Princeton University
COS 217: Introduction to Programming Systems
Assembler Assignment: Development Stages

Stage 0: The Development Environment

Create a development environment.

Study the pertinent lecture notes and the assignment statement.
Copy directory arizona:/u/cs217/Assignment5 to an appropriate place in your home directory.
    cd *appropriatedirectory*
    cp –r ~cs217/Assignment5 .
Study file input.h.
    Note the declaration of the variable "first" (which points to the beginning of the instruction list).
Study file output.h.
Study file main.c.
    Note the global variables symbol_table, data, bss, and text.
    Note the calls to functions pass1() and pass2() – which you must create.
Create file pass1.c containing a pass1() function.
    Temporarily define your pass1() function to traverse the list referenced by "first", and simply print a count of the number of instructions in the list.
Create file pass2.c containing a pass2() function.
    Temporarily define your pass2() function so it does nothing but "return 0;"
Note the Hanson Table ADT in the cii directory.
Study file Makefile, and edit it to add pass1.c and pass2.c to the project.
Use the Makefile to build the project.
Note the contents of the testing_programs directory:
    Test assembly language programs.
    Programs objcmp and objcmp_detail (as described below).
Run your assembler using the test assembly language programs as input.
    Verify that it prints the proper instruction count for each test program.

Stage 1: Pass 1

Enhance the pass1() function so it generates a symbol table.

Assume that all expressions within directives are simple values.
    That is, assume that expressions that are relevant to pass 1 contain no operators or labels.
    Exception: The expression within a .global directive is a label.
Edit main.c by adding a call to print_passes() after the call to pass1().
    print_passes() prints the symbol table to stdout.
Test your assembler using testing programs 1 – 3.
    Analyze the output produced by print_passes().
    Add more labels to testing programs 1 – 3, and again analyze the output produced by print_passes()

Stage 2: Pass 2 Data Section

Enhance the pass2() function so it generates a <u>complete</u> data section and a <u>minimal</u> text section.

Hardcode the pass2() function to generate a text section containing the machine code for the one instruction "add %r1, %r2, %r3", that is, 0x86004002.

Test your assembler using testing programs 1 – 3.
For each testing program:
Use your assembler to produce a .o file.
Use the "as" assembler to produce another .o file.
Use the objcmp –d command to compare the data sections of the two .o files; they should be identical. If not, use the objcmp_detail command to find the differences.
Use the objcmp –t command to compare the (minimal) text sections of the two .o files; they should be identical. If not, use the objcmp_detail command to find the differences.

Stage 3: Pass 2 Text Section

Enhance the pass2() function so it generates a text section, assuming that all expressions are either simple values or simple labels that can be resolved by the assembler.

Test your assembler using testing programs 1 – 7 and the objcmp and objcmp_detail programs.

Stage 4: Expressions with Operators

Enhance the pass1() and pass2() functions so they evaluate expressions that involve operators.

Test your assembler using testing programs 1 – 8 and the objcmp and objcmp_detail programs.

Stage 5: Expressions with Labels

Enhance the pass2() function so it evaluates expressions that involve labels, and thus generates relocation information within the text section.

Test your assembler using testing programs 1 – 11 and the objcmp and objcmp_detail programs.
Examine the relocation information using the elfdump UNIX utility.

Stage 6: Complete Executable Programs

Validate your assembler by using it to produce complete executable programs.

Test your assembler using testing programs app_01fibonacci.s and app_02bubblesort.s
For each testing program:
Use your assembler to produce a .o file.
Use the "as" assembler to produce another .o file.
Use the objcmp –d command to compare the data sections of the two .o files; they should be identical. If not, use the objcmp_detail command to find the differences.
Use the objcmp –t command to compare the text sections of the two .o files; they should be identical. If not, use the objcmp_detail command to find the differences.
Use the elfdump utility to compare the relocation information in the two .o files.
Use the gcc command to produce an executable file from .o file generated by your assembler.
Use the gcc command to produce another executable file from the .o file generated by "as".
Execute the two programs; their behavior should be identical.

Stage 7: Error Handling (for extra credit)

Enhance your assembler to detect and report the errors illustrated by the five "err_" assembly language programs.

Your assembler should produce the same error messages as the "as" assembler does.

Copyright © 2001 by Robert M. Dondero, Jr.