Everton Lincoln Uehara - 14131889

Conversor Analógico-Digital (ADC) Delta-Sigma

Este relatório traz o resultado da implementação de um ADC com duas arquiteturas de filtros digitais. Foi escrito um *testbench* no qual utiliza-se valores lidos de um arquivo CSV como valores de entrada do ADC, e a sua saída é escrito em outro arquivo CSV, e depois do processamento faz-se a comparação entre os dois sinais. Neste trabalho foi utilizado a ferramenta Octave para gerar e comparar estes sinais.

O código e a teoria foi baseado principalmente no material que a fabricante Lattice forneceu neste link.

Arquitetura de um ADC Delta-Sigma

Em suma, no ADC Delta-Sigma um sinal analógico de entrada é sobreamostrado (*oversampling*) e convertido para um valor digital. De dentro da arquitetura do ADC um sinal PWM de feedback é utilizado para ser comparado ao sinal de entrada. O streaming de bits da saída do comparador de 1 bit é armazenado em acumuladores, e sobre estes acumuladores são aplicados filtros digitais para obter o valor digital equivalente ao do analógico.

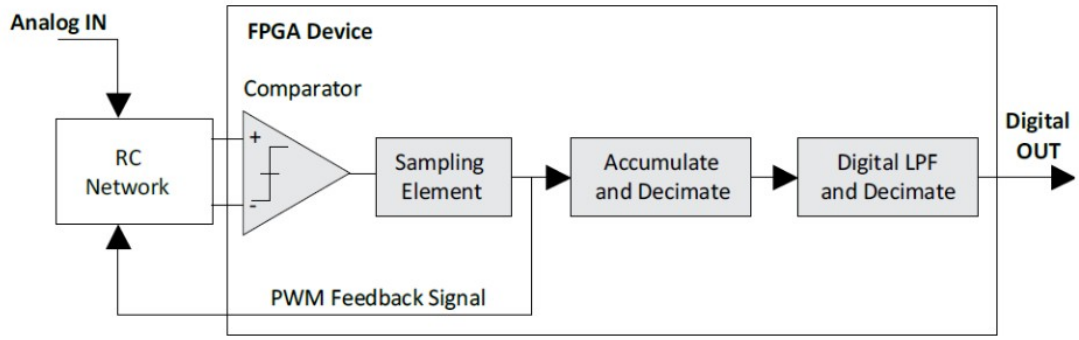

Figura 1 – Diagrama de blocos de um ADC Delta-Sigma

O ADC é implementado utilizando uma combinação de componentes internos e/ou externos: um comparador analógico, uma rede Resistor-Capacitor (RC), acumuladores e filtro digital passa-baixa. Caso o Hardware apresente entradas L*ow-Voltage Differential Signaling* (LVDS), elas podem ser usadas como comparador, somente deixando o circuito RC como componentes externos.

Metodologia de desenvolvimento

O projeto está hospedado no GitHub publicamente sob a licença MIT, acessível neste link. Os detalhes técnicos para replicar os resultados aqui mostrados estão descritos no arquivo *README.md*. Utilizamos a ferramenta Docker para igualar o ambiente de desenvolvimento e teste dos códigos VHDL. A ferramenta Octave também vem instalado dentro do container pois é essencial para produzir os arquivos CSV que servem como sinais de entrada, como está detalhado no arquivo *README.md*.

Para fazer análise e elaboração do código VHDL foi utilizado a ferramenta GHDL.

Após uma análise do *testbench* fornecido pela fabricante Lattice, foi decidido organizar em *packages* e identar o código, o que deixou o código mais sucinto, melhorando a compreensão do código *adc_tf.vhd*. Baseado neste arquivo foi criado o arquivo *adc_tf_csv.vhd* que, ao invés do código anterior que produzia um input estilo rampa pela própria linguagem VHDL, lê de um arquivo CSV os valores de entrada. Estes valores podem ser gerados por outras linguagens tais como Python, R, Julia, Matlab ou Octave. Em nosso caso utilizamos o Octave para gerar os sinais de entrada como também comparar o input com o output.

Resultados

A primeira arquitetura de filtro digital a ser testada é do tipo "*box average*", que simplesmente faz a média dos valores de entrada. As Figuras 2 e 3 mostram a saída para os sinais de entrada do tipo degrau e uma senóide.

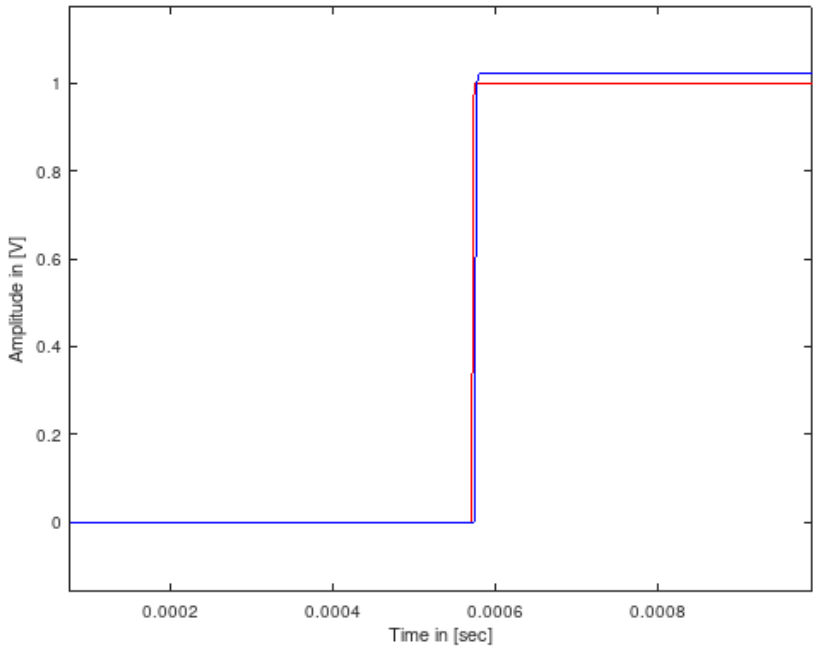

Figura 2 – Entrada degrau (vermelho) e saída (azul) no filtro "box average".

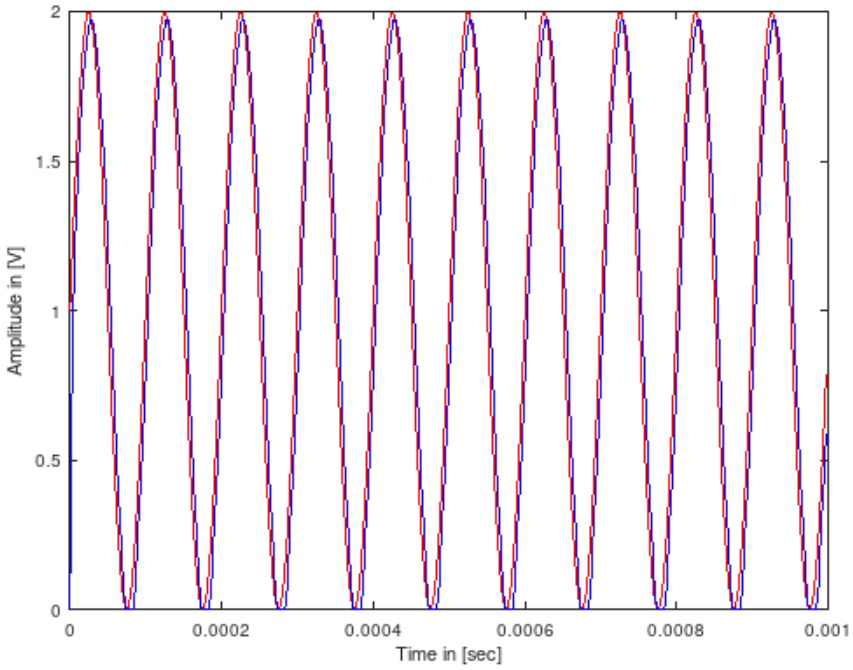

Figura 3 - Entrada senoidal (vermelho) e saída (azul) no filtro “box average”.

Na segunda arquitetura de filtro digital é do tipo “*sinc³*”, baseado no trabalho “*A Low Power Sinc³ Filter for ΣΔ Modulators*” de Lombardi *et al*. Nela é atribuído pesos às amostras coletadas. As Figuras 4 e 5 mostram a saída para os sinais de entrada do tipo degrau e uma senóide.

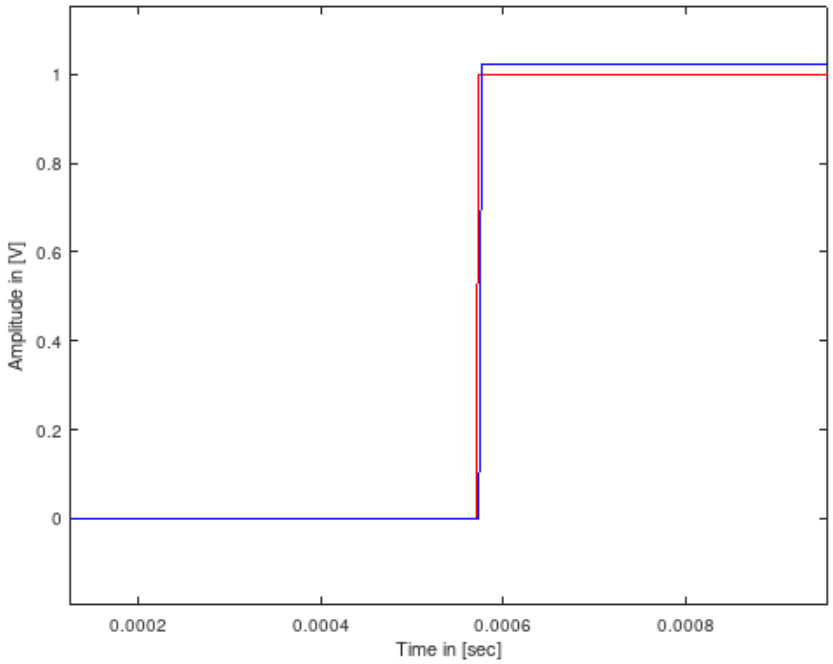

Figura 4 - Entrada degrau (vermelho) e saída (azul) no filtro “sinc³”.

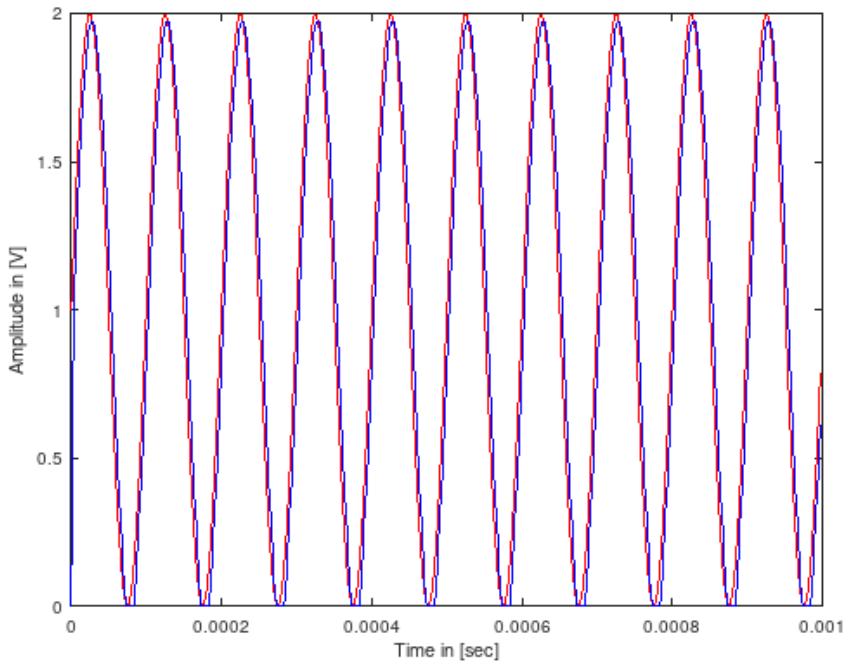

Figura 5 - Entrada senoidal (vermelho) e saída (azul) no filtro “sinc³”.

Conclusão e trabalhos futuros

A ideia de utilizar CSV como um meio de interface entre uma linguagem robusta de análise de dados (Octave, Python, R, etc.) e a linguagem que consome estes mesmos dados (VHDL) fortalece a validação da arquitetura e do filtro implementado.

Uma boa parte do tempo da elaboração deste trabalho foi usado para compreender a teoria e principalmente entender o código-fonte já fornecido pela fabricante Lattice. Ao separar parte do código em packages e identar corretamente, foi identificado trechos de códigos que não eram executados em nenhum momento. Estes trechos foram isolados em *pkg_tb*.

Apesar da saída do sinal parecer condizente com a entrada, há erros a serem corrigidos no código como o valor inicial e saturação do sinal. Um ponto que foi desconsiderado tanto no *testbench* do Lattice como no do desenvolvido é a relação entre o tempo de amostragem e o tempo de processamento do sinal.

Caso trabalhasse no item anterior, seria necessário também otimizar o filtro sinc³ pois foi feita da maneira mais simples e não da forma otimizada como mostrado no trabalho do Lombardi et al.

A organização da estrutura do código também é passível de críticas, tornando-a mais legível pela migração de trechos de códigos para *packages* e talvez a utilização de funções dentro destas.