KIT-Fakultät für Informatik
Institut für Technische Informatik

Session:6 - Bubble Sort – Simulation & Optimisation

Customized Embedded Processor Design

Application Specific Instruction-Set Processors- ASIP
Lab (Prakitikum)

Responsible/Author:
MSc. Sajjad Hussain

Supervisors:
MSc. Sajjad Hussain, Dr.-Ing. Lars Bauer, Prof. Dr.-Ing. Jörg Henkel

Chair of Embedded Systems,
Building 07.21, Haid-und-Neu-Str. 7,
76131 Karlsruhe, Germany.

August 7, 2022

Bubble Sort – Simulation & Optimisation

1 Week

Motivation and introduction

In this session, we will start applying the whole design flow to a *BubbleSort* algorithm. You will receive the C code for *BubbleSort* to be simulated. Afterwards you will optimize its performance by adding new instruction(s). In a later session, we will estimate what we have paid for this speedup; in terms of chip area, power and energy consumption, i.e. we will compare the basis processor with the modified/extended one. Finally, in a later session, we will implement both processors on the FPGA board. For every part, that starts like "a)", "b)" ... you have to mail the answers and asked files to **sajjad.hussain@kit.edu** and use the topic "asipXX-Session5", with XX replaced by your group number.

Exercices

1. **BubbleSort Algorithm**

   1.1 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.

   1.2 Look at the implementation of the algorithm. You will need a good knowledge of the algorithm for later optimizations. Compile *"BubbleSort_Index.c"* with "*gcc BubbleSort_Index.c –o BubbleSort_Index*", look at the printed output when executing the binary and understand how the algorithm is working by going through the printed output gradually.

   (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 also 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.

   1.3 To simulate *BubbleSort* with *dlxsim* and *ModelSim* it has to be translated from C to assembly, which will be done in the later exercises. To make the translation easier, the "*BubbleSort_Address.c*" has been prepared. Compile "*BubbleSort_Address.c*" with *gcc* as discussed before.

   1.4 First, make sure that the output of the *gcc* compiled versions of "*BubbleSort_Index.c*" and "*BubbleSort_Address.c*" is 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.

2. **Preparing your project**

2.1 You can use the same project as in the first session, and just create a separate application subdirectory for each example. You can start a fresh project as in the Session 1, but this would be time consuming.

2.2 For the C application, you have to create subdirectory in the "*Application*" directory (e.g. "sorting"), and copy your application from "*/home/asip00/Sessions/Session6/bubble.c*" to here.

2.3 Copy a "*Makefile*" file from the "*TestPrint*" application subdirectory to each application subdirectory.

2.4 Set proper parameters and settings in "*env_settings*.

2.5 For these exercises, we will be using pipeline forwarding (-pf1) option which is the default one.

2.6 Make sure that you already have VHDL files and GNU tools in your project's meister directory.

3. **Compiling the application in dlxsim and ModelSim with basis processor**

3.1 Now, compile "bubble.c" using "make sim".

3.2 After compiling, simulate the application in *dlxsim* and *ModelSim* and compare whether the printed results are the same compared to a *gcc*-compiled version. For dlxsim, you can use the command "get _array 20d", it should give you the sorted array. For ModelSim, you can see the sorted array in the TestData.OUT file.

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

4. **Bubble Sort – Optimisation: Customizing the basis processor**

4.1 Create a project directory for this session by copying the directory "*/home/asip00/epp/ASIP-MeisterProjects/TEMPLATE_PROJECT/*" and renaming it (e.g. *brownieOPT*).

4.2 Now we start optimizing our bubblesort application for speed. There might be two options for you.

i. Create a new application sub-directory (e.g. "sortingOptS") and copy "sorting/BUILD_SIM/bub
from the last exercise to this directory and start optimizing the code. 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. If you start with assembly file, sometime it gives errors for labels that starts with dot. Change it to dashes. like .L6 to _L6.

ii. Create a new application sub-directory (e.g. "sortingOptC") and copy "bubble.c" to this directory and start optimizing the code. You can directly look into the "bubble.*c*" and define some custom instruction to replace some part of the code with new custom instruction.

4.3 However, before optimizing, create a simple C or assembly file to test different custom instruction (e.g. OPT) into an application subdirectory i.e. *TestOPT*.

4.4 Copy a "*Makefile*" file from the "*TestPrint*" application subdirectory to each application subdirectory.

4.5 Copy the provided *ASIPMeister* CPU file "*browstd32.pdb"* from "/home/asip00//Sessions/Session1/" into your project directory, and rename it "*browstd32OPT.pdb"*

4.6 Set proper parameters and settings in "*env_settings*" as discussed in Figure 2-5 in the Laboratory Script. Specially the followings:

```
export PROJECT_NAME=brownieOPT
export CPU_NAME=browstd32OPT
export ASIPMEISTER_PROJECTS_DIR=${HOME}/ASIPMeisterProjects
export DLXSIM_DIR=/home/asip00/epp/dlxsimbr_Laboratory
```

5. **Adding the new instruction to *ASIPMeister***

1. Now we start adding new instructions to our processor to speed up the execution.
2. 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 not match with the existing instruction formats.
3. Use the available opcode.
4. You also have to define "CKF Prototype" for each new custom instruction in ASIPmeister, generate GNU tools and use inline assembly in C code.
5. Add the custom instructions to dlxsim as well.
6. Write a small C or assembly code to test your new instruction.
7. 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.
8. If everything is working fine, then simulate the *BubbleSort* C/Assembly code that uses the new instruction in *ModelSim*.
9. Compile the optimized bubblesort application using "*make sim*" and then "*make dlxsim*". Make sure, that the resulting array is still correct.

   (d) How many cycles do you need for execution?

   (e) What is the speedup compared to *original code* (i.e. #Cycles without custom instruction / #Cycles with custom instruction)?

   (f) Attach the assembly and/or C file you used to test the custom instruction and to test the given application.

   (g) Attach the modified ASIPmeister .pdb file.

Next Session: Bubble Sort - **Power &** Area

Estimation and Hardware Implementation

Readings for the next session: Laboratory Chapters 6 & 7