

Universidade do Minho
Escola de Engenharia

Diogo Miguel Cunha Fernandes, PG47150
José Tomás Lima de Abreu, PG47386

Osciloscópio baseado em FPGA

Projeto Integrador em Eletrónica Industrial e Computadores

Trabalho realizado sob a orientação de
Professor Doutor Jorge Cabral
Professor Doutor Rui Machado
Professora Sofia Paiva

maio de 2022

Índice

1	Introdução	1
1.1	Enquadramento	1
1.2	Objetivos	1
1.3	Organização do documento	2
2	Estado da Arte	3
2.1	Estudo de mercado	3
2.2	HLS	3
2.3	Amostragem	4
2.4	Filtros Digitais	5
2.5	Aritmética de Fixed Point	7
3	Especificação do Sistema	9
3.1	Análise	9
3.2	Especificação de Hardware	10
3.3	Desenho do Sistema	12
3.3.1	Read XADC	12
3.3.2	Filters IP	12
3.3.3	HDMI IP	15
4	Implementação do Sistema	19
4.1	Amostragem	19
4.2	Filters IP	19
4.3	HDMI IP	20
5	Resultados do Sistema	22
5.1	Amostragem	22
5.2	Filters IP	23
5.3	HDMI IP	26
5.4	Resultados Finais	27
5.5	Utilização de Hardware	29
6	Conclusão	30
Bibliografia		30
A	Aliasing	33
B	Protocolos AMBA	35
C	Encoder TMDS	37
D	Influência da ordem dos filtros	38

E	Implementação de um filtro FIR em HLS	39
F	Testbenches	40
F.1	Geração dos resultados esperados dos filtros	40
F.2	Validação dos filtros no Vivado HLS	42
G	Resultados	46
G.1	Resultados Esperados	46
G.2	Resultados na STM32	47
G.3	Resultados na Zybo	48
G.4	Resultados Finais na Zybo	49
H	Block Design	51
I	Diagrama de Gantt	52

Índice de Figuras

Processo da amostragem: a) Conversor tempo contínuo - tempo discreto. b) Sinal contínuo à esquerda e respetivo sinal discreto à direita.	4
Resposta em frequência típica de um filtro passa baixo, em que w_c é a frequência de corte do filtro.	6
Obtenção do filtro através do método das janelas. a) Convolução da janela, W , com a resposta impulsional do LPF ideal, H_d . b) Aproximação do resultado da aplicação de uma janela a uma resposta impulsional ideal.	7
Diagrama de blocos do sistema.	9
Máquina de estados do AXI-Lite Master que faz a leitura dos valores do XADC.	12
Algoritmo de um filtro FIR.	13
Máquina de estados do RBUF.	13
Diagrama de blocos do bloco Filters IP.	14
Resposta em frequência de um LPF para diferentes ordens.	15
Diagrama de blocos do bloco HDMI IP.	16
Diagrama de estados da Main FSM.	17
Diagrama de estados da Write FSM.	18
Diagrama de estados da Read FSM.	18
<i>Block Design</i> implementado para o sistema.	19
Configuração do IP XADC Wizard.	19
Bloco FIR Filter gerado no Vivado HLS.	20
Perspetiva de análise no Vivado HLS.	20
<i>Block Design</i> do teste ao AXI <i>master</i> de leitura do XADC.	22
Resultado do testbench do AXI <i>master</i> de leitura do XADC.	22
Saída do LPF, a vermelho, em função da entrada, a azul.	23
Resultado da execução do testbench no Vivado HLS.	23
Simulação comportamental do bloco de filtros aplicando um LPF a uma onda de entrada de 20 Hz.	24
Simulação comportamental do bloco de filtros aplicando um LPF a uma onda de entrada de 100 Hz.	24
Resultado da simulação comportamental do bloco de filtros.	25
Saída do LPF, a azul, em função da entrada, a amarelo.	25
Bloco de filtros com o LPF selecionado para uma entrada, a azul escuro, de: a) 20 Hz - Saída a azul claro; b) 100 Hz - Saída a vermelho.	26
Simulação comportamental do bloco HDMI - Escrita da frame.	26
Simulação comportamental do bloco HDMI - Leitura da frame.	27
Simulação comportamental do bloco HDMI.	27
Montagem realizada para os testes finais.	28
Visualização do sinal de entrada a diferentes frequências.	28
Visualização da saída do LPF para vários sinais de entrada com frequências diferentes.	29
Sumário de utilização de hardware para todo o sistema.	29
Efeito do <i>aliasing</i> no domínio dos tempos: a) Sinal de entrada; b) Sinal amostrado.	33

Efeito do <i>aliasing</i> no domínio das frequências: a) Espetro do sinal de entrada; b) Espetro do trem de impulsos;	
c) Espetro do sinal amostrado com $\Omega_S > 2\Omega_N$; d) Espetro do sinal amostrado com $\Omega_S < 2\Omega_N$	34
Filtro <i>anti-aliasing</i> : resposta em frequência.	34
Canais usados pela interface AXI.	35
Fluxograma do encoder TMDS. [1]	37
Resposta em frequência de um HPF para diferentes ordens.	38
Resposta em frequência de um BPF para diferentes ordens.	38
Saída do HPF em Matlab, a vermelho, em função da entrada, a azul.	46
Saída do BPF em Matlab, a vermelho, em função da entrada, a azul.	46
Saída do HPF na STM32, a azul, em função da entrada, a amarelo.	47
Saída do BPF na STM32, a azul, em função da entrada, a amarelo.	47
Bloco de filtros com o HPF selecionado para uma entrada, a azul escuro, de: a) 20 Hz - Saída a azul claro;	
b) 100 Hz - Saída a vermelho.	48
Bloco de filtros com o BPF selecionado para uma entrada, a azul escuro, de: a) 20 Hz - Saída a azul claro;	
b) 120 e 220 Hz - Saída a vermelho.	48
Visualização da saída do HPF para vários sinais de entrada com frequências diferentes.	49
Visualização da saída do BPF para vários sinais de entrada com frequências diferentes.	50

Índice de Tabelas

Pinos HDMI mapeados na Zybo Z7.	11
---	----

Acrónimos

ADC Analog to Digital Converter.

AMBA Advanced Microcontroller Bus Architecture.

AP SoC All Programmable SoC.

APU Application Processing Unit.

ASIC Application Specific Integrated Circuit.

AXI Advanced eXtensible Interface.

BPF Band-Pass Filter.

CAD Computer-Aided Design.

DAC Digital to Analog Converter.

DSP Digital Signal Processing.

EDA Eletronic Design Automation.

FIR Finite Impulse Response.

FPGA Field-Programmable Gate Array.

HDL Hardware Description Language.

HLS High-Level Synthesis.

HPF High-Pass Filter.

I2C Inter-Integrated Circuit.

IIR Infinite Impulse Response.

LPF Low-Pass Filter.

MIO Multiplexed I/O.

MSPS Million Samples Per Second.

PL Programmable Logic.

Pmod Peripheral Module.

PS Processing System.

RTL Register-Transfer Level.

SoC System-on-Chip.

TMDS Transition-minimized differential signaling.

VHDL VHSIC Hardware Description Language.

VLSI Very Large-Scale Integration.

1 Introdução

A complexidade dos sistemas digitais tem avançado continuamente, sendo produzidos circuitos cada vez mais complexos. Quando os circuitos eram mais simples, o *hardware* era facilmente desenhado manualmente, especificando-se, por exemplo, as ligações dos componentes mais elementares, como os transístores. Com o aumento da complexidade dos sistemas digitais procuraram-se soluções de desenho de *hardware* automatizado, permitindo aos engenheiros de *hardware* trabalhar em camadas de abstração mais elevadas. Estas ferramentas permitem aumentar a eficiência do desenho de sistemas, reduzindo o tempo e esforço de desenvolvimento bem como o *time-to-market*.

A partir dos anos 90, os *designers* de circuitos digitais passaram a especificar apenas as funções lógicas, permitindo às ferramentas *Computer-Aided Design (CAD)* produzirem resultados mais otimizados. As especificações são geralmente fornecidas através de uma *Hardware Description Language (HDL)*, que é um tipo de linguagem de programação usada para descrever a estrutura e comportamento de um circuito digital. A abstração *Register-Transfer Level (RTL)* é usada em *HDLs*, como Verilog e *VHSIC Hardware Description Language (VHDL)*, de forma a criar representações de um circuito digital. Através de ferramentas *Electronic Design Automation (EDA)*, especificações *RTL* podem ser traduzidas num modelo de um circuito digital, e, subsequentemente traduzidas em especificações dependentes do dispositivo que irá implementar o circuito digital. [2]

As *Field-Programmable Gate Array (FPGA)*, dispositivos de lógica programável, permitem a redução do custo e tempo de prototipagem de sistemas *Very Large-Scale Integration (VLSI)* em relação a *Application Specific Integrated Circuit (ASIC)*. Estes dispositivos são programados usando os resultados da tradução das especificações *RTL* dependentes do dispositivo. Contudo a complexidade dos sistemas digitais tem aumentando mais rapidamente do que o avanço das ferramentas de projeto baseadas em *RTL*, dificultando o processo entre a definição do modelo do sistema e o protótipo final.

1.1 Enquadramento

As ferramentas de *High-Level Synthesis (HLS)* são mais uma camada de abstração no desenvolvimento de sistemas digitais, que permite explorar soluções de prototipagem rápida. Grande parte das ferramentas *HLS* permitem o uso de linguagens de programação como C/C++, reduzindo ainda mais o esforço e tempo de desenvolvimento.

Os algoritmos de processamento de sinal são normalmente complexos e difíceis de implementar em *hardware*. Portanto, explorar uma linguagem de mais alto nível (*HLS*) e avaliar a performance do *hardware* por esta gerados para aplicações de processamento de sinal é útil para conseguir uma prototipagem rápida e eficiente deste tipo de sistemas.

Desta forma, pretende-se construir um osciloscópio com recurso a aceleração de *hardware* em *FPGA*. O projeto envolve o desenvolvimento de funcionalidades de aquisição e processamento de sinal, como por exemplo, aplicação de filtros digitais diversos, bem como suporte gráfico através de um *display*.

1.2 Objetivos

O principal objetivo deste projeto será a exploração de técnicas de *Digital Signal Processing (DSP)* recorrendo a *HLS*, usando como caso de estudo um osciloscópio digital básico. O primeiro objetivo passa pela criação de um

osciloscópio básico capaz de amostrar sinais e aplicar filtros digitais, tal como filtros passa-baixo, passa-alto, passa-banda, bem como apresentar estes sinais num *display*, utilizando um controlador implementado em *hardware*.

De forma a organizar o processo de desenvolvimento, tal como para dividir o tempo necessário para cada objetivo, desenvolveu-se um diagrama de Gantt apresentado no anexo I.

1.3 Organização do documento

No capítulo 1 é feita uma breve introdução, apresentando o enquadramento e objetivos do projeto, bem como, a estrutura do documento.

No capítulo 2 é feita uma breve pesquisa do estado da arte acerca dos tópicos envolvidos na implementação do sistema, permitindo obter os conhecimentos necessários inerentes a esta tarefa. Assim, são apresentados fundamentos teóricos sobre **HLS**, amostragem e filtros digitais, bem como aritmética de *fixed point*. Além disso, é apresentado também um estudo de mercado, mostrando as principais soluções do mercado para osciloscópios digitais usando **FPGAs**.

No capítulo 3 é feita a especificação do sistema, detalhando cada um dos componentes que compõe o sistema. Serão analisados os requisitos e restrições do sistema, bem como as componentes da Zybo Z7-10 a utilizar. É também apresentado o desenho do sistema, detalhando os algoritmos a utilizar para amostragem e desenho de filtros digitais. Por último, é apresentado o plano de testes do projeto.

O capítulo 4 descreve o desenvolvimento e implementação dos componentes do sistema.

No capítulo 5 são apresentados os resultados experimentais dos testes realizados.

Por último, no capítulo 6, são apresentadas as conclusões do trabalho realizado, assim como, algumas sugestões para trabalho futuro, que têm como objetivo melhorar e expandir o trabalho desenvolvido.

2 Estado da Arte

Este capítulo apresenta uma visão geral dos principais conceitos adjacentes a este projeto, nomeadamente amostragem, filtros digitais, aritmética de *fixed-point* e **HLS**.

2.1 Estudo de mercado

Primitivamente, os osciloscópios funcionavam com base em componentes analógicos, onde o elemento sensor é um feixe de eletrões que, devido ao baixo valor da sua massa e por serem partículas carregadas eletricamente, são facilmente aceleradas e defletidas pela ação de um campo elétrico e magnético. Atualmente, os osciloscópios são, na sua maioria, compostos por componentes digitais, na qual o sinal é amostrado através de um *Analog to Digital Converter (ADC)*, que converte os valores de tensão em valores digitais, podendo ser guardados em memória para posterior processamento e visualização. Dependendo da frequência do sinal, poderá ser necessária uma capacidade de processamento elevado, devido às elevadas frequências de amostragem. Isto não é possível usando microcontroladores, que possuem frequências de relógio relativamente baixas e não permitem atingir altas larguras de banda. Assim, soluções que exigem elevada largura de banda, como por exemplo soluções de processamento de sinal, necessitam de dispositivos com capacidades de processamento superior.

Os dispositivos mais usados no presente são as **FPGAs** e os **ASICs**, que permitem a conceção de dispositivos personalizados ao tipo de solução final. A empresa Tektronix [3], fabricante de osciloscópios e outros dispositivos relacionados, desenham os **ASICs** que são integrados como produto final nos seus osciloscópios [4]. A vantagem desta abordagem prende-se com o facto de ser mais fácil produzir **ASICs** em larga escala, uma vez que apenas os recursos necessários são acrescentados ao dispositivo final, tornando-os mais baratos. Pelo contrário, as **FPGAs**, são dispositivos reprogramáveis, possuindo um maior número de recursos que, no final, não são utilizados. A nível de esforço e tempo de desenvolvimento, as **FPGAs** são menos exigentes, pelo que serão mais adequadas para aplicações que não sejam fabricadas em larga escala. A corporação National Instruments (NI) [5], disponibiliza soluções passíveis de serem reprogramadas através de **FPGA** [6].

Relativamente ao uso de uma linguagem de alto nível para descrever *hardware*, existe alguma controvérsia, sendo que alguns *designers* de *hardware* acreditam que a síntese nestas linguagens (**HLS**), em relação às **HDL**, provoca um aumento no *bitstream* gerado, aumentando os recursos usados e, consequentemente, o tempo de cálculo. Contudo, existem sintetizadores de linguagens de alto nível disponibilizados por grandes empresas de desenvolvimento de *hardware*, como o Vivado HLS [7] e o Quartus [8], desenvolvidos pela Xilinx e pela Intel, respetivamente, e baseados nas linguagens C/C++. Apesar de serem baseados nestas linguagens de programação, há código que não é sintetizável pela ferramenta **HLS** e, além disso, torna-se difícil a identificação do paralelismo existente em *hardware*. Por fim, as ferramentas **HLS** têm vindo a ganhar popularidade, levando a crer que é uma tecnologia emergente e passível de ser melhorada.

2.2 HLS

High-Level Synthesis (HLS), também conhecido como síntese comportamental e síntese algorítmica, é um processo de desenho em alto nível, onde uma descrição funcional de um *design* é automaticamente compilada numa implementação **RTL**, que tem de cumprir especificações definidas pelo utilizador. A descrição **HLS** providencia um **elevado nível de abstração**, já que não descreve um comportamento específico por ciclo de relógio. Além disso, permite a utilização de **linguagens de especificação de alto nível**, como C, C++ e até mesmo Matlab.

Fundamentalmente, um algoritmo **HLS** automatiza processos que seriam realizados manualmente por um *designer* de **RTL**:^[9]

- analisa e executa concorrência num algoritmo;
- insere apenas os registos necessários de forma a limitar a *critical path*;
- gera lógica de controlo (*control unit*), que controla o caminho dos dados (*data path*);
- mapeia os dados em elementos de memória, balanceando os recursos usados e a *bandwidth*;
- mapeia a computação em elementos de lógica, realizando otimizações, de forma a atingir a implementação mais eficiente.

2.3 Amostragem

Um sinal em tempo discreto pode ser originado através da amostragem de um sinal em tempo contínuo ou pode ser gerado diretamente através de outro processo em tempo discreto. Este tipo de sinal tem características muito atrativas para sistemas de processamento de sinal pois pode ser usado para simular sistemas analógicos ou para realizar operações que não podem ser realizadas usando *hardware* em tempo contínuo.

A conversão de um sinal contínuo num sinal discreto denomina-se amostragem. Este processo faz a aquisição de uma sequência de valores, ou amostras, em determinados instantes de tempo, igualmente espaçados, sendo a frequência de aquisição de amostras denominada de frequência de amostragem.

Na figura 2.1, apresenta-se o processo de amostragem. A multiplicação de um sinal contínuo, $x_c(t)$, por um trem de impulsos, $s(t) = \sum_{n=-\infty}^{+\infty} \delta(t - nT)$, onde T é o período de amostragem, resulta num sinal constituído por n impulsos escalados com o valor do sinal contínuo a cada instante nT . A conversão do novo trem de impulsos, $x_s(t)$, numa sequência em tempo discreto origina o sinal discreto $x[n] = x_c(nT)$. [10]

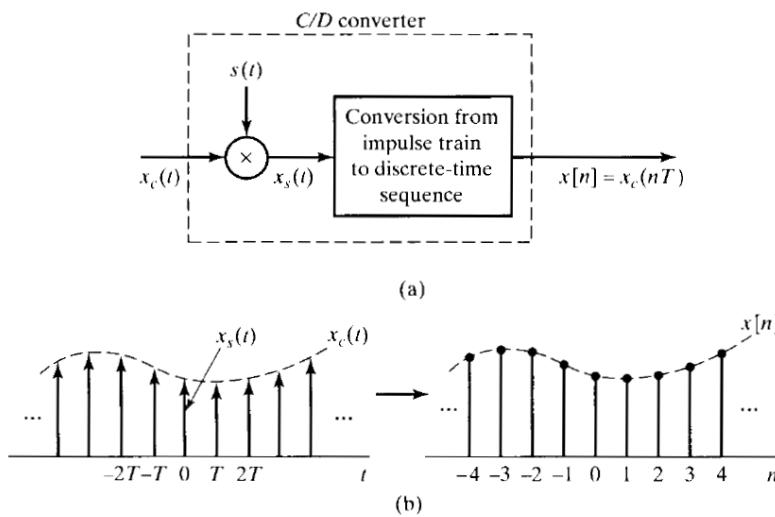


Figura 2.1: Processo da amostragem: a) Conversor tempo contínuo - tempo discreto. b) Sinal contínuo à esquerda e respetivo sinal discreto à direita.

Segundo o teorema de Nyquist, para que um sinal, com frequência máxima f_M , possa ser reconstruído é necessário que a frequência de amostragem, f_s , cumpra a equação 2.1.

$$f_s \geq 2.f_M \quad (2.1)$$

Quanto maior o número de amostras recolhidas de um sinal, melhor é a sua representação mas também mais memória será necessária para guardar o sinal. Portanto, deve existir um compromisso entre quantidade de amostras a recolher e memória a ser ocupada pelo sinal.

Aliasing

Quando o teorema de Nyquist não é cumprido, o conjunto de pontos amostrados não irá representar corretamente o sinal de entrada do sistema, tornando-se distorcido, impossibilitando a recuperação do sinal original. Dessa forma, o *aliasing* é o resultado do sinal ter sido sub-amostrado. Quando ocorre *aliasing* existe uma sobreposição de sinais a diferentes frequências, tornando-se impossível a distinção entre eles. (Anexo A)

Para se evitar que seja gerado *aliasing* deve ser aplicado um filtro *anti-aliasing* - um filtro passa baixo - à entrada do amostrador do sistema. Este tipo de filtros limitam a banda do sinal, removendo conteúdo de altas frequências, evitando a sobreposição do espetro do sinal com outras frequências. [11]

2.4 Filtros Digitais

Nos filtros digitais, cada valor da saída do filtro é calculado utilizando o valor da leitura realizada no **ADC**, $x[n]$, os valores de entrada passados, $x[n - k]$, e os valores de saída do filtro calculados nas iterações anteriores, $y[n - k]$. Através da equação de diferenças linear, mostrada na equação 2.2, é possível relacionar estas variáveis, utilizando os valores de a_k como os pesos a dar aos valores anteriores de saída do filtro e os valores de b_k como os pesos a dar aos valores anteriores de entrada do filtro.

$$y[n] = a_1y[n - 1] + \dots + a_Ny[n - N] + b_0x[n] + b_1x[n - 1] + \dots + b_Mx[n - M] \quad (2.2)$$

Filtros IIR

Os filtros *Infinite Impulse Response* (IIR), calculam a sua saída baseado não apenas nos sinais de entrada mas também nos valores de saídas anteriores. Devido a esta realimentação, este tipo de filtros pode gerar uma saída mesmo na ausência de sinal de entrada, daí o seu nome, havendo o risco de instabilidade na sua resposta impulsional. O desenho de filtros IIR pode apresentar-se difícil, no entanto, estes podem ser desenhados a partir de filtros analógicos, facilitando muito o processo de desenho de filtros IIR de ordem reduzida.

Filtros FIR

Num filtro *Finite Impulse Response* (FIR), relembrando a equação 2.2, os coeficientes a_k são nulos, fazendo com que a saída do filtro seja apenas dependente dos valores de entrada atual e anteriores. Desta forma, ao fim de M períodos, a resposta do filtro a um impulso unitário torna-se zero, ou seja, a resposta impulsional tem duração finita, daí o seu nome *Finite Impulse Response*. Esta pode ser uma vantagem pois torna este tipo de filtros estáveis. Além disso, estes filtros apresentam resposta em frequência com fase linear. A principal desvantagem deste tipo de filtros é o tempo necessário para a sua execução. Como o filtro não tem realimentação, necessita de mais coeficientes no seu sistema de forma a atingir as mesmas especificações definidas num filtro IIR. Para cada coeficiente extra é realizada uma multiplicação extra, além de um maior uso de memória. Num sistema em tempo

real, as limitações na velocidade e memória do sistema para um sistema **FIR** pode tornar o sistema inviável.

Um filtro passa-baixo (*Low-Pass Filter (LPF)*) apresenta uma resposta em frequência genericamente representada pela figura 2.2: a **banda passante** é o conjunto de frequências até a resposta do filtro ser diminuída; a **banda de transição** é o conjunto de frequências onde a resposta é atenuada, até que, na **banda de rejeição**, tratam-se do conjunto de frequências rejeitadas pelo filtro. Nas bandas passante e de rejeição é possível ver a existência de **ripple**, δ , que consiste na variação de amplitude na resposta do filtro. Num filtro **FIR**, o valor máximo do *ripple* é o mesmo em ambas as bandas. Quanto menor for o *ripple* e a largura da banda de transição, melhor será o filtro.

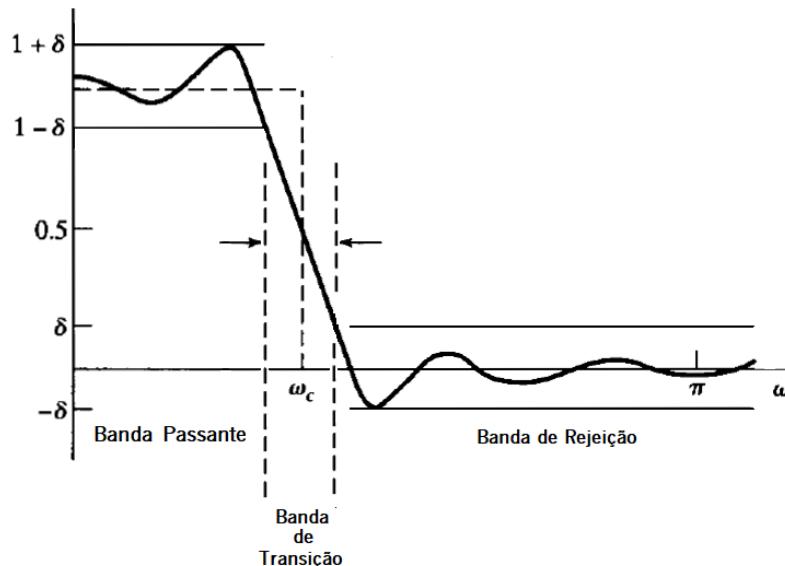


Figura 2.2: Resposta em frequência típica de um filtro passa baixo, em que ω_c é a frequência de corte do filtro.

Método das Janelas

As descontinuidades no domínio das frequências implicam respostas impulsoriais de duração infinita e não causais. O objetivo deste método é truncar a resposta impulsional do sistema discreto.

O método das janelas permite gerar uma “janela”, de comprimento $M + 1$, no domínio dos tempos discretos. O filtro é obtido através da multiplicação da janela pela resposta impulsional de um **LPF** ideal, limitando o seu comprimento e forma. No domínio das frequências isso significa que a resposta impulsional do filtro, $H(\Omega)$, será igual à convolução da janela, $W(\Omega)$, com a resposta impulsional de um **LPF** ideal, $H_d(\Omega)$, tal como demonstrado na figura 2.3.

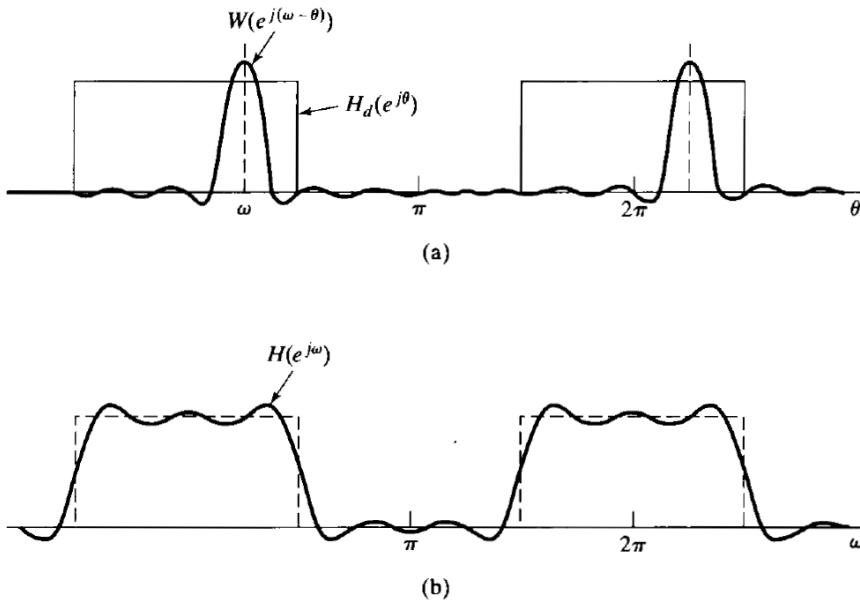


Figura 2.3: Obtenção do filtro através do método das janelas. a) Convolução da janela, W , com a resposta impulsional do LPF ideal, H_d . b) Aproximação do resultado da aplicação de uma janela a uma resposta impulsional ideal.

Janela de Kaiser

A janela de Kaiser é uma das janelas mais utilizadas neste método, por ser **flexível e evitar procedimentos de tentativa e erro** (usados na aplicação de outras janelas) para achar os parâmetros que garantem as especificações requeridas do filtro digital. Na tentativa de resolver o compromisso entre largura do lobo principal e atenuação do lobo secundário, de um filtro digital, Kaiser descobriu a janela (equação 2.3):

$$w[n] = \begin{cases} \frac{I_0[\beta(1 - [(n - \alpha)/\alpha]^2)^{1/2}]}{I_0(\beta)}, & 0 \leq n \leq M, \\ 0, & \text{caso contrário} \end{cases} \quad (2.3)$$

Com $\alpha = \frac{M}{2}$, β representa a forma da janela e I_0 é uma função de Bessel modificada do 1º tipo e de ordem zero. Sabe-se que a função de Kaiser é mais bem-sucedida quando β é constante e M aumenta, diminuindo a largura do lobo principal, afetando pouco a amplitude dos lobos secundários.

2.5 Aritmética de Fixed Point

Em sistemas embebidos, há duas formas de representar a aproximação a um número real: formato *fixed-point* ou formato *floating-point*. O formato *fixed-point* define um **número fixo de dígitos** para representar a **parte decimal** do número real. Pelo contrário, o formato *floating-point* pode ter um número variável de dígitos para representar a parte decimal do número real, dependendo da escala deste número. O processador, muitas vezes, inclui uma unidade de *floating-point* (FPU), de forma a acelerar o cálculo neste formato.

Na ausência de uma FPU, como é o caso da Zybo Z7, o uso da aritmética de *fixed-point* é preferido, uma vez que a implementação da aritmética de *floating-point* em software é mais complexa e custosa em termos de desempenho. A representação *fixed-point* de um número fracional pode ser vista como um inteiro que foi multiplicado por um fator de escala. Por exemplo, o valor real 1,234 pode ser guardado numa variável inteira com o valor de 1234, cujo fator de escala implícito é de 1/1000.

Numa variável x , de N bits, podem-se definir b bits para representar a parte fracional do número. De forma a converter o número em *fixed-point* basta multiplicar $x \cdot 2^b$, sendo 2^b o fator de escala envolvido na conversão. Desta forma, a parte inteira do número em *fixed-point* é representada por $N - b$ bits. Para converter novamente o número de *fixed-point* para fracional, basta realizar a operação contrária, multiplicando $x \cdot 2^{-b}$. Por vezes, após esta última conversão é possível verificar que o número real inicial não coincide com o número que foi convertido duas vezes, evidenciando a perda de precisão associada a este formato.

3 Especificação do Sistema

Neste capítulo serão apresentados cada um dos componentes que compõe a base do sistema a implementar. Assim, primeiro serão definidos os requisitos e restrições do sistema, tal como o seu diagrama de blocos. Depois disso será especificado o *hardware* desenhado.

3.1 Análise

Requisitos

- Taxa de amostragem de 1 kS/s;
- Implementação de um **LPF**, **HPF**, **BPF**;
- Apresentação do sinal filtrado num *display*,
- Resolução de vídeo HDMI 640 x 480;

Restrições

- Usar FPGA-SoC Zybo Z7;
- Usar HLS;

Diagrama de Blocos do Sistema

Na figura 3.4 está representado o diagrama de blocos do sistema. O sistema é composto pela Zybo Z7-10, cujas entradas serão o sinal a tratar, através dos pinos **Peripheral Module (Pmod)** do XADC (*ADCPin_n* e *ADCPin_p*) e a seleção do filtro por parte do utilizador (*filter_select*), usando os *switches* da placa de desenvolvimento. A única saída do sistema é o sinal filtrado, transmitido num *display*, através de uma interface HDMI (*HDMI-Tx Port*).

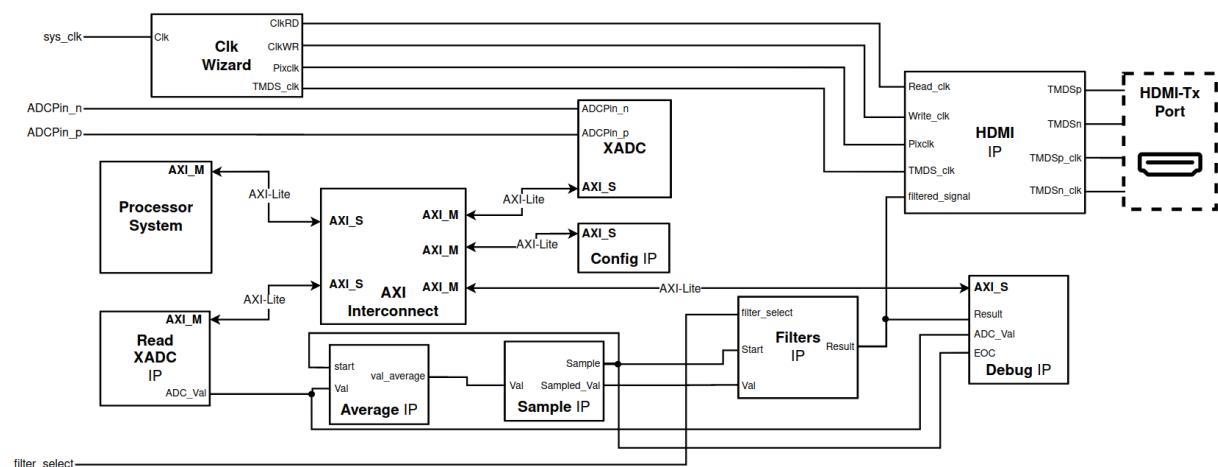


Figura 3.4: Diagrama de blocos do sistema.

O bloco **Processor System (PS)**, é responsável pela configuração e inicialização do XADC. Quando o ADC estiver configurado, deverá ser ativa uma *flag* de configuração, através do registo do *slave AXI Config*. Além disso,

a **PS** permite realizar *debug* às principais linhas do sistema, como o valor convertido pelo XADC (*ADC_Val*) ou o resultado da aplicação dos filtros (*Result*), através da utilização de um IP de AXI *slave* de *debug* (**Debug**). O bloco de **Read XADC** é um bloco AXI-Lite *master* que espera que o XADC esteja configurado pela **PS** e efetua as respetivas leituras aos registos do ADC, colocando-o na saída (*ADC_Val*). A frequência de amostragem definida para este sistema foi de 1 kHz e, uma vez que o XADC possui uma frequência de amostragem de 1 MHz, foi implementado um IP que realiza a média dos valores convertidos pelo XADC, num período de amostragem (1 ms). O valor da média é limpo a cada amostragem e esta frequência é definida pelo IP **Sample**. Estes sinais e o sinal de seleção de filtro (*filter_select*) são usados pelo bloco de filtros (**Filters**) de forma a aplicar o respetivo filtro sobre o sinal lido pelo XADC. O sinal após a aplicação do filtro (*result*) entra no bloco de produção e transmissão de imagem através de HDMI (**HDMI**). Este bloco utiliza diferentes fontes de relógio para diferentes tarefas: *clkRD* é a frequência de leitura da imagem em memória; *clkWR* para fazer a escrita da imagem na memória; *Pixclk* representa a frequência de leitura de um pixel da imagem a ser transmitida; *TMDS_clk*, definido pelo protocolo HDMI, representa a frequência com que os sinais de HDMI são comutados, conforme o pixel lido.

Todos os IPs que usam uma interface AXI-Lite, estão conectados a um IP de barramento, chamado **AXI Interconnect** e todas as comunicações devem ser iniciadas, através do *handshake* respetivo do protocolo AXI, por um IP com interface AXI-Lite *master* (*AXI_M*) que têm mapeados os endereços dos IPs com interface AXI-Lite *slave* (*AXI_S*).

3.2 Especificação de Hardware

A Zybo (Zynq Board) Z7 é uma placa de desenvolvimento de circuitos digitais e de *software* embebido, baseada na arquitetura **All Programmable SoC (AP SoC)** da Xilinx. Integra um processador de dois núcleos ARM Cortex-A9 com uma **FPGA** série 7 da Xilinx. [12]

A Zybo Z7 é uma placa com custo acessível para contextos académicos, contendo um conjunto de periféricos que podem ser acedidos pelo sistema de processamento da Zynq (**Processing System (PS)**) ou pela lógica programável (**Programmable Logic (PL)**). Esta última permite criar aceleradores de *hardware* de forma a responder a aplicações que exijam uma maior largura de banda.

Entre os diversos recursos que a Zybo Z7 apresenta, de seguida são apresentados aqueles que se demonstram importantes para a realização deste projeto:

- Frequência de relógio interna superior a 450 MHz;
- Controladores de periféricos de baixa largura de banda: SPI, UART, entre outros;
- 2 **ADC** de 16 bits *on-chip* (XADC) capaz de adquirir 1 **Million Samples Per Second (MSPS)** ;
- 6 botões de pressão, 4 interruptores, 5 LEDs;
- Conectores de expansão: 6 portos **Peripheral Module (Pmod)** ;
- 2 portas HDMI, de *input* e *output*;

A **PS** consiste em vários componentes, incluindo a **Application Processing Unit (APU)** (que inclui os dois processadores Cortex-A9), interconexão **Advanced Microcontroller Bus Architecture (AMBA)** - anexo B, e vários controladores de periféricos cujas entradas e saídas estão multiplexadas para 54 pinos dedicados (denominados **Multiplexed I/O** ou pinos **MIO**).

XADC

O módulo XADC é um porto **Pmod** conectado aos pinos de entrada analógica da **PL**. Este porto é usado para levar sinais analógicos de entrada para o **ADC** dentro do núcleo Zynq XADC. [13]

O XADC tem 2 canais **ADC** de 16-bit, com taxa de amostragem máxima de 1 MSPS, com um multiplexador analógico (até 17 canais de entrada analógica externa). Estes canais **ADCs** podem ser configurados para amostrar simultaneamente oito sinais analógicos de entrada externa, com largura de banda máxima de 500 kHz. [12] Os oito sinais de dados do **Pmod** estão agrupados em 4 pares, em que cada par tem um filtro *anti-aliasing* parcialmente disponível.

No modo de entrada analógica, a tensão nos pinos do porto deve ser limitada a 1 V pico-a-pico. Desta forma, no modo unipolar (por defeito) quando a entrada é 1 V, o **ADC** converte no valor digital FFFFh (16-bit). Através da equação 3.4 é possível calcular a tensão obtido à entrada do **ADC**. [14]

$$ADC_{value} = \frac{V_{in}}{1.0V} \cdot FFFFh \quad (3.4)$$

O módulo XADC comunica com os vários IPs através do protocolo de comunicação AXI-Lite. O valor convertido pelo **ADC** é guardado no registo reservado ao respetivo canal, sendo acedido diretamente pela **PS** ou através da **PL** utilizando um IP AXI_Master personalizado, que controla as transações **AXI**.

HDMI

De forma a mostrar o sinal processado foi usada uma interface HDMI Tx para transmitir a respetiva imagem para um *display*. A Zybo Z7 tem disponível 2 portas HDMI: saída (sem *buffer*, configurada para transmissão de dados) e entrada (com *buffer*, configurada para receção de dados). Neste caso, é apenas analisado o caso do HDMI como saída, uma vez que há apenas transmissão de dados.

O sistema de conexão HDMI é compatível com o *Digital Visual Interface* (DVI), uma vez que usam o mesmo *standard* de sinalização - *Transition-minimized differential signaling* (TMDS) . Na tabela , estão representados os pinos relevantes da interface HDMI. Os canais TMDS são canais diferenciais (pino positivo, negativo e de proteção) que permitem a transmissão de vídeo digital a altas velocidades, de forma síncrona. Todos os outros canais são canais auxiliares (opcionais). O *Display Data Channel* (DDC), que utiliza o protocolo I2C, permite configurar os parâmetros como brilho e contraste do *display* que realiza a interface. O protocolo *Consumer Electronics Control* (CEC) permite que mensagens sejam transmitidas numa cadeia de HDMI entre diferentes dispositivos.

Tabela : Pinos HDMI mapeados na Zybo Z7.

Pino / Canal	Descrição	Pino Zybo Z7
TMDS canal 0 (+,-)	Saída de dados	D19, D20
TMDS canal 1 (+,-)	Saída de dados	C20, B20
TMDS canal 2 (+,-)	Saída de dados	B19, A20
TMDS Clock (+,-)	Saída de clock	H16, H17
SCL, SDA	DDC	G17, G18
CEC		E19
Hot plug detect	Deteção de conexão	
+5 V e GND	Alimentação	

3.3 Desenho do Sistema

Nesta secção são apresentados todos os diagramas necessários para a compreensão do sistema a desenvolver, detalhando os blocos identificados na figura 3.4.

3.3.1 Read XADC

De forma a obter o valor digital convertido pelo XADC, é necessário implementar um AXI-Lite *Master* que controle as transações de leitura para o *slave* XADC, respeitando o *handshake* do protocolo **AXI**. Na figura 3.5 é apresentada a máquina de estados que faz a leitura dos valores do XADC.

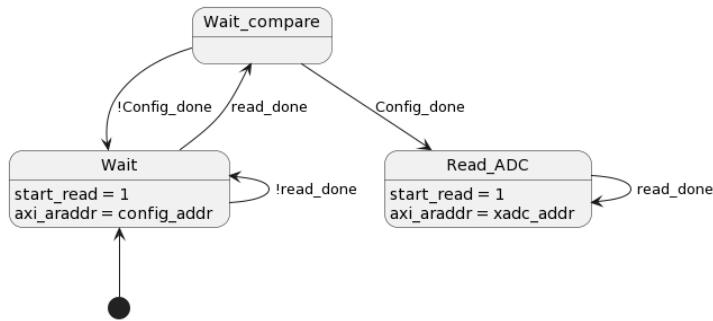


Figura 3.5: Máquina de estados do AXI-Lite Master que faz a leitura dos valores do XADC.

Ao *reset*, a máquina de estados inicia no estado de *Wait*, que espera que seja efetuada uma leitura ao IP *slave* de configuração (*Config*). Quando uma leitura for efetuada, o próximo estado é *Wait_compare*, que faz a comparação do valor lido. Se este valor for 0 significa que o que o XADC ainda não foi configurado pela **PS**, pelo que se volta para o estado de *Wait* para continuar a fazer leituras, até que o valor lido seja 1. Assim, com o ADC configurado, a máquina de estados avança para o estado *Read_ADC*, onde efetua leituras consecutivas ao registo do XADC, de forma a retirar o valor convertido por este. Os registos do XADC podem ser consultados na tabela 2-3 do guia do produto *XADC Wizard v3.3* [15].

3.3.2 Filters IP

Algoritmo de Filtragem Digital

Começando por fazer a especificação de um filtro **FIR**, utilizando por base a equação diferenças 2.2, pode-se escrever a equação diferenças 3.5 que representa um filtro **FIR**, em que M indica a ordem do filtro e b_k os seus coeficientes.

$$y[n] = b_0x[n] + b_1x[n - 1] + \dots + b_Mx[n - M] \quad (3.5)$$

A partir da equação 3.5 é possível criar um algoritmo que permita atualizar os valores das entradas anteriores e calcular a saída do filtro, representado na figura 3.6. A variável x_{ant} representa os valores anteriores da entrada, x . O elemento $x_{ant}[0]$ será o valor x lido do ADC no instante n . O elemento $x_{ant}[M]$ será o valor x lido do ADC no instante $n-M$, sendo assim o elemento mais antigo em memória. Da mesma forma, a variável y_{ant} representa os valores anteriores da saída y .

A cada valor novo na entrada do filtro, a variável x_{ant} é atualizada, removendo o valor de x mais antigo,

$x_ant[M]$, e adicionando o valor de entrada atual, x , em $x_ant[0]$.

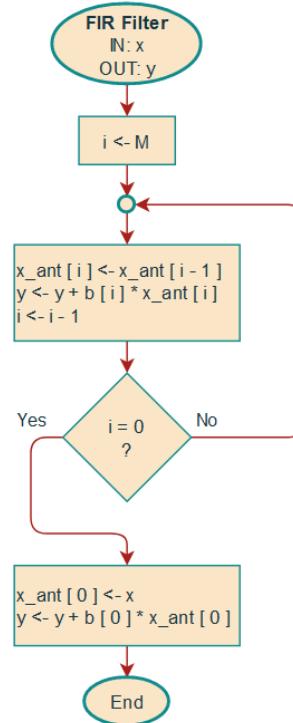


Figura 3.6: Algoritmo de um filtro FIR.

De forma a implementar o mecanismo de atualização dos valores de entrada anteriores definiu-se um bloco, **RBUF**, descrito pela FSM apresentada na figura 3.7. Este bloco permite controlar a escrita de uma BRAM externa, para guardar os valores de entrada anteriores. Sempre que o sinal *en* for ativo, adicionar-se-á o novo valor, *data in - di*, ao buffer interno, *buff*, no estado *S_ADD*. No estado seguinte, geram-se os sinais de *addr*, *output write enable - owe* e *data out - do*, para utilizar na BRAM. De notar que o *addr* deverá ter como valor máximo o número de entradas anteriores a guardar, que, como visto na equação 3.5 é definido por M . Terminado o estado *S_WRITE* avança-se para o estado *S_SHIFT* realizando-se a atualização do buffer interno. Enquanto esta atualização é feita, já é possível a leitura da BRAM com os valores de entrada atualizados, podendo ser utilizados pelos filtros.

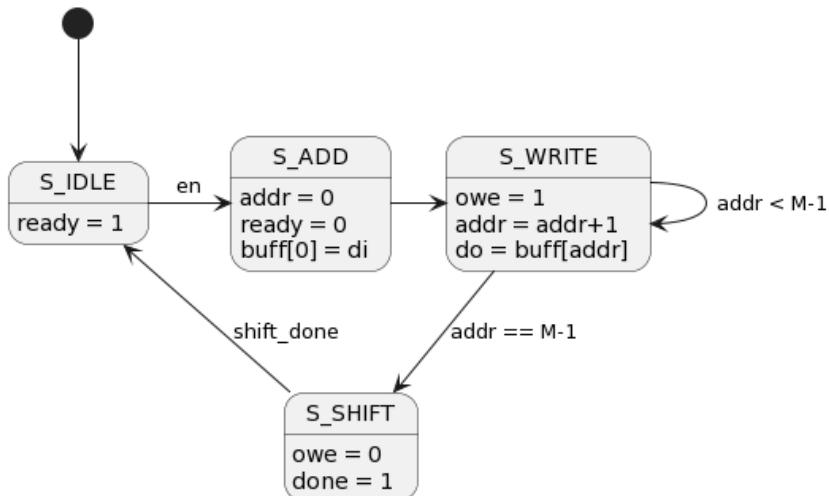


Figura 3.7: Máquina de estados do RBUF.

Diagrama de Blocos

Na figura 3.8 é possível analisar o bloco Filters em maior detalhe. Este bloco pode ser visto como um multiplexador de filtros para um dado valor de entrada. A entrada *filter-select* é o seletor do sinal a apresentar na saída do bloco (*result*). Este sinal é igual ao sinal de entrada caso o seletor seja 11; igual ao sinal de entrada após aplicação de um **LPF**, caso o seletor seja 00, e assim em diante. O sinal de *start* permite iniciar o processo de filtragem.

Inicialmente deve ser feita uma atualização dos valores de entrada anteriores, usando para isso o bloco **RBUF**. Este permite atualizar a BRAM utilizada para armazenar os valores de entrada anteriores, a **BRAM_X_ANT**. Após essa atualização ser feita, sinalizada através do sinal *done*, é dado o sinal de *start* ao filtro que se encontra selecionado. Cada filtro necessita dos valores de entrada anteriores e dos respetivos coeficientes, sendo por isso necessário controlar os endereços de leitura das BRAMs.

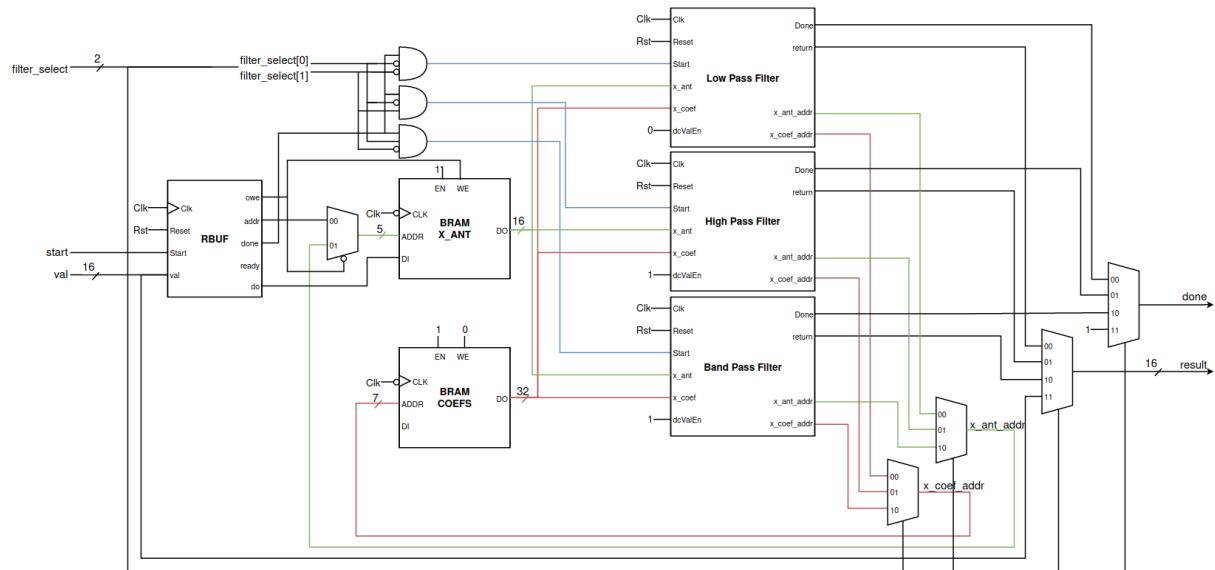


Figura 3.8: Diagrama de blocos do bloco Filters IP.

Desenho de Filtros Digitais

Com recurso à ferramenta Matlab é possível determinar os coeficientes da janela de Kaiser de uma forma mais rápida, tal como se pode ver na listagem 3.1. É necessário definir a frequência de amostragem do sistema, *fsamp*, as frequências de corte, *fcuts* e os *ripples* das bandas, *devs*. Desta forma é possível obter os *n* coeficientes da janela em *hh*.

```

1 % low pass filter
2 mags = [1 0];
3
4 % get kaiser window
5 [n,Wn,beta,ftype] = kaiserord ( fcuts ,mags,devs,fsamp );
6 % calculate coefficients
7 hh = fir1 (n,Wn,ftype , kaiser (n+1,beta), ' noscale ' );

```

Listing 3.1: Algoritmo para Desenho de um LPF.

Os coeficientes obtidos serão números reais, e portanto, variáveis de vírgula flutuante. Usando aritmética *fixed-point* permite diminuir o espaço de memória ocupado pelos coeficientes. Usando variáveis do tipo `uint16_t`, reservando 15 bits para a parte decimal do número, podem-se obter os coeficientes do filtro, fazendo:

```
1 x_coefs = hh * 2^-15;
```

Na figura 3.9 está apresentado um estudo realizado para determinar a ordem de um **LPF** (e consequentemente o número de coeficientes do filtro) que apresente melhor compromisso entre a memória ocupada pelos coeficientes e a qualidade do filtro. O mesmo estudo para o **HPF** e **BPF** é apresentado no anexo D.

É possível verificar que para uma ordem mais elevada do filtro, a resposta em frequência do filtro é melhor definida do que para ordens mais baixas. A banda passante do filtro para $M=224$ apresenta atenuação constante nos 0 dB até à frequência de corte, 50 Hz. Na banda de rejeição, a partir dos 60 Hz, as componentes são atenuadas a pelo menos 40 dB. No entanto, a implementação deste filtro requer a utilização de 224 coeficientes, que terão de estar guardados em memória. Utilizando aritmética *fixed-point* com variáveis `uint16_t`, significa que seriam necessários 3,5 Kbit para armazenar estes coeficientes.

Neste projeto serão implementados filtros com **ordem** $M=21$, havendo alocação de menos memória para o filtro, apesar da pior resposta em frequência.

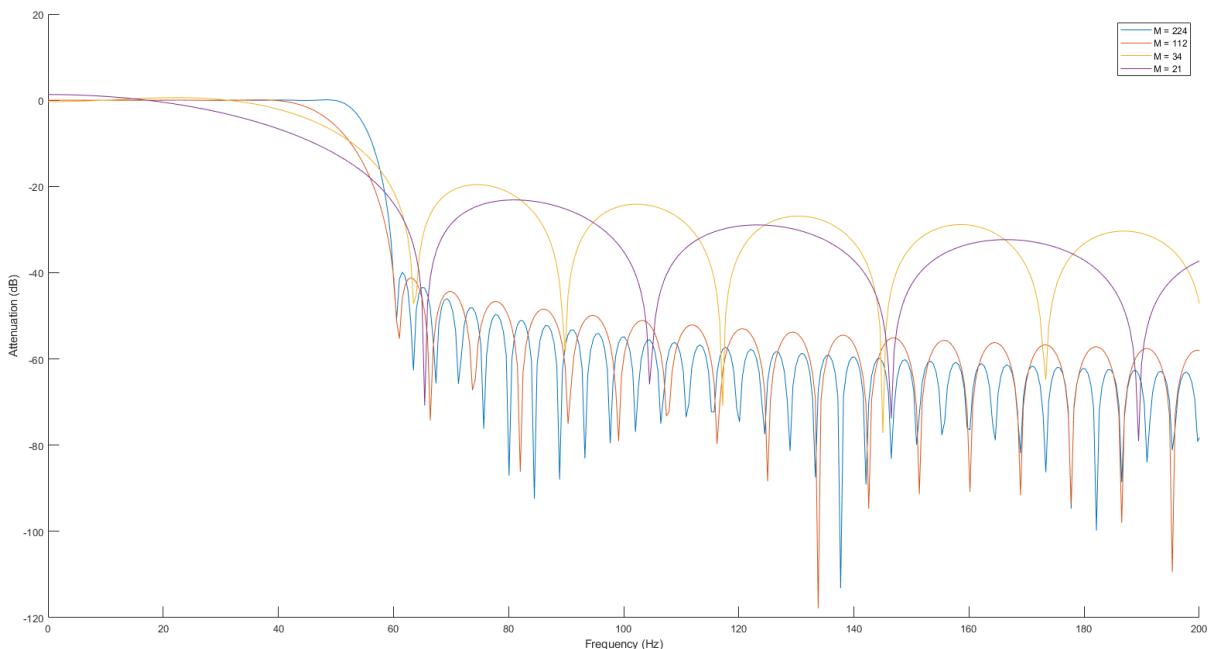


Figura 3.9: Resposta em frequência de um **LPF** para diferentes ordens.

3.3.3 HDMI IP

Na figura 3.10 está representado o diagrama de blocos do IP que faz interface com a porta HDMI, sendo responsável por, através do sinal filtrado, produzir e transmitir uma imagem à freqüência padrão de 60 Hz.

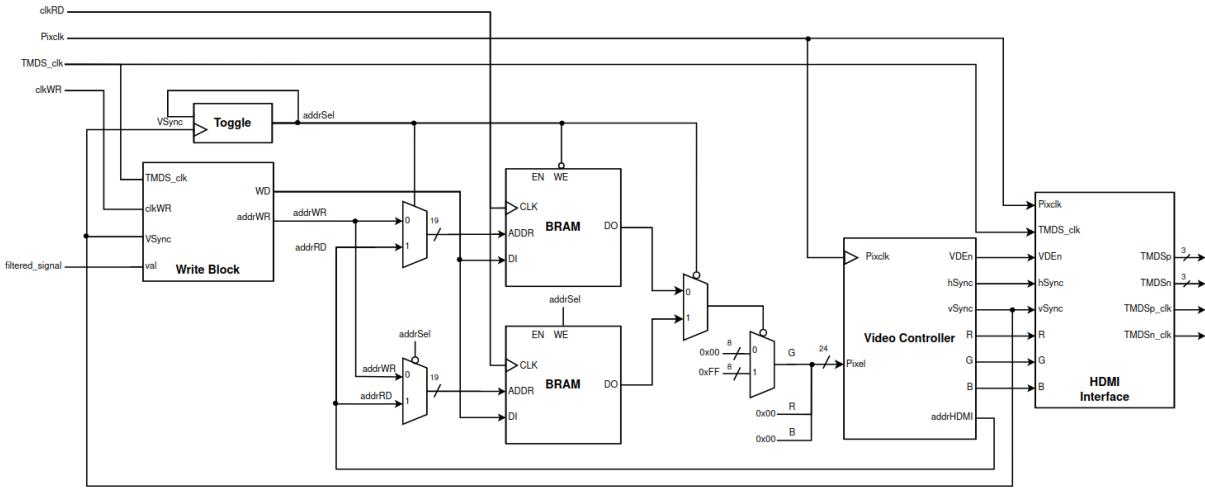


Figura 3.10: Diagrama de blocos do bloco HDMI IP.

O bloco *Write Block* tem como objetivo a produção da imagem, através do valor calculado pelo IP *Filters*, *filtered_signal*. Assim, deverá ser representada uma onda na imagem, de acordo com este valor em cada instante, sendo que a onda é representada em verde (RGB - 00FF00h) e o resto da imagem a preto (RGB - 000000h). O valor de entrada é mapeado no eixo dos yy, de acordo com um escalar, ou seja, para o valor constante máximo de entrada (FFFFh), deverá ser transmitida uma imagem com uma linha horizontal na linha superior do ecrã, enquanto que para um valor constante mínimo (0h), será transmitida uma imagem com uma linha horizontal na linha inferior do ecrã.

Existem duas memórias *BRAM*, uma para leitura da *frame* a transmitir e outra para a escrita da próxima imagem a ser transmitida. De forma a determinar qual das *BRAMs* será de escrita e de leitura, é efetuada uma lógica inversa usando multiplexadores, através do sinal *addrSel* fazendo com que as memórias troquem de funções (escrita ou leitura) a cada imagem transmitida (*vSync*). Apesar de um pixel ter um tamanho de 24 bits, na escrita da *BRAM*, é apenas usado um bit, de forma a indicar o endereço da imagem que estará a verde. Assim, durante a leitura, usando o endereço *addrRD* (*addrHDMI*), o endereço que estiver ativo (1b), irá saturar o canal verde (G - FFh) e quando estiver inativo (0b), não irá saturar o canal verde (G - 00h). Os outros canais nunca são saturados (R e B - 00h), completando assim um pixel RGB (24-bit). O IP *Video Controller*, é responsável por gerar o *index* de leitura da imagem (*addrHDMI*), os sinais de sincronização HDMI: *vSync* (no final de cada imagem) e *hSync* (no fim de cada linha). Os pixels são transmitidos para o bloco *HDMI Interface*, que faz a transmissão dos sinais HDMI para a porta de transmissão.

As fontes de relógio apresentadas têm frequências diferentes, de acordo com o seu propósito: *TMDS_clk-clock* que respeita o protocolo HDMI (250 MHz); *Pixclk* - sabendo que temos uma imagem de 640 por 480, cada pixel com 24 bits, de forma a transmitir a uma frequência padrão de 60 Hz, precisamos de um *clock* com uma frequência de 25 MHz. Esta é a frequência de leitura de um pixel; *clkWR* - frequência de escrita da imagem em memória. É pretendido que a imagem possua uma resolução de 100 ms, de forma a representar ondas de frequências relativamente baixas e, sabendo que a imagem possui 640 pixels de comprimento, é necessária uma frequência de escrita para a memória de 6400 Hz; *clkRD* - frequência de leitura da imagem em memória (50 MHz) - dobro de *Pixclk*, devido aos atrasos introduzidos pela máquina de estados de leitura, apresentada abaixo na subsecção 3.3.3.

Main FSM

Na figura 3.11 está representada a máquina de estados do IP *Write Block*, em que o primeiro estado corresponde ao estado de *S_CLEAN*, onde a *BRAM* é preparada para a escrita, através da sua limpeza. Quando a memória for toda limpa (contadores do eixo dos xx e dos yy, *counterX* e *counterY*, respetivamente, nos valores máximos - *cleanDone*), passa para o estado *S_WRITE*, onde a memória é escrita através da lógica descrita acima. Após a última coluna ser escrita (*counterX* no valor máximo - *writeDone*), avança para o estado *S_IDLE*, onde espera que a imagem anterior acabe de ser lida, representado através de um pulso de *vSync*.

Os estados *S_CLEAN* e *S_WRITE* realizam ambos a escrita de uma *BRAM*, variando apenas no valor de escrita das *BRAM*, *write data - WD*. No entanto, a limpeza da *frame* requer que todos os pixels sejam reescritos, enquanto que na escrita apenas são necessários escrever *width* pixels. Portanto a coluna a utilizar na Write FSM será diferente para estes dois estados, sendo no primeiro definido pelo valor do contador vertical, *counterY*, e no segundo definido pelo valor de entrada do bloco, *valIndex* - detalhado na Write FSM.

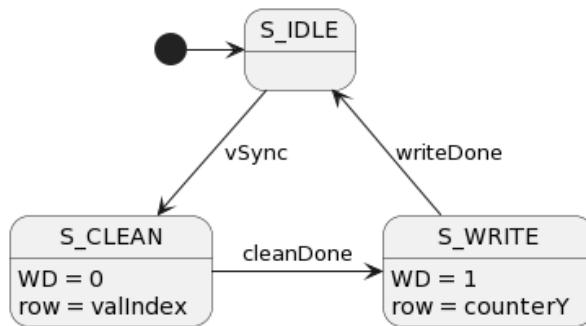


Figura 3.11: Diagrama de estados da Main FSM.

Write FSM

Na figura 3.12 é apresentada a máquina de estados Write FSM, que é executada nos estados de *S_CLEAN* e *S_WRITE* da Main FSM. A única diferença entre estes dois estados é a variável *row*, que é utilizada para determinar o *adMul* e posteriormente o *addrWR*, que é o endereço de escrita da *BRAM*, relativo à posição da imagem em memória, no qual é pretendido atribuir a cor verde.

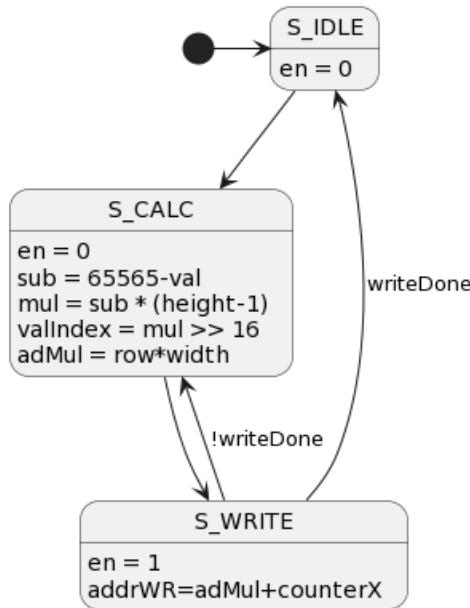


Figura 3.12: Diagrama de estados da Write FSM.

Em *S_WRITE* realiza-se a escrita do *write data* - *WD* definido na Main FSM, para o *write address* - *addrWR*, estando para isso o *enable* da BRAM ativo. Em *S_CALC* implementa-se uma série de operações para determinar qual a posição nos eixos dos yy da *frame* que deve ser preenchido para um determinado valor de entrada (*val*), como mostra a equação 3.6, sendo que *height* representa a altura da imagem transmitida, em pixels. Estas operações são *pipelined*, dando a ilusão que demoram apenas 1 ciclo a serem executadas.

$$valIndex = \frac{(FFFFh - val).(height - 1)}{FFFFh} \quad (3.6)$$

A determinação da posição no eixo dos xx, é feita através do instante de tempo respetivo. Sabendo que é pretendido representar 10 ms de onda em cada *frame*, um pixel no eixo dos xx, representará $10 / width$ ms, sendo que *width* representa o comprimento da imagem em pixels. Assim, após termos a posição do eixo dos xx e yy, é determinado o endereço de escrita, relativo à posição da imagem em memória, no qual é pretendido atribuir a cor verde, através da equação: *addrWR* = *row.width* + *x*, onde *row* é definido pela Main FSM como descrito anteriormente.

Read FSM

Na figura 3.13 está representada a máquina de estados de leitura da BRAM. A cada pulso de relógio comuta-se do estado *S_READ* para *S_HOLD* até que a leitura acabe, isto é, quando *addrRd* chegar ao último endereço da *frame* e se emitir um *vSync*. Cada leitura demora dois ciclos de relógio, em que apenas no primeiro está o *enable* (*en*) ativo.

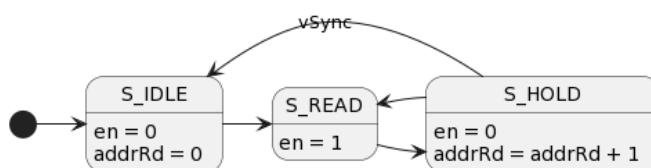


Figura 3.13: Diagrama de estados da Read FSM.

4 Implementação do Sistema

Na figura 4.14 está representado o *block design* de todo o sistema, integrado com todos os IPs, tal como apresentado no capítulo de especificação. No anexo H mostra-se o *block design* ampliado.

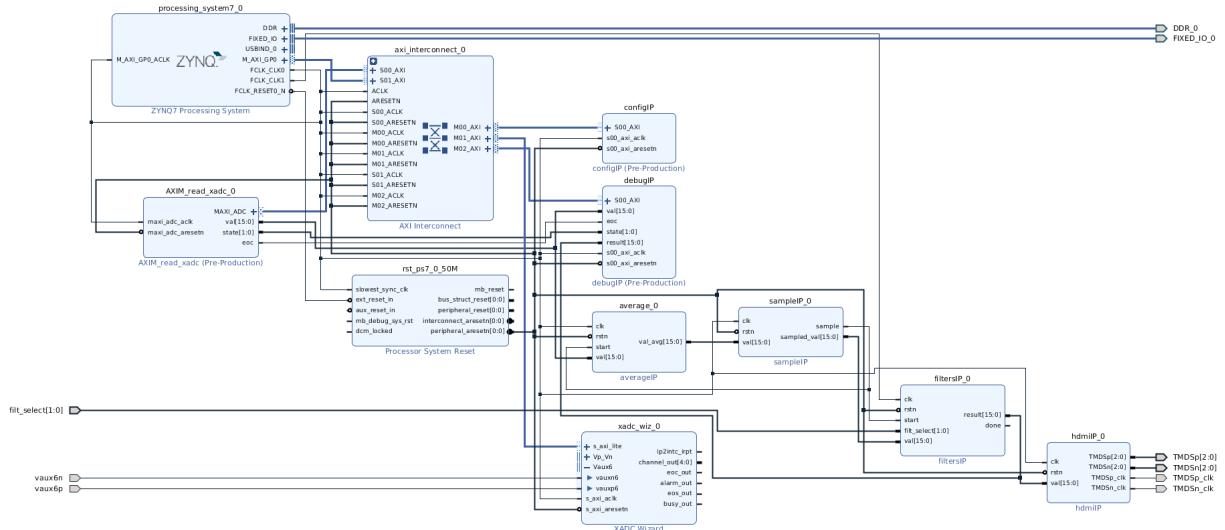


Figura 4.14: *Block Design* implementado para o sistema.

4.1 Amostragem

Na figura 4.15 é mostrada a configuração do IP XADC Wizard, que faz a conversão do sinal analógico para digital e comunica através da interface AXI-Lite. A seleção do canal é *Single Channel*, configurando o canal *Vaux6* como o canal de amostragem. A taxa de amostragem foi ajustada para o máximo (aproximadamente 1 MS/s).

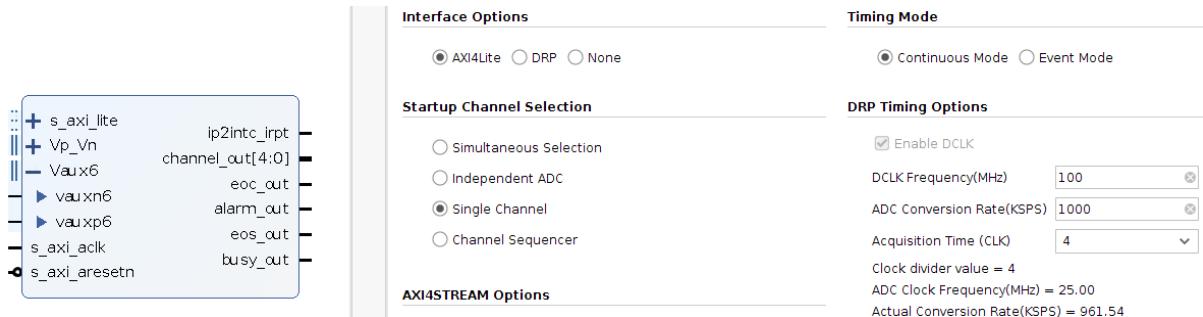


Figura 4.15: Configuração do IP XADC Wizard.

4.2 Filters IP

Na figura 4.16 é apresentado o bloco que implementa um filtro FIR, obtido através da implementação do algoritmo detalhado anteriormente (figura 3.6) no Vivado HLS, usando a linguagem C. No Anexo E pode-se analisar a implementação realizada.

Este bloco gera os sinais de *address* e *chip enable (ce)* para aceder às BRAMs que contém os valores *x_ant* e *x_coefs*. Dependendo se o filtro é um passa baixo ou passa alto, pode ser necessário a adição de um valor contínuo à saída do filtro, existindo para isso a entrada *dcValEn*. O valor filtrado é retornado pela saída *ap_return*.

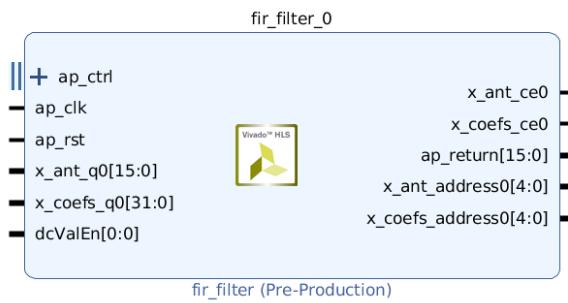


Figura 4.16: Bloco FIR Filter gerado no Vivado HLS.

Na figura 4.17 é apresentada a vista de análise no Vivado HLS. É possível ver-se as operações realizadas pelo bloco FIR Filter para cada ciclo de relógio (*control step*). Inicialmente faz-se o *fetch* de portas I/O e *memory ports* relativas às entradas x_ant e x_coefs . Ao mesmo tempo inicia-se a execução de expressões lógicas e algébricas (anexo E). Além disso, pode ver-se que foram instanciadas 4 DSPs para realizar operações de multiplicação, soma e subtração de 64 bits.

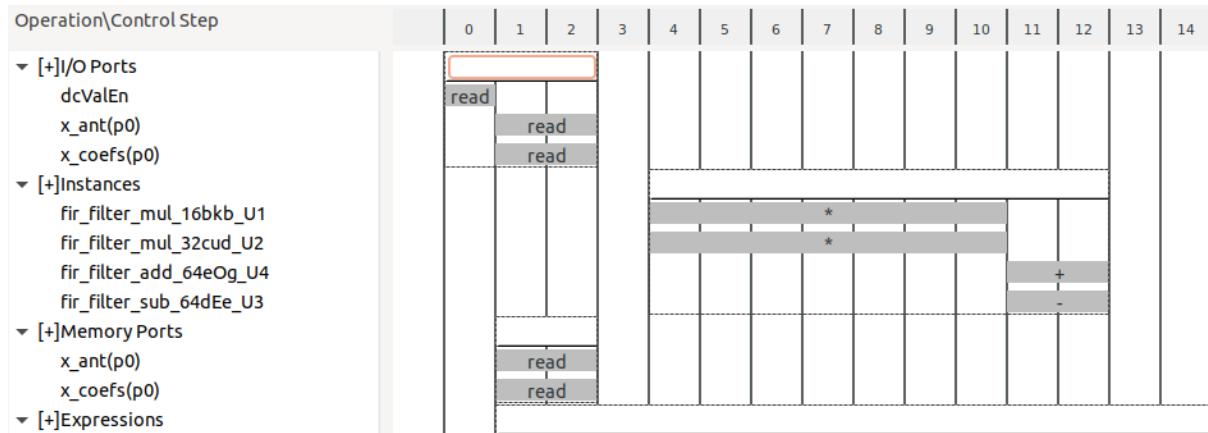


Figura 4.17: Perspetiva de análise no Vivado HLS.

4.3 HDMI IP

O algoritmo de codificação do **TMDS** procura minimizar as transições nas linhas, reduzindo a interferência entre diferentes canais. Mantendo o número de 1s e 0s na linha aproximadamente igual, o equilíbrio DC permite melhorar a margem de ruído.

Na listagem 4.2 é apresentada a implementação do *encoder TMDS*, baseada no fluxograma apresentado no anexo C.

```

1 // count the 1s in data byte
2 wire [3:0] Nb1s = VD[0] + VD[1] + VD[2] + VD[3] + VD[4] + VD[5] + VD[6] + VD[7];
3 wire XNOR = (Nb1s>4'd4) || (Nb1s==4'd4 && VD[0]==1'b0);
4 // minimize transitions
5 wire [8:0] q_m = {~XNOR, q_m[6:0] ^ VD[7:1] ^ {7{XNOR}}, VD[0]};
6
7 // count the 1s in q_m[7:0]
8 wire [3:0] balance = q_m[0] + q_m[1] + q_m[2] + q_m[3] + q_m[4] + q_m[5] + q_m[6] + q_m[7] - 4'd4;
9 reg [3:0] balance_acc = 0;
10 wire balance_sign_eq = (balance[3] == balance_acc[3]);

```

```

11 wire invert_q_m = ( balance ==0 || balance_acc ==0) ? ~q_m[8] : balance_sign_eq ;
12
13 // calculate the difference between number of 1s and 0s in VD
14 wire [3:0] balance_acc_inc = balance - ({q_m[8] ^ ~balance_sign_eq } & ~(balance==0 || balance_acc ==0));
15 // calculate disparity
16 wire [3:0] balance_acc_new = invert_q_m ? (balance_acc - balance_acc_inc) : (balance_acc + balance_acc_inc);
17
18 // output data byte
19 wire [9:0] TMDS_data = { invert_q_m , q_m[8], q_m[7:0] ^ {8{ invert_q_m }}};
20 // send control word based on CD[1] and CD[0]
21 wire [9:0] TMDS_code = CD[1] ? (CD[0] ? 10'b1010101011 : 10'b0101010100) :
22 (CD[0] ? 10'b0010101011 : 10'b1101010100);
23
24 always @(posedge clk) begin
25   TMDS <= (VDE) ? TMDS_data : TMDS_code;
26   balance_acc <= (VDE) ? balance_acc_new : 4'h0;
27 end

```

Listing 4.2: Implementação do codificador TMDS do HDMI IP.

5 Resultados do Sistema

5.1 Amostragem

Tal como mencionado na subsecção 3.3.1, foi implementada um AXI *master* que faz a leitura do *slave* XADC. Na figura 5.18 está representado o *Block Design* que testa o AXI *master* responsável pela leitura do IP *slave* XADC. Para este teste foi usado apenas um *slave*, com 2 registos, sendo que um simula o registo de configuração (endereço 44A00000h) e outro o registo do XADC (endereço 44A00004h).

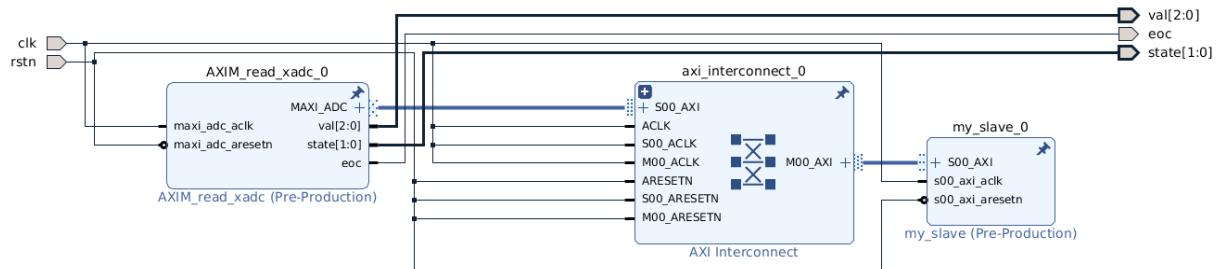


Figura 5.18: *Block Design* do teste ao AXI *master* de leitura do XADC.

Na figura 5.19 está representado o resultado do *testbench* ao AXI *master* responsável pela leitura do IP *slave* XADC.



Figura 5.19: Resultado do *testbench* do AXI *master* de leitura do XADC.

É possível observar que no estado 0, o *master* realiza uma leitura ao registo de configuração, *ARADDR*. A máquina de estados avança para o estado 1, onde efetua a comparação do valor lido, *RDATA*. Como o registo de configuração está a 1, então a máquina de estados avança para o estado 2, onde são feitas leituras consecutivas. A cada leitura, o sinal de *eoc* é ativo, para sinalizar o fim de uma leitura e o valor lido é colocado na saída, *val*.

5.2 Filters IP

Resultados Esperados

De forma a testar o correto funcionamento do filtro **FIR**, implementou-se em Matlab um conjunto de funções que permitem simular o funcionamento do filtro. Desta forma foi possível obter os resultados esperados para valores de entrada conhecidos. A implementação deste *testbench* é apresentada no anexo F.1. Assim, tornou-se possível a utilização de vetores (*golden vectors*) para validação dos filtros.

Na figura 5.20 pode ver-se a resposta de um **LPF** com frequência de corte de 40 Hz, para uma entrada sinusoidal de 20 Hz e 100 Hz. É possível verificar o começo em funcionamento do filtro na figura 5.20 b), onde o valor da saída do filtro tende para o valor desejado ao fim de alguns períodos. Este valor digital corresponde a aproximadamente 32767, igual a metade do valor máximo medido pelo ADC, sendo assim equivalente a uma onda sinusoidal de valor médio nulo. O mesmo estudo foi realizado para o **HPF** e **BPF**, como mostra o anexo G.1.

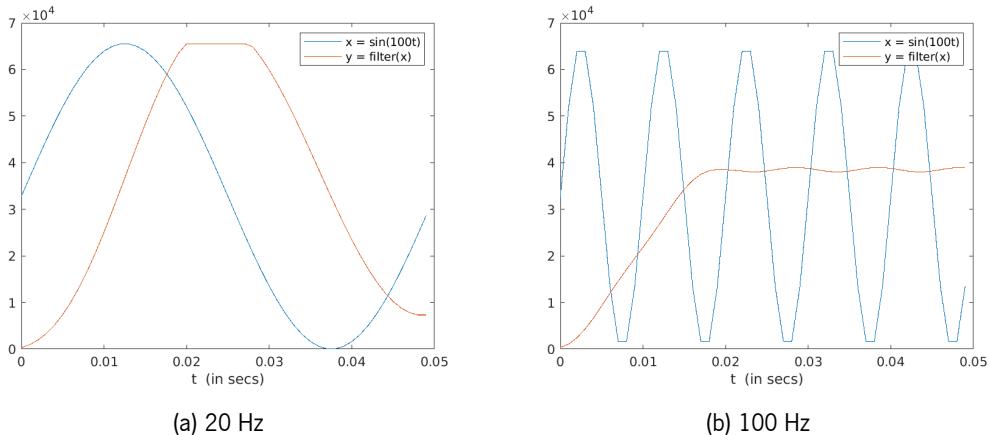


Figura 5.20: Saída do LPF, a vermelho, em função da entrada, a azul.

Utilizando os *golden vectors* criados no Matlab, escreveu-se um *testbench* no Vivado HLS que insere valores conhecidos na entrada do filtro e verifica se a saída corresponde ao esperado, como apresentado no anexo F.2.

Na figura 5.21, apresenta-se o resultado do *testbench*, confirmando-se o correto funcionamento dos três filtros, **LPF**, **HPF** e **BPF**, às frequências de 20 Hz, 100 Hz, 120 Hz e 220 Hz.

```
***** C Simulation *****
LPF Test @20Hz - PASSED
LPF Test @100Hz - PASSED
HPF Test @20Hz - PASSED
HPF Test @100Hz - PASSED
BPF Test @20Hz - PASSED
BPF Test @120Hz - PASSED
BPF Test @220Hz - PASSED
***** End C Simulation *****
INFO: [SIM 211-1] CSim done with 0 errors.
```

Figura 5.21: Resultado da execução do testbench no Vivado HLS.

Resultados em Simulação

Através de um *testbench* criado no Vivado visualizou-se as formas de onda envolvidas no bloco de Filters IP através de uma simulação comportamental, como mostra a figura 5.22. Foi aplicado um LPF, como se pode ver na entrada *filt_select* a 0. É possível ver a produção do sinal *filt_done* aquando o término da filtragem do novo valor de entrada, *input_val*.

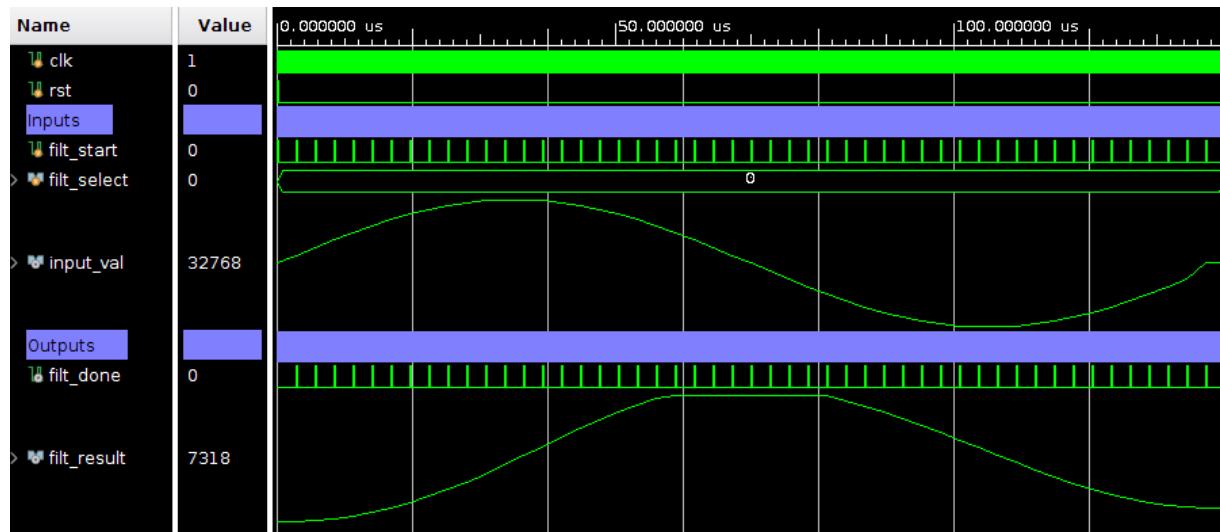


Figura 5.22: Simulação comportamental do bloco de filtros aplicando um LPF a uma onda de entrada de 20 Hz.

Na figura 5.23 vê-se a aplicação do LPF a uma onda de entrada de 100 Hz, que tal como esperado é atenuada.

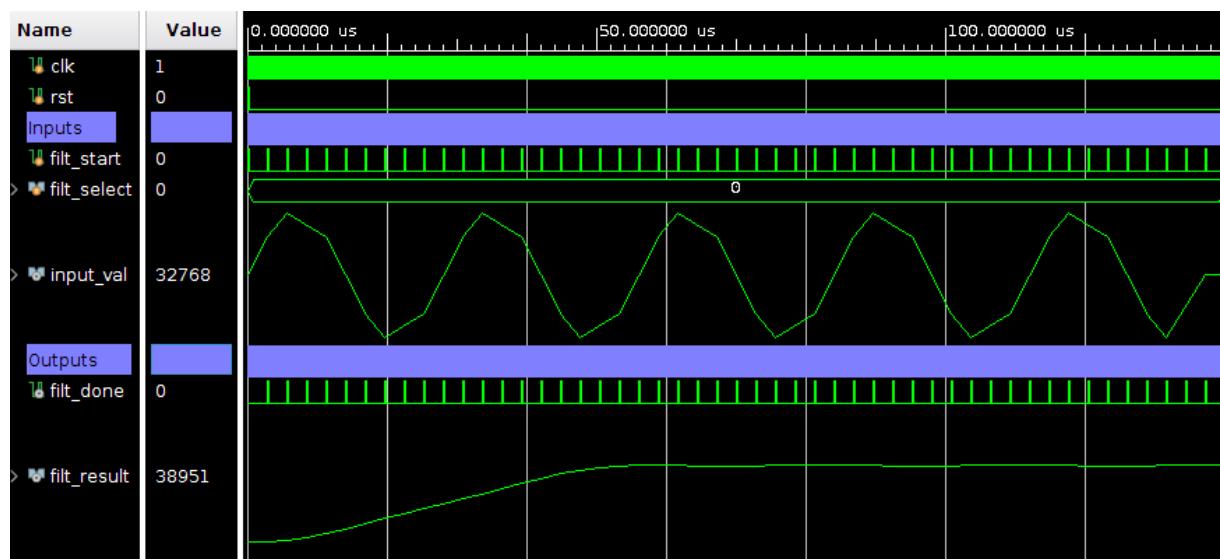


Figura 5.23: Simulação comportamental do bloco de filtros aplicando um LPF a uma onda de entrada de 100 Hz.

Tal como foi realizado no Vivado HLS, adequou-se esse *testbench* para Verilog, sendo usado na simulação comportamental mostrada acima. Na figura 5.24 pode verificar-se novamente que os valores obtidos coincidem com os valores esperados.

Testing LPF:

```
Writing output results to ../../../../../../sim/LPF/LPF_100sim_output.txt
Comparing output results with ../../../../../../golden_vectors/LPF/LPF_100out_golden.txt ...
Simulation completed with 0 errors!
```

Figura 5.24: Resultado da simulação comportamental do bloco de filtros.

Resultados Experimentais

Inicialmente utilizou-se a placa de desenvolvimento STM32 para validar o funcionamento dos filtros, utilizando o seu **ADC** para receber o sinal de entrada e dispondo num **Digital to Analog Converter (DAC)** o valor filtrado, sendo assim possível visualizar as duas ondas num osciloscópio externo. Na figura 5.25 vê-se que os resultados obtidos coincidem com os esperados (figura 5.20). O mesmo teste experimental foi realizado para o **HPF** e **BPF**, como mostra o anexo G.2.



Figura 5.25: Saída do LPF, a azul, em função da entrada, a amarelo.

Testando agora o bloco de filtros na Zybo, pode ser usado o bloco *debugIP* apresentado anteriormente para disponibilizar o valor calculado pelo bloco de filtros na **PS**. Através da ferramenta Vitis criou-se uma aplicação *bare metal* para ler o registo AXI relativo ao valor filtrado e apresenta-los num terminal, através da interface de comunicação UART. Assim, foi possível traçar os gráficos apresentados na figura 5.26 e no anexo G.3. Verifica-se o correto funcionamento dos filtros.

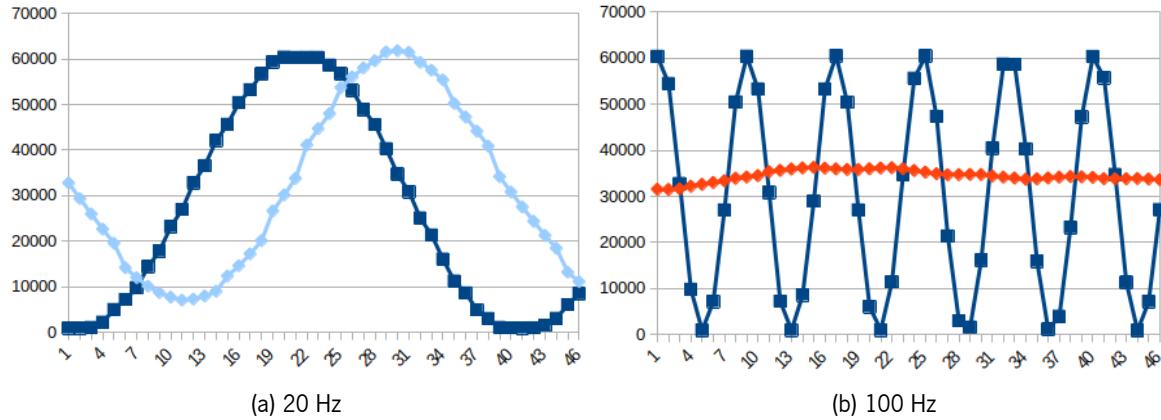


Figura 5.26: Bloco de filtros com o LPF selecionado para uma entrada, a azul escuro, de: a) 20 Hz - Saída a azul claro; b) 100 Hz - Saída a vermelho.

5.3 HDMI IP

Como mencionado na secção 3.3.3, o bloco de HDMI é regido por uma máquina de estados que controla a leitura e escrita de uma *frame* nas duas BRAMs. Nesta simulação definiu-se uma resolução de 4x4 para a *frame* e colocou-se um valor constante à entrada do bloco igual a 65535. Isto quer dizer que a primeira linha da *frame* deve ser preenchida com pixels verde, enquanto o resto se mantém a preto.

Na figura 5.27 apresenta-se a simulação comportamental do HDMI IP para o estado de limpeza (*S_CLEAN*) e escrita da *frame*, onde é executada a FSM Write. No caso da figura, a BRAM0 está a ser escrita, nos *addresses* definidos por *addrB0*, estando para isso a entrada de *enable* (EN0) e *write-enable* (WE0) ativas. Quando o *write-data* está a 0 é efetuado o *clean* da BRAM, e quando está a 1 é efetuada a escrita.

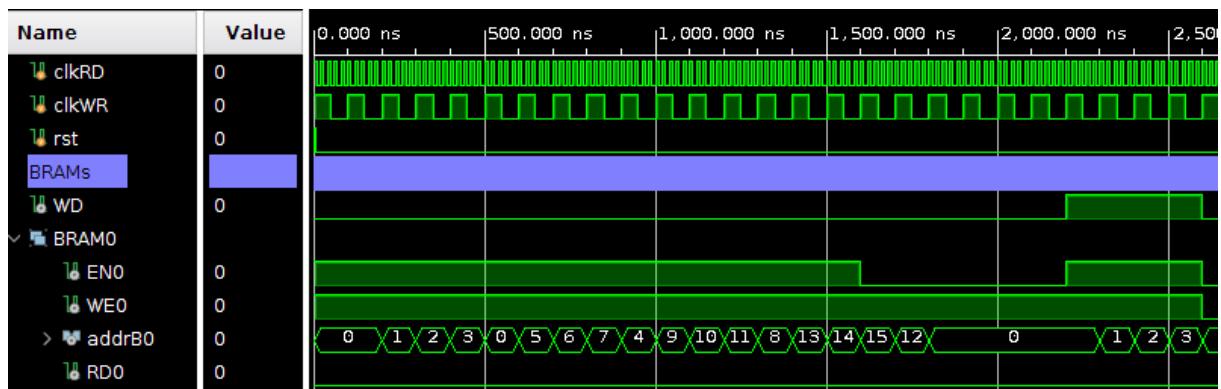


Figura 5.27: Simulação comportamental do bloco HDMI - Escrita da frame.

Na figura 5.28 são apresentadas as formas de onda envolvidas na FSM Read. Para efetuar a leitura faz-se o controlo do EN0, mantendo o WE0 a 0 durante toda a operação. Neste caso, a BRAM0 está a ser lida, e nos *addresses* de 0 a 3 têm todos o conteúdo igual a 1. Numa *frame* de largura igual a 4 pixels, isto significa que a primeira linha da *frame* deve estar toda preenchida, como se pode ver nos sinais de controlo da interface HDMI. O pixel a ser transmitido é 0x00ff00 (pixel de cor verde) apenas para a primeira linha da *frame*. Cada linha é enviada durante um período a 1 de *VDEn*. Quanto este período acaba, o sinal *hSync* indica que uma linha foi transmitida. Ao fim das 4 linhas serem transmitidas, o sinal *vSync* indica que a *frame* foi toda transmitida.

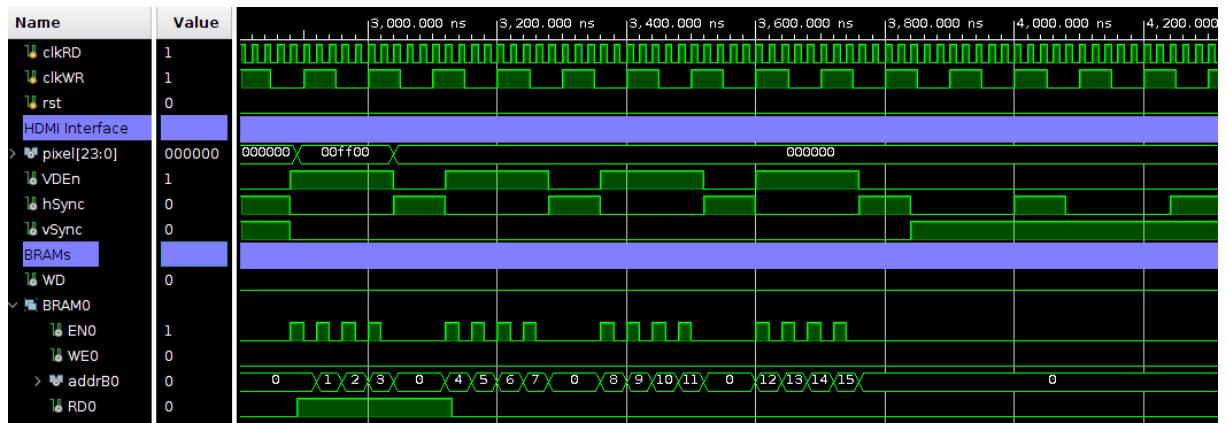


Figura 5.28: Simulação comportamental do bloco HDMI - Leitura da frame.

Na figura 5.29 pode ver-se o funcionamento da máquina de estados principal do HDMI IP. Verifica-se que as BRAMs são lidas e escritas de forma alternada, e que (para estas frequências de *clkRD* a 50 MHz e *clkWR* a 10 MHz) existe uma escrita de uma *frame* nova a cada duas leituras de uma *frame* antiga.



Figura 5.29: Simulação comportamental do bloco HDMI.

5.4 Resultados Finais

Na figura 5.30 é apresentada a montagem realizada para os testes finais ao sistema desenvolvido. De forma a facilitar os testes, utilizou-se a STM32 como gerador de sinais, colocando no seu **DAC** uma onda sinusoidal com frequência máxima de 100 Hz. Por esse motivo, visto que o **XADC** da Zybo suporta 1 V de amplitude de tensão de entrada, implementou-se um divisor resistivo, sucedido de um seguidor de tensão, para converter o sinal produzido pela STM32 com 3,3 V de amplitude, para 1 V. Além disso, pode ver-se o cabo HDMI conectado à interface HDMI Tx da Zybo.

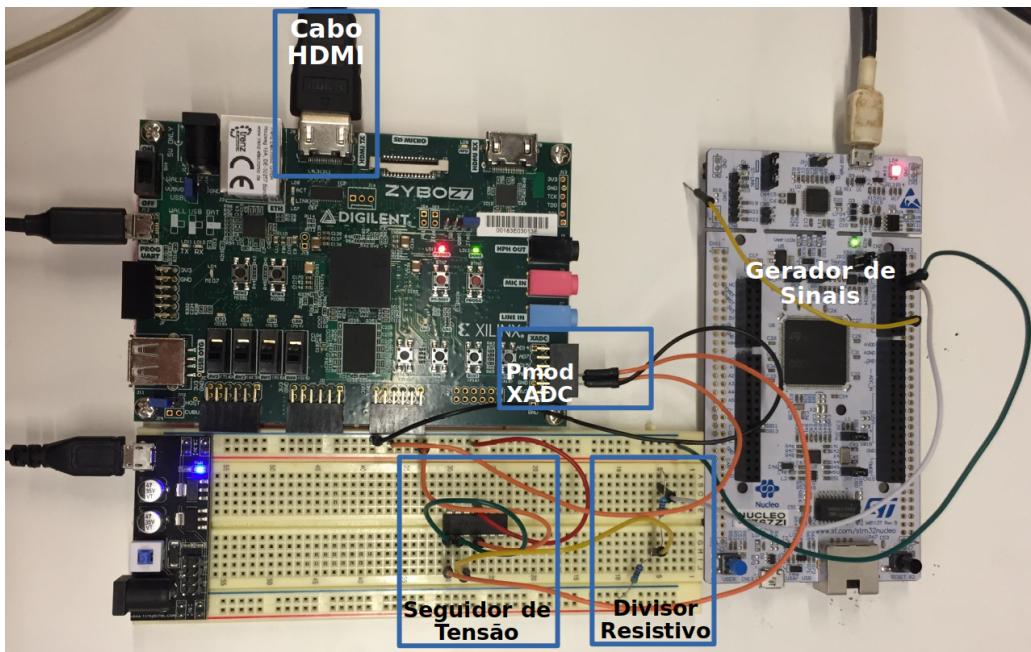


Figura 5.30: Montagem realizada para os testes finais.

Na figura 5.31 mostra-se a representação de um sinal sinusoidal a diferentes frequências num ecrã, não estando nenhum filtro aplicado ao sinal de entrada.

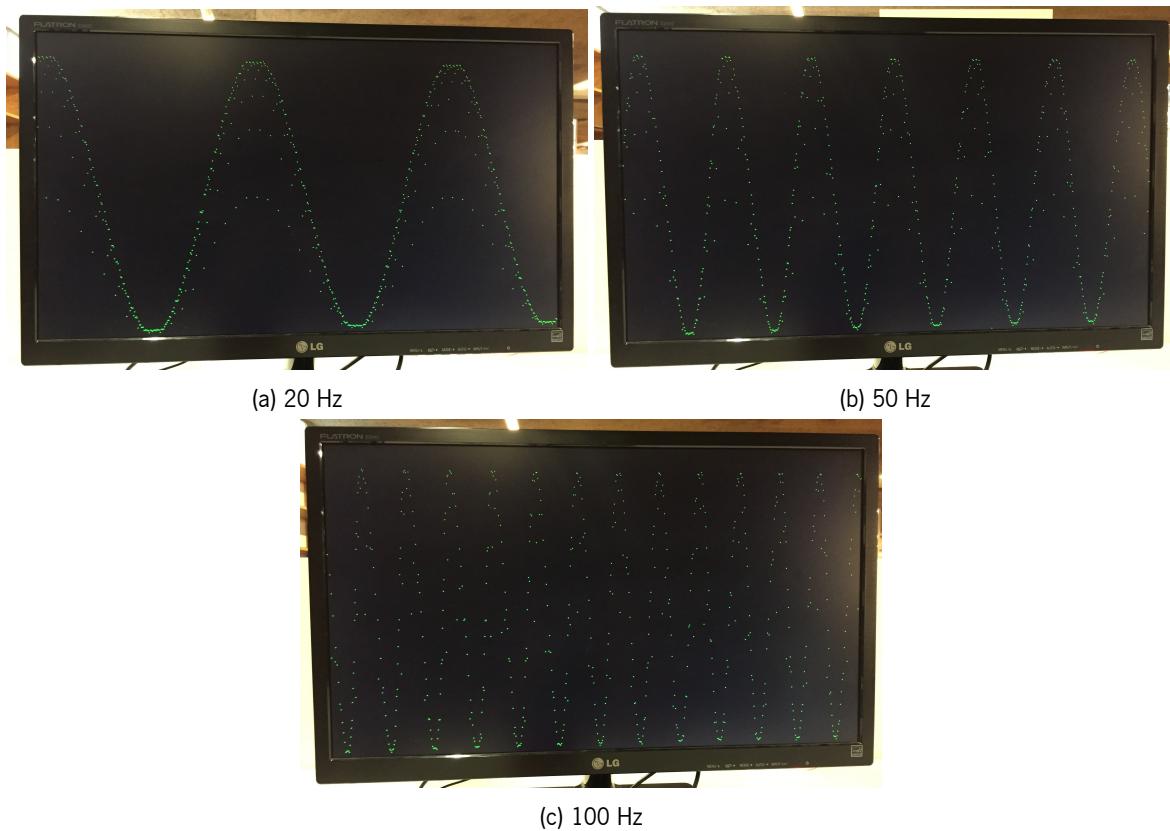


Figura 5.31: Visualização do sinal de entrada a diferentes frequências.

Na figura 5.32 é possível visualizar a saída do LPF para vários sinais de entrada com frequências diferentes, tal como para o HPF e BPF no anexo G.4. Comparando com os resultados obtidos na STM32, e com os resultados

esperados, garante-se assim a funcionalidade do desenho do sistema.

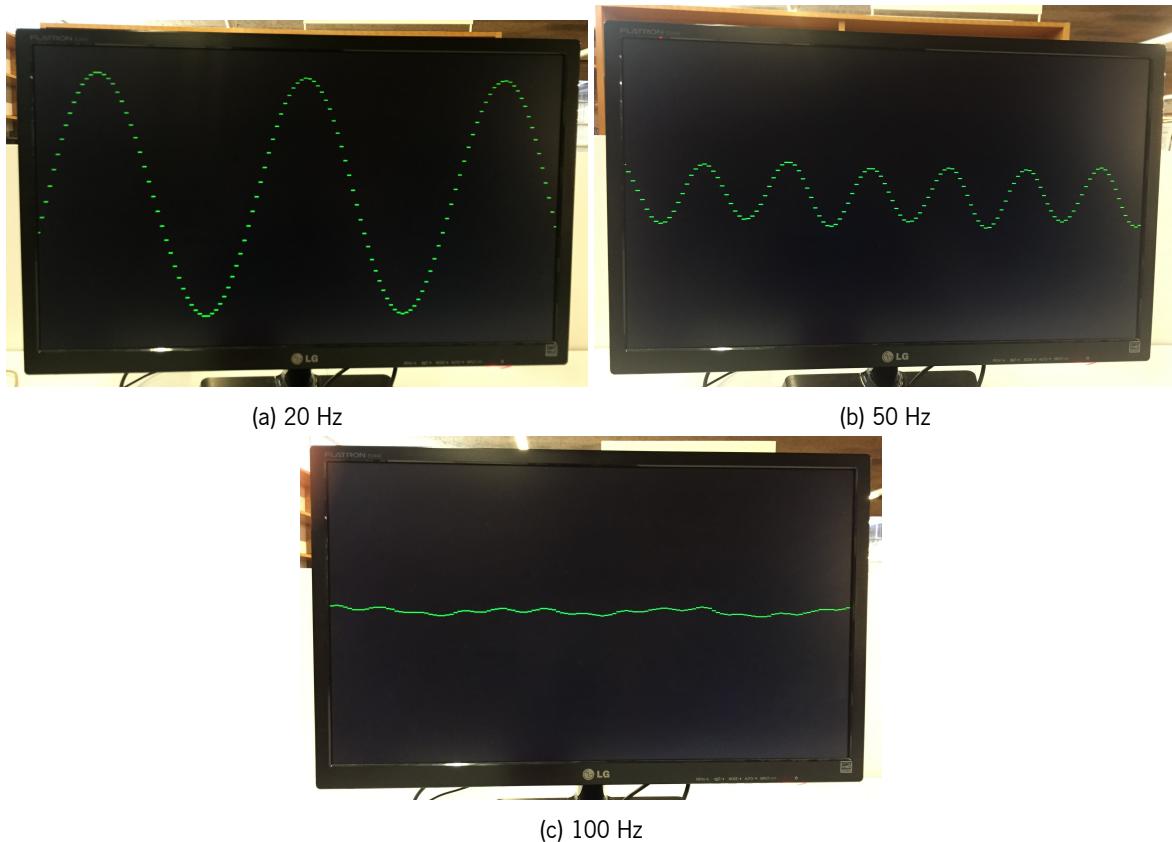


Figura 5.32: Visualização da saída do LPF para vários sinais de entrada com frequências diferentes.

5.5 Utilização de Hardware

Na figura 5.33 mostra-se o sumário de utilização gerado pelo Vivado para todo o sistema, realçando-se a percentagem de utilização de BRAMs e DSPs.

LUT %	FF %	BRAM %	DSP...	URAM %
0.00	0.00	0.0	0.00	0.00
11.34	10.03	33.3	18.75	0.00

Figura 5.33: Sumário de utilização de hardware para todo o sistema.

6 Conclusão

O projeto apresenta a implementação de um osciloscópio com funcionalidades básicas, desenvolvido em **FPGA**, utilizando **HLS** para o desenvolvimento de filtros digitais, explorando assim técnicas de **DSP**. Implementou-se também um controlador de vídeo em *hardware* para uma interface HDMI, que faça o *display* de uma onda, resultado do sinal filtrado.

Este projeto revelou-se desafiante pela introdução ao desenho em *hardware*, através de **HDL** e também de **HLS**, permitindo consolidar conhecimentos adquiridos nas unidades curriculares de Processamento de Sinal, Sistemas Embbebidos e Sistemas Digitais. Além disso, esta foi mais uma oportunidade para aumentar a experiência na utilização de C, Matlab, e da linguagem recentemente aprendida, Verilog. Permitiu também a possibilidade de trabalhar com ferramentas da Xilinx, nomeadamente o Vivado, Vitis e Vivado HLS, que são bastante usadas na indústria. Uma grande aprendizagem a retirar é a criação e utilização de *testbenches*. A utilização destes módulos, quando corretamente desenvolvidos, são cruciais para que em ambiente de simulação se verifique o correto funcionamento do sistema. Outra ideia a reforçar é a importância de um desenho do sistema completo e verificado. Na programação de *hardware* uma falha no desenho pode-se tornar incontornável face ao tempo restante para resolver o problema, enquanto que na programação de *software* tal pode não ser necessariamente verdade.

Relativamente aos objetivos do trabalho, verificou-se a existência de algumas limitações em relação à frequência de atualização da interface gráfica, tal como na aplicação de filtros passa-alto e passa-banda. A utilização de **HLS** no desenvolvimento dos filtros permitiu a sua rápida implementação, cumprindo assim um dos seus propósitos. Face à escassez de tempo, não foi possível comparar a performance de *hardware* gerado por **HLS** e **HDL**, bem como, a razão tempo de desenvolvimento / qualidade de *hardware* gerado para estas duas abordagens de desenvolvimento.

Como trabalho futuro identifica-se o aumento da frequência de atualização da interface gráfica, tal como a adição de mais funcionalidades ao sistema que permitam o controlo das escalas temporal e de tensão do osciloscópio, a representação de um eixo vertical e horizontal para facilitar a identificação da amplitude e frequência da onda. Seria relevante ainda implementar a função de *trigger*, representação de várias ondas na interface gráfica, sendo para isso necessária a utilização de vários canais **ADC**. Como substituição à interface HDMI poder-se-ia utilizar um ecrã tátil que permitiria o encapsulamento do sistema, como é o caso de um osciloscópio tradicional.

Bibliografia

- [1] DigiKey, "Tmds encoder (vhdl)," acedido em 18 junho 2022. [Online]. Available: <https://forum.digikey.com/t/tmds-encoder-vhdl/12653>
- [2] D. H. Sarah L. Harris, *Digital Design and computer architecture: RISC-V edition.* Morgan Kaufmann, 2021, ch. 4.
- [3] Tektronix, "Test and measurement equipment," acedido em 16 março 2022. [Online]. Available: <https://www.tek.com/en>
- [4] —, "Asic powers next-generation oscilloscopes," acedido em 16 março 2022. [Online]. Available: <https://www.tek.com/en/documents/whitepaper/tek049-asic-powers-next-generation-oscilloscopes>
- [5] N. Instruments, "Drive innovation and productivity," acedido em 16 março 2022. [Online]. Available: <https://www.ni.com/pt-pt.html>
- [6] —, "Get better measurements faster using oscilloscopes with user-programmable fpgas," acedido em 16 março 2022. [Online]. Available: <https://www.ni.com/pt-pt/innovations/white-papers/14/get-better-measurements-faster-using-oscilloscopes-with-user-pro.html>
- [7] Xilinx, "Vivado 2021.2 - high-level synthesis," acedido em 16 março 2022. [Online]. Available: <https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0012-vivado-high-level-synthesis-hub.html>
- [8] Intel, "Intel high level synthesis compiler," acedido em 16 março 2022. [Online]. Available: <https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html>
- [9] Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer, "Parallel programming for fpgas," 2018, acedido em 1 março 2022. [Online]. Available: <https://kastner.ucsd.edu/wp-content/uploads/2018/03/admin/pp4fpgas.pdf>
- [10] Alan V. Oppenheim, Ronald W. Schafer, John R. Buck, *Discrete-Time Signal Processing*, 2nd ed. New Jersey: Prentice Hall, 1999.
- [11] U. Hussaini, "What is aliasing in dsp and how to prevent it," 2020, acedido em 20 março 2022. [Online]. Available: <https://technobyt.org/whats-aliasing-dsp-how-to-prevent-it/>
- [12] Digilent, "Zybo z7 board reference manual," 2018, acedido em 20 março 2022. [Online]. Available: https://digilent.com/reference/_media/reference/programmable-logic/zybo-z7/zybo-z7_rm.pdf
- [13] *Zynq-7000 SoC Technical Reference Manual*, AMD Xilinx, acedido em 11 abril 2022. [Online]. Available: <https://docs.xilinx.com/v/u/en-US/ug585-Zynq-7000-TRM>
- [14] *7 Series FPGAs and Zynq-7000 SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter User Guide*, Xilinx, acedido em 16 março 2022. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug480_7Series_XADC
- [15] Xilinx, "Pg091 - xadc wizard v3.3 logicore ip product guide," acedido em 7 maio 2022.
- [16] R. Oshana, *DSP For Embedded And Real-Time Systems*. Elsevier, 2012, ch. 1.

- [17] arm Developer, "Amba overview," acedido em 11 abril 2022. [Online]. Available: <https://developer.arm.com/architectures/system-architectures/amba>
- [18] —, "Axi protocol overview," acedido em 11 abril 2022. [Online]. Available: <https://developer.arm.com/documentation/102202/0200/AXI-protocol-overview>
- [19] *Vivado Design Suite AXI Reference Guide*, AMD Xilinx, acedido em 11 abril 2022. [Online]. Available: <https://docs.xilinx.com/v/u/en-US/ug1037-vivado-axi-reference-guide>

Aliasing

Como mencionado na secção 2.3, o *aliasing* acontece quando o teorema de Nyquist não é respeitado. Na figura A.1 a) pode ver-se uma onda sinusoidal a ser amostrada a uma frequência inferior à frequência de Nyquist. Na figura A.1 b) pode ver-se que a onda reconstruída pelas amostradas adquiridas é totalmente diferente do sinal de entrada.

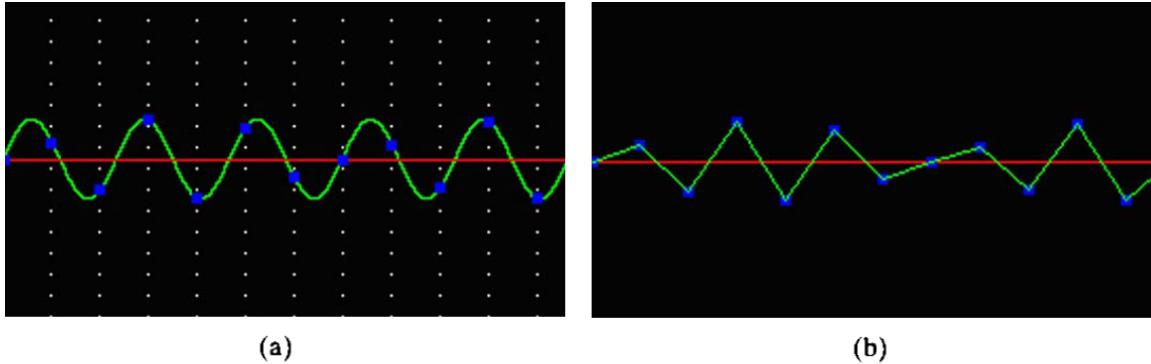


Figura A.1: Efeito do *aliasing* no domínio dos tempos: a) Sinal de entrada; b) Sinal amostrado.

Além disso, no domínio das frequências, pode-se verificar uma sobreposição dos espetros dos sinais, como representado na figura A.2. Na alínea a) é possível ver-se o espetro do sinal de entrada, X_C , com frequência normalizada Ω_N . Recordando a figura 2.1 do conversor tempo contínuo - tempo discreto, na figura A.2 b) está apresentado o trem de impulsos S , espaçados pela frequência de amostragem normalizada, Ω_S . Na alínea c) está presente o resultado da multiplicação de X_C com S , verificando-se uma série de espetros de frequência Ω_N a cada Ω_S . É de notar que a condição para que os espetros não se sobreponham é: (equação A.1).

$$\Omega_S - \Omega_N \geq \Omega_N \quad (\text{A.1})$$

Simplificando a equação A.1 obtém-se $\Omega_S \geq 2\Omega_N$, que remete para o teorema de Nyquist (equação 2.1).

Na alínea d) é possível ver a sobreposição dos espetros, quando $\Omega_S - \Omega_N < \Omega_N$. Isto provoca uma deformação no sinal, impossibilitando a reconstrução do sinal original. Para evitar este efeito, deve ser utilizado um **LPF**, com frequência de corte igual a $\Omega_S/2$. Desta forma é possível remover as componentes de alta frequência, eliminando a probabilidade de haver sobreposição de espetros. Na figura A.3 é possível ver a resposta em frequência de um **LPF** ideal, com frequência de corte w_c . [10] [16]

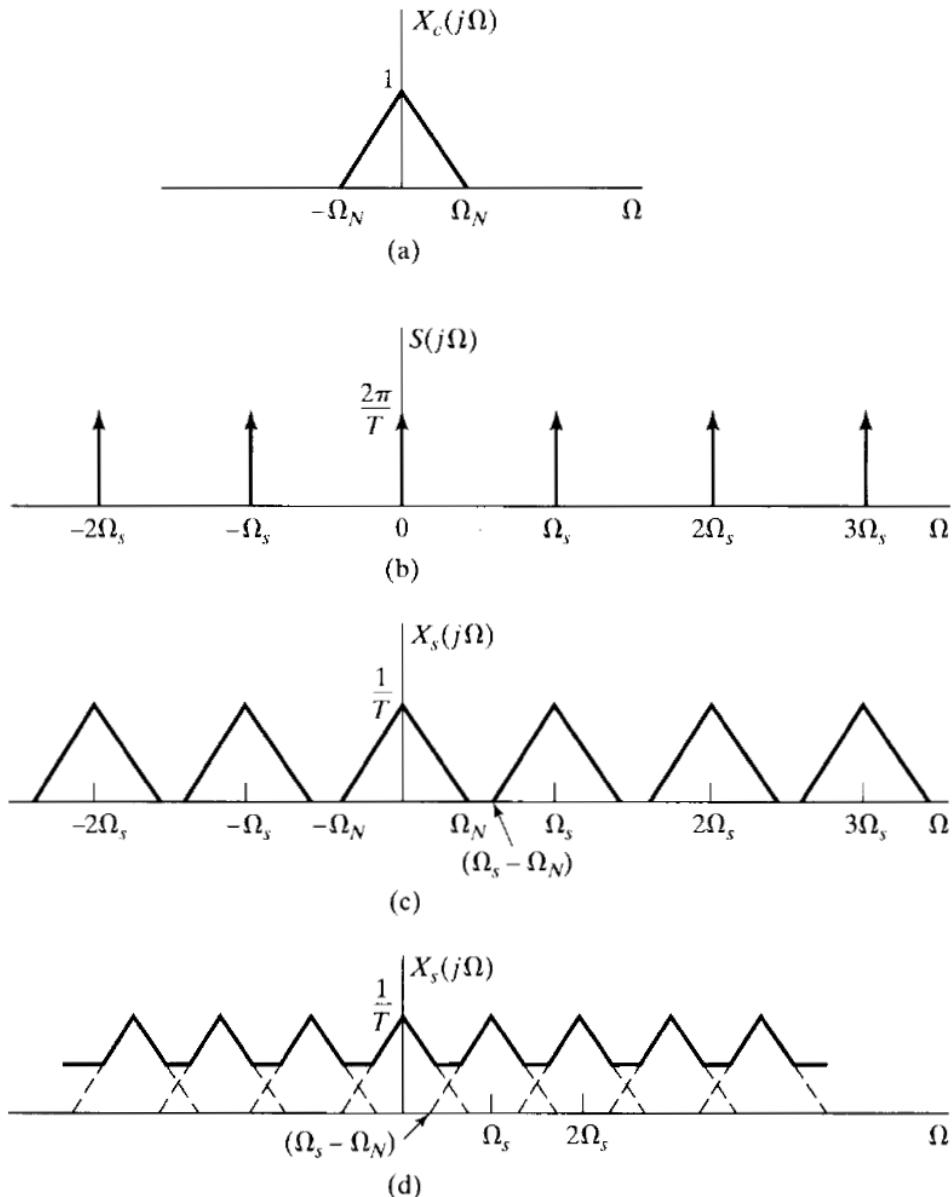


Figura A.2: Efeito do *aliasing* no domínio das frequências: a) Espetro do sinal de entrada; b) Espetro do trem de impulsos; c) Espetro do sinal amostrado com $\Omega_S > 2.\Omega_N$; d) Espetro do sinal amostrado com $\Omega_S < 2.\Omega_N$

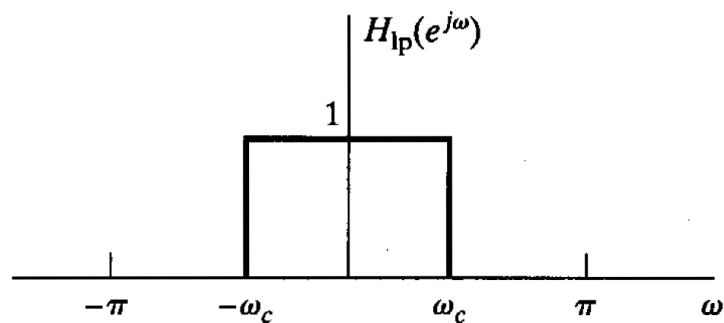


Figura A.3: Filtro *anti-aliasing*: resposta em frequência.

Protocolos AMBA

Advanced Microcontroller Bus Architecture (AMBA) é uma norma aberta para conexão e gestão de blocos funcionais num *SoC*. Facilita o correto desenvolvimento de *designs* com múltiplos processadores, com um grande número de controladores e periféricos. [17]

Advanced eXtensible Interface (AXI) é uma especificação de interface que define a interface de blocos IP, em vez da interconexão propriamente dita. O protocolo AXI é uma especificação ponto-a-ponto, portanto, descreve apenas os sinais e *timings* entre interfaces.

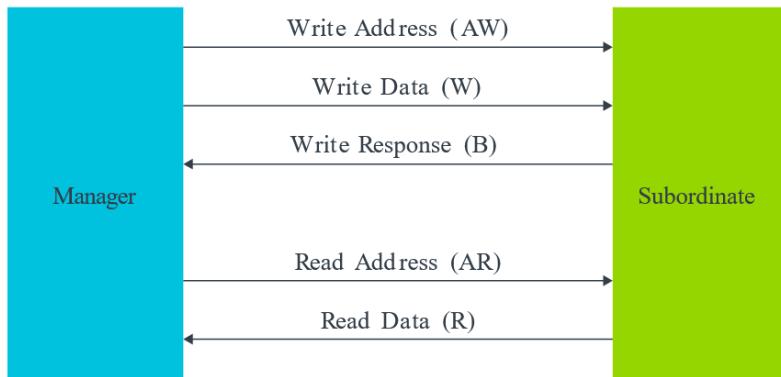


Figura B.1: Canais usados pela interface *AXI*.

Operações de escrita usam os seguintes canais:

- O gestor envia um endereço pelo canal **Write Address (AW)** e transfere dados pelo canal **Write Data (W)** para o subordinado.
- O subordinado escreve os dados recebidos no endereço especificado. Quando o subordinado completar a operação de escrita, responde com uma mensagem para o gestor através do canal **Write Response (B)**.

Operações de leitura usam os seguintes canais:

- O gestor envia o endereço que quer ler através do canal **Read Address (AR)**.
- O subordinado envia os dados do endereço requisitado para o gestor, através do canal **Read Data (R)**. O subordinado pode também retornar uma mensagem de erro usando este canal. Um erro ocorre se, por exemplo, o endereço não for válido ou os dados estiverem corrompidos.

Todos estes canais partilham o mesmo mecanismo de *handshake* baseado em sinais **VALID** e **READY**. Numa transmissão de dados da fonte para o receptor, o sinal de **VALID** é enviado pela fonte, para indicar que existe informação válida disponível. O sinal de **READY** é produzido pelo receptor quando está disponível para receber informação. [18]

A *AXI* faz uma distinção entre transferências de dados e transações: uma **transferência** é uma única troca de informação com um *handshake*; uma **transação** é uma rajada de transferências, contendo um endereço de transferência, uma ou várias transferências de dados e, para sequências de escrita, uma transferência de resposta.

AXI4 é a quarta versão de *AXI*. Existem três tipos de interfaces *AXI4*: [19]

- **AXI4**: adequado para controlar muitos periféricos; suporta transações (máximo de 256 transferências) para múltiplos periféricos e diferentes espaços de endereços de cada vez;
- **AXI4-Lite**: adequado para comunicações simples, com baixa largura de banda; transações com apenas 1 transferência; para blocos de dados de 32bits.
- **AXI4-Stream**: para transmissão de dados de alta velocidade.

Encoder TMDS

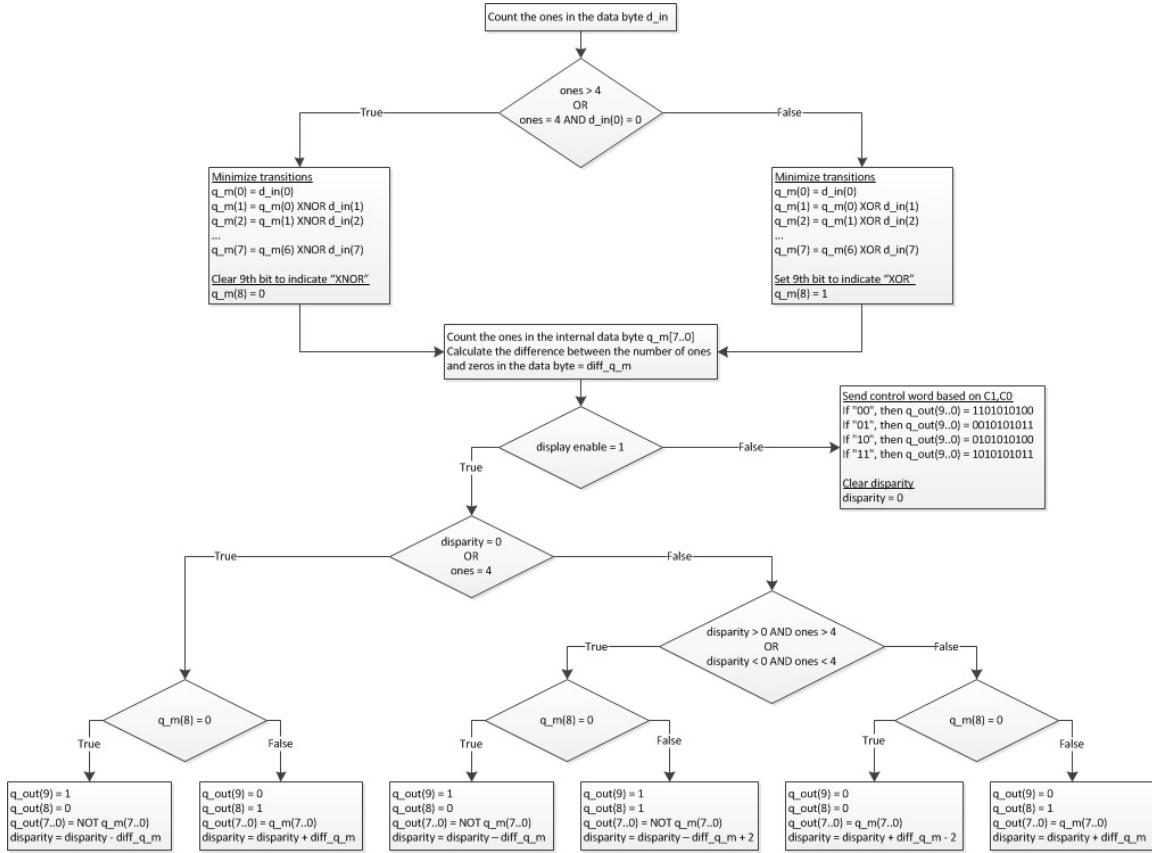


Figura C.1: Fluxograma do encoder TMDS. [1]

Influência da ordem dos filtros

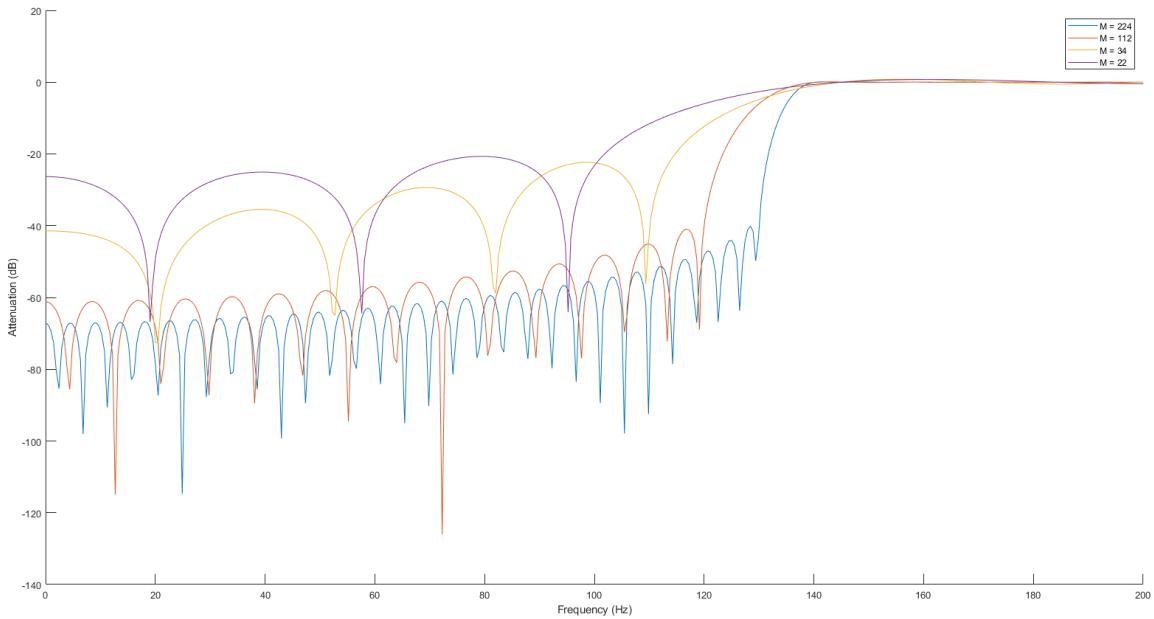


Figura D.1: Resposta em frequência de um **HPF** para diferentes ordens.

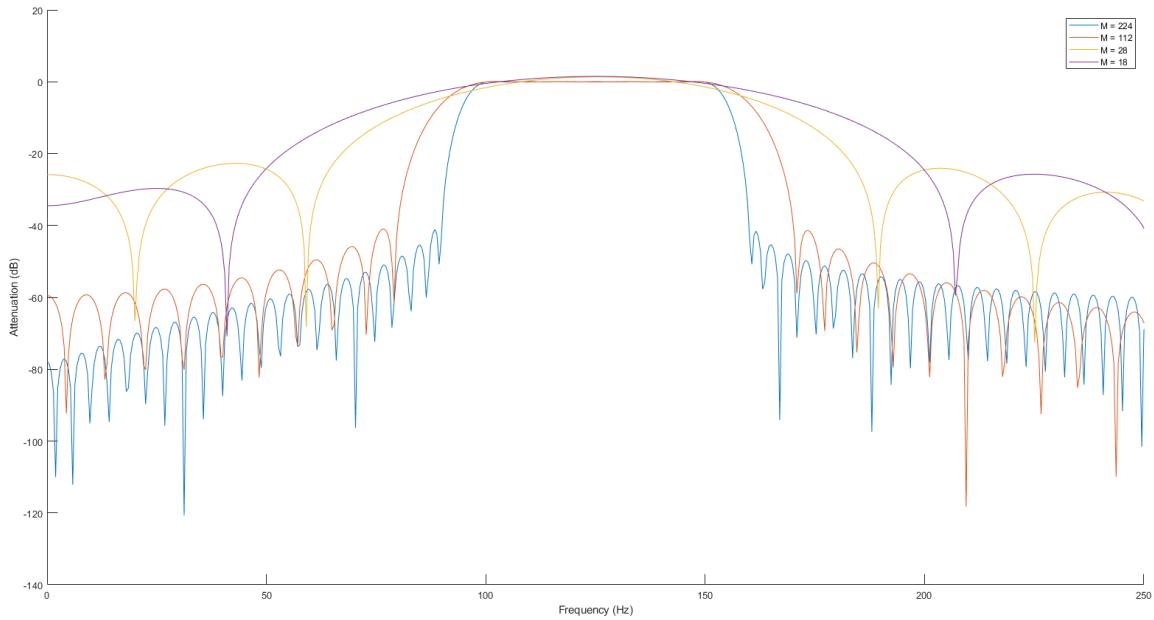


Figura D.2: Resposta em frequência de um **BPF** para diferentes ordens.

Implementação de um filtro FIR em HLS

```

1 #define _M_ 22
2 #define VAL_MAX 65535
3 #define DCVAL    32767 // = 65535/2
4 typedef uint32 fixed_point_t ;
5
6 uint16 fir_filter ( uint16 x_ant[_M_+ 1], fixed_point_t x_coefs [_M_+ 1], uint1 dcValEn ) {
7     // aux variable in filter execution
8     uint64 y64b = 0;
9     uint32 y32b;
10    uint16 i;
11
12    for(i = 0; i < _M_+1; i++) {
13        /* Multiply x_ant with x_coefs .
14         * x_coef may be negative so we need to check it
15         *
16         * (x_coefs [i]>>31) is the MSB: if asserted then it is a negative number
17         *
18         * If it is negative then:
19         *   ~(x_coefs [i] - 1) returns the number in 2's complement
20         *   So, we got: y64b = y64b - x_ant[i] * x_coefs [i]
21         *
22         * If it is not negative then:
23         *   We got y64b = y64b + x_ant[i] * x_coefs [i]
24         */
25        if ((x_coefs [i]>>31) & 0x01)
26            y64b = y64b - x_ant[i] * ( ~(x_coefs [i] - 1));
27        else
28            y64b = y64b + x_ant[i] * x_coefs [i];
29    }
30    // convert from fixed point
31    y32b = (y64b>>15);
32    // add dc value if needed. y may be negative , so we need to check it
33    if ((y32b>>31) & 0x01)
34        y32b = dcValEn*DCVAL - ( ~(y32b - 1));
35    else
36        y32b = dcValEn*DCVAL + y32b;
37
38    // truncate output value if it is a negative number
39    if ((y32b>>31) & 0x01)
40        y32b = 0;
41    // or if it is greater than 2^16-1 = ADC resolution
42    else if (y32b > VAL_MAX)
43        y32b = VAL_MAX;
44
45    return ( uint16 )(y32b);
46 }
```

Listing E.1: Implementação de um filtro FIR em HLS

Testbenches

F.1 Geração dos resultados esperados dos filtros

```

1 function filters_tb (Fc, filter , printFig , writeToFile )
2 % ----- SIMULATION PARAMETERS
3 % sampling frequency
4 fsamp = 1000;
5 % adc resolution
6 adcRes = 65535;
7
8 % ----- DESIGN FILTER
9 if (filter == "LPF") % low pass filter - order 22
10    fcuts = [21 60];
11    devs = [0.1 0.1];
12    [hh,M]= low_pass_filter (fsamp, fcuts , devs);
13    dcValEn=0;
14 elseif (filter == "HPF") % high pass filter - order 22
15    fcuts = [50 90];
16    devs = [0.1 0.1];
17    [hh,M]= high_pass_filter (fsamp, fcuts , devs);
18    dcValEn=1;
19 elseif (filter == "BPF") % band pass filter - order 22
20    fcuts = [60 100 160 200];
21    devs = [0.1 0.1 0.1];
22    [hh,M]= band_pass_filter (fsamp, fcuts , devs);
23    dcValEn=1;
24 end
25
26 % change coeffs to fixed point
27 xcoefs =round(hh*2^15);
28
29 % ----- SIMULATE INPUT
30 %%Time specifications :
31 dt = 1/fsamp; % seconds per sample
32 StopTime = 0.05; % seconds
33 t = (0:dt:StopTime-dt)'; % seconds
34
35 %%Sine wave:
36 x = round((sin(2*pi*Fc*t)* adcRes/2 + adcRes/2));
37
38 % ----- SIMULATE FILTER BEHAVIOR
39 x_ant(M+1) = 0;
40 out(size(t)) = 0;
41
42 for i = 1:size(t)
43    % rotate x_ant buffer
44    x_ant(2:M+1)=x_ant(1:M);
45    x_ant(1) = x(i);

```

```

46
47     out(i) = filtercalc (M, x_ant, xcoefs, dcValEn);
48 end
49
50 % ----- WRITE RESULTS
51 fprintf (" Filter : %s\n", filter );
52 fprintf (" Input freq : %d Hz\n", Fc);
53 fprintf (" Attenuation : %.2f dB\n", 10*log(max(out)/max(x)));
54
55 % Plot the signal versus time
56 if ( printFig )
57     fig = figure ('Name', filter + Fc + "Hz");
58     plot(t,x);
59     xlabel('t (in secs)');
60     hold on
61
62     plot(t,out);
63     legend('x = sin(100t)', 'y = filter (x)')
64     hold off
65
66     s1= sprintf ("./ figures /%s_%dinout.png", filter , Fc);
67     saveas( fig ,s1)
68 end
69
70 % Print results to file
71 if ( writeToFile )
72     % write input
73     sinput= sprintf ("./ input/%dinput.txt ", Fc);
74     fp = fopen( sinput , 'w' );
75     fprintf (fp , '%g\n', x);
76     fclose (fp );
77
78     % write filter coefs
79     scoefs= sprintf ("./ %s/%sCoefs.txt ", filter , filter );
80     fp = fopen( scoefs , 'w' );
81     fprintf (fp , "%s", regexprep (num2str( xcoefs ),'\s+', '\n') );
82     fclose (fp );
83
84     % write golden output
85     soutpt= sprintf ("./ %s/%s_%dout_golden.txt ", filter , filter , Fc);
86     fp = fopen( soutpt , 'w' );
87     fprintf (fp , "%s", regexprep (num2str(out),'\s+', '\n') );
88     fclose (fp );
89 end
90 end

```

Listing F.1: Função MATLAB para gerar os valores de saída desejados dos filtros.

```

1 function y = filtercalc (M, x_ant, x_coefs, dcValEn)
2     y = 0;
3
4     for i = 1:M+1
5         y = y + x_coefs (i) * x_ant(i);

```

```

6   end
7
8   % convert from fixed point and add dc value if needed
9   y = floor(y * 2^-15 + dcValEn*32767);
10
11  % truncate filter output
12  if (y < 0)
13      y = 0;
14  elseif (y > 65535)
15      y = 65535;
16  end
17 end

```

Listing F.2: Função MATLAB que implementa um filtro FIR.

```

1 printFig = 0;
2 writeToFile = 1;
3
4 fprintf ("*****LPF*****\n");
5 filters_tb (20, "LPF", printFig , writeToFile )
6 filters_tb (100, "LPF", printFig , writeToFile )
7
8 fprintf ("*****HPF*****\n");
9 filters_tb (20, "HPF", printFig , writeToFile )
10 filters_tb (100, "HPF", printFig , writeToFile )
11
12 fprintf ("*****BPF*****\n");
13 filters_tb (20, "BPF", printFig , writeToFile )
14 filters_tb (120, "BPF", printFig , writeToFile )
15 filters_tb (220, "BPF", printFig , writeToFile )

```

Listing F.3: Função MATLAB para gerar os valores de saída desejados dos filtros para diferentes frequências.

F.2 Validação dos filtros no Vivado HLS

```

1 // number of iterations
2 #define NUM_ITER 50
3 // filter order
4 #define _M_ 22
5
6 // output/input files path
7 #define INPUT_PATH " ../../../../../../golden_vectors "
8 #define OUTPUT_PATH " ../../../../../../sim"
9
10 int filter_tb (char * filter , uint32 x_coefs [_M_ + 1], int Fc, uint1 inputrand , uint1 dcValEn )
11 {
12     char input_file [128];
13     char outgold_file [128];
14     char output_file [128];
15
16     printf ("%s Test @%dHz\t-", filter , Fc);

```

```

17 // input files
18 snprintf ( input_file , sizeof( input_file ), "%s/ input /%dinput . txt ", INPUT_PATH, Fc);
19 snprintf ( outgold_file , sizeof( outgold_file ), "%s/%s/%s_%dout_golden . txt ", INPUT_PATH, filter , filter , Fc);
20 // output file
21 snprintf ( output_file , sizeof( output_file ), "%s/%s/%s_%dsim_output . txt ", OUTPUT_PATH, filter , filter , Fc);
22
23 // ----- LOAD FILTER INPUT VALUES
24 // input values
25 uint16 x[NUM_ITER];
26 // last M+1 input values
27 uint16 x_ant [_M_ + 1] = {0};
28
29 FILE *fp ;
30 int i ;
31
32 fp = fopen ( input_file , "r");
33 if (! fp )
34 {
35     printf ("\nERROR: simulation input file '%s' not found !\n", input_file );
36     return 0;
37 }
38
39 // load input values from file
40 for (i=0; i<NUM_ITER; i++)
41     fscanf (fp , "%d", &x[i] );
42
43 fclose (fp );
44
45 // ----- SIMULATE FILTER BEHAVIOR
46 // open output file
47 fp = fopen ( output_file , "w");
48 if (! fp )
49 {
50     printf ("\nERROR: simulation output file '%s' not created !\n", output_file );
51     return 0;
52 }
53
54 // Capture the output results of the filter to a file
55 int j ;
56 uint16 y = 0;
57
58 for (i=0; i<NUM_ITER; i++){
59     // update x ant values
60     for (j=22; j>0; j--)
61         x_ant [j] = x_ant [j -1];
62
63     x_ant [0] = x[ i ];
64
65     // apply filter
66     y = fir_filter ( x_ant , x_coefs , dcValEn );
67
68     fprintf (fp , "%d\n", y);
69 }

```

```

70
71     fclose ( fp );
72
73 // ----- COMPARE RESULTS WITH EXPECTED VALUES
74 char cmd[128];
75 sprintf (cmd, sizeof(cmd)," diff --brief -w %s %s", output_file , outgold_file );
76
77 int ret = system(cmd);
78 if (ret != 0) {
79     printf ("FAILED <<\n");
80 } else {
81     printf ("PASSED\n");
82 }
83
84 return ret;
85 }
```

Listing F.4: Testbench dos filtros em C, no Vivado HLS.

```

1 #define _M_ 22
2
3 int main (void)
4 {
5     // declare filters coefs
6     uint32 lpf_coefs [_M_ + 1];
7     uint32 hpf_coefs [_M_ + 1];
8     uint32 bpf_coefs [_M_ + 1];
9
10    FILE * fp_lpf , * fp_hpf , * fp_bpf ;
11
12    printf ("***** C Simulation *****\n");
13 // ----- LOAD FILTER COEFFICIENTS
14    fp_lpf = fopen (" ../../golden_vectors /LPF/LPFCoefs.txt ", "r");
15    fp_hpf = fopen (" ../../golden_vectors /HPF/HPFCoefs.txt ", "r");
16    fp_bpf = fopen (" ../../golden_vectors /BPF/BPFCoefs.txt ", "r");
17
18    if (! fp_lpf || ! fp_hpf || ! fp_bpf )
19    {
20        printf ("ERROR: Cannot open coefs file !\n");
21        return 1;
22    }
23
24 // load input values from file
25 for (int i=0; i<_M_+1; i++)
26 {
27     fscanf ( fp_lpf , "%d", & lpf_coefs [ i ] );
28     fscanf ( fp_hpf , "%d", & hpf_coefs [ i ] );
29     fscanf ( fp_bpf , "%d", & bpf_coefs [ i ] );
30 }
31
32 fclose ( fp_lpf );
33 fclose ( fp_hpf );
34 fclose ( fp_bpf );
```

```
35 // ----- TEST FILTERS
36 filter_tb ("LPF", lpf_coefs , 20, 0);
37 filter_tb ("LPF", lpf_coefs , 100, 0);
38
39 filter_tb ("HPF", hpf_coefs , 20, 1);
40 filter_tb ("HPF", hpf_coefs , 100, 1);
41
42 filter_tb ("BPF", bpf_coefs , 20, 1);
43 filter_tb ("BPF", bpf_coefs , 120, 1);
44 filter_tb ("BPF", bpf_coefs , 220, 1);
45
46 printf ("***** End C Simulation *****\n");
47
48 return 0;
49 }
```

Listing F.5: Testbench dos filtros para diferentes frequências, no Vivado HLS

Resultados

G.1 Resultados Esperados

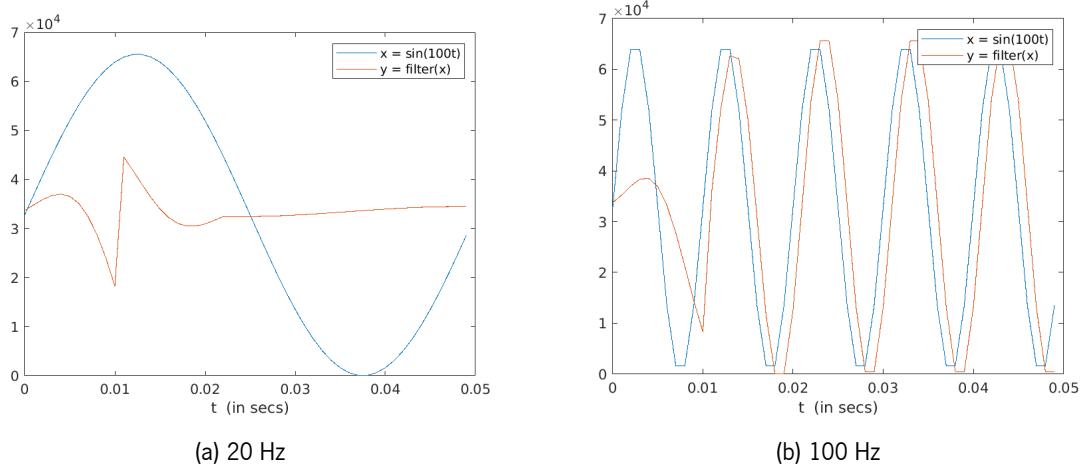


Figura G.1: Saída do HPF em Matlab, a vermelho, em função da entrada, a azul.

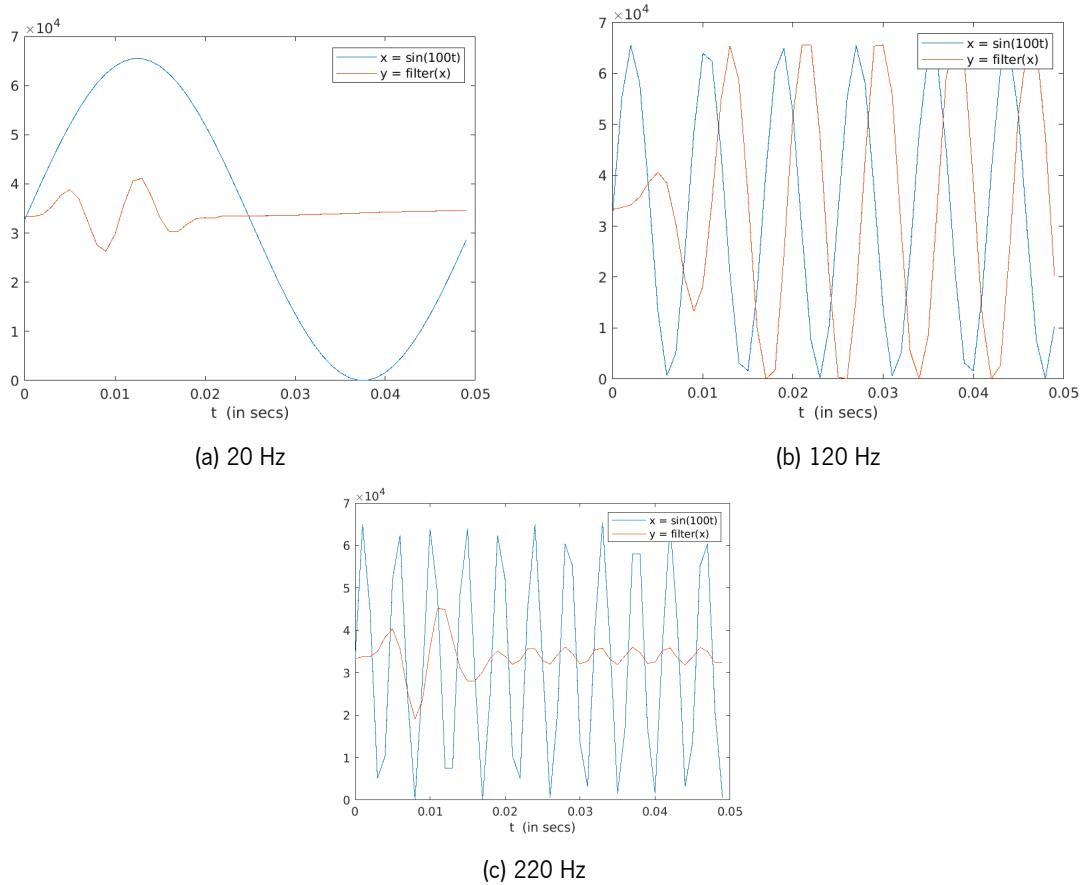


Figura G.2: Saída do BPF em Matlab, a vermelho, em função da entrada, a azul.

G.2 Resultados na STM32

Nas figuras G.3 e G.4 pode ver-se o efeito da frequência de amostragem ser 1 kHz na obtenção do sinal de saída. Para uma onda de 100 Hz, serão amostrados 10 valores por período, daí a forma de onda apresentada nas figuras.

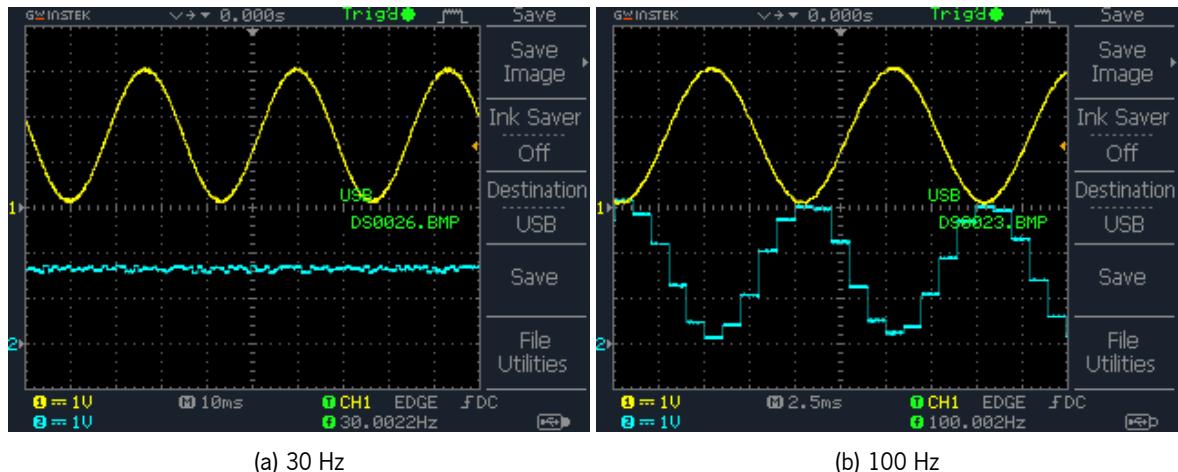


Figura G.3: Saída do HPF na STM32, a azul, em função da entrada, a amarelo.

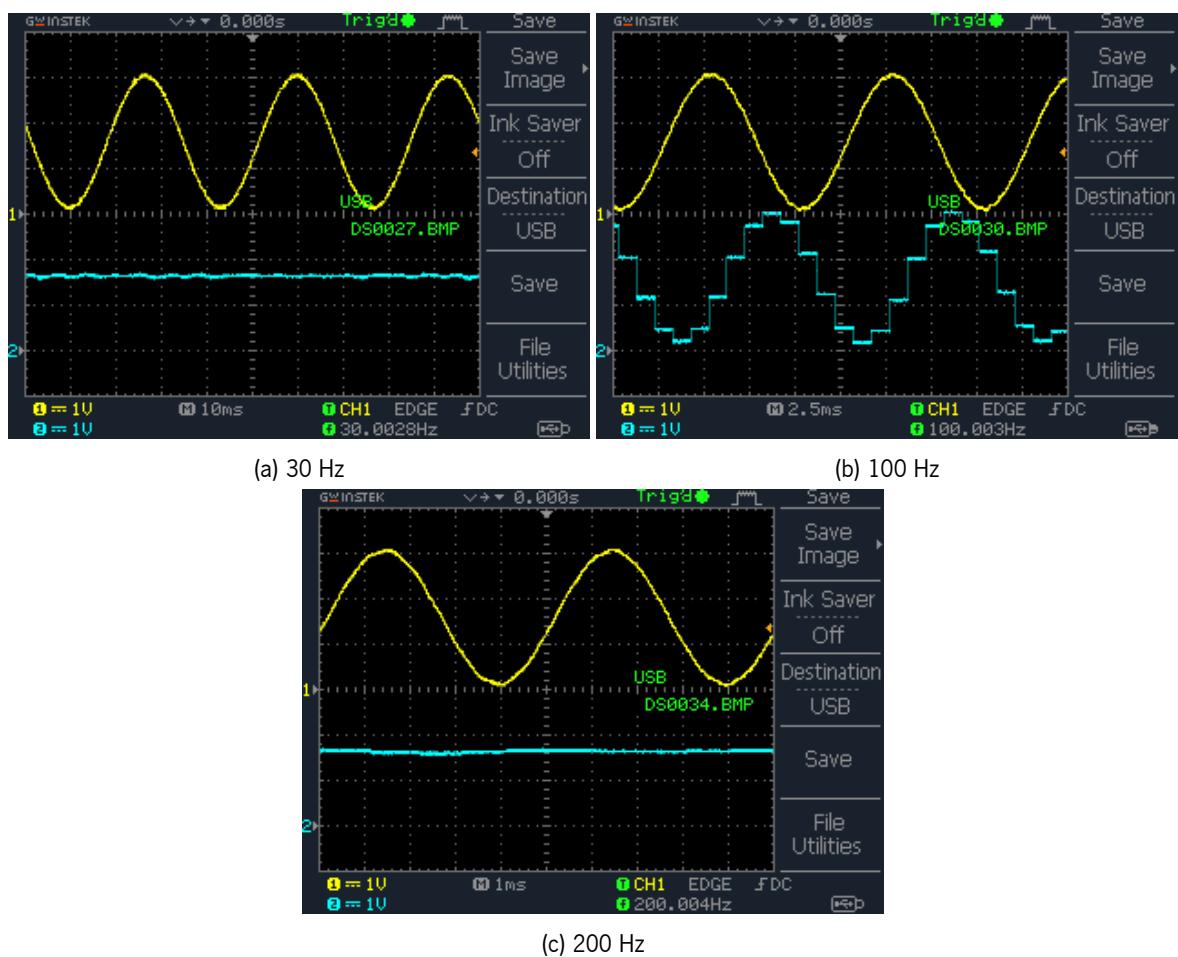


Figura G.4: Saída do BPF na STM32, a azul, em função da entrada, a amarelo.

G.3 Resultados na Zybo

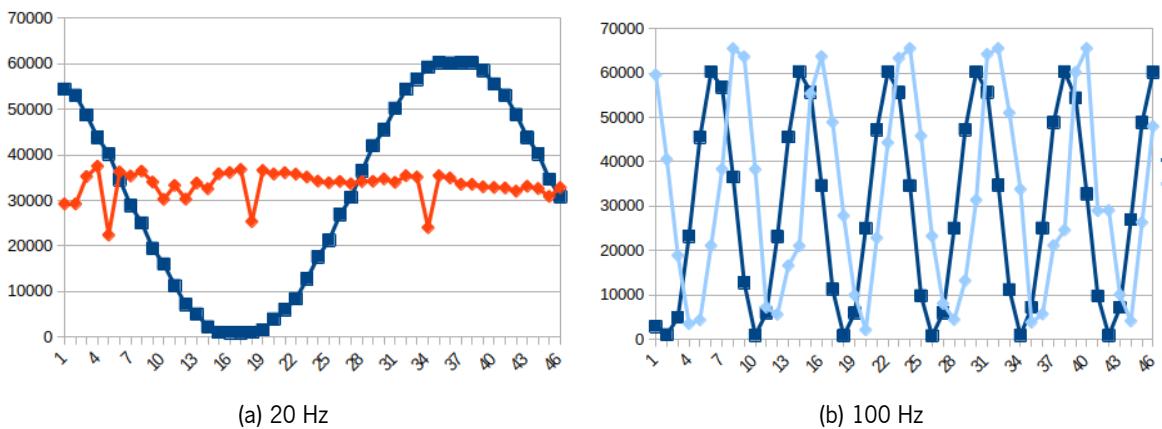


Figura G.5: Bloco de filtros com o HPF selecionado para uma entrada, a azul escuro, de: a) 20 Hz - Saída a azul claro; b) 100 Hz - Saída a vermelho.

Na figura G.6 pode ver-se que os resultados obtidos na Zybo para o BPF não corresponde ao esperado, visto que o filtro corta todas as frequências de 20 Hz a 220 Hz.

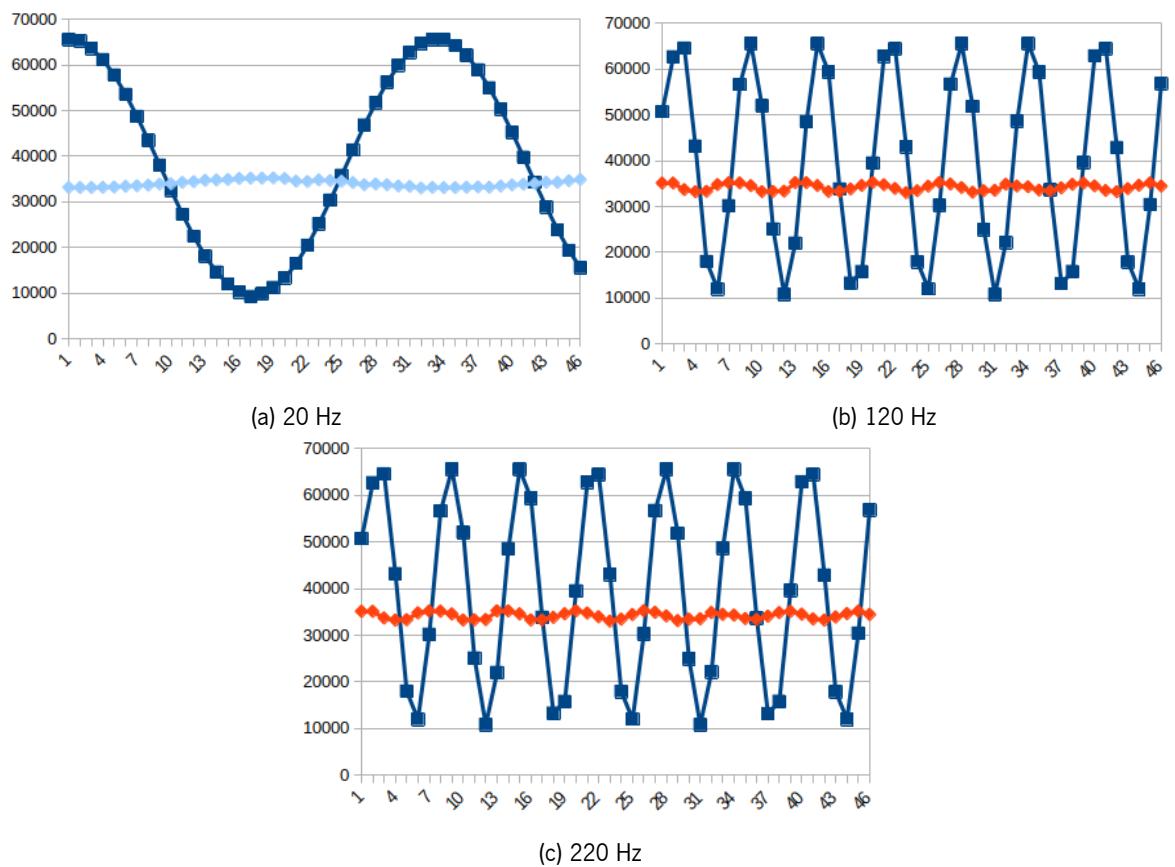


Figura G.6: Bloco de filtros com o BPF selecionado para uma entrada, a azul escuro, de: a) 20 Hz - Saída a azul claro; b) 120 e 220 Hz - Saída a vermelho.

G.4 Resultados Finais na Zybo

Na figura G.7 vê-se que a visualização da saída do HPF não é muito bem definida, principalmente na frequência de banda passante, 100 Hz, onde o sinal de saída deveria ser igual ao sinal de entrada (5.31).

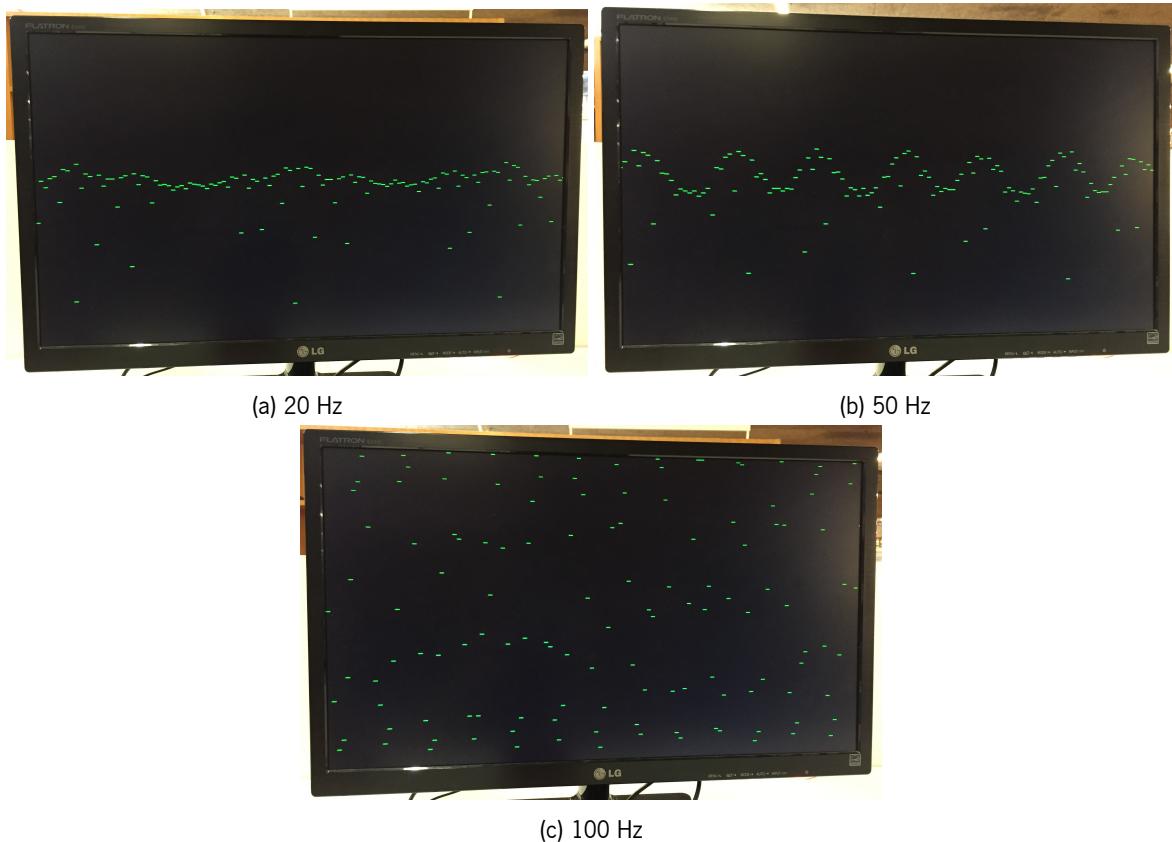


Figura G.7: Visualização da saída do HPF para vários sinais de entrada com frequências diferentes.

Na figura G.8 verifica-se o problema identificado na figura G.6, pois a saída mantém-se a zero para qualquer frequência do sinal de entrada.

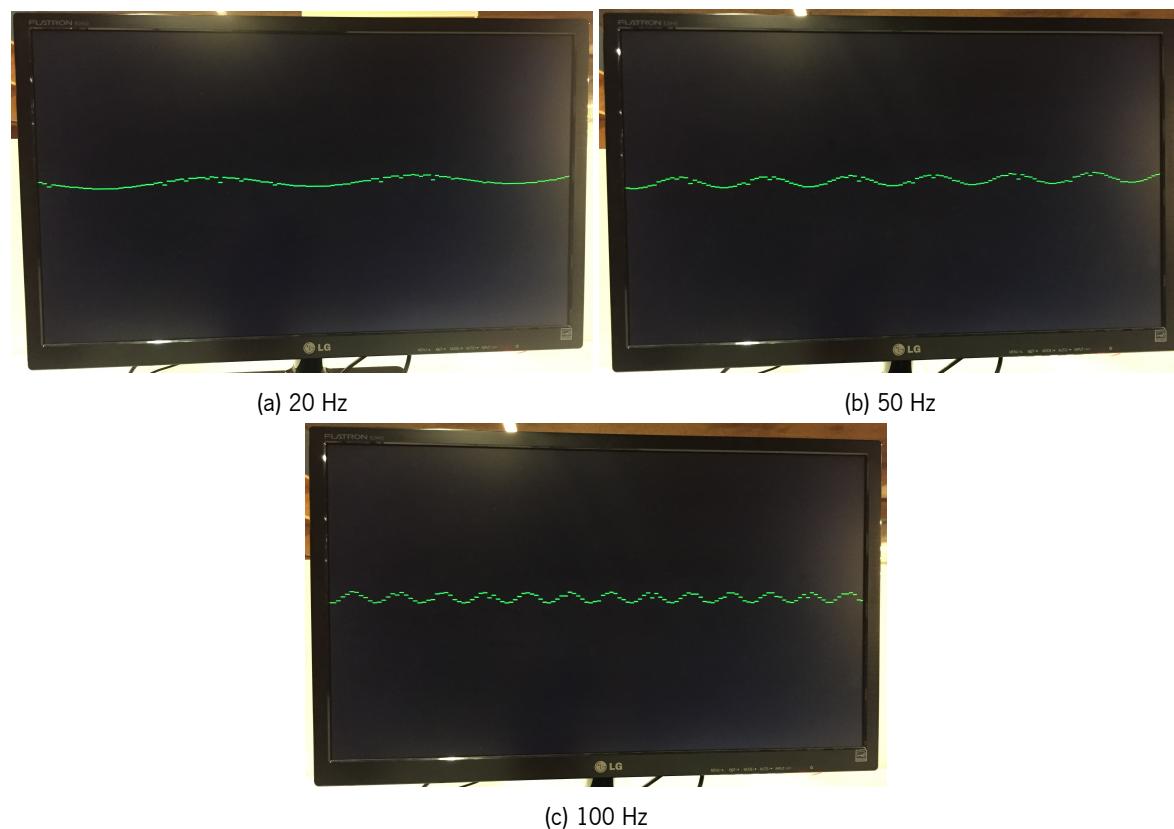


Figura G.8: Visualização da saída do BPF para vários sinais de entrada com frequências diferentes.

Block Design

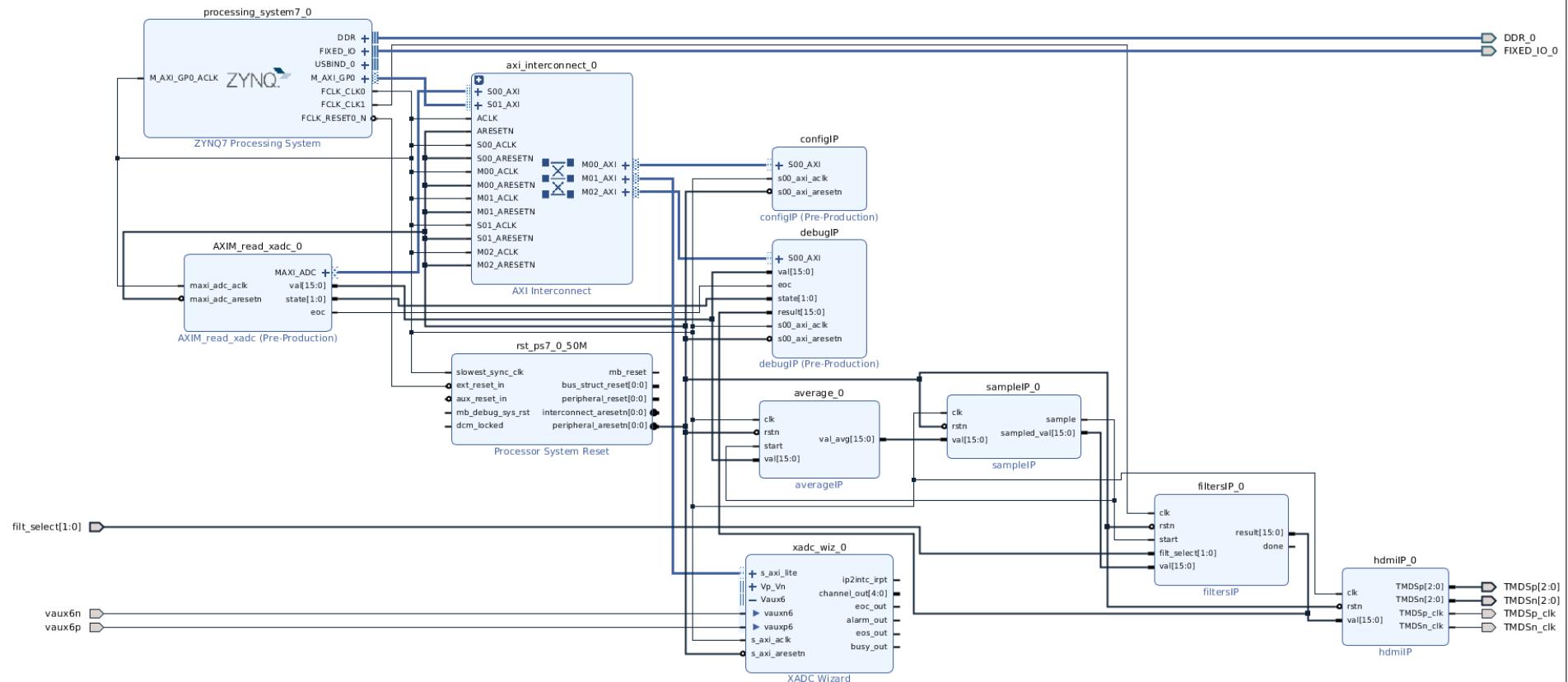


Diagrama de Gantt

