Bubble Sort - Simulation

Motivation and introduction

In this exercise we will start doing the whole design flow with a simple example. We will not finish all the steps in this session, so the whole task is separated into multiple sessions (Session 3, 4, 5 & 6). The sessions in the later weeks will build up on the results from this week. The example application is the *BubbleSort* algorithm. You will receive the C code for *BubbleSort* and you will translate it into DLX assembly. 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 *dlx_basis* 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 **asip00@ira.uka.de** and use the topic “asipXX-Session2”, with XX replaced by your group number.

Readings for the next session: Chapters 3.3.1

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 COSY”* directive. This has been done to help the CoSy compiler, which will be introduced in a later session. 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. 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.

a) 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) Translation

5. First of all, create your project directory as in the previous sessions and also create a subdirectory for your bubblesort application in “*Applications*” directory. Setup your projects by copying the required files from “*/home/asip00/Session03/*” to your project and adjusting “*env_settings*”.
6. Then you have to start ASIPmeister in your project directory and generate hardware and software files for the provided “*dlx_basis.pdb”*.
7. You have to translate *BubbleSort_Address.c* to assembly in the created subdirectory. We have prepared the *“bs_Framework.s”* for you, so that you do not have to take care about the function calling and data memory preparation. Have a look at the framework and understand how the two parameters are transferred to the *_bubbleSort*-method. “*BubbleSort_Address.c”* is written in such a way, that you can translate every single line into 1-3 assembly lines, one after the other. You only have to translate the *_bubbleSort*-method (without the *printf* parts) using the framework. Remember, that for ModelSim every register has to be written like *%r23* and every immediate value (except the offsets in load/store instructions) has to be written like *$42*. For every assembly line you should add the original C code line as comment. Name the resulting file “*bs_NoPipeline.s”*. Use the registers for the same purpose as suggested by the comments in the framework. Remember, that every array entry has the size of 4 bytes. Use a pipeline delay of 1 to simulate this version in dlxsim *(-pd1)*. Make sure that dlxsim computes the same result array as the gcc version. You will have three conditional branches in the assembly code. Realize them in the following scheme: sltu %r1, a, b beqz %r1, label
8. Simulate the translated “*bs_NoPipeline.s”* 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.

9. Make sure, that your translation is also working with ModelSim (some syntax spellings are accepted by dlxsim, but not by the assembler to create the binary for ModelSim).

a) How many cycles do you need for execution in dlxsim? Attach “*bs_NoPipeline.s”* to the mail.

4) Adding the pipeline model

10. To simulate *BubbleSort* with ModelSim, you have to consider the pipeline model of our real hardware. From now on, we will only use the pipelined version! We have prepared the “*bs_-Framework_pipelined.s”* for you. Copy your created *_bubbleSort*-method to this new framework (except the stack initialization and stack clean-up). You have to use this framework to make sure, that the implementations of all asipXX groups are comparable for the later

benchmarks.

11. Change the pipeline delay in dlxsim to 4 (default) and add NOPs to your bubblesort implementation wherever it is necessary. Do not start optimizations now (this will be done in the next session), do not change the order of instructions. Read your code carefully and try to understand the data dependency then add the <u>required number of NOPs</u> between the instructions.

- Don’t swap between the instructions to optimize the code.
- Leave the "compare nop nop nop branch" block as it is, because later it will be replaced by a new instruction called “bgeu”.

12. Name the resulting file “*bs_basis.s”* and compile it in a new application subdirectory using “*make sim*” and then “*make dlxsim*”. Make sure that the resulting array is sorted in dlxsim with “*–pd4*” (default).

NOTE: The “*bs_basis.s*” file must be in a separated subdirectory with a copy of the “*Makefile*” and NOT in the previous subdirectory where the *“bs_NoPipeline.s”* is located. This is because that “*Makefile*” script compiles all the existed files (.c and .s) in the subdirectory and combines all the results together to create one output file.

13. For this version you can try whether the assembly code is running with the VHDL code of the CPU. Create a new ASIPMeister project with the provided CPU “*dlx_basis.pdb*”. The previous CPU (without add) was just meant for testing the tutorial. This new CPU is the basis CPU for all your future projects. Use “*dlx_basis.pdb*” to make the results between the groups comparable. Simulate “*bs_basis.s”* 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? Attach “*bs_basis.s”* to the mail.