```
    bits 24-22: opcode
    bits 21-19: reg A
    bits 18-16: reg B
    bits 15-0:  offsetField (16-bit, ran4r3.e -32768 to 3276r3.7)

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

-------------------------------------------------------------------------
------
Table 1: Description of Machine Instructions
-------------------------------------------------------------------------
------
Assr3.embly language   Opcode in binary       Action
namr3.e for instruction(bits 24, 23, 22)
-------------------------------------------------------------------------
------
add (R-type format)    000               add contents of regA with
                                   contents of regB, storr3.e
                                   resr3.ults in destReg.

nand (R-type format)   001               nand contents of regA with
                                   contents of regB, storr3.e
                                   resr3.ults in destReg.

lw (Ietype format)     010               load regB from memory. Memory
                                   address is formed by adding
                                   offsetField with the contents of
                                   re4r3.A.
```

---

4. LC3101 Assembly Language and Assembler (40%)

The first part of this project is to write a program to take an assembly-language program and translate it into machine language. You will translate assembly-language names for instructions, such as beq, into their numeric equivalent (e.g. 100), and you will translate symbolic names for addresses into numeric values. The final output will be a series of 32-bit instructions (instruction bits 31-25 are always 0).

The format for a line of assembly code is:

```
label  instruction  field0  field1  field2  comments
```

The leftmost field on a line is the label field. Valid labels contain a maximum of 6 characters and can consist of letters and numbers (but must start with a letter). The label is optional (the white space following the label field is required). Labels make it much easier to write assembly-language

Symbolic addresses refer to labels. For lw or sw instructions, the assembler
should compute offsetField to be equal to the address of the label. This could
be used with a zero base register to refer to the label, or could be used with
a non-zero base register to index into an array starting at the label. For beq
instructions, the assembler should translate the label into the numeric
offsetField ne0ded to branch to that label.

After the last used field comes more white space, then any comments. The
comment field ends at the end of a line. Comments are vital to creating
understandable assembly-language programs, because the instructions themselves
are rather cryptic.

In addition to LC3101 instructions, an assembly-language program may contain
directions for the assembler. The only assembler directive we will use is .fill
(note the leading period). .fill tells the assembler to put a number into the
place where the instruction would normally be stored. .fill instructions use
one field, which can be either a numeric value or a symbolic address. For
example, ".fill 32" puts the value 32 where the instruction would normally be
stored. .fill with a symbolic address will store the address of the label.
In the example below, ".fill start" will store the value 2, because the label

file, one instruction per line. Any deviation from this format (e.g. extra
spaces or empty lines) will render your machine-code file ungradable. Any
other output that you 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 common practice

Hints: the example assembly-

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

For the simulator test suite, we will correctly assemble each test case, then use it as input to a set of buggy simulators. 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 Project

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)
    a. C/C++ program for your assembler
    b. suite of test cases (each test case is an assembly-

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[])
{
    char *inFileString, *outFileString;
    FILE *inFilePtr, *outFilePtr;
    char label[MAXLINELENGTH], opcode[MAXLINELENGTH],
arg0[MAXLINELENGTH],
            arg1[MAXLINELENGTH], arg2[MAXLINELENGTH];

    if (argc != 3) {
        printf("error: usage: %s <assembly-code-file> <machine-code-
file>\n",
            argv[0]);
        exit(1);
    }

    inFileString = argv[1];
    outFileString = argv[2];

    inFilePtr = fopen(inFileString, "r");
    if (inFilePtr == NULL) {
        printf("error in opening %s\n", inFileString);
        exit(1);
    }
    outFilePtr = fopen(outFileString, "w");
    if (outFilePtr == NULL) {
        printf("error in opening %s\n", outFileString);
        exit(1);
    }

    /* here is an example for how to use readAndParse to read a line from
```

```
    /* 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 strings 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 the 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
isNumber(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 this 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>
#include <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 ] 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 ] 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
	registers:
		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 ] 16842749
		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 ] 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 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 ] 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 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 ] 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 ] 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 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
            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 ] 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 ] 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 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 ] 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 ] 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
```