Course Project **4190.409 Compilers**

Phase 5: Code Generation

In this fifth and last phase of the project, your task is to convert the IR code into x86 assembly code. After finalizing this phase, your SnuPL/0 compiler is complete: it takes programs written in SnuPL/0 and outputs assembly code that can be compiled by an assembler into an executable file.

We implement a simple template code-based code generator. In the absence of a register allocator, we further assume a memory-to-memory model, i.e., values, including temporaries, are loaded from memory into registers before the operation and written back to memory after the operation has been executed.

The following pseudo-code describes the tasks of a minimal code generator:

```
Input: program in IR
Output: assembly
Pseudo-code:

  Emit(program):
    EmitCode(program)
    EmitData(program)


  EmitCode(program):
    forall s ∈ subscopes do
      EmitScope(s)

    EmitScope(program)


  EmitData(program):
    EmitGlobalData(program)


  EmitGlobals(program):
    forall d ∈ globals do
      EmitGlobal(d)


  EmitScope(scope):
    ComputeStackOffsets(scope)

    emit function prologue

    forall i ∈ instructions do
      EmitInstruction(i)

    emit function epilogue
```

```
EmitInstruction(i):
  load operands into register
  perform operation
  write result back to memory


ComputeStackOffsets(scope):
  forall v ∈ local variables and parameters in scope do
    compute stack offset and store in symbol tables

  return total size of stack frame
```

The format and necessary instructions of the IA-32 ISA are provided in Appendix 1, Appendix 2 contains information about the x86 AT&T assembly file format, including a skeleton file which will help you to get started. Appendix 3, finally, lists some useful GDB commands.

All the modification that you will need to make are in the file src/backend.cpp. To make things a bit easier for you, we have deleted the interesting parts from our backend and provide it to you. Positions where we have delete code are marked with // TODO.

Logistics:
Download the tarball from eTL and copy the files from the tarball into the corresponding directory of your compiler. Then run

```
$ make
```

to generate the SnuPL/0 compiler snuplc. Run

```
$ snuplc –help
```

to see a list of available command line options (feel free to modfy them).

Submission:

- the deadline for the third phase is **Decmeber 12, 2014 before midnight**. If you still have grace days, you may submit up to two days later. However, **December 14, 2014 is a hard deadline.** We want to grade your work before the make-up class on December 16.
- submit a tarball of your SnuPL/0 compiler by email to the TA (compiler-ta@csap.snu.ac.kr). The arrival time of your email counts as the submission time.

As usual: start early, ask often! We are here to help.

Happy coding!

Appendix 1: Intel IA-32

This appendix gives a minimal introduction to the Intel IA-32 instruction set and calling convention. Intel provides detailed manuals at the following address:
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

In particular, Volume 2 (http://download.intel.com/products/processor/manual/325383.pdf) contains the full instruction set reference.

1. Registers: IA-32 has 8 general-purpose registers, six of which can be used more or less arbitrarily.

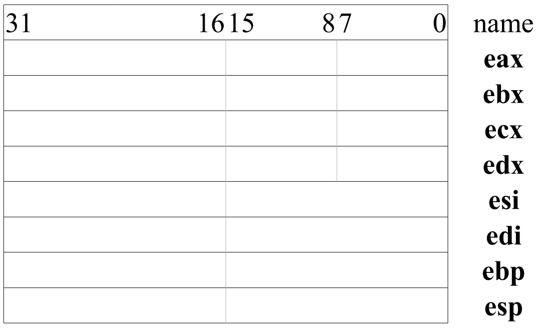


Illustration 1: General-purpose registers

esp/ebp represent the stack pointer and the base pointer; both registers are used to implement the calling convention. The other registers can be used freely with few exceptions (for example, multiplication and division use predefined registers). IA-32 is fully backwards compatible; this is why the registers can be accessed in their old 16- or 8-bit form. This is no concern for us, we will only use the full 32-bit registers.

Not visible here are two important registers: the program counter and the condition codes. The program counter is manipulated indirectly through control-flow instructions, and the condition codes are set/read implicitly by ALU operations and conditional branches.

2. Instructions: the following instructions suffice to implement a simple code generator for SnuPL/0. We use GCC to assemble our programs, hence the assembler syntax below uses the AT&T syntax. In AT&T syntax, the source is listed *before* the destination, immediate values are prefixed with a "$", and registers are prefixed with a "%" character. Memory addresses have the form

```
 displacement(%base, %index, scaling factor)
```

The generated memory address is mem[%base + %index * scaling factor + displacement]. We will only use two sub-forms: `disp` (for globals) and `disp(%base)` (for locals/parameters/temps).

Instruction	Effect	Description
`addl  S, D`	D ← D + S	addition
`subl  S, D`	D ← D – S	subtraction
`andl  S, D`	D ← D && S	logical and
`orl   S, D`	D ← D || S	logical or
`negl  D`	D ← -D	negate
`notl  D`	D ← ~D	logical not
`imull S`	[EDX:EAX] ← [EAX] * S	32-bit signed multiply
`idivl S`	[EAX] ← [EDX:EAX] / S	32-bit signed division

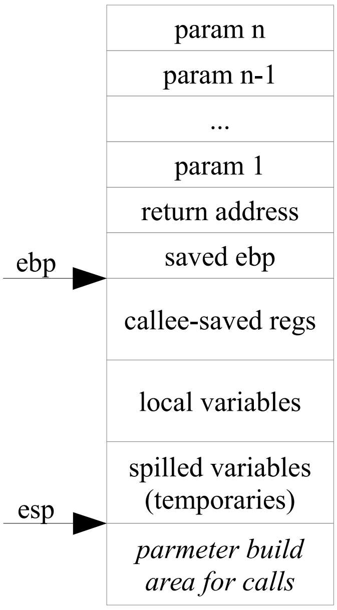
Instruction	Effect	Description
`cmpl  S2, S1`	[condition codes] ← S1 – S2	set condition codes based on the comparison S1, S2
`movl  S, D`	D ← S	move
`cdq`	[EDX:EAX] ← sign_extend([EAX])	sign-extend to 64-bit
`pushl S`	[ESP] ← [ESP] – 4 mem[ESP] ← S	push S onto stack
`popl  D`	D ← mem[ESP] [ESP] ← [ESP] + 4	pop top of stack into D
`call  T`	push return address continue execution at T	subroutine call
`ret`	pop return address from stack continue execution at return address	subroutine return
`jmp   T`	continue execution at T	unconditional branch
`je    T`	goto T if condition codes signal "equal"	conditional branch
`jne   T`	goto T if condition codes signal "not equal"	
`jl    T`	goto T if condition codes signal "less than"	
`jle   T`	goto T if condition codes signal "less equal"	
`jg    T`	goto T if condition codes signal "bigger than"	
`jge   T`	goto T if condition codes signal "bigger equal"	
`nop`		no operation

For almost all arithmetic instructions, one of the operands (but not both) can be a memory address (notable exception is the idivl instruction), a register or an immediate. The other operand is an immediate or a register.

3. Data: parameters, local variables and temporaries are stored on the stack and addressed relative to the stack and/or base pointer. Global data, however, must be allocated. The assembler allows to give names (labels) to junks of data, and you can then use those names directly as operands of instructions. Use the `.skip n` directive to allocate *n* bytes of memory.

4. Calling convention: in the Intel IA-32 calling convention, procedure activation frames are built by the stack and base pointer. The stack grows towards smaller addresses. The stack pointer points to the top of the stack. Storing data below the stack pointer may lead to unexpected results. Function arguments to subroutine calls are passed on the stack in reverse order, i.e., the first argument is on top of the stack, the second one immediately below the first one, and so on. Function return values, if any, are returned in register `eax`. The registers `ebx`, `esi`, and `edi` are callee-saved and thus must be preserved. Implicitly, `esp` and `ebp` are also callee-saved (both `esp` and `ebp` must point to the caller's activation record after returning from a subroutine call).

<u>5. Procedure/function activation frame</u>: below we give one possibility of a procedure activation frame. Of course, you are free to choose your own layout.



In the design above, procedure/function parameters are pushed onto the stack immediately before the call (and must be removed after returning). This is more convenient than a pre-computed fixed parameter area when supporting nested function calls.

The parameters and the return address are generated by the caller. The parameters by a series of push instruction, the return address implicitly by the call instruction. Upon entering the callee has to create the remaining parts of the activation frame as follows:

1. save `ebp` by pushing it onto the stack
2. set `ebp` to `esp`
1. save callee-saved registers
2. generated space on the stack for locals and spilled variables by adjusting the stack pointer

Immediately before returning to the caller, the callee needs to restore the callee-saved registers and dismantle the activation frame. This can be achieved by the following steps

3. remove space on stack for locals and spilled variables by setting the stack pointer immediately below the callee-saved registers.
4. restore callee-saved registers
5. restore `ebp`
6. issue the `ret` instruction

Appendix 2: AT&T Assembly, Assembling and Linking with the I/O Routines

Assembly programs are structured into sections. For our purposes we will only require two sections, the `.text` section which will contain the code and the `.data` section where the global variables are allocated.

Here is a skeleton file for programs in AT&T syntax:

```
# template
        .text                       # beginning of the text section
        .align 4                    # align text section at a 4-byte boundary

        .global main                # to let the assembler know that we implement
                                    # the function "main" (= module body)

        .extern Input               # the two I/O routines will be linked later
        .extern Output


main:                               # module body, followed by functions/procedures
        ...


        .data                       # beginning of the data section
        .align 4                    # align at a 4-byte boundary

huga:   .skip 4                     # define global variable 'huga' (4 bytes)
bimbo:.skip 1                       # define global variable 'bimbo' (1 byte)


        .end                        # end of program
```

Be aware that labels must be local or unique. An easy way of generating unique labels is to prefix them with the name of the scope (i.e., the procedure name) they are defined in.

To compile your program run

```
  $ gcc -m32 -o test08.o -c test08.mod.s
```

The -m32 options tells GCC that we want to build a 32-bit binary, and -c instructs the assembler just to assemble the input file into an object file. You can then disassemble the object file using

```
  $ objdump -d test08.o
```

To link your object file with the provided I/O functions, first compile the I/O module and then generate the executable:

```
  $ gcc -m32 -o IO.o -c ../rte/IO.s
  $ gcc -m32 -o test08 test08.o IO.o
```

Of course, this can all be done in one step:

```
  $ gcc -m32 -o test08 test08.mod.s ../rte/IO.s
```

Now you can run your program with

```
  $ ./test08
```

Appendix 3: GDB

Since SnuPL/0 does not support strings, debugging can be challenging. You may need to debug your x86 program using a debugger and follow the execution instruction-by-instruction to see what's going on.

The GNU debugger, gdb, is an excellent tool to debug programs. While it seems to be rather crude, it offers lots and lots of functions that most people are not aware of.

To start debugging your program using gdb, type

$ gdb ./test08

at the prompt. GDB greets you with

```
GNU gdb (Gentoo 7.5.1 p2) 7.5.1
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.gentoo.org/>...
Reading symbols from test08...(no debugging symbols found)...done.
(gdb)
```

From there, commands will help you run/stop/break your program and inspect/modify values. The following table contains a list of handy commands that you might use when debugging your program. For a complete list, type help and follow the instructions on the screen.

Command	Description
`r(un)`	run the program until - it ends - it crashes - it hits a breakpoint
`c(ontinue)`	continue a stopped program
`quit`	exit GDB
`break *address`	set a breakpoint at `address`
`break name`	set a breakpoint at `name`
`stepi`	execute one assembly instruction
`stepi n`	execute n assembly instructions
`disas`	disassemble around current program counter
`disas name`	disassemble at `name`
`display /5i $pc`	disassemble the next 5 instructions at `pc` at every stop
`display $<reg>`	display the value of `reg` at every stop
`<Enter>`	re-run the last command

Especially the display command together with stepi will be very helpful when stepping through your program.