Lab/Tutorial 3
The XM23p: More Debugger Commands
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 emulator is to test and implement the various functions of the XM23p CPU developed and produced by the XM cooperation. This cpu is an improved and updated version of the previous model's variant, The XM-23.

The design, implementation and testing of this CPU will aid their students in their journey to learning the depths of a CPU's architecture to gain the knowledge necessary to complete the required course: Computer Architecture. The course stands as a necessary asset for the computer engineers the students aspire to become.

The purpose of this tutorial/lab is to add more functions to the previously implemented loader and debugger, this will aid in debugging, troubleshooting and diagnosing errors within the CPU's system when implementing the emulator as well as using it as a final product.

The lab aims to add 4 new fundamental debugger features, these include:

- A command to display the hexadecimal value or content of the 8 CPU registers.
- A command to change the hexadecimal value or content held within these 8 CPU registers.
- A command to change a record in memory
- A command to add a breakpoint to stop execution at a certain instruction in IMEM

Design:

Data Dictionary

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

s-record = ‘s’ + type + length of record + Address + Data + Checksum
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]
Checksum = 1[Byte Pair]1

General 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*
Destination = [R0-R4] *int bits*

Register Instructions (ADD-SXT) = Opcode + Operand

Register Initialization Instructions (MOVL-MOVH) = Opcode + Operand

Breakpoint = IMEMaddress
IMEMaddress = Address *Address in Instruction memory to stop executing*
Address = 2[Byte Pair]2

Pseudo Code

Main.c

The only new change to this part of the code is adding the debugger option to take you to debugger mode. It has been added as an option “d”

```
INCLUDE "xm23p.h"

FUNCTION main(argc, argv)

   INITIALIZE IMEM and DMEM to '0'
   INITIALIZE Registers to '0'

   WHILE choice IS NOT 'q'
      PRINT “*Menu Options*”
      READ choice

      IF choice IS 'l'
          # IRRELEVANT FOR THIS LAB

      ELSE IF choice IS 'm'
          # IRRELEVANT FOR THIS LAB

      ELSE IF choice IS 'q'
         RETURN 0 # QUIT

      ELSE IF choice IS 'f' # fetch and decode choice (L2)
         CALL process_instruction()

      ELSE IF choice IS 'd'
         CALL debug()

      ELSE
         PRINT "Invalid choice"
      END IF
   END WHILE

   RETURN 0
END FUNCTION
```

ISA.c

Not many changes have been added to this file for this lab, other than the comparison of the loop now accommodates for the breakpoint.

```
INCLUDE "xm23p.h" # main emulator library

FUNCTION process_instruction()
   CONVERT start address to binary and store in program counter
# RUN AT LEAST ONCE do-while loop for instruction (NEW XM23p FUNCITON)
```

```
    DO
        CALL fetch()
        CALL decode()
    WHILE IMARValue != 0000 AND E_Start_Addresses != BreakpointValue
END FUNCTION

FUNCTION fetch()
    STORE instruction from memory into IMAR
    INCREMENT I_Start_Addresses by 4
    INCREMENT E_Start_Addresses by 2
    CONVERT IMAR to unsigned short and store in IMARValue
END FUNCTION

FUNCTION decode()
    IF E_Start_Addresses - 2 EQUALS BreakpointValue THEN
        PRINT "Breakpoint reached"
        RETURN
    END IF

    PRINT decoded instruction and address
    GET first 3 bits of IMARValue and store in opcode

    SWITCH opcode
        CASE BLCase:
            PRINT "BL - tbd.."
        CASE BEQtoBRA:
            PRINT "BEQ-BRA - tbd.."
        CASE ADDtoST:
            CALL betweenADDandST(IMARValue)
        CASE MOVLtoMOVH:
            CALL betweenMOVLandMOVH(IMARValue)
        CASE LDR:
            PRINT "LDR - tbd.."
        CASE STR:
            PRINT "STR - tbd.."
        DEFAULT:
            PRINT "instruction not yet implemented"
    END SWITCH
END FUNCTION

FUNCTION betweenADDandST(IMARValue)
    GET sub opcode from IMARValue
```

```
    IF sub opcode EQUALS 0x7 OR 0x6 THEN
        GET details for LD or ST and print
    ELSE IF sub opcode EQUALS 0x3 THEN
        PRINT "Layer2: MOV_CLRCC case"
    ELSE IF sub opcode EQUALS 0x4 THEN
        PRINT "Layer2: CEX case"
    ELSE
        GET details for ADD to BIS and print
    END IF
END FUNCTION

FUNCTION betweenMOVLandMOVH(IMARValue)
    GET Layer2 opcode from IMARValue
    GET destination register and bits from IMARValue

    SWITCH opcode
        CASE 0x00:
            PRINT "MOVL: dst bits"
        CASE 0x01:
            PRINT "MOVLZ: dst bits"
        CASE 0x02:
            PRINT "MOVLS: dst bits"
        CASE 0x03:
            PRINT "MOVH: dst bits"
        DEFAULT:
            PRINT "instruction not yet implemented"
    END SWITCH
END FUNCTION
```

Debugger.c

This is where most of the change for this lab has occurred, I have made this new file that aids the user in navigating through the different debugger options, all the debugging functions except for the breakpoint are implemented within this part of the code. I have also created an extra function which I think is beneficial and helpful, it shows the user the content of the registers in binary which will aid in implementing the various ISA directives and their execution.

```
INCLUDE PSEUDO CODE
INCLUDE "xm23p.h"

FUNCTION debug()
    DECLARE choice2 AS CHAR

    PRINT "Debugging mode!"
    WHILE choice2 IS NOT 'q'
```

```
    PRINT "Debugger Menu options"
    READ choice2

    IF choice2 IS 'R' OR 'r'

      PRINT "Registers:"

      FOR i FROM 0 TO 7 # print them as binary values
        PRINT "R" + i + ": "
        PRINT "binary:"
        FOR j FROM 0 TO 15
          PRINT RegistersBinaryString[i][j]
        END FOR

        PRINT "Hex: " #print them as hex values
        FOR j FROM 0 TO 3
          PRINT RegistersHexString[i][j]
        END FOR
        PRINT NEW LINE
      END FOR

    ELSE IF choice2 IS 'E' OR 'e'
      PRINT "Enter register number: "
      READ regnum
      PRINT "Enter new content(hex): "
      READ newcontent
      FOR i FROM 0 TO 3
        RegistersHexString[regnum][i] TO newcontent[i]
      END FOR

      CALL UpdateRegistersBinary(newcontent, regnum)
      PRINT "Register R" + regnum + " content changed to " +
RegistersHexString[regnum]

    ELSE IF choice2 IS 'M' OR 'm'
      PRINT "select Memory I=IMEM D=DMEM B=both"
      READ memchoice

      #PRINTING MEMORY PORTION
      IF memchoice IS 'I' OR 'i'
        CALL PrintIMEM()
      ELSE IF memchoice IS 'D' OR 'd'
        CALL PrintDMEM()
```

```
        ELSE IF memchoice IS 'B' OR 'b'
            CALL PrintMEM()
        ELSE
            PRINT "Invalid choice, try again"
        END IF

    ELSE IF choice2 IS 'I' OR 'i'
        PRINT "Enter address(Hex): "
        READ addresschar
        PRINT "Enter new content(hex): "
        READ newcontent
        CALL Send2IMEM(newcontent, addresschar, 2)

    ELSE IF choice2 IS 'D' OR 'd'
        PRINT "Enter address(Hex): "
        READ addresschar
        PRINT "Enter new content(hex): "
        READ newcontent
        CALL Send2DMEM(newcontent, addresschar, 2)

    ELSE IF choice2 IS 'B' OR 'b'
        PRINT "Breakpoint location in Hex?: "
        READ Breakpoint
        SET BreakpointValue TO ConvertHexToInteger(Breakpoint)
        PRINT "Breakpoint added at " + Breakpoint

    ELSE IF choice2 IS 'Q' OR 'q'
        RETURN #QUIT DEBUGGER MODE

    ELSE
        PRINT "Invalid choice, try again"
    END IF
END 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 emulator main.c loader.c isa.c debugger.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:

debugger.c:

```
#include "xm23p.h"

void debug(){
char choice2;

printf("Debugging mode!\n");
while(choice2 != 'q'){
printf("===========DEBUGGER==========\n");
printf("Choose an option: \n");
printf("R - View Registers Content\n");
printf("E - Edit Register Content\n");
printf("M - Display Memory\n"); //
printf("I - Edit in IMEM\n");
printf("D - Edit in DMEM\n");
printf("B - Add Breakpoint\n");
printf("Q - Quit\n");

printf("Enter choice: ");
scanf(" %c", &choice2);

if(choice2 == 'R' || choice2 == 'r'){ // view registers content
printf("Registers:\n");
for(int i = 0; i < 8; i++){

printf("R%d: \n", i);
printf("binary:");
for(int j = 0; j < 16; j++){
printf("%c", RegistersBinaryString[i][j]);
}
printf("\nHex: ");
for(int j = 0; j < 4; j++){
printf("%c", RegistersHexString[i][j]);
}
```

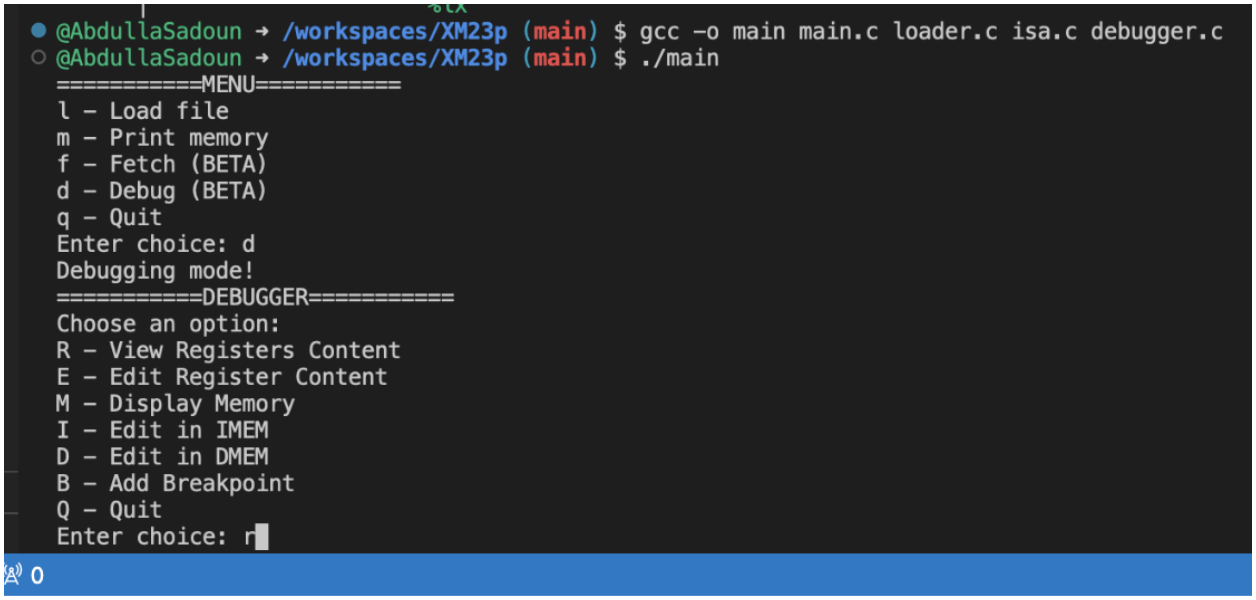
```
printf("\n");
}
} else if(choice2 == 'e' || choice2 == 'E'){ // edit a registers content
printf("Enter register number: ");
int regnum;
scanf("%d", &regnum);
printf("Enter new content(hex): ");
char newcontent[4];
scanf("%s", newcontent);
// change RegisterHexstring to new content
for(int i = 0; i < 4; i++){ // updating the register content
RegistersHexString[regnum][i] = newcontent[i];
}
UpdateRegistersBinary(newcontent, regnum); // update the binary content
printf("Register R%d content changed to %c%c%c%c\n", regnum,
RegistersHexString[regnum][0], RegistersHexString[regnum][1],
RegistersHexString[regnum][2], RegistersHexString[regnum][3]);

} else if(choice2 == 'm' || choice2 == 'M'){ // display memory
char memchoice;
printf("select Memory I=IMEM D=DMEM B=both\n");
scanf(" %c", &memchoice);

if(memchoice == 'I' || memchoice == 'i'){ // print IMEM
PrintIMEM();
} else if(memchoice == 'D' || memchoice == 'd'){ // print DMEM
PrintDMEM();
} else if(memchoice == 'B' || memchoice == 'b'){ // print both
PrintMEM();
} else {
printf("Invalid choice, try again\n");
}
} else if(choice2 == 'i' || choice2 == 'I'){ // edit a value in IMEM
printf("Enter address(Hex): ");
char addresschar[4];
scanf("%s", addresschar);
printf("Enter new content(hex): ");
char newcontent[4];
scanf("%s", newcontent);
Send2IMEM(newcontent, addresschar, 2);
```

```
} else if(choice2 == 'd' || choice2 == 'D'){ // edit in DMEM
printf("Enter address(Hex): ");
char addresschar[4];
scanf("%s", addresschar);
printf("Enter new content(hex): ");
char newcontent[4];
scanf("%s", newcontent);
Send2DMEM(newcontent, addresschar, 2);

} else if(choice2 == 'b' || choice2 == 'B'){ // add a breakpoint
// breakpoint to stop program counter from incrementing
printf("Breakpoint locaiton in Hex?: \n");
scanf("%s", Breakpoint);
BreakpointValue = strtol(Breakpoint, NULL, 16);
printf("Breakpoint added at %s\n", Breakpoint);

} else if(choice2 == 'q' || choice2 == 'Q'){
return;
} else {
printf("Invalid choice, try again\n");
}
}

}

void UpdateRegistersBinary(char newcontent[4], int regnum){
/*
function to update the registers binary content to display the new content
*/
// get the int value of the new content which is in hex (char string)
int newcontentvalue = strtol(newcontent, NULL, 16);

// convert the new content (hex char string) to binary char string
for (int i = 15; i >= 0; --i, newcontentvalue >>= 1) {
RegistersBinaryString[regnum][i] = (newcontentvalue & 1) + '0';
}

return;
}
```

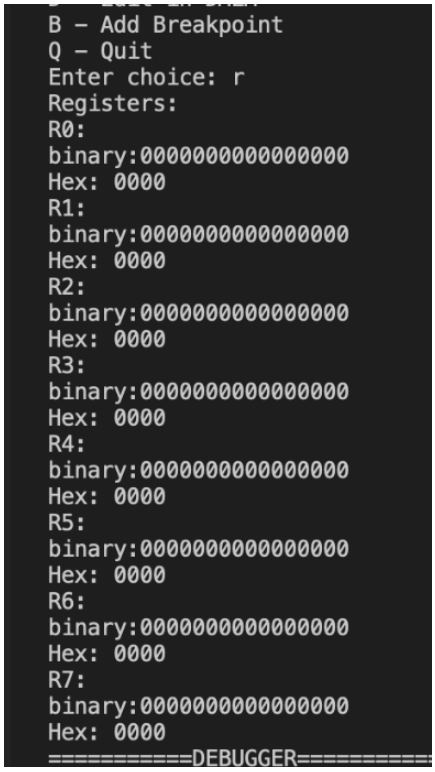
Testing

Test 1: Testing Register Function.

Purpose/Objective: The purpose of this test is to check if the software's new Register Testing function works properly and performs as it should showing the current content of the registers.
Test Configuration: For this test, I began by running the software and then entering debugger mode right away. I then proceeded to choose the view register option by entering "r".



Expected Results: The Program shouldn't have a problem executing and all the registers should be displayed with their hex and binary values which should be 0 in this case since nothing has been loaded or executed\.
Actual Results: The actual result was as expected and the software did not have any problems executing, it displayed all the registers with the their correct values both in binary and in hex.



Pass/Fail: Pass

Test 2: Changing a register value

Purpose/Objective: The purpose of this test is to check whether the function for changing a register's value works or not.
Test Configuration: For this test, I began by running the software and then entering debugger mode right away. I then proceeded to choose the view register content option "r" to view the content of the registers. They all seem to be zero for now.

```
Registers:
R0:
binary:0000000000000000
Hex: 0000
R1:
binary:0000000000000000
Hex: 0000
R2:
binary:0000000000000000
Hex: 0000
R3:
binary:0000000000000000
Hex: 0000
R4:
binary:0000000000000000
Hex: 0000
R5:
binary:0000000000000000
Hex: 0000
R6:
binary:0000000000000000
Hex: 0000
R7:
binary:0000000000000000
Hex: 0000
```

I then went into register 1 and changed its value to be 00ff.

```
Hex: 0000
==========DEBUGGER==========
Choose an option:
R - View Registers Content
E - Edit Register Content
M - Display Memory
I - Edit in IMEM
D - Edit in DMEM
B - Add Breakpoint
Q - Quit
Enter choice: e
Enter register number: 1
Enter new content(hex): 00ff
Register R1 content changed to 00ff
==========DEBUGGER==========
```

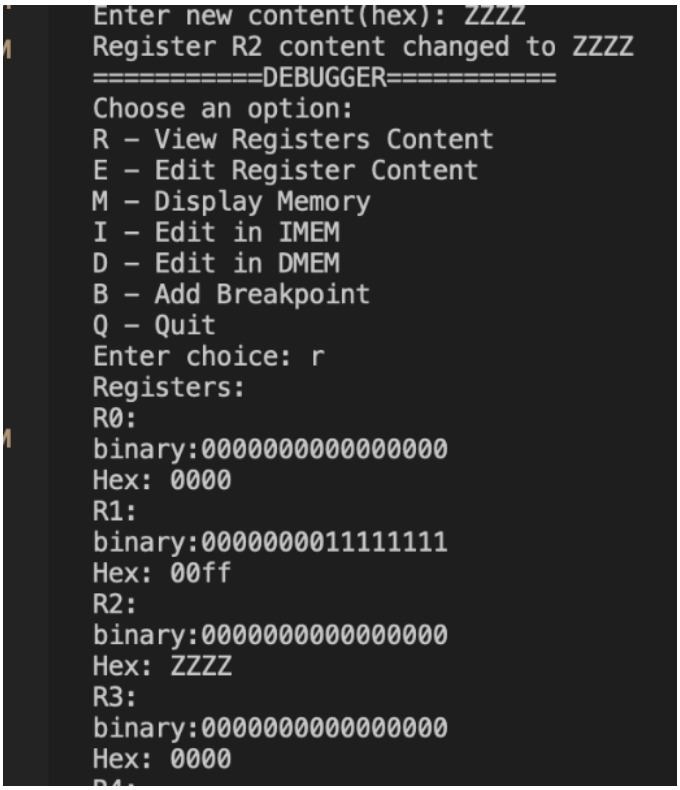
Expected Results: The function should work properly and the register's content should be changed accordingly.
Actual Results: The software did as expected updating the content of the register in hex and doing so in binary as well.

```
Register R1 content changed to 00ff
==========DEBUGGER==========
Choose an option:
R - View Registers Content
E - Edit Register Content
M - Display Memory
I - Edit in IMEM
D - Edit in DMEM
B - Add Breakpoint
Q - Quit
Enter choice: r
Registers:
R0:
binary:0000000000000000
Hex: 0000
R1:
binary:0000000011111111
Hex: 00ff
R2:
binary:0000000000000000
Hex: 0000
R3:
binary:0000000000000000
Hex: 0000
R4:
binary:0000000000000000
Hex: 0000
R5:
binary:0000000000000000
Hex: 0000
R6:
binary:0000000000000000
Hex: 0000
R7:
binary:0000000000000000
Hex: 0000
==========DEBUGGER==========
```

Pass/Fail: Pass

Test 3: Inputting an invalid value in Register

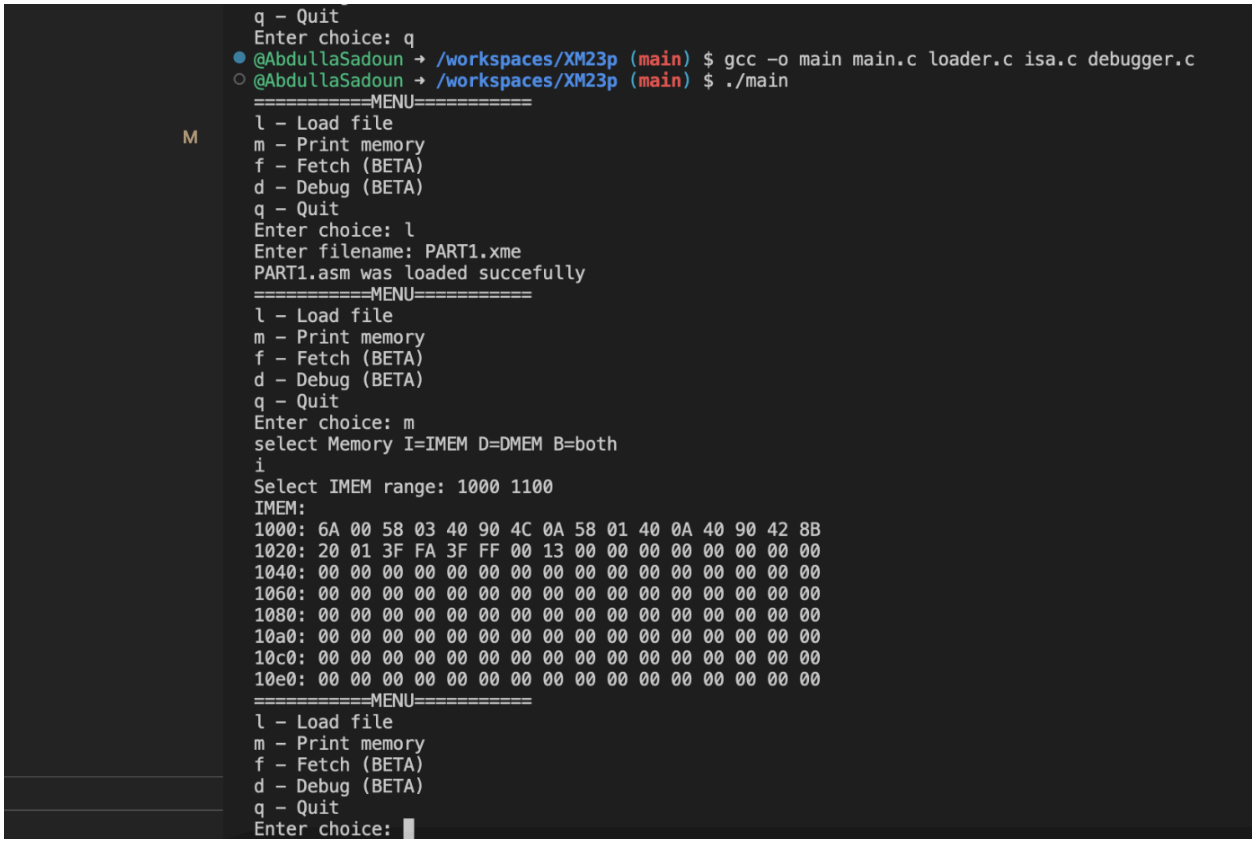
Purpose/Objective: The objective of this test is to observe how the software responds when receiving an invalid value to be put in the register.
Test Configuration: For this test, I will try to enter a value like “z” or “Z” in the register which is not possible as a hex value as they only go from “0” to “f” which is 16 in decimal values.
Expected Results: I have not accommodated for this case and I think the program will just store the value as it is.
Actual Results: The program reacted as expected, it recorded the character as it is in hex but in binary the value did not change. I will include a new range and error message to prevent this case later on.



Pass/Fail: FAIL

Test 4: Changing content in memory

Purpose/Objective: The purpose of this test is to see if the program can properly edit, change or overwrite a value in IMEM.
Test Configuration: For this test, I began by running the software, loading PART1.xme (included below in extra content) and then entered debugger mode right away. I then proceeded to choose the option “I” to edit a value in IMEM.



Expected Results: The program should update the content of the memory and store/keep the new value where it has changed.
Actual Results: The program performed as expected successfully storing the new value in IMEM.

```
I - Edit in IMEM
D - Edit in DMEM
B - Add Breakpoint
Q - Quit
Enter choice: i
Enter address(Hex): 1020
Enter new content(hex): 00
==========DEBUGGER==========
Choose an option:
R - View Registers Content
E - Edit Register Content
M - Display Memory
I - Edit in IMEM
D - Edit in DMEM
B - Add Breakpoint
Q - Quit
Enter choice: q
==========MENU==========
l - Load file
m - Print memory
f - Fetch (BETA)
d - Debug (BETA)
q - Quit
Enter choice: m
select Memory I=IMEM D=DMEM B=both
i
Select IMEM range: 1000 1100
IMEM:
1000: 6A 00 58 03 40 90 4C 0A 58 01 40 0A 40 90 42 8B
1020:  00 3F FA 3F FF 00 13 00 00 00 00 00 00 00 00
1040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
==========MENU==========
```

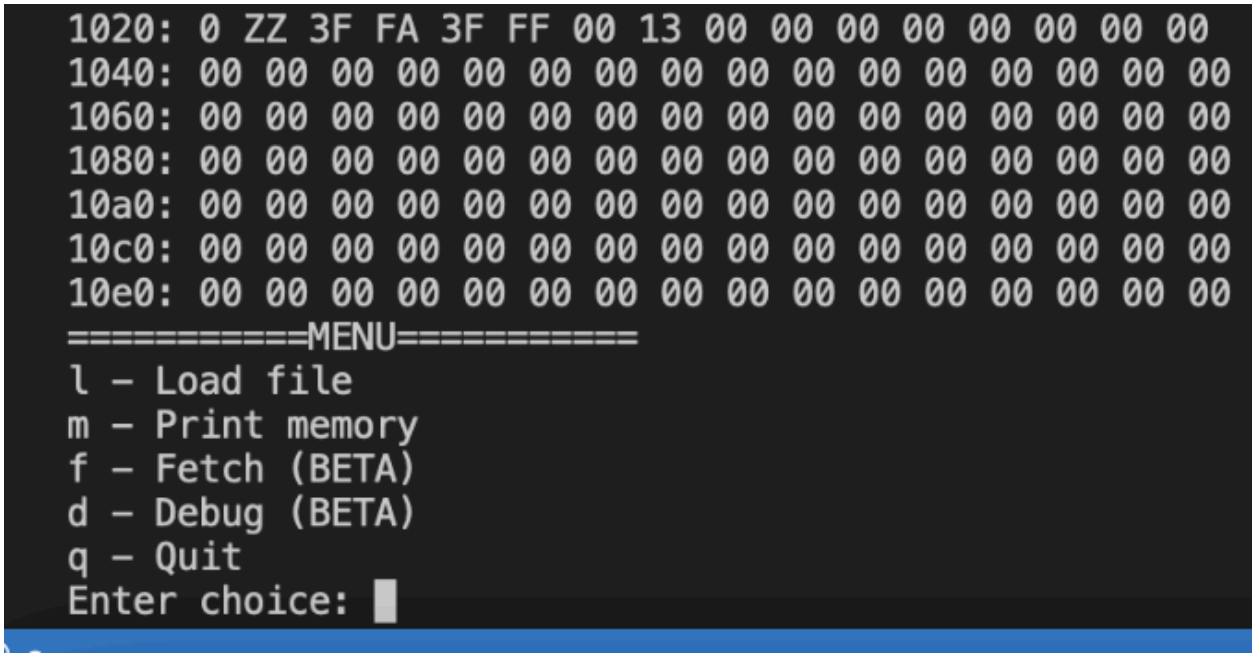
Pass/Fail: Pass

Test 5: Inputting invalid content in memory

Purpose/Objective: The objective of this test is to observe how the software responds when receiving an invalid value to be put in the instruction memory.
Test Configuration: For this test, I will try to enter a value like “z” or “Z” in a location in IMEM which is not possible as a hex value as they only go from “0” to “f”.

```
Select IMEM range: 1000 1100
IMEM:
1000: 6A 00 58 03 40 90 4C 0A 58 01 40 0A 40 90 42 8B
1020: 20 01 3F FA 3F FF 00 13 00 00 00 00 00 00 00 00
1040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
==========MENU==========
```

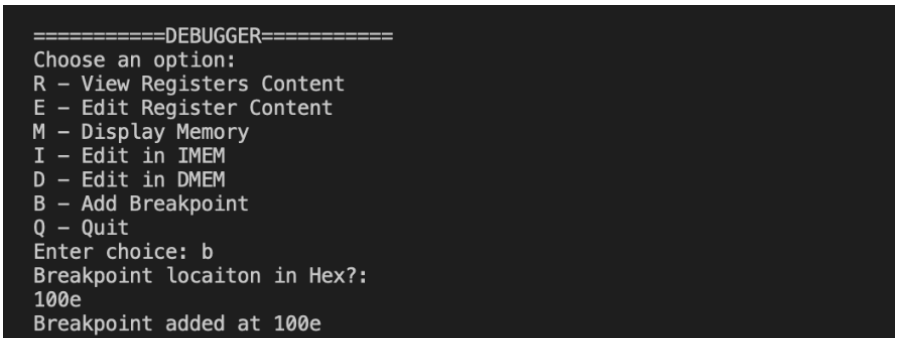
Expected Results: I have not accommodated for this case and I think the program will just store the value as it is in IMEM.
Actual Results: The program reacted as expected, it recorded the character as it is in hex. I will include a new range and error message to prevent this case later on.



Pass/Fail: FAIL

Test 6: Testing Breakpoint function

Purpose/Objective: The purpose of this test is to see if the program can properly stop executing the instructions when it encounters the breakpoint.
Test Configuration: For this test, I began by running the software, loaded PART1.xme (included below along with corresponding .lis file). I then entered debugger mode and set the breakpoint at 100e.



Expected Results: The Program should stop when the breakpoint is encountered
Actual Results: The program did stop when the breakpoint was encountered and did not proceed to execute the next instruction.

```
==========MENU==========
l - Load file
m - Print memory
f - Fetch (BETA)
d - Debug (BETA)
q - Quit
Enter choice: f
decoded@1000 instruction:6a00 MOVLZ: dst:0 bits:64
decoded@1002 instruction:5803 LD: PRPO:0 DEC:0 INC:0 WB:0 SRC:0 DST:3
decoded@1004 instruction:4090 ADD: RC=1, WB=0, SRC=2, DST=0
decoded@1006 instruction:4c0a Layer2: MOV_CLRCC case
decoded@1008 instruction:5801 LD: PRPO:0 DEC:0 INC:0 WB:0 SRC:0 DST:1
decoded@100a instruction:400a ADD: RC=0, WB=0, SRC=1, DST=2
decoded@100c instruction:4090 ADD: RC=1, WB=0, SRC=2, DST=0
```

Pass/Fail: Pass

Test 7: Inputting Breakpoint before/after first instruction's address

Purpose/Objective: The objective of this test is to see how the program behaves when a breakpoint is set before or after the address in which the instructions are located.
Test Configuration: For this test, I began by running the software, loaded PART1.xme (included below along with corresponding .lis file). I then entered debugger mode and set the breakpoint at first at 0x0900 and in the second run I set it at 0x1100 which is before and after the set of instructions loaded in memory.
Expected Results: I have not accounted for a case like this but I think the other part of my do-while loop will stop it from running since there is no instructions to run at these locations
Actual Results: The program seems like it kept running in both cases. This is probably due to the fact that the breakpoint is set in the while loop and only stops executing when encountered, furthermore the PC is changed to the address taken from the s-records before the loop, that's why it never stops in these cases and only works in an ideal scenario where the address of the breakpoint should be where an instruction is.

```
Breakpoint locaiton in Hex?:
0900
Breakpoint added at 0900
==========DEBUGGER==========
Choose an option:
R - View Registers Content
E - Edit Register Content
M - Display Memory
I - Edit in IMEM
D - Edit in DMEM
B - Add Breakpoint
Q - Quit
Enter choice: q
==========MENU==========
l - Load file
m - Print memory
f - Fetch (BETA)
d - Debug (BETA)
q - Quit
Enter choice: f
decoded@1000 instruction:6a00 MOVLZ: dst:0 bits:64
decoded@1002 instruction:5803 LD: PRPO:0 DEC:0 INC:0 WB:0 SRC:0 DST:3
decoded@1004 instruction:4090 ADD: RC=1, WB=0, SRC=2, DST=0
decoded@1006 instruction:4c0a Layer2: MOV_CLRCC case
decoded@1008 instruction:5801 LD: PRPO:0 DEC:0 INC:0 WB:0 SRC:0 DST:1
decoded@100a instruction:400a ADD: RC=0, WB=0, SRC=1, DST=2
decoded@100c instruction:4090 ADD: RC=1, WB=0, SRC=2, DST=0
decoded@100e instruction:428b SUB: RC=1, WB=0, SRC=1, DST=3
decoded@1010 instruction:2001 BEQ-BRA - tbd..
decoded@1012 instruction:3ffa BEQ-BRA - tbd..
decoded@1014 instruction:3fff BEQ-BRA - tbd..
decoded@1016 instruction:13 BL - tbd..
decoded@1018 instruction:0 BL - tbd..
```

Before instruction case

```
B - Add Breakpoint
Q - Quit
Enter choice: b
Breakpoint locaiton in Hex?:
1100
Breakpoint added at 1100
==========DEBUGGER==========
Choose an option:
R - View Registers Content
E - Edit Register Content
M - Display Memory
I - Edit in IMEM
D - Edit in DMEM
B - Add Breakpoint
Q - Quit
Enter choice: q
==========MENU==========
l - Load file
m - Print memory
f - Fetch (BETA)
d - Debug (BETA)
q - Quit
Enter choice: f
decoded@1000 instruction:6a00 MOVLZ: dst:0 bits:64
decoded@1002 instruction:5803 LD: PRPO:0 DEC:0 INC:0 WB:0 SRC:0 DST:3
decoded@1004 instruction:4090 ADD: RC=1, WB=0, SRC=2, DST=0
decoded@1006 instruction:4c0a Layer2: MOV_CLRCC case
decoded@1008 instruction:5801 LD: PRPO:0 DEC:0 INC:0 WB:0 SRC:0 DST:1
decoded@100a instruction:400a ADD: RC=0, WB=0, SRC=1, DST=2
decoded@100c instruction:4090 ADD: RC=1, WB=0, SRC=2, DST=0
decoded@100e instruction:428b SUB: RC=1, WB=0, SRC=1, DST=3
decoded@1010 instruction:2001 BEQ-BRA - tbd..
decoded@1012 instruction:3ffa BEQ-BRA - tbd..
decoded@1014 instruction:3fff BEQ-BRA - tbd..
decoded@1016 instruction:13 BL - tbd..
decoded@1018 instruction:0 BL - tbd..
```

After instructions case

Pass/Fail: FAIL

Extra Content and Notes

Pseudo Created with: https://pseudoeditor.com/app/

Note for Emad: I have fixed the breakpoint function, I implemented the while loop and made it stop when the breakpoint is encountered, now there is no way to step over the breakpoint and overpass it in the code.

PART1.xme (file used)

PART1.lis (file used)

X-Makina Assembler - Version XM-23P Single Pass+ Assembler - Release 24.04.17
Input file name: PART1.asm
Time of assembly: Wed 15 May 2024 21:33:36

```
 1
 2                      ;
 3                      ; Sum an array of 16-bit numbers
 4                      ; ECED 3403
 5                      ; 15 May 24
 6                      ;
 7                            CODE
 8                            org     #1000
 9                      ;
10                      ;
11     1000   6A00   Main   movlz   Array,R0       ; r0=Address of the array
12     1002   5803          ld      R0,R3         ; load stopper into r3
```

```
13    1004  4090          add   #2,R0          ; move r0 to the next element (first element
to be summed) increment by 2 as bytes are in pairs
14    1006  4C0A          mov   R1,R2          ; setting r2 as sum register and making it 0
15                  ;
16    1008  5801  loop    ld    R0,R1          ; load the array's element into r1
17    100A  400A          add   R1,R2          ; Add the element to the sum
18    100C  4090          add   #2,R0          ; Increment R0 to point to the next element
in the array
19                  ;
20                  ; check if stopper is 0 to stop summing
21                  ;
22    100E  428B          sub   #1,R3          ; stopper - 1
23    1010  2001          bz    Done           ; end loop if stopper is 0
24                  ;
25    1012  3FFA          bra   loop           ; continue adding
26                  ;
27                  ; adding complete, result are in r2
28                  ;
29                  Done
30                  ;
31                  ; Finished - busy wait
32                  ;
33    1014  3FFF  BWait   bra   BWait
34                  ;
35                  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
36                  ;
37                  ; Data space
38                  ;
39                        DATA
40                        org   #40
41                  ;
42                  ; the array of integers used:
43                  ;
44    0040  FFFF  Array   word  $-1            ; (5=stopper in r3)
45    0042  1000          word  #1000
46    0044  2000          word  #2000
47    0046  3000          word  #3000
48    0048  4000      word      #4000
49    004A  5000      word      #5000
50                  ;
51                  ; no store for result they remained in register
52                  ;
53                        end   Main
```

Successful completion of assembly - 2P

** Symbol table **

Constants (Equates)

Name	Type	Value	Decimal	

Labels (Code)

Name	Type	Value	Decimal	
BWait	REL	1014	4116	PRI
Done	REL	1014	4116	PRI
loop	REL	1008	4104	PRI
Main	REL	1000	4096	PRI

Labels (Data)

Name	Type	Value	Decimal	
Array	REL	0040	64	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\Lab 1\PART1.xme