Bass Active Crossover Filter

Group M4

Albert Hwang
John Kuhns
Adrian Loh
Andrew Ryan

Carnegie Mellon University
May 2, 1997

1 Bass Active Crossover Filter

- Advanced IIR (Infinite Impulse Response) DSP
  - 44.1 to 48 kHz sampling frequency
  - 8th-order Chebyshev Type II filter algorithm
  - Implemented using cascaded 2nd-order sections
  - ±0.5 dB pass-band ring error
  - ±0.5 dB combined low-/high-pass gain error
  - 48 dB/octave roll-off rate
  - More than 80 dB of stop-band attenuation
  - High precision 20-bit filter coefficients
  - Selectable low-/high-pass filter
  - Selectable cut-off frequencies: 80 Hz, 120 Hz, or 160 Hz
- Innovative microarchitecture
  - 16-bit external I/O for high-fidelity applications
  - 32-bit internal datapath for protection against internal overflow
  - Over 700,000 32-bit serial multiply & accumulates per second
  - Serial I/O for reduced pin count.
- IC Technology
  - 2.0 µm n-well CMOS process
  - 2.3 mm x 2.3 mm die size
  - 10,310 transistors (not including the pad frame), 320 λ^2/transistor
  - 5 V operation
- Applications
  - High-fidelity car stereos
  - Home theatre systems

The Bass Active Crossover Filter is the perfect DSP solution for today's high-fidelity audio systems. The DSP allows an amp or pre-amp to take a CD-quality digital signal and either attenuate all non-low frequencies, or all low frequencies. Using this chip in a preamp to split a digital audio signal will help greatly keep the lows clean and the highs crisp in a speaker system.

Our chip leverages the speed of the IC technology to give the user high precision (16-bit) along with high speed (up to 48 kHz samping rate). This combination equates to high fidelity and optimal audio performance.

2 Design

C Code

We first sought out a DSP algorithm that would implement a filter with the desired characteristics. On the Internet, at *http://www.ece.rutgers.edu/~orfanidi/intro2sp.html*, we found Matlab code that would implement a Chebyshev Type II filter and its coefficients. After considerable coding, we were able to write a C program called **DSP_FREQ** (see Appendix A) that would output the gain and the phase shift of the output of the DSP compared to its input at any frequency. We then graphed the output using Excel (see Appendix B).

Micro-architecture

Next, we designed an architecture that would allow us to do the necessary 32-bit multiplies and additions. One complication is that multiplication must be done in sign and magnitude format, while addition must be done in two's complement format. So on our chip we store the SRAM intermediate values and the ROM coefficients in sign and magnitude format, so they can be multiplied directly. Then when the value from temporary register *dff_t* (which holds the running sum of the previous terms) is added, the product is switched to two's complement on the fly.

Over ninety-five percent of the time the datapath is doing a multiply and accumalate operation (or MAC). To do a MAC, the first step is to multiply the 32-bit SRAM value from *dff_s* times the 0th bit of the ROM coefficient (which comes in on the *a2_sel* line) and add the product to 0. The next eighteen steps is multiplying the SRAM value times 1st through 18th bit of the ROM, shifting the output to the right by one bit each time. In the last step, the 19th bit of the ROM and the 31st bit of the SRAM is used to calculate the sign of the product. If the product is negative, it is transformed into its two's complement representation, and then added with the value from *dff_t*, and then stored back into *dff_t*, all in one clock cycle.

So in twenty cycles, a 32-bit SRAM value from *dff_s* is multiplied by a 20-bit ROM coefficient, and added to the 32-bit value from *dff_t*.

Another consideration in designing the microarchitecture was to isolate the SRAM from the rest of the datapath using DFFs. This simplified timing.

Since both input and output is serial, we designed the datapath with shift registers both on the input and output. Both input and output operate on the least significant bit of the sample first.

A diagram of the datapath of our circuit is on the following page.

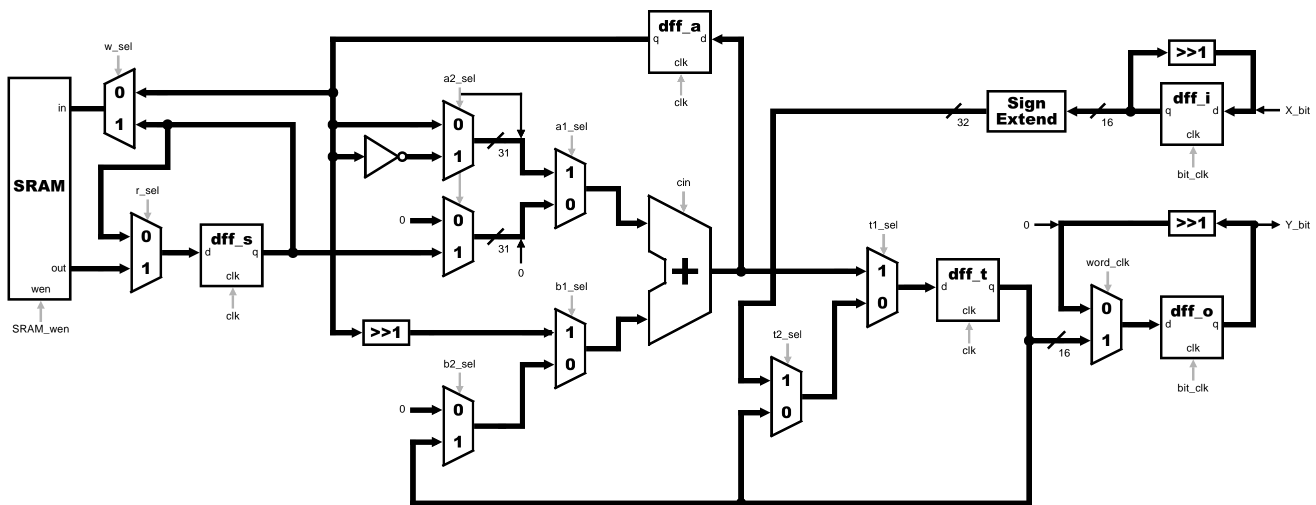


Figure 1. The datapath of the DSP

Verilog

Next, we wrote a Verilog simulation of our chip (see Appendix C). We then went back and modified our C code so that is would reflect the precision and rounding behavior of our microarchitecture. Eventually, we were able to get the Verilog and the C to output the exact same outputs given the same inputs. We tested both positive and negative vector runs.

After the successful implementation of our design in Verilog, we took the "glue" logic, the logic which controls the state ROM, and optimized it using Synopsys (see Appendix D). It had approximately 150 transistors of random logic.

Magic

To make a long story short, we implemented our Verilog code in Magic. We first floorplanned (see Appendix E) and then layed out out the SRAM, the two ROMs, the datapath, and the four counters. We then layed out the random "glue" logic and did global routing. Because we knew our C code and our Verilog code worked, we stuck close to our Verilog model when we layed out our chip (see Appendix F).

One notable innovation that allowed us to have such a high transistor density was our use of what we called "channels" in our datapath (see Appendix G). We very carefully rationed out all the Metal 2 routing on top of the datapath before we layed out the datapath. This careful planning allowed us to totally eliminate all corner routes of our 32-bit data bus.

3 Testing

IRSIM

We quickly began to test our layout using IRSIM, a digital simulator. After fixing a few misroutes and floating nodes, the vectors that we had run through C and Verilog came out of IRSIM correctly, too. One notable "hack" that we did was to add two sets of buffers infront of our adder in the datapath to get IRSIM to work. It did not seem to like the MUX-MUX-XOR line we had in our datapath; the values in the offending gates oscillated wildly, often changing values faster than 100 ps. Once we added buffers, everything worked as expected. But because of spatial constraints on our chip, we could not permanately add the buffers to our chip. So technically, the chip that we got working under IRSIM is not the same chip that we claim that works. But we are confident that chip will work, with or without the buffers.

HSPICE

Before all our IRSIM testing, we had already begun testing many of our components for correctness and speed using HSPICE, a much more accurate simulator (see Appendix H). We tested the SRAM, the two ROMs, the 4-bit adder unit, and the counter unit. We also verified that our datapath critical path was not going to be the chip's critical path. On the datapath's critical path is the muxes (14 ns), the 32-bit adder (37 ns), and the DFFs at the end (3 ns), which equals 54 ns. Our clock speed is 16 MHz, so the clock period is 62.5 ns.

4 Conclusions

We have all confidence that when this IC is fabricated, it will work exactly as designed. We tested the chip for correctness using IRSIM, and for its critical path using HSPICE. We were able to listen to sample sound clips outputed from our DSP implemented in C. We would get the exact same results from Verilog and from the fabricated IC. The Bass Active Crossover Filter lives up to our claim of it being a high performance audio processor.

A Appendix: C Code

There were three main C programs that were heavily used in the design and implementation of our chip:

DSP_FREQ This program calculated the rms gain and the phase shift of the DSP as a function of frequency. DSP_IIR.CPP and DSP_IIR.H are linked files.

BUFFER This program found optimal buffer sizing, given a timing constraint, for any inverter configuration.

SMASH This program was used to compress the the width of the state ROM from 17 bits to 8 bits using a minimum number of extra logic gates.

B Appendix: Frequency response plots

There are a total of fiveteen plots, five for each of the three selectable frequencies. For each frequency, there are gain and phase plots for a low-pass filter, gain and phase plots for a high-pass filter, and a gain plot for the low-pass and high-pass filters "added together."

C Appendix: Verilog code

The Verilog code consists of two files:

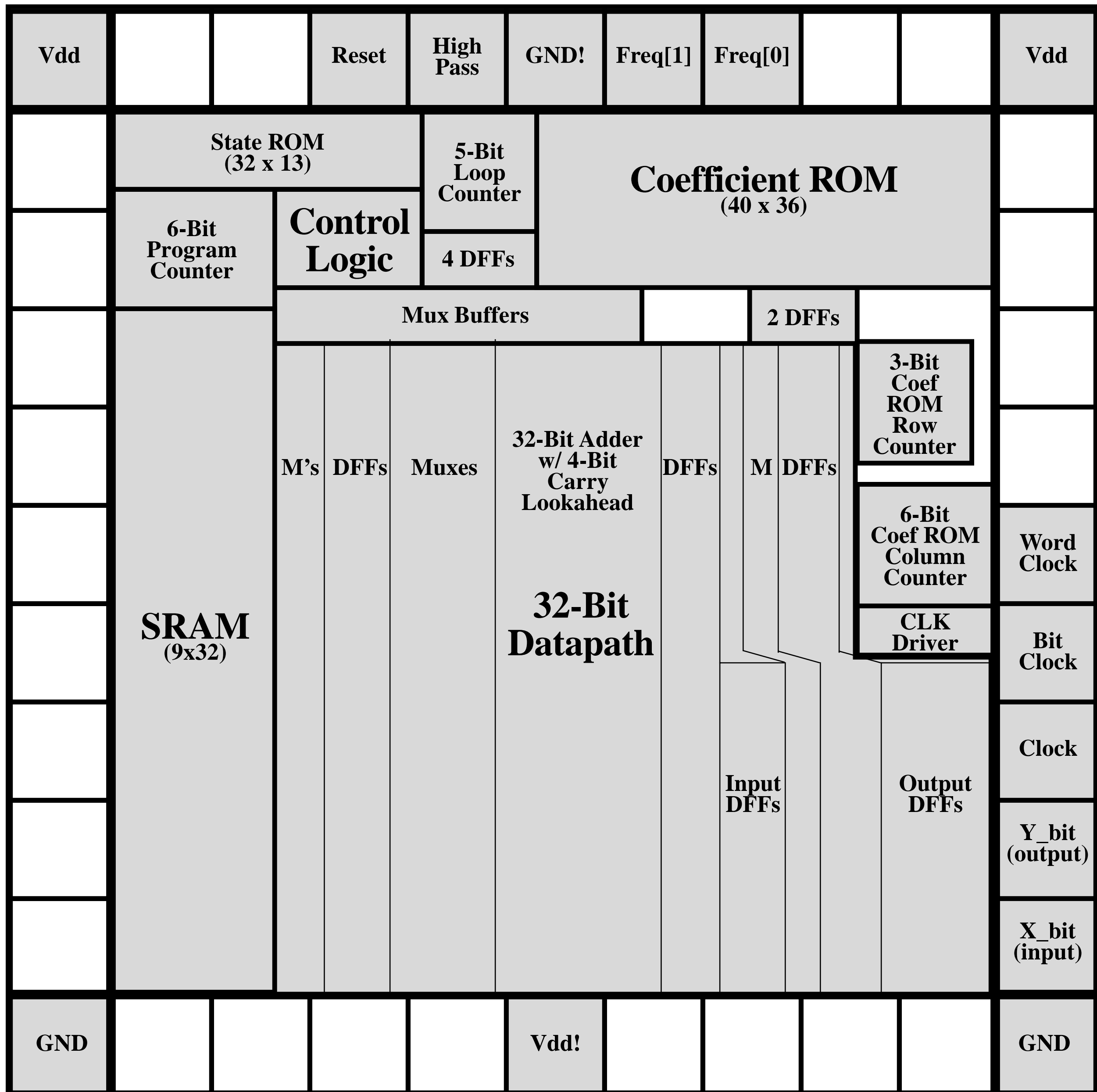
DSP This is the implementation of the chip in Verilog.

DSP_TEST This either inputs a positive or negative run of vectors through the DSP for testing purposes.

D Appendix: Synopsys schematic

On the next page is a Synopsys-synthesized schematic of the "glue" logic, the random logic of the DSP.

E Appendix: Floorplan



F Appendix: Layout

G Appendix: Channels

H Appendix: HSPICE output

On the following pages are HSPICE outputs. They should be self-explanatory. The first page is the clock waveform with a large load on it. The second page shows the timing of the coefficient ROM. The third and fourth pages show the timing of a large number of muxes.. The fifth page shows the timing of the 32-bit adder.