Project 1 -- CDA 3101 (Spring 2014)

Worth: 100 points (10% of course grade)
Assigned: Friday, Jan 24, 2014
Due: 1:25 pm, Monday, Feb 24, 2014

1. Purpose

This project is intended to help you understand the instructions of a very simple assembly language and how to assemble programs into machine language.

2. Problem

This project has three parts. In the first part, you will write a program to take an assembly - language program and produce the corresponding machine

bits 24 - 22: opcode
bits 21 - 19: reg A
bits 18 - 16: reg B
bits 15 - 0: offsetField (16 - bit, range of - 32768 to 32767)

O- type instructions (halt, noop):
bits 24 - 22: opcode
bits 21 - 0: unused (should all be 0)

--------------------------------- --

Table 1: Description of Machine Instructions

Assembly language name for instruction	Opcode in binary (bits 24, 23, 2 2)	Action
add (R - type format)	000	add contents of regA with contents of regB, store results in destReg.
nand (R - type format)	001	nand contents of regA with contents of regB, store results in destReg.
lw (I - type format)	010	

Symbolic addresses refer to labels. For lw or sw instructions, the

```
neg1    .fill       - 1
stAddr              .fill      start                      will contain the address of start (2)
```

file, one instruction per line. Any deviation from this format (e.g. extra spaces or8empty lines) will render8your machine - code file ungradable. Any other output that8you want the program to generate (e.g. debugging output) can be printed to standard output.

4.2. Error Checking

Your assembler should catch the following errors in the assembly - language program: use of undefined labels, duplicate labels, offsetFields that don't fit in 16 bits, and unrecognized opcodes. Your assembler should exit(1) if it detects an error and exit(0) if it finishes without detecting any errors. Your assembler should NOT catch simulation - time errors, i.e. errors that would occur at the time the assembly - language program executes (e.g. branching to address - 1, infinite loops, etc.).

4.3. Test Cases

An integral (and graded) part of writing your assembler will be to write a suite of test cases to validate any LC3101 assembler. This is c ommon practice in the real world -- software companies maintain a suite of test cases for thei4(o)9(ur)7(a)7(ss)7(em)7(bl)7(er)l7(r.)7(r.)7(r.)7(r.)7(r.)7(rsC /P <</MCI7(r.)7(r()7(an)7(d)7(gr)7(ad)-4(e)9[(pr)-

Hints: the example assembly - language program above is a good case to includ e
in your test suite, though you'll need to write more test cases to get full
credit. Remember to create some test cases that test the ability of an

As with the assembler, you will write a suite of test cases to validate the LC3101 simulator.

The test cases for the simulator part of this project will be short assembly - language programs that, after being assembled into machine code, serve as input to a simulator. You will submit your suite of test cases together with your simulator, and we will grade your test suite according to how thoroughly it exercises an LC3101 simulator. Each test case may execute at most 200 instructions on a correct simulator, and your test suite may contain up to 20 test cases. These limits are much larger than needed for full credit (the solution test suite is composed of a couple test cases, each executing less than 40 instructions). See Section 7 for how your test suite will be

program halts. You may assume that the two input numbers are at most 15
bits
and are positive; this ensures that the (positive) result fits in an
LCts101
word. See the algorithm on page 252 of the   textbook for how to multiply.
Remember that shifting left by one bit is the same as adding the number
to
itself. Given the LCts101 instruction set, it's easiest to modify the
algorithm so that you avoid the right shift. Submit a versior of the
program
that computes (32766 * 10383).

Your multiplicatior program must be reasonably efficiert   -- it must be at
most
50 lines long and execute at most 1000 instructions for any valid numbers
(this
is several times longer and slower than the solution). To ac   hieve this,
you
must use a loop and shift algorithm to perform the multiplication;
algorithms
such as successive addition (e.g. multiplying 5 * 6 by adding 5 six
times)
will take too lorg.

7. Grading and Formatting

correctly   3.11 Tm [(su)-4(c)9(h)7(as)78>> BDC BT 1 0 0 1 75.024 377.93 Tm [(as)-4(s)9(em)7(bl)7(in)7(g)7(a

The best way to debug your program i   s to generate your own test cases,
figure
out the correct answers, and compare your program's output to the correct
answers. This is also one of the best ways to learn the concepts in the
project.

The student suite of test cases for the assembler an   d simulator parts of
this
project will be graded according to how thoroughly they test an LC3101
assembler or simulator. We will judge thoroughness of the test suite by
how
well it exposes potentially bugs in an assembler or simulator.

For the assem   bler test suite, we will use each test case as input to a
set
of buggy assemblers. A test case exposes a buggy assembler by causing it
to generate a different answer from a correct assembler. The test suite
is graded based on how many of the buggy ass   emblers were exposed by at
least one test case. This is known as "mutation testing" in the research
literature on automated testing.

For the simulator test suite, we will correctly assemble each test case,
then use it as input to a set of buggy simu lators. A test case exposes a buggy simulator by causing it to generate a different answer from a correct simulator. The test suite is graded based on how many of the buggy
simulators were exposed by at least one test case.

8. Turning in the Projec t

Submit you files through blackboard.
Each part should be archived in a .tar or .zip file to help with grading.

Here are the files you should submit for each project part:

1) assembler (part 1a)

buggyla

You may also choose to not use this fragment.

```
/* Assembler code fragment for LC3101 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MAXLINELENGTH 1000

int readAndParse(FILE *, char *,          char *, char *, char *, char *);
int isNumber(char *);

int
main(int argc, char *argv[])
{
```

```
	/* after doing a readAndParse, you may want to do the following to
test the
		opcode	*/
	if (!strcmp(opcode, "add")) {
		/* do whatever you need to do for opcode "add" */
	}

	return(0);
}

/*
 * Read and parse a line of the assembly - language file.  Fields are
returned
 * in label, opcode, arg0, arg1, arg2 (these s trings must have memory
already
 * allocated to them).
 *
 * Return values:
 *     0 if reached end of file
 *     1 if all went well
 *
 * exit(1) if line is too long.
 */
int
readAndParse(FILE *inFilePtr, char *label, char *opcode, char *arg0,
	char *arg1, char *arg2)
{
	char line[MAXLINELENGTH];
	char *ptr = line;

	/* delete prior values */
	label[0] = opcode[0] = arg0[0] = arg1[0] = arg2[0] = ' \ 0';

	/* read the line from the assembly - language file */
	if (fgets (line, MAXLINELENGTH, inFilePtr) == NULL) {
	   /* reached end of file */
		return(0);
	}

	/* check for line too long (by looking for a \ n) */
	if (strchr(line, ' \ n') == NULL) {
		/* line too long */
	   printf("error: line too long \ n");
	   exit(1);
	}

	/* is there a label? */
	ptr = line;
	if (sscanf(ptr, "%[^ \ t \ n ]", label)) {
	   /* successfully read label; advance pointer over the label */
		ptr += strlen(label);
	}

	/*
```

```
         * Parse the rest of th          e line.  Would be nice to have real regular
         * expressions, but scanf will suffice.
         */
        sscanf(ptr, "%*[       \ t \ n ]%[^   \ t \ n ]%*[   \ t \ n ]%[^   \ t \ n ]%*[   \ t \ n ]%[^   \ t \ n
]%*[  \ t \ n ]%[^   \ t \ n ]",
                opcode, arg0, arg1, arg2);
        return(1);
}

int
is  Number(char *string)
{
        /* return 1 if string is a number */
        int i;
        return( (sscanf(string, "%d", &i)) == 1);
}
```

10. Code Fragment for Simulator

Here is some C code that may help you write the simulator. Again, you should
take th is merely as a hint. You may have to re - code this to make it do exactly
what you want, but this should help you get started. Remember not to
change stateStruct or printState.

```
/* instruction       - level simulator for LC3101 */

#include <stdio.h>
#inclu   de <string.h>

#define NUMMEMORY 65536 /* maximum number of words in memory */
#define NUMREGS 8 /* number of machine registers */
#define MAXLINELENGTH 1000

typedef struct stateStruct {
        int pc;
        int mem[NUMMEMORY];
        int reg[NUMREGS];
        int numMemory;
} stateType;

void printState(stateType *);

int
main(int argc, char *argv[])
{
        char line[MAXLINELENGTH];
        stateType state;
        FILE *filePtr;

        if (argc != 2) {
```

```
        printf("error: usage: %s <machine-code file>\n", argv[0]);
        exit(1);
    }

    filePtr = fopen(argv[1], "r");
    if (filePtr == NULL) {
        printf("error: can't open file %s", argv[1]);
        perror("fopen");
        exit(1);
    }

    /* read in the entire machine-code file into memory */
    for (state.numMemory = 0; fgets(line, MAXLINELENGTH, filePtr) !=
NULL;
        state.numMemory++) {
        if (sscanf(line, "%d", state.mem+state.numMemory) != 1) {
            printf("error in reading address %d\n", state.numMemory);
            exit(1);
        }
        printf("memory[%d]=%d\n", state.numMemory,
state.mem[state.numMemory]);
    }

    return(0);
}

void
printState(stateType *statePtr)
{
    int i;
    printf("\n@@@\nstate:\n");
    printf("\tpc %d\n", statePtr->pc);
    printf("\tmemory:\n");
        for (i=0; i<statePtr->numMemory; i++) {
            printf("\t\tmem[ %d ] %d\n", i, statePtr->mem[i]);
        }
    printf("\tregisters:\n");
        for (i=0; i<NUMREGS; i++) {
            printf("\t\treg[ %d ] %d\n", i, statePtr->reg[i]);
        }
    printf("end state\n");
}
```

11. Programming Tips

Here are a few programming tips for writing C/C++ programs to manipulate bits:

1) To indicate a hexadecimal constant in, precede the number by 0x. For example, 27 decimal is 0x1b in hexadecimal.

2) The value of the expression (a >> b) is the number "a" shifted right by "b"

bits. Neither a nor b are changed. E.g. (25 >> 2) is 6. Note that 25 is
11001 in
binary, and 6 is 110 in binary.

3) The value of the expression (a << b) is the number "a" shifted left by
"b"
bits. Neither a nor b are changed. E.g. (25 << 2) is 100. Note that 25 is
11001
in binary, and 100 is 1100100 in binary.

4) To find the value of the expression (a & b), perform a logical AND on
each
bit of a and b (i.e. bit 31 of a ANDED with bit 31 of b, bit 30 of a
ANDED with
bit 30 of b, etc.). E.g. (25 & 11) is 9, since:

```
      11001 (binary)
   & 01011 (binary)
---------------------
  =  01001 (binary), which is 9 decimal.
```

5) To find the value of the expression (a | b), perform a logical OR on
each bit
of a and b (i.e. bit 31 of a ORED with bit 31 of b, bit 30 of a ORED with
bit 30
of b, etc.). E.g. (25 | 11) is 27, since:

```
      11001 (binary)
   & 01011 (binary)
---------------------
  =  11011 (binary), which is 27 decimal.
```

6) ~a is the bit-wise complement of a (a is not changed).

Use these operations to create and manipulate machine-code. E.g. to look
at bit
3 of the variable a, you might do: (a>>3) & 0x1. To look at bits (bits
15-12) of
a 16-bit word, you could do: (a>>12) & 0xF. To put a 6 into bits 5-3 and
a 3
into bits 2-1, you could do: (6<<3) | (3<<1). If you're not sure what an
operation is doing, print some intermediate results to help you debug.

------

12. Example Run of Simulator

```
memory[0]=8454151
memory[1]=9043971
memory[2]=655361
memory[3]=16842754
memory[4]=16842749
memory[5]=29360128
memory[6]=25165824
```

memory[7]=5
memory[8]= - 1
memory[9]=2

@@@
state:
pc 0
memory:
mem[0] 8454151
mem[1] 904397 1
mem[2] 655361
mem[3] 16842754
mem[4] 16842749
mem[5] 29360128
mem[6] 25165824
mem[7] 5
mem[8] - 1
mem[9] 2
registers:
reg[0] 0
reg[1] 0
reg[2] 0
reg[3] 0
reg[4] 0
reg[5] 0
reg[6] 0
reg[7] 0
end state

@@@
state:
pc 1
memory:
mem[0] 8454151
mem[1] 9043971
mem[2] 655361
mem[3] 16842754
mem[4] 16842749
mem[5] 29360128
mem[6] 25165824
mem[7] 5
mem[8] - 1
mem[9] 2
register s:
reg[0] 0
reg[1] 5
reg[2] 0
reg[3] 0
reg[4] 0
reg[5] 0
reg[6] 0
reg[7] 0
end state

@@@
state:
pc 2
memory:
mem[0] 8454151
mem[1] 9043971
mem[2] 655361
mem[3] 16842754
mem[4] 168427 49
mem[5] 29360128
mem[6] 25165824
mem[7] 5
mem[8] - 1
mem[9] 2
registers:
reg[0] 0
reg[1] 5
reg[2] - 1
reg[3] 0
reg[4] 0
reg[5] 0
reg[6] 0
reg[7] 0
end state

@@@
state:
pc 3
memory:
mem[0] 8454151
mem[1] 9043971
mem[2] 655361
mem[3] 16842754
mem[4] 16842749
mem[5] 29360128
mem[6] 25165824
mem[7] 5
mem[8] - 1
mem[9] 2
registers:
reg[0] 0
reg[1] 4
reg[2] - 1
reg[3] 0
reg[4] 0
reg[5] 0
reg[6] 0
reg[7] 0
end state

@@@
state:
pc 4

```
        memory:
                mem[ 0 ] 8454151
                mem[ 1 ] 9043971
                mem[ 2 ] 655361
                mem[ 3 ] 16842754
                mem[ 4 ] 16842749
                mem[ 5 ] 29360128
                mem[ 6 ] 25165824
                mem[ 7 ] 5
                mem[ 8 ]    - 1
                mem[ 9 ] 2
        registers:
                reg[ 0 ] 0
                reg[ 1 ] 4
                reg[ 2 ]      - 1
                reg[ 3 ] 0
                reg[ 4 ] 0
                reg[ 5 ] 0
                reg[ 6 ] 0
                reg[ 7 ] 0
end state

@@@
state:
        pc 2
        memory:
                mem[ 0 ] 8454151
                mem[ 1 ] 9043971
                mem[ 2 ] 65536    1
                mem[ 3 ] 16842754
                mem[ 4 ] 16842749
                mem[ 5 ] 29360128
                mem[ 6 ] 25165824
                mem[ 7 ] 5
                mem[ 8 ]    - 1
                mem[ 9 ] 2
        registers:
                reg[ 0 ] 0
                reg[ 1 ] 4
                reg[ 2 ]      - 1
                reg[ 3 ] 0
                reg[ 4 ] 0
                reg[ 5 ] 0
                reg[ 6 ] 0
                reg[ 7 ] 0
end state

@@@
state:
        pc 3
        memory:
                mem[ 0 ] 8454151
                mem[ 1 ] 9043971
                mem[ 2 ] 655361
```

```
                mem[ 3 ] 16842754
                mem[ 4 ] 16842749
                mem[ 5 ] 29360128
                mem[ 6 ] 25165824
                mem[ 7 ] 5
                mem[ 8 ]     - 1
                mem[ 9 ] 2
        registers:
                reg[ 0 ] 0
                reg[ 1 ] 3
                reg[ 2 ]       - 1
                reg[ 3 ] 0
                reg[ 4 ] 0
                reg[ 5 ] 0
                reg[ 6 ] 0
                reg[ 7 ] 0
end state

@@@
state:
        pc 4
        memory:
                mem[ 0 ] 8454151
                mem[ 1 ] 9043971
                mem[ 2 ] 655361
                mem[ 3 ] 16842754
                mem[ 4 ] 16842749
                mem[ 5 ] 29    360128
                mem[ 6 ] 25165824
                mem[ 7 ] 5
                mem[ 8 ]     - 1
                mem[ 9 ] 2
        registers:
                reg[ 0 ] 0
                reg[ 1 ] 3
                reg[ 2 ]       - 1
                reg[ 3 ] 0
                reg[ 4 ] 0
                reg[ 5 ] 0
                reg[ 6 ] 0
                reg[ 7 ] 0
end state

@@@
state:
        pc 2
        memory:
                mem[ 0 ] 8454     151
                mem[ 1 ] 9043971
                mem[ 2 ] 655361
                mem[ 3 ] 16842754
                mem[ 4 ] 16842749
                mem[ 5 ] 29360128
                mem[ 6 ] 25165824
```

mem[7] 5
mem[8] - 1
mem[9] 2
registers:
reg[0] 0
reg[1] 3
reg[2] - 1
reg[3] 0
reg[4] 0
reg[5] 0
reg[6] 0
reg[7] 0
end state

@@@
state:
pc 3
memory:
mem[0] 8454151
mem[1] 9043971
mem[2] 655361
mem[3] 16842754
mem[4] 16842749
mem[5] 29360128
mem[6] 25165824
mem[7] 5
mem[8] - 1
mem[9] 2
registers:
reg[0] 0
reg[1] 2
reg[2] - 1
reg[3] 0
reg[4] 0
reg[5] 0
reg[6] 0
reg[7] 0
end state

@@@
state:
pc 4
memory:
mem[0] 8454151
mem[1] 9043971
mem[2] 655361
mem[3] 168 42754
mem[4] 16842749
mem[5] 29360128
mem[6] 25165824
mem[7] 5
mem[8] - 1
mem[9] 2
registers:

reg[0] 0
reg[1] 2
reg[2] - 1
reg[3] 0
reg[4] 0
reg[5] 0
reg[6] 0
reg[7] 0
end state

@@@
state:
pc 2
memory:
mem[0] 8454151
mem[1] 9043971
mem[2] 655361
mem[3] 16842754
mem[4] 16842749
mem[5] 29360128
mem[6] 25165824
mem[7] 5
mem[8] - 1
mem[9] 2
registers:
reg[0] 0
reg[1] 2
r eg[2] - 1
reg[3] 0
reg[4] 0
reg[5] 0
reg[6] 0
reg[7] 0
end state

@@@
state:
pc 3
memory:
mem[0] 8454151
mem[1] 9043971
mem[2] 655361
mem[3] 16842754
mem[4] 16842749
mem[5] 29360128
mem[6] 25165824
mem[7] 5
mem[8] - 1
mem[9] 2
registers:
reg[0] 0
reg[1] 1
reg[2] - 1
reg[3] 0

reg[4] 0
reg[5] 0
reg[6] 0
reg[7] 0
end state

@@@
state:
pc 4
memory:
mem[0] 8454151
mem[1] 9 043971
mem[2] 655361
mem[3] 16842754
mem[4] 16842749
mem[5] 29360128
mem[6] 25165824
mem[7] 5
mem[8] - 1
mem[9] 2
registers:
reg[0] 0
reg[1] 1
reg[2] - 1
reg[3] 0
reg[4] 0
reg[5] 0
re g[6] 0
reg[7] 0
end state

@@@
state:
pc 2
memory:
mem[0] 8454151
mem[1] 9043971
mem[2] 655361
mem[3] 16842754
mem[4] 16842749
mem[5] 29360128
mem[6] 25165824
mem[7] 5
mem[8] - 1
mem[9] 2
re gisters:
reg[0] 0
reg[1] 1
reg[2] - 1
reg[3] 0
reg[4] 0
reg[5] 0
reg[6] 0
reg[7] 0

end state

@@@
state:
pc 3
memory:
mem[0] 8454151
mem[1] 9043971
mem[2] 655361
mem[3] 16842754
mem[4] 16842749
mem[5] 29360128
mem[6] 25165824
mem[7] 5
mem[8] - 1
mem[9] 2
registers:
reg[0] 0
reg[1] 0
reg[2] - 1
reg[3] 0
reg[4] 0
reg[5] 0
reg[6] 0
reg[7] 0
end state

@@@
state:
pc 6
memory:
mem[0] 8454151
mem[1] 9043971
mem[2] 655361
mem[3] 16842754
mem[4] 16842749
mem[5] 29360128
mem[6] 25165824
mem[7] 5
mem[8] - 1
mem[9] 2
registers:
reg[0] 0
reg[1] 0
reg[2] - 1
reg [3] 0
reg[4] 0
reg[5] 0
reg[6] 0
reg[7] 0
end state
machine halted
total of 17 instructions executed
final state of machine:

@@@
state:

pc 7

memory:

mem[0] 8454151
mem[1] 9043971
mem[2] 655361
mem[3] 16 842754
mem[4] 16842749
mem[5] 29360128
mem[6] 25165824
mem[7] 5
mem[8] - 1
mem[9] 2

registers:

reg[0] 0
reg[1] 0
reg[2] - 1
reg[3] 0
reg[4] 0
reg[5] 0
reg[6] 0
reg[7] 0

end state