Phase 5: Code Generation

The fifth and last phase of our term project converts the IR code into x86 assembly code. After finalizing this phase, your SnuPL/1 compiler is complete: it takes programs written in SnuPL/1 and outputs assembly code that can be compiled by an assembler into an executable file.

As a reference, we implement and provide a simple template code-based code generator. In the absence of a register allocator, the reference compiler further assumes 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 this 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)


  EmitGlobalData(program):
    forall d ∈ globals do
      EmitGlobal(d)
```

```
EmitScope(scope):
  ComputeStackOffsets(scope)

  emit function prologue
  InitializeLocalData(scope)

  forall i ∈ instructions do
    EmitInstruction(i)

  emit function epilogue

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



InitializeLocalScope(scope):
  forall arrays a ∈ local variables in scope do
    initialize meta-data for a on stack



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 AT&T x86 assembly file format, including a skeleton file which will help you get started. Appendix 3, finally, lists some useful GDB commands for the GNU debugger

All necessary modifications in this phase affect the file src/backend.cpp. To make things a bit easier for you, the overall structure of the backend and some helper functions are already provided. Also, the function to emit global data has been implemented. Your job is to fill in the missing parts (the positions are marked with // TODO).

The necessary files for this assignment have been added to your SVN repositories. Copy the files into your trunk before beginning your work. Compile your code using the default make target as follows

```
$ make
```

which will generate the `snuplc` compiler.

In snuplc/reference/snuplc, you will find a binary of our reference implementation. You can use it to compare your compiler against our reference implementation. If you discover a bug, please let us know.

The assembly file generated by the compiler can be assembled into an executable file using GCC. If you use arrays or input/output, you will have to include the provided runtime libraries in the assembly process as follows:

```
$ snuplc f.mod             # generates f.mod.s
$ gcc -o f f.mod.s <path to RTL>/IO.s <path to RTL>/ARRAY.s
```

Now, you can execute the program with

```
$ ./f
```

Also consult Appendix II for more information about this.

Materials to submit:

- final source code of your compiler (use Doxygen-style comments)
- the final report describing the implementation of your entire compiler
  (the report is almost as important as your code. Make sure to put sufficient effort into it!)

Submission:

- the deadline of the final phase is **Thursday, December 3, 2019 at 14:00**.
- create a tag “Phase V” in your SVN directory. The tag creation time is regarded as the submission time.

Advance notice:

- the deadline to submit your presentation materials is **Sunday, December 8, 2019 at 14:00**. This deadline is independent of your presentation date (12/10 or 12/12).

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

Volume 2 contains the complete instruction set reference of Intel IA-32 processors.

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

`eax, ebx, ecx, edx, esi, edi, ebp, esp`

`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/1. 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)`

and the accessed location is

`mem[%base+ %index * scaling factor + displacement]`

We only require two sub-forms: `disp` and `disp(%base)` to access globals and locals/parameters/temps, respectively.

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
`cmpl S2, S1`	[condition codes] ← S1 – S2	set condition codes based on the comparison S1, S2

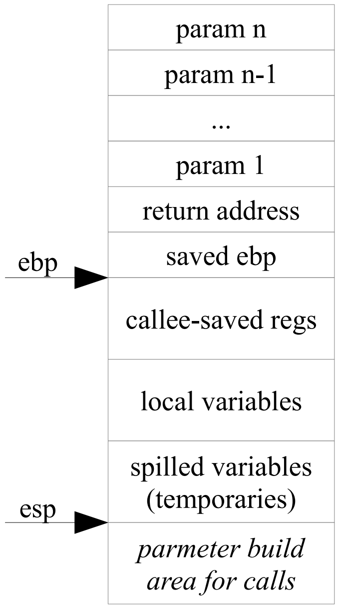
Instruction	Effect	Description
movl S, D	D ← S	move
cdq	[EDX:EAX] ← sign_extend([EAX])	sign-extend to 64-bit
pushl S	[ESP] ← [ESP] – 4 mem[ESP] ← S	push S onto stack
popl D	D ← mem[ESP] [ESP] ← [ESP] + 4	pop top of stack into D
call T	push return address continue execution at T	subroutine call
ret	pop 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 (a 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 statically. 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 .long <val> and .skip n assembly directives to allocate an initialized long value or *n* uninitialized bytes of memory, respectively. Do not forget to initialize the meta-data of arrays (dimensions) and the content of string constants.

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 is not allowed. 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 present, are returned in register eax. The registers ebx, esi, and edi are callee-saved and thus must be preserved across function calls. 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).

5. Procedure/function activation frame: below we give one possibility of a procedure activation frame. 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` instructions, the return address implicitly by the `call` instruction. Upon entering a function, 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`
3. save callee-saved registers
4. 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

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

After returning from the callee, the caller has to remove the procedure/function parameters from the stack.

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

Assembly programs are structured into sections. For our purposes, we require two sections: the `.text` section contains assembled machine code, while the `.data` section holds global variables.

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 ReadInt         # externally defined functions (I/O, array handling)
        ...
main:                           # module body, followed by functions/procedures
        ...


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

p:      .long 1                 # global array 'p': integer[10]
        .long 4
        .skip 40
x:      .skip 1                 # global variable 'x' (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.

The assembly file generated by snuplc can be compiled using gcc as follows

$ gcc -m32 -o primes.o -c primes.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.

To generate an executable file, the object file is linked together with the provided array and I/O routines as follows (here we assume that the libraries are located in subdirectory `rte/IA/`)

$ gcc -m32 -o primes primes.o rte/IA32/ARRAY.s rte/IA32/IO.s

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

$ gcc -m32 -o primes primes.mod.s rte/IA32/ARRAY.s rte/IA32/IO.s

The SnuPL/1 compiler can execute these commands for you if provided with the `--exe` option

$ snuplc --exe primes.mod

Now you can run your program with

$ ./primes

Appendix 3: GNU Debugger (GDB)

Debugging a generated assembly file can be challenging without a nice IDE. You may need to debug your x86 program using a command line 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 many people are not aware of.

To start debugging your program using gdb, type

$ gdb ./primes

at the prompt. GDB greets you with

```
GNU gdb (Gentoo 7.12.1 vanilla) 7.12.1
Copyright (C) 2017 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".
Type "show configuration" for configuration details.
…
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./primes...(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.