CS 3432 – Computer Organization Authors: Dr. Shirley Moore, Robert Alvarez

Lab 3 - Expression Compiler Backend for RISC-V

**Instructions:**

The assignment would be an INDIVIDUAL assignment. It is permitted to share ideas with other classmates BUT you must write your code. Meaning, code implementation must be unique.

**Introduction:**

A compiler typically has frontend and backend components. The front end parses a source code program and produces an Abstract Syntax Tree (AST) representation of the program. The backend traverses the AST and outputs assembly or machine code instructions for a specific ISA. Other actions may also be carried out, such as transformations of the AST to implement optimizations such as strength reduction and common subexpression elimination.

For this assignment, you are given a build\_tree() function that takes a C R-value expression as an argument and returns a pointer to an AST. That is, you are given a front end for a mini compiler that handles a subset of C expressions. An R-value is an expression that can appear on the right-hand side of an assignment statement. Your job is to write a backend that traverses the AST and outputs RISC-V assembly code.

**Task:**

1. [12 pts]. You are given files main.c, build\_tree.c, build\_tree.h, backend.c. The file, main.c contain the main() function. Construct a Makefile that:
2. Creates an executable name “expr\_comp” when “make” or “make all” is run.

* Note that if I want to run such an executable in the terminal I should be able to run it by typing “./expr\_comp”

1. Run the executable when the Makefile target “run” is called.

Notes:

* When your executable is running, you can enter an expression and press return to see how expressions are parse into a AST and what assembly instructions are generated.
* To stop entering expressions and quit the program, type Ctrl-D for end-of-file.
* You can also redirect input from a file that contains expression strings one per line, e.g., ./expr\_comp < [filename to overwrite output with]

1. [17 pts] Write a function that takes an unsigned integer as an argument and determines if that integer is a power of 2. If the integer is a power of 2, the function should return the logarithm to the base 2 of the integer, otherwise, the function should return 0. Write a driver program to test your function and test your function thoroughly to make sure it is correct.
2. Create a new file to test such a function.

* The function can be implemented in one line using the trinary operator and the bitwise AND operator.
* No need to implement log2() function, you can import it from math.h standard library.

1. Modify the Makefile so when “make testPowTwo” the tester file is compiled and executed.
2. The tester should test your function with at least 5 powers of two numbers and 5 that are not.

* Instead of hardcoding values, you can use the rand() function to generate random integers.

1. [10 pts] Compile and run the code in build\_tree.c. The postorder() function traverses the AST and outputs a postfix expression. Currently, the code allows only single-letter variables using the first 10 lower-case letters. After examination, explain the algorithm on how the functions are implemented.

* Explanation consists of at least tracing the state of the opstack and the nstack as it traverses each character of the expression: “a+23”.

1. Write a backend that traverses the AST and outputs a RISC-V translation of the expression. You may use register x0 as needed and you may use registers x5 through x31 to hold variable values and intermediate results.

* [24 pts] Add code to the generate\_code() function to handle expressions that contain constants. For binary operations, the code you are given handles the case where both the left and right children of the operator node are REG-type nodes. You will need to handle the following additional cases:
* The left child is REG type, and the right child is CONST type.
* The left child is CONST type, and the right child is REG type.
* Note that if an operator commutes and there is an I-type instruction that can implement it, the I-type instruction must be done. For example:
* A naïve implementation for 3|a needs an addi instruction for 3 into a register and then an or instruction.
* A compiler implementation for a|3 only needs an ori instruction.
* The left child is CONST type, and the right child is CONST type.
* A compiler does constant propagation/folding.
* Handle unary operators where the left child is a REG or a CONST type.
* Note that you will need to convert the unary negation and NOT operators to binary operators since RISC-V does not have any instructions for unary operations.

1. [15 pts] Implement strength reduction by transforming multiplication and division by a power of two to use a shift instruction instead.

* Note: Your function from part 2 could be useful for this part.

1. Load upper immediate number. Before starting, read over the “Load Upper Immediate” section in [riscv-arithmetic](https://projectf.io/posts/riscv-arithmetic/) and to [stackoverflow\_how-to-load-an-immediate-number-to-a-register-in-rv32i-base-instruction-set](https://stackoverflow.com/questions/56781564/how-to-load-an-immediate-number-to-a-register-in-rv32i-base-instruction-set).

* [5 points] Fill in the following table to load numbers that require more than 12 bits into a register.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Positive case | | | | |
| Hexadecimal | Decimal | LI | LI -> LUI | ADDI |
| 0x1000 | 4096 | 4096 | LUI 1 | N/A |
|  | 6143 |  |  |  |
| 0x1800 |  |  |  |  |
| 0x1801 |  |  |  |  |
|  | 8192 |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Negative case | | | | |
| Hexadecimal | Decimal | LI | LI -> LUI | ADDI |
|  | -4096 |  |  |  |
| 0xE801 |  |  |  |  |
|  |  | -4096 | LUI -1 | -2048 |
| 0xE7FF |  |  |  |  |
|  |  |  | LUI -2 | N/A |

* [10 points] Write code to handle immediate values that are larger than 12 bits.

NOTE: The use of LI is prohibited! Only use LUI and ADDI.

Hint: Positive and negative values can be handled with the same sequence of instructions. Thus, no distinction must be made.

1. [15 pts] Write a new input file (.txt file) that tests all the possible cases implemented in parts 4, 5, and 6. In a Word document, for each new input expression, justify why each line is testing a case requested above. Justification consists of:
2. Parsing the expression and creating the AST.
3. Traverse the AST, and mention how and when the registers are assigned/released, if any.
4. Mention at what point of the AST traversal an assembly instruction is generated and what branch in the backend.c is being executed.
5. [15 pts] Modify build\_tree.c so it can read variables of up to 5 characters long where each character may be a lower-case, upper-case, or underscore. After examination and code modification, explain the algorithm on how the functions are implemented.

**Grading:**

* The total maximum number of points is 123. The lab would be graded out of 100.

**Deadline (Blackboard):** April 9th, 2024, by 11:59 pm.

* Source code (Only .c and .h files, and 1 Makefile)