Design and Implementation of a 16-bit Breadboard Computer Architecture 6CCS3PRJ

Appendix

Author: Luca-Dorin Anton

Supervisor: **Dr. Christian Urban**

Student ID: 1710700

Programme of Study: **MSci Computer Science**

April 24, 2020

Contents

1 Extra Information **1**

1.1 Digital Circuits Basics 1

2 Images **5**

3 Source Code **12**

3.1 Originality Pledge 12

3.2 MicroCode generator 12

3.3 Assembler 48

3.4 Compiler 67

3.5 Assembler and Compiler Script 81

3.6 I/O Driver 84

3.7 Programming Driver 90

Bibliography 95

Chapter 1

Extra Information

This part of the appendix contains additional informations which might be useful in case the interested reader would like to dive deeper into the topics and tools presented and possibly get hands-on himself.

1.1 Digital Circuits Basics

For an in-depth understanding of a computer to be complete, one would need to understand how each individual component works, down to the transistor. Once a through understanding has been established, designing a new component seems almost trivial, as there is almost always a right combination of primitive tools" available to do the job. Here is a listing and description of those tools.

1.1.1 Types of electrical circuits

Electrical circuits can be broadly classified into two main categories: *combinatorial* and *sequential* circuits. *Combinatorial circuits* are circuits without any internal state (no memory) which implement a certain logic function. A logic function maps binary inputs to binary outputs. They are implemented as cascading layers of *logic gates.* The main techniques which can be used to transform a logic function into a combinatorial circuit are standard form reductions, map creations and adequate use of don't care conditions (input/output conditions which are considered to be invalid/ will never happen). *Sequential circuits* are composed of two parts:

1. A memory structure usually built out of *flip flops*

2. One ore more combinatorial circuits, implementing some logical functions

1.1.2 Logic Gates

Logic gates are electronic devices usually built out of transistors which perform certain logic functions. A *logic function* is a function which maps binary inputs to binary outputs.

1.1.3 The Transistor

A transistor is an electronic device which allows the control of the flow of one current source through a second, potentially smaller current source. This serves as the basic component for creating more advanced electronic components like logic gates. Today, the most popular type of transistor and the most widley produced device in the world is the MOSFET [1].

1.1.4 Buffers

Buffers are simple logic gates which just pass through the signal they receive.

A	**B**
0	0
1	1

Table 1.1: Truth table of a buffer

1.1.5 Inverters

Inverters are similar in complexity to buffers. They invert the signal they receive.

A	**B**
0	1
1	0

Table 1.2: Truth table of an inverter

1.1.6 AND Gates

And gates output a logic 1 only if all of their inputs are logic 1's.

A	**B**	**O**
0	0	0
1	0	0
0	1	0
1	1	1

Table 1.3: AND Gate Truth Table

1.1.7 OR Gates

OR gates output a logic 1 if either of their inputs or both of them are logic 1's.

A	B	O
0	0	0
1	0	1
0	1	1
1	1	1

Table 1.4: OR Gate Truth Table

1.1.8 XOR Gates

Exclusive OR, or XOR gates output a logic 1 only if either of their inputs is a 1, but not both.

A	B	O
0	0	0
1	0	1
0	1	1
1	1	0

Table 1.5: XOR Gate Truth Table

1.1.9 NAND Gates

A NAND gate is an inverted AND gate. This means it outputs a logic 1 in all situations except for when all of its inputs are logic 1's.

A	B	O
0	0	1
1	0	1
0	1	1
1	1	0

Table 1.6: NAND Gate Truth Table

1.1.10 NOR Gates

A NOR Gate is an inverted OR gate. This means it outputs a logic 1 only if all of its inputs are logic 0's. NAND and NOR gates are considered *universal gates*. This means that any logic circuit can be built exclusivley out of NAND or out of NOR gates.

A	B	O
0	0	1
1	0	0
0	1	0
1	1	0

Table 1.7: NOR Gate Truth Table

1.1.11 Flip Flops

Flip Flops are circuits usually built out of logic gates which can store one bit of information, i.e. they can be either on or off. There are many types of flip flops: RS-flip-flops (Reset-Set), D-flip-flops (Data), JK-flip flops (refined RS flip flops) and T flops (single input JK flip-flops). Each flip flop performs best in a certain scenario. Since they all implement de storage and retrieval of one bit of information, only the D-flip-flop (the most commonly used one) will be discussed.

Chapter 2

Images

This chapter contains images off all modules implemented as a part of the 16-breadboarc computer.

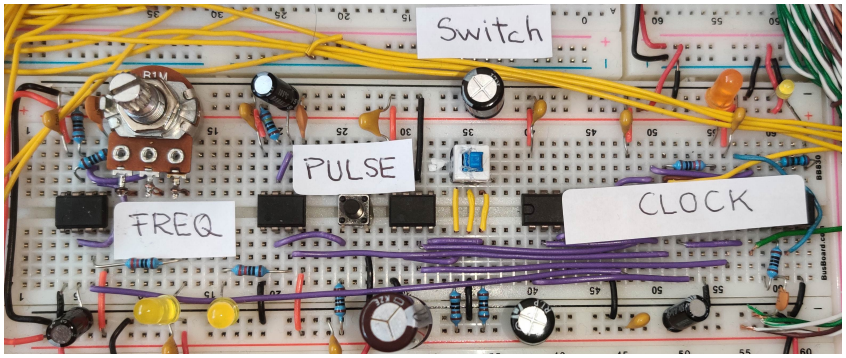

Figure 2.1: Clock implementation

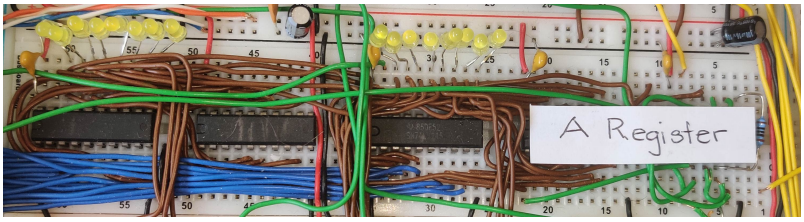

Figure 2.2: A register implementation

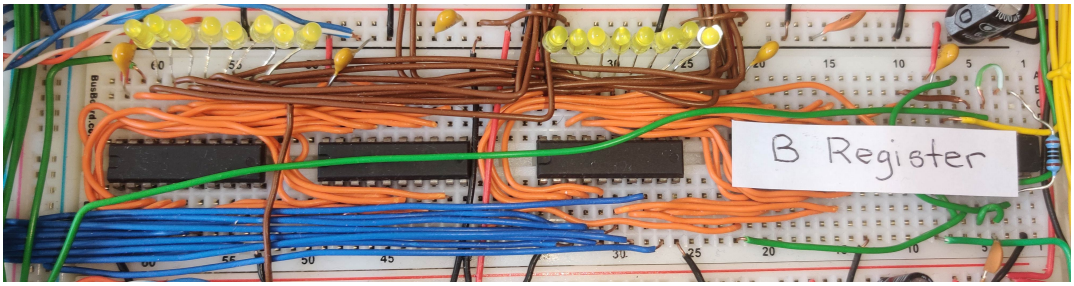

Figure 2.3: B register implementation

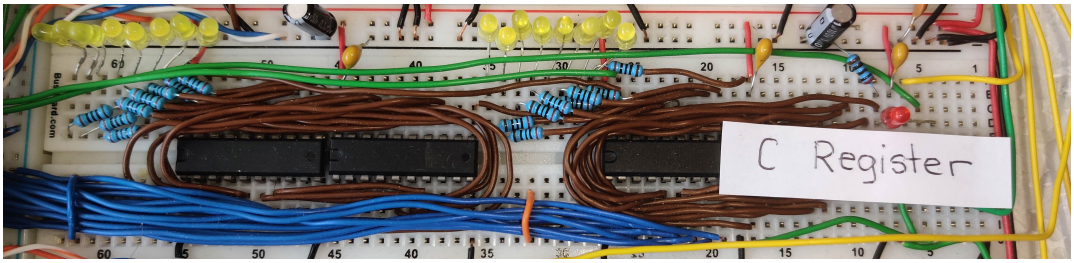

Figure 2.4: C register implementation

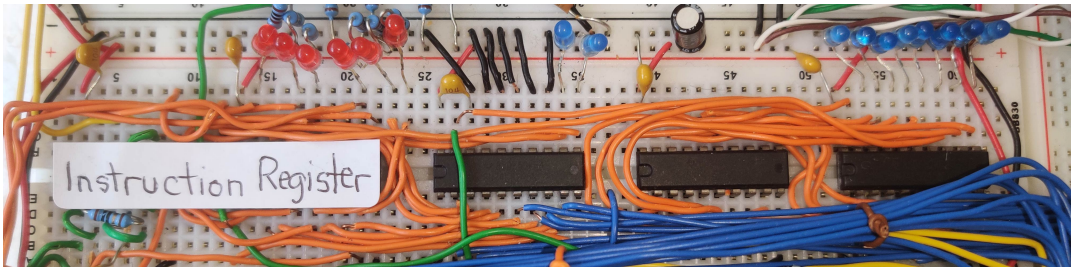

Figure 2.5: Instruction register implementation

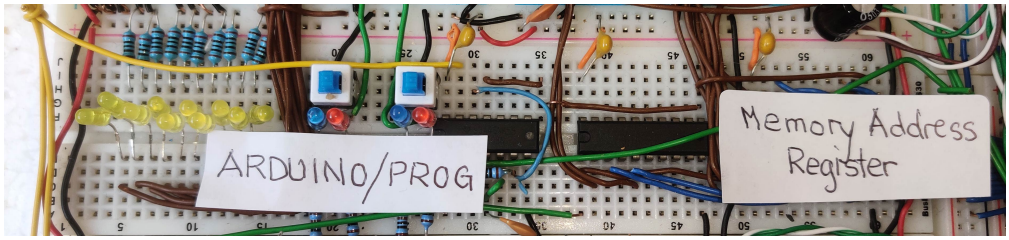

Figure 2.6: Memory address register implementation

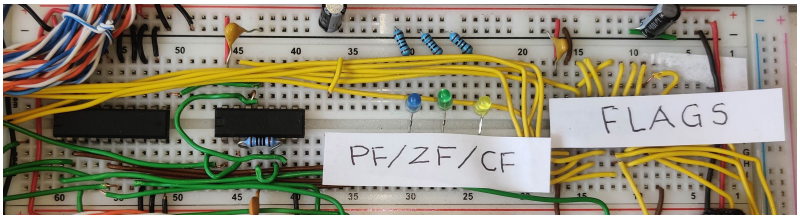

Figure 2.7: Flags register implementation

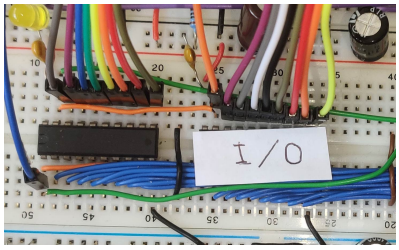

Figure 2.8: Input/Output implementation

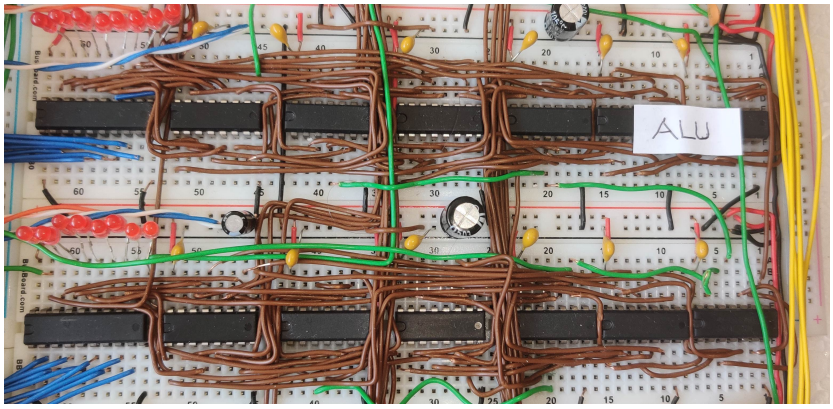

Figure 2.9: Arithmetic-Logic Unit implementation

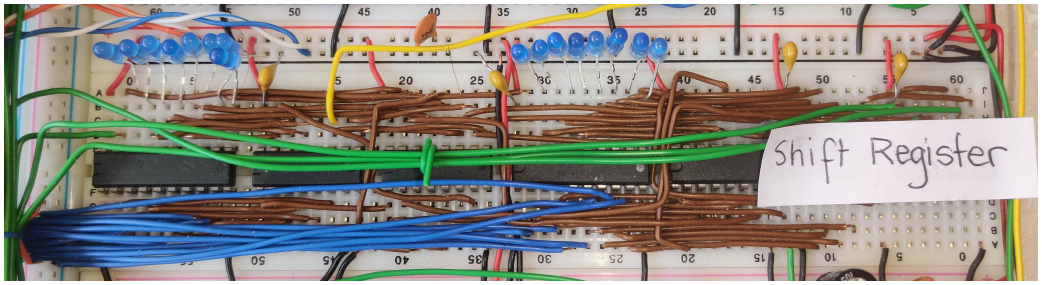

Figure 2.10: Shift Register implementation

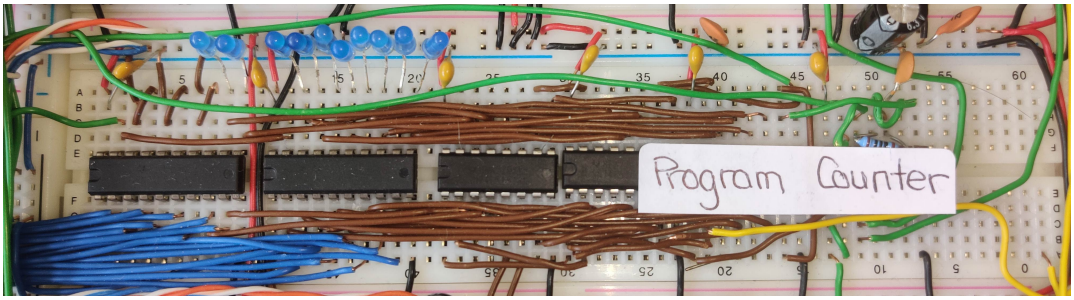

Figure 2.11: Program Counter implementation

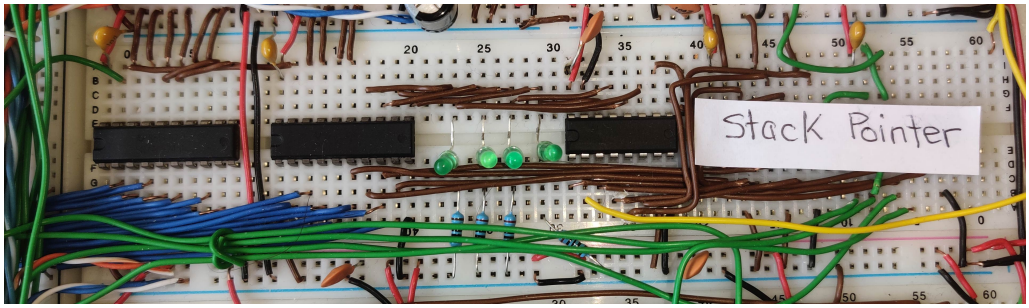

Figure 2.12: Stack Pointer implementation

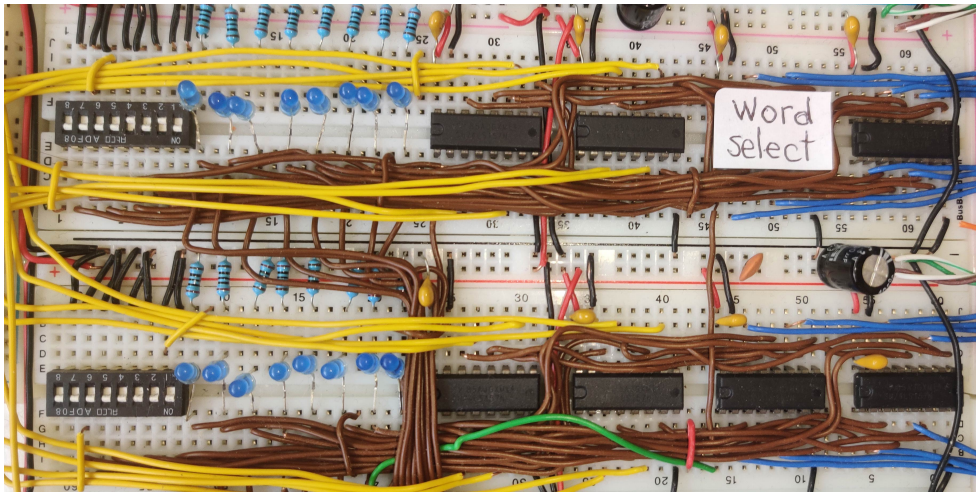

Figure 2.13: Word selector implementation

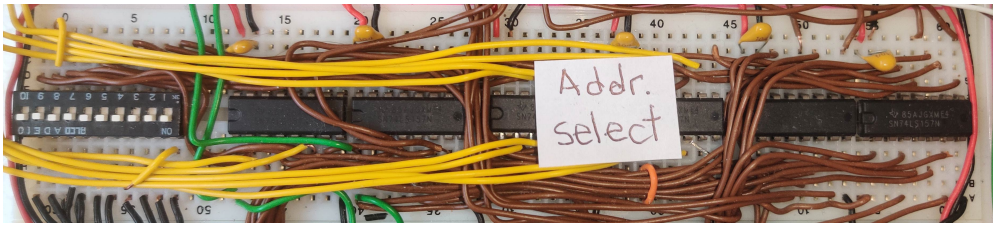

Figure 2.14: Address selector implementation

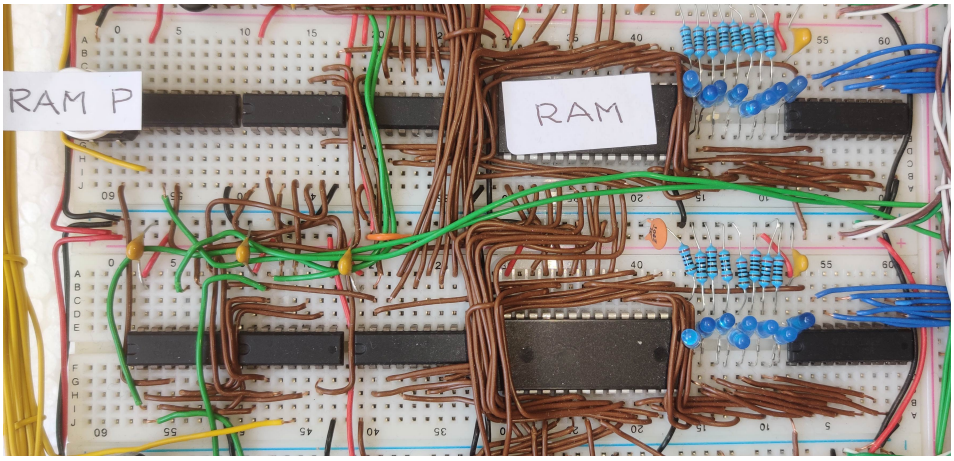

Figure 2.15: Random Access Memory implementation

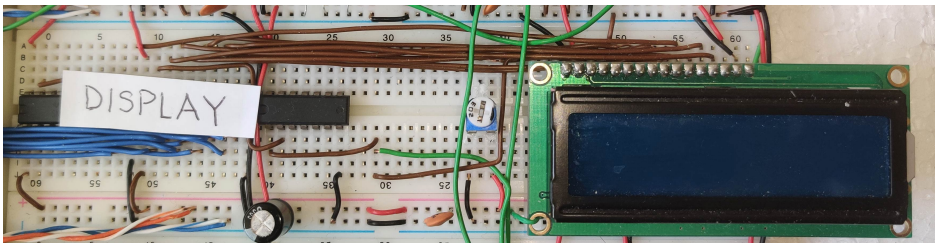

Figure 2.16: Display implementation

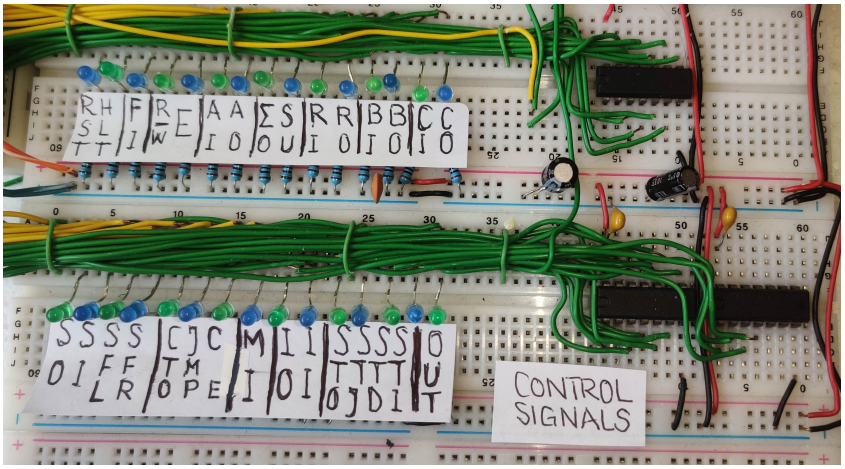

Figure 2.17: Implementation of the contol signals module

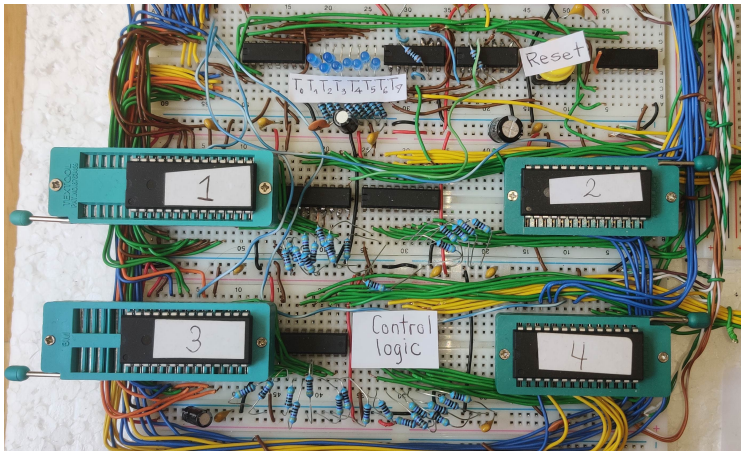

Figure 2.18: Control Logic implementation

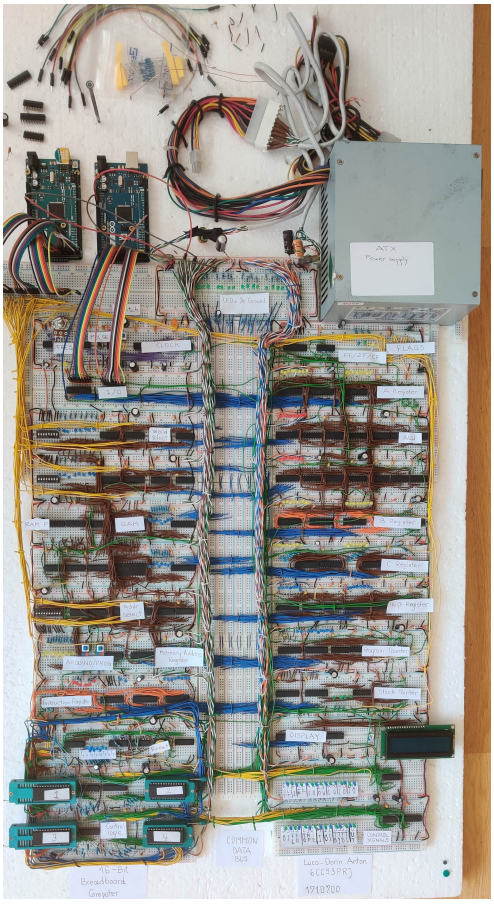

Figure 2.19: High level overview (with bus and LEDs visible)

Chapter 3

Source Code

This chapter contains the list of all source code created during this project.

3.1 Originality Pledge

"I verify that I am the sole author of the programs contained in this folder, except where explicitly stated to the contrary"
Luca-Dorin Anton - 23 April 2020

3.2 MicroCode generator

3.2.1 classes.py

```
from enum import Enum


class ControlSig():

    def __init__(self,eeprom, byte, name, desc):
        self.eeprom = eeprom
        self.byte = byte
        self.name = name
        self.desc = desc
```

```
class CtrlSigs(Enum):

    # Control Signals for EEPROM 1
    CO   = ControlSig(1, 0b00000001, 'CO',"C Register Out")
    CI   = ControlSig(1, 0b00000010, 'CI', "C Register In")
    BO   = ControlSig(1, 0b00000100, 'BO', "B Register Out")
    BI   = ControlSig(1, 0b00001000, 'BI', "B Register In")
    RO   = ControlSig(1, 0b00010000, 'RO', "RAM Out")
    RI   = ControlSig(1, 0b00100000, 'RI', "RAM In")
    SU   = ControlSig(1, 0b01000000, 'SU', "Subtract")
    DONE= ControlSig(1, 0b10000000, 'DONE', "Instruction Finished")

    # Control Signals for EEPROM 2
    EO = ControlSig(2, 0b00000001, 'EO', "Sum Out")
    AO = ControlSig(2, 0b00000010, 'AO', "A Register Out")
    AI = ControlSig(2, 0b00000100, 'AI', "A Register In")
    E  = ControlSig(2, 0b00001000, 'E', "Enable I/O")
    RW = ControlSig(2, 0b00010000, 'RW', "I/O Read/Write Flag")
    FI = ControlSig(2, 0b00100000, 'FI', "Flags In")
    HLT= ControlSig(2, 0b01000000, 'HLT', "Halt Execution")
    RST= ControlSig(2, 0b10000000, 'RST', "Reset Computer")

    # Control Signals for EEPROM 3
    X   = ControlSig(3, 0b00000001, 'X', "No Op")
    OUT= ControlSig(3, 0b00000010, 'OUT', "Display In")
    STI= ControlSig(3, 0b00000100, 'STI', "Stack Pointer Increment")
    STD= ControlSig(3, 0b00001000, 'STD', "Stack Pointer Decrement")
    STJ= ControlSig(3, 0b00010000, 'STJ', "Stack Jump")
    STO= ControlSig(3, 0b00100000, 'STO', "Stack Out")
    II  = ControlSig(3, 0b01000000, 'II', "Instruction Register In")
    IO = ControlSig(3, 0b10000000, 'IO', "Instruction Register Out")

    # Control Signals for EEPROM 4
```

```
    MI = ControlSig(4, 0b00000001, 'MI', "Memory Address Register In")
    CE = ControlSig(4, 0b00000010, 'CE', "Program Counter Enable")
    JMP= ControlSig(4, 0b00000100, 'JMP', "Program Counter In")
    CTO= ControlSig(4, 0b00001000, 'CTO', "Program Counter Out")
    SFR= ControlSig(4, 0b00010000, 'SFR', "Shift Right")
    SFL= ControlSig(4, 0b00100000, 'SFL', "Shift Left")
    SI = ControlSig(4, 0b01000000, 'SI', "Shift Register In")
    SO = ControlSig(4, 0b10000000, 'SO', "Shift Register Out")

# A microstep is one time step of the clock
class MicroStep():

    def __init__(self, sigs, desc=""):
        self.sigs = sigs
        self.desc = desc

    def end_instruction(self):
        self.sigs.append(CtrlSigs.DONE)

    def split_sigs(self):
        eeprom_bytes = [0b00000000] * 4
        for sig in self.sigs:
            eeprom_bytes[sig.value.eeprom - 1] = eeprom_bytes[sig.value.eeprom - 1] |
        return eeprom_bytes



    def __str__(self):
        if len(self.sigs) == 0:
            return "; " + self.desc
        s = self.sigs[0].name
        for sig in self.sigs[1:]:
            s += ' | ' + sig.name
        return s + "; %s" % self.desc
```

```
# All instructions start like this
def create_fetch_cycle():
    t_0 = MicroStep([CtrlSigs.CTO, CtrlSigs.MI], "Move PC Contents to MAR")
    t_1 = MicroStep([CtrlSigs.RO, CtrlSigs.II, CtrlSigs.CE], "Move RAM Contents to IR
    return [t_0, t_1]

# Instruction have flags, flags have lists of microsteps
class Instruction():

    def __init__(self, name, opcode=0b000000, desc='', op=False):
        self.flag_agnostic_steps = create_fetch_cycle()
        self.steps = {}
        self.name = name
        self.opcode = opcode
        self.desc = desc
        self.op = op

    def insert_step(self, step, flag='other'):
        if flag not in ['other', 'CF', 'ZF', 'PF']:
            raise Exception("Incorrect Flag")
        if flag == "other":
            self.flag_agnostic_steps.append(step)
            return
        if flag not in self.steps.keys():
            self.steps[flag] = create_fetch_cycle()
        self.steps[flag].append(step)


    def __str__(self):
        s = "Name: %s Opcode: %s Desc: %s Needs operand: %s\n" % (self.name, format(s
        s += "  Flag agnostic:\n"
        for i in range(len(self.flag_agnostic_steps)):
```

```
            s += "      Step %s: %s \n" % (format(i, '03b'), str(self.flag_agnostic_st
        for key in sorted(self.steps.keys()):
            s += "  Flag: %s\n" % key
            for i in range(len(self.steps[key])):
                s += "      Step %s: %s \n" % (format(i, '03b'), str(self.steps[key][i
        return s
```

3.2.2 microcode.py

```
from yamlparser import parse_instructions
from importlib import import_module
import sys, os, time



def flag_to_pattern(flag):
    if flag not in ['CF', 'ZF', 'PF']:
        raise Exception("Incorrect Flag")
    if flag == 'CF':
        pattern = 0b100
    elif flag == 'ZF':
        pattern = 0b010
    else:
        pattern = 0b100
    return pattern


# This is where the magic happens
def generate_microcode(instr_lst):
    eeproms = []
    for i in range(4):
        eeproms.append(bytearray([0b00] * 8192))
    for instr in instr_lst:
        for flag_pattern in range(8):
            steps = None
            found = False
```

```
            for flag in instr.steps.keys():
                if found:
                    break
                if flag_pattern & flag_to_pattern(flag):
                    steps = instr.steps[flag]
                    found = True
            if not found:
                steps = instr.flag_agnostic_steps
            if len(steps) < 8:
                steps[-1].end_instruction()
            for step in range(8):
                if step < len(steps):
                    # Cast three binary values together into an int
                    addr = int(format(instr.opcode, '06b') + format(flag_pattern, '03b
                    byte_lst = steps[step].split_sigs()
                    for i in range(4):
                        eeproms[i][addr] = byte_lst[i]
                else:
                    break
    return eeproms

def write_files(instr_lst):
    eeproms = generate_microcode(instr_lst)
    print("Binary translation finished")

    with open('eeprom1.bin', 'wb') as f:
        f.write(eeproms[0])
    print("eeprom1.bin saved")

    with open('eeprom2.bin', 'wb') as f:
        f.write(eeproms[1])
    print("eeprom2.bin saved")
```

```
    with open('eeprom3.bin', 'wb') as f:
        f.write(eeproms[2])
    print("eeprom3.bin saved")

    with open('eeprom4.bin', 'wb') as f:
        f.write(eeproms[3])
    print("eeprom4.bin saved")
    print("Begin generating Assembler Instruction List")
    instr_names = []
    for i in instr_lst:
        instr_names.append(i.name)
    with open('instruction_list.txt', 'w') as f:
        for i in instr_names:
            f.write(i + '\n')
    print("Assembler Instruction List generation finished")

def print_instr_lst(instr_lst, to_print):
    if len(to_print) == 0:
        print("Microcode instruction descriptions")
        for i in instr_lst:
            print(i)
        print("Printed %i instructions." % len(instr_lst))
    else:
        printed = 0
        for name in to_print:
            found = False
            for instr in instr_lst:
                if instr.name == name:
                    print(instr)
                    found = True
                    break
            if not found:
                print("Name '%s' doesn't match any instruction. Skipping..." % name)
```

```
            else:
                printed += 1
        print("Successfully printed %i instructions" % printed)



def help():
    s = "Microcode generation tool for the 16-bit breadboard computer\n"
    s += "                                    built by Luca-Dorin Anton\n"
    s += "Usage: python3 microcode.py [py/yaml] [filename] [gen/print] [.. instr_name]\
    s += "Argument description:\n"
    s += "  py/yaml: specifies if the supplied instruction file is in python or YAML f
    s += "  filename: the file from which to read instructions\n"
    s += "  gen/print: whether to generate eeprom binaries (gen) or to print out instr
    s += "  .. instr_name: if print is used, specify which instructions to print. If or
    s += "\nThis script is part of the 6CCSPRJ major project module and contributes to
    print(s)

if __name__ == "__main__":
    if len(sys.argv) < 4:
        help()
        sys.exit(1)
    if sys.argv[1].lower() not in ['py', 'yaml']:
        help()
        sys.exit(1)
    if sys.argv[3].lower() not in ['gen', 'print']:
        help()
        sys.exit(1)
    try:
        f = open(sys.argv[2], 'r')
        f.close()
    except:
        print("Error: Couldn't read file '%s', terminating." % sys.argv[2])
```

```
        help()
        sys.exit(1)
    mode = sys.argv[1].lower()
    file = sys.argv[2]
    task = sys.argv[3].lower()
    to_print = []
    if len(sys.argv) > 4:
        to_print = sys.argv[4:]
    if mode.lower() == 'py':
        name = os.path.splitext(file)[0]
        module = import_module(name)
        instr_lst = module.instr_lst
    else:
        instr_lst = parse_instructions(file)
    if task == 'gen':
        print("Starting EEPROM binary generation")
        start = time.time()
        write_files(instr_lst)
        end = time.time()
        print("Finished generating EEPROM binaries")
        print("Exec time: %s s" % str(round(end - start, 2)))
        sys.exit(0)
    else:
        print_instr_lst(instr_lst, to_print)
```

3.2.3 microinstructions.py

```
from classes import Instruction, CtrlSigs, MicroStep


instr_lst = []


# No Operation
NOP = Instruction('NOP', 0b000000, "Do absolutely nothing")
NOP.insert_step(MicroStep([], "Do absolutely nothing"))
```

```
instr_lst.append(NOP)

# A Load instructions
LDA = Instruction('LDA', 0b000001, "Load RAM contents to Register A", True)
LDA.insert_step(MicroStep([CtrlSigs.IO, CtrlSigs.MI], "Move IR Address to MAR"))
LDA.insert_step(MicroStep([CtrlSigs.RO, CtrlSigs.AI], "Move RAM contents to reg. A"))
instr_lst.append(LDA)

LAI = Instruction('LAI', 0b000010, "Load a value to Register A immediately", True)
LAI.insert_step(MicroStep([CtrlSigs.IO, CtrlSigs.AI], "Move IR value to reg. A"))
instr_lst.append(LAI)

# B Load Instructions
LDB = Instruction('LDB', 0b000011, "Load RAM contents to Register B", True)
LDB.insert_step(MicroStep([CtrlSigs.IO, CtrlSigs.MI], "Move IR contetns to MAR"))
LDB.insert_step(MicroStep([CtrlSigs.RO, CtrlSigs.BI], "Move RAM contents to reg. B"))
instr_lst.append(LDB)

LBI = Instruction('LBI', 0b000100, "Load a value to Register B immediately, True)
LBI.insert_step(MicroStep([CtrlSigs.IO, CtrlSigs.BI], "Move IR value to reg. B"))
instr_lst.append(LBI)

# C Load Instructions
LDC = Instruction('LDC', 0b000101, "Load RAM contents to Register C", True)
LDC.insert_step(MicroStep([CtrlSigs.IO, CtrlSigs.MI], "Move IR Address to MAR"))
LDC.insert_step(MicroStep([CtrlSigs.RO, CtrlSigs.CI], "Move RAM contents to reg. C"))
instr_lst.append(LDC)

LCI = Instruction('LCI', 0b000110, "Load a value to register C immediately", True)
LCI.insert_step(MicroStep([CtrlSigs.IO, CtrlSigs.CI], "Move IR value to reg. C"))
instr_lst.append(LCI)
```

```
# Shift Load Instruction
LDS = Instruction('LDS', 0b000111, "Load RAM contents to the Shift Register", True)
LDS.insert_step(MicroStep([CtrlSigs.IO, CtrlSigs.MI], "Move IR Address to MAR"))
LDS.insert_step(MicroStep([CtrlSigs.RO, CtrlSigs.SI], "Move RAM contents to the Shift
instr_lst.append(LDS)

LSI = Instruction('LSI', 0b001000, "Load a value to the shift register immediately", T
LSI.insert_step(MicroStep([CtrlSigs.IO, CtrlSigs.SI], "Move IR value to the shift regi
instr_lst.append(LSI)



# Data processing Instructions
ADD = Instruction("ADD", 0b001001, "Add up reg. A contents to reg. B contents and stor
ADD.insert_step(MicroStep([CtrlSigs.EO, CtrlSigs.AI, CtrlSigs.FI], "Move ALU contents
instr_lst.append(ADD)

SUB = Instruction("SUB", 0b001010, "Subtract reg. B contents from reg. A contents and
SUB.insert_step(MicroStep([CtrlSigs.EO, CtrlSigs.AI, CtrlSigs.SU, CtrlSigs.FI], "Set S
instr_lst.append(SUB)

SFL = Instruction("SFL", 0b001011, "Perform a left shift on the contents of the shift
SFL.insert_step(MicroStep([CtrlSigs.SFL], "Perform a left shift on the contents of the
instr_lst.append(SFL)

SFR = Instruction("SFR", 0b001100, "Perform a right shift on the contents of the shift
SFR.insert_step(MicroStep([CtrlSigs.SFR], "Perform a right shift on the contents of th
instr_lst.append(SFR)

# Jump and Branch Instructions
JMP = Instruction('JMP', 0b001101, "Jump to Specified Address", True)
JMP.insert_step(MicroStep([CtrlSigs.JMP, CtrlSigs.IO], "Move IR contents to the PC"))
instr_lst.append(JMP)
```

```
JPI = Instruction('JPI', 0b001110, "Jump indirectly to an address stored at the RAM ad
JPI.insert_step(MicroStep([CtrlSigs.IO, CtrlSigs.MI], "Move IR contents to MAR"))
JPI.insert_step(MicroStep([CtrlSigs.RO, CtrlSigs.JMP], "Move RAM contents to the progr
instr_lst.append(JPI)

BRZ = Instruction('BRZ', 0b001111, "Branch to an address if the Zero Flag is set", Tru
BRZ.insert_step(MicroStep([CtrlSigs.IO, CtrlSigs.JMP], "Move IR contents to the progra
instr_lst.append(BRZ)

BRP = Instruction('BRP', 0b010000, "Branch to an address if the Parity Flag is set", T
BRP.insert_step(MicroStep([CtrlSigs.IO, CtrlSigs.JMP], "Move IR contents to the progra
instr_lst.append(BRP)

BRC = Instruction('BRC', 0b010001, "Branch to an address if the Carry Flag is set", Tr
BRC.insert_step(MicroStep([CtrlSigs.IO, CtrlSigs.JMP], "Move IR contents to the progra
instr_lst.append(BRC)

# Move from A instructions
MAB = Instruction('MAB', 0b010010, "Move reg. A contents to the B register")
MAB.insert_step(MicroStep([CtrlSigs.AO, CtrlSigs.BI], "Move reg. A contents to the B r
instr_lst.append(MAB)

MAC = Instruction('MAC', 0b010011, "Move reg. A contents to the C register")
MAC.insert_step(MicroStep([CtrlSigs.AO, CtrlSigs.CI], "Move reg. A contents to the C r
instr_lst.append(MAC)

MAS = Instruction('MAS', 0b010100, "Move reg. A contents to the shift register")
MAS.insert_step(MicroStep([CtrlSigs.AO, CtrlSigs.SI], "Move reg. A contents to the sh
instr_lst.append(MAS)

# Move from B instructions
MBA = Instruction('MBA', 0b010101, "Move reg. B contents to the A register")
MBA.insert_step(MicroStep([CtrlSigs.BO, CtrlSigs.AI], "Move reg. B contents to the A r
```

```
instr_lst.append(MBA)

MBC = Instruction('MBC', 0b010110, "Move reg. B contents to the C register")
MBC.insert_step(MicroStep([CtrlSigs.BO, CtrlSigs.CI], "Move reg. B contents to the C r
instr_lst.append(MBC)

MBS = Instruction('MBS', 0b010111, "Move reg. B contents to the shift register")
MBS.insert_step(MicroStep([CtrlSigs.BO, CtrlSigs.SI], "Move reg. B contents to the sh
instr_lst.append(MBS)

# Move from C instructions
MCA = Instruction('MCA', 0b011000, "Move reg. C contents to the A register")
MCA.insert_step(MicroStep([CtrlSigs.CO, CtrlSigs.AI], "Move reg. C contents to the A r
instr_lst.append(MCA)

MCB = Instruction('MCB', 0b011001, "Move reg. C contents to the B register")
MCB.insert_step(MicroStep([CtrlSigs.CO, CtrlSigs.BI], "Move reg. C contents to the B r
instr_lst.append(MCB)

MCS = Instruction('MCS', 0b011010, "Move reg. C contents to the shift register")
MCS.insert_step(MicroStep([CtrlSigs.CO, CtrlSigs.SI], "Move reg. C contents to the sh
instr_lst.append(MCS)

# Move from shift register instructions
MSA = Instruction('MSA', 0b011011, "Move shift register contents to the A register")
MSA.insert_step(MicroStep([CtrlSigs.SO, CtrlSigs.AI], "Move shift register contents to
instr_lst.append(MSA)

MSB = Instruction('MSB', 0b011100, "Move shift register contents to the B register")
MSB.insert_step(MicroStep([CtrlSigs.SO, CtrlSigs.BI], "Move shift register contents to
instr_lst.append(MSB)

MSC = Instruction('MSC', 0b011101, "Move shift register contents to the C register")
```

```
MSC.insert_step(MicroStep([CtrlSigs.SO, CtrlSigs.CI], "Move shift register contents to
instr_lst.append(MSC)

# RAM write instructions
STA = Instruction('STA', 0b011110, "Write the contents of the A register to RAM addres
STA.insert_step(MicroStep([CtrlSigs.IO, CtrlSigs.MI], "Move IR contents to the MAR"))
STA.insert_step(MicroStep([CtrlSigs.AO, CtrlSigs.RI], "Move reg. A contents to RAM"))
instr_lst.append(STA)

STB = Instruction('STB', 0b011111, "Write the contents of the B register to RAM addres
STB.insert_step(MicroStep([CtrlSigs.IO, CtrlSigs.MI], "Move IR contents to the MAR"))
STB.insert_step(MicroStep([CtrlSigs.BO, CtrlSigs.RI], "Move reg. B contents to RAM"))
instr_lst.append(STB)

STC = Instruction('STC', 0b100000, "Write the contents of the C register to RAM addres
STC.insert_step(MicroStep([CtrlSigs.IO, CtrlSigs.MI], "Move IR contents to the MAR"))
STC.insert_step(MicroStep([CtrlSigs.CO, CtrlSigs.RI], "Move reg. C contents to RAM"))
instr_lst.append(STC)

STS = Instruction('STS', 0b100001, "Write the contents of the shift register to RAM ad
STS.insert_step(MicroStep([CtrlSigs.IO, CtrlSigs.MI], "Move IR contents to the MAR"))
STS.insert_step(MicroStep([CtrlSigs.SO, CtrlSigs.RI], "Move reg. shift contents to RAM
instr_lst.append(STS)

# Stack Manipulation
JSR = Instruction('JSR', 0b100010, "Jump to subroutine", True)
JSR.insert_step(MicroStep([CtrlSigs.STO, CtrlSigs.MI], "Move stack pointer to MAR"))
JSR.insert_step(MicroStep([CtrlSigs.CTO, CtrlSigs.RI], "Store program counter in RAM s
JSR.insert_step(MicroStep([CtrlSigs.STI, CtrlSigs.IO, CtrlSigs.JMP], "Increment Stack
instr_lst.append(JSR)

RET = Instruction('RET', 0b100011, "Return from subroutine")
RET.insert_step(MicroStep([CtrlSigs.STD], "Decrement Stack pointer"))
```

```
RET.insert_step(MicroStep([CtrlSigs.STO, CtrlSigs.MI], "Move stack pointer to MAR"))
RET.insert_step(MicroStep([CtrlSigs.RO, CtrlSigs.JMP], "Move RAM contents to the progr
instr_lst.append(RET)

# I/O Read Instructions
IRA = Instruction('IRA', 0b100100, "Read from I/O to the A register")
IRA.insert_step(MicroStep([CtrlSigs.E, CtrlSigs.RW, CtrlSigs.AI], "Enable IO, set in r
instr_lst.append(IRA)

IRB = Instruction('IRB', 0b100101, "Read from I/O to the B register")
IRB.insert_step(MicroStep([CtrlSigs.E, CtrlSigs.RW, CtrlSigs.BI], "Enable I/O, set in
instr_lst.append(IRB)

IRC = Instruction('IRC', 0b100110, "Read from I/O to the C register")
IRC.insert_step(MicroStep([CtrlSigs.E, CtrlSigs.RW, CtrlSigs.CI], "Enable I/O, set in
instr_lst.append(IRC)

IRS = Instruction('IRS', 0b100111, "Read from I/O to the shift register")
IRS.insert_step(MicroStep([CtrlSigs.E, CtrlSigs.RW, CtrlSigs.SI], "Enable I/O, set in
instr_lst.append(IRS)

IRR = Instruction('IRR', 0b101000, "Read from I/O to the RAM address specified in the
IRR.insert_step(MicroStep([CtrlSigs.IO, CtrlSigs.MI], "Move IR contents to the MAR"))
IRR.insert_step(MicroStep([CtrlSigs.E, CtrlSigs.RW, CtrlSigs.RI], "Enable I/O, set in
instr_lst.append(IRR)

# I/O Write Instructions
IWA = Instruction('IWA', 0b101001, "Write A register contents to I/O")
IWA.insert_step(MicroStep([CtrlSigs.E, CtrlSigs.AO], "Enable I/O, set in write mode. N
instr_lst.append(IWA)

IWB = Instruction('IWB', 0b101010, "Write B register contents to I/O")
IWB.insert_step(MicroStep([CtrlSigs.E, CtrlSigs.BO], "Enable I/O, set in write mode. N
```

```
instr_lst.append(IWB)

IWC = Instruction('IWC', 0b101011, "Write C register contents to I/O")
IWC.insert_step(MicroStep([CtrlSigs.E, CtrlSigs.CO], "Enable I/O, set in write mode. M
instr_lst.append(IWC)

IWS = Instruction('IWS', 0b101100, "Write shift register contents to I/O")
IWS.insert_step(MicroStep([CtrlSigs.E, CtrlSigs.SO], "Enable I/O, set in write mode. M
instr_lst.append(IWS)

IWR = Instruction('IWR', 0b101101, "Write RAM contents at address specified by instruc
IWR.insert_step(MicroStep([CtrlSigs.IO, CtrlSigs.MI], "Move IR contents to MAR"))
IWR.insert_step(MicroStep([CtrlSigs.E, CtrlSigs.RO], "Enable I/O, set in write mode. M
instr_lst.append(IWR)

# Display instructions
DWA = Instruction('DWA', 0b101110, "Write register A contents to the Display")
DWA.insert_step(MicroStep([CtrlSigs.AO, CtrlSigs.OUT], "Move contents of reg. A to Dis
instr_lst.append(DWA)

DWB = Instruction('DWB', 0b101111, "Write register B contents to the Display")
DWB.insert_step(MicroStep([CtrlSigs.BO, CtrlSigs.OUT], "Move contents of reg. A to Dis
instr_lst.append(DWB)

DWC = Instruction('DWC', 0b110000, "Write register C contents to the Display")
DWC.insert_step(MicroStep([CtrlSigs.CO, CtrlSigs.OUT], "Move contents of reg. C to Dis
instr_lst.append(DWC)

DWS = Instruction('DWS', 0b110001, "Write shift register contents to the Display")
DWS.insert_step(MicroStep([CtrlSigs.SO, CtrlSigs.OUT], "Move contents of shift registe
instr_lst.append(DWS)

DWR = Instruction('DWR', 0b110010, "Write RAM contents at address specified by instruc
```

```
DWR.insert_step(MicroStep([CtrlSigs.IO, CtrlSigs.MI], "Move IR contents to the MAR"))
DWR.insert_step(MicroStep([CtrlSigs.RO, CtrlSigs.OUT], "Move RAM contents to Display")
instr_lst.append(DWR)

# Computer and clock manipulation
RST = Instruction('RST', 0b111110, "Reset the computer")
RST.insert_step(MicroStep([CtrlSigs.RST], "Reset the computer"))
instr_lst.append(RST)

HLT = Instruction('HLT', 0b111111, "Stop the clock")
HLT.insert_step(MicroStep([CtrlSigs.HLT], "Stop the clock"))
instr_lst.append(HLT)
```

3.2.4 yamlparser.py

```
import yaml
from yaml import Loader
from classes import Instruction, CtrlSigs, MicroStep


# Parse a YAML object into a list of Instruction objects
def parse_instructions(f):
    instr_lst = []

    yaml_obj = yaml.load(open(f, 'r').read(), Loader=Loader)
    for instr in yaml_obj['instructions']:
        name = instr['name']
        opcode = instr['opcode']
        if opcode > 63:
            raise Exception("Invalid Opcode: %i" % opcode)
        instr_desc = instr['description']
        operand = bool(instr['requires_operand'])
        instruction = Instruction(name, opcode, instr_desc, operand)
        for key in instr['flags'].keys():
```

```
            if key.lower() not in ['other', 'cf', 'zf', 'pf']:
                raise Exception("Invalid Flag: %s" % key)
            for step in instr['flags'][key]:
                sigs = step['signals']
                if sigs is None:
                    sigs = []
                else:
                    new_sigs = []
                    for sig in sigs:
                        new_sigs.append(CtrlSigs[sig])
                    sigs = new_sigs
                step_desc = step['description']
                microstep = MicroStep(sigs, step_desc)
                instruction.insert_step(microstep, key)
        instr_lst.append(instruction)
    return instr_lst
```

3.2.5 microinstructions.yaml

```
# template:
# name:
# opcode:
# description:
# flags:
# requires_operand:
#   other:
#     -
#       signals: []
#       description:


instructions:
# No Operation
```

```
  -
    name: NOP
    opcode: 0x00
    description: Do absolutely nothing
    requires_operand: False
    flags:
        other:
          -
            signals: []
            description: Do absolutley nothing
# A Load Instructions
  -
    name: LDA
    opcode: 0x01
    description: Load RAM contents to Register A
    requires_operand: True
    flags:
      other:
        -
          signals: [IO, MI]
          description: Move IR Address to MAR
        -
          signals: [RO, AI]
          description: Move RAM contents to reg. A
  -
    name: LAI
    opcode: 0x02
    description: Load a value to Register A immediately
    requires_operand: True
    flags:
      other:
        -
          signals: [IO, AI]
```

```
        description: Move IR value to reg. A
# B Load Instructions
  -
    name: LDB
    opcode: 0x03
    description: Load RAM contents to Register B
    requires_operand: True
    flags:
      other:
        -
          signals: [IO, MI]
          description: Move IR contents to the MAR
        -
          signals: [RO, BI]
          description: Move RAM contents to reg. B
  -
    name: LBI
    opcode: 0x04
    description: Load a alue to Register B immediately
    requires_operand: True
    flags:
      other:
        -
          signals: [IO, BI]
          description: Move IR value to reg. B
# C Load Instructions
  -
    name: LDC
    opcode: 0x05
    description: Load RAM contents to Register C
    requires_operand: True
    flags:
      other:
```

```
      -
        signals: [IO, MI]
        description: Move IR contents to the MAR
      -
        signals: [RO, CI]
        description: Move RAM contents to reg. C
  -
    name: LCI
    opcode: 0x06
    description: Load a value to register C immediately
    requires_operand: True
    flags:
      other:
        -
          signals: [IO, CI]
          description: Move IR value to reg. C
# Shift Register Load Instructions
  -
    name: LDS
    opcode: 0x07
    description: Load a value to the shift register
    requires_operand: True
    flags:
      other:
        -
          signals: [IO, MI]
          description: Move IR contents to the MAR
        -
          signals: [RO, SI]
          description: Move RAM contents to the shift register
  -
    name: LSI
    opcode: 0x08
```

```
    description: Load a value to the shift register immediately
    requires_operand: True
    flags:
      other:
        -
          signals: [IO, SI]
          description: Move IR contents to the shift register
# Data Processing Instructions
  -
    name: ADD
    opcode: 0x09
    description: Add up reg. A contents to reg. B contents and store result in reg. A
    requires_operand: False
    flags:
      other:
        -
          signals: [EO, AI, FI]
          description: Move ALU contents to A register; Update flags
  -
    name: SUB
    opcode: 0x0A
    description: Subtract reg. B contents from reg. A contents and store result in reg
    requires_operand: False
    flags:
      other:
        -
          signals: [EO, AI, SU, FI]
          description: Set Subtract Flag; Move ALU contents to reg. A; Update flags
  -
    name: SFL
    opcode: 0x0B
    description: Perform a left shift on the contents of the shift register
    requires_operand: False
```

```
    flags:
      other:
        -
          signals: [SFL]
          description: Performa left shift on the contents of the shift register
  -
    name: SFR
    opcode: 0x0C
    description: Perform a right shift on the contents of the shift register
    requires_operand: False
    flags:
      other:
        -
          signals: [SFR]
          description: Perform a right shift on the contents of the shift register
# Jump and Branch Instructions
  -
    name: JMP
    opcode: 0x0D
    description: Jump to specified address
    requires_operand: True
    flags:
      other:
        -
          signals: [IO, JMP]
          description: Move IR contents to the program counter
  -
    name: JPI
    opcode: 0x0E
    description: Jump indirectly to an address stored at the RAM address
    requires_operand: True
    flags:
      other:
```

```
      -
        signals: [IO, MI]
        description: Move IR contents to the MAR
      -
        signals: [RO, JMP]
        description: Move RAM contents to the program counter
-
  name: BRZ
  opcode: 0x0F
  description: Branch to an address if the Zero Flag is set
  requires_operand: True
  flags:
    ZF:
      -
        signals: [IO, JMP]
        description: Move IR contents to the program counter if the Zero Flag is set
-
  name: BRP
  opcode: 0x10
  description: Branch to an address if the Parity Flag is set
  requires_operand: True
  flags:
    PF:
      -
        signals: [IO, JMP]
        description:  Move IR contents to the program counter if the Parity Flag is
-
  name: BRC
  opcode: 0x11
  description: Branch to an address if the Carry Flag is set
  requires_operand: True
  flags:
    CF:
```

```
      -
        signals: [IO, JMP]
        description: Move IR contents to the program counter if the Carry Flag is se
# Move from A instructions
  -
    name: MAB
    opcode: 0x12
    description: Move register A contents to the B register
    requires_operand: False
    flags:
      other:
        -
          signals: [AO, BI]
          description: Move register A contents to the B register
  -
    name: MAC
    opcode: 0x13
    description: Move register A contents to the C register
    requires_operand: False
    flags:
      other:
        -
          signals: [AO, CI]
          description: Move register A contents to the C register
  -
    name: MAS
    opcode: 0x14
    description: Move register A contents to the shift register
    requires_operand: False
    flags:
      other:
        -
          signals: [AO, SI]
```

```
        description: Move register A contents to the shift register
# Move from B instructions
  -
    name: MBA
    opcode: 0x15
    description: Move register B contents to the A register
    requires_operand: False
    flags:
      other:
        -
          signals: [BO, AI]
          description: Move register B contents to the A register
  -
    name: MBC
    opcode: 0x16
    description: Move register B contents to the C register
    requires_operand: False
    flags:
      other:
        -
          signals: [BO, CI]
          description: Move register B contents to the C register
  -
    name: MBS
    opcode: 0x17
    description: Move register B contents to the shift register
    requires_operand: False
    flags:
      other:
        -
          signals: [BO, SI]
          description: Move register B contents to the shift register
# Move from C instructions
```

```
  -
    name: MCA
    opcode: 0x18
    description: Move register C contents to the A register
    requires_operand: False
    flags:
      other:
        -
          signals: [CO, AI]
          description: Move register C contents to the A register
  -
    name: MCB
    opcode: 0x19
    description: Move register C contents to the B register
    requires_operand: False
    flags:
      other:
        -
          signals: [CO, BI]
          description: Move register C contents to the B register
  -
    name: MCS
    opcode: 0x1A
    description: Move register C contents to the shift register
    requires_operand: False
    flags:
      other:
        -
          signals: [CO, SI]
          description: Move register C contents to the shift register.
# Move from shift register instructions
  -
    name: MSA
```

```
    opcode: 0x1B
    description: Move shift register contents to the A register
    requires_operand: False
    flags:
      other:
        -
          signals: [SO, AI]
          description: Move shift register contents to the A register
  -
    name: MSB
    opcode: 0x1C
    description: Move shift register contents to the B register
    requires_operand: False
    flags:
      other:
        -
          signals: [SO, BI]
          description: Move shift register contents to the B register
  -
    name: MSC
    opcode: 0x1D
    description: Move shift register contents to the C register
    requires_operand: False
    flags:
      other:
        -
          signals: [SO, CI]
          description: Move shift register contents to the C register
# RAM write instructions
  -
    name: STA
    opcode: 0x1E
    description: Write the contents of the A register to RAM address stored in operand
```

```
    requires_operand: True
    flags:
      other:
        -
          signals: [IO, MI]
          description: Move IR contents to the MAR
        -
          signals: [AO, RI]
          description: Move register A contents to RAM
-
    name: STB
    opcode: 0x1F
    description: Write the contents of the B register to RAM address stored in operand
    requires_operand: True
    flags:
      other:
        -
          signals: [IO, MI]
          description: Move IR contents to MAR
        -
          signals: [BO, RI]
          description: Move register B contents to RAM
-
    name: STC
    opcode: 0x20
    description: Write the contents of the C register to RAM address stored in the ope
    requires_operand: True
    flags:
      other:
        -
          signals: [IO, MI]
          description: Move IR contents to MAR
        -
```

```
        signals: [CO, RI]
        description: Move register C contents to RAM
  -
    name: STS
    opcode: 0x21
    description: Write the contents of the shift register to RAM address stored in the
    requires_operand: True
    flags:
      other:
        -
          signals: [IO, MI]
          description: Move IR contents to RAM
        -
          signals: [SO, RI]
          description: Move shift register contents to RAM
# Stack Manipulation
  -
    name: JSR
    opcode: 0x22
    description: Jump to subroutine
    requires_operand: True
    flags:
      other:
        -
          signals: [STO, MI]
          description: Move stack pointer contents to the MAR
        -
          signals: [CTO, MI]
          description: Store program counter contents in RAM stack
        -
          signals: [STI, IO, JMP]
          description: Increment stack pointer; Move IR contents to the program counte
  -
```

```
    name: RET
    opcode: 0x23
    description: Return from subroutine
    requires_operand: False
    flags:
      other:
        -
          signals: [STD]
          description: Decrement stack pointer
        -
          signals: [STO, MI]
          description: Move Stack pointer contents to the MAR
        -
          signals: [RO, JMP]
          description: Move RAM contents to the program counter
# I/O Read Instructions
  -
    name: IRA
    opcode: 0x24
    description: Read from I/O to the A register
    requires_operand: False
    flags:
      other:
        -
          signals: [E, RW, AI]
          description: Enable I/O, set in read mode; Move data word to the A register
  -
    name: IRB
    opcode: 0x25
    description: Read from I/O to the B register
    requires_operand: False
    flags:
      other:
```

```
      -
        signals: [E, RW, BI]
        description: Enable I/O, set in read mode; Move data word to the B register
-
  name: IRC
  opcode: 0x26
  description: Read from I/O to the C register
  requires_operand: False
  flags:
    other:
      -
        signals: [E, RW, CI]
        description: Enable I/O, set in read mode; Move data word to the C register
-
  name: IRS
  opcode: 0x27
  description: Read from I/O to the shift register
  requires_operand: False
  flags:
    other:
      -
        signals: [E, RW, SI]
        description: Enable I/O, set in read mode; Move data word to the shift regis
-
  name: IRR
  opcode: 0x28
  description: Read from I/O to RAM address specified in the instruction operand
  requires_operand: True
  flags:
    other:
      -
        signals: [IO, MI]
        description: Move IR contents to MAR
```

```
      -
        signals: [E, RW, RI]
        description: Enable I/O, set in read mode, Move data word to RAM
# I/O write instructions
  -
    name: IWA
    opcode: 0x29
    description: Write A register contents to I/O
    requires_operand: False
    flags:
      other:
        -
          signals: [E, AO]
          description: Enable I/O, set write mode; Move register A contents to I/O
  -
    name: IWB
    opcode: 0x2A
    description: Write B register contents to I/O
    requires_operand: False
    flags:
      other:
        -
          signals: [E, BO]
          description: Enable I/O, set write mode; Move register B contents to I/O
  -
    name: IWC
    opcode: 0x2B
    description: Write C register contents to I/O
    requires_operand: False
    flags:
      other:
        -
          signals: [E, CO]
```

```
        description: Enable I/O, set write mode; Move register C contents to I/O
  -
    name: IWS
    opcode: 0x2C
    description: Write shift register contents to I/O
    requires_operand: False
    flags:
      other:
        -
          signals: [E, SO]
          description: Enable I/O, set write mode; Move shift register contents to I/O
  -
    name: IWR
    opcode: 0x2D
    description: Write RAM contents at address specified by instruction operand to I/O
    requires_operand: True
    flags:
      other:
        -
          signals: [IO, MI]
          description: Move IR contents to the MAR
        -
          signals: [E, RO]
          description: Enable I/O, set write mode; Move RAM contents to I\O
# Display instructions
  -
    name: DWA
    opcode: 0x2E
    description: Write register A contents to the Display
    requires_operand: False
    flags:
      other:
        -
```

```
        signals: [AO, OUT]
        description: Move contents of the A register to the Display
-
  name: DWB
  opcode: 0x2F
  description: Write register B contents to the Display
  requires_operand: False
  flags:
    other:
      -
        signals: [BO, OUT]
        description: Move contents of the B register to the Display
-
  name: DWC
  opcode: 0x30
  description: Write register C contents to the Display
  requires_operand: False
  flags:
    other:
      -
        signals: [CO, OUT]
        description: Move contents of the C register to the Display
-
  name: DWS
  opcode: 0x31
  description: Write shift register contents to the Display
  requires_operand: False
  flags:
    other:
      -
        signals: [SO, OUT]
        description: Move contents of the shift register to the Display
-
```

```
    name: DWR
    opcode: 0x32
    description: Write RAM contents at address specified by instruction operand to the
    requires_operand: True
    flags:
      other:
        -
          signals: [IO, MI]
          description: Move IR contents to the MAR
        -
          signals: [AO, OUT]
          description: Move RAM contents to the Display

# Computer and clock manipulation
  -
    name: RST
    opcode: 0x3E
    description: Reset the computer
    requires_operand: False
    flags:
      other:
        -
          signals: [RST]
          description: Reset the computer
  -
    name: HLT
    opcode: 0x3F
    description: Stop the clock
    requires_operand: False
    flags:
      other:
        -
          signals: [HLT]
```

```
        description: Stop the clock
```

3.3 Assembler

3.3.1 assembler.scala

```
package buildtools.assembler

// Builds assmelby file into binaries

object Assembler extends App {

  import buildtools.assembler.{Lexer, YAMLParser}
  import scala.io.Source
  import java.io.BufferedOutputStream
  import java.io.FileOutputStream

// constants
  val MAIN_BASE = 0
  val DATA_BASE = 512
  val MEM_SIZE = 1024

  type Env = Map[String, Int]



  def format_binary_string(nr: Int, length: Int) =
    ("0" * (length - nr.toBinaryString.length)) + nr.toBinaryString

  def make_bin_str2(nr1: Int, nr2: Int = 0) =
      format_binary_string(nr1, 6) + format_binary_string(nr2, 10)

// parse an operator into an int
  def parse_ope(ope: String): Int = ope.toList match {
    case '0'::'b'::res => Integer.parseInt(res.mkString, 2)
```

```
    case '0'::'x'::res => Integer.parseInt(res.mkString, 16)
    case '0'::'d'::res => Integer.parseInt(res.mkString, 10)
    case _ => Integer.parseInt(ope, 10)
  }

// parse aliases
  def create_env(tks: List[(String, String)],
                 env:Env = Map(),
                 processed: List[(String, String)] = Nil)
                : (List[(String, String)], Map[String, Int]) = tks match {
    case Nil => (processed, env)
    case ("id", a)::("ops", "=")::("ope", b)::rest =>
                 create_env(rest, env + (a -> parse_ope(b)), processed)
    case a::rest => create_env(rest, env, processed:+a)
    }

// oreder subroutines
  def get_sub_order(tks: List[(String, String)], res: List[String] = Nil) : List[Strin
    case Nil => res
    case ("id", a)::("ops", ":")::rest => get_sub_order(rest, res:+a)
    case _::rest => get_sub_order(rest, res)
  }

  def separate_subroutines(tks: List[(String, String)],
                           subroutines: Map[String, List[(String, String)]] = Map(),
                           label: String= "") : Map[String, List[(String, String)]] = t
    case Nil => subroutines
    case ("id", a)::("ops", ":")::rest => separate_subroutines(rest, subroutines, a)
    case a::rest => separate_subroutines(rest, subroutines +
                    (label -> (subroutines.getOrElse(label, Nil) :+ a)), label)
  }

// count subroutine instructions. Used to determine the value for each subroutine labl
```

```
  def cnt_subroutine_instr(tks: List[(String, String)], acc: Int=0) : Int = tks match
    case Nil => acc
    case ("ins", _)::rest => cnt_subroutine_instr(rest, acc + 1)
    case _::rest => cnt_subroutine_instr(rest, acc)
  }

// Determine subroutine labels
  def solve_references_rec(keys: List[String], subroutines: Map[String, Int],
                           env: Env, base: Int) : Map[String, Int] = keys match {
      case Nil => env
      case s::rest => solve_references_rec(rest, subroutines, env + (s -> base), bas
    }

  def solve_references(subroutines: Map[String, Int],
                       env: Env,
                       keys: List[String]) : Map[String, Int] = {
    subroutines.getOrElse("main", throw new Exception("No main section defined!"))
    val base = 0
    val env1 = env + ("main" -> base)
    val base1 = base + subroutines.get("main").get
    solve_references_rec(keys, subroutines, env1, base1)
  }

// Turn a subroutine into a map from address to binary value as a string
  def process_subroutine(tks: List[(String, String)], name: String, base: Int,
                         env: Env,
                         instrs: Map[String, YAMLParser.Instruction],
                         res: Map[Int, String] = Map()) : Map[Int, String] = name mat
    case "data" => tks match {
      case Nil => res
      case ("ope", a)::("ops", "set")::("ope", b)::rest => {
        val a_val = parse_ope(a)
        val b_val = parse_ope(b)
```

```
      if(a_val < 512 || a_val > 1007) throw new Exception("Cannot define data outs
      process_subroutine(rest, name, base, env, instrs, res + (a_val -> format_bi
    }
    case ("ope", a)::("ops", "set")::("id", id)::rest => {
      val a_val = parse_ope(a)
      val b_val = env.get(id).get
      if(a_val < 512 || a_val > 1007) throw new Exception("Cannot define data outs
      process_subroutine(rest, name, base, env, instrs, res + (a_val -> format_bi
    }
    case ("id", id)::("ops", "set")::("ope", b)::rest => {
      val a_val = env.get(id).get
      val b_val = parse_ope(b)
      if(a_val < 512 || a_val > 1007) throw new Exception("Cannot define data outs
      process_subroutine(rest, name, base, env, instrs, res + (a_val -> format_bi
    }
    case ("id", id1)::("ops", "set")::("id", id2)::rest => {
      val a_val = env.get(id1).get
      val b_val = env.get(id2).get
      if(a_val < 512 || a_val > 1007) throw new Exception("Cannot define data outs
      process_subroutine(rest, name, base, env, instrs, res + (a_val -> format_bi
    }
    case a => throw new Exception("Instructions are not allowed in the data sectio
  }
  case _ => tks match {
      case Nil => res
      case ("ins", a)::("id", b)::rest => try {
        process_subroutine(rest, name, base + 1, env, instrs,
          res + (base -> make_bin_str2(instrs.get(a).get.opcode, env.get(b).get)))
      } catch {
        case e: Exception => {
          println(a, " ", b)
          println(instrs.get(a))
          println(env.get(b))
```

```
            println(env)
            throw e
          }
        }
        case ("ins", a)::("ope", b)::rest => process_subroutine(rest, name, base + 1
                                                res + (base -> make_bin_str2(instrs.ge
        case ("ins", a)::rest => if(!instrs.get(a).get.requires_operand) process_sub
                                   env, instrs, res + (base -> make_bin_str2(instrs
                                   throw new Exception("Missing operand for " + a)
    }

  }

def process_subroutines_rec(keys: List[String],
                            subroutines: Map[String, List[(String, String)]],
                            env: Env,
                            instr_map: Map[String, YAMLParser.Instruction],
                            res: List[Map[Int, String]] = Nil) : List[Map[Int, Strin
  case Nil => res
  case n::rest => {
    val res_n = process_subroutine(subroutines.get(n).get, n, env.get(n).get, env, i
    process_subroutines_rec(rest, subroutines, env, instr_map, res:+res_n)
  }

                            }

def merge_maps_rec(acc: Map[Int, String] = Map(), maps: List[Map[Int, String]]) : Ma
  case Nil => acc
  case m::rest => merge_maps_rec(acc ++ m, rest)
}

def process_subroutines(subroutines: Map[String, List[(String, String)]],
                        env: Env,
```

```
                                instr_map: Map[String, YAMLParser.Instruction]
                              ) : Map[Int, String] = {
  val main = process_subroutine(subroutines.get("main").get, "main", MAIN_BASE, env,
  val data = process_subroutine(subroutines.getOrElse("data", Nil), "data", DATA_BAS
  val rest = subroutines.keys.toList.filter({case n => n != "main" && n != "data"})
  val result_rest = process_subroutines_rec(rest, subroutines, env, instr_map)

  val final_res : Map[Int, String] = main ++ data
  merge_maps_rec(final_res, result_rest)
  }

// Turn the string binary data into bytes
def createByteList(map: Map[Int, String], acc: List[Byte]=Nil, i: Int=0) : List[Byte
  if(i >= 1024) acc else {
    val data_word = map.getOrElse(i, "0" * 16)
    if(data_word.length != 16) throw new Exception("Invalid data word: " + data_word
    val byte1 = Integer.parseInt(data_word.slice(0, 8), 2).toByte
    val byte2 = Integer.parseInt(data_word.slice(9, 16), 2).toByte
    createByteList(map, acc ::: List(byte1, byte2), i + 1)
  }
}

def assemble(asm_text : String, isa_text : String) : List[Byte] = {
  print("Trying to parse YAML text... ")
  val yaml_obj = YAMLParser.parseYamlISA(isa_text)
  println("Done.")

  print("Trying to lex assembly source... ")
  val tks = Lexer.lex_assembly_based_on_isa(asm_text, yaml_obj._2)
  println("Done.")

  print("Extracting aliases... ")
  val env_init = create_env(tks)
```

```
    println("Done.")

    print("Ordering subroutines... ")
    val sub_order = get_sub_order(env_init._1)
    val sub_order_1 = sub_order.filter({case n => n != "data" && n != "main"})
    println("Done")

    print("Extracting subroutines... ")
    val subroutines = separate_subroutines(env_init._1)
    println("Done.")

    print("Counting subroutine instructions... ")
    val subroutine_cnt_map = subroutines map {case (name, lst) => (name -> cnt_subrou
    println("Done.")

    print("Solving references... ")
    val env = solve_references(subroutine_cnt_map, env_init._2, sub_order_1)
    println("Done.")

    print("Decoding Instructions... ")
    val final_res = process_subroutines(subroutines, env, yaml_obj._3)
    println("Done.")

    print("Generating byte array... ")
    val bytes = createByteList(final_res)
    println("Done.")
    bytes
  }
}
```

3.3.2 lexer.scala

```
package buildtools.assembler
```

```
// Based on the 6CCS3CFL coursework, taught by Dr. Christian Urban
// Based on regular expression derivatives

object Lexer {
  import buildtools.assembler.LexerCore
  import buildtools.assembler.RegexDef._

// Convenience functions
  implicit def string2rexp(s : String) : Rexp =
    charlist2rexp(s.toList)

    implicit def RexpOps(r: Rexp) = new {
      def | (s: Rexp) = ALT(r, s)
      def % = STAR(r)
      def ~ (s: Rexp) = SEQ(r, s)
      def + = PLUS(r)
      def ? = OPTIONAL(r)
      def ^ (n: Int) = NTIMES(r, n)
    }

    implicit def stringOps(s: String) = new {
      def | (r: Rexp) = ALT(s, r)
      def | (r: String) = ALT(s, r)
      def % = STAR(s)
      def ~ (r: Rexp) = SEQ(s, r)
      def ~ (r: String) = SEQ(s, r)
      def + = PLUS(s)
      def ? = OPTIONAL(s)
      def ^ (n: Int) = NTIMES(s, n)
      def $ (r: Rexp) = RECD(s, r)
    }
```

```
def charlist2rexp(s : List[Char]): Rexp = s match {
  case Nil => ONE
  case c::Nil => Character(c)
  case c::s => SEQ(Character(c), charlist2rexp(s))
}
```

```
val UPPER = RANGE(('A' to 'Z').toSet)
val LOWER = RANGE(('a' to 'z').toSet)
val ZERO_DIGITS = RANGE(('0' to '9').toSet)
val ONE_DIGITS = RANGE(('1' to '9').toSet)
val SYM = RANGE(('A' to 'Z').toSet ++ ('a' to 'z').toSet ++ Set('_'))

val NOT_QUOTES = CFUN(_ != '"')
val NOT_NEWLINES = CFUN(_ != '\n')

val WHITESPACE = PLUS(" " | "\n" | "\t" | "\r")
val ID = SYM ~ (SYM | ZERO_DIGITS).%
val BINARY_NR = "0b" ~ PLUS("0" | "1")
val HEX_NR = "0x" ~ PLUS(ZERO_DIGITS | RANGE(('a' to 'f').toSet) | RANGE
val DEC_NR = "0d" ~ PLUS(ZERO_DIGITS) | PLUS(ZERO_DIGITS)
val OPERAND = BINARY_NR | HEX_NR | DEC_NR
val COMMENT = "#" ~ STAR(NOT_NEWLINES)
val OPS = ":" | "=" | "set"
```

```
def strListToRexp(lst: List[String], r: Rexp = ZERO) : Rexp = {
  lst match {
    case Nil => r
    case name::rest => strListToRexp(rest, r | name)
  }
}
```

```
    // Remove comments and whitespace from the token List
    def sanitize(tks: List[(String, String)]) : List[(String, String)]= tks match {
      case Nil => Nil
      case t::ts => t match {
        case ("whi", _) => sanitize(ts)
        case ("com", _) => sanitize(ts)
        case _ => t::sanitize(ts)
      }
    }

    def lex_assembly_based_on_isa(assembly_text: String, instr_names: List[String]) :
                val INSTRUCTIONS = strListToRexp(instr_names)

                val ASSEMBLER_REGS = (("ins" $ INSTRUCTIONS) |
                  ("ops" $ OPS) |
                  ("ope" $ OPERAND) |
                  ("com" $ COMMENT) |
                  ("whi" $ WHITESPACE) |
                  ("id" $ ID)).%

                  sanitize(LexerCore.lex(assembly_text, ASSEMBLER_REGS))
                }
            }
```

3.3.3 LexerCore.scala

```
package buildtools.assembler

// Based on the 6CCS3CFL coursework, taught by Dr. Christian Urban
// Based on regular expression derivatives

object LexerCore {
```

```
  import scala.language.implicitConversions
  import scala.language.reflectiveCalls

  // regular expressions including records
  import buildtools.assembler.RegexDef._

  // values
  abstract class Val
  case object Empty extends Val
  case class Chr(c: Char) extends Val
  case class Sequ(v1: Val, v2: Val) extends Val
  case class Left(v: Val) extends Val
  case class Right(v: Val) extends Val
  case class Stars(vs: List[Val]) extends Val
  case class Rec(x: String, v: Val) extends Val


// Convenience functions
    implicit def string2rexp(s : String) : Rexp =
      charlist2rexp(s.toList)

      implicit def RexpOps(r: Rexp) = new {
        def | (s: Rexp) = ALT(r, s)
        def % = STAR(r)
        def ~ (s: Rexp) = SEQ(r, s)
        def + = PLUS(r)
        def ? = OPTIONAL(r)
        def ^ (n: Int) = NTIMES(r, n)
      }

      implicit def stringOps(s: String) = new {
        def | (r: Rexp) = ALT(s, r)
        def | (r: String) = ALT(s, r)
```

```
    def % = STAR(s)
    def ~ (r: Rexp) = SEQ(s, r)
    def ~ (r: String) = SEQ(s, r)
    def + = PLUS(s)
    def ? = OPTIONAL(s)
    def ^ (n: Int) = NTIMES(s, n)
    def $ (r: Rexp) = RECD(s, r)
  }



def charlist2rexp(s : List[Char]): Rexp = s match {
  case Nil => ONE
  case c::Nil => Character(c)
  case c::s => SEQ(Character(c), charlist2rexp(s))
}


// Check if a regex matches the empty string
  def nullable(r: Rexp) : Boolean = r match {
    case ZERO => false
    case ONE => true
    case CFUN(_) => false
    case ALT(r1, r2) => nullable(r1) || nullable(r2)
    case SEQ(r1, r2) => nullable(r1) && nullable(r2)
    case STAR(_) => true
    case NTIMES(r, i) => if (i == 0) true else nullable(r)
    case PLUS(r) => nullable(r)
    case OPTIONAL(r) => true
    case RECD(_, r1) => nullable(r1)
  }

// derivation with respect to a character
```

```
def der(c: Char, r: Rexp) : Rexp = r match {
  case ZERO => ZERO
  case ONE => ZERO
  case CFUN(f) => if (f(c)) ONE else ZERO
  case ALT(r1, r2) => ALT(der(c, r1), der(c, r2))
  case SEQ(r1, r2) =>
  if (nullable(r1)) ALT(SEQ(der(c, r1), r2), der(c, r2))
  else SEQ(der(c, r1), r2)
  case STAR(r) => SEQ(der(c, r), STAR(r))
  case NTIMES(r, i) =>
  if (i == 0) ZERO else SEQ(der(c, r), NTIMES(r, i - 1))
  case PLUS(r) => SEQ(der(c, r), STAR(r))
  case OPTIONAL(r) => der(c, r)
  case RECD(_, r1) => der(c, r1)
}


// extracts a string from value
def flatten(v: Val) : String = v match {
  case Empty => ""
  case Chr(c) => c.toString
  case Left(v) => flatten(v)
  case Right(v) => flatten(v)
  case Sequ(v1, v2) => flatten(v1) + flatten(v2)
  case Stars(vs) => vs.map(flatten).mkString
  case Rec(_, v) => flatten(v)
}


// extracts an environment from a value;
// used to tokenise a string
def env(v: Val) : List[(String, String)] = v match {
  case Empty => Nil
```

```
  case Chr(c) => Nil
  case Left(v) => env(v)
  case Right(v) => env(v)
  case Sequ(v1, v2) => env(v1) ::: env(v2)
  case Stars(vs) => vs.flatMap(env)
  case Rec(x, v) => (x, flatten(v))::env(v)
}

// The Injection Part of the lexer

def mkeps(r: Rexp) : Val = r match {
  case ONE => Empty
  case ALT(r1, r2) =>
  if (nullable(r1)) Left(mkeps(r1)) else Right(mkeps(r2))
  case SEQ(r1, r2) => Sequ(mkeps(r1), mkeps(r2))
  case STAR(r) => Stars(Nil)
  case PLUS(r) => Stars(List(mkeps(r)))
  case OPTIONAL(r) => if (nullable(r)) Left(mkeps(r)) else Right(Empty)
  case NTIMES(r, n) => Stars(List.fill(n)(mkeps(r)))
  case RECD(x, r) => Rec(x, mkeps(r))
}

def inj(r: Rexp, c: Char, v: Val) : Val = (r, v) match {
  case (STAR(r), Sequ(v1, Stars(vs))) => Stars(inj(r, c, v1)::vs)
  case (SEQ(r1, r2), Sequ(v1, v2)) => Sequ(inj(r1, c, v1), v2)
  case (SEQ(r1, r2), Left(Sequ(v1, v2))) => Sequ(inj(r1, c, v1), v2)
  case (SEQ(r1, r2), Right(v2)) => Sequ(mkeps(r1), inj(r2, c, v2))
  case (ALT(r1, r2), Left(v1)) => Left(inj(r1, c, v1))
  case (ALT(r1, r2), Right(v2)) => Right(inj(r2, c, v2))
  case (CFUN(d), Empty) => Chr(c)
  case (PLUS(r), Sequ(v1, Stars(vs))) => Stars(inj(r, c, v1)::vs)
  case (OPTIONAL(r), Left(v1)) => Left(inj(r, c, v1))
  case (OPTIONAL(r), Right(v2)) => Right(inj(r, c, v2))
```

```
  case (NTIMES(r, n), Sequ(v1, Stars(vs))) => Stars(inj(r, c, v1)::vs)
  case (RECD(x, r1), _) => Rec(x, inj(r1, c, v))
}

// some "rectification" functions for simplification
def F_ID(v: Val): Val = v
def F_RIGHT(f: Val => Val) = (v:Val) => Right(f(v))
def F_LEFT(f: Val => Val) = (v:Val) => Left(f(v))
def F_ALT(f1: Val => Val, f2: Val => Val) = (v:Val) => v match {
  case Right(v) => Right(f2(v))
  case Left(v) => Left(f1(v))
}
def F_SEQ(f1: Val => Val, f2: Val => Val) = (v:Val) => v match {
  case Sequ(v1, v2) => Sequ(f1(v1), f2(v2))
}
def F_SEQ_Empty1(f1: Val => Val, f2: Val => Val) =
  (v:Val) => Sequ(f1(Empty), f2(v))
  def F_SEQ_Empty2(f1: Val => Val, f2: Val => Val) =
    (v:Val) => Sequ(f1(v), f2(Empty))
    def F_RECD(f: Val => Val) = (v:Val) => v match {
      case Rec(x, v) => Rec(x, f(v))
    }
    def F_ERROR(v: Val): Val = throw new Exception("error")

    def simp(r: Rexp): (Rexp, Val => Val) = r match {
      case ALT(r1, r2) => {
        val (r1s, f1s) = simp(r1)
        val (r2s, f2s) = simp(r2)
        (r1s, r2s) match {
          case (ZERO, _) => (r2s, F_RIGHT(f2s))
          case (_, ZERO) => (r1s, F_LEFT(f1s))
          case _ => if (r1s == r2s) (r1s, F_LEFT(f1s))
          else (ALT (r1s, r2s), F_ALT(f1s, f2s))
```

```
    }
  }
  case SEQ(r1, r2) => {
    val (r1s, f1s) = simp(r1)
    val (r2s, f2s) = simp(r2)
    (r1s, r2s) match {
      case (ZERO, _) => (ZERO, F_ERROR)
      case (_, ZERO) => (ZERO, F_ERROR)
      case (ONE, _) => (r2s, F_SEQ_Empty1(f1s, f2s))
      case (_, ONE) => (r1s, F_SEQ_Empty2(f1s, f2s))
      case _ => (SEQ(r1s,r2s), F_SEQ(f1s, f2s))
    }
  }
  case RECD(x, r1) => {
    val (r1s, f1s) = simp(r1)
    (RECD(x, r1s), F_RECD(f1s))
  }
  case r => (r, F_ID)
}

// lexing functions including simplification
def lex_simp(r: Rexp, s: List[Char]) : Val = s match {
  case Nil => if (nullable(r)) mkeps(r) else
  { throw new Exception("lexing_error") }
  case c::cs => {
    val (r_simp, f_simp) = simp(der(c, r))
    inj(r, c, f_simp(lex_simp(r_simp, cs)))
  }
}

def lexing_simp(r: Rexp, s: String) =
  env(lex_simp(r, s.toList))
```

```
// escapes strings and prints them out as "", "\n" and so on
def esc(raw: String): String = {
  import scala.reflect.runtime.universe._
  Literal(Constant(raw)).toString
}

def escape(tks: List[(String, String)]) =
  tks.map{ case (s1, s2) => (s1, esc(s2))}


  def strListToRexp(lst: List[String], r: Rexp = ZERO) : Rexp = {
    lst match {
      case Nil => r
      case name::rest => strListToRexp(rest, r | name)
    }
  }

  def lex(s: String, r: Rexp) : List[(String, String)] =
    lexing_simp(r, s)



  }
```

3.3.4 regexdef.scala

```
package buildtools.assembler
// Based on the 6CCS3CFL coursework, taught by Dr. Christian Urban
// Based on regular expression derivatives

// type definitions for all lexers
object RegexDef {
  abstract class Rexp
  case object ZERO extends Rexp
  case object ONE extends Rexp
```

```
  case class CFUN(f : Char => Boolean) extends Rexp
  case class ALT(r1: Rexp, r2: Rexp) extends Rexp
  case class SEQ(r1: Rexp, r2: Rexp) extends Rexp
  case class STAR(r: Rexp) extends Rexp
  case class PLUS(r: Rexp) extends Rexp
  case class OPTIONAL(r: Rexp) extends Rexp
  case class NTIMES(r: Rexp, n: Int) extends Rexp
  case class RECD(x: String, r: Rexp) extends Rexp

  def Character(c: Char) = CFUN(_ == c)
  def RANGE(cs: Set[Char]) = CFUN(cs.contains(_))
  def ALL = CFUN(_ => true)



}
```

3.3.5 regexdef.scala

```
package buildtools.assembler
// Based on the 6CCS3CFL coursework, taught by Dr. Christian Urban
// Based on regular expression derivatives

// type definitions for all lexers
object RegexDef {
  abstract class Rexp
  case object ZERO extends Rexp
  case object ONE extends Rexp
  case class CFUN(f : Char => Boolean) extends Rexp
  case class ALT(r1: Rexp, r2: Rexp) extends Rexp
  case class SEQ(r1: Rexp, r2: Rexp) extends Rexp
  case class STAR(r: Rexp) extends Rexp
  case class PLUS(r: Rexp) extends Rexp
  case class OPTIONAL(r: Rexp) extends Rexp
  case class NTIMES(r: Rexp, n: Int) extends Rexp
```

```
  case class RECD(x: String, r: Rexp) extends Rexp

  def Character(c: Char) = CFUN(_ == c)
  def RANGE(cs: Set[Char]) = CFUN(cs.contains(_))
  def ALL = CFUN(_ => true)


}
```

3.3.6 yamlparser.scala

```
package buildtools.assembler


// Scala YAML parser based on moultingyaml
object YAMLParser {
  import net.jcazevedo.moultingyaml._
  import scala.io.Source

  case class Step(signals: Seq[String], description: String)
  case class Instruction(name: String, opcode: Int, description: String, requires_oper
  case class ISA(instructions: Seq[Instruction])
  object ISAYamlProtocol extends DefaultYamlProtocol {
    implicit val stepFormat = yamlFormat2(Step)
    implicit val instructionFormat = yamlFormat5(Instruction)
    implicit val isaFormat = yamlFormat1(ISA)
  }

  import ISAYamlProtocol._

  def parseYamlISA(yamlText: String) : (ISA, List[String], Map[String, Instruction]) =
    val isaYaml = yamlText.stripMargin.parseYaml.convertTo[ISA]

    val instr_names : List[String] = isaYaml.instructions.map(i => i.name).toList
```

```
    val instr_map : Map[String, Instruction] = instr_names.map(
      name => name -> isaYaml.instructions.find(i => i.name == name).getOrElse(isaYaml
      .toMap
    (isaYaml, instr_names, instr_map)
  }
}
```

3.4 Compiler

3.4.1 ast.scala

```
package buildtools.compiler
// Based on the 6CCS3CFL coursework, taught by Dr. Christian Urban

// WHILE Language abstract syntax trees
object AST {
  abstract class Stmt
  abstract class AExp
  abstract class BExp

  type Block = List[Stmt]

  case object Skip extends Stmt
  case class If(a: BExp, bl1: Block, bl2: Block) extends Stmt
  case class While(b: BExp, bl: Block) extends Stmt
  case class Assign(s: String, a: AExp) extends Stmt
  case class WriteAExp(a: AExp) extends Stmt
  case class Read(s: String) extends Stmt



  case class Var(s: String) extends AExp
  case class Num(i: Int) extends AExp
  case class Aop(o: String, a1: AExp, a2: AExp) extends AExp
```

```
  case object True extends BExp
  case object False extends BExp
  case class Bop(o: String, a1: AExp, a2: AExp) extends BExp
}
```

3.4.2 compiler.scala

```
package buildtools.compiler

// Based on the 6CCS3CFL coursework, taught by Dr. Christian Urban
// Based on regular expression derivatives

object Compiler {


  import scala.language.implicitConversions
  import scala.language.reflectiveCalls

  // Abstract Syntax Trees for the WHILE Language
  import buildtools.compiler.AST._

  // environments
  type Env = Map[String, Int]

  // for generating new labels
  var counter = 0

  def Fresh(x: String) : String = {
    val res =   x ++ "_" ++ counter.toString()
    counter += 1
    res
  }
  // this allows you to write things like
  // i"add" and l"Label"
```

```
  implicit def string_inters(sc: StringContext) = new {
    def i(args: Any*): String = "    " ++ sc.s(args:_*) ++ "\n"
    def l(args: Any*): String = sc.s(args:_*) ++ ":\n"
  }

// Constants
  val OPSPACE = "0x3EF"

  val VARS_BASE = 512

  val CONSTANTS = i"MAXINT = 0x3EE"

  val DATA = l"data" ++ i"MAXINT = 0xFFFF"

  val beginning = CONSTANTS ++ DATA

  val ending = i"\n# Compiled assembly for the 16-bit breadboard computer"

  // Compiler functions

  def compile_op(op: String) : String = op match {
    case "+" => i"ADD"
    case "-" => i"SUB"
    case "*" => {
      val start = Fresh("START")
      val done = Fresh("DONE")
      i"MAS" ++
      i"LAI 0" ++
      l"$done" ++
      i"MAC" ++
      i"MBA" ++
      i"LBI 0" ++
      i"ADD" ++
```

```
    i"BRZ $done" ++
    i"LBI 1" ++
    i"SUB" ++
    i"MAB" ++
    i"MCA" ++
    i"MBC" ++
    i"MSB" ++
    i"ADD" ++
    i"MCB" ++
    i"JMP $start" ++
    l"$done" ++
    i"MCA"
  }
}



// arithmetic expression compilation
  def compile_aexp(a: AExp, env : Env) : String = a match {
    case Num(i) => i"LAI 0d$i"
    case Var(s) => {
      val index = env.get(s).get
      i"LDA ${VARS_BASE + index}"
    }
    case Aop(op, a1, a2) => compile_aexp(a2, env) ++ i"STA $OPSPACE" ++
                            compile_aexp(a1, env) ++ i"LDB $OPSPACE" ++ compile_op(o
  }

// boolean expression compilation
// - the jump-label is for where to jump if the condition is not true
def compile_bexp(b: BExp, env : Env, jmp: String) : String = b match {
  case True => ""
  case False => i"JMP $jmp"
```

```
case Bop("==", a1, a2) =>
  compile_aexp(a2, env) ++
  i"STA $OPSPACE" ++
  compile_aexp(a1, env) ++
  i"LDB $OPSPACE" ++
  i"SUB" ++
  i"LDB MAXINT" ++
  i"ADD" ++
  i"BRC $jmp"
case Bop("!=", a1, a2) =>
  compile_aexp(a2, env) ++
  i"STA $OPSPACE" ++
  compile_aexp(a1, env) ++
  i"LDB $OPSPACE" ++
  i"SUB" ++
  i"BRZ $jmp"
case Bop("<", a1, a2) =>
  compile_aexp(a2, env) ++
  i"STA $OPSPACE" ++
  compile_aexp(a1, env) ++
  i"LDB $OPSPACE" ++
  i"SUB" ++
  i"BRC $jmp"
case Bop("<=", a1, a2) =>
  compile_aexp(a2, env) ++
  i"LBI 1" ++
  i"ADD" ++
  i"STA $OPSPACE" ++
  compile_aexp(a2, env) ++
  i"LDB $OPSPACE" ++
  i"SUB" ++
  i"BRC $jmp"
case Bop(">", a1, a2) =>
```

```
    compile_aexp(a1, env) ++
    i"STA $OPSPACE" ++
    compile_aexp(a2, env) ++
    i"LDB $OPSPACE" ++
    i"SUB" ++
    i"BRC $jmp"
  case Bop(">=", a1, a2) =>
    compile_aexp(a1, env) ++
    i"LBI 1" ++
    i"ADD" ++
    i"STA $OPSPACE" ++
    compile_aexp(a2, env) ++
    i"LDB $OPSPACE" ++
    i"BRC $jmp"
}

// statement compilation
def compile_stmt(s: Stmt, env: Env) : (String, Env) = s match {
  case Skip => (i"NOP", env)
  case Assign(x, a) => {
    val index = env.getOrElse(x, env.keys.size)
    (compile_aexp(a, env) ++ i"STA ${VARS_BASE + index}", env + (x -> index))
  }
  case If(b, bl1, bl2) => {
    val if_else = Fresh("If_else")
    val if_end = Fresh("If_end")
    val (instrs1, env1) = compile_block(bl1, env)
    val (instrs2, env2) = compile_block(bl2, env1)
    (compile_bexp(b, env, if_else) ++
    instrs1 ++
    i"JMP $if_end" ++
    l"$if_else" ++
    instrs2 ++
```

```
    l"$if_end" ++
    i"NOP", env2)
  }
  case While(b, bl) => {
    val loop_begin = Fresh("Loop_begin")
    val loop_end = Fresh("Loop_end")
    val (instrs1, env1) = compile_block(bl, env)
    (l"$loop_begin" ++
      compile_bexp(b, env, loop_end) ++
      instrs1 ++
      i"JMP $loop_begin" ++
      l"$loop_end" ++
      i"NOP", env1)
    }
  case WriteAExp(x) => {
    val instr = compile_aexp(x, env)
    ( instr ++ i"IWA", env)
  }
  case Read(x) => {
    val index = env.getOrElse(x, env.keys.size)
    (i"IRA" ++
      i"STA ${VARS_BASE + index}", env + (x -> index))
  }
}

// compilation of a block (i.e. list of instructions)
def compile_block(bl: Block, env: Env) : (String, Env) = bl match {
  case Nil => ("", env)
  case s::bl => {
    val (instrs1, env1) = compile_stmt(s, env)
    val (instrs2, env2) = compile_block(bl, env1)
    (instrs1 ++ instrs2, env2)
  }
```

```
  }

  // main compilation function for blocks
  def compile(bl: Block) : String = {
    val instructions = compile_block(bl, Map.empty)._1
    (beginning ++ l"main" ++ instructions ++ ending)
  }
}
```

3.4.3 lexer.scala

```
package buildtools.compiler
// Based on the 6CCS3CFL coursework, taught by Dr. Christian Urban
// Based on regular expression derivatives
object Lexer {
    import buildtools.assembler.LexerCore
    import buildtools.assembler.RegexDef._

    // Convenience functions
    implicit def string2rexp(s : String) : Rexp =
      charlist2rexp(s.toList)

      implicit def RexpOps(r: Rexp) = new {
        def | (s: Rexp) = ALT(r, s)
        def % = STAR(r)
        def ~ (s: Rexp) = SEQ(r, s)
        def + = PLUS(r)
        def ? = OPTIONAL(r)
        def ^ (n: Int) = NTIMES(r, n)
      }

      implicit def stringOps(s: String) = new {
        def | (r: Rexp) = ALT(s, r)
        def | (r: String) = ALT(s, r)
```

```
    def % = STAR(s)
    def ~ (r: Rexp) = SEQ(s, r)
    def ~ (r: String) = SEQ(s, r)
    def + = PLUS(s)
    def ? = OPTIONAL(s)
    def ^ (n: Int) = NTIMES(s, n)
    def $ (r: Rexp) = RECD(s, r)
  }

def charlist2rexp(s : List[Char]): Rexp = s match {
  case Nil => ONE
  case c::Nil => Character(c)
  case c::s => SEQ(Character(c), charlist2rexp(s))
}



val UPPER = RANGE(('A' to 'Z').toSet)
val LOWER = RANGE(('a' to 'z').toSet)
val ZERO_DIGITS = RANGE(('0' to '9').toSet)
val ONE_DIGITS = RANGE(('1' to '9').toSet)
val SYM = RANGE(('A' to 'Z').toSet ++ ('a' to 'z').toSet ++ ('0' to '9').toSet ++

val NOT_QUOTES = CFUN(_ != '"')
val NOT_NEWLINES = CFUN(_ != '\n')

val KEYWORD : Rexp =  "while" | "if" | "then" | "else" | "do" | "true" | "false" |
val OP: Rexp = "+" | "-" | "*" | "==" | "!=" | ">" | ">=" | "<" | "<=" | ":="
val PARA: Rexp = "{" | "}" | "(" | ")"
val SEMI: Rexp = ";"
val WHITESPACE = PLUS(" " | "\n" | "\t" | "\r")
val ID = SYM ~ (SYM | ZERO_DIGITS).%
val NUM = "0" | ONE_DIGITS ~ STAR(ZERO_DIGITS)
val COMM : Rexp = "//" ~ STAR(NOT_NEWLINES) ~ "\n" //in-line comments
```

```
    val WHILE_REGS = (("k" $ KEYWORD)  |
    ("op" $ OP)       |
    ("p" $ PARA)      |
    ("s" $ SEMI)      |
    ("w" $ WHITESPACE)  |
    ("n" $ NUM)       |
    ("id" $ ID)       |
    ("c" $ COMM)).%



    // Remove comments and whitespace from the token List
    def sanitize(tks: List[(String, String)]) : List[(String, String)]= tks match {
      case Nil => Nil
      case t::ts => t match {
        case ("w", _) => sanitize(ts)
        case ("c", _) => sanitize(ts)
        case _ => t::sanitize(ts)
      }
    }

    def lex_while(s: String) : List[(String, String)] =
      sanitize(LexerCore.lexing_simp(WHILE_REGS, s))

}
```

3.4.4 parser.scala

```
package buildtools.compiler
// Based on the 6CCS3CFL coursework, taught by Dr. Christian Urban
// Based on parser combinators
object Parser {
```

```
import scala.language.implicitConversions
import scala.language.reflectiveCalls

// Abstract Syntax Trees for the WHILE Language
import buildtools.compiler.AST._


// more convenience for the semantic actions later on
case class ~[+A, +B](_1: A, _2: B)

type IsSeq[A] = A => Seq[_]

abstract class Parser[I : IsSeq, T] {
  def parse(ts: I): Set[(T, I)]

def parse_all(ts: I) : Set[T] =
  for ((head, tail) <- parse(ts); if tail.isEmpty) yield head
}

class SeqParser[I : IsSeq, T, S](p: => Parser[I, T], q: => Parser[I, S]) extends P
  def parse(sb: I) =
    for ((head1, tail1) <- p.parse(sb);
    (head2, tail2) <- q.parse(tail1)) yield (new ~(head1, head2), tail2)
}

class AltParser[I : IsSeq, T](p: => Parser[I, T], q: => Parser[I, T]) extends Pars
  def parse(sb: I) = p.parse(sb) ++ q.parse(sb)
}

class FunParser[I : IsSeq, T, S](p: => Parser[I, T], f: T => S) extends Parser[I,
  def parse(sb: I) =
    for ((head, tail) <- p.parse(sb)) yield (f(head), tail)
```

```
}

case class Tok1Parser(s: String) extends Parser[List[(String, String)], (String, S
  def parse(toks: List[(String, String)]) = toks match{
    case (s, x)::tail => Set(((s, x), tail))
    case _ => Set()
  }
}

case class Tok2Parser(tok: (String, String)) extends Parser[List[(String, String)]
  def parse(toks: List[(String, String)]) = toks match{
    case (tok._1, tok._2)::tail => Set((tok, tail))
    case _ => Set()
  }
}

case object NumParser extends Parser[List[(String, String)], Int] {
  def parse(tks: List[(String, String)]) = tks match {
    case ("n", nr)::tail => Set((nr.toInt, tail))
    case _ => Set()
  }
}

case object IdParser extends Parser[List[(String, String)], String] {
  def parse(tks: List[(String, String)]) = tks match {
    case ("id", id)::tail => Set((id, tail))
    case _ => Set()
    }
}

implicit def string2parser(s : String) = Tok1Parser(s)

implicit def tuple2parser(tok: (String, String)) = Tok2Parser(tok)
```

```
implicit def ParserOps[I : IsSeq, T](p: Parser[I, T]) = new {
  def ||(q : => Parser[I, T]) = new AltParser[I, T](p, q)
  def ==>[S] (f: => T => S) = new FunParser[I, T, S](p, f)
  def ~[S](q : => Parser[I, S]) = new SeqParser[I, T, S](p, q)
}

implicit def StringOps(tok: (String, String)) = new {
  def ||(q : => Parser[List[(String, String)], (String, String)]) = new AltParser[
  def ||(r: String) = new AltParser[List[(String, String)], (String, String)](tok,
  def ==>[S] (f: => ((String, String)) => S) = new FunParser[List[(String, String)
  def ~[S](q : => Parser[List[(String, String)], S]) =
    new SeqParser[List[(String, String)], (String, String), S](tok, q)
  def ~(r: String) =
    new SeqParser[List[(String, String)], (String, String), (String, String)](tok,
}

// While Grammar definitions using Parser Combinators

// arithmetic expressions
lazy val AExp: Parser[List[(String, String)], AExp] =
  (Te ~ ("op", "+") ~ AExp) ==> { case x ~ _ ~ z => Aop("+", x, z): AExp } ||
  (Te ~ ("op", "-") ~ AExp) ==> { case x ~ _ ~ z => Aop("-", x, z): AExp } || Te
lazy val Te: Parser[List[(String, String)], AExp] =
  (Fa ~ ("op", "*") ~ Te) ==> { case x ~ _ ~ z => Aop("*", x, z): AExp } || Fa
lazy val Fa: Parser[List[(String, String)], AExp] =
  (("p", "(") ~ AExp ~ ("p", ")")) ==> { case _ ~ y ~ _ => y } ||
  IdParser ==> Var || NumParser ==> Num

// boolean expressions with some simple nesting
lazy val BExp: Parser[List[(String, String)], BExp] =
  (AExp ~ ("op", "==") ~ AExp) ==> { case x ~ _ ~ z => Bop("==", x, z): BExp } ||
  (AExp ~ ("op", "!=") ~ AExp) ==> { case x ~ _ ~ z => Bop("!=", x, z): BExp } ||
```

```
  (AExp ~ ("op", "<") ~ AExp) ==> { case x ~ _ ~ z => Bop("<", x, z): BExp } ||
  (AExp ~ ("op", ">") ~ AExp) ==> { case x ~ _ ~ z => Bop(">", x, z): BExp } ||
  (AExp ~ ("op", "<=") ~ AExp) ==> {case x ~ _ ~ z => Bop("<=", x, z): BExp} ||
  (AExp ~ ("op", ">=") ~ AExp) ==> {case x ~ _ ~ z => Bop(">=", x, z): BExp} ||
  (("k", "true") ==> (_ => True: BExp )) ||
  (("k", "false") ==> (_ => False: BExp )) ||
  (("p", "(") ~ BExp ~ ("p", ")")) ==> { case _ ~ x ~ _ => x }

// statement / statements
lazy val Stmt: Parser[List[(String, String)], Stmt] =
  ((("k", "skip") ==> (_ => Skip: Stmt)) ||
  (IdParser ~ ("op", ":=") ~ AExp) ==> { case x ~ _ ~ z => Assign(x, z): Stmt } ||
  (("k", "write") ~ AExp) ==> { case _ ~ y => WriteAExp(y): Stmt } ||
  (("k", "read") ~ IdParser) ==> { case _ ~ y => Read(y): Stmt } ||
  (("k", "if") ~ BExp ~ ("k", "then") ~ Block ~ ("k", "else") ~ Block) ==>
  { case _ ~ y ~ _ ~ u ~ _ ~ w => If(y, u, w): Stmt } ||
  (("k", "while") ~ BExp ~ ("k", "do") ~ Block) ==> { case _ ~ y ~ _ ~ w => While(

lazy val Stmts: Parser[List[(String, String)], Block] =
  (Stmt ~ ("s", ";") ~ Stmts) ==> { case x ~ _ ~ z => x :: z : Block } ||
  (Stmt ==> ( s => List(s) : Block))

// blocks (enclosed in curly braces)
lazy val Block: Parser[List[(String, String)], Block] =
  ((("p", "{") ~ Stmts ~ ("p", "}")) ==> { case _ ~ y ~ _ => y } ||
  (Stmt ==> (s => List(s) : Block)))

def parse(tks: List[(String, String)]) : Block =
    Stmts.parse_all(tks).head

}
```

3.5 Assembler and Compiler Script

3.5.1 script.scala

```
package buildtools.script

object CompAsm extends App{

  import buildtools.assembler.Assembler
  import buildtools.compiler.{Lexer, Parser, Compiler}



  import scala.io.Source
  import java.io.{BufferedOutputStream, FileOutputStream}
  import java.io.{PrintWriter, File}

  def write_bytes(bytes: List[Byte], filename: String) : Unit = {
      print("Writing bytes to file " + filename + " ... ")
      val stream = new BufferedOutputStream(new FileOutputStream(filename))
      stream.write(bytes.toArray)
      stream.close
      println("Done.")
  }

  def write_text(text: String, filename: String) : Unit = {
    print("Writing result to file " + filename + " ... ")
    val p = new PrintWriter(new File(filename))
    p.write(text)
    p.close()
    println("Done.")
  }

  def read_file(filename: String) : String = {
    print("Reading file " + filename + " ... ")
```

```
    val text = Source.fromFile(filename).mkString
    println("Done.")
    return text
  }


  def help() = {
    val s1 = "Compiler/Assembler for the 16-bit breadboard computer instrunction set\n
             "  Built by Luca-Dorin Anton. This software contributes towards a degree\
             "  Usage:\n" ++
             "    scala CompAsm [-a/-c/-d] [.asm/.while file] [isa_file] [result_file]
             "  Argument description:\n" ++
             "    -a/-c/-d: whether to assemble a binary or compile a while source to
             "    .asm/.while file: source assembly file. Should contain either some 1
             "    isa_file: YAML file containing the definitions of the instructions f
             "    result_file: where to store the resulting binaries. This file will b
             "    [Optional] dir: Directory from which to read and write files. Relati
    println(s1)
  }

  def time[R](block: => R): R = {
    val t0 = System.nanoTime()
    val result = block    // call-by-name
    val t1 = System.nanoTime()
    println("Elapsed time: " + (t1 - t0) + "ns, " + ((t1 - t0)/1000000) + "ms, " + (((
    result
  }

  def check_args(dir : String) : Unit = {
    import java.io.FileNotFoundException
    import java.io.IOException
    if(args.length < 4) {
      println("Insufficient arguments")
```

```
      help()
      System.exit(1)
    }
    val mode = args(0).toLowerCase
    if(mode != "-a" && mode != "-c" && mode != "-d") {
      println("Please specifiy -a/-c/-d")
      help()
    }
    for(i <- 1 to 2) {
      try{
        Source.fromFile(dir + args(i)).mkString
      } catch {
        case e: FileNotFoundException => println("Couldn't find file " + args(i) + "\n
        case e: IOException => println("IOException while trying to open " + args(i) +
        case e: Exception => println(e + "\n" + help()) ; System.exit(1)
      }
    }
  }
  def get_dir() : String = if (args.length > 4) args(4) + "/" else ""

  def main() : Unit = {
    val dir = get_dir()
    check_args(dir)
    val mode = args(0).toLowerCase
    val source = dir + args(1)
    val isa = dir + args(2)
    val target = dir + args(3)
    val source_text = read_file(source)
    val isa_text = read_file(isa)

    if (mode == "-a") {

      val bytes = Assembler.assemble(source_text, isa_text)
```

```
      write_bytes(bytes, target)
      println("Done! Assembly process successfull!")
    } else {

      print("Lexing source ...")
      val tks = Lexer.lex_while(source_text)
      println("Done.")

      print("Parsing tokens ...")
      val ast = Parser.parse(tks)
      println("Done.")

      print("Compiling abstract syntax tree ...")
      val asm = Compiler.compile(ast)
      println("Done.")

      if (mode == "-c") {
        write_text(asm, target)
      } else {
        val bytes = Assembler.assemble(asm, isa_text)
        write_bytes(bytes, target)
      }

      println("Done! Compiltation successfull!")
    }
  }

  time(main())
}
```

3.6 I/O Driver

3.6.1 io.ino

```
byte q1[10];  // low order byte queue
byte q2[10];  // high order byte queue
bool got1 = false;
int idx = 0;
byte byte_1;
byte byte_2;
int read_idx = 0;
// Low-order to High-order bits
const int data[] = {22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52};

#define DIR 49
#define E 2

// Write the front of the queue to the digital pins
void output_data() {
    for(int i = 0; i< 8; i++) {
    if(bitRead(q1[i], i)) {
      digitalWrite(data[i], HIGH);
    } else {
      digitalWrite(data[i], LOW);
    }

    if(bitRead(q2[i], i)) {
      digitalWrite(data[8 + i], HIGH);
    } else {
      digitalWrite(data[8 + 1], LOW);
    }
  }
}

// Read from the digital pins and built bytes. Send them over to the computer
void input_data() {
  byte b1 = 0;
```

```
  byte b2 = 0;
  for(int i = 0; i< 8; i++) {
    b1 = b1 | digitalRead(data[i]);
    b1 = b1 << 1;
    b2 = b2 | digitalRead(data[i + 8]);
    b2 = b2 << 1;
  }
  Serial.write(b1);
  Serial.write(b2);
}

// Interrupt Sub Routine for handling IO requests
void ISR_handle_io() {
  if(digitalRead(DIR)) {
    set_read_mode();
    output_data();
    idx = (idx + 1) % 10;
  } else {
    set_write_mode();
    input_data();
  }
}

// Write TO Arduino
void set_write_mode() {
    for (int i = 0; i < 16; i++) {
    pinMode(data[i], INPUT);
  }
}

// Read FROM Arduino
void set_read_mode() {
  for (int i = 0; i < 16; i++) {
```

```
    pinMode(data[i], OUTPUT);
  }
}

void setup() {

  Serial.begin(9600);
  pinMode(DIR, INPUT);
  pinMode(E, INPUT);
  set_write_mode();
  attachInterrupt(digitalPinToInterrupt(E), ISR_handle_io, RISING);



}

void loop() {
  if (Serial.available() > 0) {
    if (got1) {
      byte_2 = Serial.read();
      q1[read_idx] = byte_1;
      q2[read_idx] = byte_2;
      read_idx = (read_idx + 1) % 10;
      got1 = false;
      Serial.println(idx);
    } else {
      byte_1 = Serial.read();
      got1 = true;
    }
  }
}
```

3.6.2 io.py

```
import serial
```

```
import sys
import time



def start_io_loop(ser, data):
    print("Sending read queue ...", end="")
    for d in Data:
        ser.write(d)
    print("Done.")
    print("Starting write loop. Use Ctrl+C to exit.")
    while True:
        bytes = ser.read(2)
        if bytes:
            display_bytes(bytes)
        time.sleep(1)

def display_bytes(bytes):
    i = int.from_bytes(bytes, "big")
    print("Arduino -> %i" % i)

def get_queue(fn):


    with open(fn, "r") as f:
        data = f.readlines()
        if len(data) > 10:
            data = data[:10]
    return data

def process_data(data):
    res = []
    for d in data:
        d= d.strip()
        if d[0:2].lower() == "0x":
```

```
            res.append(int(d, 16))
        elif d[0:2].lower() == "0b":
            res.append(int(d, 2))
        elif d[0:2].lower() == "0d":
            res.append(int(d, 10))
        else:
            res.append(int(d))
    return res


def build_bytes(data):
    return [t.to_bytes(2, "big") for t in data]


def help():
    s = "I/O Script for the 16-bit BreadBoard computer.\n"
    s += "      Built by Luca-Dorin Anton. This project contributes to a degree.\n"
    s += "Usage: python3 io.py [COM] [queue_file]\n"
    s += "Argument description:\n"
    s += "      COM: COM port to which the Arduino is connected. Can be found using t
    s += "      queue_file: File containing hex/dec/bin numbers. Forms the read queue.
    print(s)


if __name__ == "__main__":
    if len(sys.argv) < 3:
        print("Insufficient Arguments")
        help()
        sys.exit(1)
    print("Initializing Arduino serial...")
    ardu_ser = serial.Serial(sys.argv[1], 9600, timeout=1)
    time.sleep(3)
    print("Done!")
    print("Reading queue...", end="")
    q = get_queue(sys.argv[2])
    d = process_data(q)
```

```
    b = build_bytes(d)
    print("Done.")
    start_io_loop(ardu_ser, b)
```

3.7 Programming Driver

3.7.1 prog.ino

```
byte byte_1 = 0;
byte byte_2 = 0;
bool got1 = false;
unsigned int idx = 0;
// Low-order to High-order bits
const int addr[] = {35, 37, 39, 41, 43, 45, 47, 49, 51 ,53};
const int data[] = {22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52};

#define WRITE 23

void setup() {

  Serial.begin(9600);

  pinMode(WRITE, OUTPUT);
  digitalWrite(WRITE, LOW);

  for (int i = 0; i < 10; i++) {
    pinMode(addr[i], OUTPUT);
  }

  for (int i = 0; i < 16; i++) {
    pinMode(data[i], OUTPUT);
  }

}
```

```
void loop() {
  if (Serial.available() > 0) {
    if (got1) {
      byte_2 = Serial.read();
      prog_instr();
      got1 = false;
      idx += 1;
      Serial.println(idx);
    } else {
      byte_1 = Serial.read();
      got1 = true;
    }
  }
}

void prog_instr() {
  // Setup address
  for(int i = 0; i < 10; i++) {
    if(bitRead(idx, i)) {
      digitalWrite(addr[i], HIGH);
    } else {
      digitalWrite(addr[i], LOW);
    }
  }

  delay(1);

  // Setup data word
  for(int i = 0; i< 8; i++) {
    if(bitRead(byte_1, i)) {
      digitalWrite(data[i], HIGH);
    } else {
```

```
      digitalWrite(data[i], LOW);
    }

    if(bitRead(byte_2, i)) {
      digitalWrite(data[i * 2 + 1], HIGH);
    } else {
      digitalWrite(data[i * 2 + 1], LOW);
    }
  }

  delay(1);
  // Pulse ARDU_PROG to program the data word at the address
  digitalWrite(WRITE, HIGH);

  delay(1);

  digitalWrite(WRITE, LOW);

}
```

3.7.2 prog.py

```
import serial
import sys
import time

def prog(ser, bytes):
    bytes = list(bytes)
    while len(bytes) != 0:
        byte_1 = bytes.pop()
        byte_2 = bytes.pop()

        bts = bytearray([byte_1, byte_2])
        ser.write(bts)
```

```
            ret = int(ser.readline().strip())
            if not ret:
                raise Exception("Error on programming instruction to Arduino! Aborting...'
            else:
                if ret == 0:
                    print("0%...", end="")
                if ret == 256:
                    print("25%...", end="")
                if ret == 512:
                    print("50%...", end="")
                if ret == 768:
                    print("75%..", end="")
                if ret == 1023:
                    print("100%")

def help():
    s = "Programmer Script for the 16-bit BreadBoard computer.\n"
    s += "      Built by Luca-Dorin Anton. This project contributes to a degree.\n"
    s += "Usage: python3 prog.py [COM] [binfile]\n"
    s += "Argument description:\n"
    s += "      COM: COM port to which the Arduino is connected. Can be found using t
    s += "      binfile: Binary file containing 16-bit BreadBoard computer executable'
    print(s)


if __name__ == "__main__":
    if len(sys.argv) < 3:
        print("Insufficient Arguments")
        help()
        sys.exit(1)
    bytes = open(sys.argv[2], "rb").read()
    print("Initializing Arduino serial...")
    ardu_ser = serial.Serial(sys.argv[1], 9600, timeout=1)
```

```
time.sleep(3)
print("Done!")
print("Writing instrutions ...")
prog(ardu_ser, bytes)
print("Done.")
```

References

[1] Laws David. Mosfet. https://computerhistory.org/blog/13-sextillion-counting-the-long-winding-road-to-the-most-frequently-manufactured-human-artifac?key=13-sextillion-counting-the-long-winding-road-to-the-most-frequently-manufactured-human-ar 2018. Accessed: 2019-12-8.