Assignment 5
XM23p Conditional Execution
Design, Implementation and Testing Document

Prepared for: Dr. Larry Hughes

Abdulla Sadoun B00900541

Table of Contents

Problem Introduction

Statement of Purpose

The purpose of this assignment is to design, implement and test Condition EXecution (CEX) instruction within our emulator program written in C, that would emulate the XM23p's execution of the instructions in charge of conditional execution.

This function serves a crucial purpose as they are deemed an valuable asset when it comes to writing .asm programs. They allow the programmer to test multiple fields in the PSW using the conditions and then based on the condition and the result (TRUE or FALSE) the programmer can choose up to 4 functions to execute per CEX function and its condition.

This is very important because without this function checking for multiple fields and using conditionals would be extremely expensive and unreliable as well as very hard to read, as it would mean that the programmer would have to use multiple branching functions which are expensive as they take up more space in the limited instruction memory space that we have.

CEX comes with 16 different conditions that can be used by the programmer to inspect different flags or multiple flags within the program status word/register (PSW). Below is a table of all the different conditions the programmer can set.

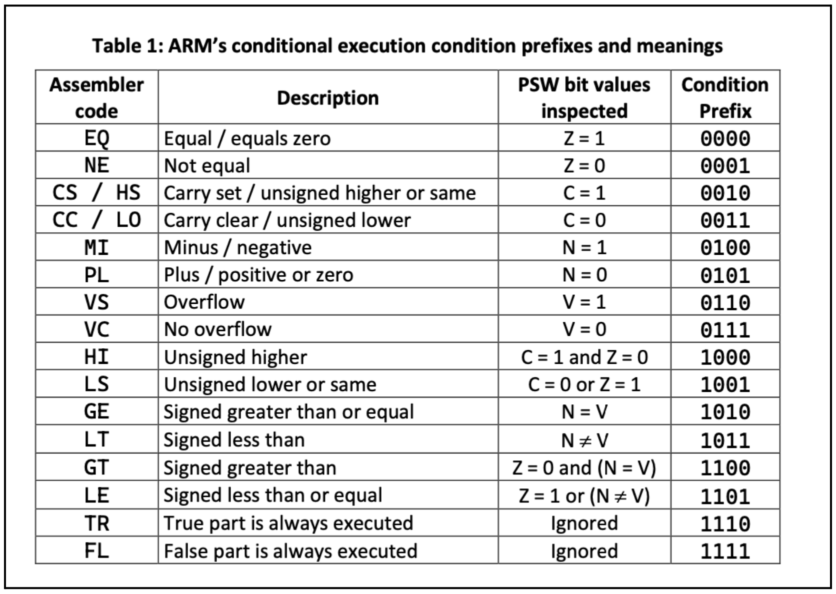


Table 1: ARM's conditional execution condition prefixes and meanings

Assembler code	Description	PSW bit values inspected	Condition Prefix
EQ	Equal / equals zero	Z = 1	0000
NE	Not equal	Z = 0	0001
CS / HS	Carry set / unsigned higher or same	C = 1	0010
CC / LO	Carry clear / unsigned lower	C = 0	0011
MI	Minus / negative	N = 1	0100
PL	Plus / positive or zero	N = 0	0101
VS	Overflow	V = 1	0110
VC	No overflow	V = 0	0111
HI	Unsigned higher	C = 1 and Z = 0	1000
LS	Unsigned lower or same	C = 0 or Z = 1	1001
GE	Signed greater than or equal	N = V	1010
LT	Signed less than	N ≠ V	1011
GT	Signed greater than	Z = 0 and (N = V)	1100
LE	Signed less than or equal	Z = 1 or (N ≠ V)	1101
TR	True part is always executed	Ignored	1110
FL	False part is always executed	Ignored	1111

The table highlights the operand for the condition prefix as well as its assembler code and the PSW bits it would inspect.

As the CEX function has 6 bits for its opcode, 4 bits reserved for the condition's prefix or code and it has 6 bits dedicated to deciding whether many of the subsequent functions are either instructions to be executed when the condition is true or instructions to be executed when the condition is false.

It can have up to 8 (000-111) True instructions to run (after the CEX) denoted as T; where TT would mean 2 true instructions. Or up to 8 (000-111) False instructions to run (after the CEX and the true instructions).

Here is the definition, description and format of this instruction from the XM23p ISA highlighting the opcode and operands within the 16 bits.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	C	C	C	C	T	T	T	F	F	F

The formatting when using the instruction in assembly can be shown in this sample:

CEX <COND>,<T-COUNT>,<F-COUNT>

Objectives

The objective of this assignment is to design, implement and test the conditional execution (CEX) instructions within the emulator. Their functionality will be implemented according to the description mentioned above.

In this assignment I will create the design for the conditional execution instructions as an extension to the loader, debugger, IMEM and DMEM instructions designed, implemented and tested within the emulator in the previous labs and assignments.

NOTE: Since this is the last assignment. I wanted to conclude the emulator by making significant changes to the code that would make it a lot more readable and presentable and efficient. After doing so I have also recreated the entire pseudo code for the emulator for other people to use when they are designing their own emulators. I have highlighted the parts that directly impact and are irrelevant to this assignment in red.

I have put the pseudo code for A5 relevant parts in Software Design and the rest is attached at the very end of the document in the "**Pseudo Code** for the rest of the emulator" section.

Software Design

Data Dictionary

s-record = 's' + type + length of record + Address + Data
Type = [0|1|2|9]
Length of record = Byte Pair
Address = 2[Byte Pair]2

Address = 0000-ffff
Data = 1[Byte Pair]30
Byte Pair = character + character
Character = [0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F]

Registers[RegisterNo][BitNo] = [General Purpose Registers|Special Purpose Registers],[Bit Number]
General Purpose Register = [R0|R1|R2|R3|R4]
Special Purpose Registers = [PC|SP|LR]
R0 = [‘0’| ‘000’]
R1 = [‘1’| ‘001’]
R2 = [‘2’| ‘010’]
R3 = [‘3’| ‘011’]
R4 = [‘4’| ‘100’]
LR = 5
SP = 6
PC = 7

Instruction = Opcode + Operand
Opcode = 4{bit}13
Bit = [0|1]

Operand = [RC|WB|Source|Destination|Byte]
RC = [Register|Constant]
Register = 0
Constant = 1

WB = [Word|byte]
Word = 2{byte}2
Byte = 8{bit}8

Source = [R0-R4] *in bits* = 000-100
Destination = [R0-R4] *int bits* = 000-100

DMEM Instructions = [LD|ST|LDR|STR]
LD = LD_Opcode_bits + PRPO_bit + DEC_bit + INC_bit + WB_bit + SRC + DST
LD_Opcode_bits = “0”+“1”+“0”+”1”+”1”+”0”
ST = ST_Opcode_bits + PRPO_bit + DEC_bit + INC_bit + WB_bit + SRC + DST
ST_Opcode_bits = “0”+“1”+“0”+”1”+”1”+”1”
PRPO_bit = [0|1] # indicates whether pre or post increment/decrement action
DEC_bit = [0|1] # 0=no decrement pre/post 1=decrement pre/post
INC_bit = [0|1] # 0=no increment pre/post 1=increment pre/post
WB_bit = [0|1] # indicates whether data used is a word or a byte
SRC = [R0-R4] *in bits* = 000-100
DST = [R0-R4] *int bits* = 000-100

LDR = LDR_Opcode_bits + OFF_bits +WB_bit + SRC + DST
LDR_Opcode_bits = “1”+“0”
STR = STR_Opcode_bits + OFF_bits + WB_bit + SRC + DST
STR_Opcode_bits = “1”+“1”
OFF_bits = 7{bit}7
Bit = [0|1]
WB_bit = [0|1] # indicates whether data used is a word or a byte
SRC = [R0-R4] *in bits* = 000-100 # indicates number of source register
DST = [R0-R4] *int bits* = 000-100 # indicates number of destination register

BRANCHING INSTRUCTIONS = [BL|BEQ/BZ|BNE/BNZ|BC/BHS|BNC/BLO|BN|BGE|BLT|BRA]

BL = BL_Opcode + OFF_bits
BL_Opcode = “0” + “0” + “0”
OFF_bits = 13{bit}13

Bit = [0|1]

BEQ/BZ = BEQ/BZ_Opcode + OFF_bits
BEQ/BZ_Opcode = “0” + “0” + “1” + “0” + “0” + “0”
OFF_bits = 10{bit}10
Bit = [0|1]

BEQ/BZ = BEQ/BZ_Opcode + OFF_bits
BEQ/BZ_Opcode = “0” + “0” + “1” + “0” + “0” + “1”
OFF_bits = 10{bit}10
Bit = [0|1]

BC/BHS = BC/BHS_Opcode + OFF_bits
BC/BHS_Opcode = “0” + “0” + “1” + “0” + “1” + “0”
OFF_bits = 10{bit}10
Bit = [0|1]

BNC/BLO = BNC/BLO_Opcode + OFF_bits
BNC/BLO_Opcode = “0” + “0” + “1” + “0” + “1” + “1”
OFF_bits = 10{bit}10
Bit = [0|1]

BN = BN_Opcode + OFF_bits
BN_Opcode = “0” + “0” + “1” + “1” + “0” + “0”
OFF_bits = 10{bit}10
Bit = [0|1]

BGE = BGE_Opcode + OFF_bits
BGE_Opcode = “0” + “0” + “1” + “1” + “0” + “1”
OFF_bits = 10{bit}10
Bit = [0|1]

BLT = BLT_Opcode + OFF_bits
BLT_Opcode = “0” + “0” + “1” + “1” + “1” + “0”
OFF_bits = 10{bit}10
Bit = [0|1]

BRA = BRA_Opcode + OFF_bits
BRA_Opcode = “0” + “0” + “1” + “1” + “1” + “1”
OFF_bits = 10{bit}10
Bit = [0|1]

CEX = CEX_Opcode + CEX_OPERANDS
CEX_Opcode = “0” + “1” + “0” + “1” + “0” + “0”
CEX_OPERANDS = CONDITION_PREFIX + TRUE_INSTR_BITS + FALSE_INSTR_BITS
CONDITION_PREFIX = 1{[EQ|NE|CS/HS|CC/LO|MI|PL|VS|VC|HI|LS|GE|LT|GT|LE|TR|FL]}1
EQ = “0” + “0” + “0” + “0” # 0 in binary
NE = “0” + “0” + “0” + “1” # 1 in binary
CS / HS = “0” + “0” + “1” + “0” # 2 in binary
CC / LO = “0” + “0” + “1” + “1” # 3 in binary
MI = “0” + “1” + “0” + “0”# 4 in binary
PL = “0” + “1” + “0” + “1” # 5 in binary
VS = “0” + “1” + “1” + “0” # 6 in binary
VC = “0” + “1” + “1” + “1” # 7 in binary
HI = “1” + “0” + “0” + “0” # 8 in binary

LS = “1” + “0” + “0” + “1” # 9 in binary
GE = “1” + “0” + “1” + “0” # 10 in binary
LT = “1” + “0” + “1” + “1”   # 11 in binary
GT = “1” + “1” + “0” + “0”   # 12 in binary
LE = “1” + “1” + “0” + “1” # 13 in binary
TR = “1” + “1” + “1” + “0” # 14 in binary
FL = “1” + “1” + “1” + “1” # 15 in binary

TRUE_INSTR_BITS = 3{bit}3 # indicating number of TRUE instructions to execute from cex
TRUE_INSTR_BITS = 1{0-8}1 *in binary*
TRUE_INSTR_BITS = 1{000-111} *in bits*

FALSE_INSTR_BITS = 3{bit}3 # indicating number of FALSE instructions to execute from cex and after TRUE_INSTR_BITS are skipped/executed
FALSE_INSTR_BITS = 1{0-8}1 *in binary*
FALSE = 1{000-111} *in bits*

Bit = 1{[0|1]}1

Pseudo Code:

Header file:

- Include standard libraries

Global Variables:
- Char array: IMEM[MEMORY_SIZE] # THIS HAS BEEN CHANGED
- Char array: DMEM[MEMORY_SIZE] # THIS HAS BEEN CHANGED
- unsigned short array: IMEM[MEMORY_SIZE/2]; THIS IS NEW
- unsigned short array: DMEM[MEMORY_SIZE/2]; THIS IS NEW
- Char program counter (int): PC
- Define global int starting address (int): Start_Address

Registers:
- Define Registers as 2D array: Char RegistersBinary[8][16]
- Define Registers values as an array: Char RegisterValue[8]
  # Where 8 is the number of registers and 16 is the amount of bits.

Internal instruction numbering system enum:
- ENUM for instructions: {BL, BEQBZ, BNEBNZ, BCBHS, BNCBLO, BN, BGE, BLT, BRA, ADD, ADDC, SUB, SUBC, DADD, CMP, XOR, AND, OR, BIT, BIC, BIS, MOV, SWAP, SRA, RRC, SWPB, SXT, SETPRI, SVC, SETCC, CLRCC, CEX, LD, ST, MOVL, MOVLZ, MOVLS, MOVH, LDR, STR, Error}

Time Counter:
- Int timecount <- 0

Implementation: **run.c** (updated for this assignment's content)

- Include Header file

```
DECLARE timecounter AS unsigned short // Time counter

FUNCTION run():
    Call fetch() // Fetches the instruction
    PRINT "IMARValue:%04X @PC=%04X \n", IMARValue, RegistersValue[PC] // Debug printf

    WHILE NOT (IMARValue == 0x0000 OR RegistersValue[PC] == BreakpointValue):
        Call fetch()
        PRINT "IMARValue:%04X @PC=%04X\t", IMARValue, RegistersValue[PC] // Debug printf

        RegistersValue[PC] <- RegistersValue[PC] + 2
        Call ChangedRegistersValue(RegistersValue[PC], PC)

        timecounter <- timecounter + 1

        DECLARE instructionnumber AS integer <- Call decode()
        timecounter <- timecounter + 1

        IF (TC == 0 AND FC == 0) OR instructionnumber == CEX THEN: // CEX CHECK
            Call execute(instructionnumber) // Executes current instruction (CEX off)
        ELSE: // CEX enabled
            Call cex_enabled(instructionnumber) // Handle executions accordingly
        END IF

        Call fetch() // Fetches next instruction
        timecounter <- timecounter + 1
    END WHILE

    PRINT "Breakpoint reached or the end of exec.\n"

END FUNCTION

FUNCTION step():
    Call fetch()
    PRINT "IMARValue:%04X @PC=%04X\t", IMARValue, RegistersValue[PC] // Debug printf
    IF IMARValue == 0x0000 OR RegistersValue[PC] == BreakpointValue THEN:
        PRINT "Breakpoint reached or the end of exec.\n"
        RETURN
    ELSE:
        Call fetch()

        RegistersValue[PC] <- RegistersValue[PC] + 2
        Call ChangedRegistersValue(RegistersValue[PC], PC)

        DECLARE instructionnumber AS integer <- Call decode()

        IF (TC == 0 AND FC == 0) OR instructionnumber == CEX THEN: // CEX disabled
            Call execute(instructionnumber) // Executes current instruction (CEX off)
        ELSE: // CEX enabled
            Call cex_enabled(instructionnumber) // Handle executions accordingly
        END IF

        RETURN
    END IF

END FUNCTION
```

```
FUNCTION step_x_times():
    DECLARE x AS integer
    PRINT "Enter the number of steps: "
    SCAN("%d", &x)

    FOR i <- 0 TO x - 1:
        Call step()
    END FOR

    RETURN
END FUNCTION
```

Implementation: **decode.c** (updated to for this assignment's content)

- Include Header file

```
DECLARE BreakpointValue AS unsigned short // Breakpoint value (global variable)

DEFINE LTCASE <- 0x7
DEFINE STCASE <- 0x6
DEFINE SUB_LD <- 0x0
DEFINE MOV_CLRCC <- 0x3
DEFINE LCEX <- 0x4
DEFINE OFFBIT <- 6
```

```
FUNCTION decode():

   ENUM {BLCase, BEQtoBRA, ADDtoST, MOVLtoMOVH, LLDR1, LLDR2, LSTR1, LSTR2} //
Grouped into shared first 3 bits

   DECLARE opcode AS unsigned short <- (IMARValue >> 13) & 0x07 // Get first 3 bits of the
instruction
   offsetbuff <- (IMARValue >> 7) & 0x7f // Get the offset bit (LDR-STR)

   SWITCH opcode: // First 3 bits cases
      CASE BLCase:
         RETURN Call BLdecode() // BL internal number sent to execute
      CASE BEQtoBRA:
         RETURN Call betweenBEQandBRA(IMARValue) // Go to further filtering
      CASE ADDtoST: // CEX IS BETWEEN ADD-ST
         RETURN Call betweenADDandST(IMARValue) // Go to further filtering
      CASE MOVLtoMOVH:
         RETURN Call betweenMOVLandMOVH(IMARValue) // Go to further filtering
      CASE LLDR1:
         RETURN Call LDRdecode() // LDR internal number sent to execute
      CASE LLDR2:
         RETURN Call LDRdecode() // LDR internal number sent to execute
      CASE LSTR1:
         RETURN Call STRdecode() // STR internal number sent to execute
      CASE LSTR2:
         RETURN Call STRdecode() // STR internal number sent to execute
      DEFAULT:
         PRINT "Error - instruction not yet implemented"
         RETURN Error
   END SWITCH
END FUNCTION

FUNCTION BLdecode():
   offsetbuff <- IMARValue & 0x1FFF // Get the offset
   offsetbuff <- SignExt(offsetbuff, BIT_NUMBER_12) // Sign-extend the offset buffer

   RETURN BL // Return the internal instruction number
END FUNCTION

FUNCTION betweenADDandST(IMARValue AS unsigned short):

   DECLARE opcode AS unsigned short <- (IMARValue >> 10) & 0x07 // Get L2 opcode
   prpobuff <- (IMARValue >> 9) & 0x01 // Get the PRPO bits
   decbuff <- (IMARValue >> 8) & 0x01 // Get the decrement bit
   incbuff <- (IMARValue >> 7) & 0x01 // Get increment bit
```

```
wbbuff <- (IMARValue >> 6) & 0x01 // Get the word/byte bit
srcbuff <- (IMARValue >> 3) & 0x07 // Get the source register
dstbuff <- IMARValue & 0x07 // Get the destination register

IF opcode == LTCASE OR opcode == STCASE THEN: // LD or ST case
    prpobuff <- (IMARValue >> 9) & 0x01 // Get the PRPO bits
    decbuff <- (IMARValue >> 8) & 0x01 // Get the decrement bit
    incbuff <- (IMARValue >> 7) & 0x01 // Get increment bit
    wbbuff <- (IMARValue >> 6) & 0x01 // Get the word/byte bit
    srcbuff <- (IMARValue >> 3) & 0x07 // Get the source register
    dstbuff <- IMARValue & 0x07 // Get the destination register

    IF opcode == 6 THEN: // LD case
        RETURN LD
    ELSE IF opcode == 7 THEN: // ST case
        RETURN ST
    ELSE:
        PRINT "instruction not yet implemented"
        RETURN Error
    END IF

ELSE IF opcode == MOV_CLRCC THEN: // MOV to CLRCC case
    // ENUM for local opcode cases
    ENUM {LMOV, LSWAP, LSRALSXT, LSETPRILCLRCC, LSVC, LSETCC, LCLRCC}

    opcode <- (IMARValue >> 7) & 0x07 // Get L3 opcode
    vbuff <- (IMARValue >> 4) & 0x01 // Get the overflow bit
    nbuff <- (IMARValue >> 2) & 0x01 // Get the negative bit
    zbuff <- (IMARValue >> 1) & 0x01 // Get the zero bit
    cbuff <- IMARValue & 0x01 // Get the carry bit

    SWITCH opcode: // L3 opcode cases
        CASE LMOV:
            RETURN MOV
        CASE LSWAP:
            RETURN SWAP
        CASE LSRALSXT:
            opcode <- (IMARValue >> 3) & 0x07 // Get the L4 opcode
            // ENUM for local L4 opcode cases
            ENUM {LSRA, LRRC, LSWPB, LSXT}
            SWITCH opcode: // L4 opcode cases
                CASE LSRA:
                    RETURN SRA
                CASE LRRC:
                    RETURN RRC
                CASE LSWPB:
                    RETURN SWPB
                CASE LSXT:
                    RETURN SXT
                DEFAULT:
                    PRINT "instruction not yet implemented"
                    RETURN Error
            END SWITCH
        CASE LSETPRILCLRCC:
            opcode <- (IMARValue >> 4) & 0xf
            IF opcode == 8 THEN:
                RETURN SETPRI
            ELSE IF opcode == 9 THEN:
                RETURN SVC
            ELSE IF opcode == 10 OR opcode == 11 THEN:
                RETURN SETCC
            ELSE IF opcode == 12 OR opcode <= 13 THEN:
                RETURN CLRCC
            ELSE:
                PRINT "instruction not yet implemented"
                RETURN Error
            END IF
        DEFAULT:
            PRINT "instruction not yet implemented"
            RETURN Error
    END SWITCH

ELSE IF opcode == LCEX THEN: // CEX case
    // Extracting operands
    condition_prefix_buff <- (IMARValue >> 6) & 0x0F // Get condition prefix
    tcountbuff <- (IMARValue >> 3) & 0x07 // Get true bits
    fcountbuff <- IMARValue & 0x07 // Get false bits
    RETURN CEX

ELSE: // ADD to BIS case
    rcbuff <- (IMARValue >> 7) & 0x01 // Get the register constant bit
    wbbuff <- (IMARValue >> 6) & 0x01 // Get the word/byte bit
    srcbuff <- (IMARValue >> 3) & 0x07 // Get the source register
    dstbuff <- IMARValue & 0x07 // Get the destination register

    opcode <- (IMARValue >> 8) & 0x0F // Get the L3 opcode
    // ENUM for local L3 opcode cases
    ENUM {LADD, LADDC, LSUB, LSUBC, LDADD, LCMP, LXOR, LAND, LOR, LBIT, LBIC,
LBIS}

    SWITCH opcode: // Opcode cases
      CASE LADD:
        RETURN ADD
      CASE LADDC:
        RETURN ADDC
      CASE LSUB:
        RETURN SUB
      CASE LSUBC:
        RETURN SUBC
      CASE LDADD:
        RETURN DADD
      CASE LCMP:
        RETURN CMP
      CASE LXOR:
        RETURN XOR
      CASE LAND:
        RETURN AND
      CASE LOR:
        RETURN OR
      CASE LBIT:
        RETURN BIT
      CASE LBIC:
        RETURN BIC
      CASE LBIS:
        RETURN BIS
      DEFAULT:
        PRINT "instruction not yet implemented"
        RETURN Error
    END SWITCH
  END IF
END FUNCTION
```

```
FUNCTION betweenMOVLandMOVH(IMARValue AS unsigned short):
  DECLARE opcode AS unsigned short <- (IMARValue >> 11) & 0x03 // Get the Layer2 opcode
  dstbuff <- IMARValue & 0x07 // Get the destination register
  bitsbuff <- (IMARValue >> 3) & 0xFF // Get the bits

  ENUM {LMOVL, LMOVLZ, LMOVLS, LMOVH} // Local L2 opcode cases

  SWITCH opcode: // Opcode cases
    CASE LMOVL:
            RETURN MOVL
        CASE LMOVLZ:
            RETURN MOVLZ
        CASE LMOVLS:
            RETURN MOVLS
        CASE LMOVH:
            RETURN MOVH
        DEFAULT:
            PRINT "instruction not yet implemented"
            RETURN Error
    END SWITCH
END FUNCTION

FUNCTION betweenBEQandBRA(IMARValue AS unsigned short):
    DECLARE opcode AS unsigned short <- (IMARValue >> 10) & 0x07 // Add a shift to get layer 2 opcode
    offsetbuff <- IMARValue & 0x03FF // Get the offset
    offsetbuff <- SignExt(offsetbuff, BIT_NUMBER_9) // Sign-extend the offset buffer (was 9 bits)

    ENUM {LBEQBZ, LBNEBNZ, LBCBHS, LBNCBLO, LBN, LBGE, LBLT, LBRA} // Local L2 opcode cases

    SWITCH opcode: // Opcode cases
        CASE LBEQBZ:
            RETURN BEQBZ
        CASE LBNEBNZ:
            RETURN BNEBNZ
        CASE LBCBHS:
            RETURN BCBHS
        CASE LBNCBLO:
            RETURN BNCBLO
        CASE LBN:
            RETURN BN
        CASE LBGE:
            RETURN BGE
        CASE LBLT:
            RETURN BLT
        CASE LBRA:
            RETURN BRA
        DEFAULT:
            PRINT "instruction not yet implemented"
            RETURN Error
    END SWITCH

END FUNCTION

FUNCTION LDRdecode():
    wbbuff <- (IMARValue >> 6) & 0x01 // Get the word/byte bit
    srcbuff <- (IMARValue >> 3) & 0x07 // Get the source register
    dstbuff <- IMARValue & 0x07 // Get the destination register
    offsetbuff <- (IMARValue >> 7) & 0x7F // Get the offset

    offsetbuff <- SignExt(offsetbuff, BIT_NUMBER_6)
    RETURN LDR

END FUNCTION

FUNCTION STRdecode():
    wbbuff <- (IMARValue >> 6) & 0x01 // Get the word/byte bit
    srcbuff <- (IMARValue >> 3) & 0x07 // Get the source register
    dstbuff <- IMARValue & 0x07 // Get the destination register
    offsetbuff <- (IMARValue >> 7) & 0x7F // Get the offset
    offsetbuff <- SignExt(offsetbuff, BIT_NUMBER_6)
    RETURN STR
END FUNCTION

FUNCTION SignExt(offset AS short, msb AS integer) -> short:
    // Sign extension function to handle sign extensions
    IF (offset >> msb) & 0x01 THEN:
        offset <- offset | ((0xFFFF) << msb) // Extend the sign if the msb is set
    END IF
    RETURN offset << 1
END FUNCTION
```

Implementation: **execute.c** (updated for A5 and increased Readability)

- Include Header file

```
FUNCTION execute(instructionnumber AS integer):
    DECLARE Temp_Destination, Temp_src_buffer AS unsigned short // Temporary destination variable for swapping/replacing etc.

    IF instructionnumber >= ADD AND instructionnumber <= BIS AND rcbuff == SET AND instructionnumber != CMP THEN: // Constant is taken from the register
        Temp_src_buffer <- RegistersValue[srcbuff] // Set the source buffer to the temporary source buffer
        RegistersValue[srcbuff] <- Call handleConstant(srcbuff) // Handle the constant value
    END IF
    SWITCH instructionnumber: // Opcode cases
        CASE BL: // Branch and link
            PRINT "BL: offset:" + offsetbuff // Debug printf
            Call bl()
        CASE BEQBZ: // Branch if equal
            PRINT "BEQ/BZ: offset:" + offsetbuff // Debug printf
            Call beqbz()
        CASE BNEBNZ: // Branch if not equal
            PRINT "BNE/BNZ: offset:" + offsetbuff // Debug printf
            Call bnebnz()
        CASE BCBHS: // Branch if carry
            PRINT "BC/BHS: offset:" + offsetbuff // Debug printf
            Call bcbhs()
        CASE BNCBLO:  // Branch if not carry
            PRINT "BNV/BLO: offset:" + offsetbuff // Debug printf
            Call bncblo()
        CASE BN: // Branch if negative
            PRINT "BN: offset:" + offsetbuff // Debug printf
            Call bn()
        CASE BGE: // Branch if greater than or equal
            PRINT "BGE: offset:" + offsetbuff // Debug printf
            Call bge()
        CASE BLT: // Branch if less than
            PRINT "BLT: offset:" + offsetbuff // Debug printf
            Call blt()
        CASE BRA: // Branch always
            PRINT "BRA: offset:" + offsetbuff // Debug printf
            Call bra()
        CASE ADD: // Add registers
            PRINT "ADD: RC=" + rcbuff + ", WB=" + wbbuff + ", SRC=" + srcbuff + ", DST=" + dstbuff // Debug printf
            Call add() // DST = DST + SRC/CON
        CASE ADDC: // Add registers with carry
            PRINT "ADDC: RC=" + rcbuff + ", WB=" + wbbuff + ", SRC=" + srcbuff + ", DST=" + dstbuff // Debug printf
            Call addc() // DST = DST + (SRC/CON + Carry)
        CASE SUB: // Subtract registers
            PRINT "SUB: RC=" + rcbuff + ", WB=" + wbbuff + ", SRC=" + srcbuff + ", DST=" + dstbuff // Debug printf
            Call sub() // DST = DST + (-SRC/CON + 1)
        CASE SUBC: // Subtract registers with carry
            PRINT "SUBC: RC=" + rcbuff + ", WB=" + wbbuff + ", SRC=" + srcbuff + ", DST=" + dstbuff // Debug printf
            Call subc() // DST = DST + (-SRC/CON + 1 + Carry)
        CASE DADD: // Add registers (decimals)
            PRINT "DADD: RC=" + rcbuff + ", WB=" + wbbuff + ", SRC=" + srcbuff + ", DST=" + dstbuff // Debug printf
            Call dadd() // DST = DST + SRC/CON
        CASE CMP: // Compare registers
            PRINT "CMP: RC=" + rcbuff + ", WB=" + wbbuff + ", SRC=" + srcbuff + ", DST=" + dstbuff // Debug printf
            Call cmp() // DST - SRC/CON
        CASE XOR: // Exclusive OR registers
            PRINT "XOR: RC=" + rcbuff + ", WB=" + wbbuff + ", SRC=" + srcbuff + ", DST=" + dstbuff // Debug printf
            Call xor() // DST = DST ^ SRC/CON
        CASE AND: // AND registers
            PRINT "AND: RC=" + rcbuff + ", WB=" + wbbuff + ", SRC=" + srcbuff + ", DST=" + dstbuff // Debug printf
            Call and() // DST = DST & SRC/CON
        CASE OR: // OR registers
            PRINT "OR: RC=" + rcbuff + ", WB=" + wbbuff + ", SRC=" + srcbuff + ", DST=" + dstbuff // Debug printf
            Call or() // DST = DST | SRC/CON
        CASE BIT: // BIT test
            PRINT "BIT: RC=" + rcbuff + ", WB=" + wbbuff + ", SRC=" + srcbuff + ", DST=" + dstbuff // Debug printf
            Call bit() // DST & SRC/CON
        CASE BIC: // Bit clear
            PRINT "BIC: RC=" + rcbuff + ", WB=" + wbbuff + ", SRC=" + srcbuff + ", DST=" + dstbuff // Debug printf
            Call bic() // DST = DST & ~(SRC/CON)
        CASE BIS: // Bit set
            PRINT "BIS: RC=" + rcbuff + ", WB=" + wbbuff + ", SRC=" + srcbuff + ", DST=" + dstbuff // Debug printf
            Call bis() // DST = DST | (1 << SRC/CON)
        CASE MOV: // Move registers
            PRINT "MOV: WB=" + wbbuff + ", SRC=" + srcbuff + ", DST=" + dstbuff // Debug printf (no need for r/c)
            Call mov() // DST = SRC/CON
        CASE SWAP: // Swap registers
            PRINT "SWAP: SRC=" + srcbuff + ", DST=" + dstbuff // Debug printf
            Call swap() // DST = SRC
        CASE SRA: // Shift right arithmetic
            PRINT "SRA: WB=" + wbbuff + ", DST=" + dstbuff
            Call sra() // DST = DST >> 1
        CASE RRC: // Rotate right through carry
            PRINT "RRC: WB=" + wbbuff + ", DST=" + dstbuff
            Call rrc() // DST = DST >> 1 + Carry
        CASE SWPB: // Swap bytes
            PRINT "SWPB: DST=" + dstbuff + " Result: " + RegistersValue[dstbuff]
            Call swpb() // DST = ((DST & 0xFF) << 8) | ((DST >> 8) & 0xFF)
        CASE SXT: // Sign extend
            Call sxt() // DST = (DST & 0x80) ? 0xFF00 | DST : DST
            PRINT "SXT: DST=" + dstbuff + " Result: " + RegistersValue[dstbuff]
        CASE SETPRI: // Not yet implemented
            PRINT "SETPRI: "
        CASE SVC: // Not yet implemented
            PRINT "SVC: "
        CASE SETCC: // Set PSW flags
            PRINT "SETCC: "
            Call setcc() // Set the PSW flags
        CASE CLRCC: // Clear PSW flags
            PRINT "CLRCC: "
            Call clrcc() // Clear the PSW flags
            CASE CEX: // Conditional execution
                Call cex() // Set the CEX condition
                PRINT "CEX: Cond:" + condition_prefix_buff + " TC:" + tcountbuff + " FC:" + fcountbuff
                PRINT cex_condition ? "TRUE" : "FALSE"
        CASE LD: // Load content to register
            PRINT "LD: PRPO:" + prpobuff + " DEC:" + decbuff + " INC:" + incbuff + " WB:" + wbbuff + " SRC:" + srcbuff + " DST:" +
    dstbuff // Debug printf
            Call ld() // DST = DMEM[SRC plus addressing]
        CASE ST: // Store content from register to DMEM
            PRINT "ST: PRPO:" + prpobuff + " DEC:" + decbuff + " INC:" + incbuff + " WB:" + wbbuff + " SRC:" + srcbuff + " DST:" +
    dstbuff // Debug printf
            Call st() // DMEM[DST plus addressing] = SRC
        CASE MOVL: // Move low bits
            PRINT "MOVL: dst:" + dstbuff + " bits:" + bitsbuff // Debug printf
            Call movl() // DST = SRC/CON
        CASE MOVLZ: // Move low bits and zero rest
            PRINT "MOVLZ: dst:" + dstbuff + " bits:" + bitsbuff
            Call movlz() // DST = SRC/CON
        CASE MOVLS: // Move low bits and set high
            PRINT "MOVLS: dst:" + dstbuff + " bits:" + bitsbuff
            Call movls() // DST = SRC/CON
        CASE MOVH: //
        // Move high bits
            PRINT "MOVH: dst:" + dstbuff + " bits:" + bitsbuff
            Call movh() // DST = SRC/CON
        CASE LDR: // Load from memory to register with offset
            PRINT "LDR: offset:" + offsetbuff + " wb:" + wbbuff + " src:" + srcbuff + " dst:" + dstbuff // Debug printf
            Call ldr() // DST = mem[SRC + sign-extended 7-bit offset]
        CASE STR: // Store from register to memory with offset
            PRINT "STR: offset:" + offsetbuff + " wb:" + wbbuff + " src:" + srcbuff + " dst:" + dstbuff // Debug printf
            Call str() // mem[DST + sign-extended 7-bit offset] = SRC
        CASE Error: // Error
            PRINT "Error: " // Return error message
        DEFAULT:
            PRINT "Instruction execution code not recognized"
      END SWITCH

    IF instructionnumber >= ADD AND instructionnumber <= BIS AND rcbuff == SET AND
                instructionnumber != CMP THEN:
        RegistersValue[srcbuff] <- Temp_src_buffer // Set the source buffer to the temporary source buffer after constant was used
    END IF

    FOR i <- 0 TO 7 DO: // Update register binary content
                Call ChangedRegistersValue(RegistersValue[i], i)
    END FOR

      RETURN
END FUNCTION

FUNCTION handleConstant(src AS integer) -> unsigned short:
    SWITCH src:
        CASE R0:
            RETURN 0
        CASE R1:
            RETURN 1
        CASE R2:
            RETURN 2
        CASE R3:
            RETURN 4
        CASE R4:
            RETURN 8
        CASE R5:
            RETURN 16
        CASE R6:
            RETURN 32
        CASE R7:
            RETURN -1
        DEFAULT:
            PRINT "Error: Invalid register/constant value"
            RETURN 0
    END SWITCH
END FUNCTION
```

Implementation: **cex.c** (NEW)

- Include Header File

```
FUNCTION Cex:
        ENUM {EQ, NE, CSHS, CCLO, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE, TR, FL}

        SWITCH (condition_prefix_buff)
           CASE EQ:
              cex_condition = (PSW.z == SET) ? TRUE : FALSE
           CASE NE:
              cex_condition = (PSW.z == CLEAR) ? TRUE : FALSE
           CASE CSHS:
              cex_condition = (PSW.c == SET) ? TRUE : FALSE
           CASE CCLO:
              cex_condition = (PSW.c == CLEAR) ? TRUE : FALSE
           CASE MI:
              cex_condition = (PSW.n == SET) ? TRUE : FALSE
           CASE PL:
              cex_condition = (PSW.n == CLEAR) ? TRUE : FALSE
           CASE VS:
              cex_condition = (PSW.v == SET) ? TRUE : FALSE
           CASE VC:
              cex_condition = (PSW.v == CLEAR) ? TRUE : FALSE
           CASE HI:
              cex_condition = (PSW.c == SET && PSW.z == CLEAR) ? TRUE : FALSE
           CASE LS:
              cex_condition = (PSW.c == CLEAR || PSW.z == SET) ? TRUE : FALSE
           CASE GE:
              cex_condition = (PSW.n == PSW.v) ? TRUE : FALSE
           CASE LT:
              cex_condition = (PSW.n != PSW.v) ? TRUE : FALSE
           CASE GT:
              cex_condition = (PSW.z == CLEAR && PSW.n == PSW.v) ? TRUE : FALSE
           CASE LE:
              cex_condition = (PSW.z == SET || PSW.n != PSW.v) ? TRUE : FALSE
           CASE TR:
              cex_condition = TRUE
           CASE FL:
              cex_condition = FALSE
           DEFAULT:
              PRINT "Error: Invalid condition prefix"
        END SWITCH

        TC <- tcountbuff
        FC <- fcountbuff
END FUNCTION

FUNCTION `cex_enabled`(internal instruction number):
        IF cex_condition == TRUE:
           IF TC > 0:
              EXECUTE(internal instruction number)
              TC--
           ELSE:
              PRINT "CEX skip"
              FC--
        ELSE:
           IF TC > 0:
              PRINT "CEX skip"
              TC--
           ELSE:
              EXECUTE(internal instruction number)
              FC--
        END IF

        RETURN
END FUNCTION

FUNCTION `cex_enabled`(internal instruction number):
        IF cex_condition == TRUE:
           IF TC > 0:
              EXECUTE(internal instruction number)
              IF internal instruction number == BRA
                 reset_CEX()
                 RETRUN
              ELSE
                 TC--
              END IF
           ELSE:
              PRINT "CEX skip"
              FC--
        ELSE:
           IF TC > 0:
              PRINT "CEX skip"
              TC--
           ELSE:
              EXECUTE(internal instruction number)
                 IF internal instruction number == BRA
```

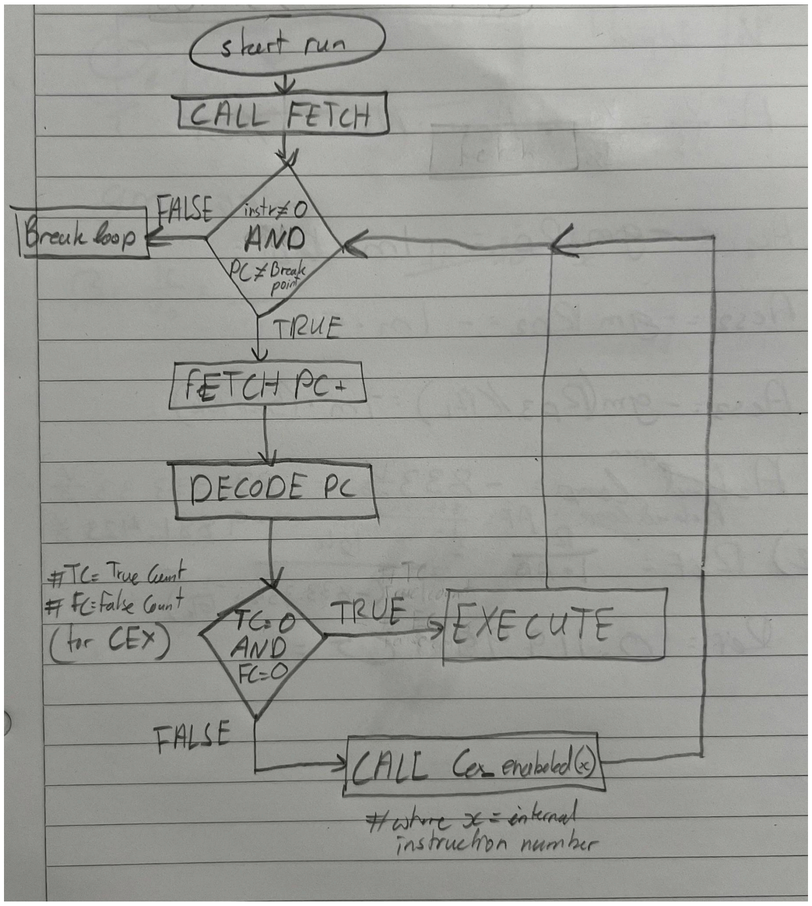
```
                reset_CEX()
                RETRUN
            ELSE
                FC--
            END IF
    END IF

    RETURN
END FUNCTION

FUNCTION `cex_enabled`
    TC = 0
    FC = 0
    RETURN
END FUNCTION
```

Other Design Content Used for A5:

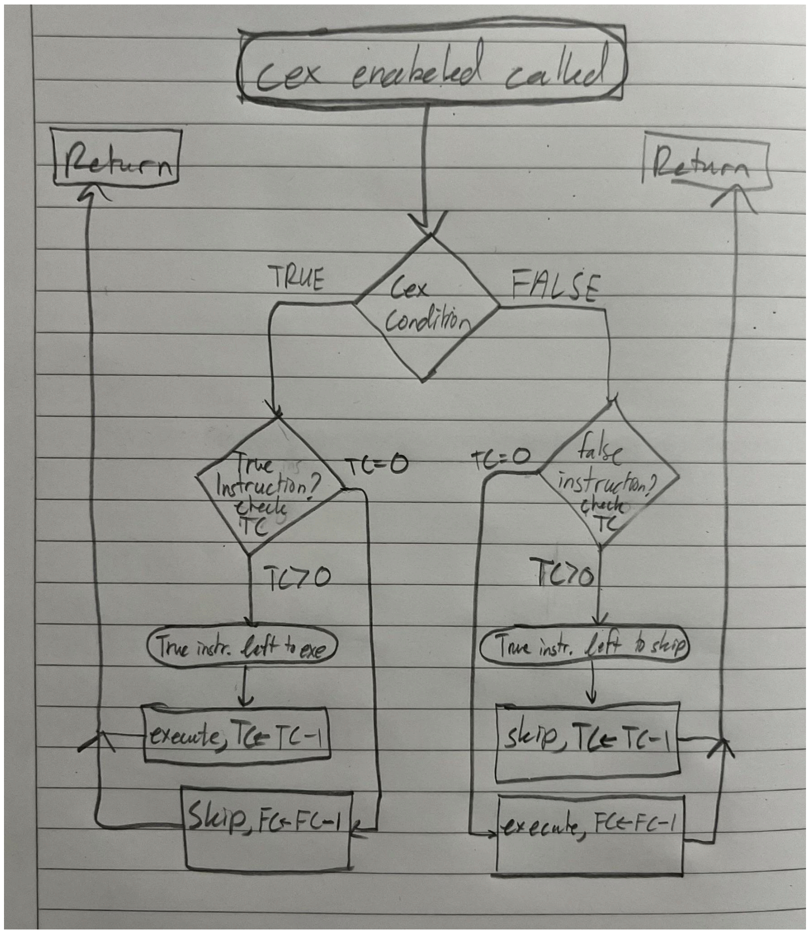
When creating the cex_enabled instruction. I needed to decide when to run the instruction and when to skip it, as I didn't want the functions after the cex instructions to run within the function cex(); as that would introduce problems later on when it comes to branch instructions after cex with the PC and LR potentially changing. So I decided to create the cex_enabled() function.

cex_enabled() would check if the CEX true count and/or false count are set to a value or they are cleared. If they are cleared then cex is disabled and the program would execute the next instruction (E0) as normal, if they contain a value then cex is enabled and is in effect so the function would be called. This can be seen in any of the “run” or “step” functions in the “run.c” file or in the “debugger.c” file (for running in debug mode and UI).

Below is the flow chart that would dictate what happens within the function if Conditional EXecution or CEX is enabled.



Flowchart highlighting how to determine whether cex is enabled or not.



Flowchart highlighting what happens in the cex_enabeled function

How to use/run the software:

To run the program, since it is written entirely in C, any machine with a gcc/gnu compiler can be used in any machine.
First ensure you have all the files in one directory or folder to be able to run this program, the files are the loader.h file, loader.c and main.c files. Navigate to that directory using the terminal using "cd <directory>". Once in the correct directory use "gcc -o main main.c debugger.c decode.c execute.c fetch.c loader.c xm23p.c run.c psw.c BRANCHinstructions.c MEMinstructions.c REGinstructions.c cex.c" to compile the program and create the ".o" file named loader, now run loader using the following command "./emulator". You should see the command window pop up and you would be able to use the menu to perform different functions.
To enter the debugger mode and test the new functions, select the "d" option.

Implementation:

Code included in the submission zip.

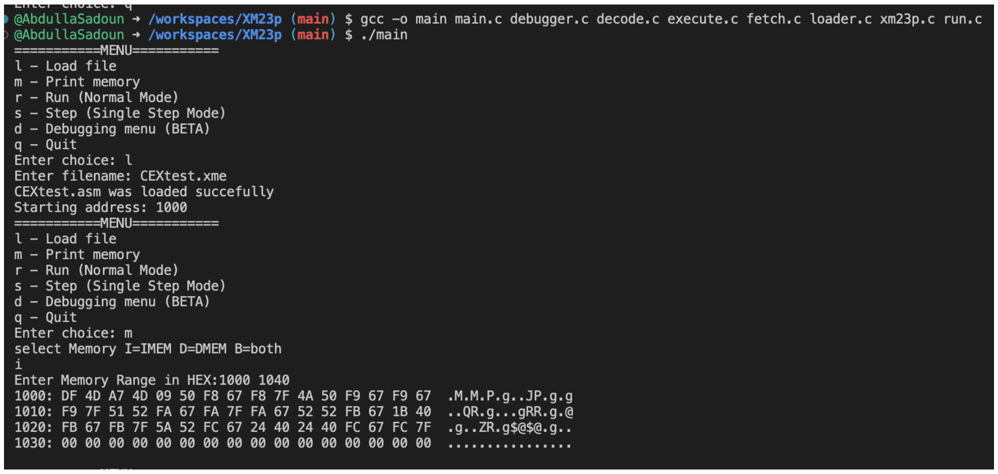
In this assignment I have significantly improved readability and code organization.

Testing

Test 1: Testing EQ condition (TRUE condition)

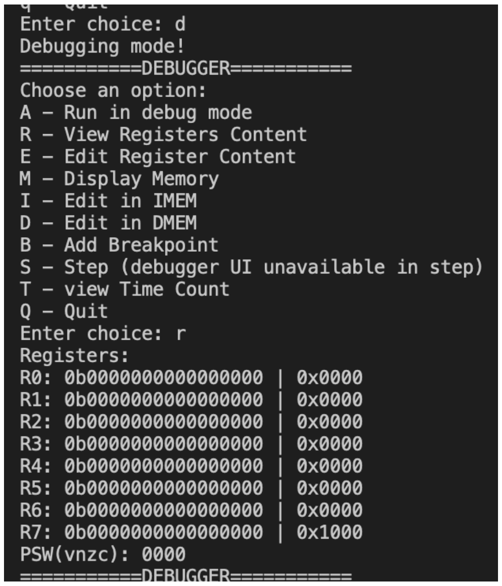
Purpose/Objective: The purpose of this test is to check if the CEX instruction in my emulator is working properly as it should (gets the condition, inspects appropriate PSW flags and executes the corresponding instructions based on the condition). For this test I'm testing the EQ condition for CEX (zero flag should be set for condition to be met)

Test Configuration: I have started the emulator loading the .xme file for the CEXtest.asm program that I have created to test the cex instruction and all the different possible conditions that it can take.



```
@AbdullaSadoun → /workspaces/XM23p (main) $ gcc -o main main.c debugger.c decode.c execute.c fetch.c loader.c xm23p.c run.c
@AbdullaSadoun → /workspaces/XM23p (main) $ ./main
===========MENU===========
l - Load file
m - Print memory
r - Run (Normal Mode)
s - Step (Single Step Mode)
d - Debugging menu (BETA)
q - Quit
Enter choice: l
Enter filename: CEXtest.xme
CEXtest.asm was loaded succefully
Starting address: 1000
===========MENU===========
l - Load file
m - Print memory
r - Run (Normal Mode)
s - Step (Single Step Mode)
d - Debugging menu (BETA)
q - Quit
Enter choice: m
select Memory I=IMEM D=DMEM B=both
i
Enter Memory Range in HEX:1000 1040
1000: DF 4D A7 4D 09 50 F8 67 F8 7F 4A 50 F9 67 F9 67  .M.M.P.g..JP.g.g
1010: F9 7F 51 52 FA 67 FA 7F FA 67 52 52 FB 67 1B 40  ..QR.g...gRR.g.@
1020: FB 67 FB 7F 5A 52 FC 67 24 40 24 40 FC 67 FC 7F  .g..ZR.g$@$@.g..
1030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

CEXtest.xme file loaded successfully and program counter set to 1000



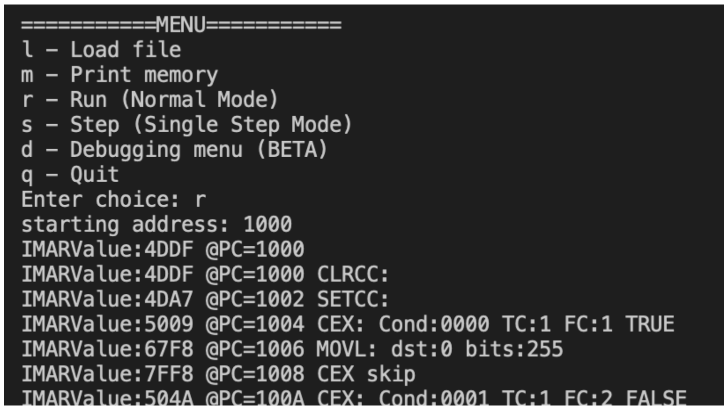
```
Enter choice: d
Debugging mode!
===========DEBUGGER===========
Choose an option:
A - Run in debug mode
R - View Registers Content
E - Edit Register Content
M - Display Memory
I - Edit in IMEM
D - Edit in DMEM
B - Add Breakpoint
S - Step (debugger UI unavailable in step)
T - view Time Count
Q - Quit
Enter choice: r
Registers:
R0: 0b0000000000000000 | 0x0000
R1: 0b0000000000000000 | 0x0000
R2: 0b0000000000000000 | 0x0000
R3: 0b0000000000000000 | 0x0000
R4: 0b0000000000000000 | 0x0000
R5: 0b0000000000000000 | 0x0000
R6: 0b0000000000000000 | 0x0000
R7: 0b0000000000000000 | 0x1000
PSW(vnzc): 0000
===========DEBUGGER===========
```

Registers and PSW before executing

```
                    ;
                    ; testing EQ - true
1004   5009            cex     eq,$1,$1 ; eq is true so r0=00ff
1006   67F8            movl    V1,R0; instruction to get r0=00FF if true
1008   7FF8            movh    V1,R0; instruction to get r0=FF00 if false
                    ; expected R0:00FF
                    ;
```

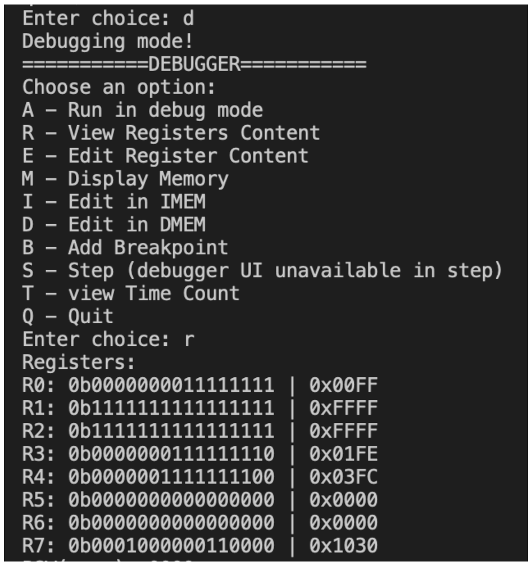
Lines of Interest from CEXtest.asm

In the CEXtest.xme I have used the instruction previously implemented to first clear all PSW flags (CLRCC vnzc), I then used the instruction SETCC to set the negative (N), zero (Z) and carry (C) flags.
I will then run through the code and observe the changes to the registers and the instructions executed based on the conditions of the CEX instructions and the PSW flags.
Expected Results: The program should work as expected running the first true instruction and skipping the next false instruction this would result in movl ff to lower byte of R0, making R0 = 0x00FF.
Actual Results: The program ran as expected, executing the first movl function (@1006) and ignoring the following instruction (@1008) since its an F instruction.



```
===========MENU===========
l - Load file
m - Print memory
r - Run (Normal Mode)
s - Step (Single Step Mode)
d - Debugging menu (BETA)
q - Quit
Enter choice: r
starting address: 1000
IMARValue:4DDF @PC=1000
IMARValue:4DDF @PC=1000 CLRCC:
IMARValue:4DA7 @PC=1002 SETCC:
IMARValue:5009 @PC=1004 CEX: Cond:0000 TC:1 FC:1 TRUE
IMARValue:67F8 @PC=1006 MOVL: dst:0 bits:255
IMARValue:7FF8 @PC=1008 CEX skip
IMARValue:504A @PC=100A CEX: Cond:0001 TC:1 FC:2 FALSE
```

List of Executed Instructions



```
Enter choice: d
Debugging mode!
===========DEBUGGER===========
Choose an option:
A - Run in debug mode
R - View Registers Content
E - Edit Register Content
M - Display Memory
I - Edit in IMEM
D - Edit in DMEM
B - Add Breakpoint
S - Step (debugger UI unavailable in step)
T - view Time Count
Q - Quit
Enter choice: r
Registers:
R0: 0b0000000011111111 | 0x00FF
R1: 0b1111111111111111 | 0xFFFF
R2: 0b1111111111111111 | 0xFFFF
R3: 0b0000000111111110 | 0x01FE
R4: 0b0000001111111100 | 0x03FC
R5: 0b0000000000000000 | 0x0000
R6: 0b0000000000000000 | 0x0000
R7: 0b0001000000110000 | 0x1030
```

Register Values after executing CEXtest.xme

Pass/Fail: Pass

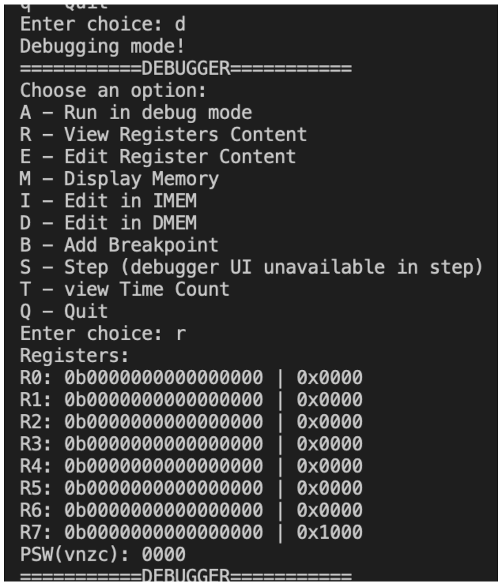
Test 2: Testing NE condition (FALSE condition)

Purpose/Objective: The purpose of this test is to check if the CEX instruction in my emulator is working properly as it should (gets the condition, inspects appropriate PSW flags and executes the corresponding instructions based on the condition). For this test I'm testing the NE condition for CEX (zero flag should be clear (0) for condition to be met)

Test Configuration: I have loaded the same .xme file for the CEXtest.asm program that I have previously used in the previous test to try out this condition within the cex instruction.

```
                    ;
                    ; testing NE - false
    100A   504A        cex     ne,$1,$2
    100C   67F9        movl    V1,R1; instruction to get r1=00FF if true
    100E   67F9        movl    V1,R1; instruction to get r1=FFFF if false
    1010   7FF9        movh    V1,R1; instruction to get r1=FFFF if false
                    ; expected R1:FFFF
                    ;
```

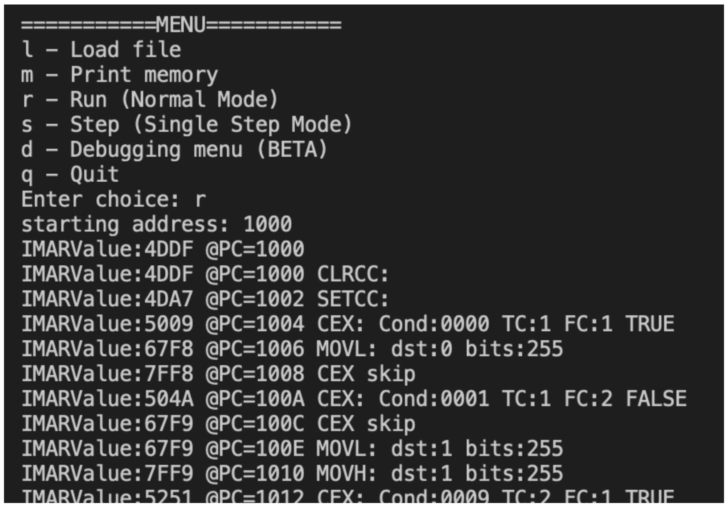
Lines of Interest from CEXtest.asm



```
Enter choice: d
Debugging mode!
===========DEBUGGER===========
Choose an option:
A - Run in debug mode
R - View Registers Content
E - Edit Register Content
M - Display Memory
I - Edit in IMEM
D - Edit in DMEM
B - Add Breakpoint
S - Step (debugger UI unavailable in step)
T - view Time Count
Q - Quit
Enter choice: r
Registers:
R0: 0b0000000000000000 | 0x0000
R1: 0b0000000000000000 | 0x0000
R2: 0b0000000000000000 | 0x0000
R3: 0b0000000000000000 | 0x0000
R4: 0b0000000000000000 | 0x0000
R5: 0b0000000000000000 | 0x0000
R6: 0b0000000000000000 | 0x0000
R7: 0b0000000000000000 | 0x1000
PSW(vnzc): 0000
===========DEBUGGER===========
```

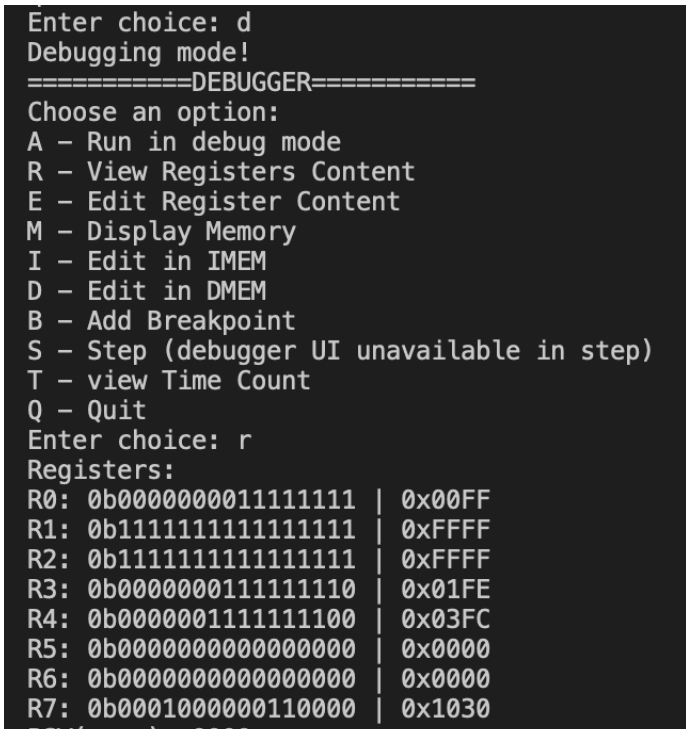
Registers and PSW before executing

In the CEXtest.xme I have used the instruction previously implemented to first clear all PSW flags (CLRCC vnzc), I then used the instruction SETCC to set the negative (N), zero (Z) and carry (C) flags.
I will then run through the code and observe the changes to the registers and the instructions executed based on the conditions of the CEX instructions and the PSW flags.
Expected Results: The program should work as expected skipping the first true instruction and running the next 2 false instructions; this would result in moving FFFF to R1, making R1 = 0xFFFF.
Actual Results: The program ran as expected, skipping the first movl function (@100C) since its a T and running the following 2 instruction (@100E & 1010) since they are F instructions.



```
===========MENU===========
l - Load file
m - Print memory
r - Run (Normal Mode)
s - Step (Single Step Mode)
d - Debugging menu (BETA)
q - Quit
Enter choice: r
starting address: 1000
IMARValue:4DDF @PC=1000
IMARValue:4DDF @PC=1000 CLRCC:
IMARValue:4DA7 @PC=1002 SETCC:
IMARValue:5009 @PC=1004 CEX: Cond:0000 TC:1 FC:1 TRUE
IMARValue:67F8 @PC=1006 MOVL: dst:0 bits:255
IMARValue:7FF8 @PC=1008 CEX skip
IMARValue:504A @PC=100A CEX: Cond:0001 TC:1 FC:2 FALSE
IMARValue:67F9 @PC=100C CEX skip
IMARValue:67F9 @PC=100E MOVL: dst:1 bits:255
IMARValue:7FF9 @PC=1010 MOVH: dst:1 bits:255
IMARValue:5251 @PC=1012 CEX: Cond:0009 TC:2 FC:1 TRUE
```

List of Executed Instructions



```
Enter choice: d
Debugging mode!
===========DEBUGGER===========
Choose an option:
A - Run in debug mode
R - View Registers Content
E - Edit Register Content
M - Display Memory
I - Edit in IMEM
D - Edit in DMEM
B - Add Breakpoint
S - Step (debugger UI unavailable in step)
T - view Time Count
Q - Quit
Enter choice: r
Registers:
R0: 0b0000000011111111 | 0x00FF
R1: 0b1111111111111111 | 0xFFFF
R2: 0b1111111111111111 | 0xFFFF
R3: 0b0000000111111110 | 0x01FE
R4: 0b0000001111111100 | 0x03FC
R5: 0b0000000000000000 | 0x0000
R6: 0b0000000000000000 | 0x0000
R7: 0b0001000000110000 | 0x1030
```

Register Values after executing CEXtest.xme

Pass/Fail: Pass

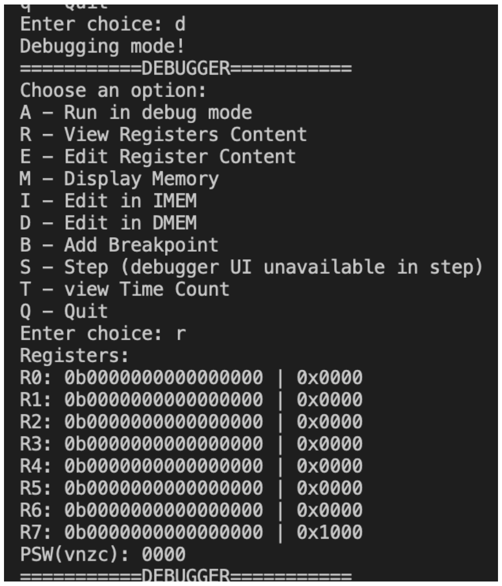
Test 3: Testing TRUE LS condition & Mismatching T/F instruction Numbers

Purpose/Objective: The purpose of this test is to check if the CEX instruction in my emulator is working properly as it should (gets the condition, inspects appropriate PSW flags and executes the corresponding instructions based on the condition). For this test I'm testing the LS condition for CEX (carry flag should be clear (0) OR the zero flag should be set (1) for condition to be met).

Test Configuration: I have loaded the same .xme file for the CEXtest.asm program that I have previously used in the previous test to try out this condition within the cex instruction.

```
                    ;
                    ; testing LS - true
      1012   5251      cex     ls,$2,$1
      1014   67FA      movl    V1,R2; instruction to get r2=FFFF if true
      1016   7FFA      movh    V1,R2; instruction to get r2=FFFF if true
      1018   67FA      movl    V1,R2; instruction to get r2=00FF if false
                    ; expected R2:FFFF
                    ;
```

Lines of Interest from CEXtest.asm

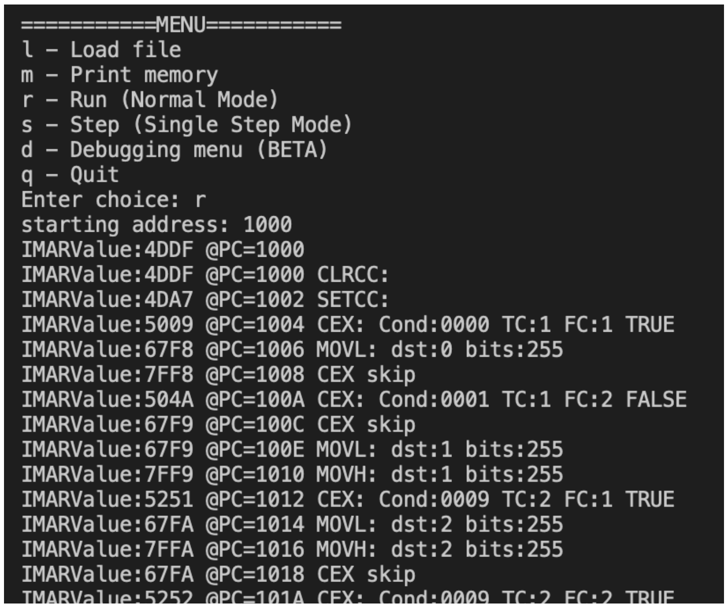


```
Enter choice: d
Debugging mode!
===========DEBUGGER===========
Choose an option:
A - Run in debug mode
R - View Registers Content
E - Edit Register Content
M - Display Memory
I - Edit in IMEM
D - Edit in DMEM
B - Add Breakpoint
S - Step (debugger UI unavailable in step)
T - view Time Count
Q - Quit
Enter choice: r
Registers:
R0: 0b0000000000000000 | 0x0000
R1: 0b0000000000000000 | 0x0000
R2: 0b0000000000000000 | 0x0000
R3: 0b0000000000000000 | 0x0000
R4: 0b0000000000000000 | 0x0000
R5: 0b0000000000000000 | 0x0000
R6: 0b0000000000000000 | 0x0000
R7: 0b0000000000000000 | 0x1000
PSW(vnzc): 0000
===========DEBUGGER===========
```

Registers and PSW before executing

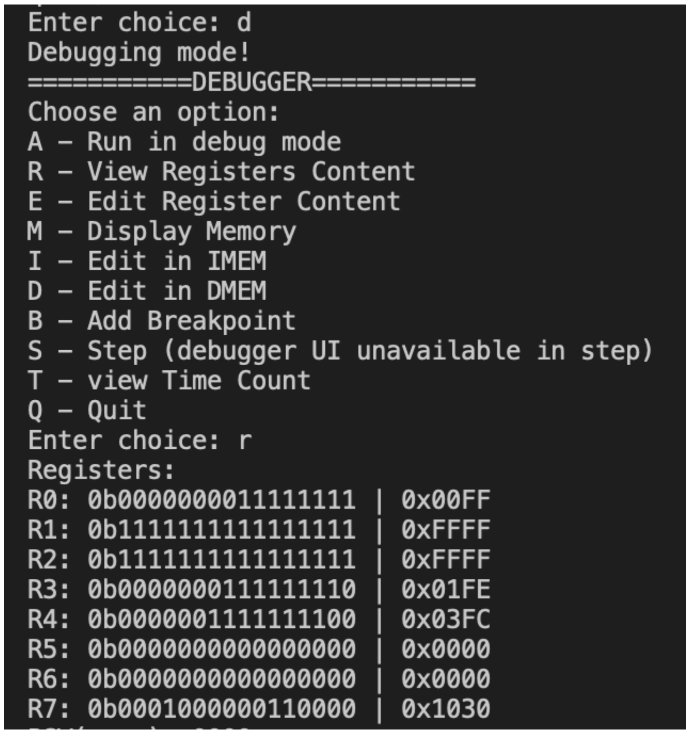
In the CEXtest.xme I have used the instruction previously implemented to first clear all PSW flags (CLRCC vnzc), I then used the instruction SETCC to set the negative (N), zero (Z) and carry (C) flags.

I will then run through the code and observe the changes to the registers and the instructions executed based on the conditions of the CEX instructions and the PSW flags.
Expected Results: The program should work as expected running the first 2 true instructions and skipping the next false instructions; this would result in moving FFFF to R2, making R2 = 0xFFFF.
Actual Results: The program ran as expected, running the first 2 instructions (@1014 & 1016) since they are T-function and skipping the following instruction (@1018) since they are F instructions.



```
===========MENU===========
l - Load file
m - Print memory
r - Run (Normal Mode)
s - Step (Single Step Mode)
d - Debugging menu (BETA)
q - Quit
Enter choice: r
starting address: 1000
IMARValue:4DDF @PC=1000
IMARValue:4DDF @PC=1000 CLRCC:
IMARValue:4DA7 @PC=1002 SETCC:
IMARValue:5009 @PC=1004 CEX: Cond:0000 TC:1 FC:1 TRUE
IMARValue:67F8 @PC=1006 MOVL: dst:0 bits:255
IMARValue:7FF8 @PC=1008 CEX skip
IMARValue:504A @PC=100A CEX: Cond:0001 TC:1 FC:2 FALSE
IMARValue:67F9 @PC=100C CEX skip
IMARValue:67F9 @PC=100E MOVL: dst:1 bits:255
IMARValue:7FF9 @PC=1010 MOVH: dst:1 bits:255
IMARValue:5251 @PC=1012 CEX: Cond:0009 TC:2 FC:1 TRUE
IMARValue:67FA @PC=1014 MOVL: dst:2 bits:255
IMARValue:7FFA @PC=1016 MOVH: dst:2 bits:255
IMARValue:67FA @PC=1018 CEX skip
IMARValue:5252 @PC=101A CEX: Cond:0009 TC:2 FC:2 TRUE
```

List of Executed Instructions



```
Enter choice: d
Debugging mode!
===========DEBUGGER===========
Choose an option:
A - Run in debug mode
R - View Registers Content
E - Edit Register Content
M - Display Memory
I - Edit in IMEM
D - Edit in DMEM
B - Add Breakpoint
S - Step (debugger UI unavailable in step)
T - view Time Count
Q - Quit
Enter choice: r
Registers:
R0: 0b0000000011111111 | 0x00FF
R1: 0b1111111111111111 | 0xFFFF
R2: 0b1111111111111111 | 0xFFFF
R3: 0b0000000111111110 | 0x01FE
R4: 0b0000001111111100 | 0x03FC
R5: 0b0000000000000000 | 0x0000
R6: 0b0000000000000000 | 0x0000
R7: 0b0001000000110000 | 0x1030
```

Register Values after executing CEXtest.xme

Pass/Fail: Pass

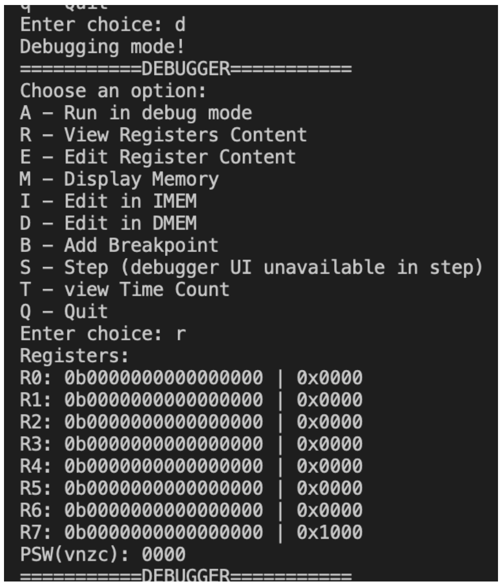
Test 4: Testing TRUE LS condition & Same T/F instruction Numbers

Purpose/Objective: The purpose of this test is to check if the CEX instruction in my emulator is working properly as it should (gets the condition, inspects appropriate PSW flags and executes the corresponding instructions based on the condition). For this test I’m testing the LS condition for CEX (carry flag should be clear (0) OR the zero flag should be set (1) for condition to be met) and im testing to see how the program reacts to having two of the same number of executions to do after the CEX instruction is called.

Test Configuration: I have loaded the same .xme file for the CEXtest.asm program that I have previously used in the previous test to try out this condition within the cex instruction.

```
                  ; testing LS - true
101A   5252         cex    ls,$2,$2
101C   67FB         movl   V1,R3; instruction to get r3=00FF if true
101E   401B         add    R3,R3; instruction to get r3=01FE if true
1020   67FB         movl   V1,R3; instruction to get r3=00FF if false
1022   7FFB         movh   V1,R3; instruction to get r3=FFFF if false
                  ; expected R3:01FE
                  ;
```

Lines of Interest from CEXtest.asm



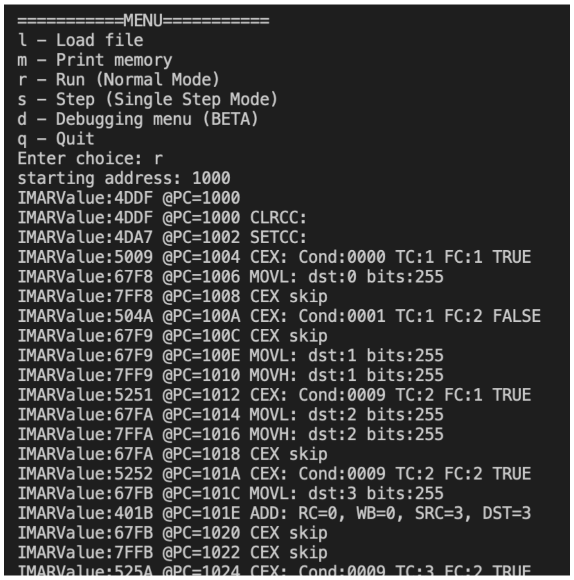
```
Enter choice: d
Debugging mode!
===========DEBUGGER===========
Choose an option:
A - Run in debug mode
R - View Registers Content
E - Edit Register Content
M - Display Memory
I - Edit in IMEM
D - Edit in DMEM
B - Add Breakpoint
S - Step (debugger UI unavailable in step)
T - view Time Count
Q - Quit
Enter choice: r
Registers:
R0: 0b0000000000000000 | 0x0000
R1: 0b0000000000000000 | 0x0000
R2: 0b0000000000000000 | 0x0000
R3: 0b0000000000000000 | 0x0000
R4: 0b0000000000000000 | 0x0000
R5: 0b0000000000000000 | 0x0000
R6: 0b0000000000000000 | 0x0000
R7: 0b0000000000000000 | 0x1000
PSW(vnzc): 0000
===========DEBUGGER===========
```

Registers and PSW before executing

In the CEXtest.xme I have used the instruction previously implemented to first clear all PSW flags (CLRCC vnzc), I then used the instruction SETCC to set the negative (N), zero (Z) and carry (C) flags.

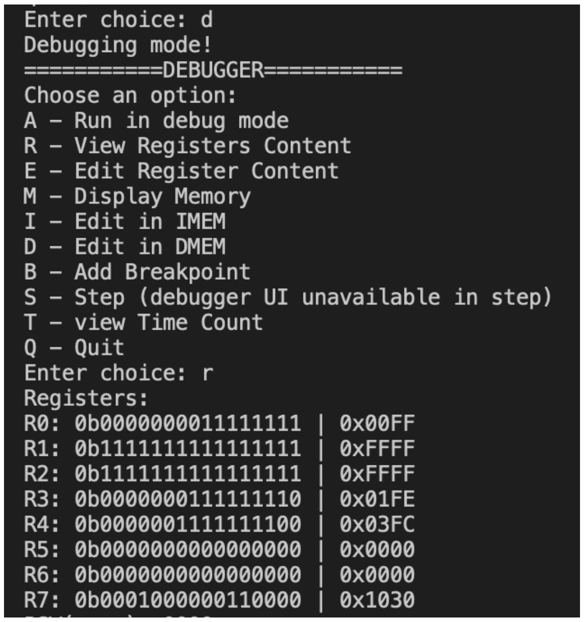
I will then run through the code and observe the changes to the registers and the instructions executed based on the conditions of the CEX instructions and the PSW flags.

Expected Results: The program should work as expected running the first 2 true instructions and skipping the next 2 false instructions; this would result in moving 00FF to R3 and then adding R3 to itself and saving the result in R3, making R3 = 0x01FE.
Actual Results: The program ran as expected, running the first 2 instructions (@101C & 101E) since they are T-function and skipping the following 2 instructions (@1020 & 1022) since they are F instructions.



```
===========MENU===========
l - Load file
m - Print memory
r - Run (Normal Mode)
s - Step (Single Step Mode)
d - Debugging menu (BETA)
q - Quit
Enter choice: r
starting address: 1000
IMARValue:4DDF @PC=1000
IMARValue:4DDF @PC=1000 CLRCC:
IMARValue:4DA7 @PC=1002 SETCC:
IMARValue:5009 @PC=1004 CEX: Cond:0000 TC:1 FC:1 TRUE
IMARValue:67F8 @PC=1006 MOVL: dst:0 bits:255
IMARValue:7FF8 @PC=1008 CEX skip
IMARValue:504A @PC=100A CEX: Cond:0001 TC:1 FC:2 FALSE
IMARValue:67F9 @PC=100C CEX skip
IMARValue:67F9 @PC=100E MOVL: dst:1 bits:255
IMARValue:7FF9 @PC=1010 MOVH: dst:1 bits:255
IMARValue:5251 @PC=1012 CEX: Cond:0009 TC:2 FC:1 TRUE
IMARValue:67FA @PC=1014 MOVL: dst:2 bits:255
IMARValue:7FFA @PC=1016 MOVH: dst:2 bits:255
IMARValue:67FA @PC=1018 CEX skip
IMARValue:5252 @PC=101A CEX: Cond:0009 TC:2 FC:2 TRUE
IMARValue:67FB @PC=101C MOVL: dst:3 bits:255
IMARValue:401B @PC=101E ADD: RC=0, WB=0, SRC=3, DST=3
IMARValue:67FB @PC=1020 CEX skip
IMARValue:7FFB @PC=1022 CEX skip
IMARValue:525A @PC=1024 CEX: Cond:0009 TC:3 FC:2 TRUE
```

List of Executed Instructions



```
Enter choice: d
Debugging mode!
===========DEBUGGER===========
Choose an option:
A - Run in debug mode
R - View Registers Content
E - Edit Register Content
M - Display Memory
I - Edit in IMEM
D - Edit in DMEM
B - Add Breakpoint
S - Step (debugger UI unavailable in step)
T - view Time Count
Q - Quit
Enter choice: r
Registers:
R0: 0b0000000011111111 | 0x00FF
R1: 0b1111111111111111 | 0xFFFF
R2: 0b1111111111111111 | 0xFFFF
R3: 0b0000000111111110 | 0x01FE
R4: 0b0000001111111100 | 0x03FC
R5: 0b0000000000000000 | 0x0000
R6: 0b0000000000000000 | 0x0000
R7: 0b0001000000110000 | 0x1030
```

Register Values after executing CEXtest.xme

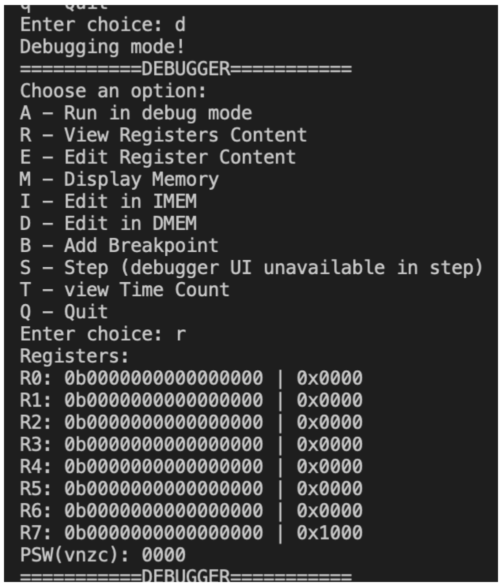
Pass/Fail: Pass

Test 5: Testing TR (Always TRUE) & 5 T/F instructions

Purpose/Objective: The purpose of this test is to check if the CEX instruction in my emulator is working properly as it should (gets the condition, inspects appropriate PSW flags and executes the corresponding instructions based on the condition). For this test I'm testing the TR condition for CEX (always runs the the true and skips the false no matter the PSW) and I'm testing to see how the program reacts to having 5 instructions to execute from after the CEX instruction is called/used.
Test Configuration: I have loaded the same .xme file for the CEXtest.asm program that I have previously used in the previous test to try out this condition within the cex instruction.

```
                    ; testing TR - always true
  1024   525A         cex    ls,$3,$2
  1026   67FC         movl   V1,R4; instruction to get r4=00FF if true
  1028   4024         add    R4,R4; instruction to get r4=01FE if true
  102A   4024         add    R4,R4; instruction to get r4=03FE if true
  102C   67FC         movl   V1,R4; instruction to get r4=00FF if false
  102E   7FFC         movh   V1,R4; instruction to get r4=FFFF if false
                    ; expected R4:03FE
                    ;
```

Lines of Interest from CEXtest.asm

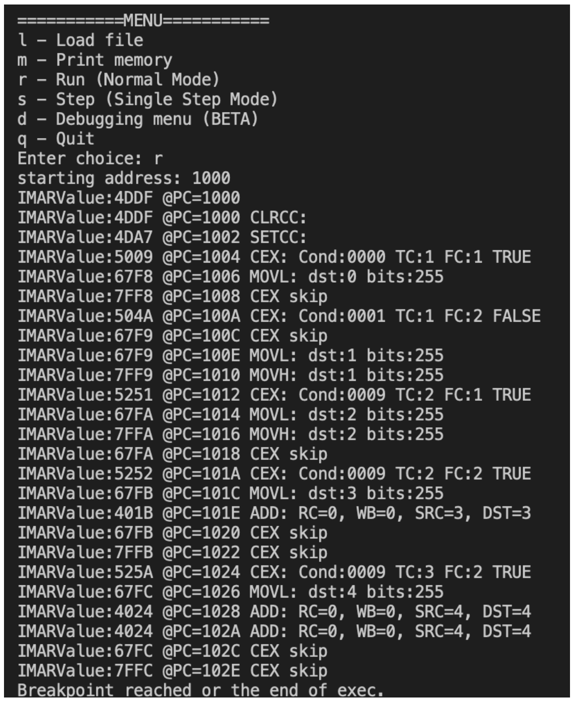


```
Enter choice: d
Debugging mode!
===========DEBUGGER===========
Choose an option:
A - Run in debug mode
R - View Registers Content
E - Edit Register Content
M - Display Memory
I - Edit in IMEM
D - Edit in DMEM
B - Add Breakpoint
S - Step (debugger UI unavailable in step)
T - view Time Count
Q - Quit
Enter choice: r
Registers:
R0: 0b0000000000000000 | 0x0000
R1: 0b0000000000000000 | 0x0000
R2: 0b0000000000000000 | 0x0000
R3: 0b0000000000000000 | 0x0000
R4: 0b0000000000000000 | 0x0000
R5: 0b0000000000000000 | 0x0000
R6: 0b0000000000000000 | 0x0000
R7: 0b0000000000000000 | 0x1000
PSW(vnzc): 0000
===========DEBUGGER===========
```

Registers and PSW before executing

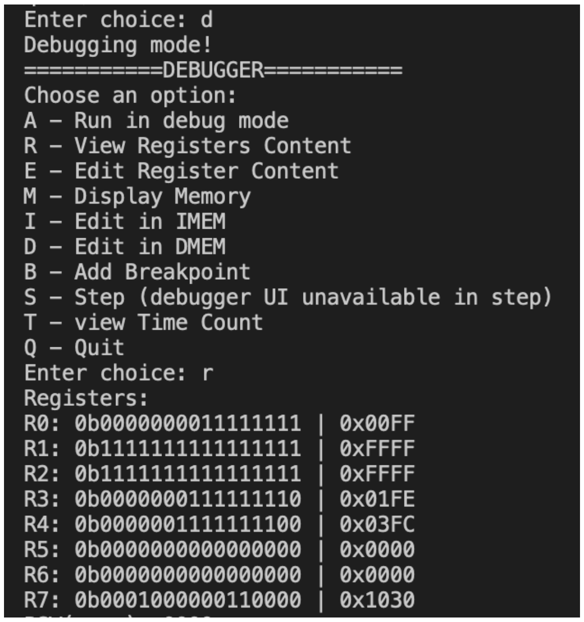
In the CEXtest.xme I have used the instruction previously implemented to first clear all PSW flags (CLRCC vnzc), I then used the instruction SETCC to set the negative (N), zero (Z) and carry (C) flags.
I will then run through the code and observe the changes to the registers and the instructions executed based on the conditions of the CEX instructions and the PSW flags.

Expected Results: The program should work as expected running the first 3 true instructions and skipping the next 2 false instructions; this would result in moving 00FF to R4 and then adding R4 to itself and saving the result in R3 and then adding R4 to itself again and storing the result in R4 again, making R4 = 0x03FE.
Actual Results: The program ran as expected, running the first 3 instructions (@1026, 1028 & 102A) since they are T-function and skipping the following 2 instructions (@102C & 102E) since they are F instructions.



```
===========MENU===========
l - Load file
m - Print memory
r - Run (Normal Mode)
s - Step (Single Step Mode)
d - Debugging menu (BETA)
q - Quit
Enter choice: r
starting address: 1000
IMARValue:4DDF @PC=1000
IMARValue:4DDF @PC=1000 CLRCC:
IMARValue:4DA7 @PC=1002 SETCC:
IMARValue:5009 @PC=1004 CEX: Cond:0000 TC:1 FC:1 TRUE
IMARValue:67F8 @PC=1006 MOVL: dst:0 bits:255
IMARValue:7FF8 @PC=1008 CEX skip
IMARValue:504A @PC=100A CEX: Cond:0001 TC:1 FC:2 FALSE
IMARValue:67F9 @PC=100C CEX skip
IMARValue:67F9 @PC=100E MOVL: dst:1 bits:255
IMARValue:7FF9 @PC=1010 MOVH: dst:1 bits:255
IMARValue:5251 @PC=1012 CEX: Cond:0009 TC:2 FC:1 TRUE
IMARValue:67FA @PC=1014 MOVL: dst:2 bits:255
IMARValue:7FFA @PC=1016 MOVH: dst:2 bits:255
IMARValue:67FA @PC=1018 CEX skip
IMARValue:5252 @PC=101A CEX: Cond:0009 TC:2 FC:2 TRUE
IMARValue:67FB @PC=101C MOVL: dst:3 bits:255
IMARValue:401B @PC=101E ADD: RC=0, WB=0, SRC=3, DST=3
IMARValue:67FB @PC=1020 CEX skip
IMARValue:7FFB @PC=1022 CEX skip
IMARValue:525A @PC=1024 CEX: Cond:0009 TC:3 FC:2 TRUE
IMARValue:67FC @PC=1026 MOVL: dst:4 bits:255
IMARValue:4024 @PC=1028 ADD: RC=0, WB=0, SRC=4, DST=4
IMARValue:4024 @PC=102A ADD: RC=0, WB=0, SRC=4, DST=4
IMARValue:67FC @PC=102C CEX skip
IMARValue:7FFC @PC=102E CEX skip
Breakpoint reached or the end of exec.
```

List of Executed Instructions



```
Enter choice: d
Debugging mode!
===========DEBUGGER===========
Choose an option:
A - Run in debug mode
R - View Registers Content
E - Edit Register Content
M - Display Memory
I - Edit in IMEM
D - Edit in DMEM
B - Add Breakpoint
S - Step (debugger UI unavailable in step)
T - view Time Count
Q - Quit
Enter choice: r
Registers:
R0: 0b0000000011111111 | 0x00FF
R1: 0b1111111111111111 | 0xFFFF
R2: 0b1111111111111111 | 0xFFFF
R3: 0b0000000111111110 | 0x01FE
R4: 0b0000001111111100 | 0x03FC
R5: 0b0000000000000000 | 0x0000
R6: 0b0000000000000000 | 0x0000
R7: 0b0001000000110000 | 0x1030
```

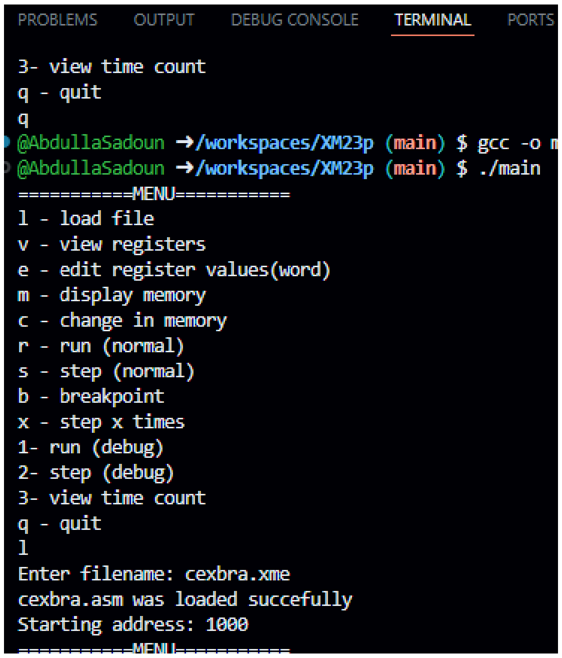
Register Values after executing CEXtest.xme

Pass/Fail: Pass

Test 6: Testing Branching after cex (EMAD RECOMMENDATION)

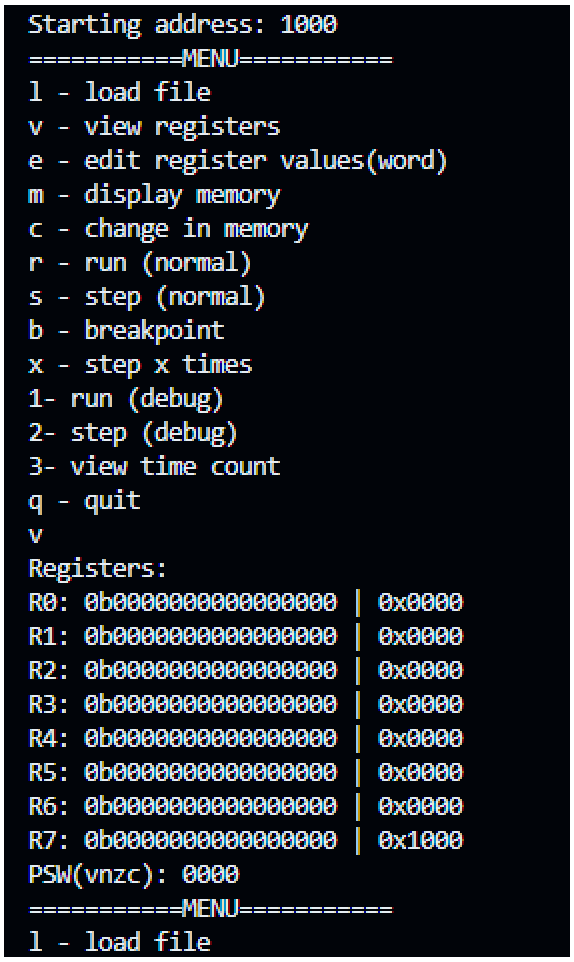
Purpose/Objective: The purpose of this test is to check if reset_cex function is working properly disabling the enabled cex after a branching instruction that is allowed to execute because of a cex's condition.

Test Configuration: I have loaded a new .xme file called cexbra.asm.I will branch after the cex and then test if cex is still enabled or not.



```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

3- view time count
q - quit
q
@AbdullaSadoun →/workspaces/XM23p (main) $ gcc -o m
@AbdullaSadoun →/workspaces/XM23p (main) $ ./main
===========MENU===========
l - load file
v - view registers
e - edit register values(word)
m - display memory
c - change in memory
r - run (normal)
s - step (normal)
b - breakpoint
x - step x times
1- run (debug)
2- step (debug)
3- view time count
q - quit
l
Enter filename: cexbra.xme
cexbra.asm was loaded succefully
Starting address: 1000
===========MENU===========
```

Loaded cexbra.asm



```
Starting address: 1000
===========MENU===========
l - load file
v - view registers
e - edit register values(word)
m - display memory
c - change in memory
r - run (normal)
s - step (normal)
b - breakpoint
x - step x times
1- run (debug)
2- step (debug)
3- view time count
q - quit
v
Registers:
R0: 0b0000000000000000 | 0x0000
R1: 0b0000000000000000 | 0x0000
R2: 0b0000000000000000 | 0x0000
R3: 0b0000000000000000 | 0x0000
R4: 0b0000000000000000 | 0x0000
R5: 0b0000000000000000 | 0x0000
R6: 0b0000000000000000 | 0x0000
R7: 0b0000000000000000 | 0x1000
PSW(vnzc): 0000
===========MENU===========
l - load file
```

Registers and PSW before executing

In the cexbra.xme I have used the instruction previously implemented to first clear all PSW flags (CLRCC vnzc), I then used the instruction SETCC to set the negative (N), zero (Z) and carry (C) flags. I then used cex eq,$2,$2. With BRA used right after the cex. This should disable cex.

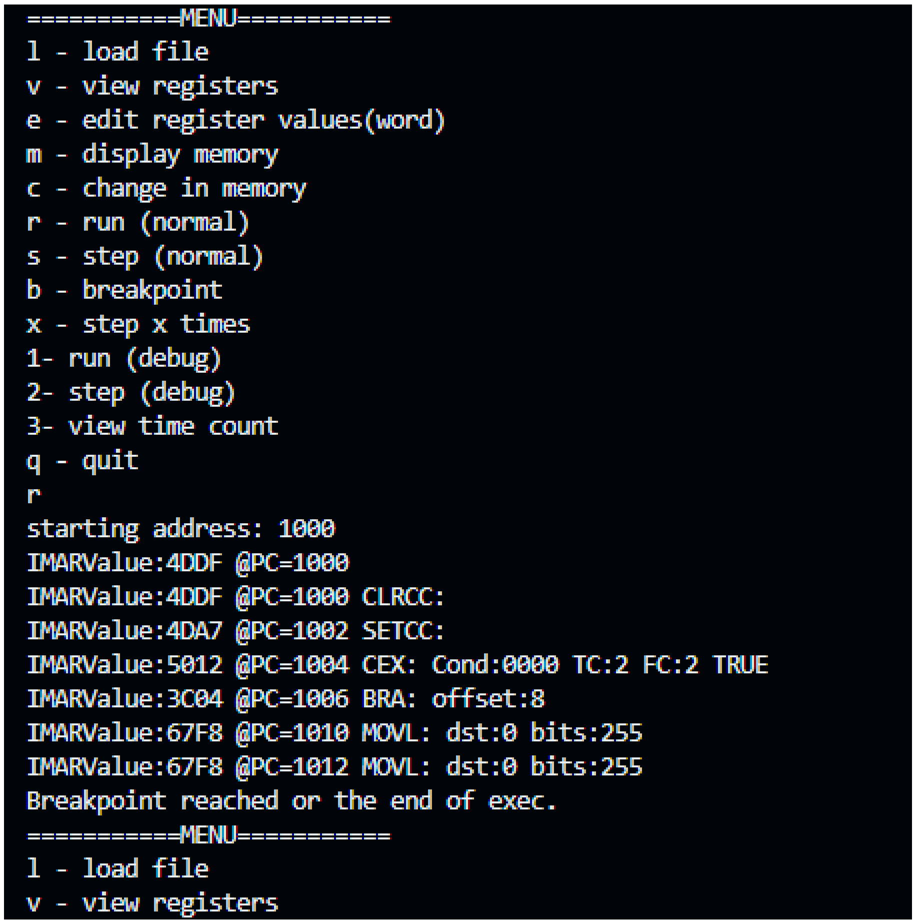
```
27    1004    5012    cex    eq,$2,$2 ; eq is true
28    1006    3C04    bra    SETR0; subr to set (r0=FFFF if true) (should execute)
29    1008    67F9    movl   V1,R1; instruction to get r0=00FF if true and executed (shouldnt execute)
30    100A    67FA    movl   V1,R2; instruction to get r2=00FF if false (shouldnt execute)
31    100C    7FFA    movh   V1,R2; instruction to get r2=FF00 if false (shouldnt execute)
```

Configuration of cex from cexbra.lis

I will then run through the code and observe the changes to the registers and the instructions executed based on the conditions of the CEX instructions and the PSW flags and the newly added cex_disabler.

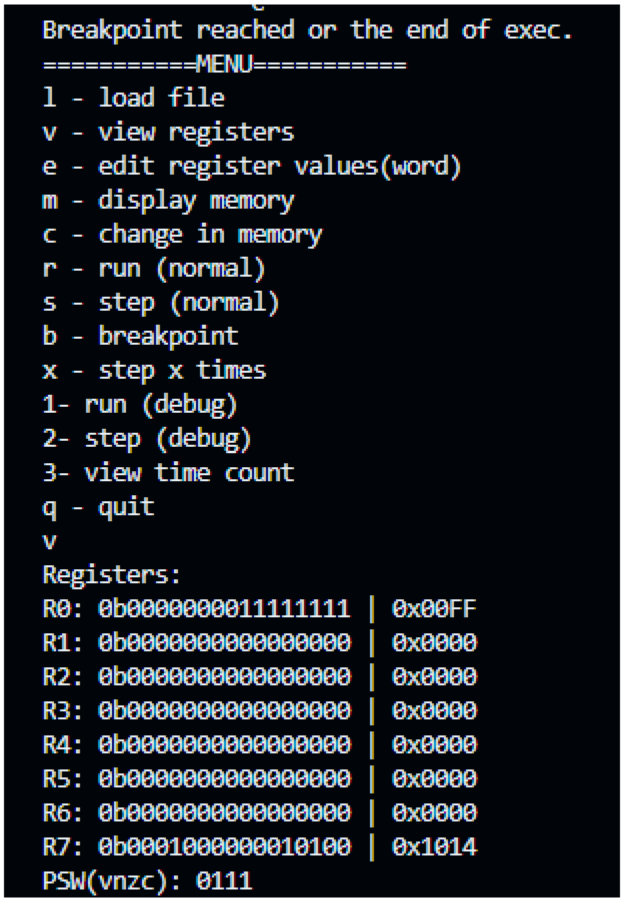
Expected Results: The program should work as expected running the cex function, getting that the condition is true and should decide to start executing the true conditions, once it does that, it should then get disabled after BRA and go to the subroutine, if this doesnt work the program will go to an infinite loop in fail, or will print into R1 or R2. If it goes as expected then it should go to the SETR0 subroutine and set R0 to 0x00FF

Actual Results: The program ran as expected, disabling cex and it's effects after running the bra instruction. It has set R0 to 0x00FF and did not alter R1 or R2 nor did it go to the infinite loop.



```
===========MENU===========
l - load file
v - view registers
e - edit register values(word)
m - display memory
c - change in memory
r - run (normal)
s - step (normal)
b - breakpoint
x - step x times
1- run (debug)
2- step (debug)
3- view time count
q - quit
r
starting address: 1000
IMARValue:4DDF @PC=1000
IMARValue:4DDF @PC=1000 CLRCC:
IMARValue:4DA7 @PC=1002 SETCC:
IMARValue:5012 @PC=1004 CEX: Cond:0000 TC:2 FC:2 TRUE
IMARValue:3C04 @PC=1006 BRA: offset:8
IMARValue:67F8 @PC=1010 MOVL: dst:0 bits:255
IMARValue:67F8 @PC=1012 MOVL: dst:0 bits:255
Breakpoint reached or the end of exec.
===========MENU===========
l - load file
v - view registers
```

List of Executed Instructions



```
Breakpoint reached or the end of exec.
===========MENU===========
l - load file
v - view registers
e - edit register values(word)
m - display memory
c - change in memory
r - run (normal)
s - step (normal)
b - breakpoint
x - step x times
1- run (debug)
2- step (debug)
3- view time count
q - quit
v
Registers:
R0: 0b0000000011111111 | 0x00FF
R1: 0b0000000000000000 | 0x0000
R2: 0b0000000000000000 | 0x0000
R3: 0b0000000000000000 | 0x0000
R4: 0b0000000000000000 | 0x0000
R5: 0b0000000000000000 | 0x0000
R6: 0b0000000000000000 | 0x0000
R7: 0b0001000000010100 | 0x1014
PSW(vnzc): 0111
```

Register Values after executing cexbra.xme

Pass/Fail: Pass

Extra Content and Notes

CEXtest.lis (file used)

```
X-Makina Assembler - Version XM-23P Single Pass+ Assembler - Release 24.04.17
Input file name: CEXtest.asm
Time of assembly: Sun 4 Aug 2024 01:38:46
                        ;
                        ; Test of A5 (Conditional Execution)
                        ; Tests multiple conditions
                        ; Tests true and false
                        ;
                        ; data segment
                        ; FFFF stored in DMEM[FFFF]
                          data
                          org #FFFF
        FFFF    FFFF    V1      word    #FFFF
                        ;
                        ;
                        ;code segment
                          code
                          org #1000
                        MAIN
                        ;
                        ; ensure psw is cleared
1000    4DDF                  clrcc     vsnzc ; check to see if the instruction works (PSW:00000)
                        ;
                        ;
1002    4DA7           setcc   nzc ; Set N, Z, C flags (PSW: 00111)
                              ; EQ, CS, MI, LE, VC should be true
                              ; NE, PL, CC, GE, LT, GT should be false
                        ;
                        ; testing EQ - true
1004    5009            cex     eq,$1,$1 ; eq is true so r0=00ff
1006    67F8            movl    V1,R0; instruction to get r0=00FF if true
1008    7FF8            movh    V1,R0; instruction to get r0=FF00 if false
                        ; expected R0:00FF
                        ;
                        ;
                        ; testing NE - false
100A    504A            cex     ne,$1,$2
100C    67F9            movl    V1,R1; instruction to get r1=00FF if true
100E    67F9            movl    V1,R1; instruction to get r1=FFFF if false
1010    7FF9            movh    V1,R1; instruction to get r1=FFFF if false
                        ; expected R1:FFFF
                        ;
                        ;
                        ; testing LS - true
1012    5251            cex     ls,$2,$1
1014    67FA            movl    V1,R2; instruction to get r2=FFFF if true
1016    7FFA            movh    V1,R2; instruction to get r2=FFFF if true
1018    67FA            movl    V1,R2; instruction to get r2=00FF if false
                        ; expected R2:FFFF
                        ;
                        ;
                        ; testing LS - true
101A    5252            cex     ls,$2,$2
101C    67FB            movl    V1,R3; instruction to get r3=00FF if true
101E    401B            add     R3,R3; instruction to get r3=01FE if true
1020    67FB            movl    V1,R3; instruction to get r3=00FF if false
1022    7FFB            movh    V1,R3; instruction to get r3=FFFF if false
                        ; expected R3:01FE
                        ;
                        ;
                        ; testing TR - always true
1024    525A            cex     ls,$3,$2
1026    67FC            movl    V1,R4; instruction to get r4=00FF if true
1028    4024            add     R4,R4; instruction to get r4=01FE if true
102A    4024            add     R4,R4; instruction to get r4=03FE if true
102C    67FC            movl    V1,R4; instruction to get r4=00FF if false
102E    7FFC            movh    V1,R4; instruction to get r4=FFFF if false
                        ; expected R4:03FE
                        ;
                        ;
                        ;
                        end MAIN
Successful completion of assembly - 1P

** Symbol table **
```

```
Constants (Equates)
Name                        Type      Value     Decimal

Labels (Code)
Name                        Type      Value     Decimal
MAIN                        REL       1000      4096      PRI

Labels (Data)
Name                        Type      Value     Decimal
V1                          REL       FFFF      -1        PRI

Registers
Name                        Type      Value     Decimal
R7                          REG       0007      7         PRI
R6                          REG       0006      6         PRI
R5                          REG       0005      5         PRI
R4                          REG       0004      4         PRI
R3                          REG       0003      3         PRI
R2                          REG       0002      2         PRI
R1                          REG       0001      1         PRI
R0                          REG       0000      0         PRI

.XME file: \\Mac\Home\Desktop\Computer Architecture\Assembler\CEXtest.xme
```

CEXtest.xme (file used)

```
S00E00004345587465737742E61736DE2
S205FFFFFFFFFE
S1211000DF4DA74D0950F867F87F4A50F967F967F97F5152FA67FA7FFA675252FB67C9
S115101E1B40FB67FB7F5A52FC6724402440FC67FC7FD0
S9031000EC
```

cexbra.lis (file used)

```
X-Makina Assembler - Version XM-23P Single Pass+ Assembler - Release 24.04.17
Input file name: cexbra.asm
Time of assembly: Wed 7 Aug 2024 14:47:35
  1                             ;
  2                             ; Test of A5 (Conditional Execution)
  3                             ; BRA test
  4                             ; Tests disabling cex after BRA execution
  5                             ;
  6                             ; data segment
  7                             ; FFFF stored in DMEM[FFFF]
  8                               data
  9                               org #FFFF
 10      FFFF      FFFF         V1        word      #FFFF
 11                             ;
 12                             ;
 13                             ;code segment
 14                               code
 15                               org #1000
 16                             MAIN
 17                             ;
 18                             ; ensure psw is cleared
 19      1000      4DDF                   clrcc     vsnzc ; check to see if the instruction works (PSW:00000)
```

```
                      ;
                      ;
1002    4DA7            setcc   nzc ; Set N, Z, C flags (PSW: 00111)
                                ; EQ, CS, MI, LE, VC should be true
                                ; NE, PL, CC, GE, LT, GT should be false
                      ;
                      ; testing EQ - true
1004    5012            cex     eq,$2,$2 ; eq is true
1006    3C04            bra     SETR0; subr to set (r0=FFFF if true) (should execute)
1008    67F9            movl    V1,R1; instruction to get r0=00FF if true and executed (shouldnt execute)
100A    67FA            movl    V1,R2; instruction to get r2=00FF if false (shouldnt execute)
100C    7FFA            movh    V1,R2; instruction to get r2=FF00 if false (shouldnt execute)
                      ; expected everything after bra shouldnt execute, output will be in r0
                      ; r0: 00FF
                      ; r1: 0000
                      ; r2: 0000
                      ;
100E    3FFF            FAILED  bra FAILED ;infinite loop if test fails
                      ;
                      ; the following subr should execute proving our fault handler works
                      SETR0
1010    67F8            movl    V1,R0; set r0 to 00FF
1012    67F8            movl    V1,R0; set r0 to 00FF
                      ;
                      ;
                      ; Terminate the program
                        end MAIN
Successful completion of assembly - 2P

** Symbol table **

Constants (Equates)
Name                  Type    Value   Decimal

Labels (Code)
Name                  Type    Value   Decimal
FAILED                REL     100E    4110    PRI
SETR0                 REL     1010    4112    PRI
MAIN                  REL     1000    4096    PRI

Labels (Data)
Name                  Type    Value   Decimal
V1                    REL     FFFF    -1      PRI

Registers
Name                  Type    Value   Decimal
R7                    REG     0007    7       PRI
R6                    REG     0006    6       PRI
R5                    REG     0005    5       PRI
R4                    REG     0004    4       PRI
R3                    REG     0003    3       PRI
R2                    REG     0002    2       PRI
R1                    REG     0001    1       PRI
R0                    REG     0000    0       PRI

.XME file: \\Mac\Home\Desktop\Computer Architecture\Assembler\cexbra.xme
```

cexbra.xme (file used)

```
S00D00006365786272612E61736D0E
S205FFFFFFFFFE
S1171000DF4DA74D1250043CF967FA67FA7FFF3FF867F867E0
S9031000EC
```

Pseudo Code for the rest of the emulator

Implementation: **debugger.c** (updated to increase readability)

```
- Include Header file

DECLARE BreakpointValue AS unsigned short
DECLARE timecounter AS unsigned short

DECLARE buffer AS integer
DECLARE D0 <- 0 AS unsigned short
DECLARE E0 <- 0 AS unsigned short
DECLARE F0 AS unsigned short
DECLARE F1 AS unsigned short
DECLARE instructionnumber AS integer

FUNCTION ChangedRegistersValue(newcontent AS unsigned short, regnum AS integer):
    CONVERT newcontent TO Hex-string:
        sprintf(RegistersHexString[regnum], "%04X", newcontent)

    // Convert register binary to hex string
    FOR i <- 15 DOWNTO 0:
        RegistersBinaryString[regnum][i] <- (newcontent & 1) + '0'
        newcontent <- newcontent >> 1
    END FOR

    RETURN
END FUNCTION

FUNCTION view_registers():
    PRINT "Registers:"

    FOR i <- 0 TO 7:
        PRINT "R" + i + ": "
        PRINT "0b"

        FOR j <- 0 TO 15:
            PRINT RegistersBinaryString[i][j]
        END FOR

        PRINT " | 0x%04X", RegistersValue[i]
        PRINT "\n"
    END FOR

    // Print PSW content
    PRINT "PSW(vnzc): " + PSW.v + PSW.n + PSW.z + PSW.c
    RETURN
END FUNCTION

FUNCTION edit_registers():
    // Edit the registers content
    // User is prompted to enter the register number and the new content

    PRINT "Reg# (word in hex)= "
    DECLARE regnum AS integer
    SCAN("%d %hx", &regnum, &RegistersValue[regnum]) // Change register content
    Call ChangedRegistersValue(RegistersValue[regnum], regnum) // Update register forms
    RETURN
END FUNCTION

FUNCTION memory_printer():
    DECLARE memchoice AS char
    PRINT "select Memory I=IMEM D=DMEM B=both"
    SCAN(" %c", &memchoice)

    IF memchoice == 'I' OR memchoice == 'i' THEN:
        Call PrintMEM((unsigned char*)IMEM) // Prints instruction memory
    ELSE IF memchoice == 'D' OR memchoice == 'd' THEN:
        Call PrintMEM((unsigned char*)DMEM) // Prints data memory
    ELSE IF memchoice == 'B' OR memchoice == 'b' THEN:
        Call PrintMEM((unsigned char*)IMEM) // Prints instruction memory
        Call PrintMEM((unsigned char*)DMEM) // Prints data memory
    ELSE:
        PRINT "Invalid choice"
    END IF

    RETURN
END FUNCTION

FUNCTION edit_memory():
    DECLARE memchoice AS char
    PRINT "select Memory I=IMEM D=DMEM"
    SCAN(" %c", &memchoice)

    IF memchoice == 'I' OR memchoice == 'i' THEN:
        DECLARE address, content AS unsigned short
        PRINT "Enter(address content): %hx %hx", address, content
        IMEM[address / 2] <- content
    ELSE IF memchoice == 'D' OR memchoice == 'd' THEN:
        DECLARE address, content AS unsigned short
        PRINT "Enter(address content): %hx %hx", address, content
        DMEM[address / 2] <- content
    ELSE:
        PRINT "Invalid choice"
    END IF

    RETURN
END FUNCTION

FUNCTION run_debugger():
    PRINT "Clock\tPC\tInstruction\tFetch\t\tDecode\tExecute\n"
    DECLARE firstrun <- TRUE
    Call fetch() // Just to stop IMAR from being 0

    WHILE NOT (IMARValue == 0x0000 OR RegistersValue[PC] == BreakpointValue):
        Call fetch()
        F0 <- RegistersValue[PC] // Initialize F0
        F1 <- IMARValue // Initialize F1

        instructionnumber <- Call decode()

        RegistersValue[PC] <- RegistersValue[PC] + 2
        Call ChangedRegistersValue(RegistersValue[PC], PC)

        PRINT timecounter + "\t" + RegistersValue[PC] + "\t" + IMARValue + "\t\tF0:" + F0 + "\t\tD0:" + D0 + "\n"
        timecounter <- timecounter + 1

        PRINT timecounter + "\t\t \t\tF1:" + F1 + "\t\t\tE0:" + E0 + " \t"

        IF firstrun == TRUE THEN:
            buffer <- instructionnumber
            PRINT "\n"
            Call fetch() // Fetches next instruction F1
            D0 <- F1
            E0 <- D0
            firstrun <- FALSE
            timecounter <- timecounter + 1
        ELSE:
            IF ((TC == 0 AND FC == 0) OR buffer == CEX) THEN:
                Call execute(buffer) // Executes current instruction. (CEX off)
            ELSE:
                Call cex_enabled(buffer) // Handle executions accordingly
            END IF

            Call fetch() // Fetches next instruction F1
            D0 <- F1
            E0 <- D0
            buffer <- instructionnumber
            timecounter <- timecounter + 1
        END IF
    END WHILE

END FUNCTION

FUNCTION step_debugger():
    PRINT "Press 's' to step through instructions, 'q' to quit"
    PRINT "Clock\tPC\tInstruction\tFetch\t\tDecode\tExecute\n"
    DECLARE firstrun <- TRUE
    Call fetch() // Just to stop IMAR from being 0
    DECLARE choice <- 's'

    WHILE NOT (IMARValue == 0x0000 OR RegistersValue[PC] == BreakpointValue OR choice == 'q'):
        Call fetch()
        F0 <- RegistersValue[PC] // Initialize F0
        F1 <- IMARValue // Initialize F1

        instructionnumber <- Call decode()

        RegistersValue[PC] <- RegistersValue[PC] + 2
        Call ChangedRegistersValue(RegistersValue[PC], PC)

        PRINT timecounter + "\t" + RegistersValue[PC] + "\t" + IMARValue + "\t\tF0:" + F0 + "\t\tD0:" + D0 + "   "
        SCAN(" %c", &choice)
        IF choice == 'q' THEN:
            RETURN
        END IF

        timecounter <- timecounter + 1

        PRINT timecounter + "\t\t \t\tF1:" + F1 + "\t\t\tE0:" + E0 + "\t      "
        SCAN(" %c", &choice)
        IF choice == 'q' THEN:
            RETURN
        END IF

        IF firstrun == TRUE THEN:
            buffer <- instructionnumber
            Call fetch()
            D0 <- F1
            E0 <- D0
            firstrun <- FALSE
            timecounter <- timecounter + 1
        ELSE:
            IF ((TC == 0 AND FC == 0) OR buffer == CEX) THEN:
                Call execute(buffer)
            ELSE:
                Call cex_enabled(buffer)
            END IF

            Call fetch()
            D0 <- F1
            E0 <- D0
            buffer <- instructionnumber
            timecounter <- timecounter + 1
        END IF
    END WHILE

    RETURN
END FUNCTION

FUNCTION add_breakpoint():
    PRINT "Enter breakpoint value: "
    SCAN("%hx", &BreakpointValue)
    PRINT "Breakpoint set at: " + BreakpointValue
    RETURN
END FUNCTION

FUNCTION print_time():
    PRINT "Time Count: " + timecounter
    RETURN
END FUNCTION
```

Implementation: **fetch.c** (updated to increase readability)

```
FUNCTION Fetch:
                IMARValue <- IMEM[PC]
END FUNCTION
```

Implementation: **loader.c** (same as A4)

- Include Header file

```
FUNCTION ProcessSRecords(filename AS const char*):
    DECLARE byte, data AS unsigned int

    DECLARE file AS FILE* <- fopen(filename, "r")
    IF NOT file THEN:
        Call perror("Error opening file")
        RETURN
    END IF

    DECLARE line AS char[MAX_S_RECORD_SIZE]
    WHILE fgets(line, sizeof(line), file) DO:
        IF line[0] != 'S' THEN: // Not an S-Record
            CONTINUE
        END IF

        DECLARE count, address AS integer
        sscanf(line + 2, "%2x%4x", &count, &address) // Read count and address
        DECLARE dataLength AS integer <- (count - 3) * BYTE_SIZE

        DECLARE calculatedChecksum AS unsigned int <- Call calculateChecksum(line, count, dataLength) // Checksum test
        DECLARE givenChecksum AS unsigned int
        sscanf(line + 2 + count * BYTE_SIZE, "%2x", &givenChecksum) // Checksum done before storing

        IF calculatedChecksum != givenChecksum THEN:
            PRINT "Checksum error in line: " + line
            CONTINUE
        END IF

        IF line[1] == '0' THEN: // S0 record processing
            FOR i <- 0 TO dataLength - 1 STEP BYTE_SIZE DO:
                sscanf(line + HEADER_START + i, "%2x", &byte)
                PRINT "%c", byte
            END FOR
            PRINT " was loaded successfully" // Print successful loading message
        ELSE IF line[1] == '1' THEN: // S1 record processing
            FOR i <- 0 TO dataLength - 1 STEP ASCII_SIZE DO:
                sscanf(line + HEADER_START + i, "%4x", &data)
                IMEM[(address >> MEM_SHIFT) + (i >> BYTE_SIZE)] <- (data >> DATA_SHIFT) | ((data & BYTE_MASK) << DATA_SHIFT) // Correctly
handle high and low byte
            END FOR
        ELSE IF line[1] == '2' THEN: // S2 record processing
            FOR i <- 0 TO dataLength - 1 STEP ASCII_SIZE DO:
                sscanf(line + HEADER_START + i, "%4x", &data)
                DMEM[(address >> MEM_SHIFT) + (i >> BYTE_SIZE)] <- (data >> DATA_SHIFT) | ((data & BYTE_MASK) << DATA_SHIFT) // Correctly
handle high and low byte
            END FOR
        ELSE IF line[1] == '9' THEN: // S9 record processing
            sscanf(line + 4, "%4x", &address) // Read starting address
            PRINT "Starting address: %04X", address // Print starting address
            RegistersValue[PC] <- address // Set PC to starting address
        END IF
    END WHILE

    fclose(file)
END FUNCTION

FUNCTION PrintMEM(MEM AS unsigned char*):
    DECLARE Range_Start, Range_End AS unsigned short

    PRINT "Enter Memory Range in HEX:"
    SCANF("%x %x", (unsigned int*) &Range_Start, (unsigned int*) &Range_End)

    FOR i <- Range_Start TO Range_End - 1 STEP BYTES_PER_LINE DO: // Loop from the start-end of mem[range]
        PRINT "%04X: ", i // Print address

        FOR j <- 0 TO BYTES_PER_LINE - 1 DO: // Loop through each byte in the current memory
            IF i + j < Range_End THEN: // Print if byte is within range
                PRINT "%02X ", MEM[i + j]
            ELSE:
                PRINT "   "
            END IF
        END FOR
        PRINT " "
        FOR j <- 0 TO BYTES_PER_LINE - 1 DO: // Loop through each byte in the current memory word
            IF i + j < Range_End THEN: // If the current byte is within the memory range
                DECLARE c AS unsigned char <- MEM[i + j]
                IF c >= ASCII_START_VALUE AND c <= ASCII_END_VALUE THEN: // If within ASCII range - print
                    PRINT "%c", c
                ELSE:
                    PRINT "."
                END IF
            ELSE: // Current byte is outside the memory range
                PRINT " "
            END IF
        END FOR
        PRINT "\n" // Separates words in memory
    END FOR
    PRINT "\n"

    Call getchar()
END FUNCTION

FUNCTION calculateChecksum(line AS const char*, count AS integer, dataLength AS integer) -> unsigned int:
    DECLARE checksum AS unsigned int <- 0
    FOR i <- 2 TO HEADER_START - 1 STEP BYTE_SIZE DO: // Add size and address bytes
        DECLARE byte AS unsigned int
        sscanf(line + i, "%2x", &byte)
        checksum <- checksum + byte
    END FOR

    FOR i <- HEADER_START TO HEADER_START + dataLength - 1 STEP BYTE_SIZE DO: // Add data bytes
        DECLARE byte AS unsigned int
        sscanf(line + i, "%2x", &byte)
        checksum <- checksum + byte
    END FOR

    checksum <- (~checksum) & 0xFF // Check if the checksum is valid
    RETURN checksum
END FUNCTION
```

Implementation: **xm23p.c** (updated to increase readability)

```
- Include Header file
DECLARE timecounter AS unsigned short // Time counter

FUNCTION InitializeCPU():
    Call memset(IMEM, 0, sizeof(IMEM)) // Initializing IMEM to 0s
    Call memset(DMEM, 0, sizeof(DMEM)) // Initializing DMEM to 0s

    FOR i <- 0 TO 7 DO:
        FOR j <- 0 TO 15 DO:
            RegistersBinaryString[i][j] <- '0'
        END FOR
        FOR j <- 0 TO 3 DO:
            RegistersHexString[i][j] <- '0'
        END FOR
    END FOR

    timecounter <- 0 // Initializing time counter to 0
END FUNCTION

FUNCTION PrintMenuOptions():
    PRINT "===========MENU==========="
    PRINT "l - load file"
    PRINT "v - view registers"
    PRINT "e - edit register values(word)"
    PRINT "m - display memory"
    PRINT "c - change in memory"
    PRINT "r - run (normal)"
    PRINT "s - step (normal)"
    PRINT "b - breakpoint"
    PRINT "x - step x times"
    PRINT "1- run (debug)"
    PRINT "2- step (debug)"
    PRINT "3- view time count"
    PRINT "q - quit"
END FUNCTION
```

Implementation: **psw.c** (updated to increase readability)

```
- Include Header file

FUNCTION updatePSW(src AS unsigned int, dst AS unsigned int, tempResult AS unsigned int, wbbuff AS integer):
    // Clear the PSW flags
    PSW.z <- CLEAR
    PSW.n <- CLEAR
    PSW.c <- CLEAR
    PSW.v <- CLEAR // Check if this should be cleared too

    // Check if the operation was on a byte or word
    DECLARE mask AS integer <- (wbbuff == WORD) ? 0xFFFF : 0xFF

    // Update the Zero (Z) flag
    IF (tempResult & mask) == CLEAR THEN:
        PSW.z <- SET
    END IF
    // Update the Negative (N) flag
    IF (tempResult & ((mask + SET) >> SET)) != CLEAR THEN:
        PSW.n <- SET
    END IF

    // Update the Carry (C) flag - In the context of CMP, carry flag is set if src < dst
    IF src < dst THEN:
        PSW.c <- SET
    END IF
END FUNCTION
```

Implementation: **BRANCHinstructions.c** (updated to increase readability)

- Include Header file

```
FUNCTION bl():
    RegistersValue[LR] <- RegistersValue[PC] // Make the link register equal to the PC
    RegistersValue[PC] <- RegistersValue[PC] + offsetbuff // Increment the PC by offset
    RETURN
END FUNCTION

FUNCTION beqbz():
    IF PSW.z == TRUE THEN:
        RegistersValue[PC] <- RegistersValue[PC] + offsetbuff // Increment the PC by offset
    END IF
    RETURN
END FUNCTION

FUNCTION bnebnz():
    IF PSW.z == FALSE THEN:
        RegistersValue[PC] <- RegistersValue[PC] + offsetbuff // Increment the PC by offset
    END IF
    RETURN
END FUNCTION

FUNCTION bcbhs():
    IF PSW.c == TRUE THEN:
        RegistersValue[PC] <- RegistersValue[PC] + offsetbuff // Increment the PC by offset
    END IF
    RETURN
END FUNCTION

FUNCTION bncblo():
    IF PSW.c == FALSE THEN:
        RegistersValue[PC] <- RegistersValue[PC] + offsetbuff // Increment the PC by offset
    END IF
    RETURN
END FUNCTION

FUNCTION bn():
    IF PSW.n == TRUE THEN:
        RegistersValue[PC] <- RegistersValue[PC] + offsetbuff // Increment the PC by offset
    END IF
    RETURN
END FUNCTION

FUNCTION bge():
    IF PSW.n == PSW.v THEN:
        RegistersValue[PC] <- RegistersValue[PC] + offsetbuff // Increment the PC by offset
    END IF
    RETURN
END FUNCTION

FUNCTION blt():
    IF PSW.n != PSW.v THEN:
        RegistersValue[PC] <- RegistersValue[PC] + offsetbuff // Increment the PC by offset
    END IF
    RETURN
END FUNCTION

FUNCTION bra():
    RegistersValue[PC] <- RegistersValue[PC] + offsetbuff // Increment the PC by offset
    RETURN
END FUNCTION
```

Implementation: **MEMinstructions.c** (updated to increase readability)

- Include Header file

```
FUNCTION ld():
    IF prpobuff != POST THEN: // PRE-INC/DEC
        IF incbuff == SET THEN: // PRE-INC
            RegistersValue[srcbuff] <- RegistersValue[srcbuff] + (wbbuff == WORD ? 2 : 1)
        END IF
        IF decbuff == SET THEN: // PRE-DEC
            RegistersValue[srcbuff] <- RegistersValue[srcbuff] - (wbbuff == WORD ? 2 : 1)
        END IF
        IF wbbuff == WORD THEN:
            RegistersValue[dstbuff] <- DMEM[RegistersValue[srcbuff] / 2]
        ELSE:
            // Preserve the high byte of the destination register and only load the low byte from DMEM
            DECLARE byteVal AS unsigned short
            IF RegistersValue[srcbuff] % 2 == 0 THEN:
                byteVal <- DMEM[RegistersValue[srcbuff] / 2] & 0x00FF // Low byte
            ELSE:
                byteVal <- (DMEM[RegistersValue[srcbuff] / 2] & 0xFF00) >> 8 // High byte
            END IF
            RegistersValue[dstbuff] <- (RegistersValue[dstbuff] & 0xFF00) | byteVal
        END IF
    ELSE: // POST-INC/DEC
        IF wbbuff == WORD THEN:
            RegistersValue[dstbuff] <- DMEM[RegistersValue[srcbuff] / 2]
        ELSE:
            // Preserve the high byte of the destination register and only load the low byte from DMEM
            DECLARE byteVal AS unsigned short
            IF RegistersValue[srcbuff] % 2 == 0 THEN:
                byteVal <- DMEM[RegistersValue[srcbuff] / 2] & 0x00FF // Low byte
            ELSE:
                byteVal <- (DMEM[RegistersValue[srcbuff] / 2] & 0xFF00) >> 8 // High byte
            END IF
            RegistersValue[dstbuff] <- (RegistersValue[dstbuff] & 0xFF00) | byteVal
        END IF

        IF incbuff == SET THEN: // POST-INC
            RegistersValue[srcbuff] <- RegistersValue[srcbuff] + (wbbuff == WORD ? 2 : 1)
        END IF
        IF decbuff == SET THEN: // POST-DEC
            RegistersValue[srcbuff] <- RegistersValue[srcbuff] - (wbbuff == WORD ? 2 : 1)
        END IF
    END IF
    RETURN
END FUNCTION

FUNCTION st():
    IF prpobuff != POST THEN: // PRE-INC/DEC
        IF incbuff == SET THEN: // PRE-INC
            RegistersValue[dstbuff] <- RegistersValue[dstbuff] + (wbbuff == WORD ? 2 : 1)
        END IF
        IF decbuff == SET THEN: // PRE-DEC
            RegistersValue[dstbuff] <- RegistersValue[dstbuff] - (wbbuff == WORD ? 2 : 1)
        END IF
        IF wbbuff == WORD THEN:
            DMEM[RegistersValue[dstbuff] / 2] <- RegistersValue[srcbuff] // Store word
        ELSE:
            DECLARE wordVal AS unsigned short <- DMEM[RegistersValue[dstbuff] / 2]
            IF RegistersValue[dstbuff] % 2 == 0 THEN:
                // Store to low byte
                wordVal <- (wordVal & 0xFF00) | (RegistersValue[srcbuff] & 0x00FF)
            ELSE:
                // Store to high byte
                wordVal <- (wordVal & 0x00FF) | ((RegistersValue[srcbuff] & 0x00FF) << 8)
            END IF
            DMEM[RegistersValue[dstbuff] / 2] <- wordVal
        END IF
    ELSE: // POST-INC/DEC
        IF wbbuff == WORD THEN:
            DMEM[RegistersValue[dstbuff] / 2] <- RegistersValue[srcbuff] // Store word
        ELSE:
            DECLARE wordVal AS unsigned short <- DMEM[RegistersValue[dstbuff] / 2]
            IF RegistersValue[dstbuff] % 2 == 0 THEN:
                // Store to low byte
                wordVal <- (wordVal & 0xFF00) | (RegistersValue[srcbuff] & 0x00FF)
            ELSE:
                // Store to high byte
                wordVal <- (wordVal & 0x00FF) | ((RegistersValue[srcbuff] & 0x00FF) << 8)
            END IF
            DMEM[RegistersValue[dstbuff] / 2] <- wordVal
        END IF

        IF incbuff == SET THEN: // POST-INC
            RegistersValue[dstbuff] <- RegistersValue[dstbuff] + (wbbuff == WORD ? 2 : 1)
        END IF
        IF decbuff == SET THEN: // POST-DEC
            RegistersValue[dstbuff] <- RegistersValue[dstbuff] - (wbbuff == WORD ? 2 : 1)
        END IF
    END IF
    RETURN
END FUNCTION

FUNCTION ldr():
    EA <- RegistersValue[srcbuff] + (offsetbuff / 2) // Get the Effective address
    IF wbbuff == WORD THEN:
        RegistersValue[dstbuff] <- DMEM[EA / 2] // DST shouldn't change
    ELSE: // BYTE
        RegistersValue[dstbuff] <- DMEM[EA / 2] & 0x00FF // DST shouldn't change
    END IF
END FUNCTION

FUNCTION str():
    EA <- RegistersValue[dstbuff] + offsetbuff // Get the Effective address
    IF wbbuff == WORD THEN:
        DMEM[EA / 2] <- RegistersValue[srcbuff] // Store the word
    ELSE: // BYTE
        DMEM[EA / 2] <- RegistersValue[srcbuff] & 0x00FF // DST shouldn't change
    END IF

    RETURN
END FUNCTION
```

Implementation: **REGinstructions.c** (updated to increase readability)

- Include Header file

```
FUNCTION add(): // Add instruction
    IF wbbuff == WORD THEN: // Word addition
        RegistersValue[dstbuff] <- RegistersValue[srcbuff] + RegistersValue[dstbuff] // Adding words
    ELSE: // Byte addition
        DECLARE srcLowByte AS unsigned short <- RegistersValue[srcbuff] & 0x00FF
        DECLARE dstLowByte AS unsigned short <- RegistersValue[dstbuff] & 0x00FF
        DECLARE result AS unsigned short <- srcLowByte + dstLowByte // Adding bytes
        RegistersValue[dstbuff] <- (RegistersValue[dstbuff] & 0xFF00) | (result & 0x00FF)
    END IF
    Call updatePSW(RegistersValue[srcbuff], RegistersValue[dstbuff], RegistersValue[dstbuff], wbbuff) // Update PSW flags
    RETURN
END FUNCTION

FUNCTION addc():
    IF wbbuff == WORD THEN: // Word addition
        RegistersValue[dstbuff] <- RegistersValue[srcbuff] + RegistersValue[dstbuff] + PSW.c // Adding words and carry
    ELSE: // Byte addition
        DECLARE srcLowByte AS unsigned short <- RegistersValue[srcbuff] & 0x00FF
        DECLARE dstLowByte AS unsigned short <- RegistersValue[dstbuff] & 0x00FF
        DECLARE result AS unsigned short <- srcLowByte + dstLowByte + PSW.c // Adding bytes and carry
        RegistersValue[dstbuff] <- (RegistersValue[dstbuff] & 0xFF00) | (result & 0x00FF)
    END IF
    RETURN
END FUNCTION

FUNCTION sub():
    IF wbbuff == WORD THEN: // Word subtraction
        RegistersValue[dstbuff] <- RegistersValue[dstbuff] + (~RegistersValue[srcbuff] + 1)
    ELSE: // Byte subtraction
        DECLARE srcLowByte AS unsigned short <- RegistersValue[srcbuff] & 0x00FF
        DECLARE dstLowByte AS unsigned short <- RegistersValue[dstbuff] & 0x00FF
        DECLARE result AS unsigned short <- dstLowByte + (~srcLowByte + 1)
        RegistersValue[dstbuff] <- (RegistersValue[dstbuff] & 0xFF00) | (result & 0x00FF)
    END IF
    RETURN
END FUNCTION

FUNCTION subc():
    IF wbbuff == WORD THEN: // Word subtraction
        RegistersValue[dstbuff] <- RegistersValue[dstbuff] + (~RegistersValue[srcbuff] + PSW.c)
    ELSE: // Byte subtraction
        DECLARE srcLowByte AS unsigned short <- RegistersValue[srcbuff] & 0x00FF
        DECLARE dstLowByte AS unsigned short <- RegistersValue[dstbuff] & 0x00FF
        DECLARE result AS unsigned short <- dstLowByte + (~srcLowByte + PSW.c)
        RegistersValue[dstbuff] <- (RegistersValue[dstbuff] & 0xFF00) | (result & 0x00FF)
    END IF
    RETURN
END FUNCTION

FUNCTION dadd():
    IF wbbuff == WORD THEN: // Word addition
        RegistersValue[dstbuff] <- RegistersValue[srcbuff] + RegistersValue[dstbuff] + PSW.c // Adding words and carry
    ELSE: // Byte addition
        DECLARE srcLowByte AS unsigned short <- RegistersValue[srcbuff] & 0x00FF
        DECLARE dstLowByte AS unsigned short <- RegistersValue[dstbuff] & 0x00FF
        DECLARE result AS unsigned short <- srcLowByte + dstLowByte + PSW.c // Adding bytes and carry
        RegistersValue[dstbuff] <- (RegistersValue[dstbuff] & 0xFF00) | (result & 0x00FF)
    END IF
    RETURN
END FUNCTION

FUNCTION cmp():
    DECLARE Temp_Destination AS unsigned short

    IF rcbuff == SET THEN: // Constant in src comparison
        DECLARE constantmask AS unsigned short <- Call handleConstant(srcbuff)
        IF wbbuff == WORD THEN: // Word comparison
            Temp_Destination <- RegistersValue[dstbuff] + ~constantmask + 1
        ELSE: // Byte comparison
            DECLARE srcLowByte AS unsigned short <- constantmask & 0x00FF
            DECLARE dstLowByte AS unsigned short <- RegistersValue[dstbuff] & 0x00FF
            Temp_Destination <- dstLowByte + ~srcLowByte + 1
        END IF
        Call updatePSW(constantmask, RegistersValue[dstbuff], Temp_Destination, wbbuff)

    ELSE: // Register in src comparison
```

```
        IF wbbuff == WORD THEN: // Word comparison
            Temp_Destination <- RegistersValue[dstbuff] + ~RegistersValue[srcbuff] + 1
        ELSE: // Byte comparison
            DECLARE srcLowByte AS unsigned short <- RegistersValue[srcbuff] & 0x00FF
            DECLARE dstLowByte AS unsigned short <- RegistersValue[dstbuff] & 0x00FF
            Temp_Destination <- dstLowByte + ~srcLowByte + 1
        END IF
        Call updatePSW(RegistersValue[srcbuff], RegistersValue[dstbuff], Temp_Destination, wbbuff)
    END IF
    RETURN
END FUNCTION

FUNCTION xor():
    IF wbbuff == WORD THEN: // If operation is word-wide
        RegistersValue[dstbuff] <- RegistersValue[dstbuff] ^ RegistersValue[srcbuff]
    ELSE: // If operation is byte-wide
        DECLARE srcLowByte AS unsigned short <- RegistersValue[srcbuff] & 0x00FF
        DECLARE dstLowByte AS unsigned short <- RegistersValue[dstbuff] & 0x00FF
        DECLARE result AS unsigned short <- srcLowByte ^ dstLowByte // XOR operation on the low bytes
        RegistersValue[dstbuff] <- (RegistersValue[dstbuff] & 0xFF00) | (result & 0x00FF)
    END IF

    Call updatePSW(RegistersValue[srcbuff], RegistersValue[dstbuff], RegistersValue[dstbuff], wbbuff)

    RETURN
END FUNCTION

FUNCTION and():
    IF wbbuff == WORD THEN: // If operation is word-wide
        RegistersValue[dstbuff] <- RegistersValue[dstbuff] & RegistersValue[srcbuff]
    ELSE: // If operation is byte-wide
        DECLARE srcLowByte AS unsigned short <- RegistersValue[srcbuff] & 0x00FF
        DECLARE dstLowByte AS unsigned short <- RegistersValue[dstbuff] & 0x00FF
        DECLARE result AS unsigned short <- srcLowByte & dstLowByte // AND operation on the low bytes
        RegistersValue[dstbuff] <- (RegistersValue[dstbuff] & 0xFF00) | (result & 0x00FF)
    END IF

    Call updatePSW(RegistersValue[srcbuff], RegistersValue[dstbuff], RegistersValue[dstbuff], wbbuff)
    RETURN
END FUNCTION

FUNCTION or():
    IF wbbuff == WORD THEN: // If operation is word-wide
        RegistersValue[dstbuff] <- RegistersValue[dstbuff] | RegistersValue[srcbuff]
    ELSE: // If operation is byte-wide
        DECLARE srcLowByte AS unsigned short <- RegistersValue[srcbuff] & 0x00FF
        DECLARE dstLowByte AS unsigned short <- RegistersValue[dstbuff] & 0x00FF
        DECLARE result AS unsigned short <- srcLowByte | dstLowByte // OR operation on the low bytes
        RegistersValue[dstbuff] <- (RegistersValue[dstbuff] & 0xFF00) | (result & 0x00FF)
    END IF

    Call updatePSW(RegistersValue[srcbuff], RegistersValue[dstbuff], RegistersValue[dstbuff], wbbuff)
    RETURN
END FUNCTION

FUNCTION bit():
    DECLARE Temp_Destination AS unsigned short

    IF wbbuff == WORD THEN: // If operation is word-wide
        Temp_Destination <- RegistersValue[dstbuff] & (1 << RegistersValue[srcbuff])
    ELSE: // If operation is byte-wide
        Temp_Destination <- (RegistersValue[dstbuff] & 0x00FF) & (1 << (RegistersValue[srcbuff] & 0x00FF))
    END IF
    PRINT "R/C: " + rcbuff + "\nW/B: " + wbbuff + "\nSource: " + RegistersValue[srcbuff] + "\nDestination: " + RegistersValue[dstbuff]
    Call updatePSW(RegistersValue[srcbuff], RegistersValue[dstbuff], Temp_Destination, wbbuff)
    RETURN
END FUNCTION

FUNCTION bic():
  IF wbbuff == WORD THEN: // If operation is word-wide
    RegistersValue[dstbuff] <- RegistersValue[dstbuff] & ~(1 << RegistersValue[srcbuff])
  ELSE: // If operation is byte-wide
    RegistersValue[dstbuff] <- RegistersValue[dstbuff] & ~(1 << (RegistersValue[srcbuff] & 0x00FF))
  END IF
  Call updatePSW(RegistersValue[srcbuff], RegistersValue[dstbuff], RegistersValue[dstbuff], wbbuff)
  RETURN
END FUNCTION

FUNCTION bis():
  IF wbbuff == WORD THEN: // If operation is word-wide
    RegistersValue[dstbuff] <- RegistersValue[dstbuff] | (1 << RegistersValue[srcbuff])
  ELSE: // If operation is byte-wide
    RegistersValue[dstbuff] <- RegistersValue[dstbuff] | (1 << (RegistersValue[srcbuff] & 0x00FF))
  END IF
  Call updatePSW(RegistersValue[srcbuff], RegistersValue[dstbuff], RegistersValue[dstbuff], wbbuff)
  RETURN
END FUNCTION

FUNCTION mov():
  IF wbbuff == WORD THEN: // If operation is word-wide
    RegistersValue[dstbuff] <- RegistersValue[srcbuff]
  ELSE: // If operation is byte-wide
    // Clear the destination's low byte and set it to the source's low byte
    RegistersValue[dstbuff] <- (RegistersValue[dstbuff] & 0xFF00) | (RegistersValue[srcbuff] & 0x00FF)
  END IF
  Call updatePSW(RegistersValue[srcbuff], RegistersValue[dstbuff], RegistersValue[dstbuff], wbbuff)
  RETURN
END FUNCTION

FUNCTION swap():
  DECLARE temp_reg AS unsigned short <- RegistersValue[dstbuff]
  RegistersValue[dstbuff] <- RegistersValue[srcbuff]
  RegistersValue[srcbuff] <- temp_reg
  Call updatePSW(RegistersValue[srcbuff], RegistersValue[dstbuff], 0, wbbuff) // Assuming PSW needs to be updated, adjust as necessary
  RETURN

END FUNCTION

FUNCTION sra():
  IF wbbuff == WORD THEN: // Word operation
    DECLARE msb AS unsigned short <- RegistersValue[dstbuff] & 0x8000 // Extract MSB (bit 15)
    RegistersValue[dstbuff] <- RegistersValue[dstbuff] >> 1 // Logical shift right by 1
    RegistersValue[dstbuff] <- RegistersValue[dstbuff] | msb // Preserve MSB (sign bit for arithmetic shift)
  ELSE:
    // Byte operation
    DECLARE msb AS unsigned char <- RegistersValue[dstbuff] & 0x80 // Extract MSB (bit 7) of the low byte
    RegistersValue[dstbuff] <- (RegistersValue[dstbuff] & 0xFF00) | ((RegistersValue[dstbuff] & 0x00FF) >> 1) // Logical shift right the low byte by 1
    RegistersValue[dstbuff] <- RegistersValue[dstbuff] | msb // Preserve MSB (sign bit for arithmetic shift) in the low byte
  END IF
END FUNCTION

FUNCTION rrc():
  DECLARE oldLSB AS unsigned short
  IF wbbuff == WORD THEN:
    // Word operation
    oldLSB <- RegistersValue[dstbuff] & 0x0001 // Extract old LSB
    RegistersValue[dstbuff] <- RegistersValue[dstbuff] >> 1 // Shift right
    IF PSW.c THEN: // If carry is set
      RegistersValue[dstbuff] <- RegistersValue[dstbuff] | 0x8000 // Put carry into MSB
    END IF
    PSW.c <- oldLSB // Update carry with old LSB
  ELSE:
    // Byte operation
    oldLSB <- RegistersValue[dstbuff] & 0x01 // Extract old LSB of the low byte
    RegistersValue[dstbuff] <- (RegistersValue[dstbuff] & 0xFF00) | ((RegistersValue[dstbuff] & 0x00FF) >> 1) // Shift right the low byte
    IF PSW.c THEN: // If carry is set
      RegistersValue[dstbuff] <- RegistersValue[dstbuff] | 0x80 // Put carry into MSB of the low byte
    END IF
```

```
        PSW.c <- oldLSB // Update carry with old LSB
    END IF
    RETURN
END FUNCTION

FUNCTION swpb():
    RegistersValue[dstbuff] <- ((RegistersValue[dstbuff] & 0x00FF) << 8) | ((RegistersValue[dstbuff] & 0xFF00) >> 8) // Swap the high byte and low byte
    PRINT "SWPB: DST:" + dstbuff + " Result: " + RegistersValue[dstbuff]
    RETURN
END FUNCTION

FUNCTION sxt():
    IF RegistersValue[dstbuff] & 0x0080 THEN: // If bit 7 of the low byte is 1
        RegistersValue[dstbuff] <- RegistersValue[dstbuff] | 0xFF00 // Set bits 8-15 to 1 for sign extension
    ELSE:
        RegistersValue[dstbuff] <- RegistersValue[dstbuff] & 0x00FF // Keep bits 8-15 as 0
    END IF
    RETURN
END FUNCTION

FUNCTION setcc():
    IF vbuff == TRUE THEN:
        PSW.v <- TRUE
    END IF
    IF nbuff == TRUE THEN:
        PSW.n <- TRUE
    END IF
    IF zbuff == TRUE THEN:
        PSW.z <- TRUE
    END IF
    IF cbuff == TRUE THEN:
        PSW.c <- TRUE
    END IF
    RETURN
END FUNCTION

FUNCTION clrcc():
    IF vbuff == TRUE THEN:
        PSW.v <-
        PSW.v <- FALSE
    END IF
    IF nbuff == TRUE THEN:
        PSW.n <- FALSE
    END IF
    IF zbuff == TRUE THEN:
        PSW.z <- FALSE
    END IF
    IF cbuff == TRUE THEN:
        PSW.c <- FALSE
    END IF
    RETURN
END FUNCTION

FUNCTION movl():
    RegistersValue[dstbuff] <- (RegistersValue[dstbuff] & 0xFF00) | bitsbuff // Set the low byte to bitsbuff, keep the high byte unchanged
    RETURN
END FUNCTION

FUNCTION movlz():
    RegistersValue[dstbuff] <- bitsbuff // Set the low byte to bitsbuff and high byte to 0
    RETURN
END FUNCTION

FUNCTION movls():
    RegistersValue[dstbuff] <- bitsbuff | 0xFF00 // Set the low byte to bitsbuff and high byte to 0xFF
    RETURN
END FUNCTION

FUNCTION movh():
    RegistersValue[dstbuff] <- (RegistersValue[dstbuff] & 0x00FF) | (bitsbuff << 8) // Set the high byte to bitsbuff, keep the low byte unchanged
    RETURN
END FUNCTION
```

Main: (same as A4, increased readability)

- Include Header file

```
FUNCTION main(argc, argv):

    DECLARE choice, memchoice, filename[MAX_FILENAME_LEN]
    DECLARE firststep <- 0

    // Initialize CPU
    Call InitializeCPU()

    WHILE choice != 'q':

        Call PrintMenuOptions()

        SCAN(" %c", &choice)

        SWITCH choice:

            CASE 'l':
                PRINT "Enter filename: "
                SCAN("%s", filename)
                Call ProcessSRecords(filename)
            CASE 'v':
                Call view_registers()
            CASE 'e':
                Call view_registers()
            CASE 'm':
                Call memory_printer()
            CASE 'c':
                Call edit_memory()
            CASE 'r':
                PRINT "starting address: %04X\n", RegistersValue[PC]
                Call run()
            CASE 's':
                Call step()
            CASE 'b':
                Call add_breakpoint()
            CASE 'x':
                Call step_x_times()
            CASE '1':
                Call run_debugger()
            CASE '2':
                Call step_debugger()
            CASE '3':
                Call print_time()
            CASE 'q':
                RETURN 0
            DEFAULT:
                PRINT "Invalid choice, try again!"
        END SWITCH
    END WHILE
    RETURN 0
END FUNCTION
```