Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Estudo de um Modelo de Conversor A/D com Níveis de Quantização Configuráveis

Mateus A. Botelho Ribeiro

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Daniel Chaves Café

Brasília
2017

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Estudo de um Modelo de Conversor A/D com Níveis de Quantização Configuráveis

Mateus A. Botelho Ribeiro

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Daniel Chaves Café (Orientador)
ENE/UnB

Prof. Dr. José Camargo Costa
UnB

Prof. Dr. Stefan Michael Blawid
UnB

Prof. Dr. Ricardo Pezzuol Jacobi
Coordenador do Curso de Engenharia da Computação

Brasília, 5 de julho de 2017

Dedicatória

Este trabalho é dedicado a

minha mãe Aparecida, meu pai Genivaldo e meu irmão Eduardo por todo amor e carinho que somente a família pode dar;

Arthur Teixeira, Deborah Torres, Denise Amorim, Edson Júnior, Felipe Milhomem, Gabriel Machado, Igor Fernandes, João Pedro Tonietti, João Vitor, Júlia de Lucena, Letícia Fontes, Luan Talles, Luis Felipe, Marcella Menezes, Marianna Camargo, Matheus Antunes, Matheus Rosendo, Patricia Monteiro, Pedro Ivo, Rafael Bispo, Rafael Guimarães, Rafael Lages, Rebeca Moura, Roberto Nishino, Tiago Pereira Vidigal e Vinícius Faria pela alegria de ter a amizade de vocês por todos esses anos;

Em especial, a Florenza Annechino Noguchi pelo apoio, carinho, alegria, amor e todos os bons momentos que você trouxe consigo ao entrar em minha vida.

Também dedico este trabalho a todos os professores, familiares e amigos que me apoiaram ao longo dos anos nesta minha jornada acadêmica.

Agradecimentos

Os meus mais sinceros agradecimentos aos professores José Camargo Costa e Stefan Michael Blawid pelo auxílio na produção deste texto.
Em especial, ao meu orientador, professor Daniel Chaves Café pela disponibilidade, esforço e dedicação em me ajudar a realizar esse trabalho.

Resumo

Neste trabalho apresentamos um estudo sobre conversores analógico-digital (A/D), com foco na comparação entre os erros das quantizações uniforme e não-uniforme. Para a quantização não-uniforme, o algoritmo de quantização se baseou na lei-μ (μ-*law*) da recomendação G.711 da ITU-T e teve seu comportamento analisado para diferentes valores de μ. Esse estudo foi realizado para demonstrar que, para um determinado sinal de entrada ("vin"), existe um intervalo de amplitudes onde a quantização não-uniforme possui um erro menor do que a quantização uniforme. Um código Verilog-AMS de um conversor A/D foi testado no Qucs e os códigos das quantizações uniforme e não-uniforme foram testados no Octave.

Palavras-chave: Conversor A/D, Verilog-AMS, Qucs, Octave, quantização não-uniforme

Abstract

In this work, we present a study on analog-digital (A/D) converters, focusing on the comparison between the errors of the uniform and nonuniform quatizations. For nonuniform quantization, the quantization algorithm was based on the μ-law of ITU-T Recommendation G.711 and had its behavior analysed for different values of μ. This study was performed to demonstrate that, for a given input signal ("vin"), there is a range of amplitudes where nonuniform quantization has a smaller error than the uniform quantization. A Verilog-AMS code from an A/D converter was tested on Qucs and the uniform and nonuniform quantization codes were tested on Octave.

Keywords: A/D Converter, Verilog-AMS, Qucs, Octave, nonuniform quantization

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

A/D Analógico-Digital.

AC Alternating Current.

ADC Analog-to-Digital Converter.

ADMS Automatic Device Model Synthesizer.

AMS Analog and Mixed Signal.

API Application Programming Interface.

DC Direct Current.

GPIO General Purpose Input Output.

GPL General Public License.

HDL Hardware Description Language.

IEEE Institute of Electrical and Electronics Engineers.

MSP430 Mixed-Signal Microcontroller.

OVI Open Verilog International.

PAM Pulse-Amplitude Modulation.

PCM Pulse-Code Modulation.

PO Ponto Ótimo.

PPM Pulse-Position Modulation.

PWM Pulse-Width Modulation.

Qucs Quite Universal Circuit Simulator.

RTL Register-Transfer Level.

SNR Signal-to-noise ratio.

SPICE Simulation Program with Integrated Circuit Emphasis.

Capítulo 1

Introdução

Este documento apresenta a metodologia, a implementação e os resultados do trabalho de graduação em engenharia de computação voltado para o Estudo de um Modelo de Conversor A/D com Níveis de Quantização Configuráveis.

1.1 Objetivos do trabalho

O objetivo deste trabalho é desenvolver um modelo de conversor A/D em Verilog-AMS e integrá-lo a plataforma openMSP430.

O modelo inicial do conversor utilizará uma quantização uniforme para criar níveis de quantização igualmente espaçados. Posteriormente, uma segunda versão do conversor A/D será desenvolvida implementando a quantização não-uniforme (baseada na lei-μ, apresentada na Equação 2.3) para criar níveis de quantização com espaçamento variáveis.

O openMSP430 é um núcleo sintetizável de microcontrolador de 16 *bits* escrito em Verilog compatível com a família de microcontroladores MSP430 da Texas Instruments[1]. Essa plataforma virtual permite a execução de códigos gerados por qualquer uma das ferramentas do MSP430 de maneira precisa.

Esse núcleo já possui alguns periféricos implementados, como por exemplo Watchdog, General Purpose Input Output (GPIO) e TimerA, mas ainda não possui um módulo para o conversor A/D.

O MSP430 (Mixed-Signal Processor) é um microcontrolador voltado para aplicações de baixo consumo de energia que podem trabalhar tanto com sinais digitais, quanto com sinais analógicos. Portanto, ter um módulo que implemente um conversor A/D na plataforma openMSP430 é considerado como sendo algo de alta importância. Este trabalho então visa complementar o modelo já existente, agregando a capacidade de se trabalhar com sinais analógicos e criar um sistema de interface analógico-digital.

[1] `https://opencores.org/project,openmsp430`

A utilidade de uma plataforma virtual como o openMSP430 vem da possibilidade de que eventuais utilizadores do microcontrolador MSP430 possam testar como os seus respectivos códigos vão interagir com o microcontrolador sem a necessidade de possuir o próprio *hardware*. O uso da plataforma virtual pode trazer benefícios para os desenvolvedores, como por exemplo:

- Permitir que os desenvolvedores avaliem a interação de seus *softwares* com o MSP430 sem a necessidade de usar um *hardware* para testes;
- Para ambientes de uso mais crítico, ela permite que simulações sejam realizadas em um ambiente virtual, sem afetar o ambiente real de uso do microcontrolador;
- Reduz possíveis custos para os desenvolvedores, como tempo e recursos financeiros.

1.2 Proposta do trabalho

A proposta deste trabalho é desenvolver o modelo de um conversor A/D em Verilog-AMS, que possa eventualmente servir de estudo para a comunidade científica (openMSP430, comunidade *open source*, etc).

Capítulo 2

Metodologia

Este capítulo descreve as ferramentas e linguagens utilizadas para a elaboração do trabalho, assim como uma breve fundamentação teórica sobre alguns pontos.

2.1 A linguagem Verilog

2.1.1 Verilog

Verilog Hardware Description Language (Verilog HDL), ou simplesmente Verilog, é uma linguagem de descrição de *hardware* digital usada para modelar sistemas eletrônicos, cujo uso mais comum é na parte de *design* e verificação de circuitos digitais a nível de abstração de transferência de registros (RTL). Verilog é a junção das palavras *"verification"* e *"logic"* [1].

A seguir, são apresentadas algumas das diferenças entre uma linguagem de descrição de *hardware* e uma linguagem de programação de *software*:

- São usadas para descrever a estrutura e o comportamento de circuitos eletrônicos;
- Capacidade de incluir maneiras de descrever o decorrer do tempo e a intensidade dos sinais;
- São linguagens naturalmente voltadas para o paralelismo (suportam múltiplas *threads*).

Alguns anos após a sua criação, a linguagem Verilog, que originalmente era propriedade da Cadence Design Systems, foi transferida para domínio público sob a organização hoje conhecida como Accellera. Posteriormente, Verilog foi submetido ao IEEE e tornou-se o padrão IEEE 1364-1995, popularmente conhecido como Verilog-95.

Devido a sua popularidade na comunidade de *design*, logo surgiram propostas de extensão para que a linguagem pudesse suportar simulações analógicas e de sinais mis-

tos, levando a criação do Verilog-A (*Analog*) e Verilog-AMS (*Analog and Mixed Signal*). Verilog-A nunca fora planejada para ser uma linguagem autônoma, sendo um subconjunto do Verilog-AMS, que por sua vez engloba tanto Verilog HDL, quanto Verilog-A, além de incluir construtores para sinais mistos. Modelos em Verilog HDL e Verilog-A podem ser usados em simuladores de Verilog-AMS, mas o contrário não é verdadeiro [1].

2.1.2 Verilog-A

Verilog-A Hardware Description Language, ou simplesmente Verilog-A, é uma linguagem para modelagem de circuitos analógicos, sendo o subconjunto contínuo no tempo do Verilog-AMS. O Verilog-A não foi projetado para trabalhar diretamente com o Verilog HDL, mas possui uma sintaxe semelhante e semântica relacionada que foi planejada para modelar sistemas analógicos e ser compatível com mecanismos de simulação de circuitos da classe SPICE[1]. Verilog-A foi padronizado pela Open Verilog International (OVI) em 1998.

2.1.3 Verilog-AMS

Verilog-AMS Hardware Description Language, ou simplesmente Verilog-AMS, foi inventado pela OVI em 1998. Em 2000 a Accelera adquiriu a Open Verilog International e todos os trabalhos seguintes foram conduzidos por ela.

O objetivo do Verilog-AMS é permitir que desenvolvedores de sistemas analógicos e circuitos integrados usem módulos que encapsulam descrições comportamentais de alto-nível, assim como descrições estruturais de sistemas e componentes. Verilog-AMS pode ser usado para descrever sistemas de sinais mistos (analógico/digital) usando descrições discretas e contínuas [2].

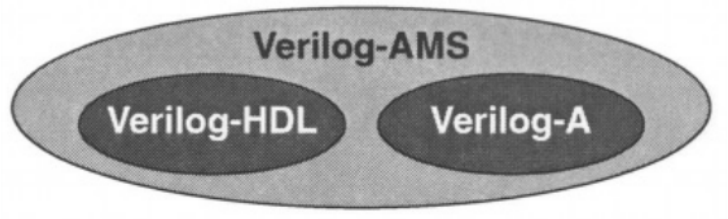

Figura 2.1: A linguagem Verilog. (Fonte: [1]).

[1] `http://semiengineering.com/kc/entity.php?eid=22027`

2.2 ADMS

O ADMS (Automatic Device Model Synthesizer) é um *software* de domínio público gerador de código. Sua função é converter modelos elétricos de dispositivos compactos especificados em uma linguagem de descrição de alto nível em código C para a API dos simuladores SPICE. Com base nas transformações especificadas na linguagem XML, o ADMS transforma o código Verilog-AMS para outra linguagem destino[2].

2.3 Quite Universal Circuit Simulator (Qucs)

Qucs é um simulador de circuito integrado que permite aos usuários utilizarem sua interface gráfica para criar circuitos e simular o seu comportamento. O simulador visa ter suporte para os mais diversos tipos de simulação, por exemplo, DC, AC, S-parameter, análise de ruído, etc. Simulações digitais também são suportadas pelo Qucs usando VHDL ou Verilog.

Qucs é um *software* livre licenciado pela GNU General Public License (GPL). Apesar do Qucs já estar bem avançado, ele ainda encontra-se em desenvolvimento[3].

2.3.1 Tutorial de como criar componentes usando Verilog-A

O Qucs possibilita que o usuário crie seus próprios componentes utilizando códigos em Verilog-A. O Qucs permite até que mais de um código Verilog-A esteja presente no mesmo projeto. O componente a ser criado neste tutorial será um resistor.

Criando um projeto

Primeiramente, criamos um novo projeto clicando na aba "Projetcs" ⇒ "New". Em seguida uma caixa de texto será aberta e inserimos o nome do nosso projeto (ex: Tuto). Após clicar em "Create", a pasta com o projeto "Tuto_prj" será criada e a aba "Content" será aberta automaticamente.
Obs: Não usar espaços no nome do projeto.

Essa aba indica tudo que está contido dentro do projeto ''Tuto_prj", como códigos Verilog, VHDL, esquemáticos de circuitos, etc. Em seguida, clicamos no menu ''File" ⇒ ''New Text", ou no ícone correspondente na barra de menus, ou pressionamos o atalho ''Ctrl + Shift + V". Um editor de texto será aberto em uma aba (do lado direito) chamada ''untitled", e será nessa aba que iremos inserir o seguinte código Verilog-A para o resistor:

[2] `https://github.com/Qucs/ADMS`
[3] `http://qucs.sourceforge.net/index.html`

```
// Linear resistor
`include "disciplines.vams"

module resistorVams (p, n);
   parameter real r=1; // resistance (Ohms)
   inout p, n;
   electrical p, n;

   analog
      I(p,n) <+ V(p,n)/r;
endmodule
```

Observações importantes

Obs1: A equação popular que modela o comportamento de um resistor é $V = R * I$, mas devido a limitação do ADMS de não permitir potenciais do lado esquerdo de atribuições de contribuição, temos de reescrever essa fórmula como I = V / R.
Obs2: Não usar "_" em nomes de arquivos/módulos Verilog-A. Esse *bug* aparece ocasionalmente. Ex: se a minha declaração do módulo for ''module resistor_Vams (p, n);", o Qucs pode apresentar algum erro ao tentar criar esse componente.
Obs3: O nome do componente a ser criado não pode possuir o mesmo nome de um componente já existente. Ex: se a minha declaração do módulo fosse ''module resistor (p, n);", o Qucs daria erro ao criar esse componente pois já existe um componente chamado "resistor".
Obs4: Cuidado ao copiar e colar códigos de fontes que não sejam um editor de texto (ex: pdfs, *browser*, etc) diretamente para o editor de texto do Qucs, pois pode ocorrer um problema de conversão de caracteres. Isso ocorre principalmente com caracteres especiais, como aspas, @, %, *, - e outros.
Obs5: O nome do arquivo ".va" tem que possuir o mesmo nome do módulo no código Verilog-A.

Após inserir o código, salve o arquivo clicando em ''File" ⇒ ''Save", ou pressionando o atalho ''Ctrl + S". Uma caixa de diálogo se abrirá e iremos digitar o nome do nosso arquivo com o código Verilog-A. O nome do arquivo tem que possuir o mesmo nome do módulo no código Verilog-A, no caso, escrevemos ''resistorVams.va". Mesmo já escrevendo a extensão ''.va" no nome do arquivo, ainda temos que definí-lo como um código Verilog-A. Para isso, clique em ''VHDL Sources" no canto inferior direito e selecione ''Verilog-A Sources" na lista (Figura 2.2). Finalize clicando em "Save". Se tudo der certo, o Qucs irá

reconhecer o código como sendo Verilog-A e irá destacar o seu código, deixando palavras reservadas na cor azul. Sempre que precisar abrir esse código, ele estará disponível no menu do lado esquerdo na aba ‘’Contens” na lista de ‘’Verilog-A” (Figura 2.3).

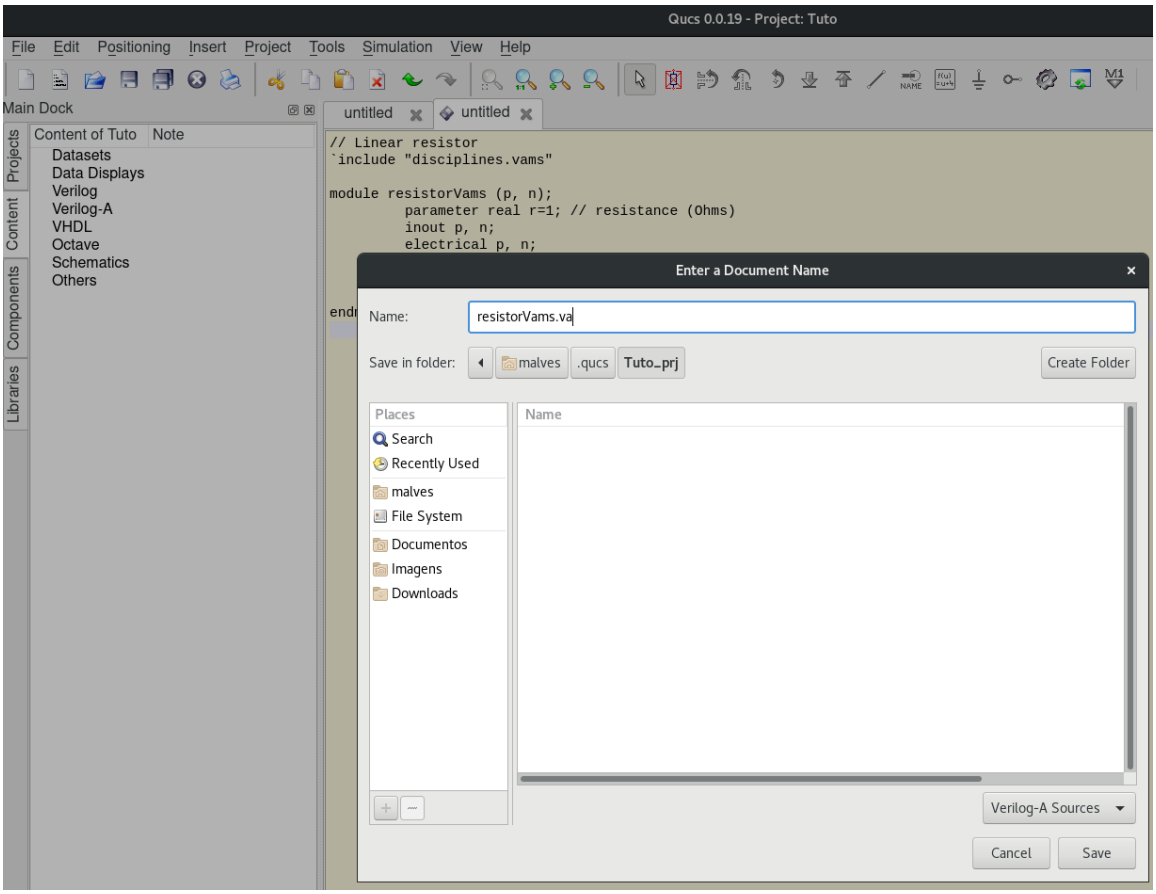

Figura 2.2: Salvando o código Verilog-A.

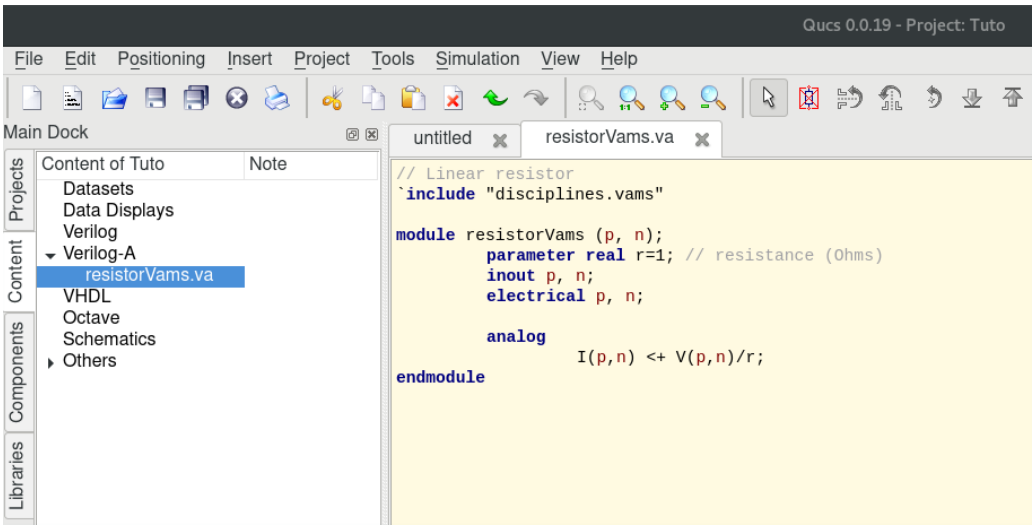

Figura 2.3: Código Verilog-A salvo no projeto ”Tuto”.

Compilando o código

Para compilar o código Verilog-AMS, com a aba ''resistorVams.va" aberta, clique no menu ''Project" ⇒ ''Build Verilog-A Module". Se no console não aparecer nenhum erro fatal, por exemplo um erro de sintaxe, e nele estiver escrito que diversos arquivos ''.cpp" e ''.h" foram criados, então o código foi compilado com sucesso (Figura 2.4).

Obs: no console podem aparecer diversos *warnings* ou *errors* da ferramenta, mas geralmente eles podem ser ignorados. O importante é não ocorrer um erro fatal.

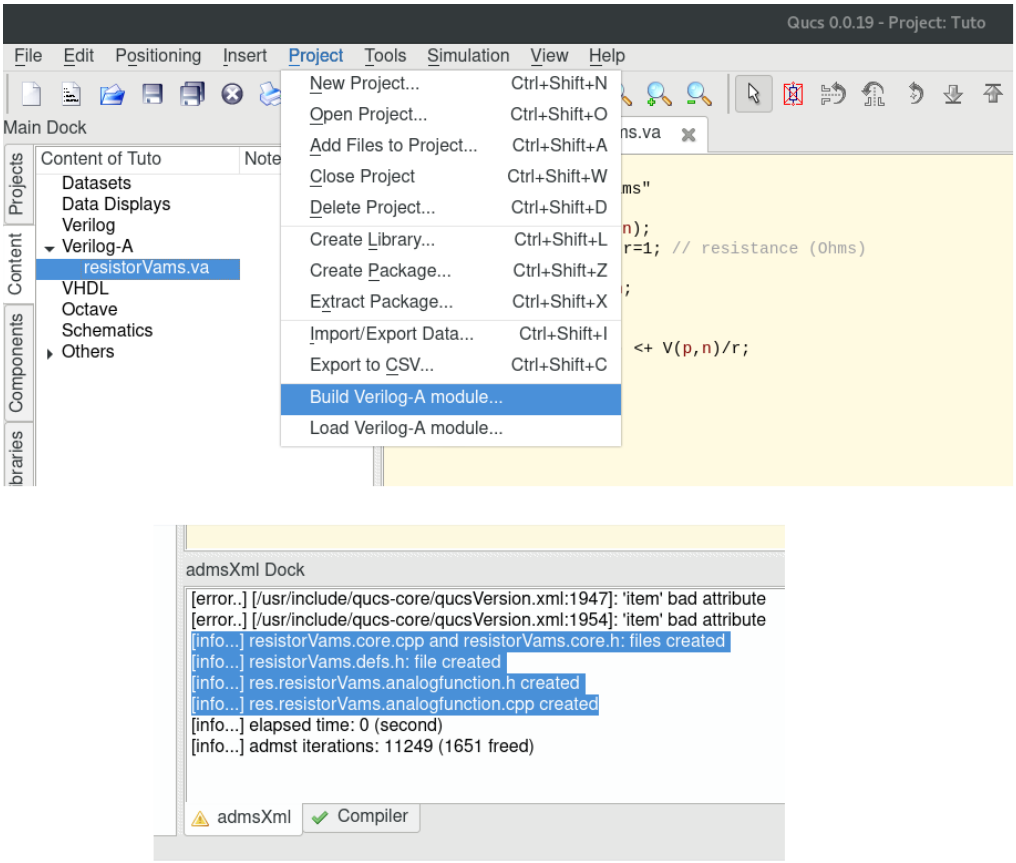

Figura 2.4: Compilando um código Verilog-A no Qucs.

Esquemático do componente

Com a aba ''resistorVams.va" aberta, clique em ''File" ⇒ ''Edit Text Symbol", ou pressione o atalho F9 (Figura 2.5). Uma nova aba se abrirá chamada ''resistorVams.sym". Nessa aba será apresentada a representação de como o seu componente será desenhado ao ser inserido em um esquemático para criar circuitos. Esse desenho inicial se baseia no código presente em ''resistorVams.va". Caso o nome das portas ou o nome do componente não tenham carregado automaticamente, apenas alterne entre as abas ''resistorVams.va" e ''resistorVams.sym" que o desenho irá ser atualizado automaticamente, ou aperte F9 novamente.

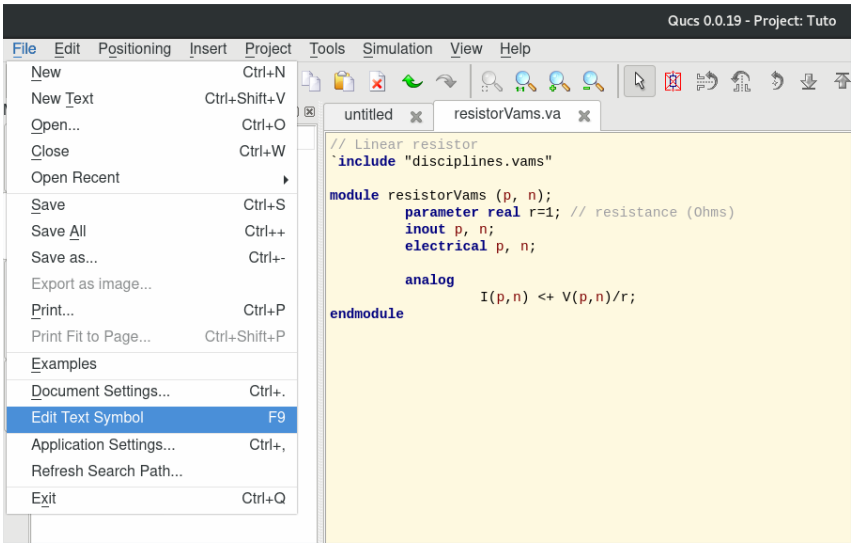

Figura 2.5: Editar o esquemático do componente.

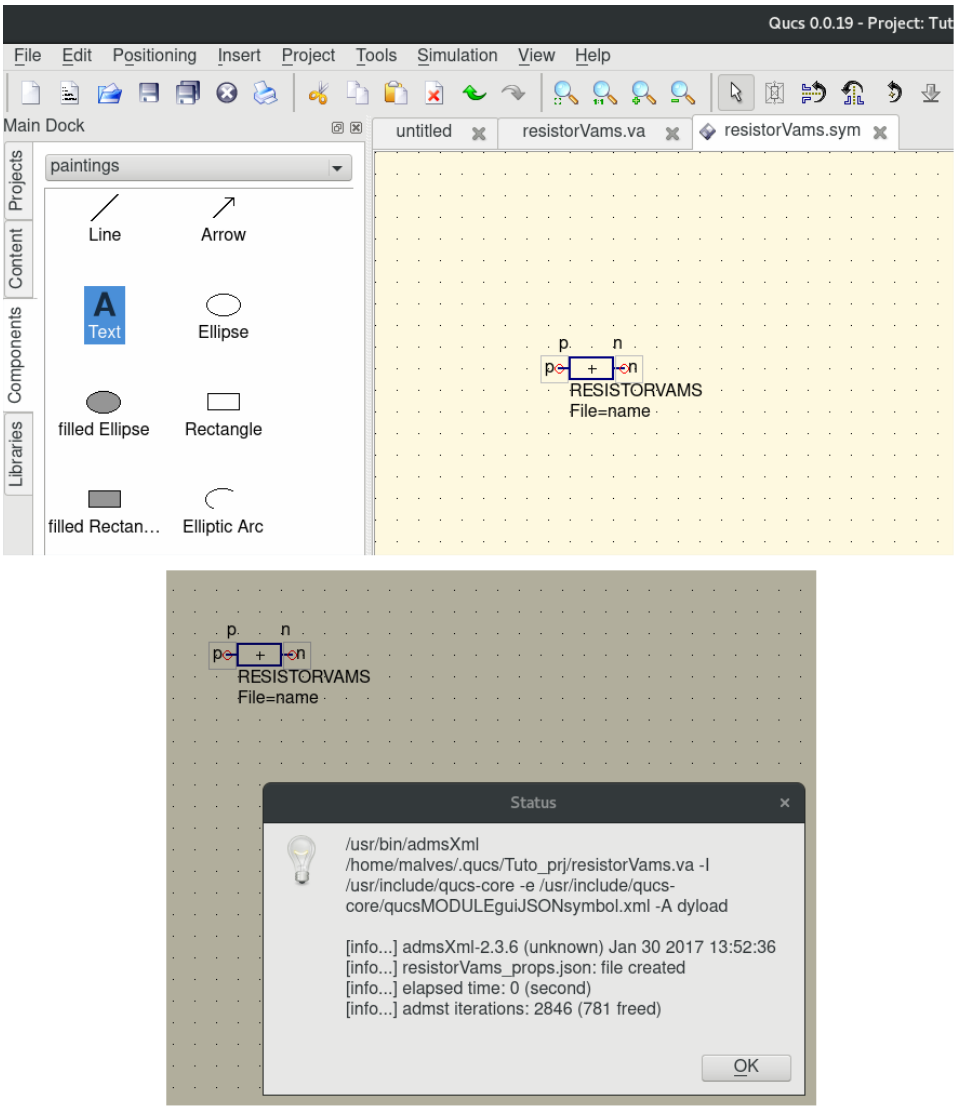

Figura 2.6: Criando o esquemático do componente.

O nome do componente é o nome do módulo presente no arquivo ''resistorVams.va", no caso, ''RESISTORVAMS". Esse módulo possui duas portas, p e n. Na aba ''resistorVams.sym", você pode redesenhar a aparência do seu componente da forma que quiser, podendo mudar a posição das portas, o formato do componente, inserir texto, etc. Recomendo utilizar a caixa de texto (Text) para inserir o nome de cada porta e identificá-las, visto que, facilita a identificação das portas na hora de inserir o componente em um esquemático.

Pressione ''Ctrl + S" para salvar o esquemático. Uma caixa de diálogo se abrirá, exibindo quatro linhas de informação. Se em nenhuma delas aparecer uma mensagem de erro, o esquemático do componente foi criado com sucesso. Pressione OK para finalizar (Figura 2.6).

Carregando o componente no Qucs

Para que o nosso componente seja exibido na aba ''Content" como se fosse um componente nativo do Qucs, é necessário carregá-lo para a interface gráfica do Qucs. Para isso, precisa-se de uma imagem para representar o componente (opcional) e carregar o esquemático do componente (obrigatório).

Para o arquivo de imagem, existe a restrição do Qucs de que ela deve ser um arquivo com as dimensões 32x32 *pixels* [3]. A imagem a ser utilizada (ex: image.png) deve ser copiada para a pasta do projeto, dentro do diretório do Qucs (o diretório do Qucs depende de onde o usuário instalou o programa. No meu caso, o caminho do diretório é ''$/home/user/.qucs/Tuto\_prj/$").
Obs: pode ser que a pasta do projeto esteja como Oculta, portanto, vá na pasta onde o Qucs foi instalado e marque a opção ''Exibir arquivos ocultos".

Para carregar o componente na interface gráfica do Qucs, clique em ''Project" ⇒ ''Load Verilog-A module". Uma mensagem de ícone não encontrado irá ser exibida. Clique em OK. Uma nova caixa de diálogo se abrirá exibindo os arquivos de símbolo Verilog-A. Nessa caixa de diálogo, teremos a imagem de uma página branca com um 'X' vermelho, indicando que o componente ainda não possui uma imagem para representá-lo (Figura 2.7). Clique em ''Change Icon" e escolha o arquivo de imagem (no caso, image.png) que fora copiado para a pasta do projeto. A imagem da página branca com um 'X' vermelho será substituída pela imagem escolhida. Clique em OK para finalizar.

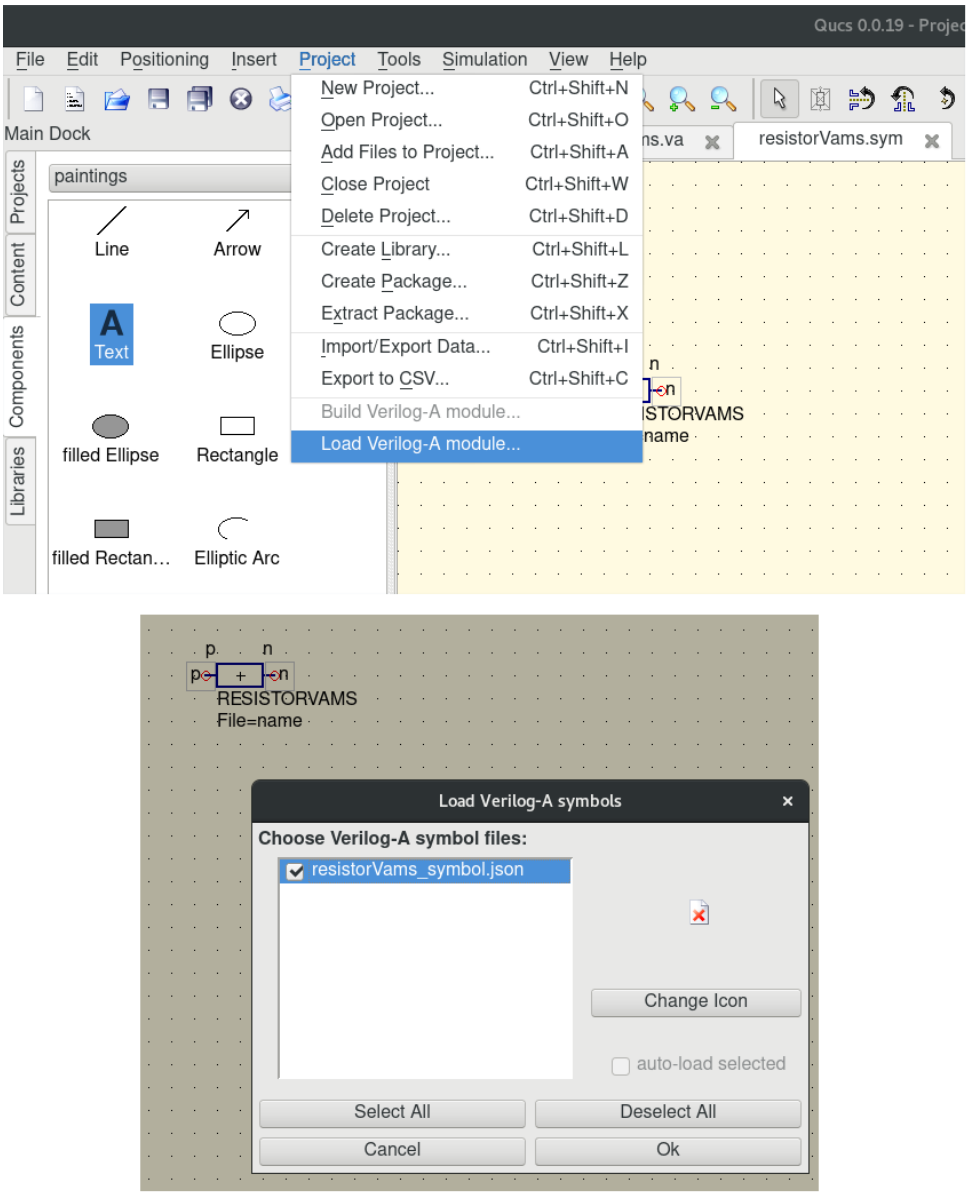

Figura 2.7: Carregando um componente no Qucs.

A aba ‘’Components” irá ser aberta automaticamente, exibindo o novo componente criado na lista ‘’verilog-a user devices”. O componente definido pelo usuário foi criado com sucesso e agora pode ser incluído em esquemáticos de circuitos como se fosse um componente nativo do Qucs (Figura 2.8).

Esse mesmo tutorial pode ser utilizado para criar componentes definidos pelo usuário usando códigos em Verilog-AMS. Vale ressaltar que o Qucs tem suporte limitado ao Verilog-AMS, visto que ele depende do ADMS para realizar o *parse* e gerar o respectivo código para o simulador a partir do código Verilog-AMS.

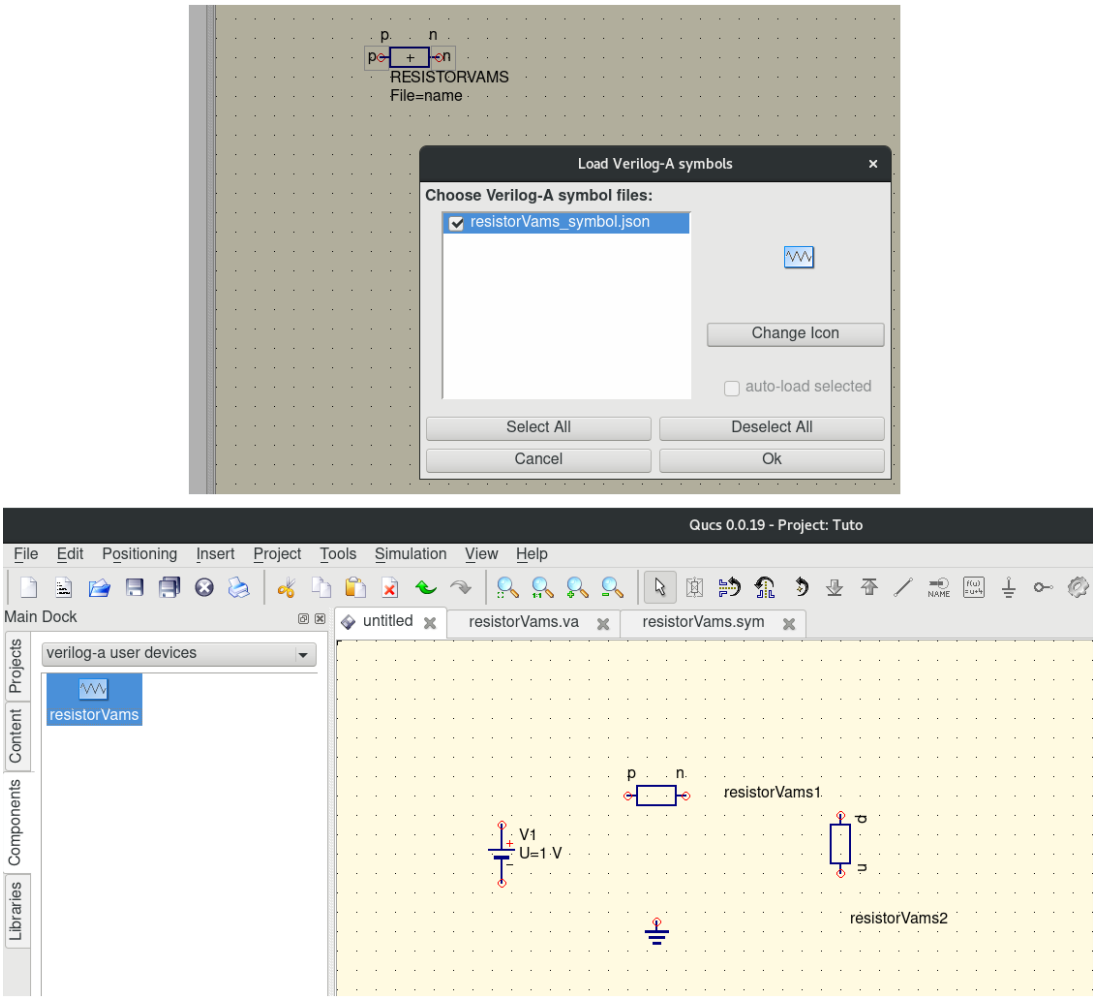

Figura 2.8: Exemplo de uso do componente definido pelo usuário.

2.4 GNU Octave

GNU Octave é um *software* com uma linguagem de programação de alto nível desenvolvida primariamente para cálculos matemáticos. Essa ferramenta consegue resolver problemas lineares e não-lineares usando uma linguagem que é, em grande parte, compatível com MATLAB. Ela também pode ser usada como uma linguagem orientada-a-*batch*. GNU Octave também é um *software* gratuitamente redistribuível sob os termos da GNU GPL.

Além de diversas funções próprias, o Octave também permite que o usuário crie suas próprias funções, utilize módulos carregados dinamicamente escritos em outras linguagens, como C++ e Fortran, crie gráficos e expanda a funcionalidade da ferramenta através do uso de pacotes com características específicas de uma área (por exemplo: processamento de imagens e sinais, estatísticas).

2.5 Conversor Analógico-Digital (A/D)

As grandezas observáveis em fenômenos físicos são em geral de natureza analógica, ou seja, podem assumir qualquer valor real. O processamento e captura dessas grandezas no universo digital exige que uma conversão de dados entre o mundo analógico e digital. Por exemplo, se um sistema digital receber um sinal analógico, este sistema não será capaz de utilizar esses dados pois eles não estão representados em um formato suportado pelo sistema, sendo necessário um processo de conversão desses dados para o formato digital. Para isso, necessitamos de circuitos capazes de realizar essa conversão ao mesmo tempo que se mantém certas características do sinal real. Ao circuito responsável por realizar esse processo dá-se o nome de conversor analógico-digital (conversor A/D ou ADC).

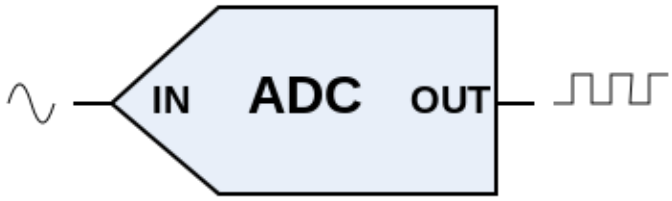

Figura 2.9: Representação de um conversor A/D.

Existem diferentes técnicas de conversores A/D, tais como aproximações sucessivas; rampa única; rampa dupla; Sigma-Delta, dentre outras. Cada uma dessas técnicas implementa o processo de conversão de maneira diferente, possuindo suas próprias vantagens e desvantagens, mas no final, todas desempenham a mesma função: converter um sinal analógico para digital.

Segundo Allen [4], a entrada de um ADC é um sinal analógico, geralmente uma tensão analógica, e a saída é um código digital. A entrada analógica pode possuir qualquer valor entre uma tensão de referência negativa (V_{ref}^{-}) e uma tensão de referência positiva (V_{ref}^{+}), enquanto que o código digital é restrito a amplitudes fixas ou discretas. Portanto, deve-se garantir que as amplitudes do sinal a ser convertido estejam dentro dos limiares estabelecidos pelo conversor A/D.

A escala de codificação mais comum para a saída digital é a binária, mas existem outras escalas, como a Grey e o complemento a dois que, para determinadas aplicações, podem possuir características mais interessantes do que a escala binária.

Os conversores analógico-digitais realizam o processo de conversão através de duas etapas: amostragem e quantização. Amostragem é o processo de obtenção de amostras de um sinal durante um determinado intervalo de tempo. A quantização consiste em mapear um conjunto de valores para um conjunto menor, sendo que esse mapeamento pode gerar,

ou um espaçamento igual (uniforme), ou um espaçamento variável (não-uniforme) entre os valores do conjunto menor.

2.5.1 Amostragem

Amostragem é o processo de obtenção de amostras de um sinal contínuo, durante um determinado intervalo de tempo.

O teorema da amostragem [5] afirma que se a maior frequência no espectro de um sinal é B (em hertz), então o sinal pode ser reconstruído a partir de suas amostras obtidas a uma frequência maior ou igual a 2B amostras por segundo. Portanto, é possível transmitir a informação de um sinal contínuo no tempo apenas transmitindo suas amostras.

O teorema da amostragem é importante pois permite que representemos um sinal contínuo no tempo por meio de um sequência discreta de números. Logo, processar um sinal contínuo no tempo é equivalente a processar uma sequência discreta de números [6].

2.5.2 Quantização

Em processamento de sinais digitais, quantização pode ser definido como o processo de mapear um grande conjunto de valores de entrada para um conjunto menor de valores discretos. Neste trabalho, a quantização irá atribuir um valor numérico discreto a uma faixa de valores contínuos de uma variável real.

Enquanto o conjunto de valores de entrada pode ser infinitamente grande ou até mesmo contínuo, o conjunto de valores de saída deve ser discreto e, portanto, contável. Como a quantização é um processo que restringe um certo conjunto de valores para um conjunto menor, onde um mesmo valor de saída é atribuído a múltiplos valores de entrada, é impossível recuperar o exato valor da entrada a partir do valor de saída.

Existem dois tipos de quantização, a quantização uniforme e a não-uniforme, e elas serão apresentadas a seguir.

Quantização Uniforme

A quantização uniforme é definida com intervalos de mesma largura, ou seja, uniformemente espaçados.

Em telecomunicações, foram desenvolvidas diferentes técnicas de transmissão de sinais contínuos no tempo através de trens de pulsos. Dado um sinal contínuo no tempo g(t) e um trem de pulsos periódicos, podemos variar as amplitudes, as larguras ou as posições dos pulsos em proporção aos valores amostrados do sinal g(t), obtendo, respectivamente, modulação por amplitude de pulso (PAM), modulação por largura de pulso (PWM) e

modulação por posição de pulso (PPM). Do ponto de vista da quantização, é importante estudarmos a modulação por código de pulso (PCM), que será apresentada na próxima sessão. Em todos esses tipos de modulação, ao invés de transmitir o sinal analógico g(t), transmite-se o sinal modulado por pulso e o receptor irá recriar g(t) a partir desse sinal modulado por pulso.

Uma vantagem de se utilizar modulação por código de pulso é que ela permite a transmissão simultânea de outros sinais com base na divisão do tempo (*time-division multiplexing*). Como um sinal modulado por pulso só ocupa uma fração do tempo do canal, é possível transmitir mais sinais modulados por pulso no mesmo canal [6].

- **Pulse-Code-Modulation (PCM)**

PCM é um tipo de modulação bastante utilizada em computadores e telefonia digital, pois ela é basicamente uma técnica para conversão de sinais analógicos em digitais.

O sinal analógico a ser convertido é representado pela curva m(t), cujas amplitudes encontram-se no intervalo $(-m_p, m_p)$. Note que m_p não é necessariamente o valor de pico da curva m(t), portanto m_p não é um parâmetro do sinal m(t), mas sim, uma constante do quantizador.

O intervalo da amplitude $(-m_p, m_p)$ é dividido em L subintervalos uniformemente espaçados, cada um com largura igual a

$$\Delta v = \frac{2 * m_p}{L} \tag{2.1}$$

O número de subintervalos L depende do número de bits (N) utilizados para representar o sinal digital, sendo que

$$L = 2^N \tag{2.2}$$

O valor da amostra é aproximado pelo ponto médio do subintervalo em que se encontra, gerando a amostra quantizada (Figura 2.10). As amostras quantizadas são codificadas para um dos valores de L e transmitidas para o receptor como pulsos binários. No receptor, alguns pulsos podem ser detectados incorretamente.

Portanto, no esquema apresentado, podemos ter dois tipos de erro, erro de quantização e erro de detecção de pulso. Na prática, o erro na detecção de pulso é bastante pequeno em comparação com o erro de quantização, podendo ser desprezado [6]. Logo, iremos assumir que o erro no sinal recebido é atribuído exclusivamente ao processo de quantização. Como o valor da amostra é aproximado pelo ponto médio do subintervalo no qual a amostra se encontra, sendo Δv a largura do intervalo, então o erro máximo de quantização é $\pm\Delta v/2$. Assim, o erro de quantização está no intervalo $(-\Delta v/2, \Delta v/2)$.

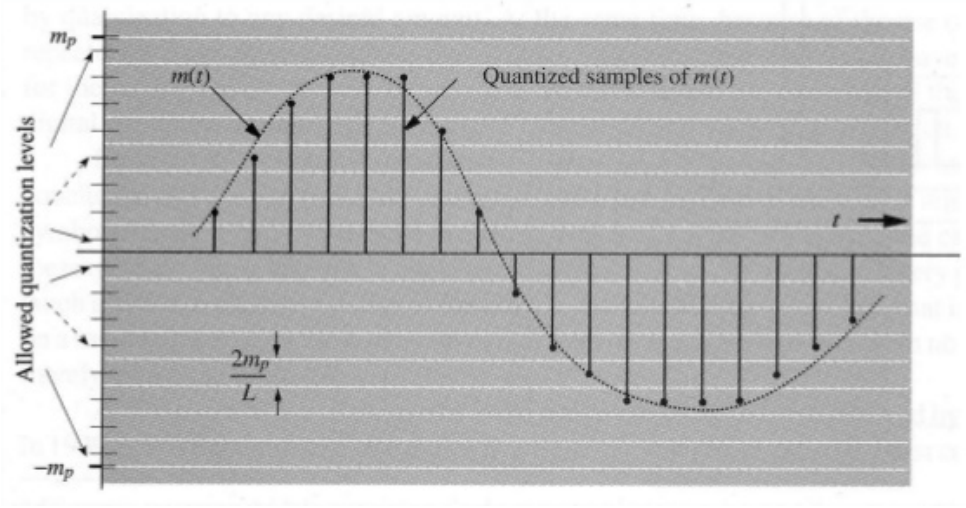

Figura 2.10: Quantização de um sinal analógico usando PCM. (Fonte: [6]).

Quantização Não-Uniforme

Na quantização uniforme temos que todos os intervalos de quantização possuem a mesma largura. Na quantização não-uniforme, esses intervalos possuem valores variados, ou seja, eles não estão igualmente espaçados. A motivação por trás dessa técnica consiste em atribuir os níveis de quantização de maneira que a maioria dos níveis fiquem nos intervalos dos valores de amostras que ocorrem com maior frequência, e uma quantidade menor de níveis de quantização para os intervalos com os valores menos frequentes. Dependendo das características do sinal (ex: intervalos bem definidos dos valores mais frequentes), essa técnica permite o aumento da SNR (Signal-to-Noise Ratio), pois os valores menos frequentes serão penalizados, mas os valores mais frequentes serão beneficiados.

Por exemplo, imagine que temos um sinal qualquer representado pela Figura 2.11 com seu respectivo histograma (Figura 2.12). Pelo histograma desse sinal, nota-se que a maioria das amostras possui uma amplitude menor ou igual 5, enquanto que as amostras com uma maior amplitude ocorrem poucas vezes. Usando a lógica da quantização não-uniforme, seria mais interessante que a maioria dos níveis de quantização se encontre no intervalo $[1, 5]$, onde as amostras são mais frequentes. Isso traria uma maior precisão na quantização desse intervalo. Por outro lado, teríamos menos níveis para representar os valores mais altos, diminuindo a precisão na quantização dessas amostras. Então, a quantização não-uniforme poderia ter um SNR maior em detrimento da representação de suas amostras menos frequentes.

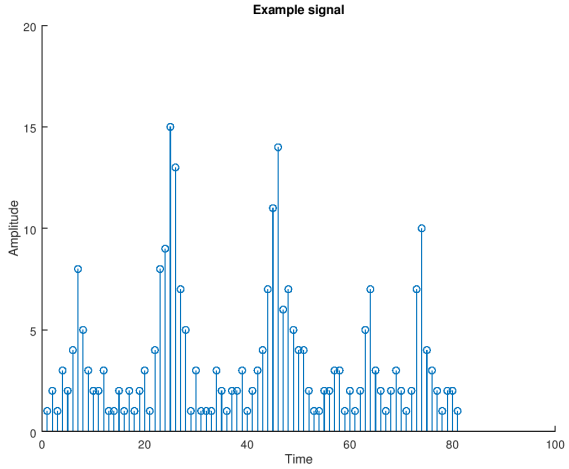

Figura 2.11: Sinal de exemplo

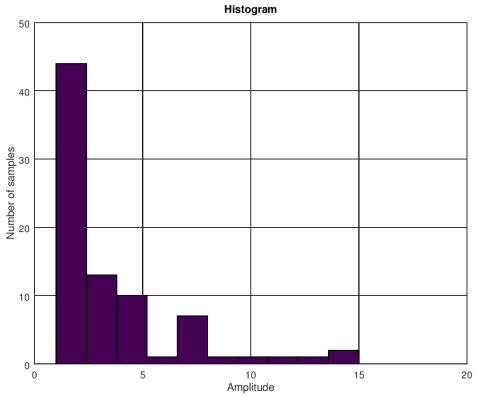

Figura 2.12: Histograma da Figura 2.11

- ***μ-Law***

O algoritmo da lei-μ (μ-*law*), também aproximada por lei-u, é um algoritmo de compressão e expansão usado principalmente em sistemas de telecomunicações por Modulação por Código de Pulso (PCM) de 8 *bits* na América do Norte e no Japão. Esse algoritmo é um dos dois algoritmos apresentados no padrão G.711 da ITU-T (International Telecommunication Union Telecommunication Standardization Sector), sendo que o outro algoritmo (lei-A) é bastante parecido, porém é usado na Europa e no resto do mundo.

Algoritmos de compressão e expansão, representados pelo termo em inglês *companding*, visam reduzir os efeitos de ruído e distorção de um canal com um intervalo dinâmico limitado. Em sistemas analógicos, pode-se obter um aumento da SNR, enquanto que em sistemas digitais pode-se obter uma redução no erro de quantização.

O algoritmo de compressão (Equação 2.3) é utilizado pelo transmissor do sinal, enquanto que o algoritmo de expansão (Equação 2.4) é aplicado no receptor. Dado um sinal de entrada "x", o algoritmo de compressão da lei-μ pode ser descrito na forma analógica segundo a equação:

$$F(x) = sgn(x) * \frac{ln(1 + \mu * |x|)}{ln(1 + \mu)} \tag{2.3}$$

onde "sgn(x)" é uma função que indica o sinal do valor de entrada "x" e $\mu = 255$ para aplicações com 8 *bits* (padrão norte americano e japonês). A curva de saída dessa equação encontra-se no intervalo $[-1, 1]$.

O algoritmo de expansão é dado pela equação inversa (Equação 2.4), que também possui a curva de saída no intervalo $[-1, 1]$:

$$F^{-1}(y) = sgn(y) * \frac{1}{\mu} * ((1+\mu)^{|y|} - 1) \tag{2.4}$$

Capítulo 3

Implementação

Este capítulo visa apresentar o modelo de quantizador não-uniforme utilizado e o modelo proposto em Verilog-A.

3.1 Relembrando objetivos e proposta

O objetivo deste trabalho é desenvolver um modelo de conversor A/D em Verilog-AMS e integrá-lo a plataforma openMSP430.

O modelo inicial do conversor utilizará uma quantização uniforme para criar níveis de quantização igualmente espaçados. Posteriormente, uma segunda versão do conversor A/D será desenvolvida implementando a quantização não-uniforme (baseada na Equação 2.3) para criar níveis de quantização com espaçamento variáveis.

A proposta deste trabalho é desenvolver o modelo de um conversor A/D em Verilog-AMS, que possa eventualmente servir de estudo para a comunidade científica (openMSP430, comunidade *open source*, etc).

3.2 Modelos de quantizadores

O modelo do quantizador uniforme se baseou na PCM para criar intervalos de quantização uniformemente espaçados. Porém, para simplificar o código, ao invés de realizarmos a aproximação do valor da amostra pelo ponto médio do subintervalo no qual a amostra se encontra, utilizou-se a função ‘’floor()”. O uso dessa função fez com que a aproximação do valor da amostra fosse para o nível cujo valor inteiro mais próximo seja menor ou igual ao valor da amostra. Essa troca de valores de aproximação não altera o fato de que as amostras foram quantizadas com intervalos uniformes, sendo apenas uma simplificação no código. Para se obter um resultado utilizando a aproximação pelo ponto médio do

subintervalo, bastaria que somássemos $\frac{V_{ref}^{+}-V_{ref}^{-}}{2^{N+1}}$ ao resultado.

O modelo do quantizador não-uniforme se baseou na lei-μ (Equação 2.3) para criar intervalos de quantização com espaçamentos não-uniformes. Como a lei-μ normaliza as amostras do sinal, foi necessário denormalizá-las antes de quantizá-las.

Em ambas as quantizações, calculamos o erro de quantização como sendo a diferença entre os valores na função de entrada ''vin" e o valor amostrado pela quantização, sendo representado pela seguinte equação:

$$\delta(t) = \alpha(t) - \beta(t) \tag{3.1}$$

onde α representa o sinal de entrada e β representa o sinal quantizado.

Então, o erro acumulado na "i-ésima" amostra pode ser calculado como:

$$\sum_{i=1}^{lenght(\delta)} \delta(i) \tag{3.2}$$

O código Octave foi desenvolvido em duas partes. A primeira é uma função para realizar o cálculo da quantização não-uniforme baseado no algoritmo de compressão da lei-μ. A segunda parte corresponde a quantização uniforme, a chamada da função de quantização não-uniforme, o cálculo dos erros de quantização, obtenção do Ponto Ótimo e a geração dos gráficos. Os códigos de ambas as partes são apresentadas mais detalhadamente em A.1.

Além disso, o código Octave foi executado várias vezes utilizando diferentes valores de μ (μ = 1, 10, 50, 100, 255, 500 e 1000) para que pudéssemos avaliar o efeito da constante μ sobre o processo de conversão A/D usando a quantização não-uniforme.

3.3 Modelo Verilog-A

O código de conversor A/D presente em I.1 é apresentado em Kundert [1] e foi utilizado como base de aprendizado da linguagem Verilog-AMS. Esse código seria utilizado no Qucs para criar um componente que representasse o funcionamento de um conversor A/D. Posteriormente, iríamos adaptar este código para as funcionalidades propostas por este trabalho.

Capítulo 4

Resultados

Este capítulo apresenta os resultados obtidos, juntamente com a interpretação dos mesmos.

4.1 Resultados

A seguir, serão apresentados os gráficos contendo as curvas que demonstram os erros das quantizações uniforme e não-uniforme aplicadas a um sinal de entrada ‘’vin”. Ambas as quantizações foram realizadas com 4 *bits*, ou seja, com 16 níveis de quantização.

A Figura 4.1 apresenta a quantização uniforme, sendo que esta imagem encontra-se dividida em dois gráficos: as amostras obtidas com a quantização uniforme e o erro de quantização.

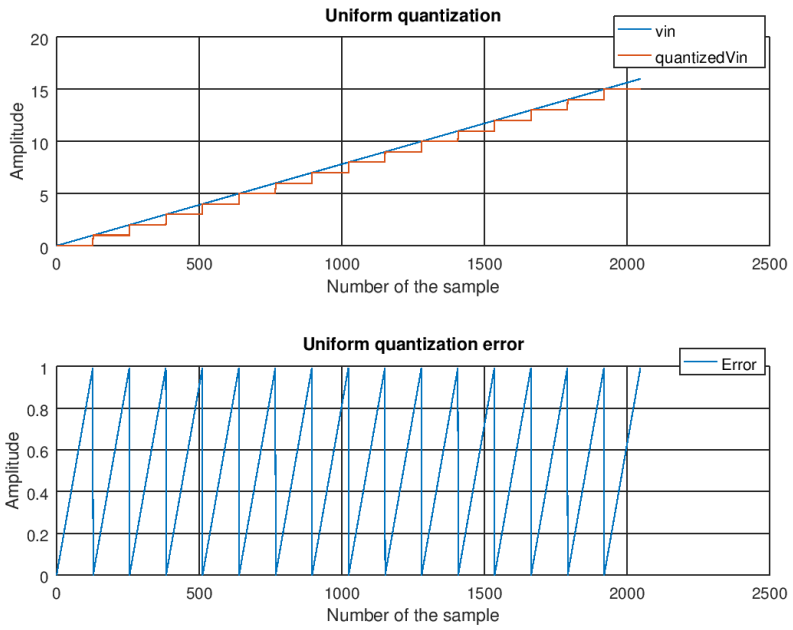

Figura 4.1: Quantização uniforme e seu erro.

O primeiro gráfico apresenta o processo de quantização uniforme de um sinal de entrada ‘’vin”, utilizando a função ‘’floor()”, conforme descrito na sessão 3.2. O segundo gráfico apresenta o erro de quantização representado pela Equação 3.1.

Podemos observar que o erro da quantização uniforme possui a aparência de uma onda serra cíclica com uma amplitude máxima definida.

Para as próximas quatro figuras (Figuras 4.2, 4.3, 4.4 e 4.5), adotaremos o valor de $\mu = 255$ para a equação da quantização não-uniforme. Esse valor segue o padrão Norte Americano e Japonês para 8 *bits*.

O fato de estarmos usando 4 *bits* ao invés de 8 é para facilitar a ilustração do princípio de operação dos quantizadores, melhorando a compreensão das imagens obtidas. Pode-se ajustar o número de *bits* no modelo em Octave. As Figuras 4.8 a 4.12 mostram os resultados obtidos utilizando-se 8 *bits* e $\mu = 255$.

A Figura 4.2 apresenta a quantização não-uniforme, sendo que esta imagem encontra-se dividida em dois gráficos: as amostras obtidas com a quantização não-uniforme e o erro de quantização.

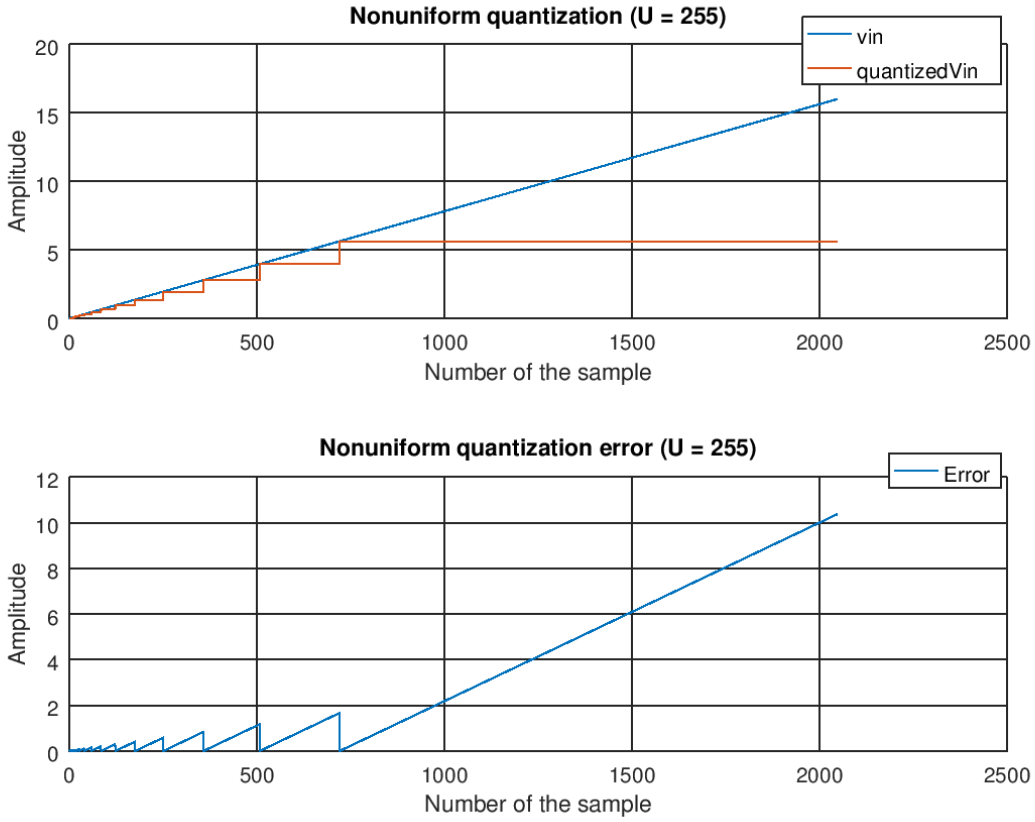

Figura 4.2: Quantização não-uniforme e seu erro ($\mu = 255$).

O primeiro gráfico apresenta o processo de quantização não-uniforme de um sinal de entrada ‘’vin” baseado na Equação 2.3 da lei-μ. O segundo gráfico apresenta o erro de quantização representado pela Equação 3.1.

O erro da quantização não-uniforme também possui a aparência de uma onda serra, mas ao contrário do erro da quantização uniforme, a amplitude máxima é crescente.

A Figura 4.3 apresenta uma comparação entre os erros das quantizações uniforme e não-uniforme. As curvas aqui apresentadas são as mesmas dos segundos gráficos das Figuras 4.1 e 4.2.

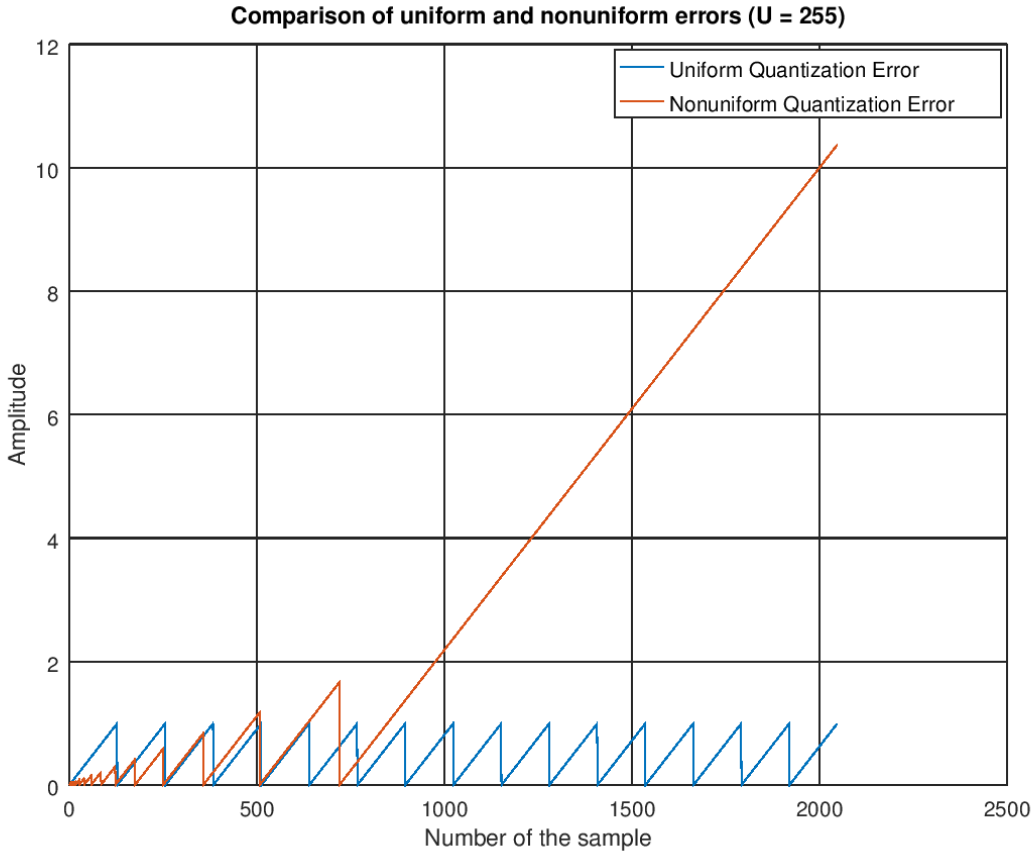

Figura 4.3: Comparação dos erros das quantizações uniforme e não-uniforme ($\mu = 255$).

Nota-se que, inicialmente, o erro da quantização não-uniforme é menor do que o erro da quantização uniforme. Posteriormente o erro não-uniforme torna-se maior do que o uniforme.

A Figura 4.4 apresenta uma comparação entre os erros acumulados das quantizações uniforme e não-uniforme. Essas curvas foram obtidas somando-se progressivamente os valores dos erros de quantização de todas as amostras, representados pela Equação 3.2.

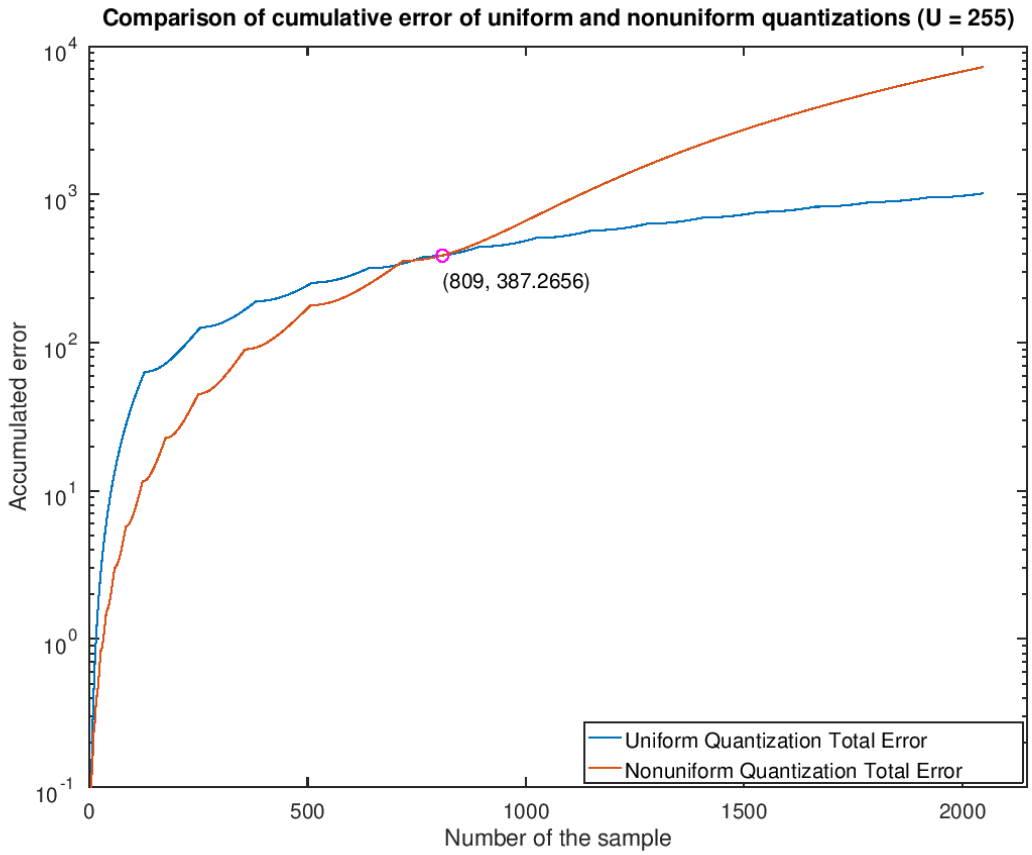

Figura 4.4: Comparação dos erros acumulados das quantizações uniforme e não-uniforme ($\mu = 255$).

Podemos observar que, inicialmente, a quantização não-uniforme possui um erro acumulado menor que o da quantização uniforme. Após um certo valor de amplitude do sinal amostrado, o erro acumulado não-uniforme passa a ser maior do que o erro acumulado uniforme.

Daremos o nome de "Ponto Ótimo" (PO) a esse valor de amplitude a partir do qual o erro acumulado não-uniforme sempre será maior do que o erro acumulado uniforme (representado pelo ponto marcado no gráfico).

A Figura 4.5 apresenta a curva característica da lei-μ sobre o sinal de entrada "vin". A lei-μ (Equação 2.3) normaliza logaritmicamente o sinal de entrada, fazendo com que, quanto maior a razão entre a amostra de "vin" e a constante m_p, maior será a amplitude do sinal de saída.

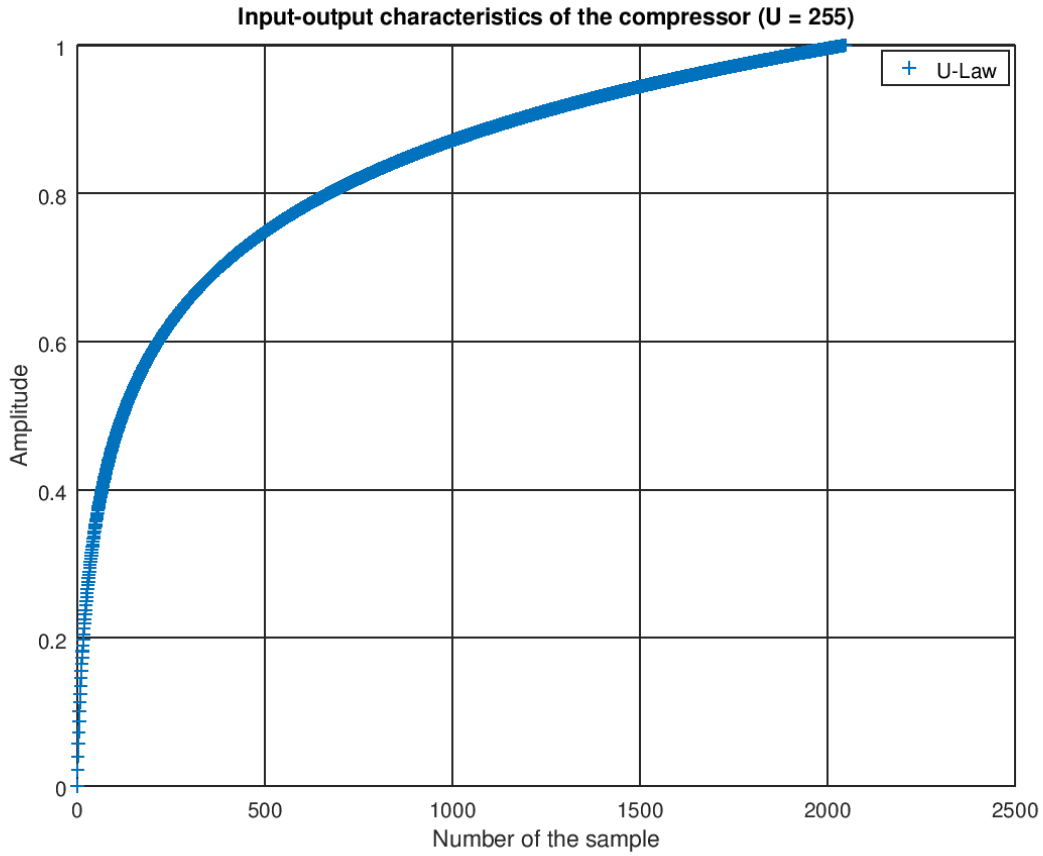

Figura 4.5: Curva característica do compressor para $\mu = 255$.

4.1.1 Quantização não-uniforme variando μ

Para avaliar o efeito da constante μ sobre o processo de conversão A/D usando quantização não-uniforme, o código Octave também foi executado utilizando os seguintes valores de μ: 1, 10, 50, 100, 255, 500 e 1000.

Como a alteração na constante μ só afeta a quantização não-uniforme, os resultados para os valores de μ previamente citados serão análogos aos apresentados nas Figuras 4.2 a 4.5.

A Figura 4.6 apresenta as curvas características do compressor para os diferentes valores de μ. Podemos observar que quanto menor o valor de μ, mais a curva se parece com a quantização uniforme ($\mu = 0$). Conforme aumenta-se o valor de μ, nota-se que a curva vai ficando mais acentuada e que os valores das amplitudes também aumentam.

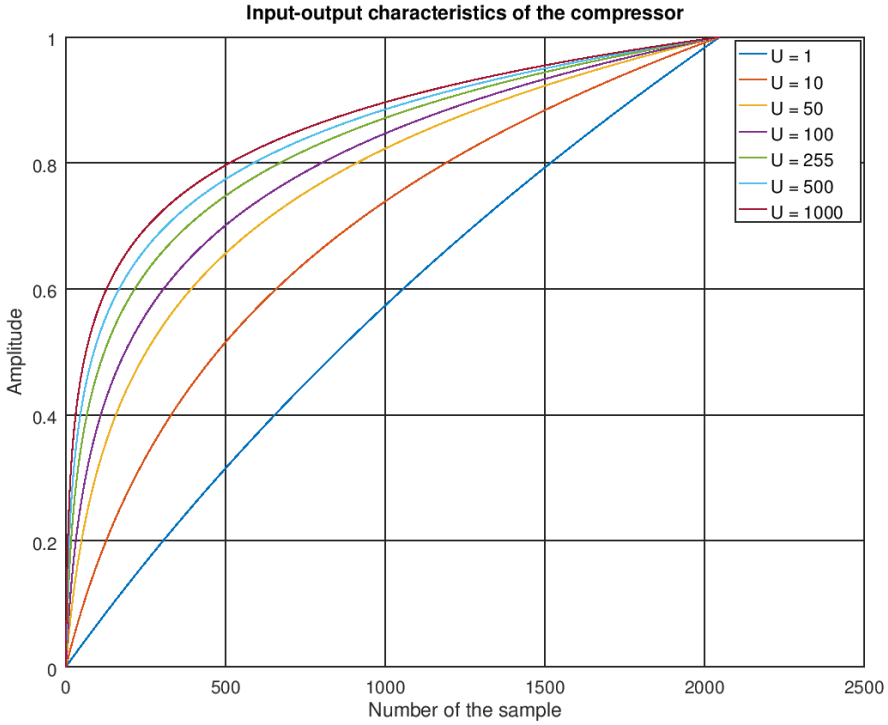

Figura 4.6: Curva característica do compressor para diferentes valores de μ.

Conforme dito anteriormente, adotamos a nomenclatura de "Ponto Ótimo" (PO) para designar o ponto a partir do qual o erro acumulado da quantização não-uniforme passa a ser maior do que o erro acumulado da quantização uniforme.

A Tabela 4.1 apresenta os valores de Pontos Ótimo para $\mu = 1$, 10, 50, 100, 255, 500 e 1000. A partir desses valores, geramos o gráfico apresentado na Figura 4.7, que representa a amplitude dos PO's em relação a variação de μ.

Tabela 4.1: PO para diferentes valores de μ

μ	Ponto Ótimo
1	829.062500
10	662.117187
50	510.109375
100	447.093750
255	387.265620
500	380.500000
1000	325.835937

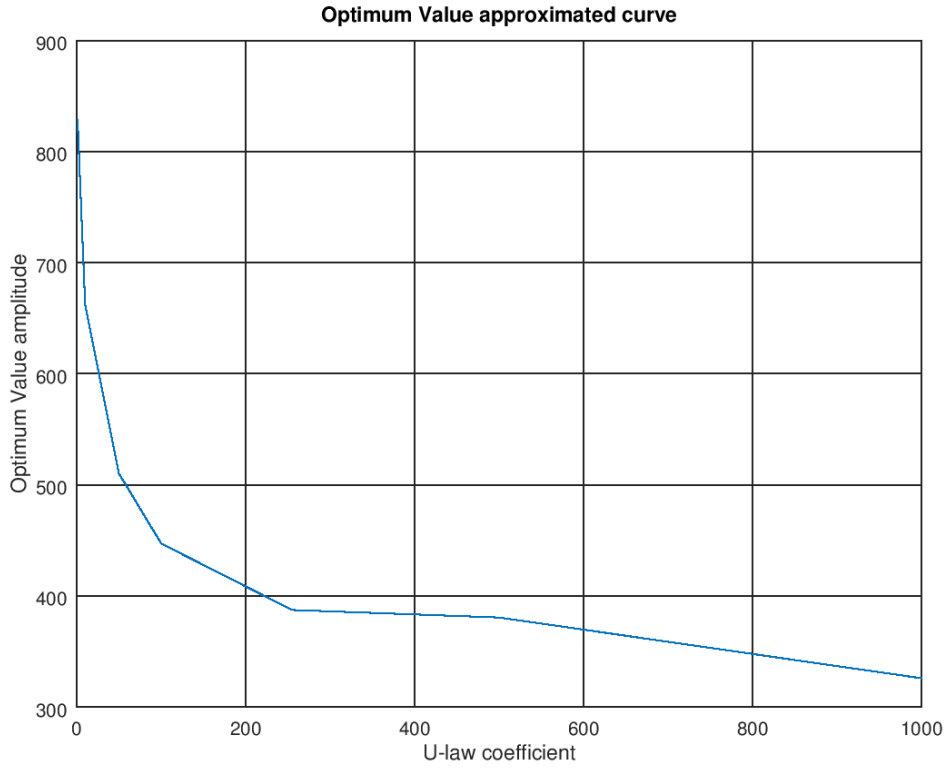

Figura 4.7: Amplitude dos PO conforme a Tabela 4.1.

Baseado na curva da Figura 4.7, podemos notar que o valor da amplitude do Ponto Ótimo diminui conforme aumenta-se o valor da constante μ. Isso indica que a quantização não-uniforme vem a ser mais interessante, do ponto de vista de aumentar a relação sinal-ruído do sinal, para sinais cuja distribuição probabilística de amplitude se encontra majoritariamente abaixo do Ponto Ótimo.

4.1.2 Quantização com 8 *bits*

Os resultados obtidos utilizando 8 *bits* e $\mu = 255$ são apresentados em Figuras 4.8 a 4.12:

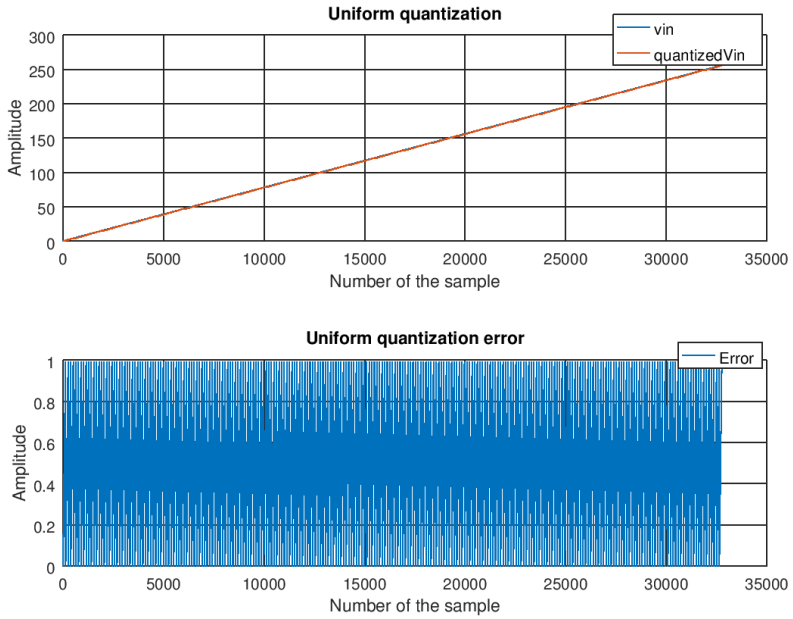

Figura 4.8: 8 bits: Quantização uniforme e seu erro.

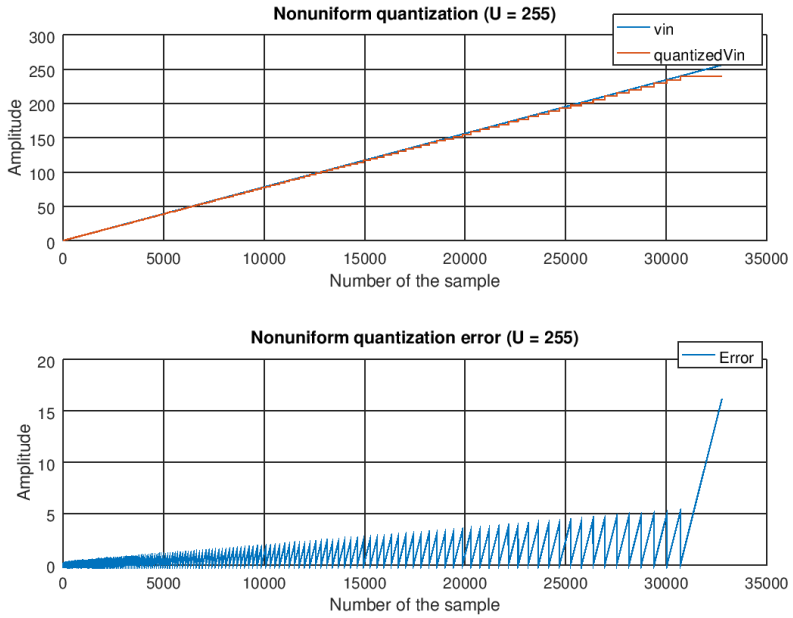

Figura 4.9: 8 bits: Quantização não-uniforme e seu erro ($\mu = 255$).

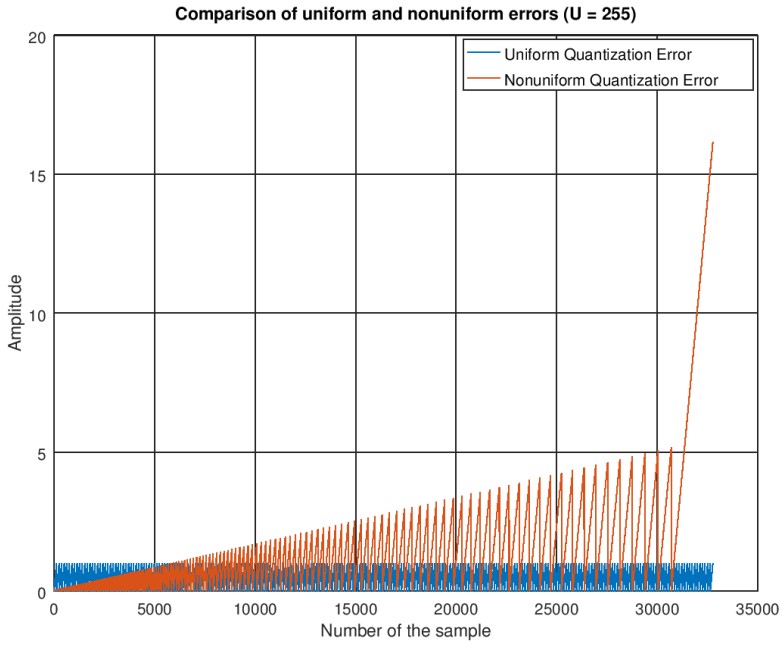

Figura 4.10: 8 bits: Comparação dos erros das quantizações uniforme e não-uniforme (μ = 255).

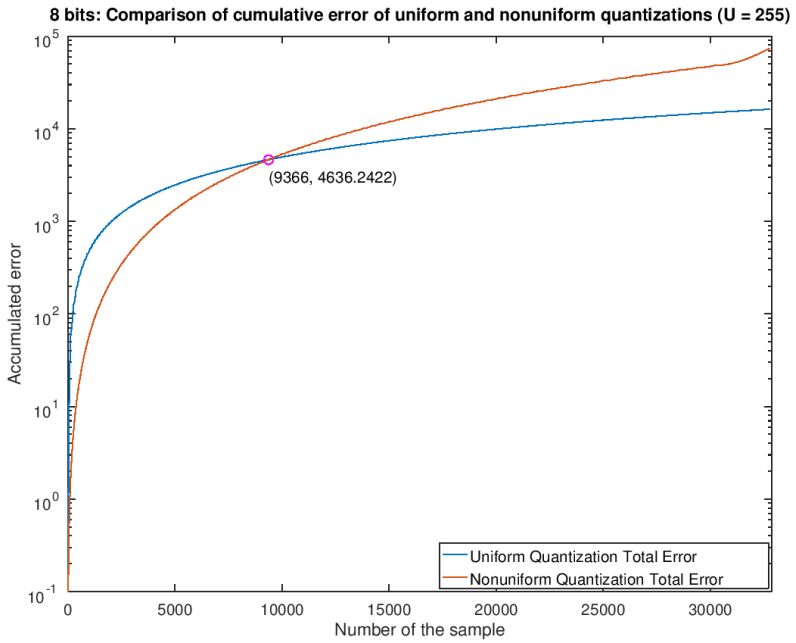

Figura 4.11: 8 bits: Comparação dos erros acumulados das quantizações uniforme e não-uniforme ($\mu = 255$).

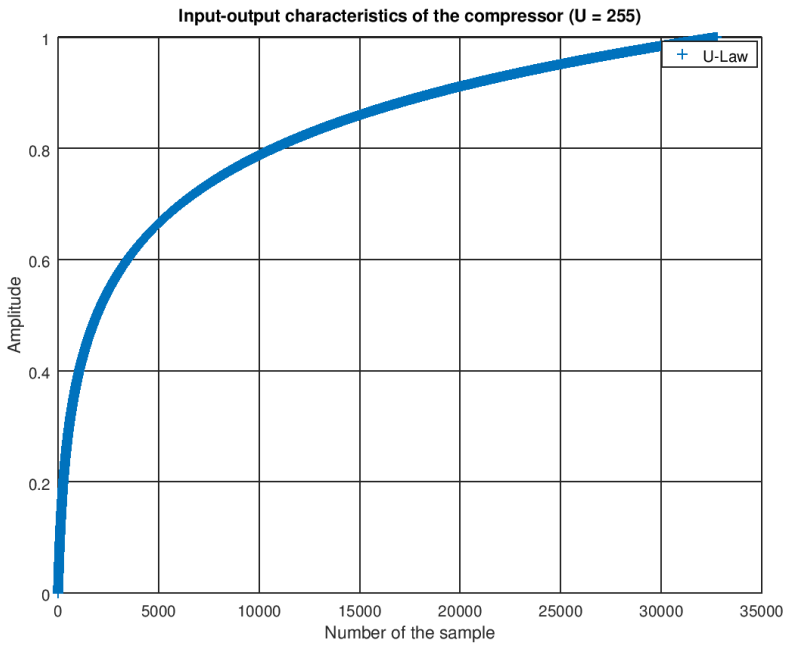

Figura 4.12: 8 bits: Curva característica do compressor para $\mu = 255$.

4.2 Relembrando os objetivos

Inicialmente o objetivo deste trabalho era desenvolver um modelo de conversor A/D em Verilog-AMS e integrá-lo a plataforma openMSP430. Esse modelo teria duas versões, uma utilizando quantização uniforme e a outra utilizando quantização não-uniforme baseada na lei-μ (Equação 2.3).

4.3 Resumo da implementação

Para o desenvolvimento do processo de quantização, utilizou-se a ferramenta Octave.

O código Octave pode ser analisado em quatro partes:

1. Implementação quantização uniforme;
2. Implementação quantização não-uniforme;
3. Obtenção dos erros das quantizações uniforme e não-uniforme;
4. Obtenção do Ponto Ótimo para um determinado valor de μ.

A quantização uniforme se baseou na PCM para criar intervalos igualmente espaçados, porém, por simplificação do código, utilizou a função ”floor()” conforme descrito na

sessão 3.2 na página 19.

A quantização não-uniforme se baseou na lei-μ (Equação 2.3) para criar intervalos de quantização com espaçamento variado.

O erro para ambas as quantizações seria calculado como sendo a diferença entre o sinal de entrada e o sinal quantizado, descrito pela Equação 3.1. O erro de quantização acumulado foi calculado usando a Equação 3.2.

O Ponto Ótimo foi definido como sendo o valor de amplitude a partir do qual o erro acumulado não-uniforme sempre será maior do que o erro acumulado uniforme.

Para calculá-lo, utilizamos as curvas dos erros acumulados de ambas as quantizações e fomos comparando os elementos da curva não-uniforme com os respectivos elementos da curva uniforme. A comparação se deu do último para o primeiro elemento.

Nas primeiras comparações, a amplitude do elemento não-uniforme é maior do que a amplitude do elemento uniforme. Enquanto a comparação for verdadeira, avançamos para o próximo elemento a ser comparado em cada curva, que no caso será um elemento com uma amplitude menor do que a anterior.

Quando a condição se tornar falsa, a amplitude do elemento não-uniforme é menor do que a do elemento uniforme, portanto, esta amplitude será o Ponto Ótimo.

4.4 Resumo dos resultados

Ao observar as Figuras 4.3, A.2, A.5, A.8, A.11, A.14 e A.17, nota-se que, para as amplitudes mais altas, a quantização não-uniforme tem um erro maior do que a quantização uniforme. Porém, para as amplitudes mais baixas, seu erro tende a ser menor do que o erro da quantização uniforme.

Esse fato fica mais evidente quando tomamos o somatório dos erros (erro acumulado) representados nas Figuras 4.4, A.3, A.6, A.9, A.12, A.15 e A.18. Nessas imagens, nota-se que as curvas se cruzam em um determinado ponto, referido neste trabalho como "Ponto Ótimo", e que a quantização não-uniforme tem um erro acumulado menor do que a quantização uniforme para as amplitudes inferiores a esse ponto.

Portanto, existe uma faixa de valores para a qual a quantização não-uniforme gera um erro acumulado de quantização menor do que a quantização uniforme.

A obtenção dos Pontos Ótimo para diferentes valores de μ (Tabela 4.1) é representado na Figura 4.7. Logo, para a entrada "vin", representada na Figura 4.1, temos que a

amplitude do ponto, a partir do qual o erro acumulado não-uniforme sempre será maior do que o erro acumulado uniforme, tende a diminuir conforme o valor de μ aumenta.

Capítulo 5

Conclusão

Este capítulo contém a conclusão referente aos resultados obtidos neste trabalho.

5.1 O código Verilog-AMS do conversor A/D

Não foi possível cumprir o nosso objetivo primário, que era desenvolver um conversor A/D em Verilog-AMS, pois não conseguimos executar um código Verilog-AMS do conversor A/D (I.1) no Qucs e, portanto, não obtivemos o resultado esperado para esse trabalho.

A escolha da ferramenta Qucs para trabalhar com códigos Verilog-AMS acabou não sendo uma boa escolha de projeto, visto que uma demasiada quantidade de tempo e esforço foram voltadas para o aprendizado do uso da ferramenta, porém a mesma não possui um suporte satisfatório para se trabalhar com a linguagem Verilog-AMS. No início do trabalho, realizou-se um estudo sobre eventuais ferramentas que poderiam ser utilizadas para a realização do mesmo e optou-se pela escolha do Qucs devido a algumas de suas características, como *software* livre; recomendado pela comunidade *open source* para simulações analógicas e digitais; e suporte para modelos definidos pelo usuário escritos em Verilog-AMS.

Apesar da página de suporte do Qucs[1] possuir uma sessão chamada "Verilog-AMS interface" e disponiblizar o arquivo "Qucs - A Description, Verilog-AMS interface" [7], o mesmo encontra-se desatualizado. Aparentemente esse tutorial foi escrito em 2006 e o código do Qucs provavelmente sofreu muitas modificações de lá para cá, pois alguns arquivos e trechos de códigos presentes no tutorial são diferentes ou inexistentes na versão atual da ferramenta.

O tutorial apresentado na sessão 2.3.1 se baseou em Brinson e Jahn [3] e Brinson et al [8], que são documentações escritas por desenvolvedores do Qucs para a criação

[1] `http://qucs.sourceforge.net/docs.html`

de modelos usando Verilog-A mas que também indicam um processo semelhante para Verilog-AMS.

Por fim, vale ressaltar que o Qucs encontra-se em desenvolvimento desde 2003 e ainda não possui uma versão final [2]. Mesmo possuindo diversas características interessantes para criar e simular circuitos, o *software* possui alguns *bugs* que podem atrapalhar a obtenção de resultados 100% confiáveis.

5.2 Algumas limitações do Qucs para Verilog-AMS

Jahn e Parruitte [7] nos apresentam algumas limitações do ADMS como:

- Não é permitido misturar contribuições de corrente e de carga em atribuições;
- Um potencial imediato no lado direito não é permitido incorporado em uma função;
- Não são permitidos potenciais no lado esquerdo de atribuições de contribuição.

Ao utilizar o código base de um conversor A/D (I.1) em Verilog-AMS e tentarmos realizar a etapa de compilação de um código no Qucs (sessão 2.3.1), obtivemos erros de compilação em alguns itens:

- Vetor: um erro de compilação indicava que o Qucs não suportava o uso de vetores, como a variável "result" utilizada para armazenar o código em 8 *bits*;
- Genvar: a palavra reservada "genvar" não era reconhecida pelo compilador;
- Declaração de contribução de tensão: não são permitidas atribuições de contribuições para tensões, como por exemplo, "$V(a,b) < +R * I(a,b)$".

Ao entrar em contato com o suporte do Qucs e pesquisando em fóruns da ferramenta, obtivemos algumas respostas para o motivo de tais erros ocorrerem. O Qucs depende do ADMS para realizar o *parse* e gerar o código a partir da fonte Verilog-AMS. Porém, apenas um subconjunto da linguagem Verilog-AMS foi implementada no *parser* e alguns construtores ainda não foram implementados no Qucs. Uma lista com os construtores disponíveis e implementados no Qucs está disponível no diretório de instalação do Qucs (ex: $''/qucs - core/src/components/verilog/''$). Portanto, para que uma instrução em Verilog-AMS seja corretamenta interpretada pelo Qucs, é necessário que ela tenha sido implementada tanto pelos desenvolvedores do ADMS, quanto pelos do Qucs.

[2] `http://qucs.sourceforge.net/index.html`

5.3 Quantização não-uniforme

A quantização não-uniforme foi estudada com o modelo escrito em Octave. Por meio desse código, pudemos analisar a resposta da quantização não-uniforme para a variação da constante μ no sinal de entrada "vin" e constatarmos a sua relação com o intervalo de amplitudes em que a quantização não-uniforme possui um erro acumulado de quantização menor do que a uniforme. A Figura 4.7 nos mostra que conforme o valor da constante μ aumenta, tem-se uma diminuição na amplitude do Ponto Ótimo. Isso indica que o intervalo de valores, para os quais a quantização não-uniforme apresenta um erro acumulado menor do que o da quantização uniforme, tende a diminuir conforme aumenta-se o valor de μ.

Referências

[1] Kundert, Ken e Olaf Zinke: *The designer's guide to Verilog-AMS*. Springer Science & Business Media, 2006. 3, 4, 20, 53

[2] Verilog, HDL: *Verilog-ams language reference manual*. 4

[3] Brinson, Mike e Stefan Jahn: *Notes on constructing qucs verilog-a compact device models and circuit macromodels*, 2011. 10, 33

[4] Allen, Phillip E e Douglas R Holberg: *CMOS analog circuit design*. Oxford Univ. Press, 2002. 13

[5] Jerri, Abdul J: *The shannon sampling theorem—its various extensions and applications: A tutorial review*. Proceedings of the IEEE, 65(11):1565–1596, 1977. 14

[6] Lathi, Bhagwandas P.: *Modern Digital and Analog Communication Systems*. Oxford University Press, Inc., New York, NY, USA, 2nd edição, 1990, ISBN 003027933X. 14, 15, 16

[7] Jahn, Stefan e Helene Parruitte: *Qucs a description - verilog-ams interface*. 2006. 33, 34

[8] Brinson, Mike, Richard Crozier, Clemens Novak, Bastien Roucaries, Frans Schreuder e Guilherme B. Torri: *Building a second generation qucs gpl circuit simulator: package structure, simulation features and compact device modelling capabilities*. 2014. 33

Apêndice A

Resultados da quantização não-uniforme e códigos utilizados

A.1 Códigos em Octave

Os dois códigos Octave presentes nessa sessão representam o processo de quantização de um conversor A/D. O código Octave foi desenvolvido em duas partes:

- Uma parte corresponde ao código principal que é responsável por realizar a quantização uniforme, a chamada da função de quantização não-uniforme, o cálculo dos erros de quantização, a obtenção do Ponto Ótimo e a geração dos gráficos.
- E outra parte é uma função para realizar o cálculo da quantização não-uniforme baseado na lei-μ.

A.1.1 Função de quantização

A função "NonuniformQuantULaw" foi desenvolvida em um arquivo a parte com o intuito de que o processo de quantização não-uniforme se tornasse parametrizável e que diferentes resultados pudessem ser facilmente obtidos apenas alterando-se os parâmetros desejados.

A função "NonuniformQuantULaw" possui como parâmetros de entrada, respectivamente: o sinal de entrada, o vetor de índices do sinal de entrada, o número de níveis de quantização e o valor da constante μ. Como parâmetros de saída temos, respectivamente: a curva característica do compressor, a curva denormalizada da característica do compressor conforme o número de níveis e o sinal quantizado. A curva característica do compressor encontra-se representada nas Figuras 4.5, 4.12 e 4.6. O sinal quantizado não-uniformemente encontra-se representado nas Figuras 4.2, 4.9, A.1, A.4, A.7, A.10, A.13 e A.16. A curva denormalizada da característica do compressor foi colocada como parâmetro de saída para um eventual uso na criação de gráficos, mas estes acabaram não

sendo relevantes para a versão final do trabalho.

O funcionamento da função "NonuniformQuantULaw" ocorre da seguinte maneira:

1. Utiliza-se um *loop* para aplicar a equação de expansão da lei-μ (Equação 2.3) e verificar se o sinal de entrada ("vin") encontra-se dentro do limite esperado, no caso, dentro do intervalo normalizado $[0, 1]$. A variável "y" é um vetor que armazena as amplitudes geradas pela equação de compressão.

2. Como a curva de saída da equação se encontra normalizada no intervalo $[0, 1]$, denormalizamos essa curva usando o número de níveis de quantização utilizados ("L"). Então, percorremos essa curva denormalizada (o vetor "denormY") e armazenamos em um vetor auxiliar ("vect") o primeiro índice cuja amplitude seja maior do que um nível de quantização. Esse processo se repete para todos os "L" níveis de quantização, portanto, "vect" possui "L" elementos.

3. O vetor auxiliar é usado para gerar a saída que encontra-se representada por "output". A saída é gerada a partir de uma comparação entre cada um dos índices do vetor de índices e os índices armazenados em "vect". Se o índice for maior do que o valor de "vect", o novo elemento de "output" será o valor de "vect". Se for menor ou igual, o novo elemento será o elemento anterior de "vect". Isso faz com que uma saída seja repetida até que o próximo nível de quantização seja alcançado. A saída encontra-se em função do vetor de índices ("index") e, portanto, deve ser adaptada para o valor máximo de "L". Esse último passo ocorre no código que realizou a chamada para esta função.

```
function [y, denormY, output] = NonuniformQuantULaw(vin, index, L
    , U)
% A/D Converter using U-law Nonuniform Quantization

for i=1:length(vin) % A-law equation
  if (0 <= (vin(i)/L)) && ((vin(i)/L) <= 1)
    y = 1./(log (1 + U)) .* (log(1 + U*vin/L)); % U-law equation
  else
    error("Invalid message value.");
  end
end
denormY = y * L;  % denormalize U-law curve (which goes from 0 to
    1), so it can be represented by the same scale as vin (from 0
   to L)
```

```
12
13    % For each level, get the first denormalized index which is
          greater than quantization level
14    vect = 0;
15    for i = 1:L
16      vect = [vect find(denormY>i,1)];
17    end
18
19    output = [];
20    j = 2;
21
22    % Quantizes nonuniform wave by giving the same output until the
          next level
23    for i = 1:length(index)
24      if (index(i) > vect(j))
25        if j < (L-1)
26          j++;
27        end
28      end
29      output = [output vect(j-1)];
30    end
31  endfunction
```

A.1.2 Código principal

O código principal é responsável por realizar a quantização uniforme, a chamada para a função de quantização não-uniforme, o cálculo dos erros de quantização, obtenção do Ponto Ótimo e a geração dos gráficos.

Inicialmente temos duas variáveis que podem ser definidas pelo usuário: o número de *bits* utilizados pela quantização ("N") e a precisão da amostragem ("overSampling"). O número de *bits* define o número de níveis de quantização conforme a Equação 2.2. O valor de "overSampling" foi escolhido arbitrariamente mas possui a ressalva de que quanto maior for o seu valor, maior será a quantidade de amostras a serem quantizadas. O vetor de índices ("index") e o sinal de entrada ("vin") são gerados a partir dessas variáveis.

A quantização uniforme é realizada a partir de um "floor()" no sinal de entrada (conforme descrito na sessão 3.2) e o erro é obtido como sendo a diferença entre o sinal de entrada "vin" e o sinal quantizado "quantizedVin" (Equação 3.1).

A constante μ também pode ser definida pelo usuário, sendo que esta encontra-se representada pela variável "U". A quantização não-uniforme é realizada chamando-se a função "NonuniformQuantULaw". A saída da função ("NonQuant") é normalizada para o intervalo [0, L], gerando a saída quantizada "vNonQuant" e o erro é obtido como sendo a diferença entre o sinal de entrada "vin" e o sinal quantizado "vNonQuant" (Equação 3.1).

Os erros acumulados das quantizações uniforme e não-uniforme são obtidos por meio da Equação 3.2 e armazenados em "error1sum" e "error2sum", respectivamente.

Conforme definido no Capítulo 4, "Ponto Ótimo" (PO) é o valor de amplitude a partir do qual o erro acumulado não-uniforme sempre será maior do que o erro acumulado uniforme. A variável "optimumPoint" foi designada para armazenar o índice do Ponto Ótimo obtido na curva do erro acumulado da quantização não-uniforme.

Para calcular esse índice, utilizamos as curvas dos erros acumulados de ambas as quantizações e fomos comparando os elementos da curva não-uniforme com os respectivos elementos da curva uniforme. Como a curva do erro acumulado possui um aspecto crescente, a comparação entre os erros acumulados se deu do último elemento da curva para o primeiro elemento. Se o erro acumulado não-uniforme não for menor do que o erro acumulado uniforme, comparamos o próximo elemento. Se o erro for menor, encontramos o Ponto Ótimo cuja amplitude será armazenada em "optimumVal".

Por fim, geramos os gráficos para a quantização uniforme, a quantização não-uniforme, a comparação dos erros das quantizações, a comparação dos erros acumulados das quantizações e a curva característica do compressor (lei-μ).

```
% Initial variables
overSampling = 2^7; % refine sampling precision
N = 4;     % number of bits
L = 2^N;   % number of levels
index = (0:(L)*overSampling-1); % number of elements to be
    sampled
vin = index/overSampling;        % samples to be quantized, from
    [0:L]

% Uniform Quantization
quantizedVin = floor(vin);     % uniform quantization using floor
    function
error1 = vin - quantizedVin;  % uniform quantization error

% Nonuniform Quantization
U = 255;  % U-law value
```

```
[compressedVin denormalizedCompressedVin NonQuant] =
   NonuniformQuantULaw(vin, index, L, U);

vNonQuant = NonQuant/overSampling;
error2 = vin - vNonQuant;

%Error studies
for i = 1:length(index)
  error1sum(i) = sum(error1(1:i));
  error2sum(i) = sum(error2(1:i));
end

optimumPoint = length(error1sum);
flag = 0; % flag to check if the error amplitudes are equal
for i=length(error1sum):-1:1
  if error2sum(i) > error1sum(i)
    optimumPoint = i;
    flag = 0;
  elseif error2sum(i) == error1sum(i)
    optimumPoint = i;
    flag = 1;
  else
    if flag == 0  % if the previous samples were not equal, the
       previous sample number will be the highest amplitude
      optimumPoint--; % samples index (chart) goes from 0 to X-1,
          while error index (vector) goes from 1 to X
    end
    break;
  end
end

optimumVal = error2sum(optimumPoint);

figure(1);
subplot(211);
plot(index, vin, ";vin;", index, quantizedVin, ";quantizedVin;");
title('Uniform quantization');
xlabel('Number of the sample');
ylabel('Amplitude');
grid on;
```

```
subplot(212);
plot(index, error1, ";Error;");
title('Uniform quantization error');
xlabel('Number of the sample');
ylabel('Amplitude');
grid on;

figure(2);
subplot(211);
plot(index, vin, ";vin;", index, vNonQuant, ";quantizedVin;");
title('Nonuniform quantization (U = 255)');
xlabel('Number of the sample');
ylabel('Amplitude');
grid on;
subplot(212);
plot(index, error2, ";Error;");
title('Nonuniform quantization error (U = 255)');
xlabel('Number of the sample');
ylabel('Amplitude');
grid on;

figure(3);
plot(index, error1, ";Uniform Quantization Error;", index, error2
    , ";Nonuniform Quantization Error;");
title('Comparison of uniform and nonuniform errors (U = 255)');
xlabel('Number of the sample');
ylabel('Amplitude');
grid on;

textOffSet = 100;       % text offset in the chart
figure(4);
semilogy(index, error1sum, index, error2sum);   % log plot
hold on;
title('Comparison of cumulative error of uniform and nonuniform
    quantizations (U = 255)');
xlabel('Number of the sample');
ylabel('Accumulated error');
semilogy(optimumPoint, optimumVal, 'om');       % optimum point
    in the chart
```

```
87 text(optimumPoint, (optimumVal-textOffSet), ['(' num2str(
       optimumPoint) ', ' num2str(optimumVal) ')']);
88 grid off;
89 legend("Uniform Quantization Total Error", "Nonuniform
       Quantization Total Error",'location', "southeast");
90 axis ([0 (length(index)+100) 10^(-1) 10^5]);
91 hold off;
92
93 figure(5);
94 plot(index, compressedVin, "+;U-Law;");
95 title('Input-output characteristics of the compressor (U = 255)')
       ;
96 xlabel('Number of the sample');
97 ylabel('Amplitude');
98 grid on;
```

A.2 Resultados da quantização não-uniforme para diferentes valores de μ

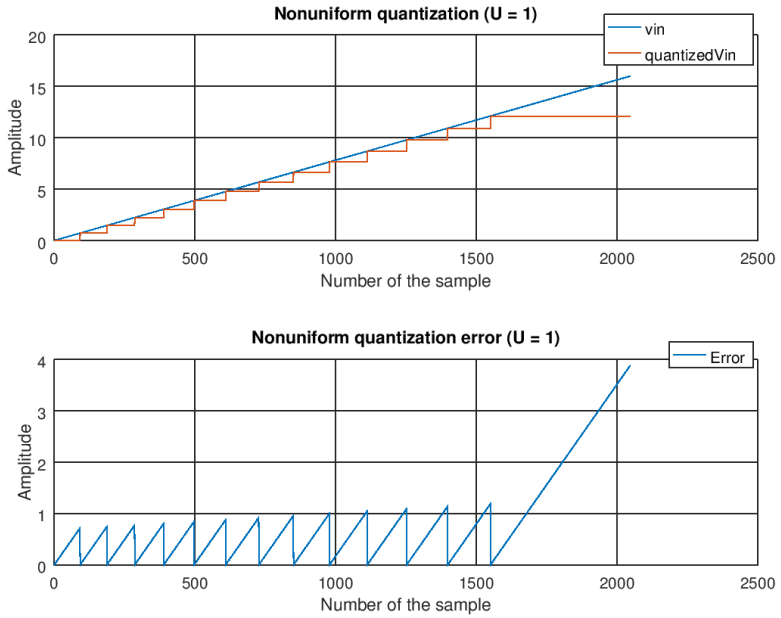

Figura A.1: Quantização não-uniforme e seu erro ($\mu = 1$).

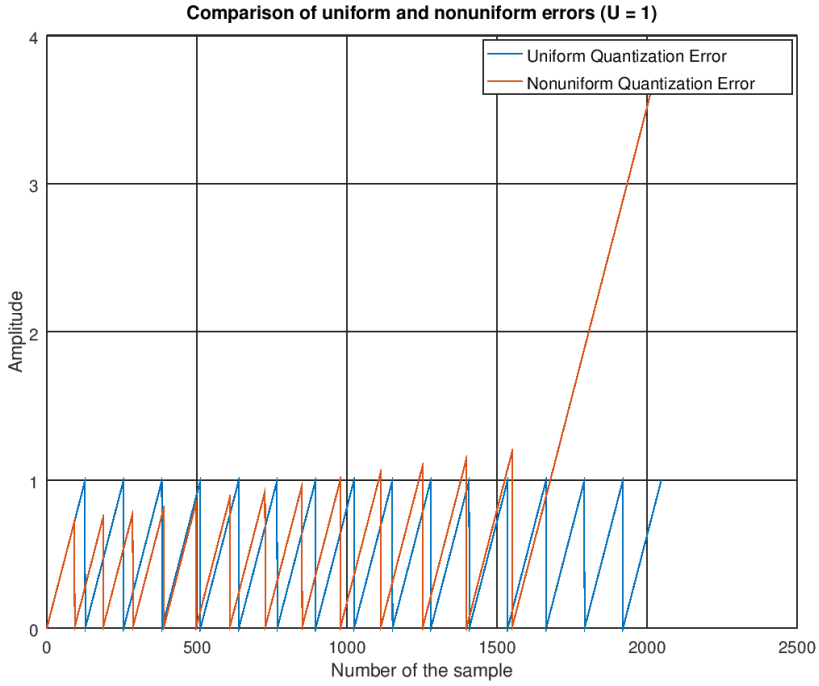

Figura A.2: Comparação dos erros das quantizações uniforme e não-uniforme ($\mu = 1$).

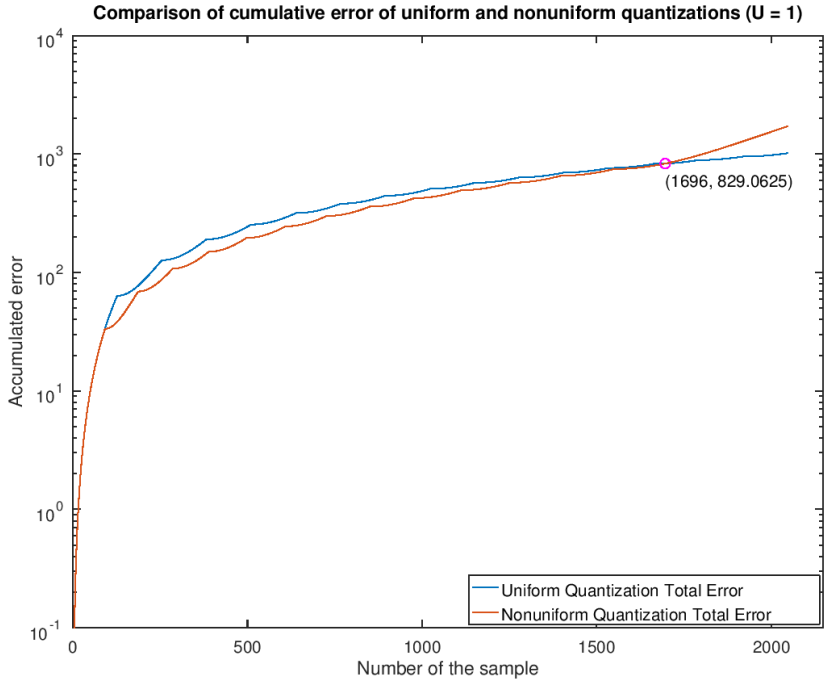

Figura A.3: Comparação dos erros acumulados das quantizações uniforme e não-uniforme ($\mu = 1$).

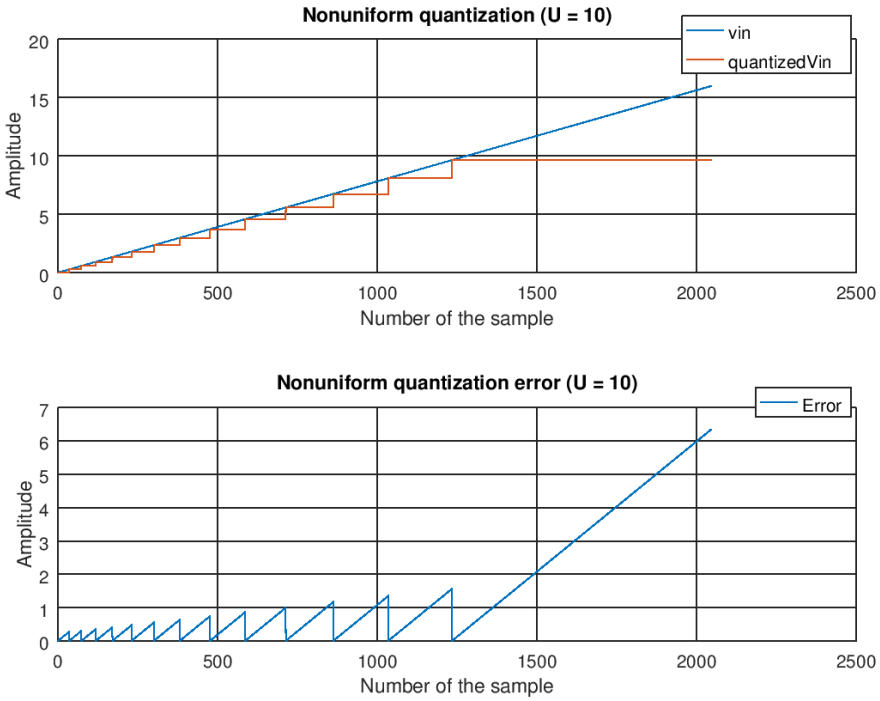

Figura A.4: Quantização não-uniforme e seu erro ($\mu = 10$).

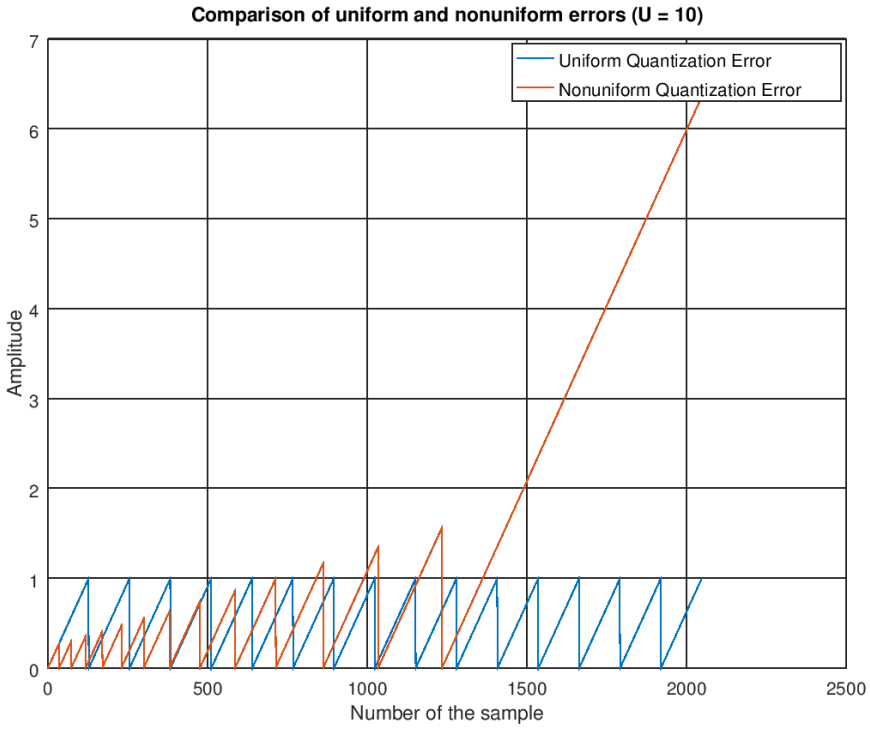

Figura A.5: Comparação dos erros das quantizações uniforme e não-uniforme ($\mu = 10$).

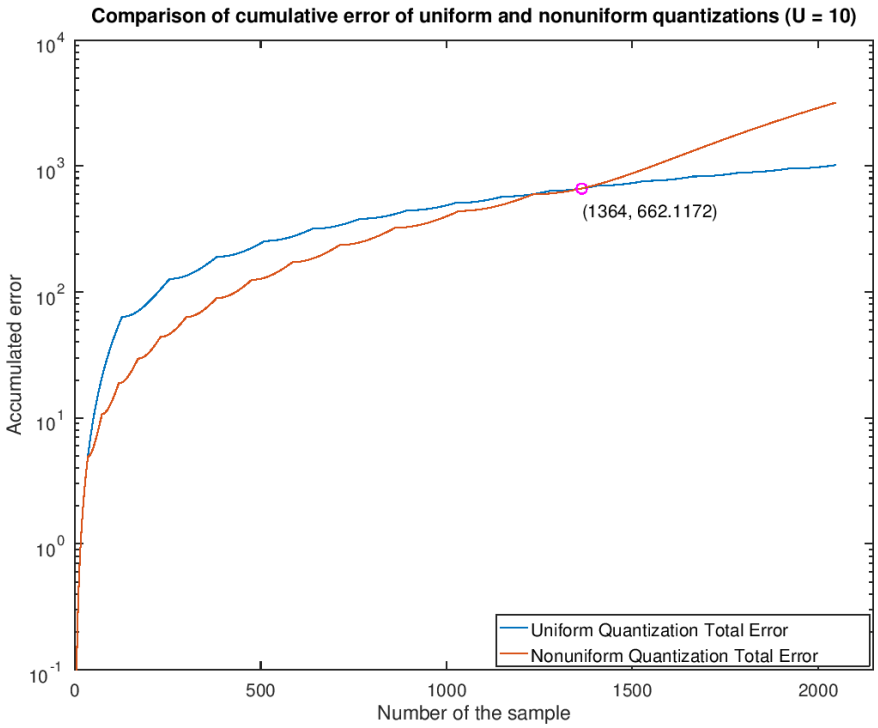

Figura A.6: Comparação dos erros acumulados das quantizações uniforme e não-uniforme ($\mu = 10$).

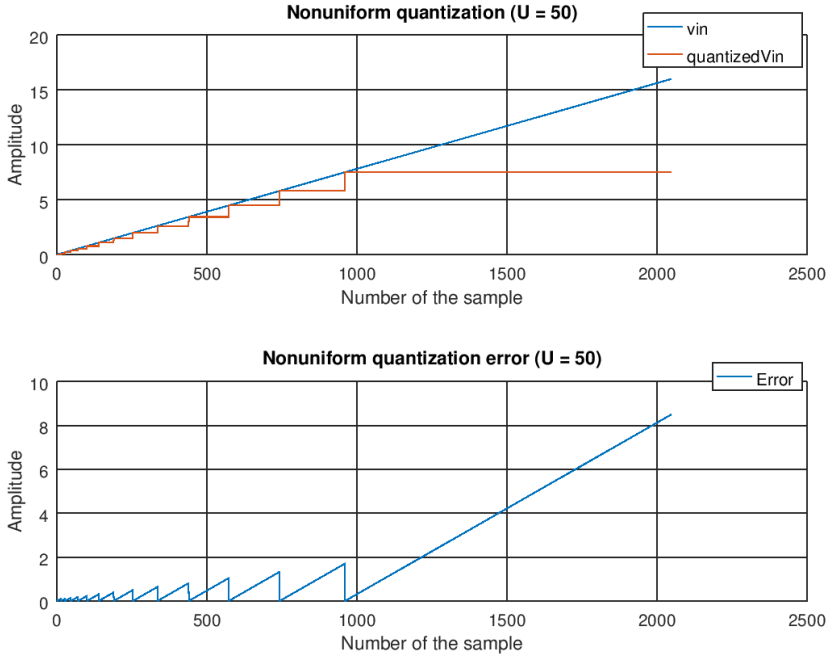

Figura A.7: Quantização não-uniforme e seu erro ($\mu = 50$).

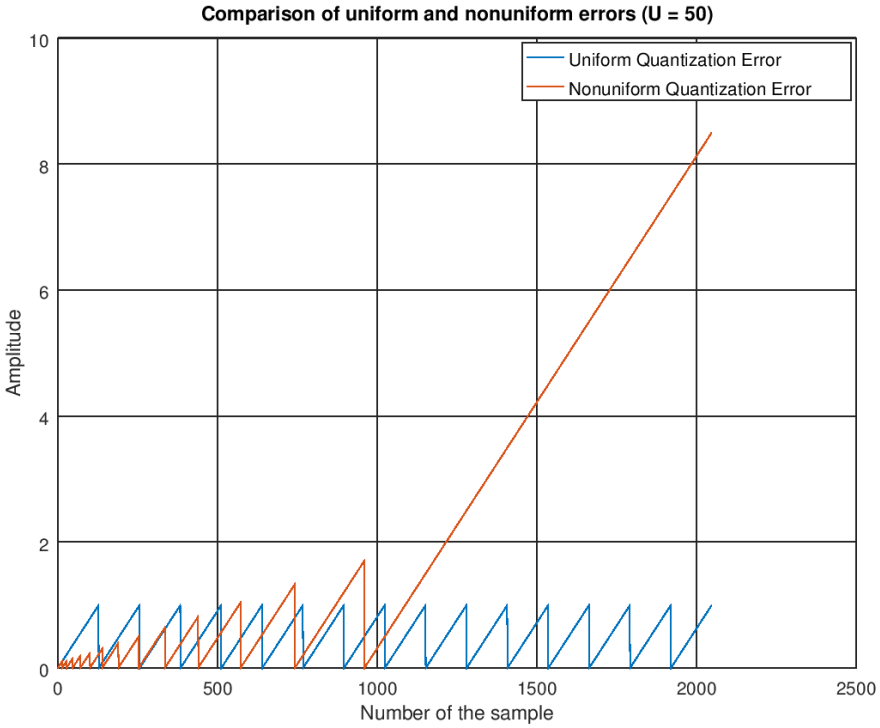

Figura A.8: Comparação dos erros das quantizações uniforme e não-uniforme ($\mu = 50$).

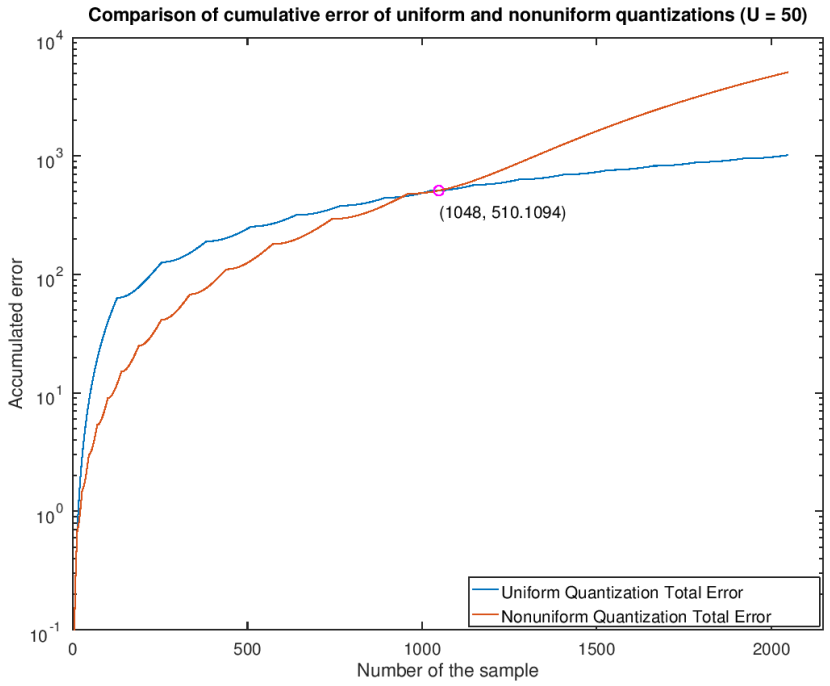

Figura A.9: Comparação dos erros acumulados das quantizações uniforme e não-uniforme ($\mu = 50$).

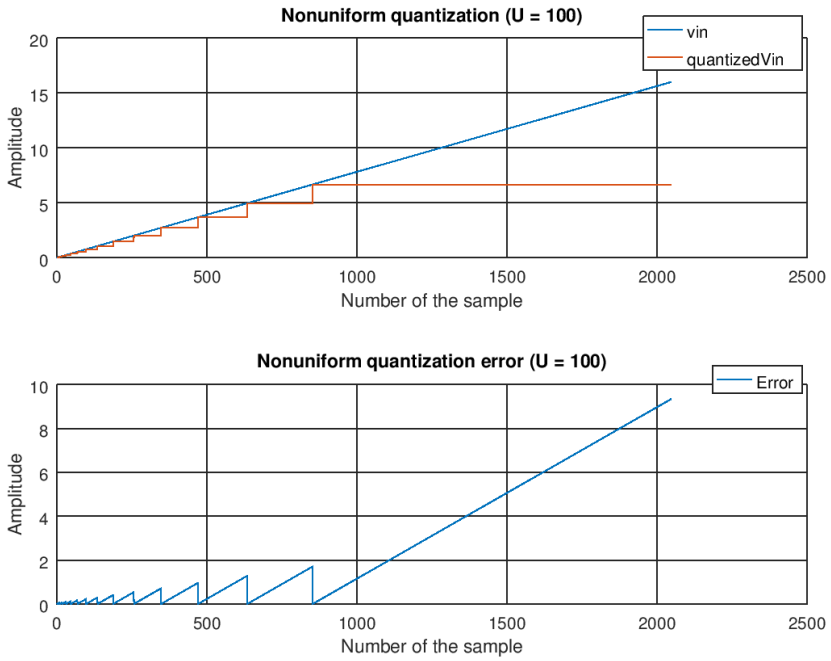

Figura A.10: Quantização não-uniforme e seu erro ($\mu = 100$).

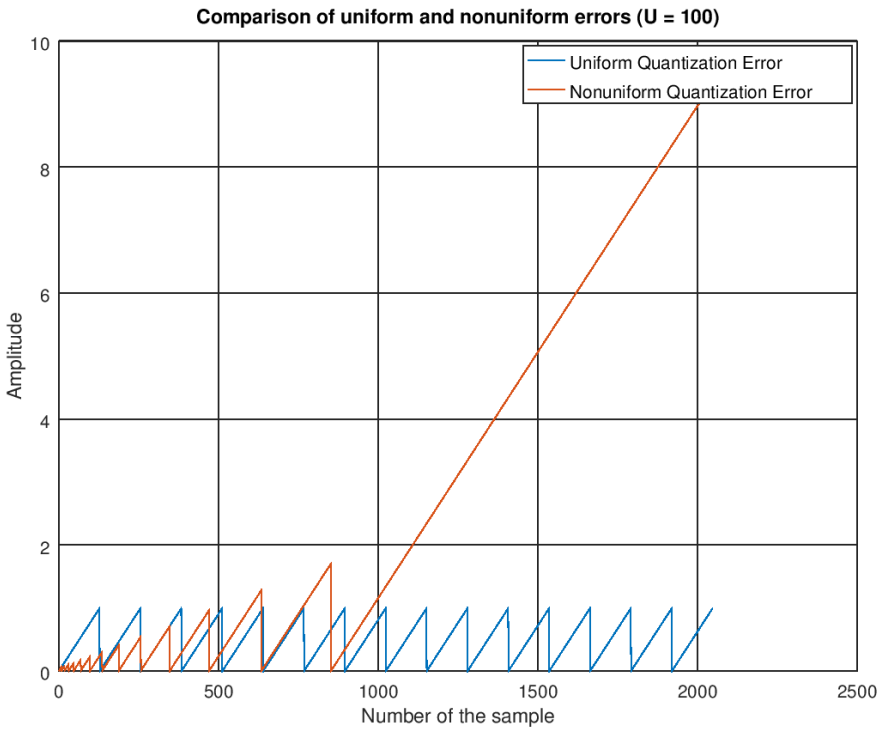

Figura A.11: Comparação dos erros das quantizações uniforme e não-uniforme ($\mu = 100$).

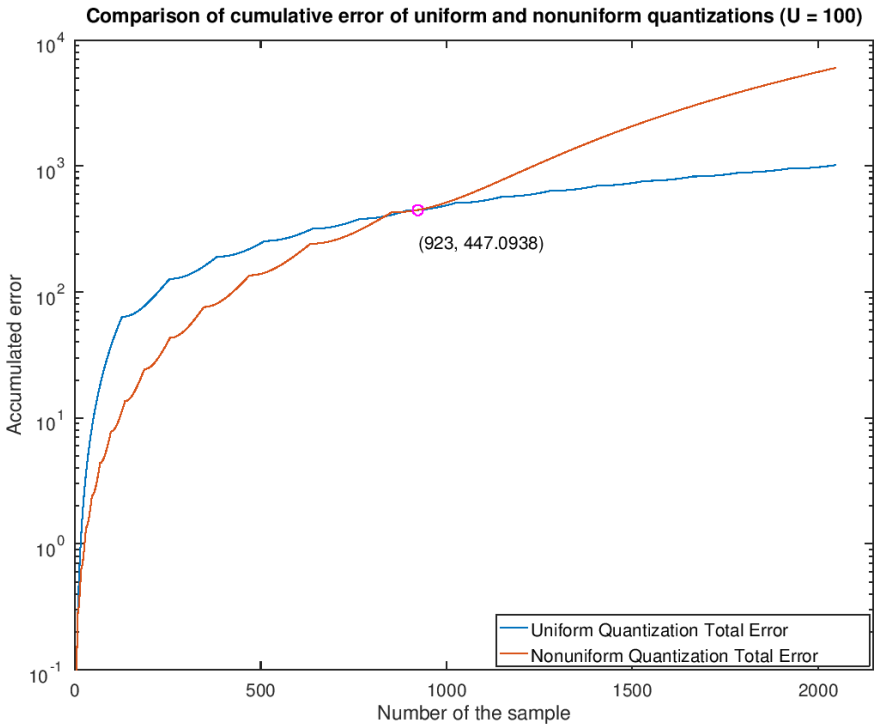

Figura A.12: Comparação dos erros acumulados das quantizações uniforme e não-uniforme ($\mu = 100$).

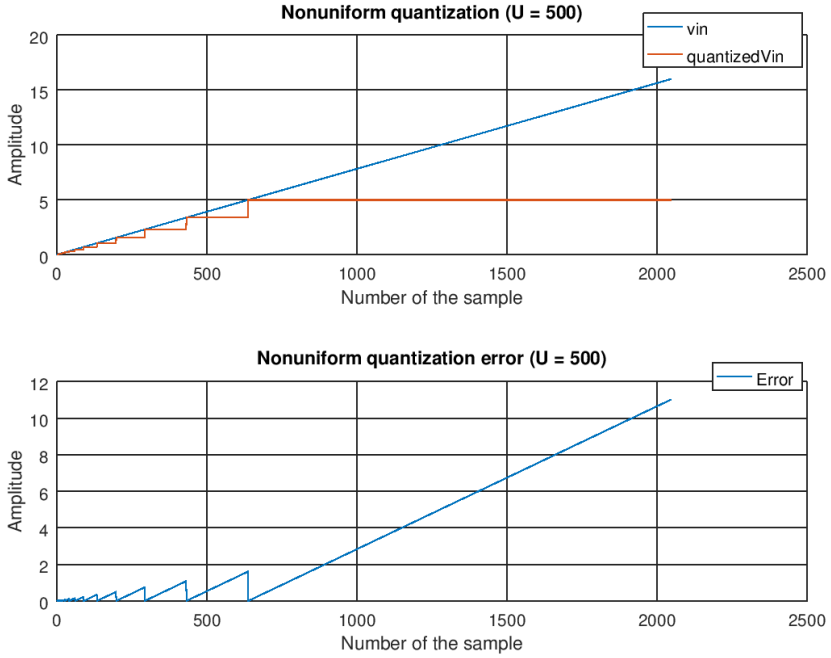

Figura A.13: Quantização não-uniforme e seu erro ($\mu = 500$).

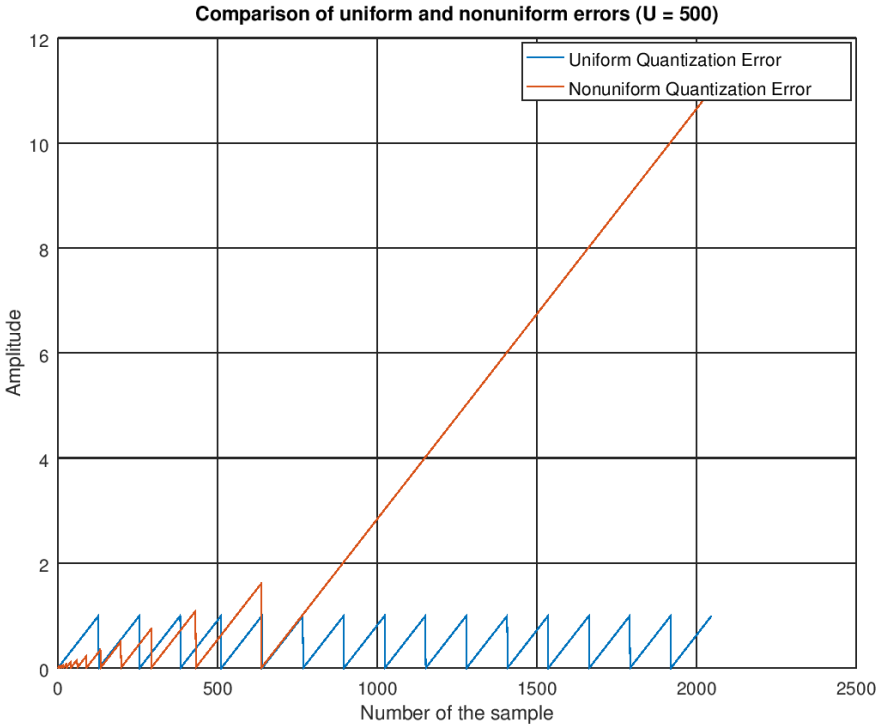

Figura A.14: Comparação dos erros das quantizações uniforme e não-uniforme ($\mu = 500$).

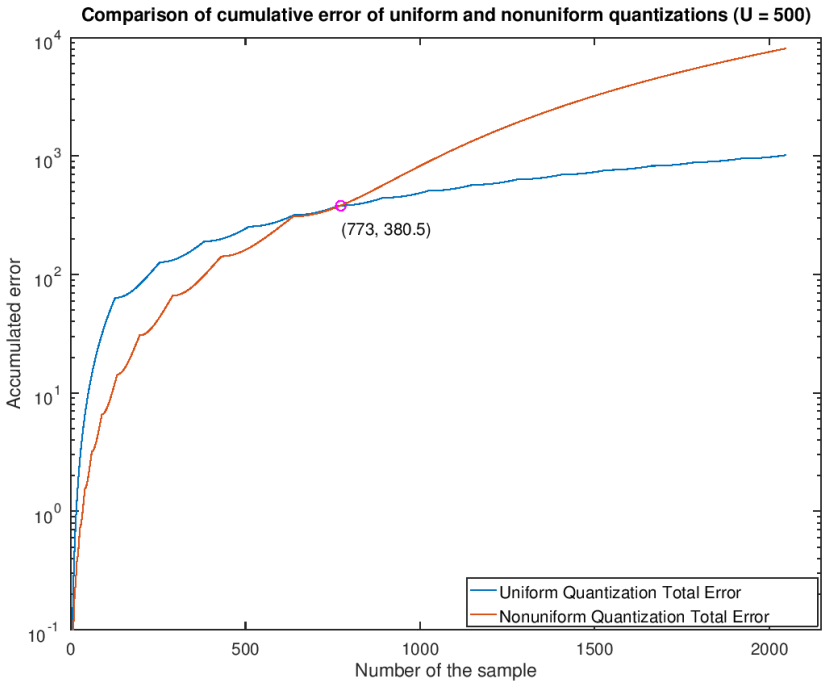

Figura A.15: Comparação dos erros acumulados das quantizações uniforme e não-uniforme ($\mu = 500$).

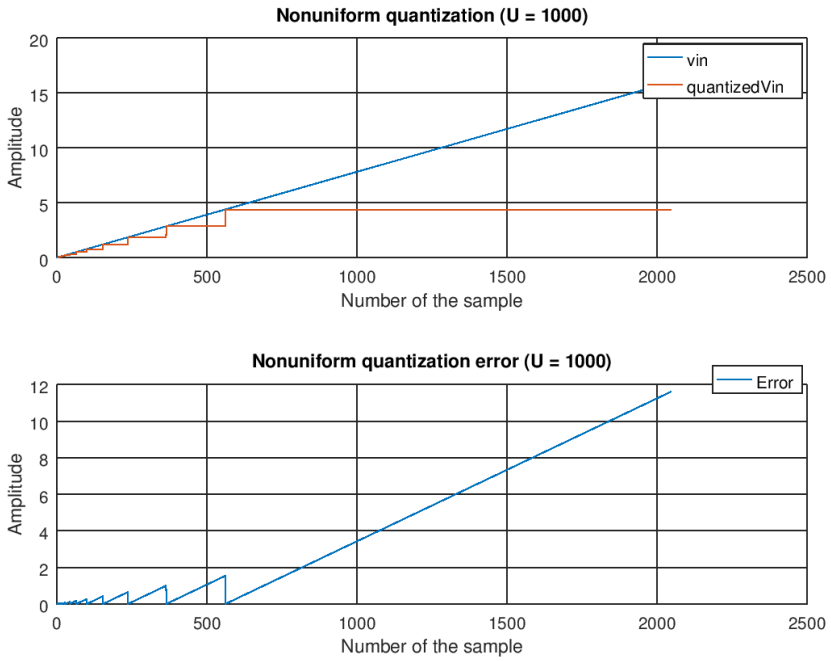

Figura A.16: Quantização não-uniforme e seu erro ($\mu = 1000$).

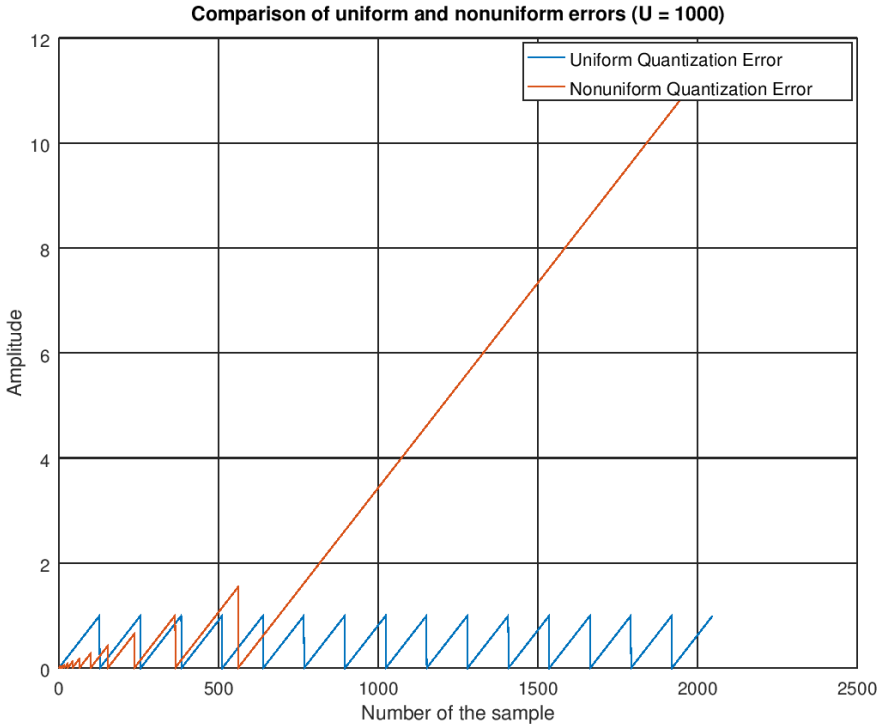

Figura A.17: Comparação dos erros das quantizações uniforme e não-uniforme ($\mu = 1000$).

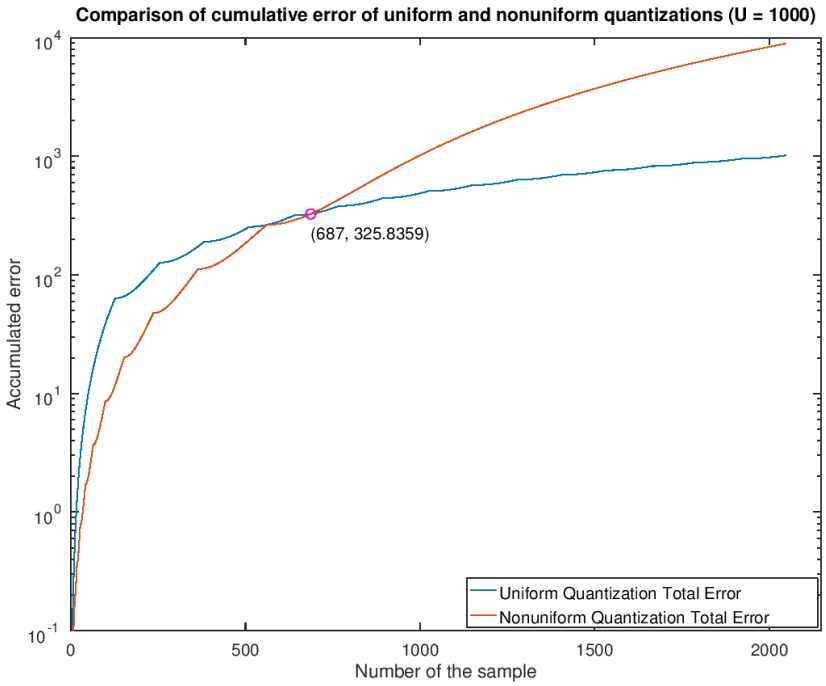

Figura A.18: Comparação dos erros acumulados das quantizações uniforme e não-uniforme ($\mu = 1000$).

Anexo I

Conversor A/D em Verilog-AMS

I.1 Modelo Verilog-A

O seguinte código de conversor A/D é apresentado em Kundert [1] como base de aprendizado da linguagem Verilog-AMS.

```
1  // N-bit Analog to Digital Converter
2  `include ''disciplines.vams''
3
4  module adc (out, in, clk);
5    parameter integer bits = 8 from [1:24];       // resolution (
       bits)
6    parameter real fullscale = 1.0;       // input range is from 0
       to fullscale (V)
7    parameter real td = 0;                        // delay from
       clock edge to output (s)
8    parameter real tt = 0;                        // transition
       time of output (s)
9    parameter real vdd = 5.0;             // voltage level of logic
        1 (V)
10   parameter real thresh = vdd/2;        // logic threshold level
       (V)
11   parameter integer dir = 1 from [-1 : 1] exclude 0;     // 1 for
       rising edges, -1 for falling
12
13   input in, clk;
14   output [0:bits-1] out;
15   voltage in, clk;
16   voltage [0:bits-1] out;
```

```
17    real sample, midpoint;
18    integer result[0:bits-1];
19    genvar i;
20
21    analog begin
22       @(cross(V(clk)-thresh, +1) or initial_step) begin
23          sample = V(in);
24          midpoint = fullscale/2.0;
25          for (i = bits - 1; i >= 0; i = i - 1 ) begin
26             if (sample > midpoint) begin
27                result[i] = vdd;
28                sample = sample - midpoint;
29             end else begin
30                result[i] = 0.0;
31             end
32             sample = 2.0*sample;
33          end
34       end
35
36       for (i = 0; i < bits; i = i + 1) begin
37         V(out[i]) <+ transition(result[i], td, tt);
38       end
39    end
40 endmodule
```