

Programming Assignment 6

Code Generation

In this assignment you will learn how to implement code generation in a compiler. Specifically, you will implement x86-64 (64-bit x86, also called AMD64 or EM64T) assembly generation for the SimpliC language.

This assignment requires a Linux environment to work. You can log in remotely to the lab computers by connecting via SSH to `login.student.lth.se`. All the tasks should then work via the remote terminal.

The most time-consuming part of this assignment will be learning the x86-64 assembly syntax sufficiently to be able to write and read it. It is therefore important that you familiarize yourself with this document.

This assignment may seem daunting at first, but it can be a lot of fun once you get the hang of it!

- To prepare, read everything up to and including section 3. Also do all the preparation tasks, indicated by a black triangle in the margin.

Tasks you need to do in advance are marked with this symbol: ►, tasks to do during the lab are marked with ▷, and optional tasks with ★.

1 x86-64 Assembly

This document contains a very quick overview of the x86 architecture. It should be sufficient for the assignment at hand, however if you are interested in learning more about x86 assembly, please look at appendix E for links to some useful resources.

This document has a minimal instruction listing in section 1.6, and appendix F is an x86 assembly cheat-sheet, for quick reference.

- Print out appendix F.

1.1 Assembly Style

You should be aware that there are two common styles of x86 assembler syntax: AT&T and Intel. We will be using the AT&T style, however some documentation uses the Intel syntax. If you use such documentation, for example the Intel software developer manuals, refer to appendix A to learn the difference between the two styles. Keep in mind that the source/destination operands have opposite positions in the two styles!

1.2 Minimal Example

Here is a minimal Linux assembly program:

```
.global _start
.text
_start:
    movq $0, %rdi    # exit code
    movq $60, %rax   # sys_exit
    syscall
```

This program just does the equivalent of `System.exit(0)`, i.e., terminating the program successfully. The `_start` label is where the program will start executing and the first line makes sure that `_start` is visible to the linker. The `.text` line tells the assembler that what follows should go into the text section, which is a read-only and executable part of the produced binary. Non-executable things, such as buffers and constants, are usually placed in the `.data`, or `.bss` sections.

To assemble, link, and run the program, save it to a file named `exit.s`, then issue the following commands in a terminal prompt:

```
as exit.s -o exit.o
ld exit.o -o exit
./exit
```

The first command runs the assembler, GNU AS, to produce an object file named `exit.o`. The object file must then be linked to produce an executable binary. The resulting executable is written to the file `exit`. The third command runs the program.

If the output file name is removed from the `ld` command, then you will get an executable named `a.out` – the historical name for the assembler output.

- Follow the instructions above to assemble and link the program, then run it.

It is possible to look at the machine code for the assembled program. Just type this in your terminal: `objdump -d exit`. You should get something like this:

```
exit:      file format elf64-x86-64
```

Disassembly of section `.text`:

```
0000000000400078 <_start>:
400078:      48 c7 c7 00 00 00 00    mov     $0x0,%rdi
40007f:      48 c7 c0 3c 00 00 00    mov     $0x3c,%rax
400086:      0f 05                   syscall
```

The bytes on the left of the instructions, printed in hexadecimal, are the machine codes for the corresponding instruction.

1.3 Registers

Temporary results are stored in the general-purpose registers `RAX`, `RBX`, `RCX`, `RDY`, `RSI`, `RDI`, and `R8` through `R15`. These registers are 64 bits wide, however different parts of the registers can be used by specifying different variations of the register names. The table below shows some available registers names and their meaning:

Intended Purpose	64-bit	32-bit	16-bit	high 8-bit	low 8-bit
Accumulator	<code>RAX</code>	<code>EAX</code>	<code>AX</code>	<code>AH</code>	<code>AL</code>
Base index	<code>RBX</code>	<code>EBX</code>	<code>BX</code>	<code>BH</code>	<code>BL</code>
Counter	<code>RCX</code>	<code>ECX</code>	<code>CX</code>	<code>CH</code>	<code>CL</code>
Accumulator	<code>RDY</code>	<code>EDY</code>	<code>DX</code>	<code>DH</code>	<code>DL</code>
Source index	<code>RSI</code>	<code>ESI</code>	<code>SI</code>		
Destination index	<code>RDI</code>	<code>EDI</code>	<code>DI</code>		
Stack pointer	<code>RSP</code>	<code>RSP</code>	<code>SP</code>		
Stack base pointer	<code>RBP</code>	<code>EBP</code>	<code>BP</code>		
Instruction pointer	<code>RIP</code>	<code>EIP</code>	<code>IP</code>		

The 32-bit version of a register is just the low 32 bits, and the 16-bit version is the lowest 16 bits. The high and low 8-bit registers refer to the high and low parts of their 16-bit counterparts.

In addition to the registers above, which were available in 32-bit mode, 64-bit x86 adds eight new general-purpose registers: `R8`, `R9`, `R10`, `R11`, `R12`, `R13`, `R14`, and `R15`.

1.4 Operands

Each instruction takes 0-2 operands. The following types of operands exist:

- Register (short name = **r**)
- Immediate value (constant) (short name = **im**)
- Memory location (short name = **m**)

Each instruction allows only some of these operand types for each of its operands. To find out which types of operands are accepted for each instruction you must refer to a good x86 instruction reference, for example the Intel manual (see appendix E).

Here are some examples of operands:

Assembler	Operand type	What it means
\$0	immediate	decimal 0
\$0x10	immediate	hexadecimal 10 (=16 decimal)
1b1	memory location	value stored at address of label 1b1
1b1+2	memory location	value stored at two bytes after label 1b1
\$1b1	immediate	address of label 1b1
\$(1b1+4)	immediate	address of label 1b1 plus 4
%rdx	register	value stored in RDX
(%rax)	memory location	value at the address stored in RAX
8(%rbp)	memory location	value at eight bytes after the address stored in RBP
-3(%rax)	memory location	value at three bytes before the address stored in RAX

Instruction set references usually list which operand types are applicable for an instruction by listing the short name of the operand plus the accepted bit size(s). For example **r/m** means either a register or memory location.

1.5 Instruction Naming

Instruction names are derived from the base instruction mnemonic, and a suffix indicating the operand sizes. For example, a **MOV** with quad word operands has the name **movq** in GNU AS.

GNU AS Instruction suffixes:

Suffix	Operand size
b	byte, 8 bits
s	short, 16 bit integer, 32 bit float
w	word, 16 bit
l	long, 32 bit integer, 64 bit float
q	quad word, 64 bit

Using incorrect operands with an instruction will result in the assembler complaining. For example the line **movq %ebx, %eax** will produce the following error message: “Error: operand type mismatch for movq” because it is not allowed to use 32-bit operands with a 64-bit **MOV**. Unfortunately the assembler does not say which operand types it expected.

1.6 Instruction Listing

Below is a listing of a set of useful instructions, for quick reference. This listing follows the AT&T convention that the first operand is the source (*src*), and the second is the destination operand (*dest*). The table is simplified, and may imply impossible operand combinations. You should refer to a more reliable source, such as the Intel manuals, to see all allowed operand combinations and get a more precise description.

Instruction < Mnemonic – Description >	Operands		Operation
	<i>src</i>	<i>dest</i>	
ADD – Add	r/m/im	r/m	$dest \leftarrow dest + src$
AND – Bitwise logical AND	r/m/im	r/m	$dest \leftarrow AND(dest, src)$
CALL – Call procedure		r/m/im	push <i>RIP</i> , then $RIP \leftarrow dest$
CMP – Compare two operands	r/m/im	r/m	modify status flags similar to SUB
DEC – Decrement by 1		r/m	$dest \leftarrow dest - 1$
IDIV – Signed divide	r/m		signed divide <i>RDX</i> : <i>RAX</i> by <i>src</i> $RAX \leftarrow quotient, RDX \leftarrow remainder$
IMUL – Signed multiply (2 op)	r/m/im	r	$dest \leftarrow dest * src$
IMUL – Signed multiply (1 op)	r/m		$RDX : RAX \leftarrow RAX * src$
INC – Increment by 1		r/m	$dest \leftarrow dest + 1$
Jcc – Jump if condition is met		m/im	conditionally $RIP \leftarrow dest$
JMP – Unconditional jump		m/im	$RIP \leftarrow dest$
LEA – Load effective address	m	r	$dest \leftarrow addressOf(src)$
MOV – Move	r/m/im	r/m	$dest \leftarrow src$
NEG – Two's Complement negation		r/m	$dest \leftarrow -dest$
NOT – One's Complement negation		r/m	$dest \leftarrow NOT(dest)$
OR – Bitwise logical OR	r/m/im	r/m	$dest \leftarrow OR(dest, src)$
POP – Pop value off the stack		r/m	$dest \leftarrow POP(stack)$
PUSH – Push value on the stack	r/m/im		$PUSH(stack, src)$
RET – Return from procedure			restore <i>RIP</i> by popping the stack
SUB – Subtract	r/m/im	r/m	$dest \leftarrow dest - src$
SYSCALL – System Call			invoke OS kernel

Remember that to use 64-bit operands with an instruction you must use the 'q' suffix, e.g. `negq %rax`.

There are many variations of the conditional jump instruction, Jcc. Here are a few:

Instruction	Description
JE	Jump if equal
JNE	Jump if not equal
JG	Jump if greater than
JGE	Jump if greater than or equal
JL	Jump if less than
JLE	Jump if less than or equal

The CMP instruction is responsible for updating the status flags used by the conditional jump instructions. For example, the JGE instruction jumps to the given target if the *dest* operand of the previous CMP was greater than the *src* operand.

1.7 Linux System Calls

The simplest way to interact with the OS, and thereby use input/output, is to use the SYSCALL instruction. Different Linux system functions are selected based on the value of the RAX register. Arguments to the system call are placed in the RDI, RSI, and RDX registers. Here is a short list of some system calls and corresponding register assignments:

RAX	Function	RDI	RSI	RDX
0	<code>sys_read</code>	file descriptor	byte buffer address	buffer size
1	<code>sys_write</code>	file descriptor	byte buffer address	buffer size
60	<code>sys_exit</code>	error code		

A complete list of available x86-64 Linux system calls is available at <http://blog.rchapman.org/post/36801038863/linux-system-call-table-for-x86-64>. Note that 32-bit system calls work differently; this list is specifically for 64-bit assembly.

1.8 Simple I/O

This program demonstrates using the `sys_write` Linux system call to write to stdout:

```
1  .global _start
2  .data
3  message: .ascii "Hello edan65!\n"
4  .text
5  _start:
6      movq $1, %rdi      # stdout file descriptor
7      movq $message, %rsi # message to print
8      movq $14, %rdx     # message length
9      movq $1, %rax      # sys_write
10     syscall
11     movq $0, %rdi      # exit code = 0
12     movq $60, %rax     # sys_exit
13     syscall
```

Note that the string to print is placed in the `.data` section. See 1.2 for an explanation of sections.

The following program just reads one line from stdin, using `sys_read`, then echoes it to stdout:

```
1  .global _start
2  .data
3  buf: .skip 1024
4  .text
5  _start:
6      movq $0, %rdi      # stdin file descriptor
7      movq $buf, %rsi    # buffer
8      movq $40, %rdx     # buffer length
9      movq $0, %rax      # sys_read
10     syscall
11
12     movq $1, %rdi      # stdout file descriptor
13     movq $buf, %rsi    # message to print
14     movq %rax, %rdx    # message length
15     movq $1, %rax      # sys_write
16     syscall
17
18     movq $0, %rdi
19     movq $60, %rax
20     syscall
21
```

The `sys_read` function returns the number of bytes that were read in the `RAX` register, so we use that as the message length in the call to `sys_write`.

1.9 Procedures

It is useful to break out common parts of your code into procedures that can be re-used for multiple operations. This is the purpose of the `CALL` and `RET` instructions.

The `RIP` register is the instruction pointer register, sometimes also called program counter. It holds the address of the next instruction to execute, thereby determining the control flow of the program. The CPU automatically increases the instruction pointer to point at the next instruction right after the current one in the machine code (normally matches the assembly code layout). However the `CALL` instruction pushes the current instruction pointer on the stack, then sets `RIP` to the value of the operand. The matching `RET` instruction pops the return address from the stack and overwrites the instruction pointer.

```

        call fail          # call procedure "fail"
        movq %rax, %rdi
        movq $60, %rax
        syscall            # sys_exit
fail:
        movq $1111, %rax   # store exit code in %rax
        ret                # return

```

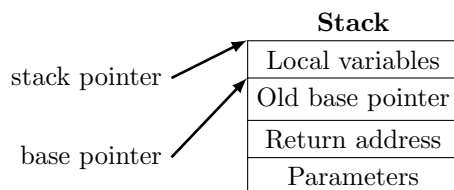
The above example shows a very simple procedure. It sets an error code in **RAX**, then it returns. The error code is then sent to the **sys_exit** function.

In order to be truly useful the procedures should be able to take arguments. This requires understanding stack frames, described in the next section.

1.10 Stack Management

The stack is composed of stack frames (activation records). The **RBP** (base pointer) register holds the address of a fixed position in the current stack frame, while **RSP** (stack pointer) holds the position of the current top of the stack. The stack “grows” toward lower addresses.

A stack frame typically looks like this:



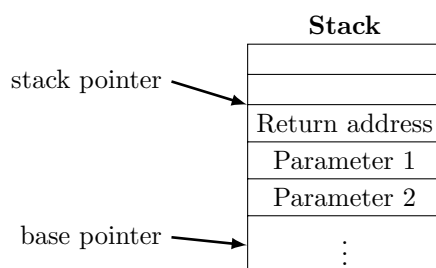
To call a procedure the arguments must first be pushed in reverse order. For example if a procedure **myproc** takes two arguments we might do this:

```

pushq param2
pushq param1
call myproc

```

Recall that the **CALL** instruction pushes the return address. This gives us the current stack:



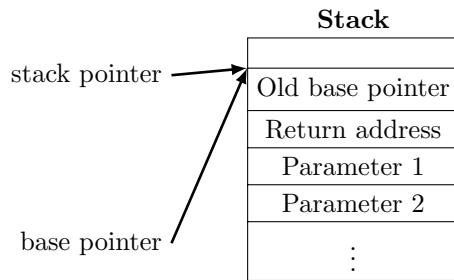
Now it is time to push the old base pointer (also called dynamic link), and update the new one:

```

myproc:
pushq %rbp
movq %rsp, %rbp

```

This gives us the current stack:

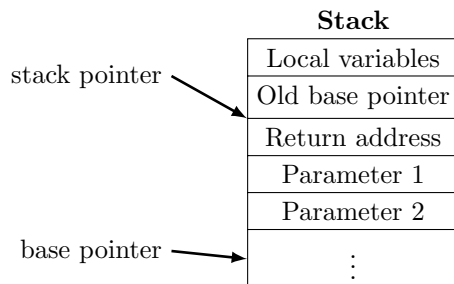


Now we can go ahead and push local variables on the stack, and use **RBP** to address those local variables. For example, the first local variable would have the address `-8(%rbp)`, because it is right above **RBP** on the stack which means it has a lower address and its size is 8 bytes, hence the `-8`.

To compute the address to some parameter we use **RBP** again, but now the parameter is lower on the stack, so it has a higher address. The first parameter is at `16(%rbp)` (can you figure out why `+16`?)

When the procedure is done doing its work it is time to restore the old **RBP**:

```
movq %rbp, %rsp
popq %rbp
```



All that remains now is to jump back to the return address and to pop parameters off the stack. We can use the **RET** instruction, which pops the return address off the top of the stack then jumps to that address. The parameters are still left on the stack after returning to the caller though, so the caller must ensure that they are removed:

```
ret
# ...
# at caller:
addq $16, %rsp # 2 parameters, 8 bytes each
```

Now the stack has been cleaned up properly – **RBP** and **RSP** are in their original state. If a return value is required then it should be stored in the **RAX** register, not on the stack.

- How would you compute the location of the return address in the stack.

1.11 Calling Procedures

To demonstrate the calling conventions presented in the previous section, here is a procedure that takes two parameters, first the pointer to a string to print, second the number of characters in the string:

```

# Procedure to print a message
print_string:
    pushq %rbp
    movq %rsp, %rbp
    movq $1, %rdi
    movq 16(%rbp), %rsi
    movq 24(%rbp), %rdx
    movq $1, %rax
    syscall
    popq %rbp
    ret

```

- Draw a diagram showing the stack after each instruction.
 - Modify the first program in section 1.8 to use the above procedure instead of using a `SYSCALL` in `_start`.
- Note:* In appendix B there are two procedures for the `read` and `print` functions in the SimpliC language. They will be useful for your generated code.

1.12 Loops and Conditionals

Coding loops and conditional expressions in assembly requires the use of conditional jumps (see section 1.6). A while loop can be generated like this:

```

    ### while ( %rax < %rdx )
loop_start:
    cmpq %rdx, %rax
    jge loop_end    # leave loop if %rax >= %rdx
    # loop body
    jmp loop_start   # jump to start, test condition again
loop_end:
    # done

```

Please note that the actual jump instruction has an inverted jump condition. This is because instead of testing whether the loop should iterate one more time, i.e. “continue if true”, we test if we want to leave the loop, “leave if false”.

There is another example of an inverted loop condition in the assembly code for the `read` procedure in appendix B.

An if-statement can be generated like this:

```

    ### if ( %rax >= 10 )
    cmpq $10, %rax
    jle else_lbl    # go to else_lbl if %rax < 10
then_lbl:
    # then body
    jmp fi
else_lbl:
    # else body
fi:
    # done

```

The `then_lbl` label was just added for clarity, it is not necessary.

In the assembler code for an if-statement the conditional jump will jump to the else-block of the statement. If the (inverted) condition is false, the execution steps to the next line, so that is where the then block is

placed. The then-code needs to be followed by a jump to the end of the if-statement so that the else-code is not executed after it.

Note that in the generated assembly all labels need to be unique!

- How would **break**- and **continue**-statements look in generated assembly?

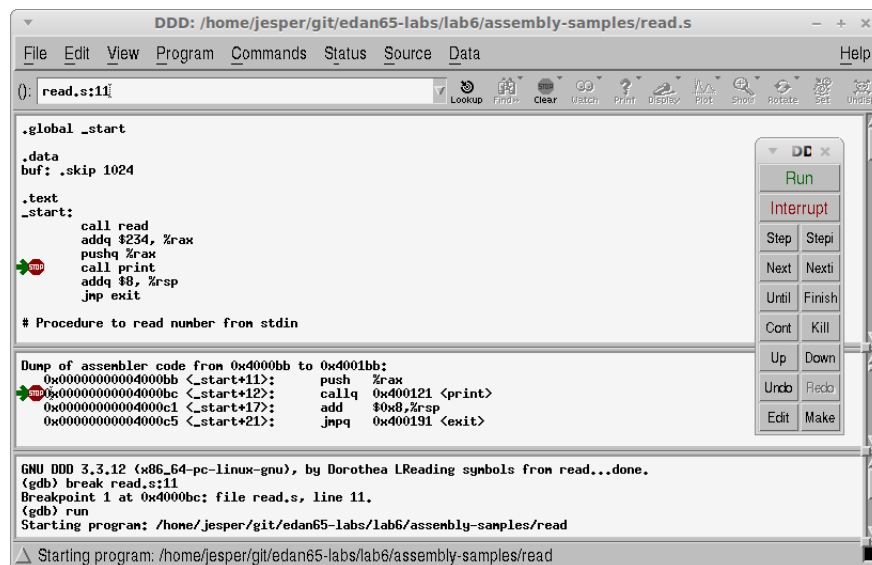
Boolean expressions in the SimpliC language are limited to a single relational expression each, such as `a == b`, `c >= d`, or `x < y`. In this case only a single conditional jump is needed for each boolean expression. If you are curious as to how more complex boolean expressions can be represented in assembly you should look at appendix D.

1.13 Debugging with DDD

The best tool to examine the inner workings of a compiled program is a debugger. For Linux the debugger of choice is often GDB, however it lacks a graphical interface. Instead you should try to use the *DDD* debugger, which runs as a graphical interface over GDB. If you have DDD installed, or are on a lab computer, you can issue the command `ddd program` in a terminal window to start the debugger. However, in order for DDD to be really useful you need to assemble with the `--gstabs` option so that DDD can display source code for your program. In other words, run the following commands to start debugging:

```
as --gstabs program.s -o program.o
ld program.o -o program
ddd program&
```

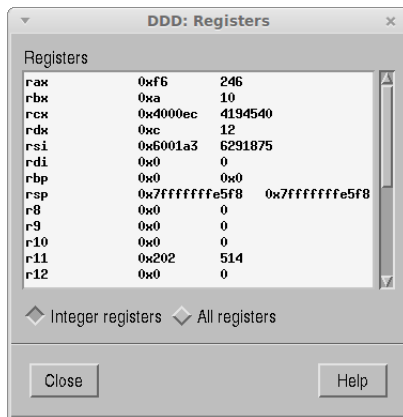
If done correctly you will see the following window (after a first-time welcome message):



If the source code is missing (as indicated by a mostly blank window), it means that you forgot to use the `--gstabs` option with `as`.

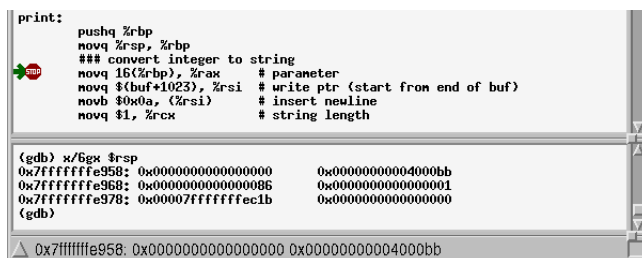
You can set a breakpoint by right-clicking on any whitespace on a source line (in the top part of the main window) and selecting the appropriate item in the pop-up context menu. After starting the program by clicking **Run** in the toolbox window, the program will run up to the breakpoint and halt. After the debugger hits the breakpoint you can single-step in the program by clicking the **Stepi** button in the toolbox window.

You can open the very useful register display via the **Status→Registers** menu item. The register display shows the current value of all registers:



Sometimes the program requires input, which is entered using the console at the bottom of the main DDD window. The Console can also be used to interact with the underlying GDB debugger.

If you wish to examine the stack, the simplest way is to use the GDB console and enter the command `x/6gx $rsp` which prints six quad-words starting at the address of `RSP` in hexadecimal format (use `x/6gd` instead to print it in decimal). This is demonstrated in the screen shot below:



In the above image you can see that the previous base pointer was zero, the return address is `0x4000bb`, and the first parameter is `0x86`.

Note on running DDD remotely: If you wish to run DDD from home, via the school's computers, you can do this by passing the `-X` flag to `ssh`. The `-X` flag enables X forwarding, so that graphical applications can be displayed through a remote connection. X forwarding is available even from Windows, however you must first install a local X server on your Windows machine.

2 CalcASM Demo

To demonstrate assembly code generation there is a demo project called CalcASM provided for this assignment.

The version of the Calc language used for this demo has been simplified somewhat in order to make the code generation much easier to read. The simplifications are:

- Only integer arithmetic is used.
- The `ask user` expression does not allow a default value.

- You should have a look at the `src/jastadd/CodeGen.jrag` file in the CalcASM project and try to understand what it does. Some points of interest are discussed in this section.

First, identify the different code parts generated by the **Program** node: the different segments (`.global`, `.data`, and `.text`), the `_start` label, the actual code of the Calc program, the system call to exit, and the helper functions for I/O. The helper functions are also listed in appendix B.

The CalcASM code generator demonstrates the following code-generation techniques:

- Allocating space for local variables

- Storing temporary values on the stack
- Basic arithmetic

Special things to note in the code generation:

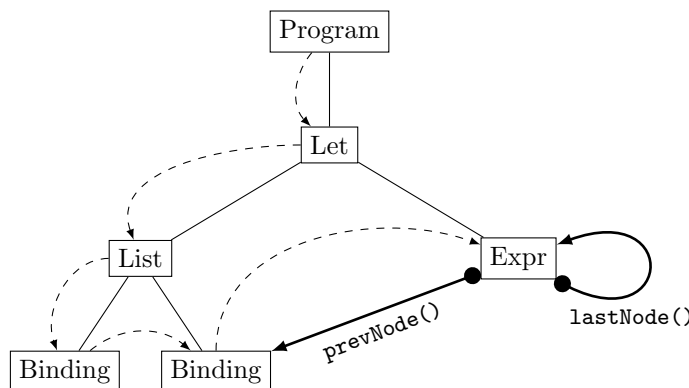
- The code generation for multiplication and division is not symmetrical. Remember to generate code specially for integer division in SimpliC.
- Every result of every expression is placed in **RAX**, then pushed on the stack, copied to another register, or used in a computation.
- To keep the code generation simple, the CalcASM compiler only uses the **RAX** and **RBX** registers to store intermediate values for *all* expressions. All other temporary results are stored on the stack.
- To allocate space for local variables (bindings), there are attributes to calculate the total number of locals, `numLocals()`, as well as the index of each local in the stack frame, `localIndex()` (and the solution is simpler than the ones sketched in lecture 12):

```
syn int ASTNode.numLocals() = lastNode().localIndex() - localIndex();
syn int ASTNode.localIndex() = prevNode().localIndex();
eq IdDecl.localIndex() = prevNode().localIndex() + 1;
```

Both these attributes are calculated using the `prevNode()` and `lastNode()` attributes which point to specific nodes relative to the current one in a pre-order traversal of the AST. The `prevNode()` attribute points to the previous node in a pre-order traversal of the whole tree, while the `lastNode()` points to the last node in a pre-order traversal of the subtree at the current node:

```
inh ASTNode ASTNode.prevNode();
eq ASTNode.getChild(int i).prevNode() = prevNode(i);
syn ASTNode ASTNode.lastNode() = prevNode(getNumChild());
syn ASTNode ASTNode.prevNode(int i) = i>0 ? getChild(i-1).lastNode() : this;
```

The dashed arrows in the figure below show a pre-order traversal in a tree. The `prevNode()` and `lastNode()` attributes are displayed for a node in the tree:



- Draw an arrow for both `prevNode()` and `lastNode()` at the **List** node in the above tree. In addition to these attributes we only need to add a start value for `localIndex()` so that the iteration over previous nodes will terminate:

```
syn int Program.localIndex() = 0;
```

- Build the CalcASM compiler and compile the `mul1.calc` program in the `testfiles/asm` directory. You should be able to understand what the generated assembly code does (if not, please refer again to the assembly introduction sections of this document).

- Compare the `mul1.calc` program with the generated assembly code. Try to find which parts of the generated assembly code correspond to parts of the Calc program.
- Look at the `src/jastadd/CodeGen.jrag` file to see how the assembly code is generated.
- Run the assembler and linker on the generated assembly code, and run the resulting binary executable program (see section 1.2). Does it compute the correct result?

2.1 Test Framework

There is a simple test framework in CalcASM to test the code generation. The framework compiles and runs test programs and checks if they compute the expected result. For each `.calc` file in the test directory `testfiles/asm` the test framework first runs the Calc compiler to generate the corresponding assembly code files (`*.s`), then the assembler, then the linker and finally the resulting binary is executed (`*.elf`). If the execution output matches the corresponding `.expected` file then the test passed. The test fails if something went wrong along the way or if the output did not match the expected output.

3 SimpliC Code Generation

You will now implement code generation for the SimpliC language, extending your compiler from assignment 5. As usual, work in small increments, and add test cases as you cover larger and larger parts of the language.

- Write down what the generated code should look like for the following small SimpliC programs:

Program 1

```
int main() {
    return 1;
}
```

Program 2

```
int f() {
    return 2;
}
int main() {
    f();
    return 1;
}
```

- Look through the remaining tasks and write down small test programs for them, as well as what the generated code should look like.
- **Task 1a** Implement code generation for function declarations. Your compiler should be able to compile program 1. To get started, look at the code generation for CalcASM. You should start with something like this:

```
public void Program.genCode(PrintStream out) {
    out.println(".global _start");
    // etc.
    out.println("_start:");
    // call main function

    for (FunctionDecl decl: getFunctionDeclList()) {
        decl.genCode(out);
    }
}
```

```

        // call sys_exit
        // helper procedures (print/read)
    }
    public void FunctionDecl.genCode(PrintStream out) {
        // your code here
    }

```

Recall that each expression should generate code that puts the resulting value in `RAX`, but so far the only expression you need to implement is integer literals.

- **Task 1b** Set up the testing framework. When running the compiled program, it should produce empty output. Check that it runs. Then set up a test framework in the same manner as for CalcASM, and add program 1 to your test suite. See section 3.2 for advice on testing. In the subsequent tasks, add your test programs to the automated test suite.

- **Task 2** Implement code generation for function calls with no arguments. Your compiler should be able to compile program 2.

- **Task 3** Implement passing of arguments in function calls.

When simple no-argument function calls have been implemented you can continue by adding parameter passing. This means that the parameter expressions must be evaluated, and their values pushed (in reverse order) on the stack before the function call. The expression evaluation code can be a placeholder for now that just puts some constant value in `RAX`.

- **Task 4** Check that you now can compile a program that prints a constant value. You can now start producing more interesting test cases that print out a specific value. Make sure your test programs are added to the test suite.

- **Task 5** Support constant-value expressions, e.g., $3 + (4/2) * 3$. Recall that the result of each expression should be stored in `RAX`. When evaluating binary expressions you can store the left operand value on the stack while the right operand is computed.

- ▷ **Task 6** Implement code generation for conditional jumps.

Boolean expressions, such as $3 > 1$, will only be used as conditions for `if`- and `while`-statements. For each boolean expression, a conditional jump will be generated. The target of the conditional jump is dependent on the control flow structure the boolean expression is used in. To generate these conditional jumps you can add a new code generation method called `genConditionalJump` in the binary expression AST class. This method should have a parameter which gives the target label to jump to if the expression is true.

- ▷ **Task 7** Implement code generation for `if`- and `while`-statements.

When boolean expression evaluation is working you can move on to the control flow statements `if` and `while`. Generating code for these requires some unique labels to be generated. See lecture 12 for ideas on how to do this.

- ▷ **Task 8** Implement addressing of formal parameters in functions.

As you should recall from section 1.10, function parameters have a positive address relative to `RBP`, and local variables have a negative address. The computation of an address of a local variable is shown in the CalcASM project. You will have to figure out how to compute the address of a function parameter.

Another issue is identifying which variable names refer to local variables, and which refer to parameters.

- How can you use an attribute to know if an `IdDecl` is a parameter declaration or a local variable declaration?

- ▷ **Task 9** Allocate space for local variables.

The number of local variables in a function decides how much stack space to allocate for local variables at the start of the function. For example, if two local variables are used then `RSP` is subtracted by 16 at the start of the function.

It is sufficient to count the total number of local variables used in a function. This will allocate enough room for all variables even if all of them are not used simultaneously. This is what the code for `numLocals()` in CalcASM does.

- ▷ **Task 10** Implement local variable addressing.

Local variable indexes are used to calculate stack addresses of local variables. Look at how local variable indexes are generated in CalcASM and implement the same for SimpliC.

You can reuse the following attributes from CalcASM to compute local variable indexes:

```
syn int ASTNode.localIndex() = prevNode().localIndex();
eq Program.localIndex() = 0;
eq IdDecl.localIndex() = prevNode().localIndex() + 1;

inh ASTNode ASTNode.prevNode();
eq ASTNode.getChild(int i).prevNode() = prevNode(i);
syn ASTNode ASTNode.lastNode() = prevNode(getNumChild());
syn ASTNode ASTNode.prevNode(int i) = i>0 ? getChild(i-1).lastNode() : this;
```

However, the above attributes enumerate variables globally. There is an additional attribute equation you should add to make the variable index numbering start over inside each function declaration.

- ▷ **Task 11** Check that your compiler works for a simple program that reads some input, performs some computation, and writes some output. Do not add this program to the automated test suite, as the testing framework does not handle programs with input.
- ▷ **Task 12** With all these tasks done, your code generation should work for any input program! Write new test cases that cover many different features of the code generation. One of these programs should be the gcd program from assignment 2. Undoubtedly there will still be bugs left, try to fix as many as you can.

3.1 Premature Optimization

A common mistake is to try and remove redundant copies of sub-expressions in the generated code. However, it is very important to above all make sure you generate *correct code*. Only when you are *absolutely certain* that the generated code is correct should you start thinking about optimizations.

For example, it may seem redundant to copy every single value into `RAX`, and in fact it is redundant in most cases. However, this is the simplest way to generate the code and it is a really good idea to aim for simple rather than efficient code, initially. In this course, we do not study optimizations, and to do them in a good way, you should take the course EDA230 (Optimizing Compilers).

3.2 Testing

To reuse the testing framework in CalcASM, simply copy the `TestCodeGeneration.java` file into your project. Note that the test framework expects the tested programs to exit with an exit code of zero. If it does not exit with zero you will get a test error.

To test single files without the test framework you can use the command line. It is then handy to combine a number of commands into a single one, as below:

```
java -jar compiler.jar x.sim > x.s && as --gstabs x.s -o x.o && ld x.o && ./a.out
```

The `&&` runs the next command in the list if the previous command succeeded.

4 Optional Challenges

★ When you have completed the compulsory part of the assignment you may try one or more of these optional challenges:

- Try linking with and calling your SimpliC functions from a C program. See appendix C.1 for more information.
- Instead of counting all variables in a method you may count the maximum number of “visible” local variables anywhere in the function. This way, variables that are not visible at the same time may reuse the same space on the stack. However, if you do this then the local variable indexing must also be altered.

A AT&T vs Intel

There are two main styles of x86 assembler syntax:

AT&T The default style used in GNU AS and GDB, and thus the most common style used in Linux assembler programming.

Intel This style is used by the NASM assembler.

The Intel style is generally considered easier to read, however we will be using the AT&T style because it is the default of many useful tools on Linux such as **as**, **gdb**, and **objdump**.

Although we will use the AT&T style it is useful to know the differences between AT&T and Intel style because both are common. Here are the most important differences:

- AT&T has the **%** prefix for all register names, and **\$** for all constant operands. The Intel style does not use prefixes for operands.
- Source and destination operands for most instructions have swapped positions. In AT&T instructions have the form **<instruction> <source> <destination>**, while Intel uses the form **<instruction> <destination> <source>**.
- In AT&T assembler, the operand size is coded using suffixes in the instruction mnemonic. In Intel assembler, special keywords are instead used on the operands when the assembler cannot figure out the operand size.

B Procedures for Input/Output

The **read** and **print** functions in the SimpliC language can be implemented in assembler like this:

```
# Procedure to print number to stdout
# C signature: void print(long int)
print:
    pushq %rbp
    movq %rsp, %rbp
    ### convert integer to string
    movq 16(%rbp), %rax    # parameter
    movq $(buf+1023), %rsi # write ptr (start from end of buf)
    movb $0x0a, (%rsi)    # insert newline
    movq $1, %rcx         # string length
    itoa_loop:            # do.. while (at least one iteration)
        movq $10, %rbx
        movq $0, %rdx
        idivq %rbx        # divide rdx:rax by 10
        addb $0x30, %dl    # remainder + '0'
        decq %rsi         # move string pointer
        movb %dl, (%rsi)
        incq %rcx         # increment string length
        cmpq $0, %rax
        jg itoa_loop     # produce more digits
    itoa_done:
        movq $1, %rdi
        movq %rcx, %rdx
        movq $1, %rax
        syscall
        popq %rbp
        ret
```



```

# Procedure to read number from stdin
# C signature: long int read(void)
read:
    pushq %rbp
    movq %rsp, %rbp
    movq $0, %rdi
    movq $buf, %rsi
    movq $1024, %rdx
    movq $0, %rax
    syscall                # %rax = sys_read(0, buf, 1024)
    ### convert string to integer:
    ### %rax contains nchar
    ### %rsi contains ptr
    movq $0, %rdx          # sum = 0
atoi_loop:
    cmpq $0, %rax          # while (nchar > 0)
    jle atoi_done          # leave loop if nchar <= 0
    movzbq (%rsi), %rbx    # move byte, and sign extend to qword
    cmpq $0x30, %rbx       # test if < '0'
    jl atoi_done           # character is not numeric
    cmpq $0x39, %rbx       # test if > '9'
    jg atoi_done           # character is not numeric
    imulq $10, %rdx        # multiply sum by 10
    subq $0x30, %rbx       # value of character
    addq %rbx, %rdx        # add to sum
    incq %rsi              # step to next char
    decq %rax              # nchar--
    jmp atoi_loop          # loop back
atoi_done:
    movq %rdx, %rax        # return value in RAX
    popq %rbp
    ret

```

Note that both procedures need a buffer named `buf` with room for at least 1024 bytes! The buffer should be allocated as in the second example of section 1.8.

The `read` and `print` procedures above correspond to the following C functions:

```

long int read() {
    long int nchar = sys_read(0, buf, 1024);
    long int sum = 0;
    char* ptr = buf;
    while (nchar > 0) {
        char c = *ptr;
        if (c < '0' || c > '9') {
            break;
        }
        sum *= 10;
        sum += c - '0';
        ptr++;
        nchar--;
    }
    return sum;
}

void print(long int num) {
    long int nchar = 1;
    char* ptr = &buf[1023];
    *ptr = '\n';
    do {
        long int remainder = num % 10;
        num = num / 10;
        ptr++;
        *ptr = ((char) remainder) + '0';
        nchar++;
    } while (num > 0);
    sys_write(1, ptr, nchar);
}

```

C C Interoperability

What we have described in section 1.10 is a calling convention, or Application Binary Interface (ABI), where parameters were placed on the stack, and the caller popped the parameters (caller clean-up). Different languages, architectures, and compilers use different calling conventions. There are even different calling conventions just for the C language.

When two procedures are compiled with different calling conventions they become incompatible. So, in order to be able to call C functions from our assembler code, or vice versa, we will have to follow the correct calling conventions. Since we are using 64-bit Linux for this assignment, we will use the *System V AMD64 ABI*.¹

The calling convention described in section 1.10 is similar to the System V convention, however some details differ:

- The first six arguments are passed in RDI, RSI, RDX, RCX, R8, and R9.
- Additional arguments are placed on the stack.
- Registers RBP, RBX, and R12-R15 are saved/restored by the callee. All other registers should be stored by the caller if needed.

The `read` and `print` procedures in appendix B break this calling convention in several ways, however you could generate assembler code that respects this calling convention and thereby becoming compatible with GCC-compiled C code.

C.1 Compiling with C

Here is a small demo of compiling assembler code together with a C program. The source of the C program, saved in a file named `main.c`:

```
#include<stdio.h>

long int sum(long int a, long int b);

int main(int argc, char** argv)
{
    long int the_sum = sum(30, 5);
    printf("the sum = %d\n", (int)the_sum);
    return 0;
}
```

The function prototype for `sum` is important because this tells GCC how it should call our assembly code. Here is the assembly code for the `sum` function, in the file `sum.s`:

```
.global sum
.text
sum:
    pushq %rbp
    movq %rsp, %rbp
    movq %rdi, %rax
    addq %rsi, %rax
    popq %rbp
    ret
```

¹http://en.wikipedia.org/wiki/X86_calling_conventions#System_V_AMD64_ABI

- ★ Copy the above code to the correct files, then compile it with GCC using the following command:
`gcc -gstabs sum.s main.c -o main` The compiled binary `main` can then be run using the command
`./main` Does it produce the expected output?

You can disassemble the executable to see how the `main` function was compiled by GCC using the following command: `objdump -d main`

D General Boolean Expressions

Generating code for general boolean expressions, such as `a>b && c==d`, gets a little more complicated than when only a single relational expression is used. The most intuitive way is to use a series of conditional jumps, for example:

```
### if (%rax == %rbx && %r9 > %r10)
    cmpq %rbx, %rax
    jne else_lbl
    cmpq %r10, %r9
    jle else_lbl
then_lbl:
    # then body
else_lbl:
```

With an and-expression no additional label is required, however if there is an or-expression (`a>b || c<d`), then an extra label is used to jump to the then-block in case the first expression was true. This short-circuits the evaluation of the second expression:

```
### if (%rax == %rbx || %r9 > %r10)
    cmpq %rbx, %rax
    je then_lbl
    cmpq %r10, %r9
    jle else_lbl
then_lbl:
    # then body
else_lbl:
```

Note that the test for the first condition became un-inverted (jump to then-label if true).

An alternative way to generate conditional expressions with AND- and OR-operators that is a bit simpler to generate, but does not have the benefit of short-circuiting redundant computations, is to evaluate each boolean expression to a value of zero or one, then conditionally jump depending on if the expression value was zero or one. This method utilizes the conditional flags set by the `CMP` instruction. The conditional flags can be copied to a one-byte register using the `SETcc` instructions, where `cc` is the condition code to copy.

Let's look at an example:

```
### if (%r8 == %r9 || %r10 > %r11)
    cmpq %r9, %r8
    movq $0, %rax
    sete %al
    pushq %rax
    cmpq %r11, %r10
    movq $0, %rax
    setg %al
    popq %rbx
    andq %rbx, %rax
    cmpq $0, %rax
    je else_lbl
then_lbl:
    # then body
else_lbl:
```

First we compare R8 and R9, then the **SETE** instruction is used to set the value of the conditional flag for equality, either 0 or 1, in the AL register. Note that **RAX** is zeroed first because **SETE** only sets one byte of **RAX**. The result is pushed on the stack, then the next relational expression is evaluated in a similar manner, however now the **SETG** instruction is used. The results of both relational expressions are combined using **AND** to compute the value of the whole boolean expression which is then used to conditionally jump as normal.

The following variants of **SETcc** exist:

Instruction	Description
SETE	Set equality flag
SETNE	Set inequality flag
SETG	Set greater than flag
SETGE	Set greater than or equal flag
SETL	Set less than flag
SETLE	Set less than or equal flag

These set the destination operand (which must be a byte register), to 0 or 1 depending on the result of the previous compare. The **SETcc** instructions work similarly to the **Jcc** instructions.

E Additional x86 Assembly Resources

We have given a short overview of x86 assembly in this document and intentionally skipped over some important parts, for example floating point arithmetic. For further information on x86 assembly please look at the following useful online resources:

- Wikibooks: x86 Assembly: http://en.wikibooks.org/wiki/X86_Assembly
- Introduction to the AT&T assembly syntax: http://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax
- The GNU Assembler (GNU AS) Manual: <http://tigcc.ticalc.org/doc/gnuasm.html>

Here are some online x86 instruction set listings:

- Wikipedia: http://en.wikipedia.org/wiki/X86_instruction_listings
- Intel Software Developer Manuals for x86-64: <http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html> (look for “instruction set reference”)
- x86asm.net: <http://ref.x86asm.net/coder64-abc.html> (very compact information)
- <http://x86.renejeschke.de/> (useful, but 32-bit only)

Additional links:

- System V ABI: <http://x86-64.org/documentation/abi.pdf>

F Assembler Cheat Sheet

General-purpose registers: RAX, RBX, RCX, RDX, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15.

Operands: **r** = register, **m** = memory location, **im** = immediate.

Operand	Type	What it means
\$0	im	decimal 0
\$0x10	im	hexadecimal 10 (=16 decimal)
lbl	m	value stored at address of label lbl
lbl+2	m	value stored at two bytes after label lbl
\$lbl	im	address of label lbl
\$(lbl+4)	im	address of label lbl plus 4
%rdx	r	value stored in RDX
(%rax)	m	value at the address stored in RAX
8(%rbp)	m	value at eight bytes after the address stored in RBP
-3(%rax)	m	value at three bytes before the address stored in RAX

Instruction < Mnemonic – Description >	Operands		Operation
	src	dest	
ADD – Add	r/m/im	r/m	$dest \leftarrow dest + src$
AND – Bitwise logical AND	r/m/im	r/m	$dest \leftarrow AND(dest, src)$
CALL – Call procedure		r/m/im	push <i>RIP</i> , then $RIP \leftarrow dest$
CMP – Compare two operands	r/m/im	r/m	modify status flags similar to SUB
DEC – Decrement by 1		r/m	$dest \leftarrow dest - 1$
IDIV – Signed divide	r/m		signed divide <i>RDX</i> : <i>RAX</i> by <i>src</i> $RAX \leftarrow quotient, RDX \leftarrow remainder$
IMUL – Signed multiply (2 op)	r/m/im	r	$dest \leftarrow dest * src$
IMUL – Signed multiply (1 op)	r/m		$RDX : RAX \leftarrow RAX * src$
INC – Increment by 1		r/m	$dest \leftarrow dest + 1$
Jcc – Jump if condition is met		m/im	conditionally $RIP \leftarrow dest$
JMP – Unconditional jump		m/im	$RIP \leftarrow dest$
LEA – Load effective address	m	r	$dest \leftarrow addressOf(src)$
MOV – Move	r/m/im	r/m	$dest \leftarrow src$
NEG – Two's Complement negation		r/m	$dest \leftarrow -dest$
NOT – One's Complement negation		r/m	$dest \leftarrow NOT(dest)$
OR – Bitwise logical OR	r/m/im	r/m	$dest \leftarrow OR(dest, src)$
POP – Pop value off the stack		r/m	$dest \leftarrow POP(stack)$
PUSH – Push value on the stack	r/m/im		$PUSH(stack, src)$
RET – Return from procedure			restore <i>RIP</i> by popping the stack
SUB – Subtract	r/m/im	r/m	$dest \leftarrow dest - src$
SYSCALL – System Call			invoke OS kernel

Operand size suffix: **b** = 1 byte, **w** = 2 bytes, **l** = 4 bytes, **q** = 8 bytes.

Use instruction mnemonic + suffix to get the instruction name. For example: `negq, movq, movl`.

Conditional jumps:

Instruction	Description
JE	Jump if equal
JNE	Jump if not equal
JG	Jump if greater than
JGE	Jump if greater than or equal
JL	Jump if less than
JLE	Jump if less than or equal

Conditional jump example:

```
cmp op1, op2
jge lbl          # jump to lbl if op2 >= op1
```