EECS 370 LC2K Midterm Reference Sheet

R-type instructions (add, nor):

bits 24-22:	opcode
bits 21-19:	reg A
bits 18-16:	reg B
bits 15-3:	unused (should all be 0)
bits 2-0:	destReg

I-type instructions (lw, sw, beq):

bits 24-22:	opcode
bits 21-19:	reg A
bits 18-16:	reg B
bits 15-0:	offsetField (a 16-bit, 2's complement number with a range of -32768 to 32767)

J-type instructions (jalr):

bits 24-22:	opcode
bits 21-19:	reg A
bits 18-16:	reg B
bits 15-0:	unused (should all be 0)

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, 22)	Action
add (R-type format)	000	Add contents of regA with contents of regB, store results in destReg.
nor (R-type format)	001	Nor contents of regA with contents of regB, store results in destReg. This is a bitwise nor; each bit is treated independently.
lw (I-type format)	010	Load regB from memory. Memory address is formed by adding offsetField with the contents of regA.
sw (I-type format)	011	Store regB into memory. Memory address is formed by adding offsetField with the contents of regA.
beq (I-type format)	100	If the contents of regA and regB are the same, then branch to the address PC+1+offsetField, where PC is the address of this beq instruction.
jalr (J-type format)	101	First store PC+1 into regB, where PC is the address of this jalr instruction. Then branch to

```
                                                the address contained in regA.
                                                Note that this implies if regA
                                                and regB refer to the same register,
                                                the net effect will be jumping to PC+1.

halt (O-type format)    110                     Increment the PC (as with all
                                                instructions), then halt the
                                                machine (let the simulator
                                                notice that the machine
                                                halted).

noop (O-type format)    111                     Do nothing.
-------------------------------------------------------------------------------

Note: all registers are initialized to zero before a program starts running.
```

3.1.1 Assembly File Format (Project 2)

Assembly language programs will be of the same format as those from Project 1, with a few extra restrictions.

The first part of the assembly file must contain only assembly instructions. The second part should contain only `.fill` assembler directives. For example, suppose an assembly file is composed of M instructions and N `.fills`. Lines 0 to (M-1) contain actual instructions, and lines M to (M+N-1) contain `.fills`, with no mixing between them. We refer to all of our instructions as belonging to the `Text` section of our program. Moreover, everything that contains a .fill statement is considered to be in the `Data` section of our program. It is important that all of your test cases separate these two sections such that no `.fill` directives are in the `Text` section and no instructions are in the `Data` section. Below the data section is the `Stack`, which is initially empty; for an instruction to access the stack, e.g load a word from the stack, we will use the label `Stack` to denote the start of the stack section.

3.1.2 Local and Global Labels

LC2K files may now use global symbolic addresses, which means we must now distinguish between local and global labels. The scope of a local label is the file the label is defined in. (This is analagous to a variable or function with the `static` keyword in C. The scope of a local variable in C is at most a function.) The scope of a global label is all object files linked together (more on this in part 2l). Because of this, different object files can use local labels with the same name and still be linked together. Local labels will start with a lowercase letter `[a, b , ... , z]` while global labels start with a capital letter `[A, B, ..., Z]`. This is unique to LC2K as a way to distinguish between local and global labels. For example, `staddr` is a local label whereas `Staddr` is a global label.
Local symbolic addresses must be defined at assembly time. However, a global symbolic address can be undefined at assembly time. It is assumed that undefined global labels are defined in another file to be resolved at link time, so they should be temporarily resolved as address 0 in the text and data segments. Defined symbolic addresses should be resolved exactly as they were in Project 1. That is, it is entirely possible that a global label is defined and referenced in the same file; if this is the case, the label should be resolved just like a local label. The `Stack` label should be treated as an undefined global label for the purposes of the assembler.

Just like P1A, you can assume assembly files max out at 65536 total instructions and data, although we'll test you on much, much less than that. As suggested in the starter code, you may assume that no input LC2K file is more than 1000 lines.

3.1.3 LC2K Peculiarities Part 1

Firstly, if a `beq` instruction contains a symbolic address, the label it refers to must be a locally defined label. This label can be either a local or global label. A `beq` should not branch to another file, and a programmer should use `jalr` in this case.
Secondly, in LC2K, loading or storing to an absolute address no longer makes much sense. The locations of data and text within the final executable file will likely be different than in the original object file, leading to unintended execution. While this isn't something we will enforce with error checking, it is recommended that labels are used when dealing with loads and stores. In reality, there are reasons to use absolute addressing: memory mapped IO for example or cache analysis (see Project 4). If you come across a label with a constant offset, assemble as in Project 1.

Thirdly, local labels should not be included in the symbol table. However, a local symbolic address does need a relocation table entry as the address of the local label might change. These addresses can be fixed by calculating the new local label location during linking.

3.1.4 Summary

In summary, assembly file formatting rules are:

1. Do not mix instructions with directives (`.fills`)
2. Instructions come first
3. Directives (`.fills`) come second
4. Defined symbolic addresses (defined local and global labels) are resolved exactly as they were in the Project 1 assembler
5. Undefined global symbolic addresses are temporarily resolved as address 0
6. Local labels start with `a…z` and must be defined at assembly
7. Global labels start with `A…Z` and can be undefined at assembly
8. Branches cannot use undefined global symbolic addresses

3.2 Object File Format

Object files will contain the following sections in the following order:

- `Header`
- `Text`
- `Data`
- `Symbol` table
- `Relocation` table

*** Refer to lecture, lab, and walkthrough for a detailed explanation of each section. ***

Table 1: Object file sections

Section Name	**Number of lines**	**Description**
`Header`	Fixed: 1	The `Header` contains the size, in lines, of the sections to follow. Sizes are listed in the following order, each separated by a space: `Text`, `Data`, `Symbol` table, `Relocation` table.
`Text`	Variable: `t` `t` = # of instr.	Each line in the `Text` segment consists of a single machine code instruction, assembled in the same way as instructions in Project 1.
`Data`	Variable: `d` `d` = # of .fills	The `Data` segment contains data stored by assembler directives, one word of data per line.
`Symbol` table	Variable: `s` `s` = # of locally-defined global labels + # of Unresolved global symbolic addresses	Each line in the `Symbol` table consists of a global label, one letter (T/D/U) corresponding to `Text`, `Data`, and `Undefined` respectively, and a line offset from the start of the T/D section (0 if the letter was 'U'). Each value separated by a space, in that order. Each symbol should only appear once in the symbol table, even if it is used multiple Times. Entries can appear in any order.
`Relocation`	Variable: `r` `r` = # of uses of symbolic addresses excluding `beq`	Each line in the `Relocation` table consists of a line offset from the start of the T/D section (whichever section the symbol was used in), an opcode, and a label. Each separated by a space, in that order. Entries can appear in any order.

IMPORTANT FORMATTING NOTES:

1. Assembly code in text should be assembled EXACTLY as it was for project 1. This means symbolic addresses are resolved the same, with the exception of undefined global symbolic addresses which are temporarily assembled as 0.

2. Offsets in the Symbol and Relocation Tables indicate the line offset of the label from the start of either the Text or the Data section (whichever section the label is defined in for the symbol table, and whichever section the instruction or .fill appears in for the relocation table).

For example, the symbol table entry `Foo D 0` indicates the label `Foo` is defined on the zeroth line in the Data section. The relocation table entry `4 lw Foo` indicates the symbolic address `Foo` is used on the fourth line (zero indexed) of the Text section by a `lw` instruction.