Grundlagen Compilersysteme

Max Strasser 1220029
Paul Huber 1320318
Team: knights

27.6.2016

Assignment 0: Your Team

"This is knights Selfie"

Assignment 1: Bitwise Shift Instructions

We used the code for ADD in MIPS as a template for implementing the shift instructions and added them to the op-code library.

Assignment 2: Bitwise Shift Operators (Scanning, Parsing)

First we implemented parsing of << and >> by adding these to the list of recognizable symbols and giving them unique names (sym_lshift, sym_rshift). Then we used gr_simpleExpression as a template and created our own procedure called gr_shiftexpression, which implemented shifts to be used in expressions(ie. a << b).

Assignment 3: Bitwise Shift Operators (Code Generation, Self-Compilation)

For this assignment we replaced the code in leftShift and rightShift by using the << and >> operators we implemented in assingment 2.

Assignment 4: Constant Folding

To implement constant folding we created a simple list in gr_expression with two integer values, one a boolean indicating whether or not the last read value is a constant and the actual value of said constant. A pointer to that list was given as a parameter to all other procedures implemting our grammar, all the way down to gr_factor. If the compiler found a constant it would be stored into the list in gr_factor instead of loaded into a register right away. Then we had to implement actual constant folding in all other grammar procedures up to gr_shiftExpression We can fold any number of consecutive constants, but no expressions containing variables(ie. $x = 1 + y + 2$ cannot be folded).

Assignment 5: Arrays

To declare arrays we first had to add the [and] symbols to our parser and extend the symbol table to also store the size of an array. We also introduced a new type for arrays. Then we added a new grammar procedure called gr_index to retrieve values like [28]. When declaring an array we simply allocate memory times the size of the array and store the size in the symbol table entry. For access we first need to get the address of the first element of the array and then subtract the offset depending on what index we want to retrieve and load the value at that address. When this was working changing the symobl list to an array was trivial.

Assignment 6: Two-Dimensional Arrays

We first had to decide on a memory layout for two-dimensional arrays(and stick with it), then make some simple changes to the parser and extend what we implemented in assignment 5.

Assignment 7: Struct Declarations

To begin with we introduced a new type for structs as well as the keyword *struct* and implemented the struct table, which serves the same purpose as the symbol table, only on the scope of each specific struct. When declaring a struct we are counting the memory we have to allocate.

Assignment 8: Struct Access

For struct access we implemented a new grammar procedure called gr_struct, which looks up the right offset of an identifier in the struct table.

Assignment 9: Boolean Operators (Individual)

After implementing parsing of the boolean operators we extended the list we use for constant folding by one value, which stores binaryLenght whenever the parser finds one of the boolean operators. Then we emit a dummy branch instruction which we fixup after we actually know where to branch to(ie. the end of an if block). This way MIPS is able to do lazy evaluation at runtime.

Assignment 10: Boolean Operators (Algebra)

For this assignment we tried to extend our code for assignment 9 but eventually ran into problems with using too many registers when combining different boolean operators in one statement. Unfortunately we were unable to debug this issue and had to revert again to preserve self-compilation.

Assignment 11: Memory Management

To free previously allocated memory of fixed size we created a global list called freeList which stores a pointer to the next free address to be used by malloc. Memory chunks being of same size was very important here since we did not have to worry about fragmentation. All our testcases in selfie's main returned expected results, however when we tried to free symbol table entries at the end of procedures we again encounterd an issue with too many registers being used eventually. Since we were not able to find the root of this bug here either we think it might be caused by some error in a previous assignment. Without freeing symbol table entries, however, we still maintain self-compilation.