Jon Bowen

CS 3210

PAL Project

Time Spent On Project

|  |  |
| --- | --- |
| Date | Hours |
| 1/27/2018 | 4.25 |
| 1/28/2018 | 7 |
| 2/3/2018 | 4.5 |
| 2/4/2018 | 3 |
| 2/5/2018 | 1.25 |
| 2/7/2018 | 1 |
| 2/10/2018 | 5.5 |
| 2/11/2018 | 3.25 |
| Total | 29.75 |

“How’d It Go?”

Overall the project went fairly well. The most challenging part was anticipating all the different errors that possibly could be made and catching them in succinct ways. Despite throwing everything I could think of at my translator, there are undoubtedly special cases that did not occur to me that are not accounted for.

It was also difficult to adjust to changes to the language specification that were added after I had completed a substantial amount of work. The addition of the SRT opcode was relatively harmless, but the new DEF opcode required a few hours of refactoring to satisfactorily account for that in the translator.

I notice that one initial error can cascade and generate many additional errors further into the program. For example, an illegal memory identifier is not added to the symbol table which will cause another error later if that memory identifier is used in a statement. This is a common experience for any coder looking through a list of compiler errors. Often there is one error that, if fixed, eliminates many others.

Schedule

I did roughly stick to my schedule. I wrote sample programs in class, and slightly improved them when I got home. I completed the reading and wrote down the BNF that following weekend. The initial design was completed by 2/1 with some slight modification and addition of specifics from then until final completion. The developed of the actual code proceeded according to the schedule as well, first building the structure and then filling in the details by carving out the smallest, well-defined piece I could think of. I would not change anything the next time I make a project schedule. It is a normal part of my full time job, so I’m fairly comfortable with my process currently.

Ins and Outs of Translator Pass

Focusing on the translation work and ignoring tasks such as file I/O, the translator first removes all blank lines and any comments found in the file. Then it checks to see if there are any lines left. If there are, it checks if the first line is the SRT opcode. If not, it adds an error, if so it consumes the line. Next it looks for any DEF opcodes and stores any memory labels in a symbol table. After DEF is not found as the opcode, it enters a loop to process the remaining lines. It checks and stores any labels found, and then removes them from the line. Then it will do checks for valid opcode, number of operands, type of operands, etc. The label in any branch instructions is also stored. If END is encountered, the loop is broken. Once the loop exits, labels declared and labels branched to are compared to find any mismatch, and an error is printed if END was not encountered.

Discussion of Errors Not Checked

1. The SRT opcode can appear more than once is a program and not throw an error. So long as the program begins with SRT, any subsequent uses are simply ignored. It would not be hard to add a check to list an error if SRT appears more than once. Simply set a flag once it’s encountered, and check the flag if it is encountered again. (I have subsequently added this feature to the translator.)
2. The translator does not check that all memory aliases that are created are also used. This would be straightforward to add. This kind of checking is done on labels by adding created labels to a defined list and adding used labels to a used list as they are used. This same strategy could be used to build a list of the identifiers created and record identifiers as they are used. Comparing the sets at the end of translation shows whether or not an identifier was created but not used.
3. The translator does not check if a label is defined more than once. Allowing a label to be defined more than once leads to ambiguity about execution path depending on if it is allowed or how the compiler handles it. Adding this check to the program would be easy. The set of defined labels is built already for another use. Convert that set data type into a map data type and store the number of times defined in the value for each key. If translation completes and any value is more than 1, the label has been defined more than once. (I have subsequently added this feature to the translator.)

Discussion of High-Level Errors Not Occurring in PAL

1. Type checking is not required in PAL since unsigned integers are the only allowed data type. This is often an aspect of high level languages. Type checking could be approached via the symbol table by adding an entry for the type of each symbol when it is declared. When operations that depend on the type are performed, this table could be referenced to determine if there is a type related issue.
2. The meaning of assignment is completely unambiguous in PAL since all types are immutable unsigned integers. In higher level languages, however, assignment can be either reference assignment or value assignment. A convention is required to alleviate this ambiguity in HLL’s. Typically reference types use reference assignment by default.
3. Memory leakage is not an issue for PAL programs because they do not explicitly hold any memory. Memory addresses and register and be directly overwritten at any moment. HLL’s do suffer from this problem as a program may never release resources it claims. This problem can be addressed in the translator by posting a warning for any memory reserved at some point, but not freed later before the end of the program.