Dalhousie University
Department of Electrical and Computer Engineering

ECED 3403 – Computer Architecture

Assignment 1: Design and implementation of an MSP-430 assembler

1 Objectives

When a new machine such as the *Eagle* in *The Soul of a New Machine* is developed, software is required to allow software designers to design and write software for the machine. Computer engineers are often responsible for designing and writing such software, including tools known as *assemblers*, which translate *assembly-language* programs into *machine code*. The machine code program can then be loaded into the machine's memory for subsequent execution.

In those situations where the new machine is unable to run the assembler (for example, it is still under development), a *cross-assembler* is required.[1] The cross-assembler runs on a second machine, producing a program that can be loaded and executed on the target machine or an emulator of the target machine.

This assignment requires the design, implementation, and testing of a two-pass cross-assembler for the Texas Instrument's MSP-430 microcomputer.[2] The design is to be implemented in a high-level language (either C or C++). It can run on the machine of your choice. All examples in class will be in C.

2 The Assembler

The assembler is to read MSP-430 assembly-language files containing assembly-language *records* and produce executable modules containing loadable MSP-430 machine code. It is to operate in two passes: the first pass builds the symbol table, while the second generates the executable module.

2.1 The assembly language record

2.1.1 Record format

An assembly-language file consists of one or more *records* containing instructions and data for the assembler to translate into machine-readable records. The record is to be *free-format*, meaning that it has no fixed fields. All records have the same format, defined as follows:

Record = (*Label*) + ([*Instruction* | *Directive*]) + (*Operand*) + (*; Comment*)
Label = Alphabetic + 0 {*Alphanumeric*} *30*
*Instruction = * An instruction mnemonic, see section 2.1.2 **
*Directive = * An assembler directive, described in section 2.1.3 **
*Operand = * The operands associated with the Instruction or Directive **
*Comment = * Text associated with the record – ignored by the assembler **

[1] This is often the situation when code is required for an entirely new machine, one without an operating system, editor, or tools such as assemblers or compilers.
[2] Unless you have a MSP-430 (with an operating system, editor, and C compiler), your assembler will produce MSP-430 machine code on a machine other than the MSP-430.

Alphabetic = [*A..Z* | *a..z* | _]
Alphanumeric = [*A..Z* | *a..z* | *0..9* | _]

Instructions and *Directives* should be treated as case insensitive (that is, the value can be upper case, lower case, or some combination thereof). However, a *Label*, if it exists, is case sensitive, meaning that the label **`Alpha`** is not the same as the label **`ALPHA`**.

The following are not supported by the assembler:

- External references (i.e., references to *Label*s in other files)
- Include files (i.e., external files to be "copied into" the program for assembly)
- In-line arithmetic expressions (i.e., operands containing arithmetic operators)

2.1.2 Instructions

The MSP-430 instruction set is to be used for this assignment. A detailed description of the MSP-430's Instruction Set Architecture (ISA) can be found in *Introduction to the MSP430 Instruction Set Architecture* (http://lh.ece.dal.ca/eced3403/MSP430_ISA.pdf).

Additional information can be found in *Texas Instruments MSP430x2xx Family User's Guide* (http://www.ti.com/lit/ug/slau144j/slau144j.pdf).

The ISA will be discussed in class. Other material relating to the ISA is available from the course website.

2.1.3 Directives

A directive (or *pseudo-instruction*) is a command in a record that is processed by the assembler (i.e., it directs the assembler to do something). It has no corresponding machine instruction, although it can produce machine code. The directives to be supported by this assignment:

ALIGN
Increments the location counter to the next even-byte address if the location counter is odd.

BSS *Value*
A block of memory of *Value* bytes is reserved. If there is a *Label*, it is stored in the symbol table with the value of the location counter. The location counter is increased by the specified number of bytes. The mnemonic **`BSS`** means Block Starting with Symbol.[3]

BYTE *Value*
One byte is stored in the memory location associated with the location counter. A *Value* larger than 8-bits in length is an error. If there is a *Label*, it is stored in the symbol table along with the location counter. The location counter is increased by one.

[3] Some assemblers support a **`BES`** directive. This refers to a block ending with a symbol (i.e., *Label*). It can be used for reserving stack space. Two interesting quiz questions: how could stack space be created using this assembler and why does a stack have its label at the end of a block of memory?

END

Denotes the end of the program. Any instructions or data that follow this record are ignored.

EQU *Value*

The record's *Label* is equated with (i.e., assigned to) the *Value*. The *Label* and *Value* are stored in the symbol table. A *Value* is an 8-bit or 16-bit quantity. Any reference to the *Label* in the program is replaced by the *Value*. A *Label* is required. The location counter is not incremented.

ORG *Value*

Change current location counter value to the address specified in *Value*.

STRING *String*

String is a string enclosed in double-quote marks. The string cannot exceed the page width. The characters in the string are to be stored as 8-bit ASCII values. If there is a *Label*, it is stored in the symbol table along with the location counter. The location counter is increased by the number of characters in the string. The string is not NUL-terminated. A single character can be represented as a one-character string; it is stored as one-byte.

WORD *Value*

Two bytes are stored in consecutive memory locations, starting at the location specified by the current value of the location counter. If there is a *Label*, it is stored in the symbol table along with the location counter. The location counter is increased by two bytes. Note that 16-bit quantities should fall on even-byte boundaries.

The following symbols are associated with the directives:

String = '"' + 1 {Char} + '"'
Value = [*UWord* | *SWord*]
UWord = [[*$0 .. $FFFF*] | [*0 .. 65535*]]
SWord = [*-32768 .. +32767*]
Char = [*Alphanumeric* | *Escaped*]
Escaped = '\' + Alphanumeric

Notes:

- The default initial value of the location counter should be zero.
- The location counter is unsigned and cannot exceed 65,535 (0xFFFF).
- The location counter should be incremented by the number of bytes associated with the *Value* or *String* length.
- Directives are reserved words (like the instructions) and cannot be used as labels.
- Duplicate labels are not permitted.
- Hexadecimal numbers are prefixed with '$'.
- Useful escape sequences are '\0' (NUL), '\t' (tab), '\r' (return or Enter), and '\n' (new line).

2.2 Execution

This is a two-pass assembler. The first pass checks each non-comment record for validity, stores the label in the symbol table (if present), and increments the location counter, while the second pass rereads the file, generating the corresponding machine code for each non-comment record.

2.2.1 First pass

The first pass repeats the following until end-of-file or an **END** directive is found:

1. Read the next record.
2. Comment records are to be ignored and the location counter is not to be incremented. A *Label* must be stored in the symbol table along with the value of the location counter (i.e., the *Label*'s address). A duplicate label is an error.
3. The next field must be an *Instruction*, *Directive*, or *Comment*. Anything else is an error.
4. If there is an *Instruction* or *Directive* and it requires an *Operand* field, the operand must exist and be correct. An invalid or unexpected *Operand* should cause an error diagnostic to be issued. It is possible that the *Operand* is undefined until the second pass (if it is a *Label* that is a forward reference). The location counter should be incremented by the number of bytes required by the instruction or directive.
5. *Comments*, if they exist, are to be ignored.

If an error is found in a record, the subsequent records should still be processed until the end-of-file or **END**.

2.2.2 Second pass

The second pass should not be performed if one or more errors are detected on the first pass.

The second pass repeats the following until end-of-file or an **END** directive is found:

1. Read the next record.
2. If the record is a comment, it should be ignored and step 1 repeated.
3. Ignore the *Label* if there is one (it was handled during the first pass).
4. If the record contains an *Instruction*, find the corresponding opcode. If the *Instruction* requires one or two *Operand*s, check them and if valid, determine the registers, addressing modes, as well as any source or destination operands. Combine the opcode, registers, and addressing modes to create the machine instruction. Then emit the location counter, machine instruction, and required operands to a machine code record (see below, section 2.3).

   If *Operand*s are not required, the location counter and opcode can be emitted directly as a machine code record. If an *Operand* refers to a *Label* that does not have a corresponding symbol-table entry, an error diagnostic should be displayed.

5. If the record contains a *Directive*, perform the steps required. If an *Operand* is required, check that it exists and use the required values.

Notes:

- If the *Value* is 8-bits but the opcode expects 16-bit, prefix the *Value* with 0x00;[4] however, if the opcode only expects an 8-bit *Value* but a 16-bit value is supplied, it is an error.
- Error diagnostics should be generated for missing *Operand*s (the number depends upon the *Instruction* or *Directive*) or a supplied *Operand* (if *Operand*s are not required by the *Instruction* or *Directive*).
- When the *Operand* is a *Label*, the value of the *Label* should be taken from the symbol table; for example, if **Alpha** has a symbol table value of **0x014A**, this value should be used.
- The different addressing modes will require special attention.

2.3 Output

2.3.1 Executable (machine code) format: The S-record

The output from the assembler is a loadable, executable file containing S-records. Motorola designed S-records as a means of transferring executable files from a source machine to a target machine without having to worry about ASCII control characters (other than CR, LF, and NUL).

Details on S-records can be found at:

http://en.wikipedia.org/wiki/SREC_(file_format)

http://www.amelek.gda.pl/avr/uisp/srecord.htm

2.3.2 List file

In addition to the .s19 file, a second file, the list file (.LIS), should contain each input record and the corresponding assembled record and its address. Diagnostics should also be written to this file.

2.4 Suggestions

It is advisable to give the machine code file an extension of .s19 as an extension of .exe may trigger various protection mechanisms in Windows and it won't be easy to inspect the output using an editor.

It is not necessary to read the entire assembly language program into memory – simply read and process one record at a time. Similarly, it is not necessary to close and reopen the assembly language file at the start of the second pass – simply rewind the file (use **lseek()**).

[4] The assembler prefixes hexadecimal values with a '$', whereas C/C++ use '0x'.

3 Requirements

The assembler is to take MSP-430 assembler files and assemble them into MSP-430 machine code files. The assembler should recognize all instructions and addressing modes. If errors are detected during either the first or second pass, these should be written to a listing file (.LIS), while successfully assembled files should have each record and its corresponding machine code written as well as the symbol table written to the listing file.

4 Marking

The assignment will be graded out of 25 using the following marking scheme:

Design
The design description must include a brief introduction as to the purpose of the software and a description of the algorithms and data structures used.

Total points: 7.

Software
A fully commented, indented, magic-numberless, tidy piece of software that meets the requirements described above and follows the design description.

Total points: 13.

Testing
A set of tests demonstrating that the software operates according to the design description. Some of the tests should exercise the software. The tests must include the name of the test, its purpose or objective, the test configuration, and the test results.

Total points: 5.

The assignment must be submitted on paper.

5 Important Dates

Available: 2 May 2017

Due: 30 May 2017 (230pm at start of lab)

Demonstration: 30 May 2017

Late assignments will be penalized two points per week or fraction thereof.

6 Miscellaneous

This assignment is to be completed individually.

It is not necessary to support the emulated instructions as they can be hand-translated.

Do *not* discard this work when completed, as it will be used with the remaining assignments.

If you are having *any* difficulty with this assignment or the course, *please* contact Dr. Hughes as soon as possible.

This is a non-trivial assignment. It should be started as soon as it is made available.