Bubble Sort – Simulation & Optimisation

1 Weeks

Motivation and introduction

In this exercise, we will start applying the whole design flow to a simple example. We will not finish all the steps in this session, so the whole task is separated into multiple sessions (Session 5, 6 & 7). The example application is the *BubbleSort* algorithm. You will receive the C code for *BubbleSort* to be simulated. Afterwards you will implement a new instruction, which will help you to speed up the execution time. In a later session, we will estimate and measure what we have to pay for this speedup, in terms of chip area, power and energy consumption and we will compare the basis processor with the modified/extended one. To make sure, that everyone starts with the same CPU, we are providing a *browstd32.pdb* CPU with the add-instructions already included. Use this CPU instead of the one, which you have created in the last session. This will make sure that you have a functionally same starting/basis CPU and that the results (area, power, performance etc.) between the groups are comparable. Finally, in later session, we will run both processors on the FPGA prototype. For every part, that starts like “a)”, “b)” … you have to mail the answers and asked files with a CC to your group members to **sajjad.hussain@kit.edu** and use the topic “asipXX-Session5”, with XX replaced by your group number.

Exercices

1) BubbleSort_Index

1. Have a look at “*BubbleSort_Index.c”*. Every part, which contains a *printf* function call, is encapsulated with a *“#ifndef ASIP”* directive. The reason is, that the *printf* function usually is resolved to an operating system call (managing the screen and other resources), but for our CPU we don’t have an operating system, thus we ignore the *printf* function for our simulations. For hardware execution in a later session, we will map this call to a UART terminal or LCD. For a *gcc* compiled version the *printf* is a helpful in debugging the output.
2. Look into the implementation of the algorithm. You will need a good knowledge of the algorithm for later optimizations. Compile *“BubbleSort_Index.c”* with “`gcc Inputfilename -o Outputfilename`”, look at the printed output when executing the binary and understand how the algorithm is working by going through the printed output step by step.
   a) How often the code of the inner loop is executed (not only the exchange part, the whole inner loop)? Please do not only answer this question, but go through the output step by step. First, you should look at the code and think about the answer and then you should add a counter to the code to compute the correct result just to make sure, your prediction is correct.

2) BubbleSort_Address

3. To simulate *BubbleSort* with dlxsim and ModelSim it has to be translated from C to assembly, which will be done in exercise 3. To make the translation easier, the “*BubbleSort_Address.c”* has been prepared. Compile “*BubbleSort_Address.c”* with *gcc* as discussed before.
4. First, make sure that the output of the *gcc* compiled versions of “*BubbleSort_Index.c”* and “*BubbleSort_Address.c”* are the same. Then have a more detailed look into the address-version. The main difference between both versions is the way of accessing the array. The index-version uses an indexed access (e.g. array [j+1]). This usually translates into a chain of assembly instructions. First the real address has to be computed and then the value can be loaded. The real address is: “starting address from array” + “size of one array entry” * “index (i.e. j+1)”. In the inner loop of *BubbleSort* we traverse through the array linearly, so we do not have to compute

the real address every time from the scratch, instead we can just update the last computed real address. Two other changes against the index-version are, that every memory access is explicitly written, like “*value_j = *j;*” and the number of memory accesses is optimized as compared to the index-version.

b) How many load- and how many store- instructions are executed for each inner loop (distinguish between when there is exchange and no exchange)? Compare the index-version against the address-version and mention the two main points, why the address-version needs less memory accesses.

3) Compiling the application in dlxsim and ModelSim with basis processor

5. First, create your project directory as in the previous sessions and create a subdirectory for your bubblesort application in “*Applications*” directory. Setup your projects by copying the required files from “*/home/asip00/Sessions/Session04/app.c*” to your project and adjusting “*env_settings*”. This is same as “*BubbleSort_Address.c”* but without printing.
6. Then you have to start ASIPmeister in your project directory and generate hardware and software files for the provided “*browstd32.pdb”*.
7. Simulate the translated “*app.c”* file with dlxsim (issue “*make sim*”, and then “*make dlxsim DLXSIM_PARAM="-fBUILD_SIM/xxx.dlxsim -da0 -pd1"*” where xxx is your application’s subdirectory name).

NOTE: Never name the subdirectory same as the name of the application that you want to compile using “*Makefile*”. This will result in problems for the Makefile script execution.

8. For this version, you can try whether the code is running with the VHDL code of the CPU. Simulate “*app.c”* with ModelSim and compare the number of executed cycles and the resulting array (TestData.OUT) with dlxsim.

a) How many cycles do you need for execution in dlxsim and ModelSim?

4) Bubble Sort – Optimisation: Customizing the basis processor

9. Now we start optimizing our bubblesort application for speed. There might be two options for you.
   Optimisation I). Take “app.s” from last exercise and copy into a new application sub-directory. In the assembly code, look for different possibility to define custom instructions and replace that part of code with the custom instruction in assemble file. OR.
   Optimisation II). You can directly look into the “*app.c*” and define some custom instruction to replace some part of the code with new custom instruction.

5) Adding the new instruction to *ASIPMeister*

10. Create a new project directory inside your *ASIPMeisterProjects* directory for the new CPU and name it “*browstd32bgeu*”. You can use a copy from the *browstd32* CPU project from the last session, but do not forget to adjust the “*env_settings*”. Also, create a subdirectory for your test application in “*Applications*” directory.
11. In your project directory start *ASIPMeister* and add the new instruction to your new CPU. First, define a new instruction format for your instruction if it does match with the existing instruction formats.
12. Use the available opcode.
13. For that, you have to define “CKF Prototype” for each new custom instructions in ASIPmeister, generate GNU tools and use inline assembly in C code.
14. Write a small C or assembly code to test your new instruction.

Hints:

- Remember, that you cannot use a hardware resource twice in the same cycle, e.g. you cannot use the ALU twice in the EXE stage. Additionally, using it in two different pipeline stage significantly complicates the whole CPU design (just think about the required wiring).

- Remember that your new instruction has to support forwarding as well.

15. Generate the hardware and software files from ASIPMeister and simulate the new instruction with ModelSim. Use the small test application that you created to test your dlxsim implementation in the previous exercises for this purpose.
16. If everything is working fine, then simulate the *BubbleSort* C/Assembly code that uses the new instruction in ModelSim.
17. Afterwards use this new instruction in your *BubbleSort* implementation from last Exercise and name the resulting file "*appbgeu.s*" in a separate subdirectory in "*Applications*".
18. Compile "*appbgeu.s*" in its application subdirectory using "*make sim*" and then "*make dlxsim*". Make sure, that the resulting array is still correct. HINT: It is ok if "*make sim"* complains that the assembler was not aware about this new instruction (and thus the binaries for ModelSim/FPGA are not created). We need to change the CPU in ASIP Meister to make the assembler aware of this instruction (next exercise). However, the files for dlxsim are created before the assembler is called.

   d) How many cycles do you need for execution?

   e) What is the speedup compared to *"app.s*" (i.e. #Cycles without bgeu / #Cycles with bgeu)?

Next Session: Bubble Sort - Hardware Implementation
Readings for the next session: Chapters 6