Introduction to ASIPMeister

1 Week

Motivation and introduction

In this exercise, you will be introduced to *ASIP Meister* and our special directory structure. The overall workflow of the tools is given at the end; you do not need to deeply look into this. It is just an overview; we will learn gradually about this flow. First, to understand the directory structure, you have to read Chapter 2.4 from the Laboratory Script. Then you will create your first *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*. Therefore, you will have to read the Chapters 2.3 of the Laboratory Script. 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-Session2", with XX replaced by your group number.

Exercises

1. **Preparing the Lab tools environment**

   1. ASIPmeister is only installed on i80pc57. It is preferred that you always SSH to this PC.
   2. Use the public-key authentication system discussed in Chapter 2.2.1 of the Laboratory Script to avoid typing your password each time when logging into frequently.
   3. To start, ASIPmeister, ModelSim & Xilinx ISE during the lab, you need to export the following variables each time, or you can add it in your "*/home/.bashrc.user*" to load automatically when you start shell terminal.

   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

4. 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*** and set "*AM_tools*" path from your home folder. It needs write permissions.

*** If the ASIPmeister gives error "could not the old work directory for binutils". Copy "AM_tools" to your home directory and set the path accordingly.

2. **Preparing your project**

    1. Create a project directory for this session by copying the directory "*/home/asip00/epp/ /TEMPLATE_PROJECT/*" and renaming it (e.g. *brownie*).

    2. For each application (C or Assembly), you have to create a separate subdirectory in the "*Application*" directory (e.g. *LoopExample*).

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.

3. Copy the given assembly file from "/home/asip00/Sessions/Sessions/Session1/" into this application subdirectory i.e. *LoopExample*.

4. 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.

5. In an application directory, typing, "*make help*" on shell terminal will show usage of the "*Makefile*" and how different parameters can be passed.

6. Copy the provided *ASIPMeister* CPU file "*browstd32.pdb"* from "/home/asip00//Sessions/Session1/" into your project directory.

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

    ```
    export PROJECT_NAME=brownie

    export CPU_NAME=browstd32

    export ASIPMEISTER_PROJECTS_DIR=${HOME}/ASIPMeisterProjects

    export DLXSIM_DIR=/home/asip00/epp/dlxsimbr_Laboratory
    ```

8. Using above steps, for each session, you need to create a separate project if you have modified the CPU or you can have separate application subdirectories for the same CPU.

3. **Using ASIP Meister**

    1. In your project directory, start *ASIPMeister* 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. The "*meister*" subdirectory is expected to be in your current project directory.

2. 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.

3. 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*", "*C Definition*" and "*Compiler Generation*" so have a detailed look at them.

4. Go through all the steps one-by-one as mentioned in the tutorial and generate VHDL files and GNU Tools. VHDL files will later be used in ModelSim for hardware simulation, while GNU tools will be used to compile, assemble, and link your application code. In this step, recommended settings are "*VHDL*" and "*sim. model and syn. model*".

5. Make sure that AM_tools path is set properly in your "bashrc.user" and is permissible.

6. Make sure that you complete both/two steps i.e. "*Input Description Generation*" and "*GNU Tools Generation*".

7. GNU Tool generation may take 10-15minutes. After GNU Tool generation, you will see compiler, assemble and linker according to your instruction sets. See the directory in ${ASIPMEISTER_PROJECTS_DIR}/${PROJECT_NAME}/meister/${CPU_NAME}.swgen/bin.

a) How many pipeline-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? For CPU related details look at the Brownie32-Std datasheet at /home/asip00/epp/Documents.

Resource Declarations:

b) See different hardware resources used in the CPU. Select ALU in the "Instance" list. What do you understand from the contents of "Function Set" tab? What is listed there?

c) How many does read/write ports GPR has?

d) CPU uses full forwarding in pipeline. How many forwarding units are used? How many intermediate register values can we forward now?

e) What should we need to change in GPR and Forwarding Units if we need an instruction, which writes two operands like quotient and remainder (DIV rd1, rd2, rs1, rs2) to be returned?

Storage Spec:

f) What does GPR0-7 are used for?

Instruction Definition:

g) Why does this CPU have **SP** instruction format? What instructions are covered by this **SP** format? Can we map these instructions with some other format and remove SP format from the CPU?

Micro Op. Description:

h) What does ForwardDataFromWB() and ForwardDataFromEXE() are doing? Which hardware resources are being used here?

i) What does GPRRead(src1) macro perform? Why FWO.forward() is used here?

j) What does "*alu_flag*" mean in ALUExec() macro? What does individual bits mean?

VHDL Generation:

k) What does different bits in "*alu_flag*" stand for? You can take help from the generated VHDL for understanding "*alu_flag*" in fhm_alu_w32.vhd.

4. **Simulating with dlxsim**

    1. Now you have a CPU that is able to execute the given example code *6_for.s*. This code has implemented the following part:

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

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

    }

2. In this session, we only simulate the assembly code with *dlxsim*. You should have a copy for the "*Makefile*" in your "*LoopExample*" subdirectory.

3. Then, go to the "*LoopExample*" subdirectory inside your Applications directory and execute "*make sim*".

4. When "*make sim*" is finished, a new subdirectory called "*BUILD_SIM*" containing some important files is created in your current directory. 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 respectively.

5. Simulate "*.dlxsim*" file with dlxsim by typing: "*make dlxsim*" with default settings or "*make dlxsim DLXSIM_PARAM="-fBUILD_SIM/LoopExample.dlxsim -da0 –pf1"*". The parameter pf1 indicates the full forwarding in CPU pipeline, for details see brownie32-std datasheet.

6. Inside dlxsim terminal, you can use “go” to execute whole program, “step” to execute one instruction at a time, and “stats” to see the statistics.
7. The command “make sim” adds some start-up and ending code to your program to generate a “.dlxsim” file which is then executed by “make dlxsim”. You can directly run your assembly program only using dlxsim like DLXSIM_PATH/dlxsim –f**Assembly.s** –da0 –pf1

a) What is meant by the following lines and values in the dlxsim:

   Biggest used address for Text Section (word aligned): 0xdc

   Biggest used address for Data Section (word aligned): 0x130

b) 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. As “get _A 10d” will show 10 values for array A in decimal. What does “get 0 10” gives you? What are these values?

c) Can you see the data _A, _B and _C variables in TestData.DM? What does the first 32-bit word indicate? This large value is a stack pointer value, used while you call many subroutines one by one. Why this value is so large?

d) How many cycles are required to execute this program?

   Next Session: Assembly Programs

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

   BrownieSTD32 Datasheet Introduction and Section 1.1 - 1.3

   2.2.3. Instruction Set Quick Reference

   4. Memory Access

   Appendix A. Delayed Load

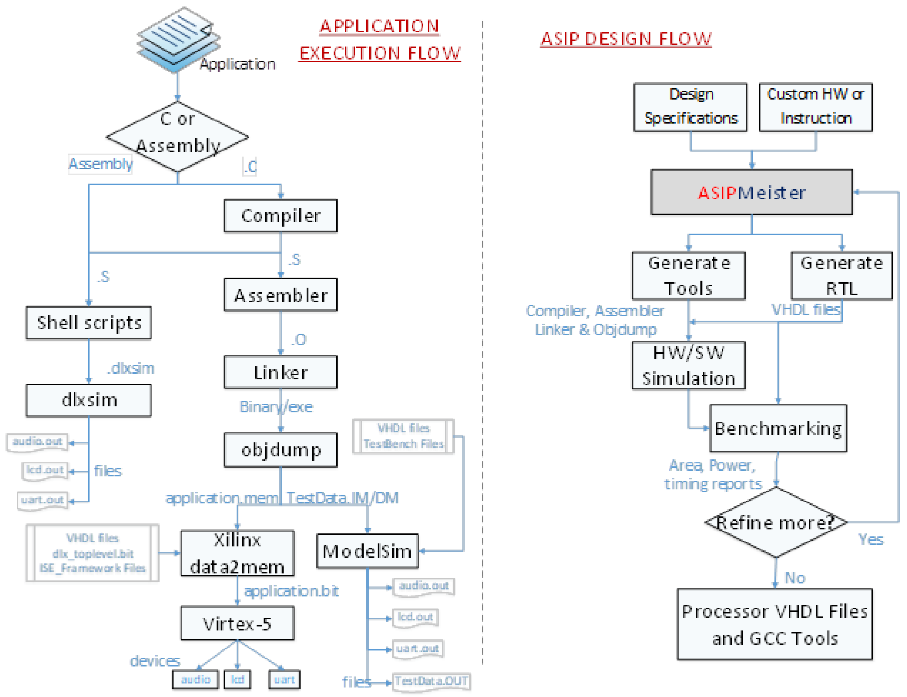

Figure 1: An overview of our Lab Tools

Assembly Programs

1 Week

Motivation and Introduction

The main goal of this session is to get used to assembly programs. 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/Session2/*". *Dlxsim* is available in the directory "*asip00/epp/dlxsimbr_Laboratory/" (default)*. You can copy *dlxsim* to your directory or you can start it from the default directory. 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 to **sajjad.hussain@kit.edu** and use the topic "asipXX-Session1", with XX replaced by your group number.

Exercises

The following exercise correspond to the given assembly files in the "*/home/asip00/Sessions/Session1*" directory.

1. **Preparing your project**

    1. You can use the same project as in the last session, and just create a separate application subdirectory for each assembly example. You can start a fresh project as in the Session 1, but this would be time consuming.
    2. For each application (C or Assembly), you have to create subdirectories in the "*Application*" directory.
    3. Copy a "*Makefile*" file from the "*TestPrint*" subdirectory to each application subdirectory.
    4. Set proper parameters and settings in "*env_settings*.
    5. For these exercises, we will run basic assembly programs in dlxsim and see the effect of pipeline forwarding (-pf1) or no forwarding (-pf0), like

    make dlxsim DLXSIM_PARAM="-da0 -pf0"

2. **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.

a) What is the reason for this high number of cycles? Which instruction causes that behaviour and why is it doing so?

b) What is the code/data address space? What is starting and ending address of code/data section?

3. **Memory Access**

a) 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*?

b) Why is it in general not possible to omit the *LSOI* instruction, although it would be possible in this special example?

c) Read about "forwarding" and "load delay slot" in brownie32 datasheet. Disable NOP after the load instruction, and try executing with –pf0 and –pf1 both.

4. **Branches**

a) Which high-level control structure (e.g. 'call subroutine' . . .) is implemented in this example code?

b) What is computed with this example? R24 = *function* (R21, R22);

c) Look at the *NOP* instructions and explain why they are placed there.

5. **Loops**

a) What is computed with this example? R23 = *function* (R21, R22). Debug the application step-by-step with the capabilities of dlxsim.

b) 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)

c) In general, how often is the loop maximally executed? How the input data has to look like to get this maximal number of iterations?

d) Enable and disable the first NOP, and try executing with –pf0 and –pf1 both.

6. **A High Level Structure**

a) Which high-level control structure do you recognize? Explain the purpose of the instruction-block between "*ADDI R23, R0, $(2)*" and "*JPR R24*".

b) 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.

c) 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)?

7. **ASM Directive in C**

1. You can use assemble instruction directly in c as follows [1,2]:

```
int res=0;

int in1=20;

int in2=30;

int main() {

___asm___ volatile (

"add %[out], %[op1], %[op2]\n"

: [out] "=&r" (res)

: [op1] "r" (in1),[op2] "r" (in2)

);

return 0;

}
```

2. In any asm block, assembly instructions appear first, followed by the inputs and outputs, which are separated by a colon. The assembly instructions can consist of one or more quoted strings. The first colon separates the output operands; the second colon separates the input operands. If there are clobbered registers, they are inserted after the third colon. If there are no clobbered inputs for the asm block, the third colon can be omitted, as Listing 2 shows.

   ```
   int A[10] = {45,23,0,0,0,0,0,0,0,0};
   int main() {
   ___asm___volatile (
   "add %[out], %[op1], %[op2] \n"
   : [out] "=&r" (A[2])
   : [op1] "r" (A[0]), [op2] "r" (A[1])
   );
   return 0;
   }
   ```

3. There is no output operand for the sw instruction, hence the outputs section of the asm is empty. None of the registers is modified, so they are all input operands, and the target address is passed in with the input operands. However, something is modified: the addressed memory location.

   ```
   int res [] = {0,0,0,0,0,0,0,0};
   int a=45;
   int main() {
   ___asm___ volatile (
   " sw %1(%2), %0 \n"
   :
   : "r"(a), "i"(sizeof(int)),"r"(&res)
   );
   return 0;
   }
   ```

4. Branching can be tricky with inline asm. Using labels, the branch-to address can be designated with a unique identifier that can be used as a target branch address.

```
int x, y, z;
int a=45, b=23, c=1;
int main() {
___asm___ volatile (
" addi %[out1], %[op1], %[op2] \n"
" brnz %[op3], here \n"
" there: add %[out2], %[op1], %[op2] \n"
" here: mul %[out3], %[op1], %[op2] \n"
: [out1] "=&r" (x), [out2] "=&r" (y), [out3] "=&r" (z)
: [op1] "r" (a), [op2] "r" (b) , [op3] "r" (c)
: "r0" );
return 0;
}
```

5. See “A guide to inline assembly for C and C++” for further details.

a) Write a C program using assembly instructions to load two consecutive values from an array and store their addition or subtraction at the fourth location of an array depending on the third value of array i.e. 0 means addition and 1 means subtraction.

Next Session: ModelSim Simulation

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

[1]. https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

[2].https://dmalcolm.fedorapeople.org/gcc/2015-08-31/rst-experiment/how-to-use-inline-assembly-language-in-c-code.html

ModelSim Simulation

1 Week

Motivation and introduction

In this session, we will compile a C-code application and simulate the result in *ModelSim and Dlxsim.* The applications can be assembled or compiled using a compiler that is already generated by ASIPmeister in the previous session. ModelSim simulates your code binaries using the VHDL files generated from the ASIPmeister. While dlxsim only simulates the instruction one by one and it does not care about the hardware implementation of the instructions. 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 your project**
    1. You can use the same project as in the last session, and just create a separate application subdirectory for the application. You can start a fresh project as in the Session 1, but this would be time consuming.
    2. For the C application, you have to create subdirectory in the “*Application*” directory (e.g. *LoopExampleC*), and copy your application from “*/home/asip00/Sessions/Session3/6_for.c*” to here.
    3. Copy a “*Makefile*” file from the “*TestPrint*” application subdirectory to each application subdirectory.
    4. Set proper parameters and settings in “*env_settings.*
    5. For these exercises, we will be using pipeline forwarding (-pf1) option which is the default one.
    6. Make sure that you already have VHDL files and GNU tools in your project’s meister directory.
2. **Compiling and Simulating the Application**
    1. Go to your application subdirectory and type “***make clean***” clean this directory it there are previously generated files.
    2. Compile the C application using “***make sim***”. A directory “***BUILD_SIM***” is created which contains different temporary files and a .dlxsim file to be simulated in dlxsim. In this directory, the files “***TestData.IM***” and “***TestData.DM***”are the file used during the ModelSim simulation.
    3. 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*”.
    4. Simulate your application in dlxsim simulator using “***make dlxsim***”, just to verify the functionality.
    5. In your project directory, go to the “ModelSim” directory and start the ModelSim using “***vsim***”
    6. If ModelSim asks for “modelsim.ini” choose the default one like “/Software/ModelSim/ModelSim_6.6d/modeltech/modelsim.ini”

7. Open File Menu > New > Project and enter a project name (e.g. browstd32) and change the project location to the ModelSim directory in your project directory. Confirm the dialog with the OK button.

8. Choose "Add Existing File" button and browse to the meister/dlx_basis.syn directory of your ASIP Meister project and select all the VHDL files for synthesis.

9. Again, choose "Add Existing File" button and add the testbench files: tb_browstd32.vhd, MemoryMapperTypes.vhd, MemoryMapper.vhd, and Helper.vhd from the ModelSim directory of your current project.

10. [Optional] Configure the CPU Frequency for which you want to simulate your CPU, default is 50 MHz. Open the ModelSim testbench ("tb_ browstd32.vhd"), search for CLK _PERIOD, and change the value accordingly in "ns".

11. Compile the project using Compile Menu > Compile Order > Auto Generate. Every file should have a green mark behind its name, showing that the compilation was successful.

12. Run the simulation using Simulate Menu > Start Simulation. Open the work library, mark the entry "***cfg***" (that is the VHDL configuration for the testbench) in the list and press OK. That will start the simulation and you will get another two tabs attached to the Workspace window (sim / Files).

13. [Optional] To load some predefined simulation settings choose Tools Menu > Tcl > Execute Macro and select the "wave_vhdl.do" file in your ModelSim directory and press OK to load it. The wave-window is filled with certain signals that are useful to evaluate the simulation of the program execution on the processor.

14. [Optional] If you want to dump VCD file of yor design for power estimation, you can enter following commands in ModelSim command prompt:

```
VSIM > vsim -t 1ns work.cfg

VSIM > vcd file test.vcd

VSIM > vcd add -r test/dut/*
```

15. Press the button "***Run all***" to run the simulation until it aborts. At the end of a simulation the message "Failure: Simulation End" is printed to show successful end of simulation. At the simulation end, the file "***TestData.OUT***" is created in your ModelSim directory. It contains the content of the simulated memory after the CPU finished working. Therefore, if your algorithm is storing the result in the memory you can find the values here.

a) How many cycles are required to execute this program DLXsim and ModelSim?

b) What are the contents of TestData.OUT? Are these correct? First value is the stack value; next 10 words belongs to array A, then 10 words belongs to array B, and C.

c) In the ModelSim waveform window, what is the starting address of PC after the reset? Moreover, after how many cycles your "**main**" function is started? In the waveform, look at the PC and IR values.

   1. 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*".

   2. Repeat this benchmarking for all compiler optimization-levels like O0, O1, O2, O3 and O4 for both dlxsim and ModelSim.

a) Does the application is executed successfully using different optimization levels? If yes, please fill the following benchmark table. These optimizations have distinct effects on the size of the code.

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

Next Session: Synthesis and Hardware Implementation

Readings for the next session: Laboratory Chapters 6

Synthesis and Hardware Implementation

1 Week

Motivation and introduction

In this session we synthesis our basis CPU with Xilinx ISE and execute a test application on real hardware. The synthesis reports tell us how much area and power is consumed by our CPU and what is the critical path of our design. In this session, we will compile a C-code application direct its output the LCD and UART with the help of some predefined libraries. This session also introduces about different peripheral where we forward our text/data, and how different libraries are used for different peripherals. 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 your project**

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

    2. For the C application, you have to create two subdirectories in the "*Application*" directory e.g. "*Hello_SW*" and "*Hello_HW*". Copy your application from "*/home/asip00/Sessions/Session4/app.c*" to these. This is a simple example to direct a text to some peripheral devices like LCD or UART. "*Hello_SW*" is aimed for dlxsim and ModelSim simulations and "*Hello_HW*" is aimed for real hardware implementation.

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

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

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

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

2. **Compiling and ModelSim Simulation**

    1. 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*".

    2. However, for compiling it, you first need to provide the required libraries from /home/asip00/epp/StdLib to your respective application, i.e. copy "*lib_lcd_dlxsim.c*", "*lib_uart.c*", "*loadStoreByte.c*", "*string.c*" and respective header files to "Hello_SW" directory. Also, copy "*lib_lcd_320.c*", "*lib_uart.c*", "*loadStoreByte.c*", "*string.c*" and respective header files to "Hello_HW" directory.

    3. Go to your application subdirectory "Hello_SW", and type "***make clean***" clean this directory it there are previously generated files.

    4. In the directory "Hello_SW", compile the C application using "***make sim***". A directory "***BUILD_SIM***" is created which contains differ-

ent temporary files and a .dlxsim file to be simulated in dlxsim. In this directory, the files "***TestData.IM***" and "***TestData.DM***"are the file used during the ModelSim simulation.

5. Simulate your application in dlxsim simulator using "***make dlxsim***", just to verify the functionality. 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 -lflcd.out -ufuart.out" writes output to the file "lcd.out" and "uart.out" in the application directory.

6. In your project directory, go to the "ModelSim" directory and start the ModelSim using "vsim". You can use the previous ModelSim project and simulate the application. Remember to generate VCD files required for power estimation while doing ModelSim simulation.

7. 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. While ModelSim automatically writes to the file "lcd.out" and "uart.out".

a. How many cycles are required to execute this program DLXsim and ModelSim?

3. **Xilinx ISE Framework for Hardware Implementation**

   1. Go to your application subdirectory "Hello_HW", and type "***make clean***" clean this directory it there are previously generated files.

   2. In the directory "Hello_HW", compile the C application using "***make sim***".

   3. Go to the project directory and type "ise &" to start Xilinx ISE.

   4. Create new project using File Menu > New Project with following project settings:

   Project Name: ISE_Framework

   Project Path: PATH_TO_YOUR_PROJECT/ ISE_Framework

   Device Family: Virtex5

   Device: xc5vlx110t

   Package: ff1136

5. Add the design and framework files by selecting "Project Menu > Add Copy of Sources" then brows to:

   1. "PATH_TO_YOUR_PROJECT***/ ISE_Framework***" and select all the files

   2. "PATH_TO_YOUR_PROJECT/ ***ISE_Framework/IP-Cores***" and select all the files
   3. "PATH_TO_YOUR_PROJECT/ ***meister/browstd32.syn***" and select all the files

6. Select top level modules "**dlx_toplevel**", and now you can synthesize, implement and generate programming file for the design using the following respectively:
   1. Processes Menu > Synthesize XST
   2. Processes Menu > Implement Design
   3. Processes Menu > Generate Programming File
7. Once the design is implemented you can see different reports using:
   1. Processes Menu > Place & Route > Generate Post Place & Route Static Timing > Detailed Reports > Place and Route Report
   2. Processes Menu > Place & Route > Generate Post Place & Route Static Timing > Detailed Reports > Post PAR Static Timing Report
   3. Processes Menu > Place & Route > Analyze Post Place & Route Static Timing > Timing Constraints
8. In the project directory and type "hterm &" to start HyperTerminal to see the UART output if there is any output. 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,
9. In the application subdirectory and type "make fpga", it will combine the generate DM/IM file with your ISE generated bitstream. 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 will be used to configure the FPGA.
10. For hardware implementation you need to connect to i80labpc10 only. Connect your FPGA to PC the i80labpc10, power the board. In the application subdirectory type "make upload": to upload the existing bitstream to the FPGA
11. If your application does not work try to RESET the FPGA/LCD board.

4. **Xilinx ISE Framework for Benchmarking**
   1. To accurately measure the critical path and area of the ASIPmeister CPU, you can use ISE_Benchmark folder instead of ISE_Framework folder.
   2. Go to the project directory and type "ise &" to start Xilinx ISE.

3. Create new project using File Menu > New Project with following project settings:

Project Name: ISE_BenchMark

Project Path: PATH_TO_YOUR_PROJECT/ ISE_ BenchMark

Device Family: Virtex5

Device: xc5vlx110t

Package: ff1136

4. Add the design and framework files by selecting "Project Menu > Add Copy of Sources" then brows to:
   1. "PATH_TO_YOUR_PROJECT/ ISE_ BenchMark" and select all the files
   2. "PATH_TO_YOUR_PROJECT***/ meister/ browstd32.syn***" and select all the files
5. Now you can synthesize, implement and generate programming file for the design as before.
6. Once the design is implemented you can see different reports as before.

a. Reports: P&R Report > Check for "Device Utilization Summary" to see #Slices and #LUT consumed.

b. Post PAR Static Timing Report: Check for "Timing Summary" to see the minimum period and maximum frequency supported by the processor architecture.

5. **Xilinx ISE Framework for XPower Power Estimation**
   1. To accurately measure the power consumption of the ASIPmeister CPU, you can create another folder ISE_XPower.
   2. Go to the project directory and type "ise &" to start Xilinx ISE.
   3. Create new project using File Menu > New Project with following project settings:

Project Name: ISE_XPower

Project Path: PATH_TO_YOUR_PROJECT/ ISE_ XPower

Device Family: Virtex5

Device: xc5vlx110t

Package: ff1136

4. Add only design files by selecting "Project Menu > Add Copy of Sources" then brows to "PATH_TO_YOUR_PROJECT***/ browstd32.syn***" and select all the files.

5. Now you can synthesize and implement the design as before.
6. Once the design is implemented you can open XPower tool using Processes Menu > Place & Route > Analyze Power Distribution (xPower Analyzer)
7. Then in XPower Tool, select "File Menu > Open Design" and set the properties as follows:
    1. Design File: PATH_TO_YOUR_PROJECT/ISE_ XPower/ BrownieSTD32.ncd
    2. Physical Constraint File: PATH_TO_YOUR_PROJECT/ ISE_ XPower/ BrownieSTD32.pcf
    3. Simulation Activity File: PATH_TO_YOUR_PROJECT/test.vcd
8. After analyzing the activity file, the CPU power is estimated. You can see total and dynamic power of the FPGA. In addition, you can confirm that the VCD file is loaded properly by verify the clock value in XPower.

a. What is the total power consumed? What is the distribution of power i.e., dynamic and static power?

b. Why static/quiescent/leakage power is so high?

c. What is the power distribution among On-Chip **1**) Clocks, **2**) Logic, **3**) Signals, **4**) BRAMS, **5**) IOs?

Next Session: Adding Custom Instructions

Readings for the next session:

Laboratory Chapters 8.2.3, 3.2.2, 3.2.3, 4.4,

ASIPmeister Tutorial

ASIPmeister User Manual

Adding New Instructions

1 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. 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-Session4", with XX replaced by your group number.

Exercises

1. **Preparing your project**
    1. You can use the same project as in the last 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. For the C application, you have to create subdirectory in the "*Application*" directory (e.g. "arrayloop"), and copy your application from "*/home/asip00/Sessions/Session5/loop.c*" to here.
    3. Copy a "*Makefile*" file from the "*TestPrint*" application subdirectory to each application subdirectory.
    4. Set proper parameters and settings in "*env_settings.*
    5. For these exercises, we will be using pipeline forwarding (-pf1) option which is the default one.
    6. Make sure that you already have VHDL files and GNU tools in your project's meister directory.
2. **Compiling and Simulating the Application**
    1. First, you have to compile the application using gcc compiler to compare with the later results from dlxsim and ModelSim. To compile with GCC, comment the line "*#define ASIP*" and compile the application like "*gcc loop.c*". 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 a program. The file '*output_gcc.txt*' will contain the printed array.
    2. Now, compile "loop.c" using "make sim". However, for compiling it, you first need to provide the required libraries, i.e. "*lib_lcd_dlxsim"* (also for dlxsim/*ModelSim*), "*loadStoreByte"*, and "*string"*.
    3. 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 –pf1 -lf**output_dlxsim.txt***" writes output to the file "*output_dsim.txt*". *ModelSim* automatically writes to the file "*lcd.out*".
    4. 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"*.
    5. Print statements should be commented out for a fair comparison. Then simulate in dlxsim and ModelSim.

a) How many cycles do you need for execution in dlxsim and ModelSim (without printing)?

3. **Adding a new instruction to *dlxsim Simulator***

   1. Create a project directory for this session by copying the directory "*/home/asip00/epp/ASIPMeisterProjects/TEMPLATE_PROJECT/*" and renaming it (e.g. *brownieAVG*).
   2. You have to create another subdirectory for our application in the "*Application*" directory (e.g. "arrayloopAVG"). Copy "loop.c" here and then you start putting new instructions here. Print statements should be comment out for a fair comparison.
   3. But before modifying "loop.c", create a simple C or assembly file to test different custom instruction (e.g. AVG) into an application subdirectory i.e. *TestAVG*.
   4. Copy a "*Makefile*" file from the "*TestPrint*" application subdirectory to each application subdirectory.
   5. Copy the provided *ASIPMeister* CPU file "*browstd32.pdb*" from "/home/asip00//Sessions/Session1/" into your project directory, and rename it "*browstd32AVG.pdb*"
   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=brownieAVG

   export CPU_NAME=browstd32AVG

   export ASIPMEISTER_PROJECTS_DIR=${HOME}/ASIPMeisterProjects

   export DLXSIM_DIR=/home/asip00/epp/dlxsimbr_Laboratory

7. 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*", "*rot rd, rs0, amount*" 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

   {"avg", ARITH_3PARAM, 0x6c1, 0x1ffff, 0x20, 0 , 0xffff8000 },

   {"swap", ARITH_2PARAM, 0x541, 0x1ffff, 0x20, 0 , 0xffff8000 },

   {"minmax", ARITH_4PARAM, 0x981, 0xfff, 0x20, 0 , 0xffff8000 },

   {"rot", ARITH_3PARAM, 0x8, 0x3f, 0, CHECK_LAST|IMMEDIATE_REQ, 0xffff8000 },

{"bgeu", BRANCH_2OP, 0x11, 0x3f, 0, 0 , 0 }***

*** Not needed in this session.

8. 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).
9. Write a small assembly code to test your new instructions in dlxsim. You can use the Session2 assembly language program as the reference.

4. **Extending the CPU with a custom instruction**

    1. In your new CPU, implement the new instruction "*avg rd, rs0, rs1*", "*swap rd, rs*", "*rot rd, rs amt*" 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:

1. You **can use the opcode and instruction formats as indicated in the figure below.**
2. 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.*
3. For instructions that return two values, you need to change # of GPR write ports. Normally, only one register is forwarded from EXE or WB stage, now you have to add new resources (Forwarding units) to forward two register values.
4. 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).
5. Remember that your new instruction has to support forwarding as well.

2. Generate the VHDL Files.
3. 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. Follow instructions at section 4.11.C in ASIPmeister tutorial and 11.2 in ASIPmeister user manual.
4. Generate GNU Tools for your new processor.
5. **Compiling and Simulating the Application with custom instructions**

    1. You can test your custom instructions with small assembly programs.

2. After testing the new instruction with a small assembly code, use *inline assembly* in the application "*loop.c*" for using the new instructions, see Chapter 8.2.3 in the Laboratory Script.
3. There are two methods to insert a explicitly insert a custom instruction inside your C code:
    1. Using ___builtin_brownie32___ABC directive: This method has a bug while using with the instruction that has zero return values or instructions that return more than one value.

Examples:

Int a,b,c,d;

c = ___builtin_brownie32_AVG(a,b);

d = ___builtin_brownie32_SWAP(a);

2. Using ___asm___ directive: This can be used for all the cases.

Branch Instruction

Int a,b,c,d,e;

```
___asm___ volatile (
"bgeu %[my_op1], %[my_op2], _here2\n"
"_there2: sub %[my_out], %[my_op1], %[my_op2]\n"
"_here2: add %[my_out], %[my_op1], %[my_op2]\n"
: [my_out] "=&r" (e)
: [my_op1] "r" (a),[my_op2] "r" (b)
);
___asm___ volatile (
"minmax %[my_out1], %[my_out2], %[my_op1], %[my_op2]\n\t"
: [my_out1] "=&r" (c), [my_out2] "=&r" (d)
: [my_op1] "r" (a), [my_op2] "r" (b)
);
```

4. 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.

5. 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!

6. Simulate the application with ModelSim.

a) How many cycles do you need for execution in dlxsim and ModelSim? Attach assembly programs you used to test custom instructions with this mail.

b) What is the speedup (i.e. #Cycles without custom instructions / #Cycles with custom instructions)?

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

d) Attach the modified ASIPmeister .pdb file.

Next Session: Bubble Sort – Simulations and Optimization

Readings for the next session: All the Chapters, especially 4 & 8, ASIP-Meister Tutorial & Manual

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. 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.

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. 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.

2. 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.

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.

2. **Preparing your project**

   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. 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.

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

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

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

   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**

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

   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.

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

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

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

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

      1. Create a new application sub-directory (e.g. "sortingOptS") and copy "sorting/BUILD_SIM/bubble.s" 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.

      2. 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.

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. Copy a "*Makefile*" file from the "*TestPrint*" application subdirectory to each application subdirectory.
5. Copy the provided *ASIPMeister* CPU file "*browstd32.pdb*" from "/home/asip00//Sessions/Session1/" into your project directory, and rename it "*browstd32OPT.pdb*"
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.

1. You also have to define "CKF Prototype" for each new custom instruction in ASIPmeister, generate GNU tools and use inline assembly in C code.
   1. Add the custom instructions to dlxsim as well.
   2. Write a small C or assembly code to test your new instruction.
   3. 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.
   4. If everything is working fine, then simulate the *BubbleSort* C/Assembly code that uses the new instruction in *ModelSim*.
   5. Compile the optimized bubblesort application using "*make sim*" and then "*make dlxsim*". Make sure, that the resulting array is still correct.

a) How many cycles do you need for execution?

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

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

d) 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

Bubble Sort – Power & Area Estimation and Hardware Implementation

1 Week

Motivation and introduction

In this exercise, you will synthesize and implement the bubblesort application and then download it to FPGA board and see the results on the UART terminal or LCD. For visualizing, the output of BubbleSort and some additional information is printed to the URAT interface. You can use t_print() for directing output to LCD or u_print() to UART. Remember, you need to add respective libraries. Using these frameworks, the Bubble sort algorithm which will be implemented using the two CPUs to form two versions:

Version1: basis CPU (*browstd32.pdb*)

Version2: optimized CPU (*browstd32opt.pdb*) which supports new instructions

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

Exercises

1. **Preparing and Simulating Version 1**

   1. 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”*. Remember, for LCD interface you need to create two separate subdirectories for the software and the hardware application in “Applications” directory.

   2. Copy the provided and “*app_UART.c”* from “/home/asip00/Sessions/Session7” to the created LCD or UART subdirectories respectively. This file directs the printing to UART, you can change it to LCD by replacing u_print() to t_print(). However, UART interfacing is sufficient.

3. Copy the C libraries from "*asip00/epp/StdLib*" to each application subdirectory. For LCD simulation in dlxsim and ModelSim, use "lib_lcd_dlxim.c". For real LCD implementation in FPGA use "lib_lcd_320.c".

4. Also, copy the "*Makefile*" to both the subdirectories.

5. Make sure that you already have generated the VHDL files and GNU Tools for your CPU.

6. Compile ("*make sim*") the application for basis CPU and generates the required .dlxsim and DM/IM file for the dlxsim and ModelSim respectively.

7. Simulate the application with dlxsim using "*make dlxsim DLXSIM_PARAM="-da0 –pf1 –ufBubbleUART.out"*". 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/UART file to print the outputs.

8. 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.

9. 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.

2. **Implementing the Project**

   1. Create your ISE project as discussed in the session 4. Synthesize, implement and generate the bitfiles.

   2. Then from the application direcotory run "make fpga" and "make upload".

   3. You can check the results on the UART. You can open HyperTerminal using "hterm &".

a) If your design works correctly, find out the design statistics (critical path, maximum frequency and area)

b) 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).

c) Analyse the time and find the critical path (see Chapter 6.5 of the Laboratory Script)

3. **Power Estimation**

    1. During ModelSim simulation also generate the VCD files for mentioned frequencies (first with 50MHz and then with Max. Frequency found in the last session).

    2. Create Xilinx ISE project to estimate power with XPower.

a) Determine the total and dynamic power.

b) Compute the total execution time (ms). You can use execution as the # of cycles multiplied by the clock cycle in ModelSim.

c) Compute the energy required. Fill in all these results in the table below e.g. *PowerReport.xlx* or *PowerReport.ods*.

d) Does using any instruction minimize the required energy? A version uses an application, which needs less number of clock cycles than another Version; is it also power and/or energy-optimized version compared to Version2?

e) 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_brownie32std.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.

4. **Preparing, Simulating, Implementing and Power Estimation for Version2**

    1. Repeat the above exercises for the optimized version a), b), c), d), e).

    2. Sample PowerReport.xlx or PowerReport.ods

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

Next Session: Adaptive Differential Pulse Code Modulation (ADPCM)

Readings: Recall relevant information from Laboratory Script, ASIPmeister Tutorial and User Manual

Adaptive Differential Pulse Code Modulation (ADPCM)

3 Weeks

Motivation and introduction

This is the final session. You have **Three** 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 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:**
    1. The application is the ADPCM audio decoder.
    2. 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).
    3. 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).
    4. 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. For dlxsim/ModelSim simulation comment the while(1) loop, while for FPGA implementation use while(1) loop without any print statement inside it.

5. 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.
6. 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.
7. You can change the CPU frequency by using the knob 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.
8. 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).

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

Table-1: Frequency Changing

2. **Your Tasks**

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

1. To make your optimization comparable with other groups start with the browstd32 CPU provided in Session 1, and test ADPCM application. Do not take any CPU that you already have modified).
2. Compile the MINI version of the application; simulate it with dlxsim/ModelSim.
3. Then compile the BRAM version and run it on the FPGA prototype.
4. 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.

5. Improve/Extend the CPU for speed, power or area. You have to create two different versions based on your improvements. You should have at least two extensions (power, performance or area).
6. Test the improved CPU version on dlxsim/ModelSim and on FPGA
7. Benchmark the basis and improved CPUs for area, frequency, power, execution time etc.
8. Prepare slides that explain your modifications, improvements and results to compete with other groups.
9. 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.

3. **Simulating the Application**
    1. 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*".
    2. To compile the application, audio data is needed for decoding; therefore, you have to copy 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 copy the required audio data *adpcmDataStereo_MINI.h or adpcmDataStereo_BRAM.h* into subdirectory as *adpcmData.h.* The compilation will take some time due to the large audio data. For short tests, e.g. to test whether the inline assembly code compiles and assembles or whether ModelSim simulation gives the correct output, use the "*adpcmDataStereo_MINI.h*" version of the file.
    3. 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.
    4. 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*".
    5. Copy the required libraries from "*/home/asip00/epp/StdLib*" to the application subdirectory and compile ADPCM application using "make sim".
    6. 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).

7. 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!

4. **Running the Application on FPGA**

   1. 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).

   2. 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.

   3. 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.

   4. 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.

      1. 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.

2. If your application runs too slow, then the FIFO will become empty, resulting in errors in the audio stream. Try it!

3. 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.

5. **Extending the Basis CPU**

   1. 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.

   2. 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.

   3. 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.

   4. **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.

   a. Attach to mail your both ASIPMeister CPU optimized for area/performance/power named like "*browstd32Area.pdb*", "*browstd32Power.pdb*" or "*browstd32Speed.pdb*" etc.

   b. Attach to mail if you have defined new hardware resources (.fhm files) in ASIPMeister.

   c. Attach with the mail your adpcm.c, which you modified with your custom instructions, for the two CPU optimizations.

d. Attach the test application files that you created to test new instructions.

6. **Benchmarking the CPUs**

    1. **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:**

        1. **Make sure, that you do all benchmarks very accurate to make them comparable!**
        2. **Always use MINI version of the application**
        3. **Always compile with "-O3" to achieve the best compiler output. Note: For debugging purpose "-O0" (default) is recommended.**
        4. **Always using ISE_Benchmark framework for the area, power and critical path analysis**
        5. **Always take execution time or number of cycles from ModelSim**
        6. **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**

        7. **You have to configure the frequency in the ModelSim testbench for power estimation. Remember that you have to configure the half clock period.**

    2. **You have to benchmark and compare the following points:**

        1. **CPU Area**

   2. **Maximal CPU frequency or Critical Path**
   3. **Number of Cycles or Execution Time**
   4. **Dynamic Power Consumption**
   5. **Energy Consumption**

7. **Presenting the Results**
   1. **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:
   2. **Explain the two different CPUs that you have created to optimize ADPCM application.**
   3. **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.**
   4. **Discuss your benchmark results; for every point you should have one slide on which the results are shown in a graph (bar graph, lines . . .).**
   5. **Print proper units your axes e.g. "Execution time [s]", "Execution time [cycles]", "Power consumption [mW]", . . .).**
   6. **For every** measurement point, print the value of this measurement result to make comparisons easier.

a) You have to mail the slides before the presentation. Name the slides like "asipXX_presentation.ppt" (or ".odp" or ".pdf").

b) You have to explain which benchmarks you used while customizing the processor and compare in the presentation.