Phase 5: Code Generation

The fifth and last phase of our term project converts the IR code into x86_64 assembly code. After finalizing this phase, your SnuPL/2 compiler is complete: it takes programs written in SnuPL/2 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 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 x86_64 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 getting started. Appendix 3, finally, lists some useful GDB commands for the GNU debugger

All necessary modifications in this phase affect the file src/backendAMD64.cpp. Most likely you will also have to revisit certain functions that convert the AST into three-address code to make it work properly with the backend. To help you get started, the overall structure of the backend and some helper functions are already provided. In addition, the function that emits the global data has been implemented. Your job is to fill in the missing parts.

The necessary files for this assignment can be downloaded from eTL. Copy the files into your repository before beginning your work. Compile your code using the default make target as follows

```
$ make
```

which will generate the `snuplc` compiler.

We have placed a copy of our reference compiler in snuplc/reference/snuplc.ref. Use its output as a guideline how to implement certain features or to verify the code generated by your compiler. We do not expect byte-by-byte equivalence, in fact, that would rather be a reason for a score deduction than anything else. Please be aware that the reference compiler does have a few known problems. If you discover a bug, please let us know.

The SnuPL/2 runtime library (RTL) that contains the implementations of DIM, DOFS, and the I/O routines is provided in snuplc/rtl. We use a static library to avoid the trouble of locating the dynamically linked shared library when executing an SnuPL/2 program. You can build the runtime library for a given architecture by running make in the respective directory (this handout only contains the library for the AMD64 architecture).

```
$ cd snupl/rtl/x86-64
$ make                      # builds SnuPL/2 runtime library
```

The assembly files generated by the compiler can be assembled into an executable file with GCC. If you use I/O routines or arrays, the SnuPL/2 RTL needs to be provided at compile time:

```
$ snuplc f.mod              # generates f.mod.s
$ gcc -L../../snuplc/rtl/x86-64 -o f f.mod.s -lsnupl
```

Now, you can execute the program with

```
$ ./f
```

Consult Appendix II for more information about the compilation process.

You will find a number of test cases in test/codegen. The Makefile conveniently compiles & links SnuPL/2 files to executable files

```
$ make fibonacci
$ ./fibonacci
```

Materials to submit:

- Final source code of your compiler (use Doxygen-style comments)
- The final report describing the implementation of your entire compiler

Submission:

- Source code submission deadline: **Sunday, December 6, 2020 at 14:00**.
  Tag your submission with “SubmissionPhase5”
- Report submission deadline: **Sunday, December 13, 2020 at 14:00**.
  Tag the report submission with “SubmissionReport”.

Advance notice:

- The deadline to submit your presentation materials is **two days before your presentation date at 15:30**. You are not allowed to modify your presentation materials after this deadline.

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

Happy coding!

Appendix 1: AMD64 / Intel 64 (x86-64)

This appendix gives a minimal introduction to the Intel 64 instruction set and calling convention. Intel provides detailed manuals for its 64 and 32 bit architectures:
Four-Volume Set of Intel® 64 and IA-32 Architectures Software Developer's Manuals

Volume 2 contains the complete instruction set reference of Intel 64 processors.

1. Registers: Intel 64 has 16 general-purpose registers, fifteen of which can be used more or less arbitrarily.

```
rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp, r8-r15
```

`rsp` is the stack pointer that always points to the top of the stack. `rbp` can be used as a frame pointer to simplify accessing arguments and locals. The other registers can be used freely with few exceptions (for example, multiplication and division use predefined registers). x86_64 is fully backwards compatible; this is why the registers can be accessed in their 32, 16 and 8-bit form.

Not visible here are two important registers: the program counter (`%rip`) 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/2. 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 access location

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

We only use two sub-forms: `disp(%rip)` to access globals in a PC-relative fashion and `disp(%rsp)` (or `disp(%rbp)` if you use a frame pointer) to access variables on the stack.

Instruction	Effect	Description
`addl/q S, D`	D ← D + S	32/64-bit addition
`subl/q S, D`	D ← D – S	32/64-bit subtraction
`andl/q S, D`	D ← D && S	32/64-bit logical and
`orl/q S, D`	D ← D || S	32/64-bit logical or
`negl/q D`	D ← -D	32/64-bit negate
`notl/q D`	D ← ~D	32/64-bit logical not
`imull/q S`	[(E/R)DX:(E/R)AX] ← [(E/R)AX] * S	32/64-bit signed multiply
`idivl/q S`	[(E/R)AX] ← [(E/R)DX:(E/R)AX] / S	32/64-bit signed division
`cmpl/q S2,S1`	[condition codes] ← S1 – S2	set condition codes based on the comparison S1, S2
`movl/q S, D`	D ← S	32/64-bit move

Instruction	Effect	Description
`cdq`	[EDX:EAX] ← sign_extend([EAX])	sign-extend 32 to 64-bit
`cqo`	[RDX:RAX] ← sign_extend([RAX])	sign-extend 64 to 128-bit
`pushq S`	[RSP] ← [RSP] – 8 mem[RSP] ← S	push S onto stack
`popq D`	D ← mem[RSP] [RSP] ← [RSP] + 8	pop top of stack into D
`callq T`	push return address continue execution at T	subroutine call
`retq`	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. Depending on the data type, you may need to use the 32-bit ('l') or 64-bit ('q') variant of the instruction.

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: Linux follows the ELF x86-64 psABI that defines, among other things, the calling convention, i.e., how function parameters and results are passed. While you can use different calling conventions, we strongly recommend that you use the ELF x86-64 psABI also internally.

The most recent version of the ELF x86-64 psABI can be found here, a compiled copy is included in the handout (specification/5.x86-64.ABI.pdf).

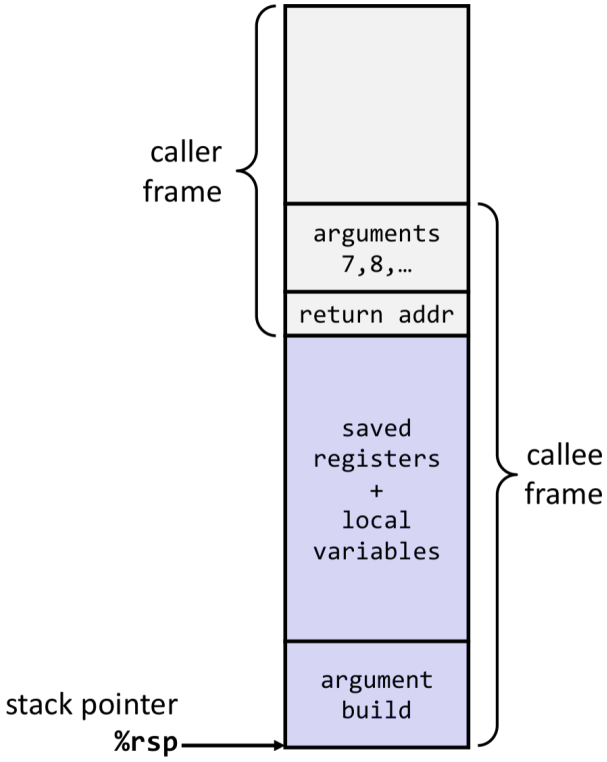
The stack grows towards smaller addresses. The stack pointer points to the top of the stack. x86-64 allows to use up to 128 bytes below the stack pointer (the so-called "red zone") without moving the stack pointer.

Parameters to functions are passed in registers and the stack. The exact rules are rather complicated, for our purposes the first six function arguments are passed in registers `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`. The 7th and further arguments on the stack reverse order. Function return values, if present, are returned in register `rax`. The registers `rbx`, `rbp`, `r12`, `r13`, `r14`, and `r15` are callee-saved and thus must be preserved across function calls. Implicitly, `rsp` is also callee-saved. The following figure illustrates the role of the registers; an illustration of the procedure activation frame is provided on the following page.

`rax`	caller saved/return value
`rbx`	callee saved
`rcx`	caller saved / argument 4
`rdx`	caller saved / argument 3
`rsi`	caller saved / argument 2
`rdi`	caller saved / argument 1
`rsp`	stack pointer
`rbp`	callee saved

`r8`	caller saved / argument 5
`r9`	caller saved / argument 6
`r10`	caller saved
`r11`	caller saved
`r12`	callee saved
`r13`	callee saved
`r14`	callee saved
`r15`	callee saved

5. Procedure/function activation frame: the standard ELF x86-64 psABI defines the following layout (this is from slide set 10 “The Runtime Environment”, page 15):



The parameters and the return address are generated by the caller. The parameters by a series of `mov` and `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. [optional] save `rbp` by pushing it onto the stack
2. [optional] set `rbp` to `rsp`
3. save callee-saved registers
4. generate space on the stack for locals and temporaries 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 temporaries by setting the stack pointer immediately below the callee-saved registers.
2. restore callee-saved registers
3. [optional] restore `rbp`
4. issue the `ret` instruction

To avoid problems with nested function calls, the reference compiler stores all argument in temporary values, then issues the opParam instruction immediately before the function call.

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 8            # align text section at an 8-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 8            # align at an 8-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 -c primes.mod.s
```

The -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 SnuPL/2 runtime library (array and I/O routines):

```
$ gcc -L../../snuplc/rtl/x86-64 -o primes primes.mod.o -lsnupl
```

Of course, this can be done in a single step:

```
$ gcc -L../../snuplc/rtl/x86-64 -o primes primes.mod.s -lsnupl
```

The SnuPL/2 compiler can execute these commands for you if provided with the `--exe` option

```
$ snuplc --exe primes.mod
```

The SnuPL/2 compiler supports a number of configuration options; have a look at

```
$ snuplc --help
```

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 9.2 vanilla) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later<http://gnu.org/licenses/gpl.html>
…
For help, type "help".
Type "apropos word" to search for commands related to "word".
(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, crashes, or hits a breakpoint
`c(ontinue)`	continue a stopped program until it ends, crashes, or hits the next breakpoint
`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`
`p(rint)[/format] expression`	inspect `expression`
`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. The GDB documentation and many GDB cheat sheets are helpful to familiarize yourself with GDB. GDB even includes a “graphical” mode. Refer to the GDB TUI documentation for more information on that.