**Simulation of Assembly Programs**

**1 Week**

**Motivation and Introduction**

The main goal of these exercises is to get used to assembly programs. You are supposed to have a look at some code examples. Every example shows a basic operation. Examine the code, simulate it with *dlxsim* and answer the corresponding questions. The assembly code is available in the directory “*/home/asip00/Sessions/Session1/*”. *Dlxsim* is available in the directory “*asip00/**epp/dlxsimbr\_**Laboratory/*”(default). Before using *dlxsim* you have to read the chapter 3 in the Laboratory Script, but you can skip the sub-chapters 3.2.2., 3.2.3 and 3.3.2 for now. You can copy *dlxsim* to your directory or you can start it from the default directory.

The below printed exercise numbers correspond to the numbers from the assembly file names. If you are familiar to assembly code then don’t get bored with the first 3 exercises, the numbers 4 to 6 are trickier. For every exercise the part that starts like “a)”, “b)” … you have to write an answer. This answer should mainly prove that you have understood the problem, so you can make your answers short, but they still have to answer everything, that was asked. Mail your answer with a CC to your group members to **sajjad.hussain@kit.edu** and use the topic “asipXX-Session1”, with XX replaced by your group number.

**Exercises**

The following exercise numbers like “**1)**”, “**2)**” … correspond to the given assembly files in the “*/home/asip00/Sessions/Session1*” directory. You have to:

1. Create a project directory for this session by copying the directory “*/home/asip00/**­epp/ASIP­Meister­Projects/TEMPLATE\_PROJECT/*” and renaming it (e.g. *browstd32*) as discussed in Chapter 2.3 and then create subdirectories for each assembly example in “*Application*” directory.
2. Set proper parameters and settings in “*env\_settings*” as discussed in Figure 2-5 in the Laboratory Script.
3. Copy a “*Makefile*” file from the “*TestPrint*” application subdirectory to each application subdirectory. This Makefile has been prepared to help you in performing different tasks during the Lab as discussed in Figure 2-3 in the Laboratory Script.
4. In an application directory, typing, “*make*” on shell will show usage of the “*Makefile*” and how different parameters can be passed if required.
5. For exercises of today, you have to configure the pipeline forwarding to 0 as it is described in chapter 3.2.1 of the Laboratory Script. For instance, to run the first example you can type the following in its subdirectory:

*make dlxsim DLXSIM\_PARAM="-f1\_Arith.s -da0 -pf0*"

1. **Basic Assembly Instructions**

Understand the functionality of every instruction and understand the values of every target register after the program run has completed (see the dlxsim chapter for reading register values in dlxsim simulation). Check the number of cycles needed to execute all instructions.

1. What is the reason for this high number of cycles? Which instruction causes that behaviour and why is it doing so?
2. What is the purpose of the “trap #0” instruction?
3. **Memory Access**
4. Explain the goal of the instruction combination *LSOI /ADDI*. What is generally (so: not only for this specific example) the register value after *ADDI*, what is it after the additional *LSOI*?
5. Why is it in general not possible to omit the *LSOI* instruction, although it would be possible in this special example?
6. **Branches**
7. Which high-level control structure (e.g. ‘call subroutine’ …) is implemented in this example code?
8. What is computed with this example? R24 = *function* (R21, R22);
9. Look at the *NOP* instructions and explain why they are placed there.
10. Which meaningful instruction (i.e. an instruction that helps executing the code; not just any dummy instruction) can replace the first *NOP* instruction without changing the final result and why does it not change the final result?
11. **Loops**
12. What is computed with this example? R23 = *function* (R21, R22). Debug the application step-by-step with the capabilities of dlxsim.
13. The approach to compute the function in the way this example is doing it has two specific names. Do you know those names? (One is founded by the main operations, while the other is founded historically. You either know the names or you don’t. If you don’t know both names, you may guess)
14. In general, how often is the loop maximally executed? How the input data has to look like to get this maximal number of iterations?
15. **A High Level Structure**
16. Which high-level control structure do you recognize? Explain the purpose of the instruction-block between “*ADDI R23, R0, $(2)*” and “*JPR R24*”.
17. Why do you have to shift by value 3? Explain it with a close view to the body of the control structure and pay attention to the addressing mode of the DLX processor.
18. What are the general differences between branch and jump instructions in the DLX instruction set (also have a look at the different instruction formats to find a part of the answer)?
19. **Writing Assembly Code**

Consider the following fragment of C code:

for (i=0; i<10; i++) {

A[i] = B[i] + 5 + C;

}

Assume that A and B are arrays of 32-bit integers in the memory and that C is a 32-bit integer, that is also placed in memory (currently not available in a register). Write the corresponding assembly code for the DLX processor. Your code should be as short as possible (in terms of number of instructions) and it should compute the same result.

1. Attach the assembly file to the mail.
2. How many assembly instructions do you need altogether?
3. How many clock cycles does your code need to execute?
4. How many memory accesses are performed altogether?

**Next Session:** ASIP Meister and ModelSim

**Readings for the next session**: Chapters 2.3, 4 & 5

**Note:**To run, ASIPmeister, Modelsim & Xilinx ISE during the lab, you need to export the following variables, or you can add it your /home/.bashrc.user.

*export ASIPS\_LICENSE=29000@i80asip.ira.uka.de*

*export PATH=/AM/ASIPmeister/bin:$PATH*

*export ASIP\_APDEV\_SRCROOT=/home/asip00/epp/AM\_tools*

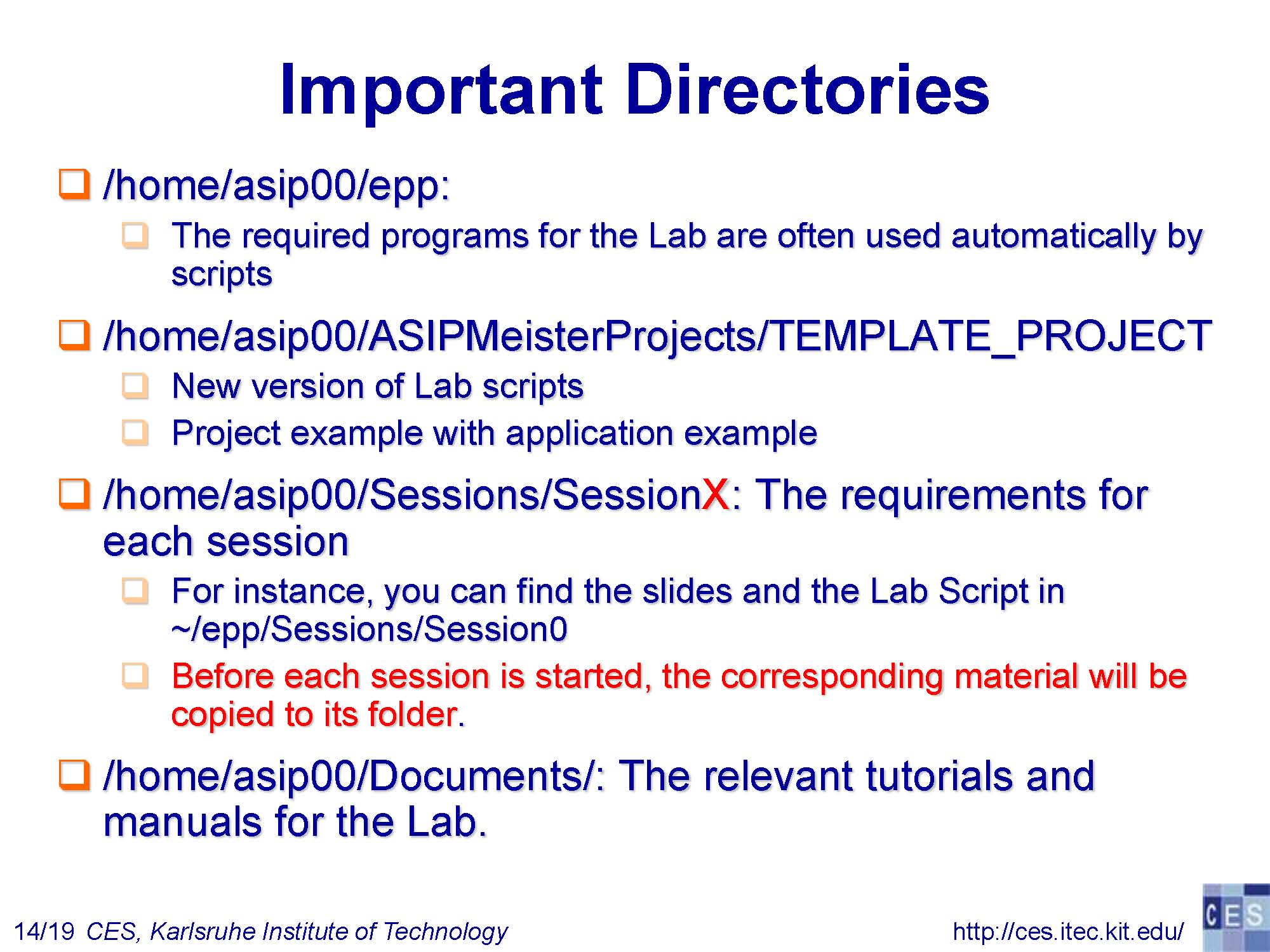
*export PATH=/usr/java/jre1.6.0\_45/bin:$PATH*

*export ASIPmeister\_Home=/AM/ASIPmeister*

*export ASIPmeister\_HOME=/AM/ASIPmeister*

*source /home/adm/modelsim\_66d.setup*

*source /home/adm/xilinx\_13.2\_32bit.setup*



**ASIP Meister and ModelSim**

**1 Week**

**Motivation and introduction**

In this exercise, you will be introduced to *ASIP Meister*, *ModelSim* and our special directory structure, which makes it easier to combine *ASIP Meister*, *ModelSim*, *dlxsim* and the later introduced tools. The overall workflow of the tools is given at the end; you don’t need to deeply look into this. It is just for an overview, we will learn about this flow gradually. First, you have to read Chapter 2.4 from the Laboratory Script to understand the directory structure. Read this chapter extremely carefully, as this one is very important for every task you have to perform during the laboratory. Then you will create your first own *ASIP Meister* project and you will go through the *ASIP Meister* “*User Manual*” and “*Tutorial*” to get used to *ASIP Meister*. Afterwards you will simulate a given Assembly Code with *dlxsim* and with *ModelSim*. Therefore, you will have to read the Chapters 2.3 (for preparing the simulation files for *dlxsim*) and 5 (for using *ModelSim*) of the Laboratory Script. 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-Session2”, with XX replaced by your group number.

**Exercises**

1. **Preparing the Project**
2. Copy “*AM\_tools*” from “*asip00/epp*” directory to your account, for example at your home folder. Export different environmental variables for ASIPmeister and ModelSim or put them in the ***bashrc.user*** as described in Chapter 2.1. Please set “*AM\_tools*” path from your home folder. It needs write permissions.
3. Copy the provided *ASIP Meister* file “*browstd32WithoutAdd.pdb*” from “/home/asip00//Sessions/Session02/” into your project directory and rename it as “*browstd32.pdb*”.
4. After you have read Chapter 2 of the Laboratory Script, you can start creating your own project directory structure in your home directory. Create a copy of the *TEMPLATE\_PROJECT* directory from “*/home/asip00/epp/­ASIP­Meister­Projects*” for your first project. Name your project directory “*brownie*” and adjust the “*env\_settings*” accordingly.
5. Create a directory for the example application in your “*Applications*” directory, name this directory “*LoopExample*” and copy the given assembly file from “/home/asip00/Sessions/Sessions /Session02/” into this directory.
6. Copy the “*Makefile*” to your application directory from “*Application/TestPrint/*”. Now your first project directory is completed and you can start using it during the following exercises.
7. **Using ASIP Meister**
8. In your project directory, start *ASIP Meister* for the given basis CPU: “*ASIPmeister browstd32.pdb &*”. Moreover, do not forget to start *ASIP Meister* in your project directory, because it will create the “*meister*” subdirectory to generate different VHDL files and GNU tools, where it is started and the “*meister*” subdirectory is expected to be in your current project directory.
9. Read the *ASIP Meister* “*User Manual*” and “*Tutorial*” to get used to the GUI. You can find the related files in the “*/home/asip00/Documents*” directory. Read systematically through both files simultaneously and play with the specific parts of the GUI for which you are currently reading the user manual and tutorial.
10. If you anyhow configure something wrong, then just reload the original file. The most important parts for the later work are the “*Resource Declaration*”, “*Instruction Definition*”, “*MicroOp Description*”, “*HDL Generation*” and “*Compiler Generation*” so have a detailed look at them.
11. What is the difference between the *MicroOp* implementation for *jp* and *jpr*. What is happening in hardware and when is it happening?
12. What is the difference between the *MicroOp* implementation for *sub* and *subi*. Why this difference is only needed for the immediate instructions?
13. What do you think that ForwardDataFromEXE() and ForwardDataFromWB() macros are used for?
14. How many stages this CPU has? Normally, CPU has fetch, decode, execute, memory and write-back stages, how these stages are mapped into brownie 4 stage CPU?
15. **Adding an Instruction**
16. The *add* commands (*add*, *addi*) have been removed from the provided CPU. Go ahead and insert them into the provided CPU. You will have to work with the *Instruction Definition*, and *MicroOp-*Description. Use the following *func/opcode* -values: *add*:00000000000/000001, *addi*: -/100000.
17. If you are unsure about any detail, then have a look at the corresponding subtract “*sub*” commands. **However, if you only copy-and-paste everything without learning anything, then you will pay for this in later sessions** where you will not have a template to copy-and-paste from. Generate the hardware and the software description files for the modified CPU. Make sure, that you do not have any mistakes in the “*add*” instructions, as this will cause problems in the later sessions.
18. Make sure that you have generated VHDL files and GNU Tools generations using “*Assembler Generation*”, “*HDL Generation*” and “*Compiler Generation*”
19. **Simulating with dlxsim**
20. Use the public-key authentication system described as the last point in Chapter 2.2.1 of the Laboratory Script to avoid typing your password multiple times when executing the Makefile.[**optional**]
21. Now you have a CPU that is able to execute the given example code *6\_for.s*. This code is similar to the code you created in the last session for exercise 6. First, simulate the assembly code with *dlxsim*. Therefore, you should have a copy for the “*Makefile*” in your “*LoopExample*” subdirectory.
22. Then, go to the “*LoopExample*” subdirectory inside your Applications directory and execute “*make sim*”.
23. When “*make sim*” is finished, a new subdirectory called “*BUILD\_SIM*” containing some important files is created in your current directory, see Figure 2-2. There is a special “*.****dlxsim***” file used for dlxsim simulation and there are “*TestData.IM*” and “*TestData.DM*” files used for ModelSim simulation. *TestData.IM* and *TestData.DM* are instruction and data memory image files.
24. Simulate “*.dlxsim*” file with dlxsim by typing: “*make dlxsim*” with default or “*make dlxsim DLXSIM\_PARAM="-fBUILD\_SIM/LoopExample.dlxsim -da0 –pf1*"
25. How many cycles are required to execute this program?
26. Does the real CPU have a *NOP* instruction? If not, into which instruction is a *NOP* translated and what parameters (register and immediate values) does this instruction get as input? In Dlxsim, you can use "*get start\_address #of\_of\_instructions*" to see the 32-bit binary values of each instructions and from this, you can also extract opcodes.
27. Can you see the data \_A, \_B and \_C variables in TestData.DM? What does the first value word indicates? This large value is a stack pointer value, used while you call many subroutines one by one. Why it is so large values? What does this value actually indicates an address? Or a data value?
28. **Simulating with *ModelSim***
29. Now we simulate the assembly program together with the real CPU VHDL files. Read Chapter 5 in the Laboratory Script, and create a ModelSim project in your “*ModelSim*” directory (see Figure 2-3) for your CPU and simulate the program.
30. Compare the created “*TestData.OUT*” with the results from dlxsim. Try to understand the program flow by looking at the “*TestData.IM”* and the *register write* part from *ModelSim* *waveforms*.
31. Write a short description what is happening in the first data memory access. Mention all the data bus signals and the final register write back. In the upper part of the *ModelSim waveform,* you can see a clock counter. Use this value to describe when something is happening. Sometimes things happen in the middle of a clock cycle (i.e. falling edge); mention this, too. **Take help from Brownie-Std32 datasheet from asip00/Documents, see "Memory Access at Page 47"**



**Next Session:** C Compiler

**Readings for the next session**: Chapters 8.1, 8.2, 8.3

**C Compiler**

**1 Week**

**Motivation and introduction**

Until now, we have used the assembly code applications. Now we will create a compiler for our customized *ASIP Meister* CPUs. With this compiler, we will compile a C-code application and simulate the result in *ModelSim* and *Dlxsim*. This session also introduces about different peripheral where we forward our text/data, and how different libraries are used for their peripherals. The information about creating and using the compiler can be found in Chapter 8 of the Laboratory Script. For every part, that starts like “a)”, “b)” … you have to mail the answers and asked files/tables to **sajjad.hussain@kit.edu** and use the topic “asipXX-Session3”, with XX replaced by your group number.

**Exercises**

1. **Preparing the Project and Creating the Compiler**
2. Based on your ASIPmeister processor design, you can generate GNU Tools accordingly. Which then can be used to compile, assemble and link different assembly and C files. Therefore, you can automatically create a compiler for your individual processor! To get an idea of how retargetable compilers are working, read Chapter 8.1 from the Laboratory Script.
3. Create a project directory as in previous sessions and copy the provided processor *browstd32.pdb* into your project directory, which already includes ADD instructions. [You can also start from the last session’s project and just create sub-directories in “Application” directory. This will save time for the following steps 3 &4.]
4. Setup your project directory by adjusting “*env\_settings*”.
5. Generate VHDL files and GNU tools as in the previous sessions.
6. Please remember that “*AM\_tools*” location is kept as in the previous session.
7. **Compiling and Simulating the Application**
8. Create a subdirectory in your “*Applications*” directory and copy “*/home/asip00/Sessions/Session4/6\_for.c*” to this subdirectory. This is the same for-loop example as in the previous session but now in C program. Also, copy the required “*Makefile*”.
9. Simulate the program in DLXsim and in ModelSim. Run “*make sim*” and “*make dlxsim*”.
10. How many cycles are required to execute this program DLXsim and ModelSim?
11. In folder BUILD\_SIM, look at the “*6\_for.s*” which is generated. Another file “*startup.s*” is used along with the generated “*6\_for.s*” to generate TestData.IM/DM files. Just understand and remember the structure of “6\_for.s” files if you have to write your own .s file, and how it is being executed along with “*startup.s*”.
12. **Compiling and Simulating another Application**
13. Create another subdirectory in your “Applications” directory and copy “*/home/asip00/Sessions/Session4/app.c*” to this subdirectory. This is a simple example to direct a text to some peripheral devices like LCD or UART. Also, copy the required “Makefile”.
14. However, for compiling it, you first need to provide the required libraries to your respective application, i.e. “*lib\_lcd\_dlxsim.c*”, “*lib\_uart.c*”, “*loadStoreByte.c*”, and “*string.c*”. Chapter 8.5 describes how to provide these libraries.
15. Simulate the program in DLXsim and in ModelSim. Run “*make sim*” and “*make dlxsim*”.
16. After compiling, simulate the application in dlxsim and ModelSim and compare whether the printed results are the same as expected. The dlxsim and ModelSim will print text to a *virtual* LCD/UART. For dlxsim you can forward the LCD/UART output to a file, using the “*-lf*” and “*-uf*” parameters respectively, e.g. “*make dlxsim DLXSIM\_PARAM=”-da0 –pf1 -lf****lcd.out*** *-uf****uart.out***” writes output to the file “*lcd.out*” and “*uart.out*”. While ModelSim automatically writes to the file “*lcd.out*” and “*uart.out*”.
17. How many cycles are required to execute this program DLXsim and ModelSim?
18. The default GCC compiler optimization is –O0. Try different optimization levels with dlxsim and ModelSim using e.g. “*make dlxsim GCC\_PARAM=-O1*” or using “*make sim GCC\_PARAM=-O1*”.
19. Repeat this benchmarking for all compiler optimization-levels like O0, O1, O2, O3 and O4 for both dlxsim and ModelSim.
20. Does the application “*app.c*” is executed successfully using different optimization levels? If yes, please fill the following benchmark table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Optimization Level** | **Executed?**  [Yes/No] | **Cycle count** ModelSim | **Cycle count** dlxsim |
| **-O0 (default)** |  |  |  |
| **-O1** |  |  |  |
| **-O2** |  |  |  |
| **-O3** |  |  |  |
| **-O4** |  |  |  |

**Next Session:** Adding Custom Instructions

**Readings for the next session**: Chapters 8.2.3, 3.2.2, ASIPmeister Tutorial

**Adding Custom Instructions**

**2 Weeks**

**Motivation and introduction**

In this session, we will implement some custom instructions for an application to speed up the execution time. Moreover, even when the compiler uses the new instructions, they might not be used in all optimization levels. For that, we will also introduce the feature, which is used to add inline assembly to the application. By using inline assembly, you can force the usage of custom instructions or you can optimize bigger blocks (e.g. application hot spots) in hand written assembler. The information about creating and using the compiler can be found in previous session. For every part, that starts like “a)”, “b)” … you have to mail the answers and asked files/tables with a CC to your group members to **sajjad.hussain@kit.edu** and use the topic “asipXX-Session4”, with XX replaced by your group number.

**Exercises**

1. **Preparing the project**
2. Create a project directory and setup your project directory by adjusting “*env\_settings*”.
3. In the project directory, create a new project and generate compiler for the basis CPU, we recommend you to copy a fresh CPU version from “*/home/asip00/Sessions/Session3/browstd32.pdb*”). If compiler generation is successful, a compiler binary will be generated in your project directory. [You can also start from the last session’s project and just create sub-directories in “Application” directory. This will save time.
4. **Compiling and Simulating the Application**
5. Create a subdirectory in your “*Applications*” directory and copy “*/home/asip00/Sessions/Session4/arrayloop.c*” to this subdirectory. Also, copy the required “*Makefile*”.
6. First, you have to compile the application using gcc compiler to compare with the later results from dlxsim and ModelSim. For *gcc* you can forward the printed output to a file, e.g. “*a.out > output\_gcc.txt*” (‘*a.out*’ is the default name of the binary that is created when you compile “*gcc arrayloop.c*” while ‘*output\_gcc.txt*’ then contains the printed array). To compile with GCC, comment the line “*#define ASIP*”.
7. The usage of the compiler is explained in Chapter 8.3 of the Laboratory Script. After reading this chapter, compile arrayloop.c using “make sim”. However, for compiling it, you first need to provide the required libraries, i.e. “*lib\_lcd\_dlxsim”* (also for ModelSim), “*loadStoreByte”*, and “*string”*. Chapter 8.5 describes how to provide these libraries.
8. After compiling, simulate the application in dlxsim and ModelSim and compare whether the printed results are the same compared to a *gcc*-compiled version. The *gcc* version will print the arrays on the screen and dlxsim and ModelSim will print them to a *virtual* LCD. For dlxsim you can forward the LCD output to a file, using the “*-lf*” parameter, e.g. “*make dlxsim DLXSIM\_PARAM=”-da0 -lf****output\_dlxsim.txt***” writes output to the file “*output\_dsim.txt*”. ModelSim automatically writes to the file “*lcd.out*”.
9. To compare, whether the files generated from gcc, dlxsim & ModelSim are identical, you can use command-line tools like “*diff output\_gcc.txt output\_ModelSim.txt*” or graphical tools like “*kompare*” or “*kdiff3*”.
10. How many cycles do you need for execution in dlxsim and ModelSim?
11. **Adding a new instruction to *dlxsim* Simulator**
12. Now we start adding new instructions to our processor to speed up the execution. These new instructions are “*avg rd, rs0, rs1*”, “*swap rd, rs*” and “*minmax rdMin, rdMax, rs0, rs1*”. First, implement the new instructions into dlxsim, as explained in the Chapters 3.2.2 and 3.2.3 of the Laboratory Script. Use the instruction format and opcodes as below:

OpcodeInfo opcodes[]

//name class op mask other flags rangeMask

{"swap", ARITH\_2PARAM, 0x541, 0x1ffff, 0x20, 0 , 0xffff8000 },

{"avg", ARITH\_3PARAM, 0x6c1, 0x1ffff, 0x20, 0 , 0xffff8000 },

{"minmax", ARITH\_4PARAM, 0x981, 0xfff, 0x20, 0 , 0xffff8000 },

{"rot", ARITH\_3PARAM, 0x8, 0x3f, 0, CHECK\_LAST|IMMEDIATE\_REQ, 0xffff8000 },

1. 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 2-5 in the Laboratory Script).
2. Write a small assembly code to test your new instructions in dlxsim. You can use the Session1 assembly language program as the reference.
3. **Extending the CPU with a custom instruction**
4. Create a new *ASIP Meister* Project “*browstd32avg*” from your old project “*browstd32.pdb*” (do not forget to adjust the “*env\_settings*”). In your new CPU, implement the new instruction “*avg rd, rs0, rs1*”, “*swap rd, rs*” and “*minmax rdMin, rdMax, rs0, rs1*” as they are used in the application. This new instruction “*minmax*” computes both the minimum and the maximum of two inputs *rs0* and *rs1* and write them simultaneously to two registers (*rdMin* and *rdMax*).

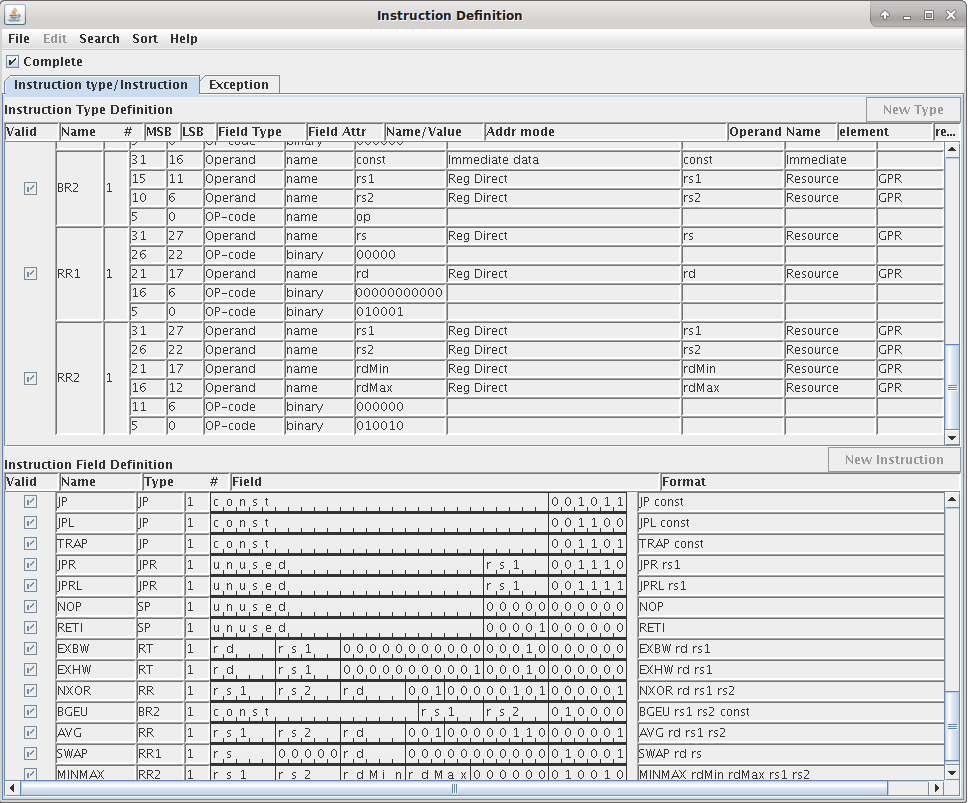
**Hint:** You can use the opcode and instruction formats as indicated in the figure below.

**Hint:** Do not implement the “*swap”* instruction as it is written in the C-code. Think what this instruction is doing and implement it without any shifts! Test the new instructions with a small assembly code in ModelSim.

1. Generate the VHDL Files.
2. Please remember that for new custom instructions defined in ASIPmeister to be used automatically with C Compiler, you have to implement relevant “**CKF Prototype**” in ASIPmeister. Following instructions at section 4.11.C in ASIPmeister tutorial and 11.2 in ASIPmeister user manual.
3. Generate GNU Tools for your new processor.
4. **Compiling and Simulating the Application with custom instructions**
5. You can test your custom instructions with small assembly programs.
6. After testing the new instruction with a small assembly code, use *inline assembly* in the application “*arrayloop.c*” for using the new instructions, see Chapter 8.2.3 in the Laboratory Script. Rename it to “*avg.*c”.
7. After modifying the application code by the *inline assembly* stuffs for a particular custom instruction, compile the application using “*make sim*”, make sure that the result is still correct (*diff* the ModelSim output with gcc-compiled version) and find out, whether the new instructions have been used or not.

**Note:** you have to check the generated assembly code to be sure that the new custom instructions are used in the code.

1. Now you have to determine the number of cycles for executing the application to compute the speedup against the old CPU with the old compiler. To determine the number of cycles you have to remove (i.e. comment out) the loop for printing the results! Otherwise, this loop is the dominating hotspot and you will not notice a significant speedup when using the new assembly instructions!
2. Simulate the application with ModelSim.
3. How many cycles do you need for execution in dlxsim and ModelSim? Attach assembly programs you used to test custom instructions with this mail.
4. What is the speedup of “*avg.c*” compared to *“arrayloop.c*” (i.e. #Cycles without custom instructions / #Cycles with custom instructions)?



**Next Session:** BubbleSort – Simulations and Optimization

**Readings for the next session**: All the Chapters, especially 4 & 8, ASIPMeister Tutorial & Manual

**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**
2. 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.
3. 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.
4. 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.
5. **BubbleSort\_Address**
6. 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.
7. 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.
8. 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.
9. **Compiling the application in dlxsim and ModelSim with basis processor**
10. 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.
11. Then you have to start ASIPmeister in your project directory and generate hardware and software files for the provided “*browstd32.pdb”*.
12. 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.**

1. 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.
2. How many cycles do you need for execution in dlxsim and ModelSim?
3. **Bubble Sort – Optimisation: Customizing the basis processor**
4. 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.

1. **Adding the new instruction to *ASIPMeister***
2. 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.
3. 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.
4. Use the available opcode.
5. 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.
6. 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.

1. 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.
2. If everything is working fine, then simulate the *BubbleSort* C/Assembly code that uses the new instruction in ModelSim.
3. 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*”.
4. 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.
5. How many cycles do you need for execution?
6. 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

**Bubble Sort - Hardware Implementation**

**1 Week**

**Motivation and introduction**

In this exercise, you will learn how to synthesize and implement your project and then download it on FPGA board and see the results on the UART terminal or LCD. You will test your BubbleSort implementations on the FPGA Board. For visualizing, the output of BubbleSort when running on the FPGA Board the resulting array plus some additional information is printed to the URAT interface. You should use “*lcd\_dlxsim.c*” for virtual LCD in dlxsim and Modelsim and use “*lcd.c*” for real LCD interfaced with FPGA. The main difference is the implementation of the “*t\_print*” method (see Chapter 8.5 of the Laboratory Script): the dlxsim/ModelSim implementation is just printing all characters to the console or to a file whereas the implementation for the FPGA sends additional control signals and waits for the corresponding answers from the UART. Using these frameworks, the Bubble sort algorithm which will be implemented using the two CPUs to form two versions:

**Version1:** basis CPU (*browstd32.pdb*) with basis bubble sort algorithm

**Version2:** modified CPU (*browstd32bgeu.pdb*) which supports new instruction “*bgeu*”

**Vesrion3**: modified CPU with any other optimization you carried (*browstd32opt.pdb*).

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-Session6”, with XX replaced by your group number.

**Exercises**

1. **Creating the applications**
2. You have to create the software and the hardware sub directories under “*Application*” for browstd32.pdb CPU. First, create a new project directory inside your *ASIPMeisterProjects* directory for the new CPU and name it “*browstd32”*. 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 two subdirectory for the software and the hardware application in “Applications” directory.
3. Copy the provided “*app\_LCD.c”* and “*app\_UART.c”* from “/home/asip00/Sessions/Session6” to the created LCD or UART subdirectories respectively.
4. Copy the C libraries from “*asip00/epp/StdLib*” to each application subdirectory. For LCD simulation in dlxsim and ModelSim use “dlx\_lcd.c”. For real LCD implementation in FPGA use “lcd.c”.
5. Also, copy the “*Makefile*” to both the subdirectories.

* Dlxsim will write all the “*t\_print*” output (including the control signals; each character is written to a new line) to the console. However, you can also forward the output to a file and look at it after the simulation has finished. Use the “*-lf{fileName}*” or “*-uf{fileName}*”.
* Similarly, ModelSim will write all the “*t\_print*” output to “*ModelSim/lcd.out*” and “*ModelSim/uart.out*”.
* You can restart the CPU by pressing the “*reset*” push button on the small mini board on the FPGA board, but REMEMBER, that your array in data memory is already sorted after the first run, so the second, third … run will be significantly faster than the first one.
* The BubbleSort framework is measuring the number of cycles for the execution of the bubbleSort methods. This measurement is done by a counter on the FPGA Board or in dlxsim/ModelSim respectively. This measurement only measures the bubbleSort method, but not the overhead for e.g. printing the result.

1. Compile (“*make sim*”) the application for basis CPU and generates the required .dlxsim and DM/IM file for the dlxsim and ModelSim respectively.
2. Simulate the application with dlxsim using “*make dlxsim*”. It will start the dlx simulator to simulate the compiled file generated in the previous stage. Here, you have to pass some parameters to dlxsim such as the LCD file to print the outputs. For that, you have to type “*make dlxsim DLXSIM\_PARAM="-da0 -lfputc.out"*” where “*putc.out*” is the file than contains the printed output. Note: the pipeline is by default has the value 4. You have to check “*putc.out*” file in order to see the result of your code and check the sorted array.
3. Generate ASIPMeister hardware and software description files for browstd32.pdb CPU and then simulate the application with ModelSim, which will use DM/IM files already generated. Compare the resulted array from “*putc.out*” and “*lcd.out*”, it should be same.
4. **Implementing Project Version1**
5. Every ASIP project has “*ISE\_Framework*” directory, as they all instantiated from the default *TEMPLATE\_PROJECT* in “*/home/asip00/­ASIP­Meister­Projects*”. This “*ISE\_Framework*” directory contains all the required IPs and VHDL files to build you FPGA project. Remember to adjust “*env\_settigs*” accordingly. Create Xilinx ISE project using “*ISE\_Framework*” as discussed in Chapter 6 of the Laboratory Script, and add required source files. Synthesize, implement and generate programming file for the FPGA.
6. Compile your application using “*make sim*” and then “*make fpga*”, it will compile your file and generate the required DM/IM file and combine them with your bitstream which is generated from the Xilinx ISE tool (note that “*env\_settings*” will tell the Makefile where the ISE project folder is located). Finally, a new bitstream file containing your hardware CPU along with corresponding IM/DM files of your application will be generated in the folder “*BUILD\_FPGA*”. This bitstream must be used to configure the FPGA.
7. Upload the existing bitstream to the FPGA prototype board using “*make upload*” (note: this command does not generate a new bitstream) then you can check the results on the UART. You can open HyperTerminal using “hterm &” and adjust its settings like: Baud rate=115200, Stop bit=1, Data bits=8, Parity=None, COM Port=ttyUSB0 (for example), Newline at=CR+LF,
8. If your design works correctly, find out the design statistics (speed and area, see Chapter 6.4 of the Laboratory Script)
9. Compute the accurate time (in ms) required to sort the 20 numbers. Use the number of executed cycles (printed on the URAT interface) and the max. CPU frequency on the FPGA board, where the sorting is still correct) HINT: When you run bubble sort second time by just pressing the reset button, then it will be significantly faster, as the array in the memory was already sorted from the first run! You have to upload the Bitstream again for a second test
10. Analyse the time and find the critical path (see Chapter 6.5 of the Laboratory Script)
11. **Implementing Project Version2**
12. Create the software and the hardware applications for “*browstd32bgeu.pdb*”, simulate and implement this version on FPGA prototype board using the steps above.
13. If your design works correctly, find out the design statistics (speed and area).
14. Compute the accurate time (in ms) required to sort the 20 numbers.
15. Find the critical path.
16. Compare the design statistics between the two versions.
17. How does this affect the execution time (in cycles and ms)?
18. **Implementing Project Version3**
19. Create the software and the hardware applications for “*browstd32opt.pdb*”, simulate and implement this version on FPGA prototype board using the steps above.

**Next Session:** Bubble Sort - Power Estimation

**Readings for the next session:** Chapters 7

**Bubble Sort - Power Estimation**

**1 Week**

**Motivation and introduction**

In this exercise, you will learn how to estimate the power consumption on gate-level accuracy for different processors and different applications, which were generated in the previous sessions. Then you will compare the power consumption for a certain clock period. We will use the basis CPU and the CPU which supports the “bgeu” instruction or any other version. The applications which will be used are basis bubble sort algorithm, the bubble sort algorithm which uses bgeu instruction “bs\_bgeu.s” and the optimized version of the bubble sort algorithm. Before you start the exercises, read the Power Estimation tutorial to get a close view about the software tools that will be used in Chapter 7 of the Laboratory Script. The different processors and applications, mentioned previously can be combined to form three versions as following:

**Version1**: basis CPU “*browstd32.pdb*” with basis bubble sort algorithm

**Version2**: CPU, which supports the instruction bgeu “*browstd32bgeu.pdb*” with bubble sort algorithm, which uses bgeu instruction.

**Version3**: CPU, which supports the optimized version that you optimized the bubble sort algorithm “*browstd32opt.pdb*”.

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-Session7”, with XX replaced by your group number.

**Exercises**

1. Create three different project directories in your “*ASIPMeisterProjects*” directory for three versions and set the “*env\_settings*” accordingly.
2. For all the above three versions, generate ASIPmeister hardware and software description files and simulate with ModelSim. During simulation also generate the VCD files for mentioned frequencies (first with 50MHz and then with Max. Frequency found in the last session).
3. Create Xilinx ISE project to estimate power with xPower for three versions as discussed in Chapter 7 of the Laboratory Script.
4. For all three versions, determine the total and dynamic power.
5. Compute the total execution time (ms) for every version. You can use execution as the # of cycles multiplied by the clock cycle in ModelSim.
6. Compute the energy required for every version. Fill in all these results in the table below e.g. *PowerReport.xlx* or *PowerReport.ods*.
7. Does using “*bgeu*” instruction minimize the required energy? Version3 uses an application, which needs less number of clock cycles than Version2; is it also power and/or energy-optimized version compared to Version2?
8. Repeat a-d, but instead of taking the default of 50 MHz, use the individual maximum CPU frequency on which a CPU can run (You can get it from ISE\_Benchmark). This frequency has to be configured in tb\_ASIPmeister.vhd (search for CLK\_PERIOD; e.g. 10 ns half period = 20 ns period = 50 MHz). XPower will automatically load this frequency from the VCD file.

Sample PowerReport.xlx or PowerReport.ods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Total Power [mW] | Dynamic Power [mW] | Execution Time [ms] | Energy [nJ] |
| **Version1**  50 MHz |  |  |  |  |
| **Version2**  50 MHz |  |  |  |  |
| **Version3**  50 MHz |  |  |  |  |
| **Version1**  Max. Freq: -----MHz |  |  |  |  |
| **Version2**  Max. Freq: -----MHz |  |  |  |  |
| **Version3**  Max. Freq: -----MHz |  |  |  |  |

**Next Session:** An IoT Application: Adaptive Differential Pulse Code Modulation (ADPCM)

**An IoT Application: Adaptive Differential Pulse Code Modulation (ADPCM)**

**2 Weeks**

**Motivation and introduction**

This is the final session. You have **TWO** weeks to complete this session, but you will need these weeks! In this exercise, you will work with an IoT application for which you have to create an optimized CPU. There are different possible ways to modify the CPU, depending on your goals and the **area/power** that you want to spend for the custom instructions. After the CPU has been modified, you will benchmark it to get an idea, what you have to pay for your optimizations. In the last semester week, you will present your results to the other groups. The presentations will take place in the Meeting Room 316.2, the exact date and time will be decided mutually. 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-Session8”, with XX replaced by your group number.

**Exercises**

1. **The Application:**

* The application is the ADPCM audio decoder.
* The term Pulse-Code Modulation (PCM) denotes uncompressed audio samples and Adaptive Differential Pulse-Code Modulation (ADPCM) use an adaptive prediction for the next audio sample with a lossy quantization (i.e. the audio signal will not be exactly the same after encoding and decoding).
* You can find the source code for ADPCM decoder with some ADPCM encoded audio data in /home/asip00/Sessions/Session8. When you run the ADPCM decoder, you can recover the original audio samples (approximately).
* Two different versions of the encoded audio data are provided, differing in the size of input data. The MINI version is meant for the initial tests, i.e. use it to test whether your application compiles and whether dlxsim and ModelSim can simulate it. However, the MINI version is too short to hear anything meaningful when playing it on the FPGA prototype board. The BRAM version is the biggest possible version that fits into the FPGA-internal memory (called Block RAM). While testing your application on FPGA you have to use this BRAM version.
* In our application, the uncompressed audio data has 16 Bits per sample. The ADPCM encoded audio data has 4 Bits per sample; two samples are stored together in one Byte.
* The provided encoded audio data is sampled with a certain frequency (i.e. samples per second = sample rate); in our case 96000 samples per second. ADPCM does not need this information; it simply looks at one sample after the other.
* You can change the CPU frequency by using the knop existed on the small extra PCB which is connected to the FPGA board. Here, you can find a table describing knop position and corresponding frequencies.
* Changing the frequency you can figure out what is the slowest possible frequency that makes the CPU decode correctly the audio samples (i.e. the sound still hearable enough without corruption).

|  |  |
| --- | --- |
| Knop value | Frequency (MHz) |
| 0 | 100 |
| 1 | 80 |
| 2 | 66 |
| 3 | 50 |
| 4 | 40 |
| 5 | 25 |
| Else | 100 |

Table-1: Frequency Changing

1. **Your Tasks**

You have to perform the following tasks, the details for which are given in the following exercises.

* To make your optimization comparable with other groups start with the browstd32 CPU provided in Session 3, and test ADPCM application. Do not take any CPU that you already have modified).
* Compile the MINI version of the application; simulate it with dlxsim/ModelSim.
* Then compile the BRAM version and run it on the FPGA prototype.
* Which frequency do you need until the decoding is fast enough? You can hear the difference when gradually increasing the frequency. When there is no difference from one frequency to the other, then the slower one was fast enough.
* Improve/Extend the CPU for speed, power or area. You have to create two different versions based on your improvements. You should have atleast two extensions.
* Test the improved CPU version on dlxsim/ModelSim and on FPGA
* Benchmark the basis and improved CPUs for area, frequency, power, execution time etc.
* Prepare slides that explain your modifications, improvements and results to compete with other groups.
* We will create a new compiler for the ADPCM ASIPs. Typically, we will not simulate in dlxsim but mainly in ModelSim. For the extended CPUs you have to generate new compiler as we did in the previous sessions.

1. **Simulating the Application**
2. Prepare your new project directory and create a subdirectory in your “*Applications*” directory and copy “*/home/asip00/Sessions/Session8/adpcm.c*” to this subdirectory. Also copy the required “*Makefile*”.
3. To compile the application, audio data is needed for decoding; therefore you have to create a link to the audio data that shall be used for decoding. Two different-sized versions of the same audio stream are provided in “*/home/asip00/Sessions/Session8/*”. To create the required link in the application subdirectory you have to execute e.g. “*link –s adpcmDataStereo\_MINI.h adpcmData.h*”. The compilation will take some time due to the large audio data. For short tests, e.g. to test whether the SINAS code compiles and assembles or whether ModelSim simulation gives the correct output, use the “*adpcmDataStereo\_MINI.h*” version of the file.
4. Copy dlxsim simulator to your home directory to implement new custom instruction here. Set “*env\_settings*” accordingly. Usually it is sufficient to simulate the application with ModelSim, but you can also simulate it with dlxsim.
5. First you have to compile the application using gcc compiler to compare with the later results from dlxsim and ModelSim, forward the gcc printed output to a file, e.g. “*a.out > output\_gcc.txt*”.
6. Link the required libraries from *“/home/asip00/epp/StdLib*” to the application subdirectory and compile ADPCM application using “make sim”.
7. After compiling, simulate the application in dlxsim and ModelSim and compare whether the printed results are the same compared to a *gcc*-compiled version. The *gcc* version will print the arrays on the screen and dlxsim and ModelSim will print them to a *virtual* LCD. For dlxsim you can forward the LCD output to a file, using the “-lf” parameter or forward the audio channel data to a file using “*-af*” parameter. ModelSim automatically writes LCD output to the file ‘lcd.out’ and audio channel data to “audio.out” The application can write the decoded audio data to the audio output (to hear it) or it can write the data to the LCD/UART (to see it). You can define this behavior with the “*#define PRINT\_ARRAY*” switch, when set to 1 the decoded hexadecimal data is print in ModelSim generated lcd.out and in a file generated with –lf option in dlxsim. When “*PRINT\_ARRAY*” is set to 0, decode data for left/right channel is saved in ModelSim generated audio.out and in a file generate with “*–af*” option in dlxsim. ModelSim will create an ‘audio.out’ file and dlxsim will write the data to screen (unless you use the “*-af{filename}*” parameter then it will write it to file).
8. Save the printed results from the ModelSim simulation of the original CPU. Then you can compare them with the printed results from your modified CPU; they have to be identical!
9. **Running the Application on FPGA**
10. To test whether the decoder is working correct with the base CPU and later with your modified CPU, you have to run the application on the FPGA prototype (test it with ModelSim first).
11. We have a simple digital- analog converter (DAC) periphery. This DAC is memory mapped attached to the CPU, i.e. the applications ‘saves’ the decoded audio values to a certain address. The methods “*writeToAudioOutR(int data)*” and “*writeToAudioOutL(int data)*” are provided in the *lib\_audio* library.
12. The hardware will automatically send the audio samples with a certain sample rate to the audio out pin. The sample rate of the hardware has to match the sample rate of the audio data; otherwise, the audio will play too fast or too slow. The sample rate of the hardware can be configured in the file “*dlx\_Toplevel.vhd*” i.e. “*KSAMPLES\_PER\_SECOND*” should be set to 96). This is the correct sample rate for the provided audio data.
13. Whenever you write audio data to the hardware audio out it will be buffered in a FIFO. The data of this FIFO is automatically read with the (above-mentioned) sample rate. You may write to this FIFO as fast as you can compute the data. However, if the FIFO is full, then the store instruction “sw” will stall until some space becomes free.

* Therefore, if your application executes faster than this FIFO is read, then the FIFO will slow down your execution. This gives you the possibility to slow down the clock to save energy, or to run other tasks in the case of a multi-tasking environment.
* If your application runs too slow, then the FIFO will become empty, resulting in errors in the audio stream. Try it!
* These effects do not appear in dlxsim or ModelSim simulation, as they do not model/simulate the FIFO, but just perform a simple “*sw*” operation.

1. **Extending the Basis CPU**
2. You may add new custom instructions to speed up frequent computations in adpcm.c. If your custom instruction delays to clock too much, then you can change it into a multi-cycle instruction (i.e. an instruction that is allowed to stay in EXE stage for multiple cycles, similar to “mult”). The details about multi-cycle FHM are given in Chapter 4.4 of the Laboratory Script.
3. You may change parameters for existing hardware blocks. One typical example is the number of read/write ports of the register file, depending on the requirements of your custom instructions.
4. It is complicated (but possible) to change the number of registers in the register file. To do this, all instruction formats have to be modified. If you for example, only use 16 registers, then you only need 4 bits in the 32-bit instruction to denote which register you want to access. Therefore, you have to adapt the instruction formats such that only 4 bits are used to address the register (simplest way is to make one of the bits a constant ‘0’ in the instruction format). Additionally you have to modify the assembly code (or directly the compiler, but changing the assembly code seems simpler) to make sure that only the lower 16 registers are used. Creating a compiler with 16 register is still possible, but needs some debugging.
5. **You have to create, test, and benchmark TWO** **different** **CPUs with different optimizations**. For example, you might create one CPU that is optimized for performance considering the cycles in ModelSim or the CPU that is optimized for the power/energy due to a reduced clock frequency that is possible due to a faster computation or the CPU that is optimized for area, e.g. by removing not required instructions/hardware blocks etc.
   1. Attach to mail your both ASIPMeister CPU optimized for area/performance/power named like “*dlx\_area.pdb*”, “*dlx\_power.pdb*” etc.
   2. Attach to mail if you have defined new resources (.fhm files) in ASIPMeister.
   3. Attach with the mail your adpcm.c which you modified with your custom instructions using SINAS, for the two CPU optimizations.
6. **Benchmarking the CPUs**
7. Make benchmarks for the old (*browstd32*) and the two modified CPUs and compare them with each other. For the benchmarking you have to take care of the following points:

* Make sure, that you do all benchmarks very accurate to make them comparable!
* Always use MINI version of the application
* Always compile with “-O3” to achieve the best compiler output. Note: For debugging purpose “-O0” (default) is recommended.
* Always using ISE\_Benchmark framework for the area, power and critical path analysis
* Always take execution time or number of cycles from ModelSim
* For execution time, power, and energy use the following three CPU frequencies:
  1. 50 MHz; to make it comparable among the groups
  2. The slowest frequency that is sufficient to execute the application fast enough; to see the lowest power consumption. To calculate the slowest still fast enough frequency you have to consider the number of cycles that your application requires to execute the decoder and the time-budget that you have for decoding. The time-budget depends on the number of audio samples and the sample-rate. The sample-rate is configured to 96000 Samples per second. The number of samples depends to the number of entries in your audio-data array. Remember, that – in the compressed array – each sample just requires 4 Bit.
  3. The fastest frequency that your CPU supports (given by ISE\_Benchmark framework); to see the peak performance
* You have to configure the frequency in the ModelSim testbench for power estimation. Remember that you have to configure the half clock period.

1. You have to benchmark and compare the following points:

* CPU Area
* Maximal CPU frequency or Critical Path
* Number of Cycles or Execution Time
* Dynamic Power Consumption
* Energy Consumption

1. **Presenting the Results**
2. In the last week of the semester, you have to present your results to the other groups. Therefore, every group has to prepare slides to:

* Explain the two different CPUs that you have created to optimize ADPCM application.
* Present the problems that you faced while implementing the two new CPUs. This is the interesting part! Maybe also talk about some implementations that you thought about but which you did not realize.
* Discuss your benchmark results; for every point you should have one slide on which the results are shown in a graph (bar graph, lines …).
* Print proper units your axes e.g. “Execution time [s]”, “Execution time [cycles]”, “Power consumption [mW]”, …).
* For every measurement point, print the value of this measurement result to make comparisons easier.

1. You have to mail the slides before the presentation. Name the slides like “asipXX\_presentation.ppt” (or “.odp” or “.pdf”).

**Another IoT Application**

**Motivation and introduction**

1. .