Compiler Construction Project - Assignment 4: Loop Statements

November 11, 2022

1 Introduction

In this assignment, you will add a few features so that your compiler accepts a basic language. You will implement `while`-loops and two new operators: division (`/`) and modulo (`%`).

The assignment is described in Section 2. Section 4 lists what should be handed in to complete this assignment, the deadline of handing in the result of this first practical is set to the 25th of November at 23:59. Also, make sure to update your final report with the new changes. In order to run the framework for assignment 4 you have to run the following command in the build directory: `meson -Dwith-assignment=4 --reconfigure`

2 Assignment

In this assignment you are expected to support while-loops and two additional operators: integer division and the modulo operator. This will allow you to write the *greatest common divisor* (gcd) algorithm in two different ways, one of which including the modulo operator. After implementing the assignment, you will be able to compile the code in Listing 1.

```
int main() {
    int i;
    int j;
    i = 0;
    while (i < 5)
        i = i + 1;
    j = i % 2;
    print j;
}
```

Listing 1: Example of a program to be supported

2.1 While-loops

While-loops are very similar to if-statements. You will implement a while-loop in four steps:

1. Add the `while` keyword to the lexer.
2. Add the `while_statement` to the grammar. A `while_statement` consists of a relational expression (i.e. The condition for the while-loop to continue) and a statement.
3. In the intermediate code generator, add function that visits while-statements. For a binary node that contains a while-statement `WHILE COND STMT`, you should produce the following intermediate code:

```
1     LABEL1
2     JUMP_IF_NOT COND LABEL2
3     STMT
4     GOTO LABEL1
5     LABEL2
```

A while-loop first checks the condition, and if it is not true, skips the statement entirely (Line 2). Else, you execute the statement (Line 3), and then jump to the condition check again (Line 4).

2.2 Division / and modulo % operators

First, some terminology. Let's consider the following operation:
9 / 2
Here:
9 is the *dividend*.
2 is the *divisor*.
The *quotient* is 4 (because 9 / 2 = 4.5, but decimals are discarded in integer math).
The *remainder* is 1 (because 9 % 2 = 1).

At the syntax and intermediate code-level, division and modulo are simply binary operators. However, at the assembly level, they are a bit more complex than the other arithmetic operations.
The `div` (unsigned) and `idiv` (signed) assembly instructions calculate the quotient (= division) and the remainder (= modulo) at the same time. `div` and `idiv` take a single argument: The divisor. The divisor may either be a register or a memory location, but not an immediate. The dividend is given in a set register (see Table 1).

Operand size	**Before div/idiv**	**Dividend source**	**Divisor source**	**Quotient destination**	**Remainder destination**
Byte		AX	8-bit register or memory location	AL	AH
Word	Zero/sign extend into DX	AX	16-bit register or memory location	AX	DX
DoubleWord	Zero/sign extend into EDX	EAX	32-bit register or memory location	EAX	EDX
QuadWord	Zero/sign extend into RDX	RAX	64-bit register or memory location	RAX	RDX

Table 1: Overview of operand sources and result destinations of the div/idiv instructions, per operand size.

2.2.1 Zero/sign-extending into a register

As per Table 1, all operand sizes larger than Byte require you to zero/sign-extend into a certain register. This is because for those operand sizes, the dividend source is actually given in *two* registers: One high, one low. Together they form a single number. However, for simplicity, we will only handle division/modulo where both the dividend and the divisor are similar-sized, and therefore only use the low register. This means the high register must be all 0 (`div`) or all equal to the sign-bit (`idiv`).
To zero-extend into a register, say `DX`, simply `xor` it with itself:

```
    xor %dx, %dx
```

To easily sign-extend into a register, use the `cwd`, `cld`, and `cqo` instructions for Word, DoubleWord and QuadWord respectively:

```
    cwd     ; sign-extend AX into DX
    cdl     ; sign-extend EAX into EDX
    cqo     ; sign-extend RAX into RDX
```

3 Report

We require you to provide a `README` file including any design choices you have made during this assignment. This also includes a summary of the functionalities of each file you have created or modified. Furthermore, the `README` includes a paragraph on what you have learned from this assignment, what the most challenging parts were and how you dealt with these challenges. Include the two different implementations of the *greatest common divisor* algorithm with your `README` file and the output assembly code your compiler generates. Is either of the two implementations more efficient?

4 Submission

This submission is handled through Brightspace. Go to the course website, and hand in assignment 4.
In Brightspace, hand in

1. a tarball:
   - named `assXgroupY`, with X the assignment number and Y your group number. Name your main folder in the same way (so do not leave it as `assX`).
   - with all source code (not only the modified files).
   - **without** the build directory. In general: do not hand in larger submissions than required.
2. the `README` reporting on the assignment.

Failing to adhere to these instructions will result in a penalty to your grade. Please also be aware of the fact that we will not grade work that does not compile. Warnings will result in a penalty in your grade, even if you get warnings when building the framework as is (e.g. unused variable/function warnings).
We use `huisuil` as a reference. So, make sure that your submission compiles on `huisuil`!

You will be graded on the quality of your `README` file, the layout of the code including the modularity and quality of comments, and the functionality of your implementation.