

# IN4303 Practical work Assignment

Gertjan Halkes

August 16, 2010

## 1 Introduction

The practical work for the Compiler Construction course consists of writing a peephole optimizer for x86 assembly. The main purposes of this course are to get acquainted with compiler writing tools, i.e. flex and bison, and to apply some of the compiler construction theory.

## 2 Assignments

The practical work is split into two assignments. The first is on creating a parser for assembly language, the second is on optimization. Carefully read the assignments before you start programming.

### 2.1 Assignment 1

The first assignment is intended to get acquainted with the tools, and to create a front end for the peephole optimizer that is sufficiently detailed to allow easy construction of the peephole optimizer in Assignment 2. In the first assignment you will make a syntax translator for x86 assembly. The input to your program will be an assembly program in AT&T syntax, while the output will be that program in Intel syntax. For example:

```
popl %eax          pop  eax
popl %ecx          pop  ecx
cld               cld
idiv %ecx          idiv ecx
movl %eax, -4(%ebp) → mov  dword ptr [ebp - 4], eax
imul $2, %eax     imul eax, 2
movb string + 7, %dl  mov  dl, byte ptr [string + 7]
movl $string, %ebx  mov  ebx, offset string
```

As the Intel x86 instruction set is very large, this assignment is limited to the integer and control flow instructions. The restrictions on the instruction set are there to allow a more generic implementation without excessive numbers of exception cases. However, some exceptions are still present in the remaining instruction set. See below for a more detailed overview of the excluded instructions.

For this exercise the GNU assembler is used. The GNU assembler used on the lab PCs accepts both AT&T syntax (the default) and Intel syntax. To accept Intel syntax it requires the directive `.intel_syntax noprefix`. A description of the input syntax, including the differences between AT&T and Intel syntax, is provided in a separate document (see Section 4).

You may make the following simplifying assumptions about the input:

- The input is valid assembly. However, when your program encounters unexpected input, it should fail gracefully with an error message rather than crash.
- A size suffix is appended to all instructions which reference a memory location, even when the GNU assembler does not require it. Therefore, there is no need to analyze the arguments of an operation to determine the size of the memory reference.

- The input will not contain long versions of the `call`, `jmp` and `ret` instructions. These do not have to be supported by your translator.
- Instruction prefixes (e.g. `rep`) do not have to be accepted by the translator.
- Floating point instructions and SIMD instructions such as MMX and SSE instructions are excluded.
- Instructions concerning memory protection and processor management (e.g. `lgdt`) are excluded.
- Instructions for manipulating segment registers (e.g. `lds`) are excluded. This does not exclude the generic instructions such as `mov`, with segment registers as arguments.
- Directives will not introduce new symbols nor will they change other processing requirements (e.g. no `.macro` directives will be used).
- Local labels and Dollar local labels are not used, nor is the symbol setting using `=` and `==`.

To ease implementation, a list of instructions which have an optional operand-size suffix is provided, as well as a list of instructions for which the ‘w’ suffix should be retained to ensure proper translation by the assembler. Note that the addition of the ‘w’ suffix is specific to the GNU assembler and not part of the normal Intel syntax.

**Warning:** The GNU assembler is more permissive than mandated by the specification. For example, it allows spaces between the name of a label and the following colon character, while the specification says this is not allowed. We will test your submission according to the specification.

## Requirements

Your translator must meet the following requirements:

- Your translator must be written in C, and should use flex and bison to parse the input.
- The executable for the translator must be named `translate`.
- The translator should build an AST from the input, before writing the output. The data structures used to build the AST must be defined in a file named `ast.h`.
- The input for your translator must be read from `stdin`, and the output must be written to `stdout`. Error messages must be written to `stderr`. Running your program with `./translate <in.s >out.s` should result in a file named `out.s` which is the Intel syntax version of the input file `in.s`
- Your program should have an exit code of 0 on success and anything else on failure.
- Running `make clean all` in the directory with your source should build the executable.

Along with the list of suffixed instructions, a Makefile and some utility code is provided which should help you get started (see Section 4).

## Documentation

This assignment should be accompanied by separate documentation, in PDF format, named `documentation1.pdf`. The documentation should reflect on your design choices and link them to the general theory of compiler construction as provided by the book and lectures. Nevertheless, the documentation must be to the point and concise, so limit your writing to around 4 pages. The documentation will also be graded.

## Tips

- Write clean C code, with appropriate comments. Your code will be graded for quality as well as its functionality.
- Make sure you initialise all your variables **and** malloc'ed memory, as the C runtime will otherwise leave you random values.
- After running the assembler, you can use `objdump -d` on the output to disassemble the result. By comparing the disassembled versions of the original and the translation you can verify that you made a correct translation. The script `check1.sh` included with the utility code automates this process.
- After running the assembler on a test file in AT&T syntax, you can run `objdump -d -M intel` on an assembled file to see what the Intel syntax equivalent is.
- Keep in mind that in the next assignment you will have to build on the code you write for this assignment to create a peephole optimizer. However, the examples provided for assignment 2 do **not** fully cover the requirements for assignment 1.

## 2.2 Assignment 2

In this second assignment, you have to apply peephole optimization to assembly input. The input for your peephole optimizer will be generated by a compiler that uses the Intel processor as a stack machine. All calculations and comparisons are performed by first pushing the operands onto the stack, then popping the operands, executing the instruction, and finally pushing back the result onto the stack. This of course leads to suboptimal code, which lends itself well to peephole optimization. As an example, consider the code generated for the statement `a := a + 7;` (comments and empty lines added for clarity):

```
pushl    -4(%ebp)      # Push the value of 'a' onto the stack

pushl    $7            # Push the immediate value 7 onto the stack

popl     %ecx          # Retrieve the arguments for the addition operator
popl     %eax
addl    %ecx, %eax     # Perform addition ...
pushl    %eax          # ... and push the result back onto the stack

leal    -4(%ebp), %eax # Get the address of 'a' ...
pushl    %eax          # ... and push it onto the stack

popl     %eax          # Retrieve the destination address ...
popl     (%eax)        # ... and use a clever pop instruction to store
                        # the addition result at the destination
```

The empty lines delineate the separate blocks generated by the compiler. One obvious optimization is to remove the sequence `pushl %eax; popl %eax` near the end, because it has no effect. Examples of the generated assembly code are provided to help you find optimizations you can apply.

You may make the following simplifying assumptions about the input:

- After a register has been pushed onto the stack, its contents will not be used again.
- The source operand (*not* the destination operand) of an arithmetic or logic instruction will not be used again after the arithmetic instruction if it is a register.
- Only `cmp` and test instructions are used to set flags for conditional operations.
- The compiler follows the C calling convention:
  - Arguments are passed on the stack (in reverse order, so the first argument lies on top).

- A function result is passed back in the accumulator register (`%eax`).
- The `%eax`, `%ecx`, and `%edx` registers may be used as scratch registers; all other registers are callee-saved.

A peephole optimizer can optimize both for speed and for size. Most optimizations will do both. Because it is very hard to quantify the speed effect of most optimizations, only the size optimizations are used in grading. Grading of the assignment will be done by comparing the achieved size optimization with a benchmark, together with evaluation of the correctness of the optimizations.

## Requirements

In addition to the requirements mentioned in Assignment 1, your submission must meet the following requirements:

- Except for bugs explicitly marked for correction in assignment 2, you do not have to correct bugs in assignment 1.
- You must implement at least 6 working optimizations, and you should at least achieve the following sizes for the supplied test-cases:

Test	Size	Original Size
primes	499	(564)
golf	1050	(1144)
tree	1158	(1244)

**Note:** these are the minimums to be considered for grading, not for a passing grade. Also, at least 6 optimizations must be applicable to the example code.

- For each optimization you apply, you must provide documentation in the form:

```
mov Y, X    →  add Z, X
add Z, Y
```

Conditions:<sup>1</sup>

- X and Z are not both memory references (add does not allow this).
- Y is not a memory reference (otherwise the memory contents would be different).

This documentation must be written with the provided `documentation2.tex` L<sup>A</sup>T<sub>E</sub>X template; `pdflatex` can be used to generate a PDF file.

- In your code, each optimization should be implemented in such a way that it can be disabled by commenting out a single line of code.
- Your optimizations should consider a fixed number of instructions. Optimizations that consider a dynamic range of instructions will not be accepted.

## Tips

- Use the x86 instruction reference (see Section 4) to find out what instructions exist and how many bytes they use.

---

<sup>1</sup>This example uses the assumption that the source operand of an arithmetic instruction is not used again.

- Always verify that the optimized program executes correctly. See the file README provided with the examples on how to compile the examples. The script `check2.sh` included with the utility code automates the verification process. The `check2.sh` script also shows the result of your optimizations.
- Make sure that your optimizations are valid, ie. that the values in the registers after the replacement sequence are the same as in the original sequence, to the extent required by the assignment description.
- The examples were created with Panoramix, which compiles the Asterix language. If you want to, you can create more test cases using this tool. See Section 4 for more information.

### 3 General remarks

#### 3.1 Deadlines

The assignments should be turned in no later than 08:59 on the dates listed below:

Assignment	Date
1	October 22 <sup>nd</sup> , 2010
2	January 14 <sup>th</sup> , 2011

For every day that you hand in your assignment late we will subtract 1 point.

#### 3.2 Instructors

The following instructors will be assisting and grading the practical work.

Person	Email address	Room
Alexander van Amesfoort	A.S.vanAmesfoort@tudelft.nl	HB 09.320
Venkat Iyer	V.G.Iyer@tudelft.nl	HB 09.070

Contact Venkat Iyer by email for assistance.

#### 3.3 Grading

Grading is done per assignment. According to the following table, points can be obtained for *test results*, *source code* and *documentation*. The total amounts to a 64 points maximum; 26 points are reserved for the final written exam.

Assignment	Test results	Source code	Documentation
1	20	8	8
2	20	8	-

#### 3.4 How to submit your work

- In your own source directory, type
 

```
make (1|2)
```

 which will pack the complete compiler and documentation for assignment 1 or 2 into a file named *submit.tgz*. Warning: if you not only modified files, but also added new ones, be sure to add them at the appropriate place in the Makefile.
- Test if you can unpack and compile your compiler with
 

```
tar xfz submit.tgz
make
```

 (Go to some temporary directory before unpacking the compiler.)

- Upload the *submit.tgz* file to the CPM system:
  - login to CPM (<http://cpm.ewi.tudelft.nl/>) using your NetID and password from Blackboard.
  - select the 'compiler construction' course.
  - click on the 'right' entry in the overview table, which will pop-up a browse menu for submitting a deliverable.
  - click on the 'notification' button if you want to receive e-mails about the progress of your submission (conformity test, grading, etc.).
  - if your submission does not pass the automatic conformity test by the CPM system (run every 10 minutes), login to CPM again and check the error report, fix the problem, and resubmit. We will not consider your work submitted until the status is set to ScriptApproved.

Do not submit unless you managed to unpack and compile your own sources.

It is useless to turn in your solution far in advance of a deadline; your compiler will be tested and examined (shortly) after the deadline.

### 3.5 Important announcements

Frequently check the blackboard (in4303); announcements and updates to this course will be posted there.

## 4 Documentation and Downloads

- Concise assembly syntax <http://www.st.ewi.tudelft.nl/~koen/in4303/current/assembly-grammar.pdf>.
- Short x86 instruction reference <http://courses.ece.uiuc.edu/ece390/books/labmanual/inst-ref.html>.
- Basic empty project with utility code and code examples [http://www.st.ewi.tudelft.nl/~koen/in4303/current/compiler\\_construction.tgz](http://www.st.ewi.tudelft.nl/~koen/in4303/current/compiler_construction.tgz).
- The Panoramix compiler for the Asterix language <http://www.st.ewi.tudelft.nl/~koen/compilerbouw/www/panoramix.html>.
- The Asterix language specification <http://www.st.ewi.tudelft.nl/~koen/compilerbouw/www/refman/refman.html>.