ADPCM: Adaptive Differential Pulse Code Modulation

Motivation and introduction

This is the final session. You have **three** weeks to complete this session, but you will need these three weeks! In this exercise, you will work with a new 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 **asip00@ira.uka.de** and use the topic "asipXX-Session2", 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/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 size. 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; 2 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

2) 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 dlx_basis 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.
- Note: We will not create a new compiler for the ADPCM ASIPs. Typically, we will not simulate in dlxsim but mainly in ModelSim. May be for the extended CPUs you have to generate new compiler using "*makeCoSy*" and "*contCoSy*".

3) Simulating the Application

1. Prepare your new project directory and create a subdirectory in your "*Applications*" directory and copy "*/home/asip00/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 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/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.
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. Link the required libraries from *"/Software/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

8. 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).
9. 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.
10. 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.
11. 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.

5) Extending the Basis CPU

12. 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.
13. 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.

14. 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 with "*contCoSy*".
15. **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 "*dlx_area.pdb*", "*dlx_power.pdb*" etc.
    b. Attach to mail if you have defined new resources (.fhm files) in ASIPMeister.
    c. 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

16. Make benchmarks for the old (*dlx_basis*) 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:
        i. 50 MHz; to make it comparable among the groups
        ii. 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.
        iii. 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.
17. 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

7) Presenting the Results

18. 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.
        a) You have to mail the slides before the presentation. Name the slides like "asipXX_presentation.ppt" (or ".odp" or ".pdf").