**E 2.15 Language Processors Compilers Coursework**

**Dylan Auty, EIE2**

**Username: da2412**

**Email: da2412@ic.ac.uk**

**CID: 00733844**

**24/3/2014**

The purpose of this coursework is to design a compiler that will convert standard ANSI C code into ARM Assembly, to be run on a Raspberry Pi. The Raspberry Pi can run ARM v6 Assembly.

There are several parts to the delivered compiler source code:

* Compiler.cpp
* Compiler.l
* Compiler.y
* PreProcessor.cpp
* stackType.h
* Parser.h

The main grammar was written using bisonc++ and flex++. It is designed to handle standard C expressions that have been tested in samples 1 to 6. Compilation and running instructions can be found in the README.

The preprocessor’s job is to remove comments and includes from the code. Where it encounters these, it replaces them with a blank space. This preserves line number such that the line number that is output by the compiler when throwing an error message corresponds to the line number of the error in the input file. The output of this is then passed to the compiler itself.

Register R0 is used for function returns and the calling of the printf function. Registers R1-R4 are used for passing values between functions as arguments. Registers R12-R13 are in use by the processor – the link register, the program counter, the stack pointer etc. This leaves registers R5-R12 free for potential use. When searching for (and not finding) the register containing a given variable, an error will be thrown and the compiler will exit, displaying a line number on which the fault may be found.

A stack type was implemented in “stackType.h”, which contains all of the possible types that the Yacc file may use to pass values up the tree.

Sample 1 contains basic control flow, declaration of a main function, variable declaration and assignment, and the printf function call. The grammar assumes that the only things separating statements are semicolons – whitespace and newlines are meaningless. A grammar rule was created that could handle expressions of the form “type name <something>,” where <something> could be a function definition, a global variable declaration or a global variable assignment. This was used inside functions also to allow statements and variable declarations to be made. The exception is that if the user attempts to define a function inside another, an error will be thrown and the program will halt execution.

Statements can take the form “type name assignOp expression,” or “name assignOp expression”. Therefore, only an assignment was implemented at this level, such that the register or value returned from the “expression” term would be moved into the register associated with variable “name” – i.e. “MOV Rn, Rd” or “MOV Rn, #k”.

To handle registers, every time a variable was declared, a free register would be found. The register would then be bound to a variable name and have itself labelled as being in use. This protects it from being overwritten, and allows it to be referenced during compilation. At the end of a function definition, all registers are freed up.

The “expression” rule describes a branch of the tree that inherently implements the correct order of operations – bracketed, then multiplicative, then additive. Each atomic statement will return a result register or a value to the rule above it, and this means that at the end of this when returning to the rule that called “expression”, the return holder “$$.RETURN” will contain a string with the result register in it, allowing its immediate use by other functions.

If loops are implemented using the “branch <condition>” instructions. The top and bottom of the loop are labelled in the ARM assembly code, then a suitable “CMP” operation is done based on what is contained in the arguments (a Boolean expression, e.g. a ==b or c < d). Immediately after, a branch conditional instruction is implemented such that if the condition is NOT met, then the program will jump over the if block and carry on. To allow multiple if blocks, each labels are appended with a number that is set using a global variable, that is initialized to 0 and incremented on the creation of each if statement.

While loops follow a similar kind of labelling system as if loops, and again use a Boolean expression rule to produce a suitable CMP statement. Then a branch conditional statement is produced, which will go to the start of the while loop again if the condition is still met.

For loops operate in a similar way to while loops. The only differences are that they change a variable (the ARM Assembly code counterpart of “i++”) before executing the compare and branch.

Function calls are implemented by defining the function as it is come across, but setting the initialisation flag “.global main” at the start of the program. With this, the assembler will execute the “main” function first, and maintain the desired sequence. Each function has its own label (including the main function). When a function is called, arguments may be passed to it using the first 4 registers, therefore a maximum of 4 arguments may be used. The remaining registers are pushed onto the stack to preserve them, then a branch and link is put in place (“BL label”). The first thing the function does is push the link register onto the stack to know where to return to, and the last thing it does is pop this value off the stack into the program counter to effect a “go back to where you were called from”. Then the registers 5-12 are popped off the stack again so that the function which called the other function can continue to function with no interruption.