CoSy Compiler

Motivation and introduction

This session consists of **two** weeks. Until now, we have written the assembly code by hand. Now we will create 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. For every part, that starts like "a)", "b)" … you have to mail the answers and asked files/tables with a CC to your group members to **asip00@ira.uka.de** and use the topic "asipXX-Session2", with XX replaced by your group number.

Readings for the next session: All the Chapters, especially 4 & 8, ASIPMeister Tutorial & Manual

Exercises

1) Creating the First CoSy Compiler

1. 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.
2. Create a project directory and setup your project directory by adjusting "*env_settings*". For this session you will use another version of DLX simulator instead of the previous one. The new version supports the new custom instructions like "*avg*", "*swap*" and "*minmax*" that are required in this session, therefore set "*export DLXSIM_DIR=/Software/epp/dlxsim*" in "*env_settings*".
3. In the project directory, create a compiler for the basis CPU using "*makeCoSy*" (the CPU without "*bgeu*" if you are not sure about your CPU version, we recommend you to copy a fresh CPU version from "*/home/asip00/Session3/dlx_basis.pdb*"). Read Chapter 8.2 from the Laboratory Script to know how to create the compiler.
4. If compiler generation is successful, a compiler binary will be generated in your project directory. In case of any errors use "*contCoSy*" to debug the errors and manually correct these errors to generate the compiler.

2) Compiling and Simulating the Application

5. Create a subdirectory in your "*Applications*" directory and copy "*/home/asip00/Session7/arrayloop.c*" to this subdirectory. Also copy the required "*Makefile*".
6. First you have to compile the application using gcc compiler to compare with the later results from dlxsim and ModelSim. 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" while '*output_gcc.txt*' then contains the printed array).

7. The usage of the compiler is explained in Chapter 8.3 of the Laboratory Script. After reading this chapter, compile arrayloop.c using "make sim". But, for compiling it, you first 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.
8. 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 output to the file "*output_dsim.txt*". ModelSim automatically writes to the file "*lcd.out*".
9. To compare, whether the files generated from gcc, dlxsim & ModelSim 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

10. 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. You can take help from "*asm.c*" in "*/Software/epp/dlxsim*" regarding the instruction type and their op-codes.
11. Test the new instruction with a small assembly code in ModelSim to verify its functionality.
12. 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*" computes 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.

13. After testing the new instruction with a small assembly code, use *SINAS* in the application for using the new instructions.
    Hint: Do not add the SINAS stuff for all the custom instructions together as a one step. This may complicate figuring out the problem later.
14. 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.
    Note: you have to check the generated assembly code to be sure that the new custom instructions are used in the code.
15. 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. <u>To determine the number of cycles you have to remove (i.e. comment out) the loop for printing the results</u>! Otherwise, this loop is the dominating hotspot 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 "*NUMBER_OF_HW_NOPS=0*" for the "*Makefile*" and simulate the application with the provided version of dlxsim e.g. "*make dlxsim COSY_PARAM=-O0 NUMBER_OF_HW_NOPS=0*".

16. Simulate the application with ModelSim, using "*NUMBER_OF_HW_NOPS=3*" like: "*make sim COSY_PARAM=-O0 NUMBER_OF_HW_NOPS=3*".
17. Repeat this benchmarking for all compiler optimization-levels like O0, O1, O2, O3 and O4 for both dlxsim and ModelSim. Always use NUMBER_OF_HW_NOPS=0; for dlxsim and NUMBER_OF_HW_NOPS=3 for ModelSim.

Benchmark Table:

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			