CoSy Compiler (2 weeks)

Motivation and introduction

Until now, we have written the assembly code by hand. Now we will use a compiler that is dedicated to our individual *ASIP Meister* CPUs. With this compiler, we will compile a C-code application and simulate the result in *ModelSim*. Afterwards we will implement some custom instructions for this application to speed up the execution time. Moreover, even when the compiler uses the new instructions, they might not be used in all optimization levels. For that, we will also introduce the *CoSy* feature *SINAS*, which is used to add inline assembly to the application. By using inline assembly, you can force the usage of custom instructions or you can optimize bigger blocks (e.g. application hot spots) in hand written assembler. The information about creating and using the compiler can be found in Chapter 8 of the laboratory script.

Exercises

1) Creating the first CoSy Compiler
The *CoSy* Compiler System is a retargetable compiler generator. This means, that the *CoSy* Compiler System gets an architecture description as input and it generates a compiler for this architecture as output. *ASIP Meister* is able to generate the description files that the *CoSy* Compiler System needs as input automatically. Therefore, you can automatically create a compiler for your individual processor! To get an idea of how retargetable compilers are working, read Chapter 8.1 from the laboratory script.
To create the compiler of the basis CPU, you need to type makeCoSy after checking your evn setting to make sure which CPU version will be targeted.

2) Compiling and simulating the application
First you have to compile the application by the gcc compiler to compare later the results from *dlxsim* and *ModelSim* with a *gcc*-compiled version. For *gcc* you can forward the printed output to a file, e.g. “a.out > output_gcc.txt” (‘a.out’ is the default name of the binary that is created when you compile “gcc arrayLoop.c” and ‘output_gcc.txt’ then contains the printed array).

For compiling arrayloop.c using *dlxsim* and *ModelSim*, you need to provide the required libraries, i.e. *lib_lcd_dlxsim* (also for *ModelSim*), *loadStoreByte*, and *string*. Chapter 8.5 describes how to provide these libraries. Also these libraries can be found in:
(~asip00/ASIPMeisterProjects/TEMPLATE_PROJECT/Applications/TestPrint/)

After providing the required libraries, you can compile the application by typing “make sim”
After compiling, simulate the application in *dlxsim* and *ModelSim* and compare whether the printed results are the same compared to a *gcc*-compiled version. The *gcc* version will print the arrays on the screen and *dlxsim* and *ModelSim* will print them to a ‘virtual’ lcd.

For *dlxsim* you can forward the LCD output to a file, using the “-lf” parameter, e.g. make dlxsim DLXSIM_PARAM="-da0 -lfoutput_dlxsim.txt” writes to the file ‘output_dlxsim.txt’.
Note: from this session you will start to use another version of DLX simulator instead of the old

one that used to be run. This new version supports the new custom instructions that required later to be implemented. To this end, you have to adjust the env_setting as follows:
"export DLXSIM_DIR=/Software/epp/dlxsim"

ModelSim automatically writes to the file 'lcd.out'. To compare, whether the files are identical, you can use command-line tools like "diff output_gcc.txt output_ModelSim.txt" or graphical tools like 'kompare' or 'kdiff3'.

3) Extending the CPU with a custom instruction

1. Create a new *ASIP Meister* Project "dlx_avg" from your old project "dlx_basis" (**not** dlx_bgeu) (do not forget to adjust the *env_settings*). In your new CPU implement the new instruction `"avg rd, rs0, rs1"` as it is used in the application. Test the new instruction with a small assembly code in *ModelSim*.

2. In your new CPU implement the new instructions `"swap rd, rs"` and `minmax rdMin, rdMax, rs0,` rs1 as they are used in the application. This new instruction "`minmax"` shall compute both the minimum and the maximum of two inputs *rs0* and *rs1* and write them simultaneously to two registers (*rdMin* and *rdMax*).
   Hint:
   - Do not implement the *swap* instruction as it is written in the C-code. Think what this instruction is doing and implement it without any shifts! Test the new instructions with a small assembly code in *ModelSim*.
   - As we do not create a new compiler (in fact from now on we will not create any new compiler), the *Behavior* section in ASIP Meister is not too important, as this part is only used for creating the compiler. Just write there something that is syntactically correct.

3. After testing the new instructions with a small assembly code, use *SINAS* in the application for using the new instructions.

After modifying the application code by the *SINAS* stuffs for a particular custom instruction, compile the application, make sure that the result is still correct (*diff* the *ModelSim* output with gcc-compiled version) and find out, whether the new instructions have been used or not.

HINT: Do not add the *SINAS* stuffs for all the custom instructions together as a one step. This may complicate figuring out where the problem later.

Now you have to determine the number of cycles for executing the application to compute the speedup against the old CPU with the old compiler. To determine the number of cycles you have to remove (i.e. comment) the loop for printing the results! Otherwise, this loop is the dominating hot spot and you will not notice a significant speedup when using the new assembly instructions! Furthermore, the amount of NOPs is limiting the noticeable speedup for the *ModelSim* simulation. Typically, the CPU would provide a Data Forwarding Unit (though ASIP Meister does not provide such a module) and thus no NOPs (except in delay slots) would be required. To mimic this behavior, configure the parameter "ADD_NOPS" in the mkimgSettings to "0" and simulate the application with the provided version of *dlxsim*. Instructions like *avg* are already available in a new dlxsim version (adjust your env_settings to use this dlxsim version "Software/epp/dlxsim"). Repeat this benchmark for all compiler optimization-levels (O0, O1, …, O4; e.g. `"../mkimg arrayloop.c -O3"`) for *ModelSim* (ADD NOPs: '3') and *dlxsim* (ADD NOPs: '0').

Note: you have to check the generated assembly code to be sure that the new custom instructions are used in the code.

Fill out the following table and mail it to asip00@ira.uka.de.

Optimization Level		**Cycle count with dlx_basis**	**Cycle count with dlx_avg**	**Speedup (basis / avg)**
O0	ModelSim			
	dlxsim			
O1	ModelSim			
	dlxsim			
O2	ModelSim			
	dlxsim			
O3	ModelSim			
	dlxsim			
O4	ModelSim			
	dlxsim			