Dalhousie University
Department of Electrical and Computer Engineering

ECED 3403 – Computer Architecture

Assignment 1: Designing, implementing, and testing a preprocessor for handling XM2 legacy instructions

1 The problem

This summer, we are using a 16-bit load-and-store RISC emulator known as X-Makina to help us understand the fundamentals of computer architecture and the relationship between hardware and software. It combines the characteristics of several different machines, including Intel's x-86, ARM's Cortex, and TI's MSP 430). In addition to the emulator, there is also an assembler which takes assembly-language programs and assembles them into machine code.

X-Makina underwent a redesign this past winter. The upgraded version (formerly XM2, now XM3) has a new exception handling subsystem, four new instructions:

CEX (Conditional Execution): Execute the next set of instructions conditionally.

SETPRI (Set Priority): Set the current machine priority to a value between 0 and 7.

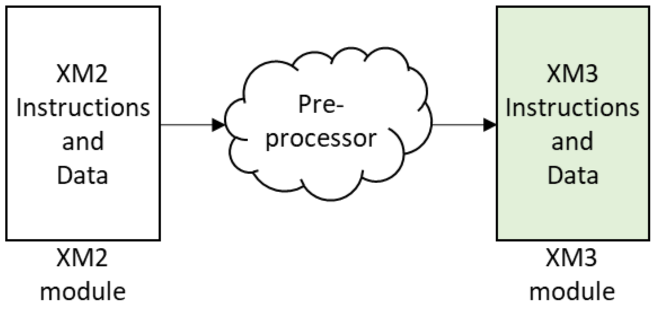
SETCC (Set Condition Code): Set the one or more of the condition codes and sleep-bit state.

CLRCC (Clear Condition Code): Clear the one or more of the condition codes and sleep-bit state.

For as useful as these instructions are, they do pose a problem because some of them are replacing existing instructions that are no longer supported. In other words, we have a problem with legacy software, for example:

- Conditional branching instructions, such as BEQ (Branch if Equal), have been replaced by CEX (Conditional Execution).
- Priority setting instructions (SPL0 through SPL7) need to be supported by the equivalent SETPRI instruction (SETPRI #0 through SPRI #7).
- The four instructions for setting condition codes (SEC, SEV, SEN, and CLZ) have been replaced by the single instruction SETCC.
- The four instructions for clearing condition codes (CLC, CLV, CLN, and CLZ) have been replaced by the single instruction CLRCC.

This means that we need a preprocessor that takes some of X-Makina's existing XM2 instructions and translates them into their XM3 equivalents:



The resulting XM3 module can then be assembled into an XM3 load module by the XM3 assembler.

2 Objective

You are to write an XM2-to-XM3 preprocessor.

Creating a solution for this problem is a three-step process. First, a solution must be designed, then the design must be implemented, and finally, the design must be tested. The first step is to be completed before the next two steps are undertaken (this is your design document). Once the design is complete, you can proceed to the implementation and testing steps. The designs are to be implemented in a high-level language (although C is preferred, C++ is permitted) and run on the machine of your choice.

Although your final implementation may have deviated from the original design, by doing the design, you will get insights into the problem and its possible solution.

3 The preprocessor

The preprocessor is a stand-alone program which will take a legacy XM2 assembly language program as input and convert any "old" XM2 instructions into their XM3 equivalents. All other XM2 instructions should be left unchanged, for example:

Old XM2 record	New XM3 record	Comments
`ADD R1,R2`	`ADD R1,R2`	Unchanged, since ADD instruction is not affected
`SEC`	`SETCC C`	Changed, SEC replaced by SETCC C in XM3
`SPL5`	`SETPRI #5`	Changed, SPL5
`BRA Loop`	`BRA Loop`	Unchanged, since BRA instruction is not affect

The assembly language files consist of records (a stream of characters terminated by a newline character). If a record from the XM2 input file contains an instruction (such as ADD, SEC, or SPL5), the preprocessor must check the instruction to determine if it should be converted following these rules:

1. If the XM2 input record contains an instruction to be converted, it should be converted to its XM3 equivalent and a new record with the converted instruction written to the XM3 output file. The XM2 record can be discarded or written to the XM3 as a comment.
2. If the XM2 input record does not contain an instruction to be converted, the XM2 record can be written directly to the XM3 output file.

3.1.1 XM2 instructions to be converted

The preprocessor is to read a record from an existing XM2 file, scan the record for an instruction, and, if it is to be converted, translate it into the equivalent XM3 instruction. The following table lists the XM2 instructions to be converted and their XM3 equivalents (XM2's branching instructions require two XM3 instructions):

XM2 instruction	**XM3 instruction**	**Description**
`SPLn`	`SETPRI #n`	Set the current CPU priority to 'n'
`CLC`	`CLRCC C`	Clear carry bit
`CLN`	`CLRCC N`	Clear negative bit
`CLZ`	`CLRCC Z`	Clear zero bit
`CLV`	`CLRCC V`	Clear overflow bit
`SEC`	`SETCC C`	Set carry bit
`SEN`	`SETCC N`	Set negative bit
`SEZ`	`SETCC Z`	Set zero bit
`SEV`	`SETCC V`	Set overflow bit

`BEQ|BZ label`	`CEX EQ,#1,#0`
	`BRA label`
		Branch equal | Branch zero

`BNE|BNZ label`	`CEX NE,#1,#0`
	`BRA label`
		Branch not equal | Branch not zero

`BGE label`	`CEX GE,#1,#0`
	`BRA label`
		Branch greater than or equal

`BLT label`	`CEX LT,#1,#0`
	`BRA label`
		Branch less than

`BC label`	`CEX CS,#1,#0`
	`BRA label`
		Branch carry set

`BNC label`	`CEX CC,#1,#0`
	`BRA label`
		Branch carry clear

`BN label`	`CEX MI,#1,#0`
	`BRA label`
		Branch negative (minus)

In addition to the above, the following eight XM2 instructions should also be converted to their XM3 equivalents:

XM2 instruction	**XM3 instruction**	**Description**
`CALL subr`	`BL subr`	Branch using link register to subr
`PULL arg`	`LD R6+,arg`	Stack Pull (POP) arg
`PUSH arg`	`ST arg,-R6`	Stack Push arg
`RET`	`MOV R5,R7`	Return from subroutine or ISR
`JUMP arg`	`MOV arg,R7`	Jump to location specified in arg
`CLR.B arg`	`MOVL #0,arg`	Clear low byte of arg
`CLR arg`	`MOVLZ #0,arg`	Clear arg
`CLR.W arg`	`MOVLZ #0,arg`	Clear arg

Notes regarding the XM3 output records:

- The '#' (hash) symbol must precede the numbers used in the XM3 instruction. For example, the number '3' required for SPL3 must be written to the output files as "#3". There are no spaces between the '#' and the number. Since '#' is used to denote hexadecimal numbers (valid sequences are #0 through #9 and #A through #F) and the above instructions only use #0 through #7, you may substitute '#' with '$' (decimal; valid sequences are $0 through $9).

- White space (e.g., spaces or tabs) is not allowed to precede or follow the commas (',') used with the CEX instruction. There are no spaces between the condition specifier (such as EQ), the true-count (#1), and the false-count (#0).
- White space is not allowed to precede or follow the arguments in the MOV (move), LD (load), or ST (store) instructions. For example, the arguments for PUSH r3 should be written as "r3,-R6".
- There must be at least one white space between any instruction (such as CLRCC) and its argument.

It is *not* the responsibility of the preprocessor to check the validity of the instruction arguments (such as label, subr, or arg), that is the job of the XM3 assembler. However, if the argument is missing, the preprocessor should issue a warning.

As an example, the subroutine call instruction, CALL, has an argument (a label specifying the entry point of the subroutine, this must be used as the argument for the BL (Branch with Link) instruction:

XM2 record:	`CALL AddFunc`	`; Call to AddFunc()`
XM3 record:	`BL AddFunc`	`; Call to AddFunc()`

`AddFunc` should be a label. It is not the responsibility of the preprocessor to determine if it exists – this is the responsibility of the assembler.

You can run your XM3 output file through the XM3 assembler to check the validity of your results. If there is an error, *make sure the XM2 input file is correct before modifying your preprocessor*.

3.2 Requirements

The preprocessor is to take an X-Makina assembler file and replace the XM2 instructions specified in section 3.1.1 with their XM3 counterparts. The preprocessor should recognize all XM2 instructions, regardless of case. If errors are detected, they should be flagged; however, pre-assembly should continue for the remaining records. The output of the preprocessor should be free of the unsupported XM2 instructions and ready for the assembler.

Your XM3 output file should be test on the XM3 assembler

A well-designed, implemented, and tested program written in either C or C++, meeting the above description, is required in this assignment.

3.3 Points to consider:

Here are some points to consider when implementing the preprocessor (you are not required to submit answers to these questions; however, your design should take them into account):

- How are comments containing an emulated instruction as a name handled?

- Should the preprocessor write to the XM2 input file or write to an entirely new XM3 file?
- How are incorrect instructions or arguments, or both, handled?

4 Examining an XM record

4.1 The XM record structure[1]

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 *free format*, meaning that it has no fixed fields. All records have the same format, defined as follows (**bold** indicates terminal symbols):[2]

Record = (*Label*) + ([*Instruction* | *Directive*]) + (*Operand*) + (; *Comment*)

Label = *Alphabetic* + 0 {*Alphanumeric*} 30

Instruction = * *An instruction mnemonic* *

Directive = * *An assembler directive* *

Operand = * *The operand(s) associated with the Instruction or Directive* *

Comment = * *Text associated with the record – ignored by the assembler* *

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

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

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

4.2 Scanning states

When a record is read, the record should be scanned for tokens. Each token (a string delimited by white space) should be examined and from that, the next action determined, for example:

- If the token is a semicolon (";"), a comment has been encountered, causing the preprocessor to ignore the remainder of the string.
- If the token starts with an *Alphabetic*, it might be an XM2 instruction. This will need to be compared with the list of instructions to translate. The possible outcomes are either it is an instruction to convert or it is not. The preprocessor would then change state to either look for possible arguments (if they are required) or resume looking for the next instruction.

5 Importance of design

The importance of putting your effort into designing the solution cannot be stressed enough. Understanding the problem is essential – if you do not understand the problem you will have very difficult time trying to solve it.

[1] This section is from the *XM3 Assembler User's Guide* (supplied on Brightspace).

[2] The example is shown in a Data Dictionary format that can be used to define data structures: '(' ')' – optional; '[' '|' ']' – choice; 'LB{' '}UB' – sequence from LB (default 1) to UB (default ∞); and '+' – AND or concatenate.

Working through the problem with pen/pencil-and-paper is a good first step. Ask yourself, what would you do if you were given a record that might contain an instruction to be converted? You would scan it from left-to-right, comparing each "token" (i.e., characters separated by white space) found in the record with the list of strings to be converted; if found, you would need to convert it, if not, you'd move on to the next record.

Once you are satisfied with your method (or algorithm), you can try implementing it. By getting a working design, you can be pretty sure that your implementation will work (other than overcoming typos).

You might also consider doing a stepwise design. That is, rather than trying to design everything at once, you might want to do a bit at a time, especially if you are unsure about, for example, file I/O or scanning a string for tokens. A possible approach could be:

1. Open a file and then read and display each record until end-of-file.
2. Using the code from 1, expand it to include opening a new file and writing each record from the input file to the new output file.
3. Using the code from 2, take each record read, extract each symbol (or token) from it, and display the token.

By taking the stepwise approach, if an error occurs, it is because of something new added to the program. If you try doing everything at once, you will spend (waste) hours or days fixing things that can cause other errors.

Finally, you do not need to know X-Makina's assembly language to solve this problem. Your solution is to find specific tokens in a record and change them according to a set of rules.

6 Marking

The assignment will be graded out of 20 using the following marking scheme:

Design Document
The design document is to include an introduction as to the purpose of your software (i.e., your understanding of the problem), a description of the algorithms (using tools such as state diagrams or structured-English), and a description of the major data structures required to solve the problem (in a data dictionary).

Total points: 6.

Software
A fully commented, indented, magic-numberless, tidy piece of software that meets the requirements described above and follows the design description. The software must work with the VM described in the labs.

Total points: 10.

System Testing
A set of system tests demonstrating that the software operates according to the design description. The submission must include the name of the test, its purpose or objective, the test configuration (i.e., the XM2 input file), and the test results (the XM3 output file). The

designer of the software is responsible for defining and supplying the tests; several sample test files will be supplied.[3]

Total points: 4.

Each part of the assignment must be submitted through Brightspace.

7 Important Dates

Available: 8 May 2020 (00h ADT)

Design document submission: 18 May 2020 (24h ADT)

Software (source only) and testing submission 28 May 2020 (24h ADT)

Late submissions for this assignment will be penalized 1 point-per-day.

8 Miscellaneous

This assignment is to be completed individually.

Do *not* discard this work when completed, as it can be used with the remaining assignments or potentially in future courses.

This assignment is worth 10% of your overall course grade.

If you are having *any* difficulty with this assignment or the course, *please* contact Dr. Hughes or one of the TAs, Gary Hubley or Zach Thorne, as soon as possible.

[3] For more information on testing in general, see Welcome to Software Testing Fundamentals and on system testing in particular, see System Testing.