Adding Custom Instructions

2 Weeks

Motivation and introduction

In this session, we will implement some custom instructions for an 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 feature, 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 previous session. 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 **sajjad.hussain@kit.edu** and use the topic "asipXX-Session4", with XX replaced by your group number.

Exercises

1) Preparing the project

1. Create a project directory and setup your project directory by adjusting "*env_settings*".
2. In the project directory, create a new project and generate compiler for the basis CPU, we recommend you to copy a fresh CPU version from "*/home/asip00/Sessions/Session3/browstd32.pdb*"). If compiler generation is successful, a compiler binary will be generated in your project directory. [You can also start from the last session's project and just create sub-directories in "Application" directory. This will save time.

2) Compiling and Simulating the Application

3. Create a subdirectory in your "*Applications*" directory and copy "*/home/asip00/Sessions/Session4/arrayloop.c*" to this subdirectory. Also, copy the required "*Makefile*".
4. 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). To compile with GCC, comment the line "*#define ASIP*".
5. 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". However, 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.
6. 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 -lf**output_dlxsim.txt***" writes output to the file "*output_dsim.txt*". ModelSim automatically writes to the file "*lcd.out*".
7. 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*".
   a) How many cycles do you need for execution in dlxsim and ModelSim?

3) Adding a new instruction to *dlxsim* Simulator

8. Now we start adding new instructions to our processor to speed up the execution. These new instructions are "*avg rd, rs0, rs1*", "*swap rd, rs*" and "*minmax rdMin, rdMax, rs0, rs1*". First, implement the new instructions into dlxsim, as explained in the Chapters 3.2.2 and 3.2.3 of the Laboratory Script. Use the instruction format and opcodes as below:
```
OpcodeInfo opcodes[]
//name       class          op      mask      other  flags  rangeMask
{"swap",    ARITH_2PARAM, 0x541,  0x1ffff,   0x20,  0 ,    0xffff8000 },
{"avg",     ARITH_3PARAM, 0x6c1,  0x1ffff,   0x20,  0 ,    0xffff8000 },
{"minmax", ARITH_4PARAM, 0x981,  0xfff,     0x20,  0 ,    0xffff8000 },
{"rot",     ARITH_3PARAM, 0x8,    0x3f,      0,  CHECK_LAST|IMMEDIATE_REQ,
0xffff8000 },
```
9. Therefore, you have to copy dlxsim to your local home (to be able to modify it) and you have to configure the "*env_settings*" to use your local dlxsim (see Figure 2-5 in the Laboratory Script).
10. Write a small assembly code to test your new instructions in dlxsim. You can use the Session1 assembly language program as the reference.

4) Extending the CPU with a custom instruction

11. Create a new *ASIP Meister* Project "*browstd32avg*" from your old project "*browstd32.pdb*" (do not forget to adjust the "*env_settings*"). In your new CPU, implement the new instruction "*avg rd, rs0, rs1*", "*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: You can use the opcode and instruction formats as indicated in the figure below.
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.
12. Generate the VHDL Files.
13. Please remember that for new custom instructions defined in ASIPmeister to be used automatically with C Compiler, you have to implement relevant "**CKF Prototype**" in ASIPmeister. Following instructions at section 4.11.C in ASIPmeister tutorial and 11.2 in ASIPmeister user manual.
14. Generate GNU Tools for your new processor.

5) Compiling and Simulating the Application with custom instructions

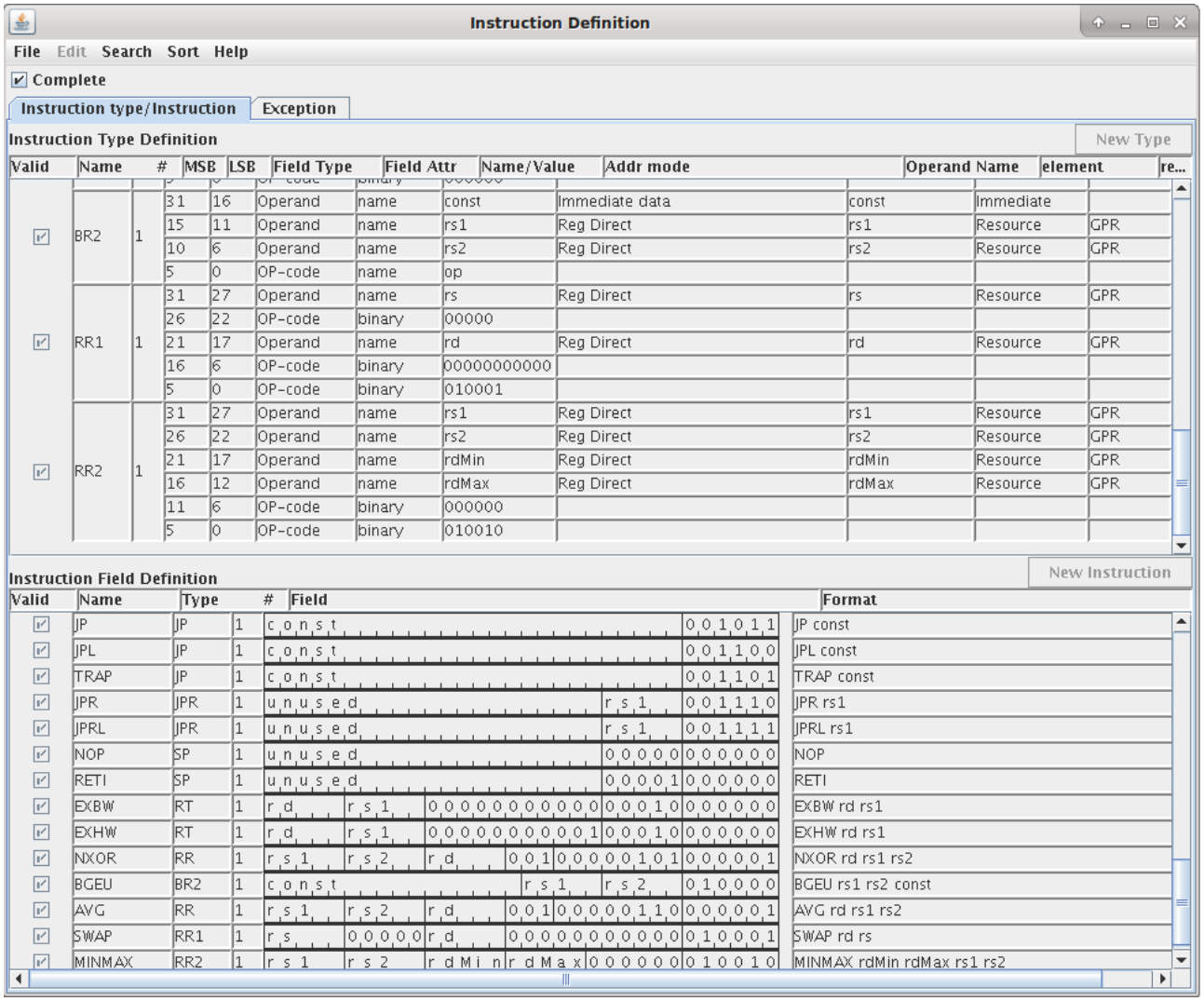
15. You can test your custom instructions with small assembly programs.
16. After testing the new instruction with a small assembly code, use *inline assembly* in the application "*arrayloop.c*" for using the new instructions, see Chapter 8.2.3 in the Laboratory Script. Rename it to "*avg*.c".
17. After modifying the application code by the *inline assembly* stuffs for a particular custom instruction, compile the application using "*make sim*", 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.
18. 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!

19. Simulate the application with ModelSim.

b) How many cycles do you need for execution in dlxsim and ModelSim? Attach assembly programs you used to test custom instructions with this mail.

c) What is the speedup of "*avg.c*" compared to *"arrayloop.c*" (i.e. #Cycles without custom instructions / #Cycles with custom instructions)?



Next Session: BubbleSort – Simulations and Optimization

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