CSEE 4840
Project Presentation

Audio Sampler

Spandan Das (sd3506)
Avik Dhupar (ad3910)
Agenda

- Project Overview
- System Architecture
- Hardware design
- Software design
- Challenges
Overview

● Goal
  ○ Create an audio sampler and playback system that supports:
    ■ MIDI input using a USB-MIDI Keyboard
    ■ Sample storage
    ■ Sample Playback
    ■ Sample Playback with undersampling
    ■ ASR amplitude envelope

● HPS
  ○ Reads input from USB-MIDI keyboard and passes to FPGA
  ○ Loads sample from HPS to FPGA
  ○ Allows changing envelope parameters

● FPGA
  ○ Loads WT from HPS and saves in BRAM
  ○ Drives & configures CODEC
  ○ Reads BRAM data, applies any algorithms, and outputs to CODEC
System Architecture

- Input

USB-MIDI Keyboard

MIDI over USB
System Architecture

- Input
- Input Processing
System Architecture

- Input
- Input Processing
- Data Transfer to FPGA
- Processing on FPGA
System Architecture

- Input
- Input Processing
- Data Transfer to FPGA
- Processing on FPGA
- Audio output
Hardware Design: SV Modules

- **I2C_programmer**
  - Implements I2C and writes configuration to WM8731
- **Hpsfreq:**
  - Implements 3x RAM to accept and store wavetable from HPS
- **Serial_dac**
  - Accepts 33bit word and outputs 33 bits serially to the CODEC
- **Key_parser**
  - Accepts input from HPS, parses & outputs MIDI keypress
- **Summer**
  - Sums 3 signals to allow polyphony
- **Asr_envelope**
  - Implements Attack, Sustain, Release envelope
- **Audio**
  - Binds all other modules together
  - Implements logic and routing of internal signals
Hardware Design

- Reuse I2C driver and WM8731 driver written fully in FPGA-land
  - Someone else's code almost always doesn't do what we want it to do
  - The existing WM8731 driver code simply enabled mic bypass, and claimed to be a system which reads mic input to FPGA and writes this data to line-out
- Modified the I2C driver to configure the audio codec to accept digital data
- Samples incoming from the HPS are stored in the BRAM of the FPGA
- There are 3 such samples stored in memory.
- Audio Sampling
  - The audio.v module reads samples from the stored memory
  - Undersampling implemented on memory read addresses
  - The address is calculated using fractional values in verilog (MSB represents the addr and LSB represents the fractional increment)
Hardware Design

- The summer module takes 3 inputs and gives a amplitude adjusted summed output
  - The outputs are dependent upon the key presses coming from the MIDI keyboard

- The ASR envelope module takes the input from the summer module to implement the final output
  - Attack
  - Sustain
  - Release
Hardware Design

- The ASR envelope state machine:
HPS software is divided into 2 parts:

- **USB userspace program**
  - USB-MIDI keyboard is read using libusb
  - Incoming data is parsed and on keypress, if the key is between G2 to C#5, then values ranging from 0x01 to 0x1F are sent over a socket

- **Memory mapped drivers+Userspace program**
  - Data received over socket is written to the memory mapped device, routed through the drivers
Challenges, Resolutions & Lessons

● Only X number of samples being played back despite changing the value of parameter in sv code
  ○ Professor, Oscilloscope & RTL viewer to the rescue
  ○ Quartus UI saves the parameter values internally and doesn’t seem to update them despite the change of parameter values in the code
    ■ Is this a bug? Do we not understand Quartus workflow?

● Getting sine output with audible harmonics
  ○ Oscilloscope to the rescue
  ○ There was discontinuity in the data, and only 500 of 512 samples were being played

● Reworking large parts of verilog modules to accommodate more features
  ○ Define and architect the system well, before writing a single line of code

● A one-off multiplication or division by an odd number in verilog doesn’t hurt (yet)
Challenges, Resolutions & Lessons

● I2C drivers and ensuring the I2C device is configured as required
  ○ Simulate, simulate, simulate… but we were late to realize this…
  ○ Use SignalTap
  ○ Use a $10 logic analyzer (it decodes the protocol too!)
● Ensuring the WM8731 does what we want it to do, i.e. output the correct wave
  ○ Turns out to be an incorrect Master CLK being fed to the WM8731
  ○ Professor & oscilloscope to the rescue
● Multi-port RAM to read more than two samples from memory at one point in time to enable polyphony
  ○ Limit size of each wavetable to 32k and write the same wavetable to 3 distinct RAMs
  ○ Ideally, this could be done by scheduling access to the same memory blocks since audio output is only generated at 48KHz as compared to the 50MHz CLK of FPGA