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. The session in the next week 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 helps to speed up the execution time. In a later session we will estimate and measure what we have to pay for this speedup, regarding the needed chip area, power and energy consumption and we will compare the old processor with the modified one. To make sure, that everyone starts with the same CPU, we are providing a *dlx_basis* CPU with the add-instructions included. Use this CPU instead of the one, which you have created in the last session. This will make sure that you have a functional starting CPU and that the results (area etc.) between the groups are comparable. Finally we will let both processors run on the FPGA prototype. For every part, that starts like “a)”, “b)”, … you have to write an answer and mail it with a CC to your group members to asip00@ira.uka.de and use the topic “asipXX-Session3”, with XX replaced by your individual number.

Exercises

1) BubbleSort_Index

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 resolves 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 debugging output. 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

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. 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 go linear through the array, so we do not have to compute the real address every time from the scratch, but we can just update the last computed real address. Two other changes against the index-version are, that every memory access is explicit written, like “value_j = *j;” and the number of memory accesses is optimized compared to the index-version.

a) How many load- and how many store- instructions are executed for each inner loop (distinguish between 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

Translate *BubbleSort_Address.c* to assembly. 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. Translate the *bubbleSort*-method (without the *printf* parts) into the framework and simulate it with *dlxsim* (Make sim, then Make dlxsim). Therefore create a new subdirectory in your *Applications*-directory.

NOTE: Never call the subdirectory as the name of the application that you want to compile using "Make". This will result in problems for the script.

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*). Remember, that for *ModelSim* every register has to be written like ***%r23*** and every immediate (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 the comment in the framework suggests. 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
```

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

4) Adding the pipeline model

To simulate *BubbleSort* with *ModelSim*, we 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 the asipXX groups are comparable for later benchmarks. Change the pipeline delay in dlxsim to 4 (default) and add NOPs to your bubblesort implementation wherever they are 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 required number of NOPs between the instructions

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

Name the resulting file *bs_basis.s*. Make sure that the resulting array is sorted in dlxsim with –pd4 (default). For this version we 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.

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.

In the new folder, you can do "Make sim" then "Make dlxsim" to check you code if it works fine.
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 it 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? Attach *bs_basis.s* to the mail.