Software Compiler Assignment on:

Vector Operations

The Software Compiler course exam is composed by two parts. One is a written test, the other is an homework, to be terminated before course last class. The written test contributes with the 40% to the whole grade, while the homework contributes with the remaining 60%. To pass the whole exam, you must get a pass grade from both the test and the homework.

During the lab classes, we show you the ACSE compiler and the associated assembler [2]. They must be used as starting point for your homework. During the last class, you must present your work, showing your edits, giving a brief demo, and responding at some questions given by course lecturers.

Sources of ACSE can be found on the course site [3]. It is a tarball of the ACSE mercurial [1] repository. You are required to version your code with mercurial. A copy of your edits must be submitted to course lectures in the form of a patch with respect to the version of ACSE you have used as starting point.

Assignment

You are required to modify both the front-end of the compiler and the back-end.

Front-end

Vector operations allows to perform simple operations – e.g. adding, subtracting, ... – to all elements stored in an ACSE array. Figure 1 shows some examples.

You are required to add support to the ACSE front-end for handling the following vector operations:

Addition The `vec_add(c, a, b)` statement perform an element wise addition between arrays `a` and `b`, storing the result into the array `c`, that is:

$$\forall i \,|\, 0 \leq i < length(c),\, c[i] \leftarrow a[i] + b[i]$$

The three arrays must have the same length. If this does not hold, a compile-time error must be raised. Code generator must target the standard ACSE assembly.

```
int a[10], b[10],
    c[10], d[10];

vec_add(c, a, b);
vec_sub(d, a, b);
```

```
int a[10], b[10], c[10];
int cond;

vec_write(a);
vec_blend(c, a, b, cond);
```

Figure 1: Example of vector operations

Subtraction The `vec_sub(c, a, b)` statement perform an element wise subtraction between arrays `a` and `b`, storing the result into the array `c`, that is:

$$\forall i \,|\, 0 \leq i < length(c),\, c[i] \leftarrow a[i] - b[i]$$

The three arrays must have the same length. If this does not hold, a compile-time error must be raised. Code generator must target the standard ACSE assembly.

Printing The `vec_print(a)` statement print the whole array. Code generator must target the standard ACSE assembly.

Blending The `vec_blend(c, a, b, cond)` allows setting `c` according to expression `cond`. If it holds, then vector `a` is copied into vector `c`. If it doesn't hold, then vector `b` is copied into vector `c`:

$$c \leftarrow \begin{cases} a & \text{if} \quad cond \neq 0 \\ b & \text{if} \quad cond = 0 \end{cases}$$

The three arrays must have the same length, otherwise a compile-time error must be raised. Code generator must target standard ACSE assembly.

Back-end

Peephole optimizations are optimizations applied later in the compilation process. Their goal is to identify groups of subsequent instructions that can be substituted with another, more efficient, instruction.

Let `a`, `b`, `c`, three labels in the data segment associated to `SPACE` directives. They can be seen as a representation of ACSE arrays in assembler language. Suppose that the following instructions are added to the ACSE assembly language:

- `VADD c a b`: interprets memory locations identified by `a`, `b`, and `c` as three 4 elements arrays, and performs element-wise addition of `a` and `b`, storing the result in `c`
- `VSUB c a b`: interprets memory locations identified by `a`, `b`, and `c` as three 4 elements arrays, and performs element-wise subtraction of `a` and `b`, storing the result in `c`

```
...
MOVA R1 L1
ADD R2 R0 (R1)
MOVA R1 L2
ADD R3 R0 (R1)
ADD R1 R2 R3
MOVA R2 L0
ADD (R2) R0 R1
...
```

(a) Source

```
...
VADD L0 L1 L2
...
```

(b) Optimized

Figure 2: Folding an addition

You are required to implement a simple peephole optimization, looking for instruction sequences in the ACSE assembler that can be implemented using `VADD` or `VSUB`. Figure 2 shows an example.

If the pattern reported in Figure 2(a) is repeated 4 times without interruptions, then the optimizer can fold the addition and generate the code reported in Figure 2(b).

Your optimizer must be a new ACSE tool, like `asm` – which can be used as starting point. The tool must read an input text assembly, apply the optimization where possible, and write in a new file the output assembly.

Creating and Applying Patches

The ACSE distribution is already provided as a mercurial repository. See [4] for the basic mercurial commands.

Assuming that your work has been committed with revision X, to generate a patch use the following command:

```
hg diff -r 0 -r X
```

Assuming the patch has been saved in the file `patch.diff`, use the following command to apply it to a freshly unpacked ACSE source tree:

```
hg import patch.diff -m "Applied patch"
```

The submitted project must successfully pass the above process, i.e. before submitting your patch check that it can be applied to a freshly unpacked ACSE source tree including compilation and testing process where applicable.

References

[1] Mercurial. http://mercurial.selenic.com, 2011.

[2] A. Di Biagio and G. Agosta. Advanced Compiler System for Education. http://compilergroup.elet.polimi.it, 2008.

[3] Formal Languages and Compilers Group. Software Compilers. http://compilergroup.elet.polimi.it, 2010.

[4] J. Spolsky. Hg Init: a Mercurial Tutorial. http://hginit.com, 2011.