Assignment 4
XM23p Transfer of Control (Branching) Design, Implementation and Testing Document

Prepared for: Dr. Larry Hughes

Abdulla Sadoun B00900541
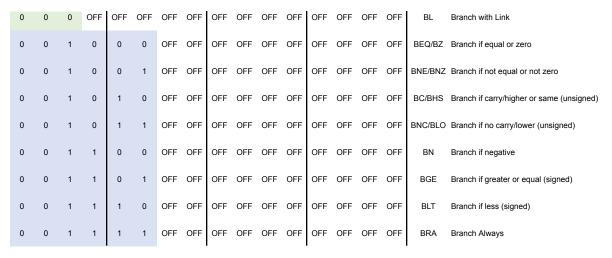
Table of Contents

Problem Introduction

Statement of Purpose

The purpose of this assignment is to design the implementation of an extension to the emulator program written in C, that would emulate the XM23p's execution of the instructions in charge of changin the transfer of control or branching. These functions serve a cruicial purpose as they are necessary to implement conditionals, comparison and looping which are vital concepts when it comes to computer programming. The descriptions and format for these branching instructions yet to be implemented are highlighted below in the following figure.

0	0	0	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	BL	Branch with Link
0	0	1	0	0	0	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	BEQ/BZ	Branch if equal or zero
0	0	1	0	0	1	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	BNE/BNZ	Branch if not equal or not zero
0	0	1	0	1	0	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	BC/BHS	Branch if carry/higher or same (unsigned)
0	0	1	0	1	1	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	BNC/BLO	Branch if no carry/lower (unsigned)
0	0	1	1	0	0	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	BN	Branch if negative
0	0	1	1	0	1	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	BGE	Branch if greater or equal (signed)
0	0	1	1	1	0	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	BLT	Branch if less (signed)
0	0	1	1	1	1	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	BRA	Branch Always

As well as this table which highlights Program status register (PSW) changes for each instruction:

Mnemonic	Instruction	PSW bits to inspect			
		V	N	Z	C
BL	Branch with Link	-	-	-	-
BEQ/BZ	Branch if equal or zero	-	-	= 1	-
BNE/BNZ	Branch if not equal or not zero	-	-	= 0	-
BC/BHS	Branch if carry/higher or same (unsigned)	-	-	-	= 1
BNC/BLO	Branch if no carry/lower (unsigned)	-	-	-	= 0
BN	Branch if negative	-	= 1	-	-
BGE	Branch if greater or equal (signed)	N ⊕ !V		-	-
BLT	Branch if less (signed)	N ⊕ V		-	-
BRA	Branch Always	-	-	-	-

Objectives

The objective of this assignment is to design, implement and test the BL-BRA 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 branching instructions as an extension to the loader, debugger, IMEM and DMEM instructions designed, implemented and tested in the previous labs and assignments.

The initial loader part section of the code will be disregarded and ignored for this assignment.

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]

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: fetch.c (same as earlier assignments)

- Include Header file
FUNCTION Process Instruction:

```
            WHILE (TRUE)
            Call function fetch

                  Call function decode

                  Call function execute (decode value/internal execute number)
                  Call function fetch
                        IF instruction is not equal to 0000 and PC is not equal to breakpoint)
                              BREAK
                        End IF
                  END WHILE
END FUNCTION


FUNCTION Fetch:
      IMAR = IMEM[PC]
END FUNCTION
```

Implementation: loader.c (NEW implementation due to changed IMEM & DMEM)

```
FUNCTION ProcessSRecords (filename):
DECLARE byte AS unsigned int
DECLARE data AS unsigned int

SET file = fopen(filename, "r")
IF file equals NULL:
   PRINT "Error opening file"
   RETURN
END IF

DECLARE line AS char[MAX_S_RECORD_SIZE]
WHILE fgets(line, sizeof(line), file) is not NULL:
   IF line[0] not equals 'S':
      CONTINUE  // Not an S-Record
   END IF

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

   SET calculatedChecksum = calculateChecksum(line, count, dataLength)  // Check sum test
   DECLARE givenChecksum AS unsigned int
```

```
    sscanf(line + 2 + count * BYTE_SIZE, "%2x", &givenChecksum)  // Checksum done before
storing

    IF calculatedChecksum not equals givenChecksum:
        PRINT "Checksum error in line:", line
        CONTINUE
    END IF

    SWITCH (line[1]):
        CASE '0':  // S0 record processing
            FOR i = 0 TO dataLength STEP BYTE_SIZE:
                sscanf(line + HEADER_START + i, "%2x", &byte)
                PRINT byte AS char
            END FOR
            PRINT " was loaded successfully"
            BREAK

        CASE '1':  // S1 record processing
            FOR i = 0 TO dataLength STEP ASCII_SIZE:
                sscanf(line + HEADER_START + i, "%4x", &data)
                SET IMEM[(address >> MEM_SHIFT) + (i >> BYTE_SIZE)] = (data >> DATA_SHIFT)
OR ((data AND BYTE_MASK) << DATA_SHIFT)  // Correctly handle high and low byte
            END FOR
            BREAK

        CASE '2':  // S2 record processing
            FOR i = 0 TO dataLength STEP ASCII_SIZE:
                sscanf(line + HEADER_START + i, "%4x", &data)
                SET DMEM[(address >> MEM_SHIFT) + (i >> BYTE_SIZE)] = (data >>
DATA_SHIFT) OR ((data AND BYTE_MASK) << DATA_SHIFT)  // Correctly handle high and low
byte
            END FOR
            BREAK

        CASE '9':  // S9 record processing
            sscanf(line + 4, "%4x", &address)  // Read starting address
            PRINT "Starting address:", address
            SET RegistersValue[PC] = address  // Set PC to starting address
            BREAK
    END SWITCH
END WHILE
fclose(file)
END FUNCTION
```

```
FUNCTION PrintMEM(MEMTYPE)
DECLARE Range_Start AS unsigned short
DECLARE Range_End AS unsigned short

PRINT "Enter Memory Range in HEX:"
scanf("%x %x", &Range_Start, &Range_End)

FOR i = Range_Start TO Range_End STEP BYTES_PER_LINE:
    PRINT i AS HEX  // Print address

    FOR j = 0 TO BYTES_PER_LINE:
        IF i + j < Range_End:
            PRINT MEM[i + j] AS HEX
        ELSE:
            PRINT "   "
        END IF
    END FOR

    PRINT " "

    FOR j = 0 TO BYTES_PER_LINE:
        IF i + j < Range_End:
            SET c = MEM[i + j]
            IF c >= ASCII_START_VALUE AND c <= ASCII_END_VALUE:
                PRINT c AS char
            ELSE:
                PRINT "."
            END IF
        ELSE:
            PRINT " "
        END IF
    END FOR

    PRINT newline
END FOR

PRINT newline

getchar()
END FUNCTION
```

Implementation: decode.c (mostly same as A3)

- Include Header file

DEFINE LTCASE 0x7
DEFINE STCASE 0x6

FUNCTION Decode:
    ENUM {BLCase, BEQtoBRA, ADDtoST, MOVLtoMOVH, LLDR, LSTR}
    SET opcode = (IMARValue >> 13) & 0x07 # get first 3 bits

    SWITCH (opcode)
      CASE BLCase:
        RETURN BL
      CASE BEQtoBRA:
        RETURN betweenBEQandBRA(IMARValue)
      CASE ADDtoST:
        RETURN betweenADDandST(IMARValue)
      CASE MOVLtoMOVH:
        RETURN betweenMOVLandMOVH(IMARValue)
      CASE LLDR:
        RETURN LDR
      CASE LSTR:
        RETURN STR
      DEFAULT:
        PRINT "Error - instruction not yet implemented"
        RETURN Error
    END SWITCH
END FUNCTION

FUNCTION betweenADDandST (IMARValue):

    SET opcode = (IMARValue >> 10) & 0x07 # layer 2 opcode
    SET prpobuff = (IMARValue >> 9) & 0x01
    SET decbuff = (IMARValue >> 8) & 0x01
    SET incbuff = (IMARValue >> 7) & 0x01
    SET wbbuff = (IMARValue >> 6) & 0x01
    SET srcbuff = (IMARValue >> 3) & 0x07
    SET dstbuff = IMARValue & 0x07

    IF opcode equals LTCASE OR opcode equals STCASE
      SET opcode = (IMARValue >> 10) & 0x01 # layer 3 opcode
      IF opcode equals SUB_LD

```
                RETURN LD
            ELSE
                RETURN ST
            END IF

        ELSE - CONTINUE PROCESSING ADD to CEX..

END FUNCTION


FUNCTION betweenBEQandBRA (IMARValue):
        SET opcode = (IMARValue >> 10) AND 0x07 // Add a shift to get layer 2 opcode
        SET offsetbuff = (IMARValue AND 0x03FF) // Get the offset
        CALL SignExt(offsetbuff, BIT_NUMBER_9) // Sign-extend the offsetbuffer

        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:
        SET wbbuff = (IMARValue >> 6) AND 0x01 // Get the word/byte bit
        SET srcbuff = (IMARValue >> 3) AND 0x07 // Get the source register
        SET dstbuff = IMARValue AND 0x07 // Get the destination register
        SET offsetbuff = (IMARValue >> 7) AND 0x7F // Get the offset
        CALL SignExt(offsetbuff, BIT_NUMBER_6)  // Sign-extend the offset
        RETURN LDR
END FUNCTION

FUNCITON STRdecode:
SET wbbuff = (IMARValue >> 6) AND 0x01 // Get the word/byte bit SET srcbuff = (IMARValue
>> 3) AND 0x07 // Get the source register SET dstbuff = IMARValue AND 0x07 // Get the
destination register SET offsetbuff = (IMARValue >> 7) AND 0x7F // Get the offset
```

11

CALL SignExt(offsetbuff, BIT_NUMBER_6) // Sign-extend the offset RETURN STR
END FUNCTION

FUNCTION SignExt (offset, msb):
        IF (offset >> msb) AND 0x01 equals 1:
                SET offset = offset OR ((0xFFFF) << msb)
        END IF
        RETURN offset << 1
END FUNCTION

Other functions are irrelevant to A3 & A4

Implementation: execute.c (changes in BL-BRA, LD-ST & LDR-STR)

- Include Header file

```
FUNCTION execute(int instruction number)
        SWITCH (instruction number)

        CASE "INSTRUCTION":
                # execution for "INSTRUCTION"
        BREAK
        CASE BL:
                PRINT "BL:"
                SET RegistersValue[LR] = RegistersValue[PC]
                SET RegistersValue[PC] = RegistersValue[PC] + offsetbuff
        BREAK

        CASE BEQBZ:  // Branch if equal
                PRINT "BEQ/BZ:"
                IF PSW.z equals TRUE
                        SET RegistersValue[PC] = RegistersValue[PC] + offsetbuff  // Increment the PC
        by offset
                END IF
        BREAK

        CASE BNEBNZ:  // Branch if not equal
                PRINT "BNE/BNZ:"
                IF PSW.z equals FALSE
                        SET RegistersValue[PC] = RegistersValue[PC] + offsetbuff  // Increment the PC
        by offset
                END IF
        BREAK
```

12

```
CASE BCBHS:  // Branch if carry
    PRINT "BC/BHS:"
    IF PSW.c equals TRUE
        SET RegistersValue[PC] = RegistersValue[PC] + offsetbuff  // Increment the PC
         by offset
    END IF
BREAK

CASE BNCBLO:  // Branch if not carry
    PRINT "BNV/BLO:"
    IF PSW.c equals FALSE
        SET RegistersValue[PC] = RegistersValue[PC] + offsetbuff  // Increment the PC
by offset
    END IF
BREAK

CASE BN:  // Branch if negative
    PRINT "BN:"
    IF PSW.n equals TRUE
        SET RegistersValue[PC] = RegistersValue[PC] + offsetbuff  // Increment the PC
by offset
    END IF
BREAK

CASE BGE:  // Branch if greater than or equal
    PRINT "BGE:"
    IF PSW.n equals PSW.v
        SET RegistersValue[PC] = RegistersValue[PC] + offsetbuff  // Increment the PC
by offset
    END IF
BREAK

CASE BLT:  // Branch if less than
    PRINT "BLT:"
    IF PSW.n not equals PSW.v
        SET RegistersValue[PC] = RegistersValue[PC] + offsetbuff  // Increment the PC
by offset
    END IF
BREAK

CASE BRA:
    PRINT "BRA:"
```

```
        SET RegistersValue[PC] = RegistersValue[PC] + offsetbuff  // Increment the PC by
offset
BREAK

CASE LD:
        PRINT "LD:
        IF prpobuff not equals POST: // PRE-INC/DEC
                INCrement or DECrement source register
                IF(WORD):
                        SET REG[DST] = DMEM[SRC]
                ELSE: // BYTE
                        MASK DMEM[SRC]
                        SET REG[DST] = MASKED DMEM[SRC]
                ENDIF
        ELSE: // POST-INC/DEC
                IF(WORD):
                        SET REG[DST] = DMEM[SRC]
                ELSE: // BYTE
                        MASK DMEM[SRC]
                        SET REG[DST] = MASKED DMEM[SRC]
                ENDIF
                INCrement or DECrement source register
        ENDIF
        BREAK

CASE ST:
        PRINT "ST:"
        IF prpobuff not equals POST: // PRE-INC/DEC
                INCrement or DECrement destination register
                IF(WORD):
                        DMEM[DST] = REG[SRC]
                ELSE: // BYTE
                        MASK REG[SRC]
                        SET DMEM[DST] = MASKED REG[SRC]
                ENDIF
        ELSE: // POST-INC/DEC
                IF(WORD):
                        DMEM[DST] = REG[SRC]
                ELSE: // BYTE
                        MASK REG[SRC]
                        SET DMEM[DST] = MASKED REG[SRC]
                ENDIF
                INCrement or DECrement source register
```

14

```
                ENDIF
                BREAK

        CASE: LDR
                PRINT: "LDR: "
                SET EA = RegistersValue[srcbuff] + (offsetbuff / 2)
                IF wbbuff equals WORD:
                        SET RegistersValue[dstbuff] = DMEM[EA / 2]
                ELSE:
                        SET RegistersValue[dstbuff] = DMEM[EA / 2] AND 0x00FF
                END IF
        BREAK

        CASE: STR
                PRINT:"STR: "
                SET EA = RegistersValue[dstbuff] + offsetbuff
                IF wbbuff equals WORD:
                        SET DMEM[EA / 2] = RegistersValue[srcbuff]
                ELSE:
                        SET DMEM[EA / 2] = RegistersValue[srcbuff] AND 0x00FF
                END IF
        BREAK


        END SWITCH
END FUNCTION
```

Main: (same as previous Assignments)

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" 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 realized there have been many issues with the loader when trying to load multiple s1, and s2 files. So I have redesigned and reimplemented the loader, to put it in shorter terms, the IMEM and DMEM were initially set as character arrays. I have now set them to be unsigned short arrays and they are taken and read as values from the .xme file rather than characters. This has significantly reduced bugs, and makes my code a lot simpler and more functional. As for this assignment's instructions, I have mainly implemented the changes in decode.c and execute.c.

I have decided to do this since I realized the loader was causing the instructions and data to sometimes be either in an offset or in general just not saved to I/DMEM properly. Causing issues all throughout the emulator. After the new implementation, I am no longer facing these issues and bugs.

That being said, I have also completely changed the implementation of my A3 instructions executions (LD, ST, LDR, and STR) because I have realized that these functions only seem to work with the .asm files (tests) that I have made which are very simple and just focus on a single instruction at a time. So I will include the pseudo code for these functions in the design section and they will also be highlighted in red. I also included new tests for all these functions.

Testing

Test 1: Testing BL instruction

Purpose/Objective: The purpose of this test is to check if the BL function is working properly as it should (saving the incremented value of the Program Counter Register (PCorR7) into the link Register(LRorR5), and then changing the value of the program counter).

Test Configuration: I have started the emulator loading the .xme file for the A3A4test.xme program that was given to test our emulators for A3(loading and storing) and A4(Branching) content.

A3A4test.xme file loaded and program counter set

A3A4test.xme instructions in IMEM and Data in DMEM before executing

Content of the registers before executing (PC set to 0x0108)

I will then step through the code to get to the instruction right before calling the BL function and I will record the values of the registers before executing BL and then I will record the values of the registers after executing BL.

Before BL.

Expected Results: The program should work as expected saving the value of the incremented Program Counter to the Link Register and changing the value of the program counter according to the offset.

Actual Results: After executing BL, we can see that the previous value of the PC (0110) was incremented and is now the LR (0112) and the PC has changed to go to the subroutine at 0100. Meaning the program ran as expected.

Register Values after executing BL

Pass/Fail: Pass

Test 2: Testing ld instruction

Purpose/Objective: The purpose of this test is to check if the LD function is working properly as it should load the value from DMEM at the source destination address into the destination register.

Test Configuration: I have continued stepping through the previous program (A3A4test.xme included in Extra content and notes section below) using the emulator. I will stop stepping through right before executing the ld function. I will record the content in DMEM and register values before executing. And I will then observe what happens to the value of the registers after executing.

DMEM before executing LD

Register Values before executing LD

Expected Results: The program should work as expected storing the byte 0x49 located in 0x0060 in DMEM into R0.

Actual Results: The program ran as expected, getting the byte value from DMEM and saving it into R0 and then incrementing R1 to be 0x0061.

Values of registers after executing ld

Pass/Fail: Pass

Test 3: Testing st instruction

Purpose/Objective: The purpose of this test is to check if the ST function is working properly as it should (storing the value into DMEM with the source register containing the address, from the destination register).

Test Configuration: I have continued stepping through the previous program (A3A4test.xme included in Extra content and notes section below) using the emulator. I will stop stepping through right before executing the st instruction call. I will record the content in DMEM and register values before executing. And I will then observe what happens to the content of the DMEM and the registers after executing.

```
============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: m
select Memory I=IMEM D=DMEM B=both
d
Enter Memory Range in HEX:fef0 fff0
FEF0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
FF00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
FF10: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
FF20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
FF30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
FF40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
FF50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
FF60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
FF70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
FF80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
FF90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
FFA0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
FFB0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
FFC0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
FFD0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
FFE0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Content in DMEM before executing ST

```
MOV: WB=0, SRC=5, DST=7
===========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: 0b1111111101001001 | 0xFF49
R1: 0b0000000001100001 | 0x0061
R2: 0b0000000010000000 | 0x0080
R3: 0b0000000000000000 | 0x0000
R4: 0b0000000000000000 | 0x0000
R5: 0b0000000100010010 | 0x0112
R6: 0b1111111100000000 | 0xFF00
R7: 0b0000000100010010 | 0x0112
PSW(vnzc): 0000
===========DEBUGGER===========
```

Values of registers before executing ST

Expected Results: The program should work as expected storing the word 0xFF49 located in R0 into the new memory location in DMEM at the address in the stack pointer.

Actual Results: The program ran as expected, storing the value 0xFF49 into DMEM in the proper location at the address 0xFEFE.

Content of DMEM after executing ST

Values of registers after executing ST

Pass/Fail: Pass

Test 4: Testing LDR instruction

Purpose/Objective: The purpose of this test is to check if the LDR function is working properly as it should (loading the value from DMEM with the source register containing the address + the offset from the address into the destination register).

Test Configuration: I have continued stepping through the previous program (A3A4test.xme included in Extra content and notes section below) using the emulator. I will stop stepping through right before executing the LDR instruction call. I will record the content in DMEM and register values before executing. And I will then observe what happens to the content of the registers before and after executing.

Content in DMEM before executing LDR

Register Values before executing LDR

Expected Results: The program should work as expected loading the value from DMEM at address in R6 (0xFEFC) into the destination register (R0)

Actual Results: The program ran as expected, storing the value in DMEM[FEFC+offset(0)] into the R0 register

Register values after executing LDR

Pass/Fail: Pass

Test 5: Testing str instruction

Purpose/Objective: The purpose of this test is to check if the STR function is working properly as it should (storing the value into DMEM with the source register containing the address + the offset from the address, from the destination register).

Test Configuration: I have continued stepping through the previous program (A3A4test.xme included in Extra content and notes section below) using the emulator. I will stop stepping through right before executing the STR instruction call. I will record the content in DMEM and register values before executing. And I will then observe what happens to the content of the DMEM and the registers after executing.

DMEM before executing STR

Register Values before executing STR

Expected Results: The program should work as expected storing the word (0x0081) into the new memory location in DMEM at FEFC.

Actual Results: The program ran as expected storing 0x0081 at FEFC in DMEM.

DMEM after executing STR

Register Values after executing STR

Pass/Fail: Pass

Test 6: Testing BNE instruction

Purpose/Objective: The purpose of this test is to check if the BNE instruction implemented with my emulator is working properly as it should. This instruction should branch if the value of the PSW zero flag is not set ie PSW.z = 0.

Test Configuration: I have continued stepping through the previous program (A3A4test.xme included in Extra content and notes section below) using the emulator. I will stop stepping through right before executing the st instruction call. I will record the values of the registers before and after executing to tell if the program branched or not (value of PC).

```
B - Add Breakpoint
S - Step (debugger UI unavailable in step)
T - view Time Count
Q - Quit
Enter choice: r
Registers:
R0: 0b1111111101001001 | 0xFF49
R1: 0b0000000001100001 | 0x0061
R2: 0b0000000010000001 | 0x0081
R3: 0b0000000000000000 | 0x0000
R4: 0b0000000000000000 | 0x0000
R5: 0b0000000100011000 | 0x0118
R6: 0b1111111100000000 | 0xFF00
R7: 0b0000000100011110 | 0x011E
PSW(vnzc): 0000
===========DEBUGGER===========
```

Register Values before executing BNE

Expected Results: The program should work as expected by branching as the zero flag in the PSW (psw.z) is not set. Therefore we should see the PC or R7 change value and not increment as it usually would.

Actual Results: The program ran as expected it branched by changing the value of PC.

Register values after BNE execution

Pass/Fail: Pass

Test 7: Testing Implementation (A3A4test.xme)

Purpose/Objective: The purpose of this test is to check the output of the A3A4test.xme, to make sure the emulator works as it should by copying and pasting the string.

Test Configuration: For this test I reloaded the emulator with the .xme file and set the breakpoint to be after 450 lines are executed to eliminate going into the infinite loop.

DMEM before Executing the program

Expected Results: In the output we should see the values in DMEM repeated in another location in memory where the 'xxxx…' string was.

Actual Results: The program executed as expected copying and pasting the string where the "xxxx…" string was.

```
LD: PRPO:0 DEC:0 INC:1 WB:0 SRC:6 DST:0
CMP: RC=1, WB=1, SRC=0, DST=0
BNE/BNZ: offset:-16
BL: offset:-18
LD: PRPO:0 DEC:0 INC:1 WB:1 SRC:1 DST:0
MOVH: dst:0 bits:255
MOV: WB=0, SRC=5, DST=7
===========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
d
Enter Memory Range in HEX:0050 0200
0050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0060: 49 6E 70 75 74 20 73 74 72 69 6E 67 20 74 6F 20  Input string to
0070: 63 6F 70 79 00 E2 00 00 00 00 00 00 00 00 00 00  copy............
0080: 49 6E 70 75 74 20 73 74 72 69 6E 67 20 74 6F 20  Input string to
0090: 63 6F 70 79 00 E2 00 00 00 00 00 00 00 00 00 00  copy............
00A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

DMEM after executing the program

Pass/Fail: Pass

Extra Content and Notes

A3A4test.lis (file used)

X-Makina Assembler - Version XM-23P Single Pass+ Assembler - Release 24.04.17
Input file name: A3A4test.asm
Time of assembly: Wed 17 Jul 2024 17:00:08

```
1                    ;
2                    ; Test of A3 and A4
3                    ; A3: Memory reads and writes
4                    ; A4: Branching
5                    ;
6                    PC      equ      R7
7                    SP      equ      R6
```

```
8                          LR       equ      R5
9                          NUL      equ      '\0'
10                         ;
11                         ; Data segment
12                         ;
13                                  Data
14                                  org      #60
15    0060    6E49    InStr    ascii    "Input string to copy\0"
15    0062    7570
15    0064    2074
15    0066    7473
15    0068    6972
15    006A    676E
15    006C    7420
15    006E    206F
15    0070    6F63
15    0072    7970
15    0074    0000
16                                  org      #80
17    0080    7878    OutStr   ascii    "xxxxxxxxxxxxxxxxxxxxxxxx"
17    0082    7878
17    0084    7878
17    0086    7878
17    0088    7878
17    008A    7878
17    008C    7878
17    008E    7878
17    0090    7878
17    0092    7878
17    0094    7878
17    0096    7878
18                         ;
19                         ; Stack segment
20                         ;
21                                  org      #FE00
22    FE00    0000             bss      #100
23                         StkTop
24                         ;
25                         ; Code segment
26                         ;
27                                  Code
28                                  org      #100
29                         ;
30                         ; ReadSubr - returns a char from R1 in R0
31                         ;
32                         ReadSubr
33    0100    58C8             ld.b     R1+,R0          ; Read R1 into R0
34    0102    7FF8             movh     #FF00,R0; Overwrite MSByte
35    0104    4C2F             mov      LR,PC           ; Return
36    0106    40BF             add      #FF,PC          ; Must not execute
37                         ;
38                         ; Mainline
39                         ;
40                         RWStr
41    0108    6006             movl     StkTop,SP
42    010A    7FFE             movh     StkTop,SP
43                         ;
44    010C    6B01             movlz    InStr,R1
45    010E    6C02             movlz    OutStr,R2
46                         ;
```

47			RWLoop		
48	0110	1FF7	bl	ReadSubr	
49	0112	5F06	st	R0,-SP	; Char as parameter
50	0114	5F16	st	R2,-SP	; Addr as parameter
51			;		
52	0116	0005	bl	WriteSubr	
53	0118	58B2	ld	SP+,R2	; Restore Addr
54	011A	58B0	ld	SP+,R0	; Restore Char
55			;		
56	011C	45C0	cmp.b	NUL,R0	
57	011E	27F8	bne	RWLoop	; Not NUL, repeat
58			;		
59			; NUL found, infinite loop until ^C		
60			;		
61	0120	3FFF	RWDone bra	RWDone	
62			;		
63			; Writes a character to the destination string		
64			; Stack has Output address and char to output		
65			;		
66			WriteSubr		
67	0122	8030	ldr	SP,$0,R0 ; R0 <- Addr of output string	
68	0124	8132	ldr	SP,$+2,R2	; R2 <- Character to output
69			;		
70	0126	5C50	st.b	R2,R0	
71			;		
72	0128	4088	add	#1,R0	; Increment address
73	012A	C006	str	R0,SP,$0	
74			;		
75	012C	4C2F	mov	LR,PC	; Return
76	012E	40BF	add	#FF,PC	; Must not execute
77			;		
78			end	RWStr	

Successful completion of assembly - 2P

** Symbol table **

Constants (Equates)

Name		Type	Value	Decimal	
NUL		CON	0000	0	PRI

Labels (Code)

Name		Type	Value	Decimal	
RWDone		REL	0120	288	PRI
WriteSubr		REL	0122	290	PRI
RWLoop		REL	0110	272	PRI
RWStr		REL	0108	264	PRI
ReadSubr		REL	0100	256	PRI

Labels (Data)

Name		Type	Value	Decimal	
StkTop		REL	FF00	-256	PRI
OutStr		REL	0080	128	PRI
InStr		REL	0060	96	PRI

Registers

Name		Type	Value	Decimal	
LR		REG	0005	5	PRI
SP		REG	0006	6	PRI
PC		REG	0007	7	PRI

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\A3A4test.xme

Input file name: LDST1.asm

A3A4test.xme (file used)

S00F00004133413474657374 2E61736DD8
S2180060496E70757420737472696E6720746F20636F707900E2
S21B0080787878787878787878787878787878787878787878787878787824
S204FE0000FD
S1210100C858F87F2F4CBF400660FE7F016B026CF71F065F165F0500B258B058C04503
S115011EF827FF3F30803281505C884006C02F4CBF4057
S9030108F3