CSE 465M Digital Systems Laboratory
Assignment #1

Due: Friday, Sept. 15, 2005.

The purpose of this assignment is to familiarize you with the software development environment that we will be using this semester. You will write a small program and observe its operation on a processor loaded onto the FPGA.

The program will decode audio data that has been compressed. Yes, you are building an iPoD, just not quite to the specs of the real thing.

The compression algorithm uses the following pipeline of functions (remember, your job is decompression). Each stage will be described in detail below.

Audio samples are first aggregated into frames of size FRAME_SIZE (a compile time constant). All decisions and encoding downstream from the framer operate on complete frames. An encoded audio frame consists of a header (definition below) followed by the encoded audio samples themselves. When the number of audio samples is not an integral number of FRAME_SIZE, additional samples (all equal zero) are padded to the end of the audio sample stream to fill the last frame.

The header starts with a frame delimiter synchronization bit pattern (16 bits of all 1) followed by a number of fields that describe properties of the frame. The specification of the individual fields is given in the file “audio.h” on the class web page. Note that in the type t_header, the fields are contiguous.

The delta encoder replaces each sample with the difference between successive samples. A baseline value is stored in the frame header. The first sample is represented by the difference between the raw sample and this baseline value. The second sample is represented by the difference between the first and second raw sample, etc.

The scaling stage ensures that each sample (now delta encoded) fits in 8 bits (i.e., fits in the range -128 to +127). This is accomplished by dividing each sample by a common value chosen from a fixed set of choices made available in a table. Which scaling value chosen is indicated in the frame header (actually what is stored in the frame header is the table index).

The variable length coder replaces the fixed-length, 8-bit samples that come out of the scaling stage with variable length samples (as in a Huffman code). The coding table is constant, but will likely change in the future (maybe even before the assignment is due)! The encoding table
will be provided, you will be responsible for building the appropriate decoding data structures and algorithm.

At the end of the variable length codes, enough bits of 0 are added to ensure that the samples comprise an integral multiple of 4 bytes.

The error check generate block computes the 32-bit checksum of the variable length sample data and an independent 16-bit checksum of the frame header. The frame header checksum excludes the checksum field itself.

Your software decoder will be required to reverse the above process, generating audio samples from a series of encoded audio frames. After the audio samples have been output, the count of error check errors should be reported, to a distinct output stream if feasible (e.g., stderr). The organization of the code must contain modules (subroutines) for at least the 4 primary functions corresponding to the above pipeline’s delta encoder, scaling, variable-length coder, and error check. Good design practice likely requires additional decomposition.

Once your program is written you will exercise it using an unmodified LEON processor. Input encoded audio will be via compile time initialized data contained in a file called “inputXX.h” where XX is replaced by a two-digit value (e.g., input01.h, input22.h). Output is to be delivered to stdout in a form that can be readily viewed. When executing, collect a screenshot or other visual output that indicates success.

Choose one of the subroutines in your program for further investigation. Print the assembly language version of the routine, and identify the mapping between C source and assembly language. Using the debugger, set a breakpoint at the beginning of the subroutine and single step through the instructions of the subroutine. Examine one or more variables as well. What is the physical address of those variables? Ask the debugger for a disassembled listing of your subroutine. What is the physical address of the subroutine entry point?

Use the debugger to measure the execution time of your program.

Turn in a copy of your source, assembly output annotated with the source, memory map, each of the collected outputs, and enough description that we can follow what you gave us.