsync();
            }
        }

        View.Closing -= OnClosingWindow;
        View.Close();
    }

    private async void OnProjectUpdated()
    {
        if (!_projectManager.IsOpened)
        {
            return;
        }

        var projectTab = _tabManager.Tabs.SingleOrDefault(IsProjectTab);
        if (projectTab != null)
        {

```

```

        var fileOnDisk = await
_fileManager.OpenFileAsync(projectTab.File.FilePath);
        projectTab.File.Text = fileOnDisk.Text;
        this.RaisePropertyChanged(nameof(FileContent));
    }
}

private async Task BuildProjectAsync()
{
    await SaveAllFilesAsync();

    var assembler = new Compiler();

    try
    {
        await assembler.Compile(_projectManager.Project);
        await _messageBoxManager.ShowMessageBoxAsync("Build", "Completed",
ButtonEnum.Ok, Icon.Info, View);
    }
    catch (Exception e)
    {
        await _messageBoxManager.ShowErrorMessageBox(e.Message, View);
    }
}
}

```

5.2 Текст класу SettingsViewModel

```

public class SettingsViewModel : WindowViewModel<SettingsWindow>, ISettingsViewModel
{
    private readonly IProjectManager _projectManager;
    private readonly IFileManager _fileManager;
    private readonly IDeviceValidator _deviceValidator;
    private readonly IMessageBoxManager _messageBoxManager;

    public SettingsViewModel(SettingsWindow window, IProjectManager projectManager,
IFileManager fileManager,
    IDeviceValidator deviceValidator, IMessageBoxManager messageBoxManager) :
    base(window)
    {
        _projectManager = projectManager;
        _fileManager = fileManager;
        _deviceValidator = deviceValidator;
        _messageBoxManager = messageBoxManager;

        AddDeviceCommand = ReactiveCommand.CreateFromTask(AddDeviceAsync);
        DeleteDeviceCommand = ReactiveCommand.CreateFromTask(DeleteDevices);
        ValidateDevicesCommand =
            ReactiveCommand.CreateFromTask(() =>
ValidateDevices(SelectedDevices.Any() ? SelectedDevices : Devices));

        projectManager.PropertyChanged += ProjectPropertyChanged;

        window.Closed += async (_, _) =>
        {
            projectManager.PropertyChanged -= ProjectPropertyChanged;
            await SettingsManager.Instance.SaveGlobalSettingsAsync();
        };

        InitContext();
    }
}

```



```

public ReactiveCommand<Unit, Unit> AddDeviceCommand { get; }
public ReactiveCommand<Unit, Unit> DeleteDeviceCommand { get; }
public ReactiveCommand<Unit, Unit> ValidateDevicesCommand { get; }

public ObservableCollection<string> Devices => (_projectManager.IsOpened
    ? _projectManager.Project.Devices
    : Array.Empty<string>()).ToObservableCollection();

public ObservableCollection<string> SelectedDevices { get; set; } = new();

private async Task AddDeviceAsync()
{
    var options = new FilePickerOpenOptions
    {
        Title = "Open device library...",
        AllowMultiple = false,
        FileTypeFilter = new[] { new FilePickerFileType("DLL") { Patterns = new[]
{ "*.dll" } } }
    };

    var file = await _fileManager.GetFileAsync(View.StorageProvider, options);

    if (file == null)
    {
        return;
    }

    try
    {
        _projectManager.AddDeviceToProject(file);
        await _projectManager.SaveProjectAsync();
    }
    catch (ValidationException e)
    {
        await _messageBoxManager.ShowErrorMessageBox(e.Message, View);
    }
}

private async Task DeleteDevices()
{
    var devices = SelectedDevices.ToList();
    foreach (var device in devices)
    {
        _projectManager.RemoveDeviceFromProject(device);
    }

    await _projectManager.SaveProjectAsync();
}

private async Task ValidateDevices(IEnumerable<string> devices)
{
    foreach (var device in devices)
    {
        try
        {
            _deviceValidator.ThrowIfInvalid(device);
        }
        catch (ValidationException e)
        {
            await _messageBoxManager.ShowErrorMessageBox(e.Message, View);
        }
    }
}

```

```

        }
    }
}

private void ProjectPropertyChanged(object sender, PropertyChangedEventArgs args)
{
    if (args.PropertyName is nameof(_projectManager.Project) or
        nameof(_projectManager.Project.Devices))
    {
        this.RaisePropertyChanged(nameof(Devices));
    }
}
}

```

5.3 Текст класса FileTabViewModel

```

public class FileTabViewModel : BaseViewModel<FileTab>, IFileTabViewModel
{
    public static readonly IBrush DefaultBackground = new
    SolidColorBrush(Colors.White);
    public static readonly IBrush SelectedBackground = new
    SolidColorBrush(Colors.LightGray, 0.5D);

    public static readonly IBrush DefaultForeground = new
    SolidColorBrush(Colors.Black);
    public static readonly IBrush NeedSaveForeground = new
    SolidColorBrush(Colors.DodgerBlue);

    private IBrush _currentBackground;

    public FileTabViewModel(FileTab fileTab, FileModel file, Func<FileTabViewModel,
    Task> selectCommand,
    Func<FileTabViewModel, Task> closeCommand) : base(fileTab)
    {
        File = file;
        TabBackground = DefaultBackground;
        SelectTabCommand = ReactiveCommand.CreateFromTask(async () => await
        selectCommand(this));
        CloseTabCommand = ReactiveCommand.CreateFromTask(async () => await
        closeCommand(this));

        InitContext();
    }

    public FileModel File { get; }

    public string TabHeader => File.FileName;

    public IBrush TabForeground => File.IsNeedSave ? NeedSaveForeground :
    DefaultForeground;

    public IBrush TabBackground
    {
        get => _currentBackground;
        set => this.RaiseAndSetIfChanged(ref _currentBackground, value);
    }

    public bool IsSelected
    {
        get => ReferenceEquals(TabBackground, SelectedBackground);
        set => TabBackground = value ? SelectedBackground : DefaultBackground;
    }
}

```

```

    }

    public ReactiveCommand<Unit, Unit> SelectTabCommand { get; }
    public ReactiveCommand<Unit, Unit> CloseTabCommand { get; }

    public void NotifyHeaderChanged()
    {
        this.RaisePropertyChanged(nameof(TabHeader));
    }

    public void NotifyForegroundChanged()
    {
        this.RaisePropertyChanged(nameof(TabForeground));
    }
}

```

5.4 Текст класса ExecutorWindowViewModel

```

public class ExecutorViewModel : WindowViewModel<ExecutorWindow>,
    IExecutorWindowViewModel
{
    private readonly IMessageBoxManager _messageBoxManager;
    private readonly Executor.Executor _executor;
    private bool _memoryAsWord = true;
    private Tab _currentTab = Tab.State;

    private ObservableCollection<IMemoryModel> _memory;
    private ObservableCollection<CodeModel> _code;
    private CancellationTokenSource _cancelRunToken;

    public ExecutorViewModel(ExecutorWindow view, IMessageBoxManager
messageBoxManager, Executor.Executor executor) :
        base(view)
    {
        _messageBoxManager = messageBoxManager;
        _executor = executor;

        StartExecutionCommand = ReactiveCommand.CreateFromTask(RunAsync);
        PauseExecutionCommand = ReactiveCommand.Create(PauseAsync);
        MakeStepCommand = ReactiveCommand.CreateFromTask(MakeStepAsync);
        ResetExecutorCommand = ReactiveCommand.CreateFromTask(ResetExecutorAsync);
        ChangeMemoryModeCommand = ReactiveCommand.Create(ChangeMemoryMode);
        FindAddressCommand =
ReactiveCommand.CreateFromTask<string>(FindAddressAsync);

        Tabs = Enum.GetValues<Tab>().ToObservableCollection();
        Memory = AsWords().ToObservableCollection();
        CodeLines = InitCode().ToObservableCollection();

        InitContext();

        View.CodeGrid.SelectedIndex = 0;
    }

    public ReactiveCommand<Unit, Unit> StartExecutionCommand { get; }
    public ReactiveCommand<Unit, Unit> PauseExecutionCommand { get; }
    public ReactiveCommand<Unit, Unit> MakeStepCommand { get; }
    public ReactiveCommand<Unit, Unit> ResetExecutorCommand { get; }
    public ReactiveCommand<Unit, Unit> ChangeMemoryModeCommand { get; }
    public ReactiveCommand<string, Unit> FindAddressCommand { get; }
}

```

```

    public ObservableCollection<RegisterModel> Registers =>
        _executor.Registers.Select((m, i) => new RegisterModel($"R{i}",
m)).ToObservableCollection();

    public ObservableCollection<ProcessorStateWordModel> ProcessorStateWord =>
        new[] { new
ProcessorStateWordModel(_executor.ProcessorStateWord) }.ToObservableCollection();

    public ObservableCollection<IMemoryModel> Memory
    {
        get => _memory;
        set => this.RaiseAndSetIfChanged(ref _memory, value);
    }

    public ObservableCollection<Device> Devices => _executor.Devices
        .Select(m => new Device(
            m.Name,
            m.ControlRegisterAddress,
            m.ControlRegisterValue,
            m.BufferRegisterAddress,
            m.BufferRegisterValue,
            m.InterruptVectorAddress))
        .ToObservableCollection();

    public ObservableCollection<CodeModel> CodeLines
    {
        get => _code;
        set => this.RaiseAndSetIfChanged(ref _code, value);
    }

    public ObservableCollection<Tab> Tabs { get; }

    public string ChangeMemoryModeCommandHeader => _memoryAsWord ? "As Bytes" : "As
Word";

    public Tab CurrentTab
    {
        get => _currentTab;
        set
        {
            _currentTab = value;
            this.RaisePropertyChanged(nameof(IsStateVisible));
            this.RaisePropertyChanged(nameof(IsMemoryVisible));
            this.RaisePropertyChanged(nameof(IsDevicesVisible));
        }
    }

    public bool IsStateVisible => CurrentTab == Tab.State;

    public bool IsMemoryVisible => CurrentTab == Tab.Memory;

    public bool IsDevicesVisible => CurrentTab == Tab.Devices;

    private Task FastStep() => Task.Run(() => _executor.ExecuteNextInstruction());

    private async Task MakeStepAsync()
    {
        try
        {
            await FastStep();
            UpdateState();
        }
    }

```

```

    }
    catch (HaltException e) when (e.IsExpected)
    {
        await _messageBoxManager.ShowMessageBoxAsync("Execute", "HALT is
executed", ButtonEnum.Ok, Icon.Info,
            View);
    }
    catch (Exception e)
    {
        await _messageBoxManager.ShowErrorMessageBox(e.Message, View);
    }
}

private async Task RunAsync()
{
    _cancelRunToken = new CancellationTokenSource();

    while (!_cancelRunToken.IsCancellationRequested)
    {
        try
        {
            await FastStep();
        }
        catch (HaltException e) when (e.IsExpected)
        {
            await _messageBoxManager.ShowMessageBoxAsync("Execute", "HALT is
executed", ButtonEnum.Ok, Icon.Info,
                View);
            break;
        }
        catch (Exception e)
        {
            await _messageBoxManager.ShowErrorMessageBox(e.Message, View);
        }
    }

    _cancelRunToken.Dispose();
    _cancelRunToken = null;

    UpdateState();
}

private void PauseAsync() => _cancelRunToken?.Cancel();

private async Task ResetExecutorAsync()
{
    await _executor.Reload();
    UpdateState();
}

private void ChangeMemoryMode()
{
    _memoryAsWord = !_memoryAsWord;
    this.RaisePropertyChanged(nameof(ChangeMemoryModeCommandHeader));
    Memory = _memoryAsWord ? AsWords().ToObservableCollection() :
AsBytes().ToObservableCollection();
}

private IEnumerable<IMemoryModel> AsWords()
{
    var count = _executor.Memory.Data.Count;

```

```

        for (ushort i = 0; i < count; i += 2)
        {
            yield return new WordModel(i, _executor.Memory.GetWord(i));
        }
    }

    private IEnumerable<IMemoryModel> AsBytes()
        => _executor.Memory.Data.Select((m, i) => new ByteModel((ushort)i, m));

    private async Task FindAddressAsync(string text)
    {
        var converter = new NumberStringConverter();
        var address = await converter.ConvertAsync(text);

        if (_memoryAsWord)
        {
            if (address % 2 == 1)
            {
                await _messageBoxManager.ShowErrorMessageBox("Word address must be
even", View);
                return;
            }

            address /= 2;
        }

        View.MemoryGrid.SelectedIndex = address;
        View.MemoryGrid.ScrollIntoView(View.MemoryGrid.SelectedItem,
View.MemoryGrid.Columns.FirstOrDefault());
    }

    private IEnumerable<CodeModel> InitCode()
    {
        var start = _executor.Project.ProgramAddress;
        var count = _executor.LengthOfProgram;

        for (var i = start; i <= start + count; i += 2)
        {
            yield return new CodeModel(i, _executor.Memory.GetWord(i), string.Empty);
        }
    }

    private void UpdateState()
    {
        CodeLines = InitCode().ToObservableCollection();
        Memory = _memoryAsWord ? AsWords().ToObservableCollection() :
AsBytes().ToObservableCollection();
        this.RaisePropertyChanged(nameof(Registers));
        this.RaisePropertyChanged(nameof(ProcessorStateWord));
        this.RaisePropertyChanged(nameof(Devices));
        var line = CodeLines.FirstOrDefault(m =>
            Convert.ToUInt16(m.Address, 8) == _executor.Registers.ElementAt(7));
        if (line != null)
        {
            var index = CodeLines.IndexOf(line);

            View.CodeGrid.SelectedIndex = index;
        }
        else
        {
            View.CodeGrid.SelectedIndex = -1;
        }
    }

```

```

    }
}

```

5.5 Текст класса ProjectManager

```

public class ProjectManager : PropertyChangedNotifier, IProjectManager
{
    public const string ProjectExtension = "pdp11proj";

    private readonly IProjectProvider _provider;
    private readonly IDeviceValidator _deviceValidator;
    private Project _project;

    private Project SafeProject => _project ?? throw new
InvalidOperationException("Project is not opened");

    public ProjectManager(IProjectProvider provider, IDeviceValidator
deviceValidator)
    {
        _provider = provider ?? throw new ArgumentNullException(nameof(provider));
        _deviceValidator = deviceValidator ?? throw new
ArgumentNullException(nameof(deviceValidator));
    }

    public IProject Project
    {
        get => SafeProject;
        private set => SetField(ref _project, value as Project);
    }

    public bool IsOpened => _project != null;

    public async Task<bool> CreateProjectAsync(IStorageProvider storageProvider,
string projectName)
    {
        if (storageProvider == null)
        {
            throw new ArgumentNullException(nameof(storageProvider));
        }

        if (string.IsNullOrEmpty(projectName))
        {
            throw new ArgumentException("Project name cannot be empty",
nameof(projectName));
        }

        var projectDir = await storageProvider.OpenFolderPickerAsync(new
FolderPickerOpenOptions
        {
            Title = "Choose project folder...",
            AllowMultiple = false
        });

        if (!projectDir.Any())
        {
            return false;
        }

        var filePath =
            PathHelper.Combine(projectDir[0].Path.LocalPath, $"{projectName}.

```

```

{ProjectExtension}");
    var project = new Project
    {
        ProjectFile = filePath
    };

    await project.ToJsonAsync();
    Project = project;

    return true;
}

public async Task<bool> OpenProjectAsync(IStorageProvider storageProvider)
{
    if (storageProvider == null)
    {
        throw new ArgumentNullException(nameof(storageProvider));
    }

    var projectFile = await storageProvider.OpenFilePickerAsync(new
FilePickerOpenOptions
    {
        Title = "Open project file...",
        AllowMultiple = false,
        FileTypeFilter = new[]
        {
            new FilePickerFileType(ProjectExtension)
            {
                Patterns = new[] { $"*.{ProjectExtension}" }
            }
        }
    });

    if (!projectFile.Any())
    {
        return false;
    }

    await LoadProjectAsync(projectFile[0].Path.LocalPath);
    return true;
}

public async Task LoadProjectAsync(string projectFilePath)
{
    Project = await _provider.OpenProjectAsync(projectFilePath);
}

public Task ReloadProjectAsync() => LoadProjectAsync(SafeProject.ProjectFile);

public async Task SaveProjectAsync()
{
    await SafeProject.ToJsonAsync();
    OnPropertyChanged(nameof(Project));
}

public void AddFileToProject(string filePath)
{
    filePath = PathHelper.GetFullPath(filePath);
    if (SafeProject.Files.Contains(filePath))
    {
        return;
    }
}

```



```

    }

    SafeProject.Files.Add(filePath);
    OnPropertyChanged(nameof(SafeProject.Files));
}

public void RemoveFileFromProject(string filePath)
{
    filePath = PathHelper.GetFullPath(filePath);

    if (SafeProject.Executable == filePath)
    {
        SafeProject.Executable = string.Empty;
        OnPropertyChanged(nameof(SafeProject.Executable));
    }

    SafeProject.Files.Remove(filePath);
    OnPropertyChanged(nameof(SafeProject.Files));
}

public void SetExecutableFile(string filePath)
{
    filePath = PathHelper.GetFullPath(filePath);
    if (SafeProject.Files.Contains(filePath))
    {
        SafeProject.Executable = filePath;
        OnPropertyChanged(nameof(SafeProject.Executable));
    }
    else
    {
        throw new ArgumentException($"The file '{filePath}' does not belong to
the project", nameof(filePath));
    }
}

public void AddDeviceToProject(string filePath)
{
    filePath = PathHelper.GetFullPath(filePath);
    if (SafeProject.Devices.Contains(filePath))
    {
        return;
    }

    _deviceValidator.ThrowIfInvalid(filePath);

    SafeProject.Devices.Add(filePath);
    OnPropertyChanged(nameof(SafeProject.Devices));
}

public void RemoveDeviceFromProject(string filePath)
{
    filePath = PathHelper.GetFullPath(filePath);
    SafeProject.Devices.Remove(filePath);
    OnPropertyChanged(nameof(SafeProject.Devices));
}
}

```

5.6 Текст класса FileManager

```

public class FileManager : IFileManager
{

```

```

    public async Task<string> GetFileAsync(IStorageProvider storageProvider,
PickerOptions options)
    {
        if (storageProvider == null)
        {
            throw new ArgumentNullException(nameof(storageProvider));
        }

        switch (options)
        {
            case FilePickerSaveOptions saveOptions:
            {
                var newFile = await storageProvider.SaveFilePickerAsync(saveOptions);
                return newFile?.Path.LocalPath;
            }
            case FilePickerOpenOptions { AllowMultiple: true }:
                throw new
InvalidOperationException($"{nameof(FilePickerOpenOptions.AllowMultiple)} must be
false");
            case FilePickerOpenOptions openOptions:
            {
                var file = await storageProvider.OpenFilePickerAsync(openOptions);
                return file.Any() ? file[0].Path.LocalPath : null;
            }
            default:
                throw new InvalidOperationException($"Invalid type of
{nameof(options)} - {options.GetType().Name}");
        }
    }

    public async Task<FileModel> CreateFile(IStorageProvider storageProvider, string
directoryPath, string fileName)
    {
        if (storageProvider == null)
        {
            throw new ArgumentNullException(nameof(storageProvider));
        }

        var options = new FilePickerSaveOptions
        {
            Title = "Create file...",
            ShowOverwritePrompt = true,
            SuggestedFileName = fileName,
            SuggestedStartLocation = await
storageProvider.TryGetFolderFromPathAsync(directoryPath)
        };

        var filePath = await GetFileAsync(storageProvider, options);

        if (filePath == null)
        {
            return null;
        }

        var file = new FileModel
        {
            FilePath = filePath
        };

        await WriteFileAsync(file);
    }

```

```

        return file;
    }

    public async Task<ICollection<FileModel>> OpenFilesAsync(IStorageProvider
storageProvider)
    {
        if (storageProvider == null)
        {
            throw new ArgumentNullException(nameof(storageProvider));
        }

        var files = await storageProvider.OpenFilePickerAsync(new
FilePickerOpenOptions
        {
            Title = "Open files...",
            AllowMultiple = true
        });

        if (!files.Any())
        {
            return Array.Empty<FileModel>();
        }

        var filesList = new List<FileModel>();

        foreach (var file in files)
        {
            filesList.Add(await OpenFileAsync(file.Path.LocalPath));
        }

        return filesList;
    }

    public async Task<FileModel> OpenFileAsync(string filePath) => new()
    {
        FilePath = filePath,
        Text = await File.ReadAllTextAsync(filePath)
    };

    public async Task WriteFileAsync(FileModel file)
    {
        await File.WriteAllTextAsync(file.FilePath, file.Text);
        file.IsNeedSave = false;
    }

    public Task DeleteAsync(FileModel file) => Task.Run(() =>
File.Delete(file.FilePath));
}

```

5.7 Текст класса FileModel

```

public record FileModel
{
    public string FilePath { get; set; }
    public string FileName => PathHelper.GetFileName(FilePath);
    public string Text { get; set; } = string.Empty;
    public bool IsNeedSave { get; set; }
}

```

5.8 Текст класса SettingsManager

```
public sealed class SettingsManager : PropertyChangedNotifier
{
    private FontFamily _fontFamily;
    private double _fontSize;

    public CommandLineOptions CommandLineOptions { get; }

    public FontFamily FontFamily
    {
        get => _fontFamily;
        set => SetField(ref _fontFamily, value);
    }

    public double FontSize
    {
        get => _fontSize;
        set => SetField(ref _fontSize, value);
    }

    public static ObservableCollection<FontFamily> AllFontFamilies =>
        FontManager.Current.SystemFonts.ToObservableCollection();

    public static SettingsManager Instance { get; private set; }

    private SettingsManager(EditorOptions options, CommandLineOptions
commandLineOptions)
    {
        FontFamily = new FontFamily(options.FontFamily);
        FontSize = options.FontSize;
        CommandLineOptions = commandLineOptions;
    }

    public static void Create(EditorOptions editorOptions, CommandLineOptions
commandLineOptions)
    {
        Instance ??= new SettingsManager(editorOptions, commandLineOptions);
    }

    public async Task SaveGlobalSettingsAsync()
    {
        await ConfigurationHelper.SaveToJson(new Dictionary<string, object>
        {
            {
                nameof(EditorOptions), new EditorOptions
                {
                    FontFamily = FontFamily.Name,
                    FontSize = FontSize
                }
            }
        });
    }
}
```

5.9 Текст класса TabManager

```
public class TabManager : PropertyChangedNotifier, ITabManager
{
    private FileTabViewModel _tab;
```

```

public IFileTabViewModel Tab
{
    get => _tab;
    set => SetField(ref _tab, value as FileTabViewModel);
}

public ObservableCollection<IFileTabViewModel> Tabs { get; } = new();

public IFileTabViewModel CreateTab(FileModel file, Func<IFileTabViewModel, Task>
selectCommand,
    Func<IFileTabViewModel, Task> closeCommand)
{
    if (file != null)
    {
        var existingTab = Tabs.SingleOrDefault(t => t.File.FilePath ==
file.FilePath);
        if (existingTab != null)
        {
            throw new TabExistsException("Tab for that file already exists")
            {
                Tab = existingTab
            };
        }
    }

    var viewModel = new FileTabViewModel(new FileTab(), file ?? new FileModel(),
selectCommand, closeCommand);
    Tabs.Add(viewModel);
    return viewModel;
}

public void DeleteTab(IFileTabViewModel tab)
{
    var index = Tabs.IndexOf(tab) - 1;

    Tabs.Remove(tab);

    var tabToSelect = Tabs.ElementAtOrDefault(index == -1 ? 0 : index);
    SelectTab(tabToSelect);
}

public void SelectTab(IFileTabViewModel tab)
{
    if (_tab != null)
    {
        _tab.IsSelected = false;
    }

    Tab = tab;

    if (_tab != null)
    {
        _tab.IsSelected = true;
    }
}

public void UpdateForeground(IFileTabViewModel tab)
{
    (tab as FileTabViewModel)?.NotifyForegroundChanged();
}

```

```
public void UpdateHeader(IFileTabViewModel tab)
{
    (tab as FileTabViewModel)?.NotifyHeaderChanged();
}
}
```

Лист регистрации изменений

[illegible]