Programming Assignment 4 – Disassemble an LC3 object file.

1. Create a new directory on the *student* machine under your home directory named ***assign4***.

   Parts of this assignment will require software that is installed on *student*. That software is also usable from any Linux distribution with GCC installed so it can work from a Linux machine or a Mac. To work under Windows would require installing of a product called Cygwin. The easiest thing to do is to do all of the work on *student* so this document is going to be assuming *student*.

2. This programming assignment is to interpret a program file written for the LC3. Create a file named ***assign4.c*** in your assign4 directory. The rest of this document is going to assume you will copy and create files to the assign4 directory.

3. For this to work you have to have an LC3 hex file and open it for reading. Section 18.5 of your book talks about file I/O in C. I am not going to cover it in class. I will answer questions about it, however. The file that you will be using for testing is called ***test1.hex***. Copy ***test1.hex*** from the AsULearn page.

4. Create a function in *assign4.c* named ***printAssembly*** which takes a string parameter (char array). The string argument will be the name of the "hex" file to open and parse. Your ***printAssembly*** should open the file as specified in the parameter, read an instruction as a hex value, and print the instruction by calling the appropriate print method as listed below. Your ***printAssembly*** should then read the next instruction and repeat until no more instructions are available.

5. The formats shown at the end of this document are the formats I expect you to print EXACTLY for any instruction. Note that this is an example program. Your program should interpret each instruction. I will be testing with different instructions. Different instructions will have different registers, immediate values, and addresses.

6. Create a different function for printing each instruction. Each function should take the integer instruction as a parameter and the program counter as an integer, **but only if needed,** and print the instruction in the specified format. The name of the methods you should create are; ***printAdd, printAnd, printBr, printJmpRet, printJsrJsrr, printLd, printLdi, printLdr, printLea, printNot, printRti, printSt, printSti, printStr, and printTrap***. Note that RET is a special form of JMP. Any JMP which has a base register of 7 is a return. Note also that JSR and JSRR use the same opcode.

   Create all of these functions in assign4.c. Initially have them simply print a newline. Download ***check4.c*** and compile it with the following command.

   ```
   gcc -o check4 check4.c
   ```

   Run the program by simply typing ***check4***. It will tell you if you have any errors with the return types, parameters, or naming of your functions. You may have to run this on *student*.

7. An example: Interpreting the first instruction from test1.hex.
   The first hex value is x1647

   x1647
   0001011001000111
   0001 – The instruction is an ***add*** so call ***printAdd*** and pass it the integer 0x1647. Function printAdd(0x1647) should print the word **ADD** and a **tab**.
   011 – The destination register is 3 So print **R3**, a **comma** and a **space**
   001 – The first source register is 1, so print **R1**, a **comma** and a **space**
   0 – Bit 5 tells if this is *Register Add* or an *Immediate Add.* 0 Means this is a *Register Add*. It tells you how to interpret bits [4:0]. [4:3] are ignored and [2:0] is the second source register.
   111 – The second source register is 7, so print **R7** and a newline.

8. Any immediate values or offset6 values (NOT PCoffset) should be interpreted as 2's complement and written in decimal preceded with a # sign. DO NOT write + sign for positive numbers.

9. Any PCoffset (the PC means Program Counter) should be interpreted as a 2's complement number and added to the program counter to calculate an effective address. The value should be printed in HEX, should always be four digits (i.e. leading zeros), and always be preceded with the letter x. The first number in the "hex" file (x3000 in test1.hex) is the address of the first instruction. Each instruction is then loaded sequentially into the next memory location. The instruction register, when loaded with the instruction in question, was loaded by using the PC which MUST have contained the memory location of that instruction. Remember, the PC is incremented after the instruction is loaded but before the instruction is executed. That means you have to keep track of the address of the instruction you are interpreting. (Make your own PC variable, set it to the initial number and increment it before interpreting the output of each instruction.)

10. The branch instruction should print an upper case letter for any flag (N, Z, or P in that order) that has a 1 in the location. Note that there is no space between BR and the flags.
    a. For example:
    ```
    0000011000000110   (N=0, Z=1, P=1)
    BRZP

    0000100000000110   (N=1, Z=0, P=0)
    BRN
    ```
11. Trap vectors should print one of these three options. There are only three used trap vectors 25 for halt, 23 for reading from keyboard, and 21 for sending to the display. Note there is a tab after the word TRAP.

F025 should print TRAP	HALT
F023 should print TRAP	KEYBOARD
F021 should print TRAP	DISPLAY

12. List any registers (DR, SR, SR1, SR2, or BaseR in order that they appear in the instruction.

13. The opcode word (ADD, AND, BRZ, etc…) should always be capitalized and should be separated from the operands by a single tab.

14. Each line should have a single new line at the end.

15. Separate multiple operands with a comma then a single space as shown in the examples below.

16. Any bits that are always 1 or always 0 should not be printed in any way and should normally be ignored.

17. Ignore any reserved instructions (1101).

18. DON'T PRINT ANY BLANK LINES IN THE OUTPUT!

19. At the end of this file is some help on getting started.

20. Add a main method which simply calls printAssembly("test1.hex");

21. Compile and make sure your output matches the output shown below for the given input. I will be testing to make sure it works for other values so you should too. Your output MUST EXACTLY MATCH!!! NO DEVIATIONS AT ALL!!!

    Compile command (Spaces have been exaggerated.)
    gcc -std=c99 -Wall -o assign4 assign4.c

22. Test your program and make sure it exactly matches the output for test1.hex.

23. Change your main so it calls printAssembly("test2.hex");

24. Compile and run and make sure it matches the output shown at the end of this document for test2.hex.

25. REMOVE MAIN BEFORE UPLOADING!

26. Run ***check4*** again for good measure.

27. Submit your ***assign4.c*** file on the Web-CAT server. A link to the Web-CAT server can be found on the AsULearn page under this assignment and listed below.

        http://webcat.cs.appstate.edu:8080/Web-CAT

    The grade shown on Web-CAT will be the grade you receive.

    So if you don't get 100% and can't figure out why, ask.

test1.hex – This is an example of each instruction

For this example only, note that the program was read into x3000. This is one big program so the first "ADD" is stored at x3000, the second "ADD" is at x3001, etc…

Values in the test1.hex file	What your program should print. *Note that there should be a single TAB between the instruction and the parameter list. There should be a newline printed after each line.*
3000	This is the start address for the program. Do not print anything for the first number, but you will need it to calculate offsets.
1647	ADD R3, R1, R7
153F	ADD R2, R4, #-1
1D6F	ADD R6, R5, #15
5000	AND R0, R0, R0
567B	AND R3, R1, #-5
5923	AND R4, R4, #3
0C0F	BRNZ x3016
07F4	BRZP x2FFC
C0C0	JMP R3
C000	JMP R0
4FFC	JSR x3007
480F	JSR x301B
4080	JSRR R2
41C0	JSRR R7
21FC	LD R0, x300B
2E07	LD R7, x3017
A601	LDI R3, x3012
A180	LDI R0, x2F92
6542	LDR R2, R5, #2
6AF8	LDR R5, R3, #-8
E9FC	LEA R4, x3011
96BF	NOT R3, R2
9C7F	NOT R6, R1
C1C0	RET
8000	RTI
31FC	ST R0, x3016
3E07	ST R7, x3022
B601	STI R3, x301D
B180	STI R0, x2F9D
7543	STR R2, R5, #3
7AFC	STR R5, R3, #-4
F021	TRAP DISPLAY
F023	TRAP KEYBOARD
F025	TRAP HALT

This is not a functioning program, so don't try to execute it on any LC3 simulators. It will give errors. This program is simply meant to give you an example of all of the statements you should be able to interpret.

Add a main for testing.

YOU MUST REMOVE MAIN BEFORE SUBMITTING!

The following will run your printAssembly function and, when done, should print the information exactly as listed in the right column of the table above.

```
int main()

{

        printAssembly("test1.hex");

}
```

To compile type:

```
gcc  -Wall  -std=c99  -o assign4  assign4.c
```

To run type:

```
assign4
```

Note that you can only compile if there is a main in assign4.c, but you MUST remove main before submitting.

Here is some help for some tricky pieces.

```
//Read an integer from the file you should have open.
//I am assuming your FILE pointer is named infile.
        int inst = 0;
        fscanf(infile, "%x", &inst);

//Pull out just the opcode
        int opcode = inst >> 12 & 15;

//Pull out the destination register
        int dr = inst >> 9 & 7;

//Check if IR[5] is a 1
        if( inst  & 32 > 0)
```

Here are the functions you must create:

```
void printAdd(int instruction);
void printAnd(int instruction);
void printBr(int instruction, int pc);
void printJmpRet(int instruction);
void printJsrJsrr(int instruction, int PC);
void printLd(int instruction, int pc);
void printLdi(int instruction, int pc);
void printLdr(int instruction);
void printLea(int instruction, int pc);
void printNot(int instruction);
void printRti(int instruction);
void printSt(int instruction, int pc);
void printSti(int instruction, int pc);
void printStr(int instruction);
void printTrap(int instruction);
```

Help with printAssembly

```
void printAssembly (char const *filename)  //Stick a const in here
{

      //Open your filename
      //read PC
      while (not at end of file)
      {
            //Read instruction
            //Figure out the opcode (first 4 bits)
            //Use a big if or switch to call correct function
            //increment pc

      }
}

//You should add all of the print functions and make them simply print
//a blank line until you get around to implementing them.
```

Hex file ***test2.hex*** is the same as ***test1.hex*** with a different starting memory location. The output is listed below. **NOTE THE DIFFERENCE IN THE PCOFFSETS FROM** text1.hex.

1. In main change printAssembly("test1.hex") to printAssembly("test2.hex").
2. Recompile your program.
3. Run the program and make sure the program matches the output shown below. Pay special attention to the bold PCOFFSET values.

```
ADD   R3, R1, R7
ADD   R2, R4, #-1
ADD   R6, R5, #15
AND   R0, R0, R0
AND   R3, R1, #-5
AND   R4, R4, #3
BRNZ  x2416
BRZP  x23FC
JMP   R3
JMP   R0
JSR   x2407
JSR   x241B
JSRR  R2
JSRR  R7
LD    R0, x240B
LD    R7, x2417
LDI   R3, x2412
LDI   R0, x2392
LDR   R2, R5, #2
LDR   R5, R3, #-8
LEA   R4, x2411
NOT   R3, R2
NOT   R6, R1
RET
RTI
ST    R0, x2416
ST    R7, x2422
STI   R3, x241D
STI   R0, x239D
STR   R2, R5, #3
STR   R5, R3, #-4
TRAP  DISPLAY
TRAP  KEYBOARD
TRAP  HALT
```