Bubble Sort - Optimisation

Motivation and introduction

In the last session, you have translated the C code for the *BubbleSort* algorithm to assembly code and you simulated the results with *dlxsim* and *ModelSim*. In this session, we will start to add a new instruction to the CPU to speed up the application. This new instruction should be implemented in both of *dlxsim* and the real CPU (*ASIPMeister*). Afterwards, you will manually optimize your individual assembly code in a competition against the other groups! 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.

Exercises

1) Adding a new instruction to *dlxsim*

Now we start adding a new instruction to our processor to speed up the execution. This new instruction will be *bgeu* (branch if greater or equal (unsigned)). This instruction is going to replace the three appearing of the combination from sltu and beqz. First implement the new instruction into *dlxsim*, like explained in the chapters 3.2.2 and 3.2.3 (use the instruction format BRANCH_2_OP with the opcode 0xB0000000). 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 2-4 in the script). Write a small assembly code to test your new instruction (also try things like: -4 ≥ 5; this should be true, as the unsigned version of –4 is bigger than 5). Afterwards use this new instruction in your *BubbleSort* implementation from exercise 4 and name the resulting file *bs_bgeu.s*. 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 session). However, the files for *dlxsim* are created before the assembler is called.

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

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

2) Adding the new instruction to *ASIPMeister*

Create a new project directory inside your *ASIPMeisterProjects* directory for the new CPU and name it *dlx_bgeu*. You can use a copy from the *dlx_basis* CPU project from the last session, but do not forget to adjust the *env_settings*.
After preparing your new project directory start *ASIPMeister* and add the *bgeu* instruction to your new CPU. First you will need a new instruction format for *bgeu*, but the *MicroOp Description* will become the most difficult part (use the “Micro-Operation Description Coding Guide” in /Software/epp/ASIPmeister/share). Have a look at *sltu* and *beqz*, as those commands are the origin for the new *bgeu* instruction and understand the flags of *ALU.cmpu()*. Try to change only few things compared to these instructions! Unfortunately, not everything that *should* work will produce correct VHDL code with ASIPMeister.

Hints:

- Avoid using macros in your new *MicroOp Description*. You can test things faster if you can write them directly instead of writing them to a macro.
- 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 number of delay-slots depends on the pipeline stage in which you write the PC. If *bgeu* writes the PC in another stage than the other jump instructions do, then *bgeu* will have a different amount of delay slots, which then would have to be considered in your assembly code.
- You have to add a new instruction format for *bgeu*. Have a look at the other branch/jump instructions and see how they handle the labels.

Simulate the new instruction with *ModelSim*. Use the small test application that you created to test your *dlxsim* implementation in the last session for this purpose. If everything is working fine, then simulate the *BubbleSort* assembly code that already uses the *bgeu* instruction in *ModelSim*. There might be some problems if you want to use *bgeu* with negative values. Test it for negative values too, but if it is working fine for everything except negative values, then it is ok.

a) Attach the *ASIPMeister* file for the *bgeu* CPU to the mail.

3) Optimizing the assembly code

In the session last week, the *NOP* instructions have only been used to fulfil the data dependencies. This time we will try to reduce the number of *NOP's* to reduce the total number of executed cycles. **The goal is to get the code (correctly) executed in as few cycles as possible**! This is a **competition** against the other groups and to make sure that everyone uses the same environment you have to use the framework *bs_Framework_pipelined.s*. In this framework, the main method and the remainder of the *bubblesort* method have been prepared with a non-optimized usage of *NOP's*, which is **not** allowed to be changed for your optimizations! You are also **not** allowed to remove the stack operations for e.g. saving the registers on the stack, but you are allowed to reschedule them to get rid of some *NOP's*. Of course, you are also **not** allowed to just write the correct result into the data memory, as one could say that the initial data memory is static and so the result is always the same. Your algorithms still has to compute the correct result itself.

Start with creating a copy of your assembly code with the pipelined framework and name the resulting file *bs_bgeu_opt????.s*, where the four question marks are replaced by the needed number of cycles *dlxsim* needs to execute the program. Although you should work with *dlxsim* the most of the time, it is important to test whether your code is also running correct in *ModelSim*, as this is the accurate simulation. If an assembly code is running with *dlxsim* but not running with *ModelSim*, then this is a bug in *dlxsim* not vice versa and thus it is a bug in your assembly code.

a) Attach your fastest-but-still-correct *bs_bgeu_opt????.s* to the mail.

Hints:
- For every version of your program, that is faster than the successor you should create a new *bs_bgeu_opt????.s*, as sometimes optimizations are no longer correct and then you will need a backup point from where you can start your next try.
- If you consciously use an old value of a register (with an unresolved data dependency) then you should mark this in the comment, e.g. "*r10'OLD*". This information will become very important when you later think about further optimizations or when you search for a bug in your code.

- Carefully read and understand the examples about optimization in chapter 3.1.1.

To give you some kind of motivation and to show you what is possible:
The students in the prior semesters managed to go down below 2,500 cycles to sort the array. Some of the groups were even able to break the 2,000 cycle's barrier by using some aggressive optimizations. It is not necessary to reach these numbers but you have to try to run your application as fast as possible.