

1 Objectives

- Generating x86 target code.
- (finally) completing your compiler!

2 Problem Statement

You are to write a program that:

- reads a filename from the command line
- using the grammar from project 2, creates a filename (ending in `.s`) that is the equivalent of the source code found in the input file. filename specified contains text that matches the grammar.
- the generated file should then be able to be assembled using `gcc` to give an executable. (You don't have to automate this process, but it is not hard to do, and will give a "truly complete" compiler.)

3 Input/Output

you are being given assembly code for two input/output functions, as follows:

- `void printi(integer value)`, which simply takes an integer as a parameter and prints it to the screen on its own line.
- `integer readi(void ignored)`, which will read in a single integer from the keyboard and return it.

The code for these functions is as follows - note the need for the data space (`.rodata` and `.data` sections).

```
        .section    .rodata
_PRLBL:
        .string    "%d\n"
_SCLBL:
        .string    "%d"

        .section    .data
        .align     4
_scantmp:
        .skip      4
```

/ code continued on next page*/*

```

        .section      .text
printi:
        pushl        %ebp
        movl         %esp,%ebp

        movl         8(%ebp),%eax
        pushl        %eax
        pushl        $_PRLBL
        call         printf
        addl         $8, %esp

        leave
        ret

readi:
        pushl        %ebp
        movl         %esp,%ebp

        movl         $_scantmp,%eax
        pushl        %eax
        pushl        $_SCLBL
        call         scanf
        addl         $8, %esp
        movl         (_scantmp), %eax

        leave
        ret

```

This code should basically be part of any program you generate - that is, your compiler should include this as part of any resulting assembly code file.

Note that the `readi` function follows the gcc standard of returning its value through the `%eax` register. Also note that `readi` ignores any parameter that was pushed onto the stack - it is up to you as to whether or not you actually want to bother pushing on void parameters; just be sure that you follow the same standard in making function calls as you do in handling them.

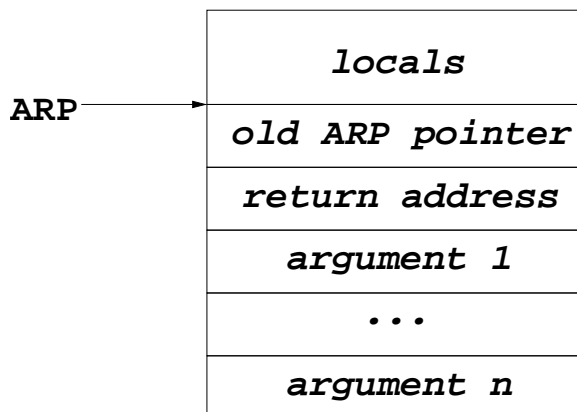
4 Memory to Memory Model & Virtual Registers

The easiest way to generate target code is to use a memory to memory model. Recall that this entails storing all results to appropriate memory locations, as opposed to remembering what values were stored in which registers at which points in time. This latter problem would entail a register allocation (and assignment) problem, which you are *not* being asked to solve.

Using this model may at first make it seem difficult to handle your virtual registers. However, there is a simple method of dealing with these that utilizes the memory to memory model: simply add all of your virtual registers to your symbol table (ideally as they are being created). Then, each operation that utilizes a virtual register will actually end up performing a memory “load” and/or “store” (although x86 syntax doesn’t distinguish between `mov`’s and explicit memory loads/stores) to access that virtual register.

5 Activation Records

Should you choose to use the gcc standard of returning function values through the `%eax` register (and I recommend that you do so, although you are not required to do so), your activation record should look something like:



In this diagram, the stack would grow upwards towards the top of the page, and addresses increase towards the bottom of the page (and thus decrease towards the top of the page). As a reminder, the parameters pictured are *incoming* parameters for the current function. So long as you are 100% consistent, it does not matter what order you place the parameters in, provided that you utilize the same order in passing them from a calling function as you do in “extracting” them when in the called function.

6 Extra Credit

Don't forget – get this to me 48 hours or more early and you get a free 5 points. So, submit this project by 4/26 at 11:59 PM and you'll get a free 5 points!

7 Grading Breakdown

Correct Submission	10%
Successful Compilation & Makefile	35%
Correct Execution	40%
Comments & read.me File	15%
Extra Credit	5%

8 What to Submit

You will be submitting a tar file containing all source code, a makefile, and a `read.me` file for the project. If necessary, we will discuss the creation of tar files as well as how to build (and use) an appropriate `Makefile` in class. The `read.me` file should be a plain ASCII text file that overviews your project, including:

- your name.
- how to compile your project (if you have submitted a Makefile as required, this should be something like “type `make` to build the project”).
- how to run your project

- any known bugs your project has, and possible fixes to those bugs (partial credit abounds here).
- an overview of how you solved the project.
- what you named your output files(the file that contains intermediate code, broken down by basic block).