ADPCM: Adaptive Differential Pulse Code Modulation

Motivation and introduction

This is the final exercise. You have three weeks to complete this exercise, 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 Meeting Room in the second stage (exact date to be discussed).

The application:

- The application is the ADPCM audio decoder.
- The term PCM would denote uncompressed audio samples (Pulse-Code Modulation) and ADPCM is using an adaptive prediction (Adaptive Differential PCM) 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 will find the source code for ADPCM decoder with some ADPCM encoded audio data in ~asip00/Session8. When you run the ADPCM decoder, you will receive the (nearly) original audio samples.
- We provide two different versions of the encoded audio data, differing in size. The MINI version is meant for very first 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. The BRAM version is the biggest possible version that fits into the FPGA-internal memory (called BlockRAM).
- 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 96.000 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 connected to the FPGA board. At the end of this scrip, you can find a table describes the 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 fine hearable without corruption).

Your tasks (details are given below):

- Take the dlx_basis CPU (provided in Session 3; we want to make it comparable, so do not take one of your already modified CPUs) and test the application. Compile it, simulate it in ModelSim (MINI version), and run it on the FPGA prototype (BRAM version). 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 on frequency to the other, then the slower one was fast enough.

- Improve the CPU (create two different versions).
- Test the improved CPU (ModelSim & FPGA).
- Benchmark the CPUs (area, frequency, power, …).
- Prepare slides that explain your modifications and results and present them to the group.
- Note: We will not create a new compiler for the ADPCM ASIPs. Typically, we will not simulate in dlxsim but mainly in ModelSim.

Simulating the application:

- Add the required libraries from /Software/epp/StdLib and add a copy of the *mkimgSettings* file
- 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 (1=yes). Printing to LCD/UART is meant for debugging on the FPGA. In dlxsim and ModelSim you can additionally see the data that shall be sent to the audio output (note: use /Software/epp/dlxsim, not the dlxsim version that you created as you did before in the previous version). 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).
- 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!
- To compile the application with *mkall* you have to create a link to the audio data that shall be used for decoding. We have prepared three different-sized versions of the same audio stream. To create the required link you have to execute e.g. "`link -s adpcmDataStereo_BRAM.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.

Running the application on the hardware prototype:

- To test whether the decoder is working correct with the base CPU and with your modified CPU, you will run the application on the FPGA prototype (test it with ModelSim first).
- 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.
- The hardware will automatically give 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 changed in the file dlx_Toplevel.vhd (KSAMPLES_PER_SECOND) and is set to 96). This is the correct sample rate for the provided audio data.
- 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.

Extending the CPU:

- You may add new instructions to speed up frequent computations.
- If your custom instruction delays to clock too much, then you can change it into a multicycle instruction (i.e. an instruction that is allowed to stay in EXE stage for multiple cycles, similar to mult). Details about multicycle fhm are given in the script.
- 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.
- 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 e.g. 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.
- **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 (only considering the cycles in ModelSim due to the FIFO delay (see below)) or the power/energy (due to a reduced clock frequency that is possible due to a faster computation) or that is optimized for area, e.g. by removing not required instructions/hardware blocks etc.

Benchmarking:

Make benchmarks for the old (*dlx_basis*) and the two modified CPUs and compare them with each other. You will have to present your results. You have to benchmark and compare the following points:

- the CPU area
- the maximal CPU frequency (find and show the critical path)
- the number of cycles and the time for executing the application (ModelSim & FPGA)
- the power consumption while executing the application
- the consumed energy for executing the application
- How to benchmark?
  - Make sure, that you do all benchmarks very accurate to make them comparable!
  - Compile with "-O3" to achieve the best compiler output. Note: For debugging purpose "-O0" (default) is recommended.
  - For CPU area and frequency, you have to use the ISE Benchmark Framework.
  - For getting the cycles, use the Clock Counter library as described in Chapter 8.5.2.
  - For the ModelSim simulations, configure the mkimgSettings to SW and use the lib_lcd for dlxsim.
  - For cycles, power, and energy use the application data (Mini).

- For execution time, power, and energy use the following three CPU frequencies:
  - 50 MHz; to make it comparable among the groups
  - the slowest frequency that is sufficient to execute the application fast enough; to see the lowest power consumption
  - the fastest frequency that your CPU supports (reported by ISE); 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.
  - 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.

Presenting the results:

In the last week of the semester, you will present your results to the other groups. Therefore, every group has to prepare slides:

- Explain the two different CPUs that you have created to support adpcm.
- Present the problems that you had 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 the units that your axis are showing (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.
- You have to mail the slides to amrouch@kit.edu and Lars.Bauer@ira.uka.de before the presentation. Name the slides in the following scheme: “asipXX_presentation.ppt” (or “.odp” or “.pdf”).

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

Table. Frequency changing