THE UNIVERSITY OF SYDNEY

COMP3109 Assignment 3

Writing a Compiler – Group Assignment (10 marks)

This third assignment is a **group assignment**, and is due **on Friday Week 11, 5pm**. Your assignment will not be assessed unless all two of the following criteria are met:

1. Hand in a signed group academic honesty form in the tutorial
2. For this assignment write at least 15 test cases and submit it with your assignment
3. Submit a tarball of your source code to elearning for all solved tasks with plenty of remarks, i.e., documentation should be in form of remarks.

Please form groups of two or three students (groups of three preferred). Your code must run on the `ucpu[01]` machines. When submitting, please provide a `README` file instructing the marker how to run your code.

Task 1 (10 marks)

We want to implement a mini compiler for a vector language that translates input programs to Intel's assembly code using SSE extensions. The compiler should be implement in a language tool of your choice that is able to deal with attribute grammars (e.g. ANTLR, YACC, etc; note that we recommend ANTLR). A module in the vector language consists of several functions. Each function has several input and output vectors. We further assume that all vectors of a function have the same length. A function in the vector language consists of a vector assignments, sequences of vector assignments, if- and while-statements, and calls to other functions defined in the vector language.

The grammar of the language is given below, where M is the start symbol.

$$
\begin{array}{rcl}
M & \rightarrow & F \quad M \\
M & \rightarrow & \varepsilon \\
F & \rightarrow & \textbf{func} \quad \textbf{ident} \quad P \quad D \quad S \quad \textbf{end} \\
P & \rightarrow & \textbf{(} \quad L \quad \textbf{)} \\
L & \rightarrow & \textbf{ident} \\
L & \rightarrow & \textbf{ident} \quad \textbf{,} \quad L \\
D & \rightarrow & \textbf{var} \quad L \quad \textbf{;} \\
D & \rightarrow & \varepsilon \\
S & \rightarrow & \textbf{if}\ C\ \textbf{then}\ S\ \textbf{else}\ S\ \textbf{endif} \\
S & \rightarrow & \textbf{while}\ C\ \textbf{do}\ S\ \textbf{endwhile} \\
S & \rightarrow & S \quad \textbf{;} \quad S \\
S & \rightarrow & \textbf{ident} \quad \textbf{=} \quad E \\
S & \rightarrow & \varepsilon
\end{array}
$$

$$
\begin{array}{rcl}
E & \rightarrow & E \quad \textbf{+} \quad E \\
E & \rightarrow & E \quad \textbf{-} \quad E \\
E & \rightarrow & E \quad \textbf{*} \quad E \\
E & \rightarrow & E \quad \textbf{/} \quad E \\
E & \rightarrow & \textbf{min} \quad \textbf{(} \quad E \quad \textbf{,} \quad E \quad \textbf{)} \\
E & \rightarrow & \textbf{(} \quad E \quad \textbf{)} \\
E & \rightarrow & \textbf{ident} \\
E & \rightarrow & \textbf{num} \\
C & \rightarrow & \mathrm{E} < \textbf{num} \\
\textbf{ident} & \rightarrow & \texttt{[a-zA-Z\_][a-zA-Z0-9\_]}\star \\
\textbf{num} & \rightarrow & \texttt{[0-9]+(\textbackslash.[0-9]+)?}
\end{array}
$$

Write an attribute grammar that translates an input program in the vector language to an assembly file. The assembly output should take advantage of the SSE extension (high speed floating point operations which can operate on 4 single precision numbers at the same time) of the x86-64 architecture. The produced assembly code should define functions that can be used in C programs. The structure of this exercise is to put together assembly templates (i.e. blocks of text) which are given in the rest of this assignment. These text blocks need to be pasted and written to a file. Some of these text blocks need specific parameters that are to be computed by the attributes of the grammar.

For example the following module in the vector language defines a function **mymin**

```
func mymin(a, b, c)
 c = min(a,b) + 20
end
```

that has three parameters **a**, **b** and **c**. The element-wise minimum is computed, the constant vector 20 is added, and the result is stored in parameter **c**. The function **mymin** can be used as sketched in the following C code fragment:

```
#include <stdlib.h>

/* alignment macro: aligns a memory block a to multiplies of a */
#define align(s,a) (((size_t)(s) + ((a) - 1)) & ~((size_t) (a) - 1))
/* Alignment for SSE unit */
#define SSE_ALIGN (16)
/* Number of elements */
#define NUM (100)

extern void mymin(long, float *, float *, float *);

int
main(void) {
  float *a = malloc(sizeof(float)*NUM + SSE_ALIGN),
        *b = malloc(sizeof(float)*NUM + SSE_ALIGN),
        *c = malloc(sizeof(float)*NUM + SSE_ALIGN);
  /* make sure that pointers are aligned to multiples of 16 bytes */
  a = (float *) align(a, SSE_ALIGN);
  b = (float *) align(b, SSE_ALIGN);
  c = (float *) align(c, SSE_ALIGN);
  ...
  /* write values to a and b */
  ...
  /* invoke the function written in the vector language */
  mymin(NUM, a, b, c);
  ...
  /* read values from c */
  ...
  return 0;
}
```

In this code fragment three single precision floating point arrays **a**, **b** and **c** are declared and used as parameters of the vector language function **mymin**. Before using the arrays as vectors they need to be aligned to multiplies of 16 – this is an `SSE` requirement. Additionally, we assume that the length of a vector is a multiple of 4 because a single `SSE` instruction can operate on 4 single precision floating point numbers at the same time. The first parameter gives the length of the vectors followed by the vector parameters. Depending on the semantics of the function, the function either reads or writes from the arrays.

When you write assembly code you need to be aware of the processor architecture and how the program stack is structured (see textbook) so that you can access the length of the vectors and vector parameters. The 64bit extension of the Intel's 32bit architecture is called *x86-64* and it has some differences to the standard Intel 32bit architecture. The *x86-64* architecture has following features:

- We have sixteen 64bit-registers called `%rax`, `%rbx`, `%rcx`, `%rdx`, `%rsi`, `%rdi`, `%rbp`, `%rsp`, `%r8`, and `%r9` - `%r15`. The register `%rsp` is a special register that holds the top of the stack pointer and the register `%rbp` is normally used as a frame pointer register.
- Memory addresses are 64bit long.
- The first six integer and pointer parameters are passed via registers rather than the program stack. The seventh parameter and any following parameters after the seventh parameter are passed via the program stack. Refer to following table:

Argument	Register
1.	`%rdi`
2.	`%rsi`
3.	`%rdx`
4.	`%rcx`
5.	`%r8`
6.	`%r9`

- Register `%rax` stores the return value of a function.
- Register `%rbx` and `%r12` - `%r15` are callee-saved, i.e., if a function modifies these registers it needs to restore them before the function terminates.
- Register `%xmm0` - `%xmm15` are floating point registers that can store four floating point numbers simulatenously.

Recall that the program stack grows downwards. For the x86-64 architecture, the stack layout is given below,

...	higher addresses	
return address	8	*saved by the call instructions*
frame pointer of calling functions	0	*saved as soon as the function is invoked*
first free element on stack	-8	*value of frame pointer*
...	lower addresses	

The stack pointer in x86-64 is denoted by `%rsp` and the frame pointer by `%rbp`. A function definition in the vector language uses following template to produce assembly code:

```
.text
.global <name>
.type <name>, @function
.p2align 4,,15

<name>:
    # save current frame pointer on stack
    pushq   %rbp
    # set frame pointer
    movq    %rsp, %rbp
    # save callee-save registers that are used on stack
    pushq   %rbx

    <allocate memory for local vector variables>

    <Insert the body of function here>

    # epilog of a function
    popq   %rbx      # restore reg %rbx
    leave            # restore frame pointer
    ret              # return
```

where `.text` denotes that the machine code should be stored in the text segment (see textbook) of the memory. The pseudo-mnemonic `.global <name>` defines a global name `<name>` for the linker, and the pseudo-mnemonic `.type` declares that the symbol `<name>` is a function. With `<name>:` you define an address for the function entry point. The next instruction pushes the frame pointer of the calling function onto the stack. The second instruction moves the current value of the stack pointer to the frame pointer. The third instruction saves register `%rbx` onto to stack since it is callee save, i.e., the callee expects that `%rbx` does not change its value. Note we will use `%rbx` for the code templates and hence it needs to be saved at the entry and restored at the exit.

After the first three instructions you add the code of reserving space on the stack for local variables and the assignments of the vector function. A function definition ends with loading the previous value of `%rbx` from the stack, and executing **`leave`**, which restores the frame pointer and the stack. Instruction **`ret`** loads the return address into the instruction pointer of the CPU and returns to the next instruction of the caller.

Local variables in the vector language are stored on the stack from `-16(%rbp)` onward. The memory block for all local variables are created by subtracting the product of the length of a vector and the number of local stack variables from the stack pointer and adding a padding for pointer alignment, i.e.,

```
# allocate <NUM> local variables
movq  %rdi, %rax
imulq $4, %rax, %rax
addq  $16, %rax
imulq $<NUM>, %rax, %rax
subq  %rax, %rsp
andq  $-16, %rsp
```

where `<NUM>` is the number of local variables. The number of local variables can be computed by an attribute in your attribute grammar. This text block needs to be generated as a first text block in the function body (if local vector variables are in the function).

The following three templates are concerned about computing addresses of vector parameters, local variables, and constants. These three text blocks will be used for address computations in templates that either compute new vectors or assign a vector/constant to another vector. The address of a vector parameter is stored in an argument register. We assume that there are at most 5 vector parameters for a function since we have only six argument registers and the first one is used for storing the length of the vectors.

```
# place address of <N>th parameter into <destreg>
movq   <argreg-N+1>, <destreg>
```

where `<argreg-N+1>` is the $(n+1)$'th argument register as listed above. The destination register `<destreg>` is one of the following registers `%rax`, `%r10` or `%r11` that are used in the code templates for vector assignment and vector operations. An address of a local vector variable is computed by

```
# place address of <N>th local variable into <destreg>
movq  %rdi, <destreg>
imulq $4, <destreg>, <destreg>
addq  $16, <destreg>
imulq $<N>, <destreg>, <destreg>
subq  %rbp, <destreg>
negq  <destreg>
andq  $-16, <destreg>
```

where `<N>` is the nth local variable (starting from 1) of the vector function. The address has to be computed at runtime because the length of the vectors is not known ahead of time. The last template loads the address of a constant `<X>` into a destination register:

```
# place address of <X> into <destreg>
movq  $.const<X>, <destreg>
```

where the label `.const<X>` is a new label. For generating constants we need a postamble at the end of the assembly code that describes the constant:

```
.data
.align 16
.const<X>:
    .float <X>
    .float <X>
    .float <X>
    .float <X>
```

where `<X>` is the floating point number `X`. We need four repetitions because an `SSE` operation does four operations in a single step.

In the following we introduce assembly code templates for basic forms. In the following, we will introduce the code templates for the basic forms. There are two types of basic forms as shown below,

ident = factor

or

ident = factor **op** factor

where “factor” is either a variable or a constant. Conceptually, you break up the right-hand side of an expression in basic forms. In this conceptual transformation (which is also known as linearization of the code), you will introduce numerous local variables, e.g., the input statement

```
x = a + b + c + d
```

will be conceptually transformed to

```
x1 = a + b;
x2 = x1 + c;
x3 = x2 + d;
```

Note that I do not suggest to perform this transformation. I suggest to create local variables on the fly by using attributes and an expression node in the syntax tree representing a basic form. Furthermore, you want to reuse local variables as much as you can too avoid stack size explosion.

The basic form “**ident** = factor” is translated to assembly with following template:

```
    <load source address into  %rax>
    <load destination address into %r10>

    movq   %rdi, %rbx     # load vector length into counter %rbx
    shrq   $2,    %rbx    # divide counter reg by 4
                          #  (per loop iteration 4 floats)
    jz     .loop_end<X>   # check whether number is equal to zero

.loop_begin<X>:           # loop header

    movaps (%rax), %xmm0  # load source  into %xmm0
    movaps %xmm0, (%r10)  # store %xmm0

    # IMPORTANT: remove the following line only if %rax is
    # pointing to a constant
    addq   $16, %rax      # increment source pointer by (4 x float)

    addq   $16, %r10      # increment destination pointer by (4 x float)

    decq   %rbx           # decrement counter
    jnz    .loop_begin<X> # jump to loop header if counter is not zero

.loop_end<X>:
```

where `<X>` is a unique number for this basic form. This text block loads the source and destination address into `%rax` and `%r10` by using the templates as previously shown. The assembly code inside loads the vector length and divides it by 4 (i.e. shift of number by two digits). Inside the loop the **`movaps`** instructions load the instructions into the first SSE register. After performing the move instruction, the first SSE register will contain 4 consecutive floating point numbers. With the second **`movaps`** instruction the 4 consecutive floating point numbers are written to the destination.

The basic form **ident** = factor **op** factor is translated with following template:

```
    <load source1 address into  %rax>
    <load source2 address into  %r10>
    <load destination address into %r11>

    movq   %rdi, %rbx     # load vector length into counter %rbx
    shrq   $2, %rbx       # divide counter reg by 4
                          #  (per loop iteration 4 floats)
    jz     .loop_end<X>   # check whether number is equal to zero

.loop_begin<X>:           # loop header

    movaps (%rax), %xmm0  # load first operand into %xmm0
    movaps (%r10), %xmm1  # load second operand into %xmm1

    # perform operation
    <operation> %xmm1, %xmm0

    movaps %xmm0, (%r11)  # store result

    # increment pointers

    # IMPORTANT: remove following line if %rax is pointing to a constant
    addq   $16, %rax

    # IMPORTANT: remove following line if %r10 is pointing to a constant
    addq   $16, %r10

    addq   $16, %r11

    decq   %rbx           # decrement counter
    jnz   .loop_begin<X>  # jump to loop header if counter is not zero
.loop_end<X>:
```

Use the operation **`addps`** for addition, **`subps`** for subtracting, **`divps`** for division, **`mulps`** for multiplication, and **`minps`** for minimum.

Make sure that the code generation works for sequence assignments first before you attempt if-statements and while-loops. If-statements and whil-loops have a condition of the form E**< num** that computes the sum of the vector elements of E and checks whether the sum is less than num. Depending on the outcome either the true-branch or false-branch is executed or the while loop terminated. For the generation of assembly code for if-statements and while-loops you need templates that generate the necessary jump instructions.

For if-statements the template is the following

```
  <template for condition(.true-branch<num> , .false-branch<num>)

 .truebranch<num>:

  <emit code for true-branch here>

  jmp .endif<num>
```

```

.falsebranch<num>:

  <emit code for false-branch here>

.endif<num>:
```

where for each if-statement you need unique labels. The template for a while loop is given below

```
  jmp .loopcond<num>

.loopbegin<num>:
  <emit code for loop-body here>

.loopcond<num>:
  <template for condition (.loopbegin<num>,.loopexit<num>)

.loopexit<num>:
```

where for each while-statement you need unique labels as well.

The templates for the sums are given below:

```
    <load source address into  %rax>

    xorps  %xmm1, %xmm1    # set register %xmm1 to zero
    movq   %rdi, %rbx      # load vector length into counter %rbx
    shrq   $2,    %rbx     # divide counter reg by 4
                           # (per loop iteration 4 floats)
    jz     .loop_end<X>    # check whether number is equal to zero

.loop_begin<X>:            # loop header


    addps  (%rax), %xmm1   # add source to %xmm0

    # IMPORTANT: remove the following line only if %rax is
    # pointing to a constant
    addq   $16, %rax       # increment source pointer by (4 x float)

    decq   %rbx            # decrement counter
    jnz    .loop_begin<X>  # jump to loop header if counter is not zero

.loop_end<X>:
    xorps  %xmm0, %xmm0      # set register %xmm0 to zero
    addss  %xmm1,%xmm0       # add all four numbers in %xmm1 to %xmm0
    shufps $147,%xmm1,%xmm1  # note that
    addss  %xmm1,%xmm0       # a shuffle operation rotates the single precision
    shufps $147,%xmm1,%xmm1  # to the left. In each step
    addss  %xmm1,%xmm0       # a number is read and added to the single
    shufps $147,%xmm1,%xmm1  # precision number in %xmm0
    addss  %xmm1,%xmm0
```

```

                                # compare sum with number and jump
                                # to the true/false target
        ucomiss .L<number>,%xmm0
        ja      <true>

        jmp     <false>

        .align  4
    .L<number>:
        .float  <number>
```

After you have written the stub in C and produced code you can create an executable with

```
gcc -Wall -W -g -o myexe stub.c compiled.s
```

and you can trace the executable with `gdb`. The emission of the assembly code might be done as a traversal in your parser generator tool to avoid awkward data structures for storing the text file.

For speed fanatics: you can achieve substantial more speed by unrolling the loop and pre-computing the addresses of local variables.

Two scripts which might come in use. This first script called `compile.sh`, is used to compile our sample solution to the final executable. Our ANTLR grammar file is called `VPL.g` (Vector Processing Language).

```bash
#!/bin/bash
set -e

BUILD_DIR=build

if [ ${#} != 1 ]; then
  echo "Usage: ${0} filename.vpl" >&2
  exit 1
fi

# builds the ANTLR-generated parser using the grammar file
java -cp antlr-3.1.2.jar org.antlr.Tool -o ${BUILD_DIR} VPL.g
touch ${BUILD_DIR}/__init__.py

# uses the ANTLR-generated parser to convert the VPL program to ASM
./vpl2asm.py < ${1} > ${1}.s

# compiles the ASM and C file together
gcc -Wall -W main.c ${1}.s -o my_program
```

The second script, `vpl2asm.py`, invokes our Python parser generated by ANTLR:

```python
#!/usr/bin/env python
import sys
import antlr3
from build.VPLLexer import VPLLexer
from build.VPLParser import VPLParser

char_stream = antlr3.ANTLRInputStream(sys.stdin)
lexer = VPLLexer(char_stream)
tokens = antlr3.CommonTokenStream(lexer)
parser = VPLParser(tokens)
root = parser.prog()
```

These scripts are useful if you use Python as your target langauge within ANTLR. However, they can be easily adapted for your own chosen target language.