UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

ESCOLA DE ENGENHARIA

DEPARTAMENTO DE ENGENHARIA ELÉTRICA

NIGEL DOS SANTOS FOSTER

RECONFIGURAÇÃO DINÂMICA DE DISPOSITIVOS ANALÓGICOS PROGRAMÁVEIS: CONCEITOS E APLICAÇÕES

Porto Alegre

2014

NIGEL DOS SANTOS FOSTER

RECONFIGURAÇÃO DINÂMICA DE DISPOSITIVOS ANALÓGICOS PROGRAMÁVEIS: CONCEITOS E APLICAÇÕES

Projeto de Diplomação apresentado ao Departamento de Engenharia Elétrica da Universidade Federal do Rio Grande do Sul, como parte dos requisitos para a Graduação em Engenharia Elétrica.

Orientador: Prof. Dr. Tiago Roberto Balen

Porto Alegre

2014

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

ESCOLA DE ENGENHARIA

DEPARTAMENTO DE ENGENHARIA ELÉTRICA

NIGEL DOS SANTOS FOSTER

RECONFIGURAÇÃO DINÂMICA DE DISPOSITIVOS ANALÓGICOS PROGRAMÁVEIS: CONCEITOS E APLICAÇÕES

Este projeto foi julgado adequado para fazer jus aos créditos da Disciplina de "Projeto de Diplomação" do Departamento de Engenharia Elétrica e aprovado em sua forma final pelo Orientador e pela Banca Examinadora.

Orientador: _____
Prof. Dr. Tiago Roberto Balen, UFRGS
Doutor pela Universidade Federal do Rio Grande do Sul – Porto Alegre, Brasil

Banca Examinadora:

Prof. Dr. Tiago Roberto Balen, UFRGS
Doutor pela Universidade Federal do Rio Grande do Sul – Porto Alegre, Brasil

Prof. Dr. Marcelo Götz, UFRGS
Doutor pela Universität Paderborn – Paderborn, Alemanha

Prof. Dr. Hamilton Duarte Klimach, UFRGS
Doutor pela Universidade Federal de Santa Catarina – Florianópolis, Brasil

Porto Alegre

2014

DEDICATÓRIA

Dedico este trabalho aos meus pais, Jorge e Marcia, e ao meu irmão Alan. O apoio incondicional durante esta longa jornada foi fundamental para a conclusão desta etapa. Compartilho com eles a conquista do título que postulo com este trabalho.

AGRADECIMENTOS

Agradeço aos meus familiares e amigos pelo apoio e compreensão durante esta longa jornada.

Agradeço aos colegas de graduação pelos momentos de descontração e estudos.

Agradeço à UFRGS e aos professores da graduação pelos ensinamentos e oportunidades oferecidas.

Agradeço ao Prof. Dr. Tiago Roberto Balen pela orientação neste trabalho.

Agradeço à ThyssenKrupp Elevadores S/A e aos colegas pela experiência proporcionada e apoio no meu desenvolvimento profissional.

Agradeço aos membros da banca avaliadora por aceitarem o convite.

RESUMO

Os Dispositivos Analógicos Programáveis, mais conhecidos como *Field Programmable Analog Arrays*, FPAAs, são circuitos integrados tipicamente baseados em amplificadores operacionais (ou outro dispositivo analógico similar) com alguns recursos computacionais incorporados, os quais possibilitam um novo método para prototipagem de sistemas analógicos. Este trabalho visa o estudo de tais dispositivos, abordando conceitos, especificações, possibilidades de utilização e com ênfase em uma nova possibilidade para circuitos analógicos: a reconfiguração dinâmica. Para demonstrar o funcionamento desta técnica, o trabalho aborda a implementação de um sistema de controle analógico com reconfiguração dinâmica utilizando o FPAA AN221E04 da Anadigm Company.

Palavras-chave: FPAA, reconfiguração dinâmica, sistema de controle analógico com reconfiguração dinâmica, AN221E04.

ABSTRACT

The *Field Programmable Analog Array*, FPAAs, are integrated circuits based on operational amplifiers (or another similar analog device) with some computer resources incorporated, which enable a new method for analog system prototyping. This paper aims the study of these devices, addressing concepts, specifications, potential uses and emphasizing a new possibility for analog circuits: dynamic reconfiguration. To show this technique, the paper discusses the implementation of an analog control system with dynamic reconfiguration using the FPAA AN221E04 from Anadigm Company.

Keywords: FPAA, dynamic reconfiguration, analog system control with dynamic reconfiguration, AN221E04.

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS

ABK: Anadigm Boot Kernel
ADC: Analogic to Digital Converter
ASCII: American Standard Code for Information Interchange
ASIC: Application Specific Integrated Circuit
CAB: Configurable Analog Block
CAM: Configurable Analog Modules
CMOS: Complementary Metal Oxide Semiconductor
COM port: Communication Port
CRC: Cyclic Redundancy Check
DC: Direct Current
DSP: Digital Signal Processor
EDA: Electronic Design Automation
EEPROM: Electrical Eraseble Programmable Read Only Memory
FPAA: Field Programmable Analog Array
FPGA: Field Programmable Gate Array
I/O: Input/Output
IA: Input Amplifiers
ispPAC: In System Programmable Analog Circuits
IMP: International Microelectronics Products
MOSFET: Metal Oxide Semiconductor Field Effect Transistor
NOL Clock Generator: Non-Overlapping Clock Generator
PLD: Programmable Logic Device
PROM: Programmable Read Only Memory
PSoC: Programmable System On Chip
SAR: Successive Approximation Register
SNR: Signal Noise Ratio
SPI: Serial Peripheral Interface
SRAM - Static Random Access Memory.SoC: System On Chip
SSI: Synchronous Serial Interface (SSI)
THD: Total Harmonic Distortion
TTL: Transistor – Transistor Logic
VMR: Voltage Main Reference

1. Introdução

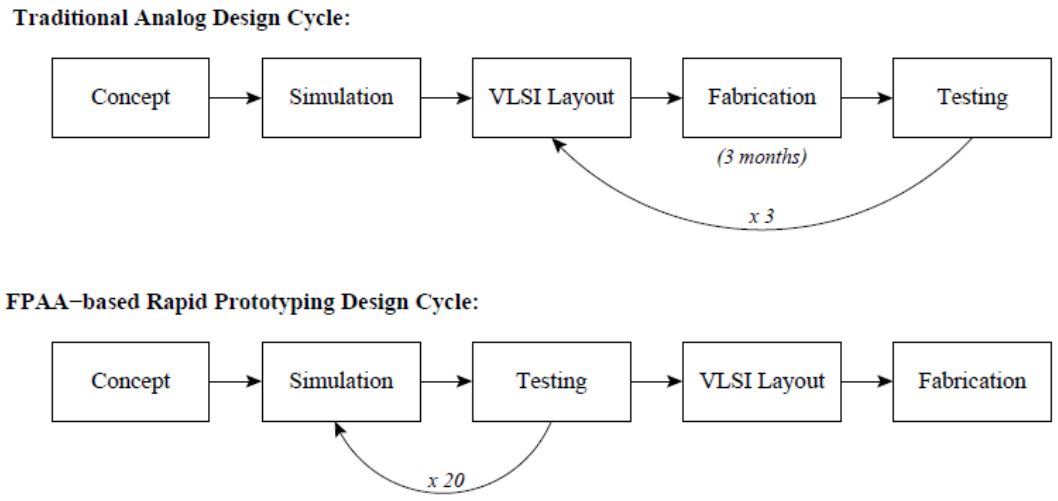
1.1. Motivação

Precisão, baixo custo e técnicas de rápida prototipagem para circuitos analógicos tem sido um desenvolvimento muito aguardado por desenvolvedores de circuitos analógicos (GULAK, 1995).

No domínio digital, os *Programmable Logic Devices*, PLDs, surgiram entre o final dos anos 1960 e início dos anos 1970, provocando um forte impacto no desenvolvimento de circuitos digitais, os quais acabaram evoluindo e dando origem aos hoje conhecidos e amplamente utilizados *Field Programmable Gate Array*, FPGA. No domínio analógico, o progresso foi mais lento e apenas no final dos anos 1980 surgiram os primeiros *Field Programmable Analog Array*, FPAA, disponíveis apenas para fins acadêmicos. Somente em 1996, surgiram os primeiros FPAAs comerciais.

Contudo, o surgimento dos FPAAs revolucionou a maneira de se projetar circuitos analógicos. A Figura 1 mostra um comparativo entre o método tradicional de *design* de um *chip* analógico e o método que utiliza um FPAA para o desenvolvimento.

Figura 1: Ciclo de Design Analógico Tradicional x FPAA



FONTE: (HALL, 2004)

Além de reduzir o tempo de projeto, os FPAAs permitiram o desenvolvimento de um novo conceito em circuitos analógicos, a reconfiguração dinâmica. A nova técnica de reconfiguração dinâmica dos *Field Programmable Analog Array* melhora a maneira de implementação de circuitos analógicos (LITA et al.,2009).

1.2. Objetivo

Como objetivo principal, este trabalho apresenta o estudo da reconfiguração dinâmica de dispositivos analógicos programáveis. Este trabalho também realiza uma abordagem técnica dos *Field Programmable Analog Array*, FPAA, para que se possa compreender o funcionamento do dispositivo e quais são as características necessárias para a implementação da técnica de reconfiguração dinâmica.

Também faz parte do objetivo deste trabalho a implementação de um sistema de controle que utilize o conceito de reconfiguração dinâmica. Com isso, realizou-se um estudo do FPAA AN221E04 da Anadigm Company, que foi o dispositivo utilizado para desenvolvimento de um sistema de controle analógico.

Como objetivo complementar, este trabalho conta com uma pequena demonstração do *software* Anadigm Designer 2, necessário para a configuração dos FPAA's fornecidos pela Anadigm Company.

1.3. Organização do Trabalho

Os capítulos a seguir contemplam o desenvolvimento realizado durante este trabalho, abordando os conceitos envolvidos, descrevendo as etapas de desenvolvimento e por último, analisando os resultados obtidos.

O trabalho está distribuído da seguinte forma:

- Capítulo 2 apresenta uma introdução ao *Field Programmable Analog Array*, FPAA, abordando a evolução ao longo dos anos, além de abordar os diferentes tipos destes dispositivos e suas vantagens e desvantagens na utilização.

- Capítulo 3 aborda o dispositivo AN221E04, utilizando neste trabalho, além de abordar o *software* AnadigmDesigner2.
- Capítulo 4 aborda o conceito de reconfiguração dinâmica aplicado aos FPAA's, explicando o funcionamento e os requisitos para utilização desta técnica.
- Capítulo 5 demonstra a implementação de dois sistemas: um circuito amplificador não inversor e um sistema de controle analógico utilizando a técnica de reconfiguração dinâmica. Dentro deste capítulo está contemplado o funcionamento, implementação, código utilizado, ensaios e resultados obtidos.
- Capítulo 6 apresenta as conclusões obtidas com a realização da implementação dos sistemas, além de descrever sugestões de melhoria e propostas para trabalhos futuros.

2. Dispositivos Analógicos Programáveis

Os FPAAs são circuitos analógicos programáveis que podem ser reconfigurados durante as etapas de desenvolvimento de um projeto bem como em campo, durante a utilização do sistema no qual o componente se encontra (BALEN, 2006).

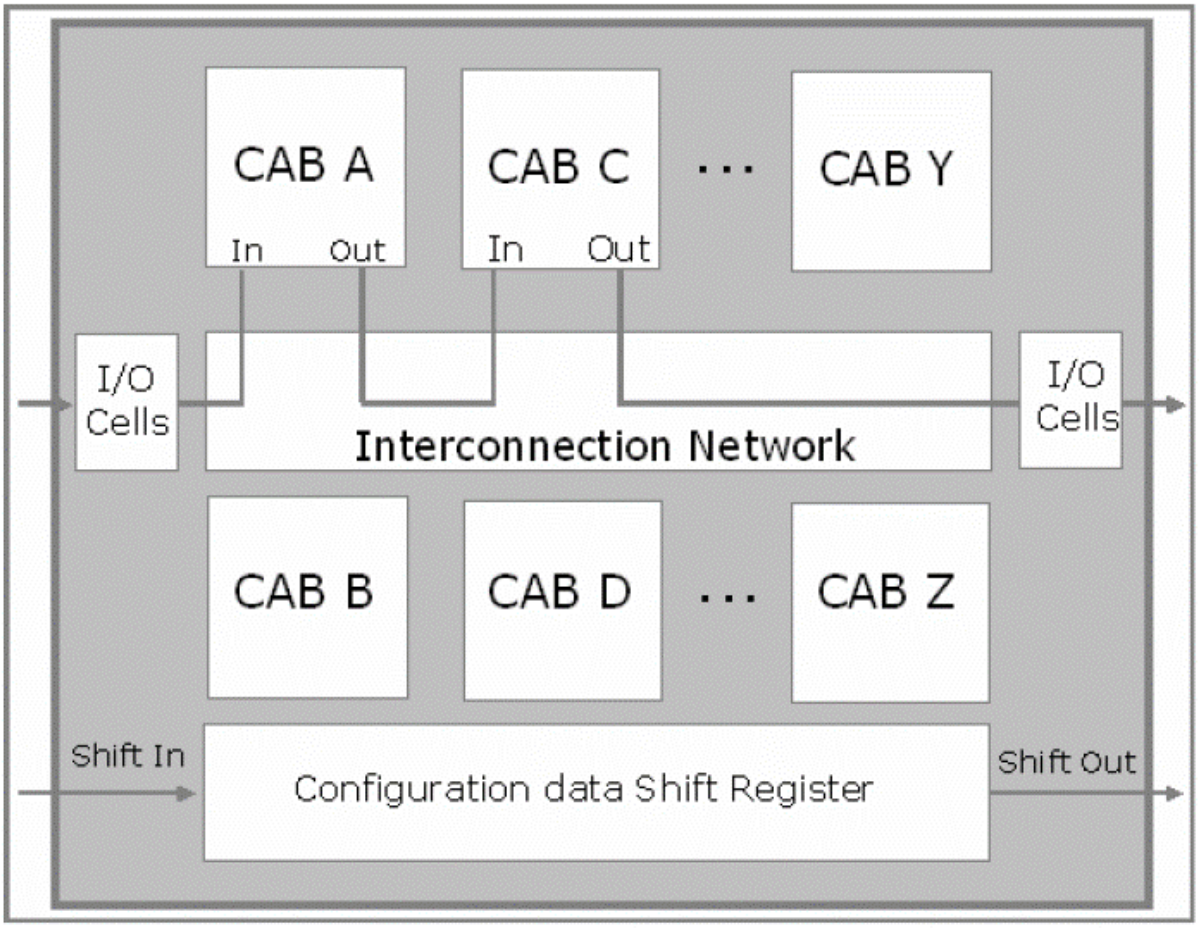
O processo de projeto, fabricação e teste de um circuito analógico requer profissionais qualificados e experientes, além de ser um processo longo e com alto custo. Redução do tempo e recursos necessários foram os principais motivadores para a concepção dos FPAAs, permitindo uma redução no ciclo de concepção dos circuitos, através da possibilidade de rápida prototipagem e a não necessidade de um processo de fabricação para testes.

Apesar de todas as vantagens, um projeto implementado através de um FPAA resulta em um dispositivo com um maior nível de ruído/interferências bem como uma maior possibilidade de falha, portanto, o desenvolvimento destes dispositivos possui sempre algumas ineficiências, isto é, menor largura de banda e maior consumo de energia (SANTOS, 2010).

Contudo, essas características adversas acabam por ser equilibradas devido a facilidade de manipulação das ferramentas de desenvolvimento, não exigindo profissionais altamente qualificados, pela redução do *time-to-market* (tempo de colocação do produto no mercado), sem afetar o nível de demanda e oferta do mesmo, e a possibilidade de atualização dos produtos após a implementação.

Um FPAA genérico possui uma arquitetura típica como ilustra a Figura 2, formada por blocos analógicos programáveis, CABs – *Configurable Analog Blocks*, células de entrada e saída, I/O, uma rede de interconexões e registradores de memória, onde os dados digitais são armazenados para a programação dos componentes. A conexão entre os diferentes CABs e as células de I/O são realizadas pela rede de interconexões, enquanto as células de I/O são responsáveis pela interface do FPAA com o sistema externo, podendo ser compostas por *buffers*, filtros *anti-aliasing* ou *smoothing*, entre outras funções de condicionamento de sinal.

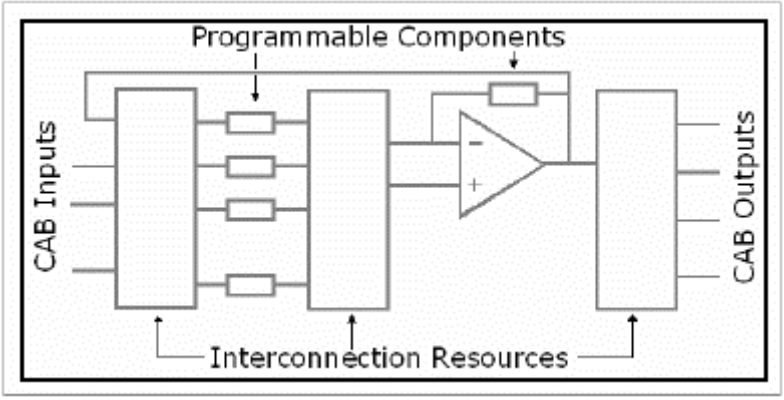
Figura 2: Arquitetura de um FPAA genérico



FONTE: (BALEN, 2006)

A maior parte do processamento do sinal analógico ocorre internamente aos CABs (DAMIANI, 2010). São compostos por um conjunto de componentes analógicos programáveis, blocos de interconexões e um amplificador operacional de saída. Os seus componentes podem ser configurados como resistores, capacitores ou até mesmo uma simples linha de interconexão, um fio, por exemplo. Os parâmetros programáveis de um CAB, em geral, costumam ser ganho de amplificadores, valores de resistores e capacitores e a habilitação de laços de realimentação. A Figura 3 ilustra a representação genérica da arquitetura de um CAB.

Figura 3: Arquitetura de um CAB genérico



FONTE: (BALEN, 2006)

2.1. História

Os primeiros FPAAs desenvolvidos tinham fins acadêmicos e começaram a surgir apenas no final dos anos de 1980. Somente a partir de 1996 que as primeiras ofertas comerciais começaram a surgir.

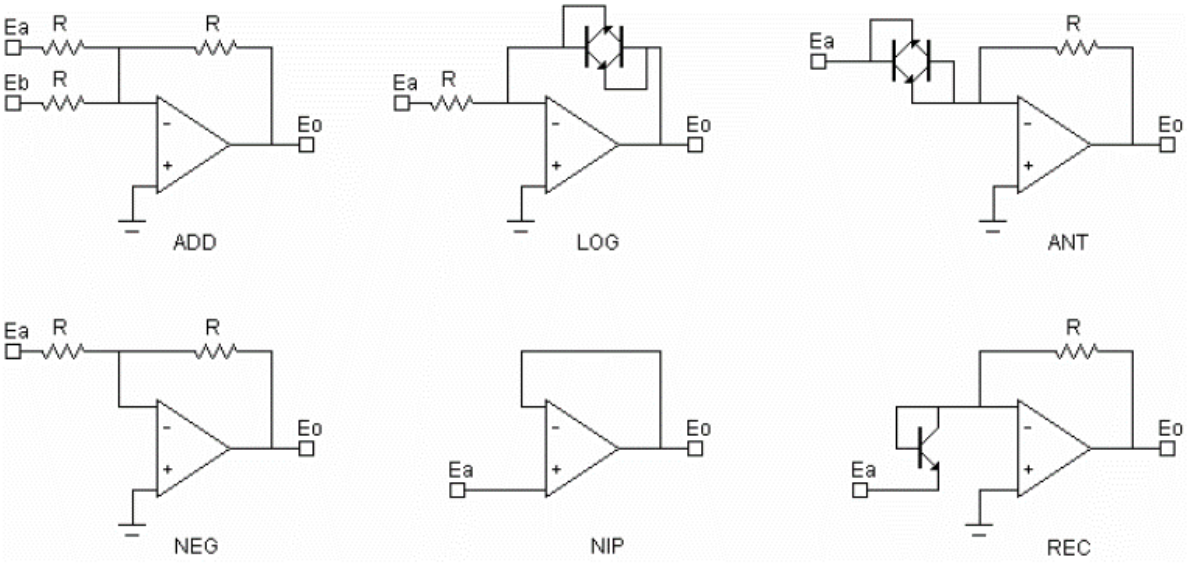
O desenvolvimento de circuitos integrados ASIC, *Application Specific Integrated Circuit*, apenas eram rentáveis para vendas em grande escala, o que fez com que o desenvolvimento dos dispositivos analógicos reconfiguráveis começassem a ter algum peso na indústria eletrônica (SANTOS, 2010).

Assim, diversos fornecedores voltaram-se para este novo mercado de dispositivos analógicos programáveis. A empresa Zetex foi a pioneira, lançando o seu TRAC, *Tottaly Reconfigurable Analog Circuit*, em 1996.

O TRAC é um FPAA de tempo contínuo, formado por 20 CABs com possibilidade de serem interligados localmente (internamente) ou de modo global (externamente). O CAB do TRAC permite a ligação de transistores bipolares de maneira a implementar amplificadores logarítmicos. Os TRACs eram especializados em tratamento matemático de sinais analógicos, já que o mesmo permitia a implementação de funções como soma, inversão, logaritmo, anti-logaritmo, retificação e ganho. Através da associação de somadores a amplificadores log e anti-log permitia a implementação de multiplicação e divisão. A Figura 4 apresenta os blocos analógicos que podem ser implementados no TRAC020LH. A Figura 5

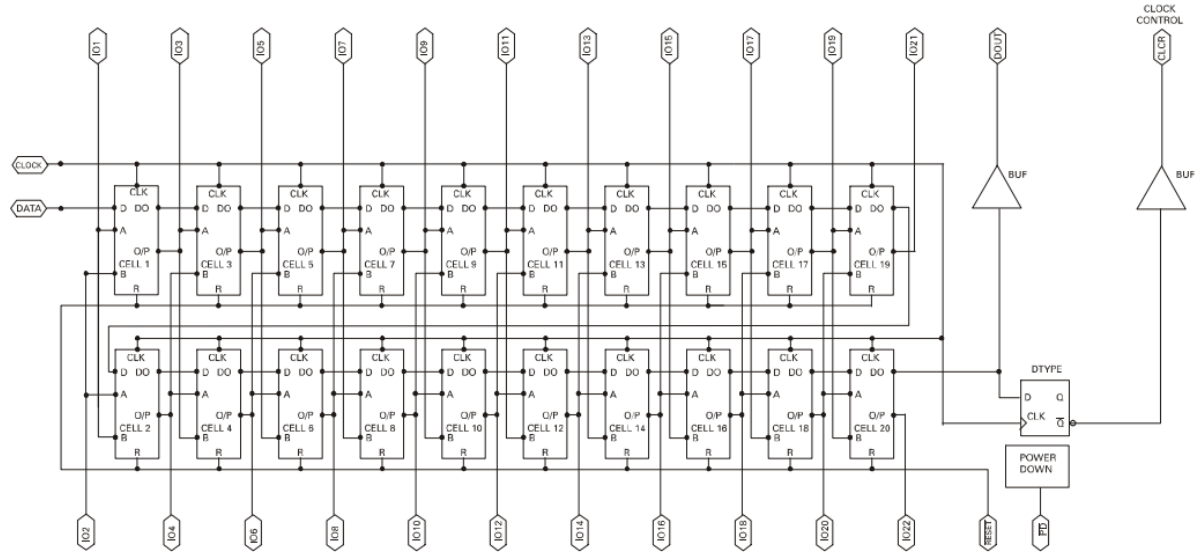
representa o diagrama de blocos do dispositivo. Atualmente o TRAC020LH não é mais comercializado.

Figura 4: Funções analógicas implementáveis no TRAC020LH



FONTE: (ZETEX, 1999)

Figura 5: Diagrama esquemático do TRAC020LH



FONTE: (ZETEX, 1999)

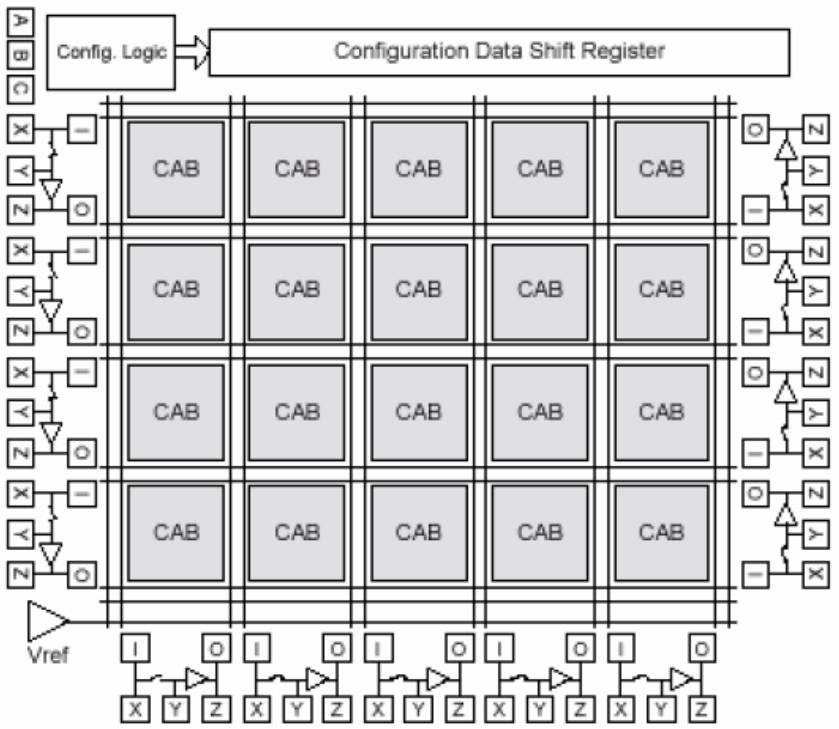
Ainda em 1996 a Pilkinton Microelectronics, empresa situada na Inglaterra, desenvolvia um FPAA baseado na tecnologia de capacitores chaveados,

denominado DPAD2 (BRATT; MACBETH, 1998). Em 1997 a Motorola fundou a *Motorola Programmable Technologies Center* após a compra da empresa inglesa, lançando o FPAA com o nome de MPAA020.

Em 2000, a Motorola vendeu a tecnologia para uma nova empresa, chamada de Anadigm, que relançou o mesmo FPAA agora com o nome de AN10E40, realizando aprimoramentos apenas no *software* de configuração do dispositivo, chamado de *AnadigmDesigner*.

O AN10E40 é composto de uma matriz de 4x5 CABs, 13 células de I/O e uma nova rede de intercomunicação composta por cinco linhas e seis colunas, possibilitando a interligação entre qualquer CAB, independente de sua posição, conforme ilustra a Figura 6. Por se tratar de uma FPAA baseado na tecnologia de capacitores chaveados, o sinal analógico precisa ser amostrado, onde a frequência de amostragem desse sinal depende do *clock* que o dispositivo trabalha. Como o AN10E40 trabalha com um *clock* máximo de 1 MHz, de acordo com o teorema de Nyquist da amostragem (HAYKIN; VAN VEEN, 2001) a largura de banda deste dispositivo se limita a 500 KHz.

Figura 6: Diagrama de blocos do AN10E40

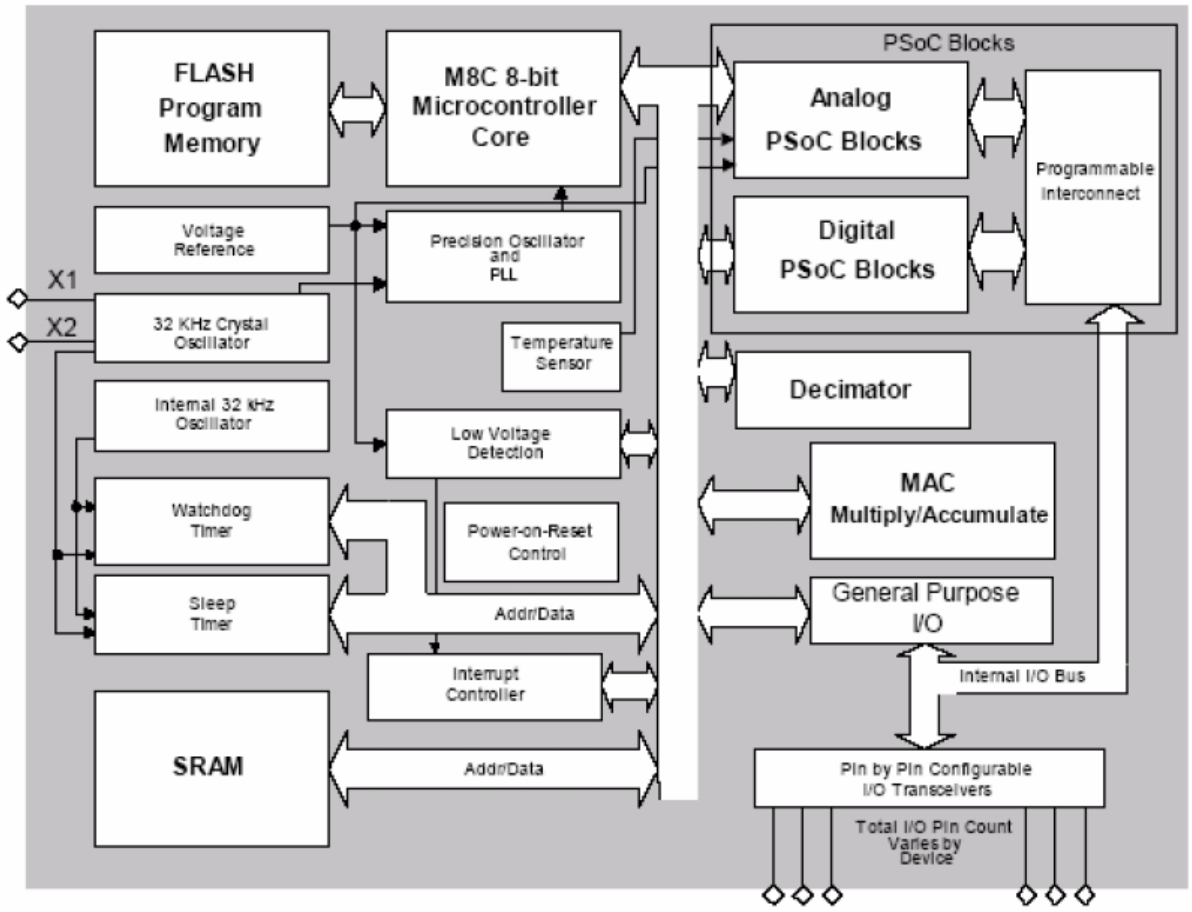


FONTE: (ANADIGM, 2003)

Atualmente, além da Anadigm, a Lattice Semiconductors e a Cypress são os principais fabricantes de FPAAs.

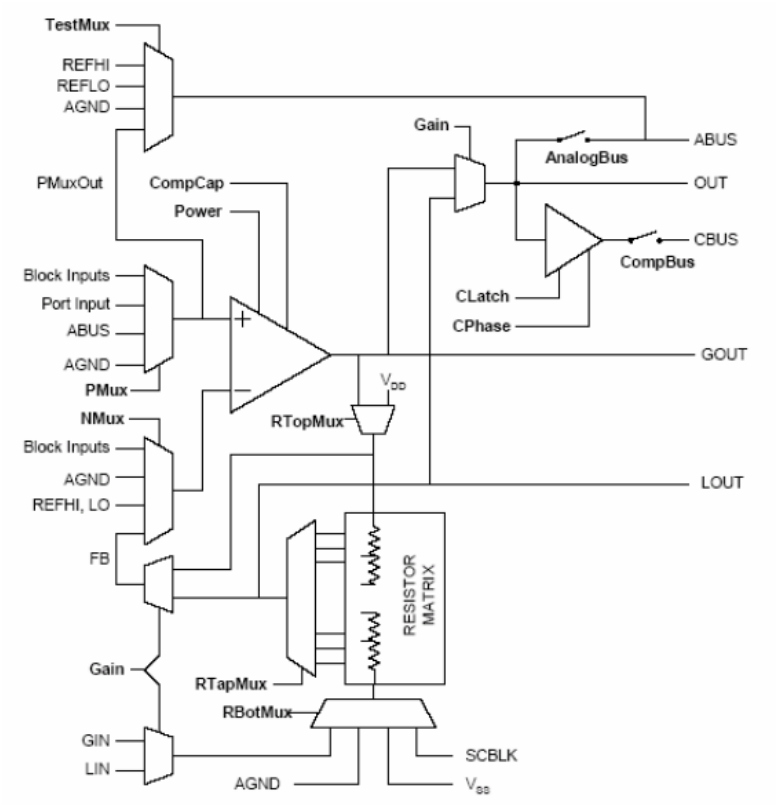
A Cypress possui a família CY8C2XXXX, baseada na tecnologia SoCs, *System On Chip*, referenciada como PSoC, *Programmable System On Chip*. Os PSoCs possuem blocos analógicos de tempo contínuo e a capacitores chaveados, além de elementos digitais e mistos, como microcontrolador, memórias e conversores de dados. A Figura 7 ilustra o diagrama de blocos do PSoC, enquanto as Figuras 8 e 9 apresentam, respectivamente, os blocos analógicos de tempo contínuo e a capacitor chaveado do PSoC.

Figura 7: Diagrama de blocos do PSoC



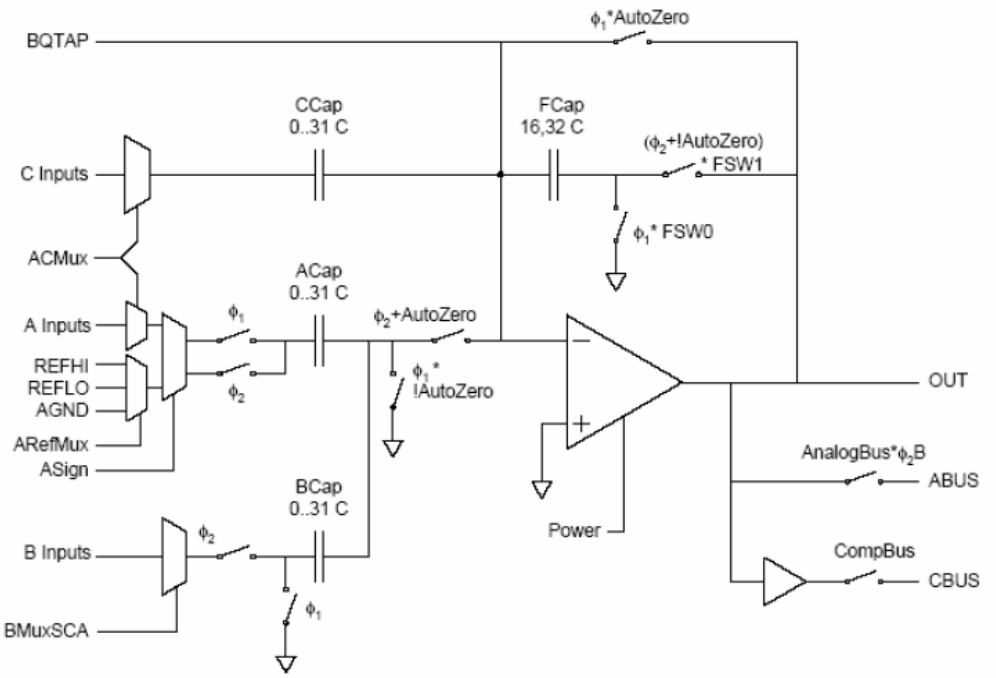
FONTE: (CYPRESS, 2002)

Figura 8: Esquemático da célula analógica de tempo contínuo do PSoC



FONTE: (CYPRESS, 2002)

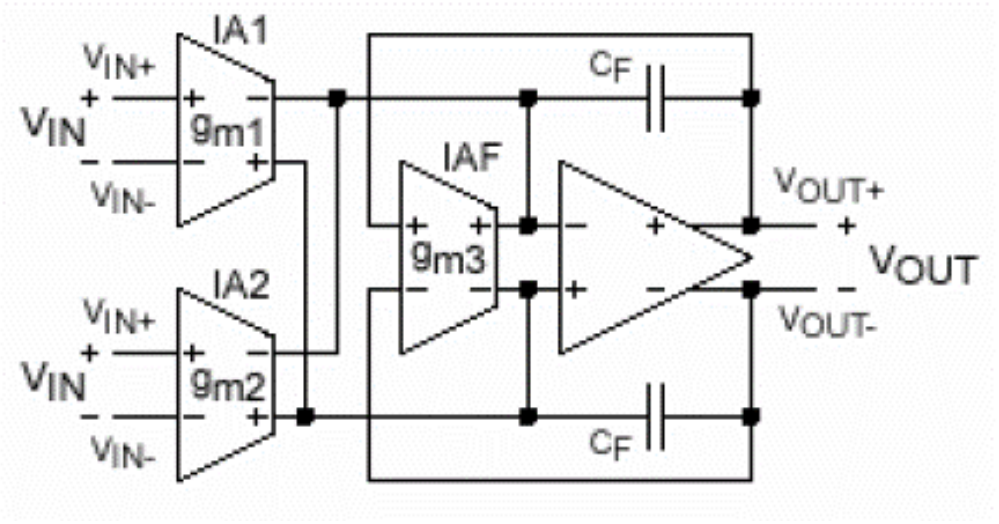
Figura 9: Esquemático da célula analógica a capacitores chaveados do PSoC



FONTE: (CYPRESS, 2002)

A Lattice Semiconductors possui a família ispPAC, *In System Programmable Analog Circuits*, com origem em uma estrutura desenvolvida pela empresa IMP, *International Microelectronics Products*, que atualmente pertence a Lattice. Uma das principais características dessa família é o fato de possuir memória de programação do tipo EEPROM, *Electrical Eraseble Programmable Read Only Memory*. A família ispPAC é composta por seis componentes, sendo alguns membros voltados para a realização de funções analógicas como filtros, somadores, integradores e amplificadores, enquanto os outros membros da família são voltados para a implementação de filtros programáveis. O número de CABs varia de acordo com o modelo da família, porém a sua composição é sempre a mesma, sendo baseado em transcondutores, conforme ilustra a Figura 10.

Figura 10: Esquemático do CAB dos componentes da família ispPAC

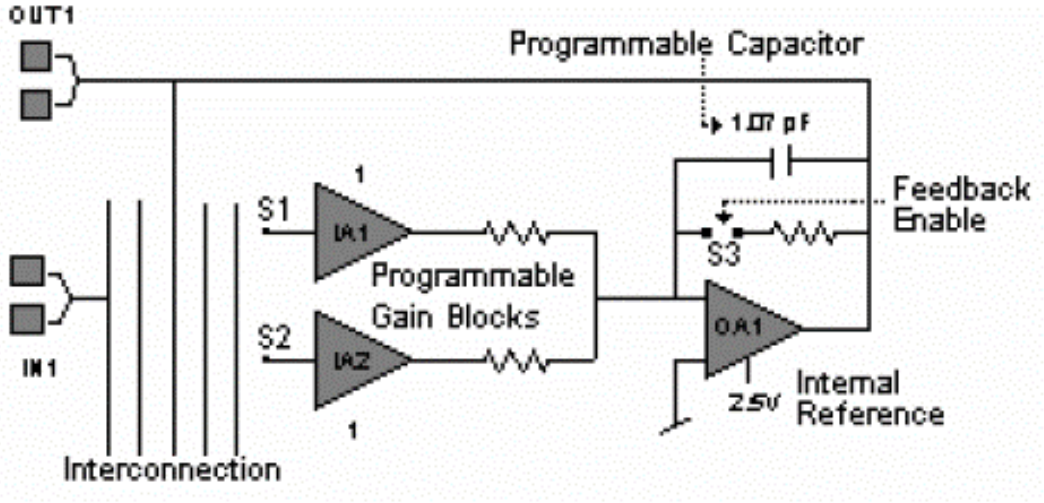


FONTE: (LATTICE, 2000)

O fabricante representa os transcondutores por amplificadores de entrada, IA – *Input Amplifiers*, e resistores, visando a facilitação de compreensão do circuito, conforme pode ser verificado na Figura 11.

A largura de banda dos dispositivos da família ispPAC é 650 KHz (LATTICE, 2000).

Figura 11: Representação do CAB dos componentes da família ispPAC



FONTE: (LATTICE, 2000)

2.2. Tipos de FPAA

Os FPAAs além de serem classificados como comerciais ou acadêmicos, possuem outro importante tipo de classificação já citado anteriormente, a técnica de *design* utilizada. Estas técnicas são classificadas em duas principais categorias: *continuos time* (tempo contínuo) e *discrete time* (tempo discreto).

Um FPAA de tempo contínuo normalmente utiliza transcondutores na sua fabricação, sendo vantajoso em termos de largura da banda, mas oferece pouca flexibilidade na sua programação, além de ter o desempenho sujeito a interferência de correntes parasitas no circuito.

Já o FPAA de tempo discreto utiliza a técnica de *switched capacitor* (capacitor chaveado) ou *switched current* (corrente chaveada), oferecendo maior flexibilidade de programação e menor variação no valor das resistências emuladas, porém sua desvantagem fica por conta da limitação de operação em sinais de altas frequências.

Por este trabalho ser voltado para um dispositivo que utiliza a técnica de tempo discreto, esta será melhor detalhada na sessão 2.2.2.

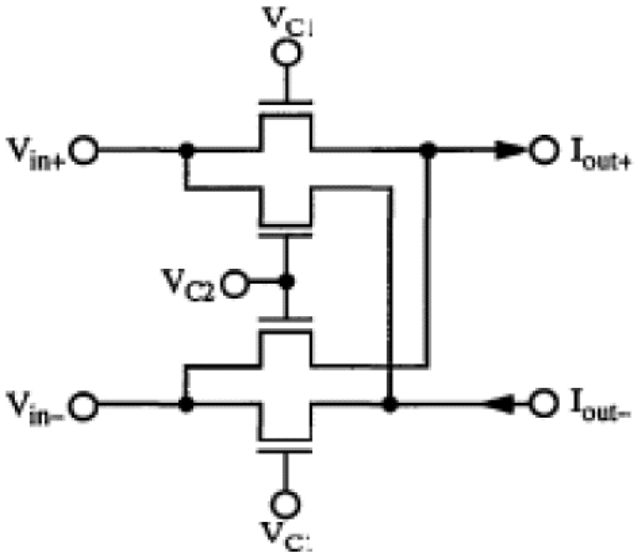
2.2.1. Continuos Time

As FPAAs do tipo tempo contínuo são tipicamente uma matriz de componentes fixos (amplificadores operacionais e/ou transistores) que são ligados por uma matriz de interconexões. Esta matriz é usualmente controlada por registradores digitais, os quais podem ser configurados por um controlador externo, permitindo assim a FPAA ser configurada para implementar uma série de projetos diferentes (SANTOS, 2010).

Devido ao fato de não utilizar uma técnica de amostragem do sinal de entrada, este tipo de FPAA não necessita a utilização de filtros *anti-aliasing* para a aquisição de sinais. A não utilização destes filtros permite suportar maiores faixas de frequência de sinais sem comprometer o desempenho.

Por outro lado, a largura de banda passa a ter um limitador devido a tecnologia da matriz de interconexões, que acabam por introduzir impedâncias parasitas nos sinais recebidos. Além do impacto na largura de banda, essas capacitâncias também introduzem ruídos/interferências no sistema. Visando solucionar esta deficiência, tenta-se reduzir o número de conexões desta matriz de comutação, porém esta redução acaba por implicar na programação do dispositivo, tornando-o suas funcionalidades limitadas.

Figura 12: Transistores MOSFET como transcondutores



FONTE: (REISER, 1999)

Para a implementação de resistências, esta tecnologia utiliza transcondutores, onde uma tensão de referência gerada por um conversor de sinal é utilizada para a programação do valor da transcondutância. Os transcondutores utilizam transistores operando em sua faixa linear de funcionamento, onde os mesmos apresentam larguras de banda relativamente grandes.

A Figura 12 apresenta um transcondutor composto por quatro transistores MOSFETs que fabricados em um processo *CMOS* de 1,2 µm podem atingir uma largura de banda de 100 MHz. O acoplamento cruzado dos transistores garante um elevado funcionamento linear.

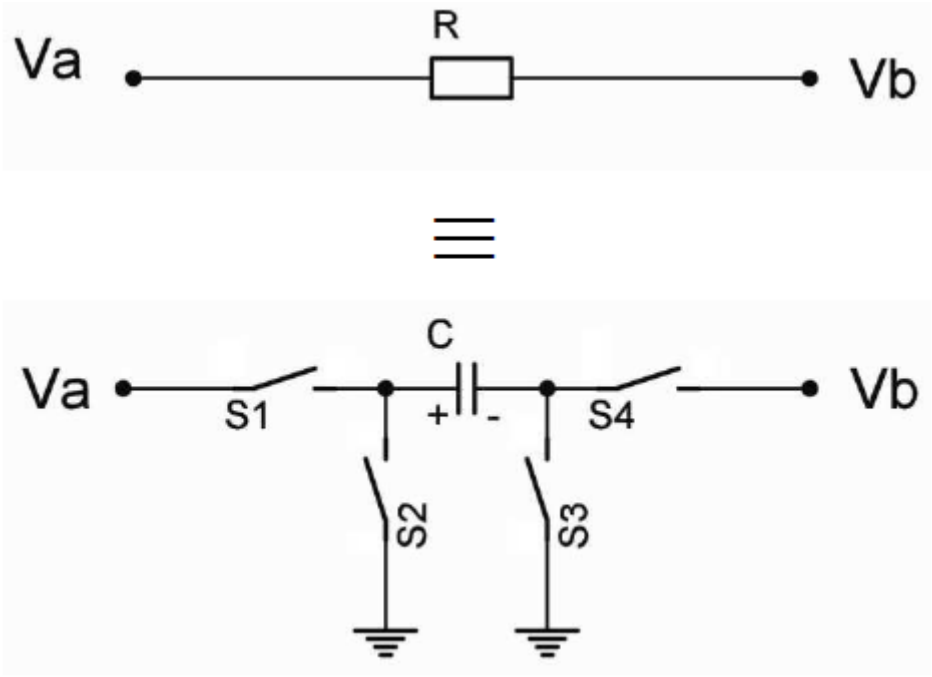
2.2.2. Discrete Time

FPAAs de tempo discreto normalmente são baseados na técnica de capacitor chaveado. Para estes circuitos, a tensão de entrada é amostrada através da abertura e fechamento de uma chave que conecta a entrada a um capacitor inicial. A chave e o capacitor formam uma espécie de registrador analógico, e o caminho do sinal do sistema é dividido por estes registradores (HALL, 2004).

Este tipo de FPAA costuma ser composto por amplificadores operacionais, capacitores e chaves seletoras. As chaves seletoras e os capacitores permitem a emulação de resistores lineares através da variação da frequência de chaveamento e o valor dos capacitores. Isto permite uma grande flexibilidade de configuração para o dispositivo, entretanto, torna mais difícil a elaboração do projeto do dispositivo, pois as chaves e os capacitores podem introduzir ruídos e não linearidades ao sistema. Além disso, esta técnica possui o limite de banda determinada pela frequência de amostragem do FPAA, visto a necessidade de utilização de filtros de *anti-aliasing* e reconstrução nas entradas e saídas de sinal.

A técnica de capacitor chaveado, Figura 13, consiste em quatro chaves, S1, S2, S3 e S4, que abrem e fecham periodicamente. As chaves S1 e S4, assim como S2 e S3, trabalham aos pares e são ativadas em sincronismo, mas em oposição de fase.

Figura 13: Equivalência de resistores e capacitores chaveados



FONTE: (REISER, 1999)

Quando as chaves S1 e S4 estão fechadas, as chaves S2 e S3 estão abertas e o capacitor é carregado com uma tensão dada pela Equação (1):

$$V_c = V_a - V_b \quad (1)$$

Os termos V_a, V_b e V_c indicam tensão de entrada, tensão de saída e tensão no capacitor, respectivamente, tendo como unidade *Volts* [V].

Na próxima etapa, as chaves S1 e S4 abrem, enquanto as chaves S2 e S3 fecham, descarregando o capacitor. Este processo se repete periodicamente, provocando uma carga e descarga constante do capacitor e o seu período de duração é determinado pela frequência de *clock* do dispositivo. A carga do capacitor pode ser determinada pela Equação (2):

$$q = C \; x \; V_c \quad (2)$$

q indica a carga do capacitor em *Coulombs* [C] e C a capacitância do capacitor em *Farads* [F].

De acordo com a Equação (3), a corrente média, I_{med}, que é dada em *Amperes* [A], é definida como a razão entre q e T, que indica o período de chaveamento dado em segundos [s].

$$I_{med} = \frac{q}{T} = \frac{C\ x\ V_c}{T} \tag{3}$$

Comparando a Lei de *Ohm*, Equação (4), com a Equação (3), chega-se na Equação (5), que define o valor da resistência equivalente dada em *Ohms* [Ω]:

$$R = \frac{V}{I_{med}} \tag{4}$$

$$R = \frac{V_c\ x\ T}{C\ x\ V_c} = \frac{T}{C} \tag{5}$$

Como o inverso do período de chaveamento é a frequência de chaveamento, f_c, dada em *Hertz* [Hz], a Equação (5) pode ser reescrita obtendo-se:

$$R = \frac{1}{C\ x\ f_c} \tag{6}$$

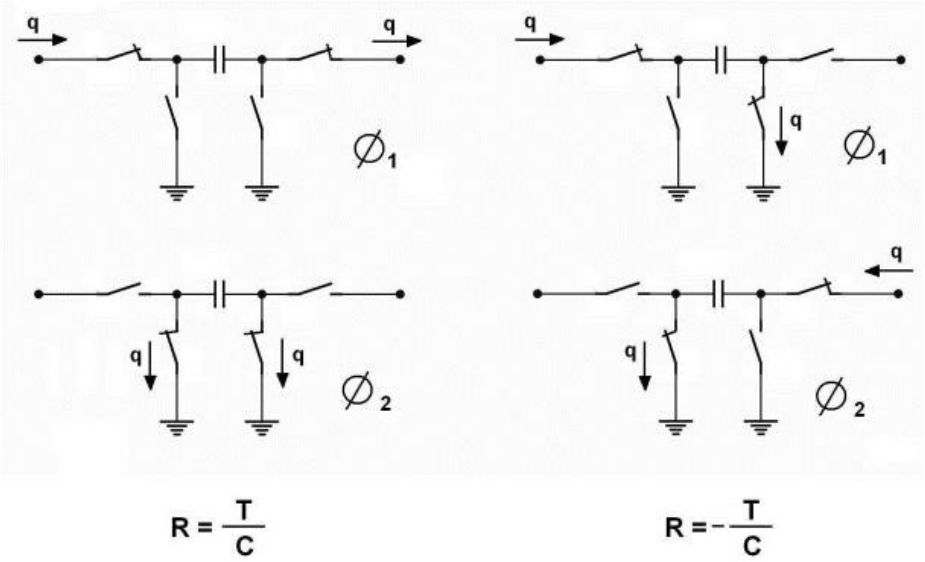
Assim, é possível verificar que a implementação de resistores de diferentes valores em um FPAA de tempo discreto são facilmente configurados, dependendo apenas do valor do capacitor e da frequência de chaveamento. Isto permite que um único arranjo de chaves e capacitor assumam uma vasta gama de valores de resistores.

Outra possibilidade ao se utilizar a técnica de capacitor chaveado, é a criação de resistências negativas. Ao alterar a sequência de abertura e fechamento das chaves, fazendo com que os pares agora sejam S1 com S3 e S2 com S4, conforme a Figura 14, é possível inverter o sentido da corrente no capacitor, criando resistências negativas.

As resistências negativas são teóricas e não existem como componentes discretos, mas teoricamente podemos dizer que uma resistência negativa é uma resistência em que o aumento da corrente que a atravessa provoca uma diminuição de tensão aos seus terminais. (SANTOS, 2010).

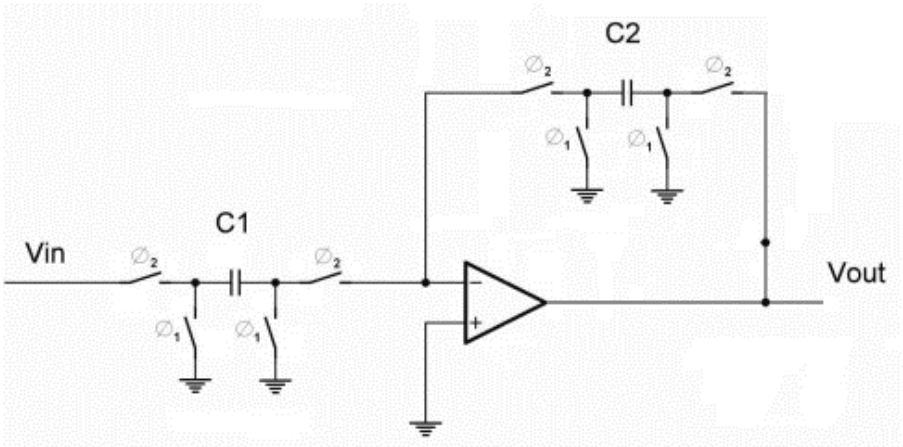
A utilização de resistências negativas permite a implementação simplificada em alguns circuitos, como é o caso para filtros do tipo biquad. Em uma implementação com componentes discretos, são necessários três amplificadores operacionais, enquanto ao implementar através da técnica capacitor chaveado, utilizando o recurso de resistência negativa, pode-se implementar utilizando apenas dois amplificadores operacionais, reduzindo assim a quantidade de componentes e melhorando a eficiência energética do circuito.

Figura 14: Resistências positivas e negativas através de capacitor chaveado



FONTE: (REISER, 1999)

Figura 15: Amplificador inversor utilizando capacitor chaveado



FONTE: adaptado de (REISER, 1999)

Além da possibilidade de resistências negativas, a técnica de capacitor chaveado facilita a implementação de configurações de amplificadores operacionais e construção de filtros ativos. No exemplo da Figura 15, para alterar o ganho do amplificador inversor, basta alterar a razão dos capacitores C2 e C1, além de aumentar a performances dos amplificadores operacionais com relação a tensão de *offset*, largura de banda, *slew rate*, etc. Para o caso de filtros ativos, a frequência de corte pode ser alterada por uma simples mudança da frequência de chaveamento dos capacitores.

Portanto, as principais vantagens são:

- Melhor aproveitamento da área através da obtenção de maior gama de valores de resistência em um menor espaço;

- Menor variação da resistência nominal, referente à tolerância, linearidade e largura de banda;

- Maior precisão no ajuste de frequência de corte em filtros;

- Menor sensibilidade a variações de temperaturas.

A principal desvantagem fica por conta da banda de operação do dispositivo, que de acordo com o teorema de amostragem de *Nyquist,* fica limitado a menos da metade do valor de *clock* do dispositivo.

3. Anadigm Company

A Anadigm Company possui até o momento três gerações de dispositivos FPAAs. Porém, a primeira geração não é mais comercializada pela empresa e a segunda geração já teve o seu fim de sua produção anunciado, sendo ainda possível adquirir exemplares quem restam em estoque. A principal diferença entre as gerações é o modelo de arquitetura adotado.

Utilizando a tecnologia *CMOS*, todas as gerações empregam a técnica de capacitores chaveados, tornando o circuito menos vulnerável a variações de processos e correntes parasitas, contudo, limitando a faixa de operação em frequência do dispositivo devido ao fato da necessidade de amostrar os sinais de entrada.

As FPAAs da Anadigm não são mais que do que um equivalente analógico de uma FPGA, que permite projetar complexos circuitos para processamento e condicionamento de sinais analógicos em um único circuito integrado (SANTOS, 2010).

A utilização do *software AnadigmDesigner2* permite a implementação de complexos circuitos analógicos utilizando módulos analógicos reconfiguráveis, os CAMs – *Configurable Analog Modules*.

Os FPAAs da Anadigm podem ser divididos em duas categorias:

- FPAA com reconfiguração estática;
- dpASPs com reconfiguração dinâmica.

As dpASPs permitem a reconfiguração dinâmica em tempo real, permitindo que a configuração do dispositivo seja alterada mesmo durante o funcionamento, enquanto os FPAAs requerem uma reinicialização do dispositivo. Ambos apresentam as demais característica idênticas.

A programação para a implementação de múltiplas funções analógicas e/ou rápidas adaptações para a manutenção do funcionamento exigidas pelas aplicações implementadas, tais como processamento analógico com condicionamento de sinal, filtragem, aquisição de dados e sistema de controle em malha fechada são alteradas via utilização do *software* ou até mesmo por um microcontrolador.

As características dos FPAAs e dpASPs da Anadigm são:

- Alimentação de 3,3 V e 5 V;
- Tecnologia de capacitores chaveados e matriz de interconexão;
- *Software* de desenvolvimento com tecnologia *Drag and Drop*;
- Funções analógicas pré inseridas nos CAMs;
- *Software* com geração automática de código C;
- Largura de banda até 2 MHz;
- Relação Sinal/Ruído (SNR) em banda larga de até 90 dB;
- Relação Sinal/Ruído (SNR) em banda estreita de até 120 dB;
- Conversores de sinais diferenciais;
- Entradas e saídas com inclusão de amplificadores *chopper* para a obtenção de baixo nível de tensão *offset* e a existência de filtros *anti-aliasing.*

Com exceção da primeira geração, que possuía uma matriz de 4 x 5, as FPAAs da Anadigm são compostas por matrizes 2 x 2 de blocos analógicos configuráveis (CABs), envolvidos por uma rede de interconexões programáveis, possibilitando que um CAB se conecte a qualquer outro da matriz ou a qualquer célula de entrada ou saída.

Os dados da reconfiguração são guardados em uma memória do tipo *SRAM - Static Random Access Memory.*

A partir da segunda geração, os FPAAs passam a contar com funções não lineares como linearização de resposta de sensores e síntese de formas de ondas arbitrárias.

Os dispositivos da *Anadigm Company* seguem o seguinte padrão (com exceção da primeira geração) para identificação de suas características:

ANxxxExx

onde:

- primeiro “x” - tipo de reconfiguração (1 – estática e 2 – dinâmica);
- segundo “x” - geração (2 – segunda e 3 – terceira geração);
- terceiro “x” – interface I/O e conversor AD (0 – I/O fixas e conversor AD interno apenas e 1 – interface I/O flexível com conversor AD por CAM);
- quarto e quinto “x” – quantidade de CABs disponível.

3.1. dpASP AN221E04

A dpASP AN221E04 faz parte da família *Anadigm Vortex*, nome dado a segunda geração, que empregam arquitetura de 5 V e as seguintes características:

- Quatro CABs em uma matriz 2 x 2;
- Quatro células de entrada e saída configuráveis, sendo uma delas com multiplexação 4:1;
- Duas células de saídas dedicadas;
- Uma tabela *LUT – Look Up Table* de 256 *bytes;*
- Um conversor AD do tipo *SAR, Sucessive Approximation Register,* por CAB;
- Um bloco gerador de tensão de referência.
- Um bloco de sistema de *clock*;
- Um bloco de configuração de células de entrada/saída.
- Largura de banda de até 2 MHz;
- Relação Sinal/Ruído (*SNR)* em banda larga de 80 dB;
- Relação Sinal/Ruído (*SNR)* em banda estreita de 100 dB;
- Distorção Harmônica Total (*THD)* de 80 dB;
- Offset *DC:* < 100 µV.

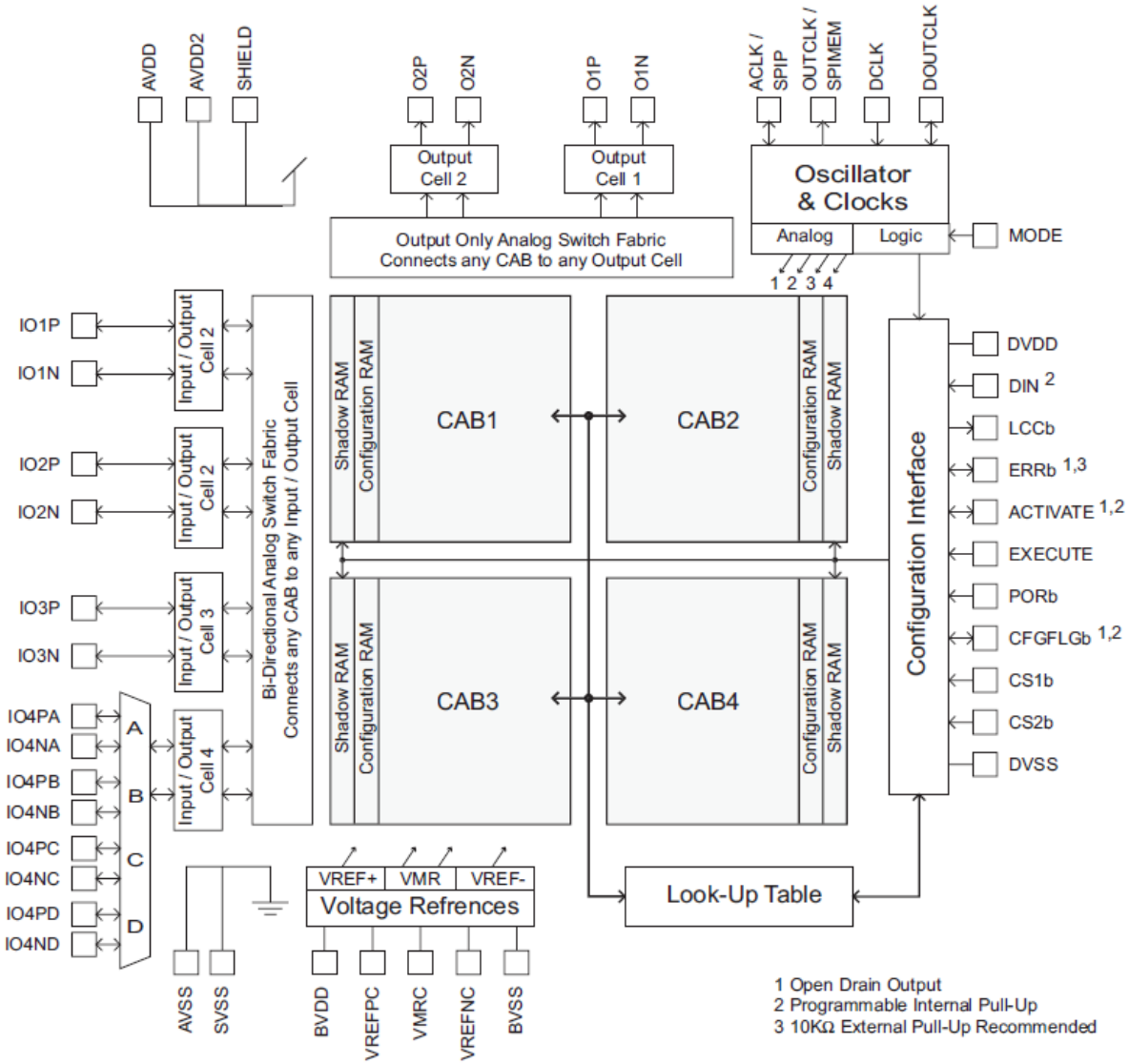
Alguns exemplos de aplicações são:

- Controle em tempo real de sistemas analógicos;
- Sensores inteligentes;
- Controle e filtragem adaptativa;
- *DSP – Digital Signal Processor* – adaptativo;
- Controle industrial e automação adaptativa;
- Sistemas de autocalibragem;
- Compensação de envelhecimento de componentes/sistemas;
- Recalibração dinâmica de sistemas remotos;
- Condicionamento de sinais de baixa frequência;
- Processamento de sinais analógicos.

Pelo fato da dpASP AN221E04 se tratar de uma FPAA com reconfiguração dinâmica, é possível a sua atualização parcial ou total mesmo com o circuito em funcionamento. Enquanto a FPAA recebe a nova configuração, a mesma continua a executar a configuração antiga. Com a transferência completa da nova configuração, após um ciclo de *clock* a FPAA assume os novos parâmetros sem a interrupção do sinal.

A Figura 16 ilustra a estrutura interna do AN221E04.

Figura 16: Estrutura interna do AN221E04

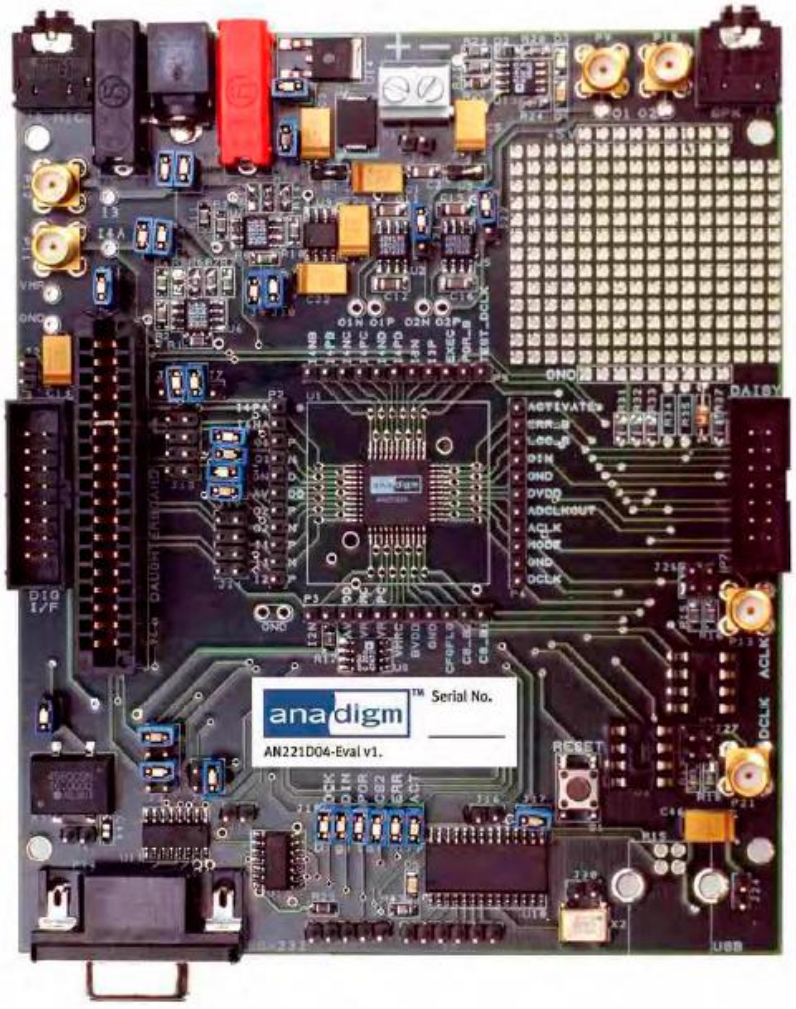


FONTE: (ANADIGM, 2003)

3.2. Kit de Desenvolvimento AN221D04

O kit de desenvolvimento AN221D04 da Anadigm, Figura 17, foi projetado para ajudar no processo de familiarização com dispositivo e teste dos projetos analógicos desenvolvidos. Seu componente principal é o FPAA AN221E04, já previamente descrito neste trabalho.

Figura 17: Kit de desenvolvimento AN221D04



FONTE: (ANADIGM, 2003)

O kit permite que o FPAA seja configurado a partir de memórias, pois o mesmo conta com soquetes para a instalação na placa, um computador utilizando a comunicação serial RS-232, através do *software* AnadigmDesigner2 ou código desenvolvido pelo projetista, além de permitir outras interfaces digitais como programador do dispositivo, por exemplo, um microcontrolador.

A placa também conta com barra de pinos, que oferece acesso direto a todas entradas e saídas do FPAA, além de possuir *jacks* de entrada e saída projetados especialmente para a utilização de alto falantes/fones de ouvido e conectores SMA. A alimentação pode ser realizada por uma fonte regulada de 5 VDC, uma fonte 9 VDC (o kit contém um circuito regulador de tensão) ou até mesmo ser alimentado por um sistema de processamento externo.

3.2.1. Anadigm Boot Kernel - ABK

Para que a interface de comunicação serial RS-232 funcione corretamente, o kit de desenvolvimento conta com dois circuitos integrados importantes nesta placa: o MAX232A e um microprocessador PIC16C745.

O MAX232A é um circuito integrado com a função de converter os níveis de tensão do protocolo RS-232 para níveis compatíveis com o padrão TTL (*Transistor - Transistor Logic)*, tornando compatível a troca de informação entre o FPAA e o dispositivo que utilize a RS-232. O PIC16C745 é o responsável pela execução do Anadigm Boot Kernel – ABK.

O ABK é um conjunto de requerimentos de *software* e protocolos de comunicação desenvolvido para permitir que um computador rodando o *software* AnadigmDesigner2 ou uma aplicação do usuário possa programar e controlar o dispositivo FPAA (Anadigm, 2003).

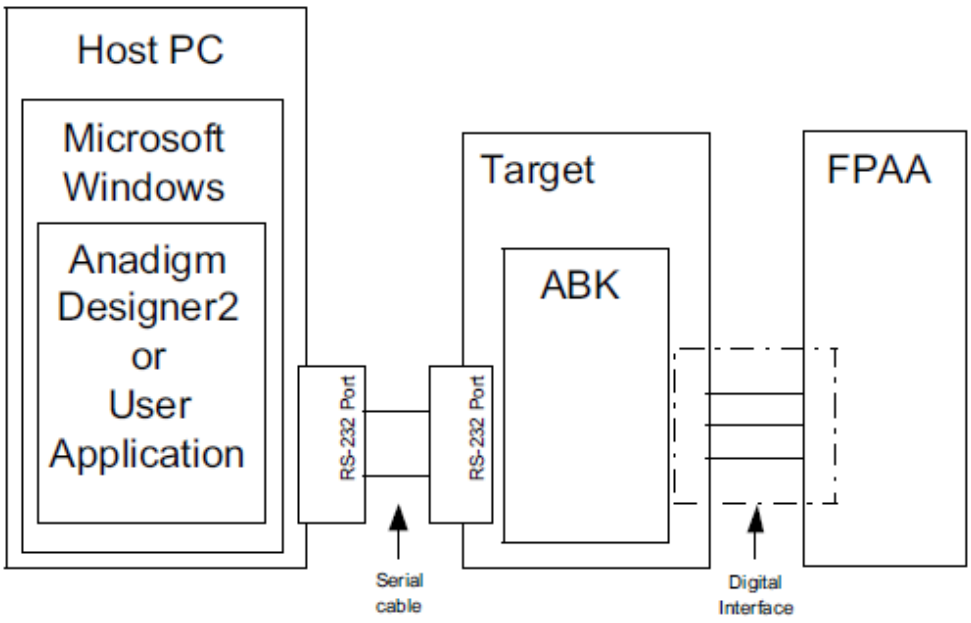
Através de um conjunto de arquivos de código fonte implementados em linguagem C, o ABK funciona como uma interface entre o computador e o FPAA, conforme ilustra a Figura 18.

O ABK não se destina a ser o *software* de controle de um FPAA, mas sim fornecer um meio rápido e fácil de comunicar dispositivos através da RS-232 com o FPAA, sendo necessário o desenvolvimento do *software* de controle. O ABK não é necessário para a implementação de um sistema embarcado.

Entre as capacidades do ABK estão: envio da *Primary Configuration* (Configuração Inicial) e *Update Format* (Atualização) para um ou mais FPAAs, *Reset* de um ou mais FPAAs, leitura dos canais do ADC, leitura dos dispositivos SPI, informação do *target ID* e versão, verificação de status operacional e de

programação, exibição de status de programação do FPAA via LEDs, identificação do FPAA e retorno de comandos suportados pela versão do ABK do kit.

Figura 18: Interface implementada pelo ABK

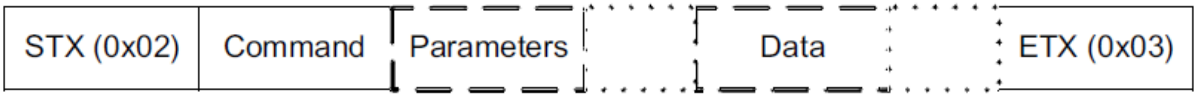


FONTE: (ANADIGM, 2003)

3.2.2. Estrutura de Comandos ABK

A estrutura de comandos para o funcionamento do ABK é apresentado na Figura 19:

Figura 19: Estrutura de comando ABK



FONTE: (ANADIGM, 2003)

A estrutura de comando inicia com o envio do caractere ASCII (*American Standard Code for Information Interchange*) “STX” (Start of Text), definido como valor hexadecimal 0x02 e sempre deve terminar com o envio do caractere ASCII “ETX” (End of Text), com o valor hexadecimal de 0x03. O campo *Command* depende da ação requerida do ABK e o campo *Parameters* é opcional e associado ao

Command. Já o campo *Data* tem o número de caracteres definidos de acordo com a configuração a ser enviada. Caso o comando não envie nenhuma configuração ao FPAA, esse campo não precisa ser enviado.

Todos os parâmetros devem ser enviados no formato ASCII. Para cada byte de informação, dois caracteres ASCII são necessários. Por exemplo, caso o byte a ser enviado seja 0xA5, devem ser enviados o caracteres “A” e “5”. Caso seja enviado uma palavra, por exemplo 0xC420, quatro caracteres ASCII devem ser enviados, sendo eles “C”, “4”, “2” e “0”. Todos os valores precisam ser números inteiros e sem sinal e as letras precisam ser maiúsculas.

A Figura 20 apresenta a lista de comandos suportados pelo ABK, com os suas respectivas ações, parâmetros necessários e notas de aplicação. Importante lembrar que nem todos os comandos estarão disponíveis, pois é necessário que a versão do ABK suporte os comandos a serem executados. Para uma lista de comandos suportados, recomenda-se a utilização do comando *Command Set*, cujo caractere ASCII para o campo *Command* é “S”.

Figura 20: Lista de comandos para ABK

Feature	Command	Parameters	Data	Notes
Download Configuration Data	'0'	One word for the number of Data bytes to follow.	The configuration information to be forwarded to the FPAA(s)	This command will issue a full reset to all devices before sending the Data bytes to the FPAA(s).
	'1'	One word for the number of Data bytes to follow.	The configuration information to be forwarded to the FPAA(s)	This command will **NOT** issue a full reset to the devices. This is useful for sending partial update information to the FPAA(s).
Device Reset	'R'	One byte for the FPAA ID to be reset, or 0xFF for all FPAAs	None	If the same primary or secondary IDs are assigned to a group of FPAAs, then all can be reset using the one ID.
Pin Status	'P'	Two bytes for the pin and its state:- Byte 1: 'A' or 'E' for ACTIVATE or ERRb Byte 2: 'H' or 'L' for High or Low	None (see Notes)	This command should be followed by a '?' Status check. If the Status check returns '0' (OK) then the pin status is correct, if it returns '1' (error) then the pin status is wrong. e.g. 'P' followed by 'A' and H' followed by status check which returns '1' means that the ACTIVATE pin is low.
Target ID and Version info	'#'	None	None	The target returns a null-terminated string with the desired information. AnadigmDesigner2® will display this information in a dialog box when the "Target->Display board information" menu item is selected.
Status	'?'	None	None	Target returns an ASCII character representing the status, with the following values: '0' – OK '1' – Configuration error '4' – Unknown command
Analog-to-Digital Reading	'V'	ASCII character for ADC channel to read. i.e. '0' for channel 0, '1' for channel 1, etc.	None	Target returns value of ADC for desired channel. Data size may depend on ADC capabilities.
SPI-port Reading	'W'	None	None	Target returns value of SPI port. This is only available on ABK v3.1 and later Note: this may be used for reading the SAR ADC of the FPAA
Chip identification	'T'	None	None	Target returns a null-terminated string that gives the types of FPAA(s) in the system. Possible identifiers are: '220E04' – AN220E04 '221E04' – AN221E04 '120E04' – AN120E04 '121E04' – AN121E04 Each chip type is separated by a vertical bar '|' character. When all chips have been listed, the ABK sends an 'END'. If a chip cannot be identified, it is reported as 'UNKNOWN', and identification stops.
Command Set	'S'	None	None	Target returns a null-terminated string that lists the commands that are supported by the ABK. For example, a target that supports all ABK commands in this table would return a string like this: '01PRSTVW?#'

FONTE: (ANADIGM, 2003)

3.3. AnadigmDesigner2

Desenvolvido pela própria Anadigm Company e distribuído gratuitamente, o *software* AnadigmDesigner2 trate-se de uma ferramenta de interface gráfica com o

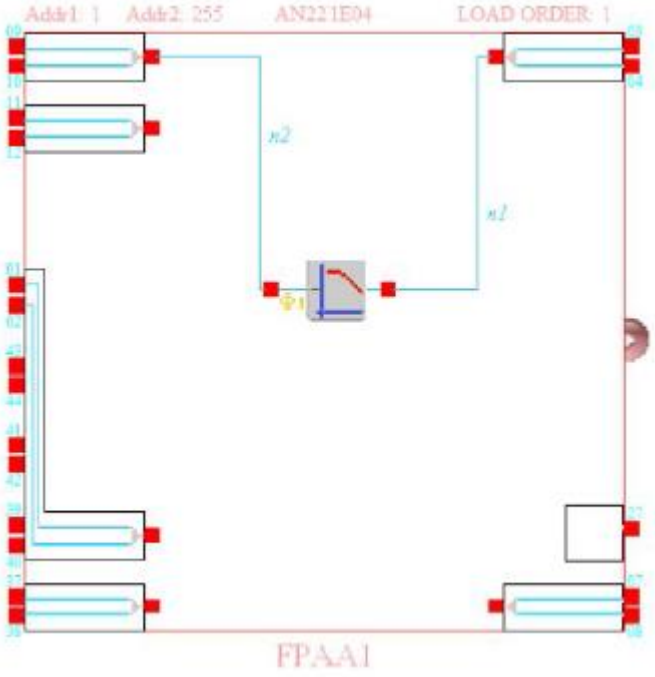
objetivo de permitir que o processo de construção de circuitos analógicos complexos seja rápido e fácil. Através da utilização do *software*, basta selecionar um sub-circuito, denominado CAM (*Configurable Analog Module*), arrasta-lo para o ponto que deseja-se inseri-lo e efetuar as conexões, bastando clicar no primeiro ponto de ligação desejado e arrastar até o próximo. Um exemplo de um circuito configurado é ilustrado na Figura 21.

3.3.1. Configurable Analog Module – CAM

Os CAMs são representados sob a forma de um símbolo abstraído a um nível funcional de programação. Cada CAM é disponibilizado visando a implementação de circuitos/funções simples, mas que da interligação de vários CAMs permitem a criação de circuitos mais complexos.

Entre os CAMs disponíveis, pode-se citar como exemplos o comparador diferencial, inversor, filtro bilinear, filtro biquad, alimentação de tensão DC, gerador de sinais senoidais, integrador, derivador, detecção de zero, entre outros.

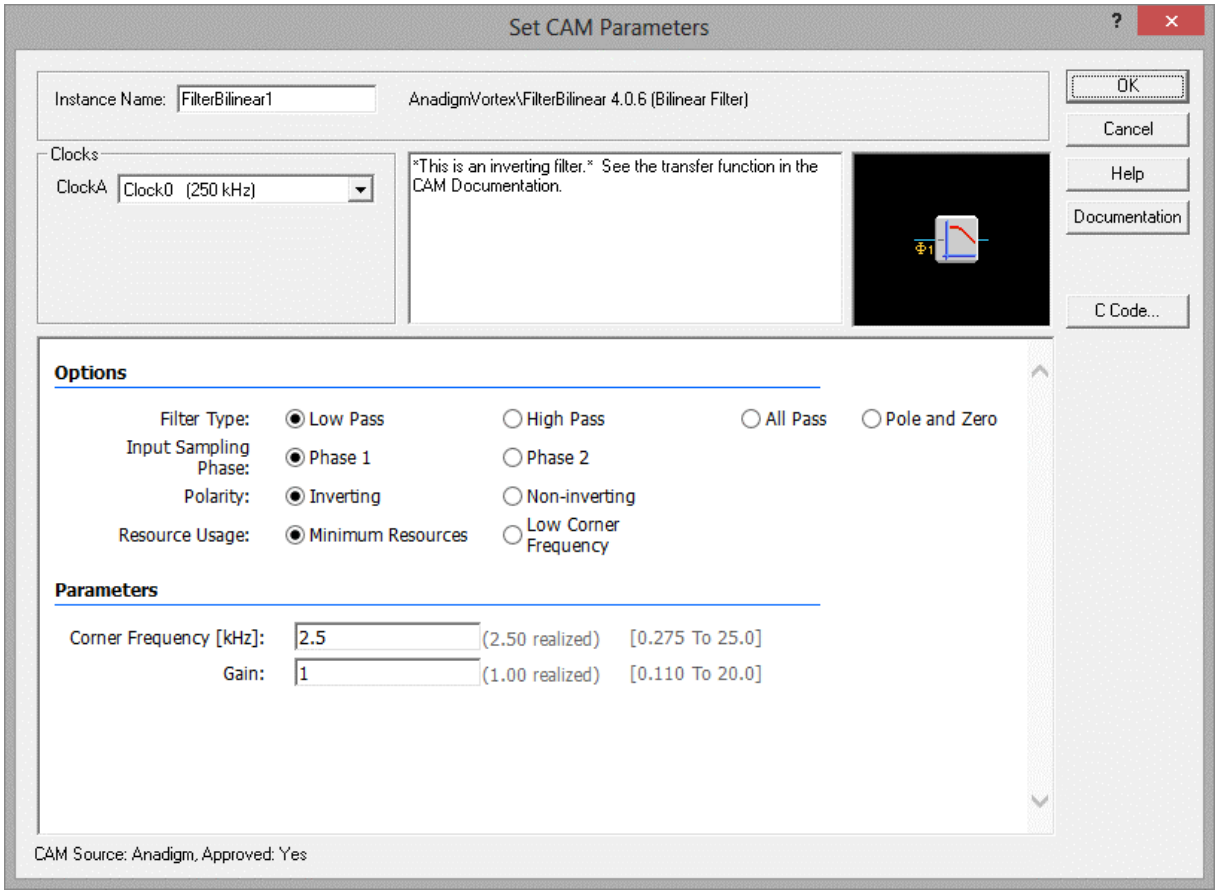
Figura 21: Circuito composto por um CAM de Filtro Bilinear



É possível realizar a alteração dos parâmetros dos circuitos, bastando para isso clicar duas vezes em cima do CAM a ser alterado. Contudo, suas alterações são limitadas, pois o *software* permite a alteração de alguns parâmetros do CAM, mas não a alteração direta dos componentes discretos implementados no CAB (*Configurable Analog Block*). No exemplo da Figura 22, trata-se do CAM de um filtro, pode-se alterar o tipo de filtro, topologia, frequência de corte, ganho e outros parâmetros, porém não é possível a escolha direta dos valores dos componentes discretos utilizados.

Existe a possibilidade de o usuário desenvolver o seu próprio CAM, mas para isso é necessário entrar em contato com a Anadigm Company e obter uma licença especial para desenvolvimento de CAMs. Estes CAMs desenvolvidos podem ser disponibilizados para que outros usuários possam fazer uso dos mesmo, colaborando para a ampliação da biblioteca de CAMs já existentes.

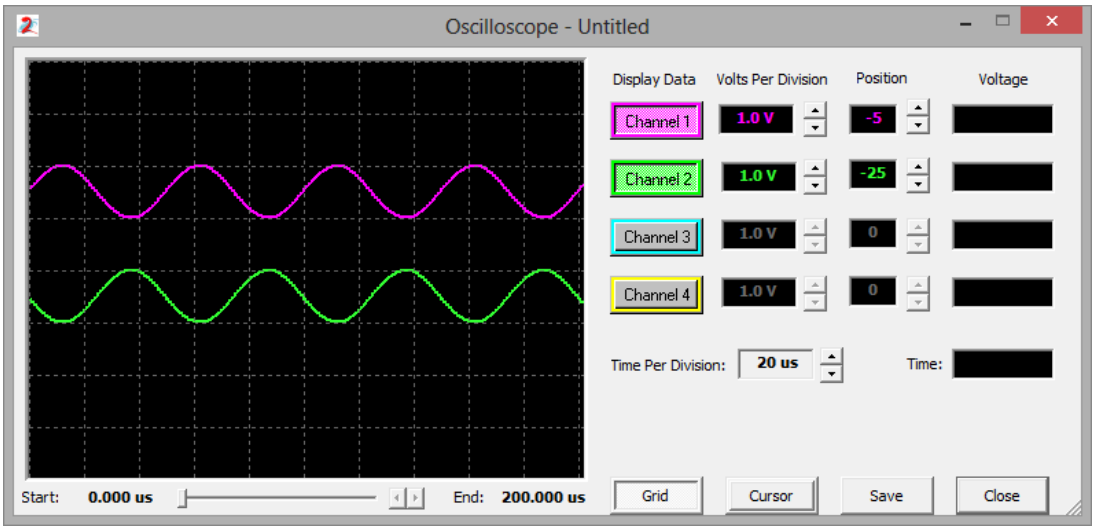
Figura 22: Tela de ajuste de parâmetros de um CAM



3.3.2. Simulações

O AnadigmDesigner2 também oferece suporte a simulações. Após a conclusão da configuração do circuito analógico a ser implementado no FPAA, é possível simular o seu funcionamento antes mesmo de efetuar a configuração do dispositivo. Para isso, o *software* dispõe de geradores de sinais de entrada e um osciloscópio virtual, o qual permite fazer algumas verificações no comportamento dos sinais do circuito, conforme mostra a Figura 23.

Figura 23: Tela de simulação do osciloscópio



3.3.3. Configurações, Gravações e demais Funcionalidades

Além das funções já mencionadas, o AnadigmDesigner2 possui algumas outras atribuições importantes. Além de criar o circuito e realizar os ajustes do parâmetros de cada CAM, é possível realizar ajustes como os *clocks* que serão utilizados pelo FPAA, se deseja-se ter um circuito visando menor consumo de energia elétrica ou maior largura de banda, entre outros.

É também através do *software* que se efetua a gravação do circuito desenvolvido para o *chip*, permitindo colocar o mesmo em funcionamento. Pode-se ainda utilizar o AnadigmDesigner2 para a criação de funções em linguagem em C que irão ser usadas em códigos de programação para a implementação de reconfiguração dinâmica do dispositivo, conforme será melhor abordado posteriormente neste trabalho.

Por último, o programa oferece duas *toolboxes* interessantes de serem exploradas. Trata-se do AnadigmFilter e AnadigmPID.

O AnadigmFilter, Figura 24, tem a função de auxiliar o processo de criação de circuitos que implemente a função de filtros, onde ao alterar os parâmetros do filtro, o usuário já consegue visualizar na tela o seu gráfico de resposta em frequência, fase, entre outros.

Já o AnadigmPID, Figura 25, tem a função de auxiliar o processo de criação de controladores analógicos P, PI, PD e PID. A vantagem se dá no momento de alteração dos ganhos de cada controlador, pois dependendo do tipo de controlador selecionado, o parâmetro de um CAM pode interferir no parâmetro de mais de um controlador, sendo necessário estabelecer relações entre os parâmetros dos CAMs.

Para a obtenção de maiores informações a respeito do programa, recomenda-se a leitura do manual desenvolvido pelo próprio fabricante, o AnadigmDesigner2 User Manual, disponível no site da Anadigm Company.

Figura 24: tela do AnadigmFilter

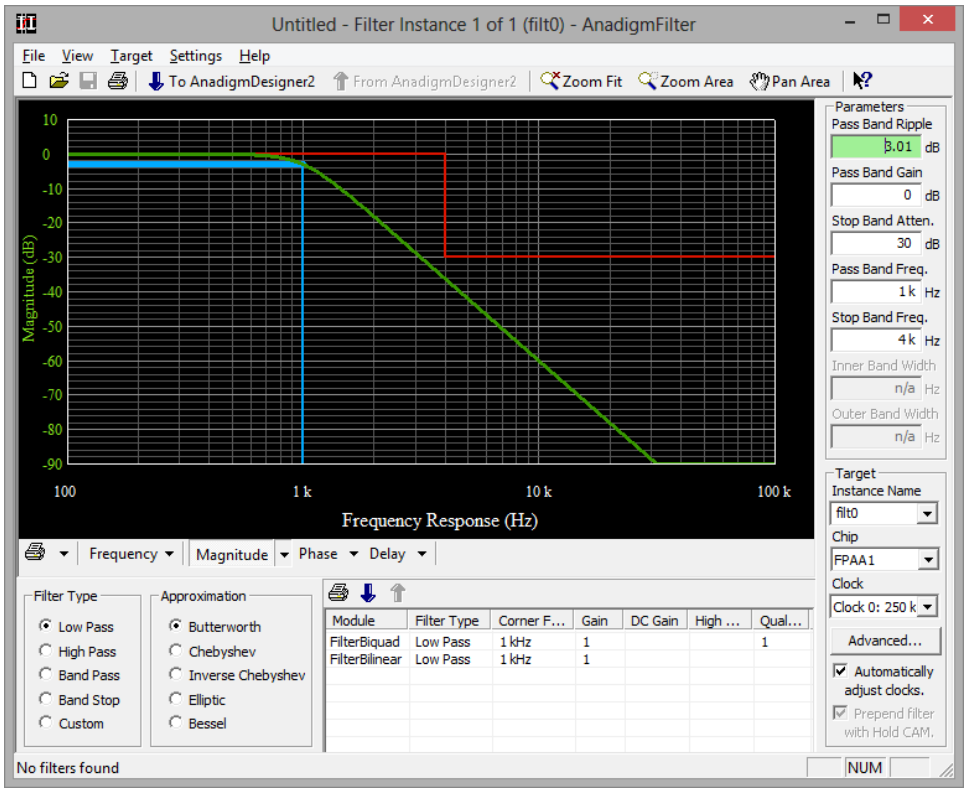
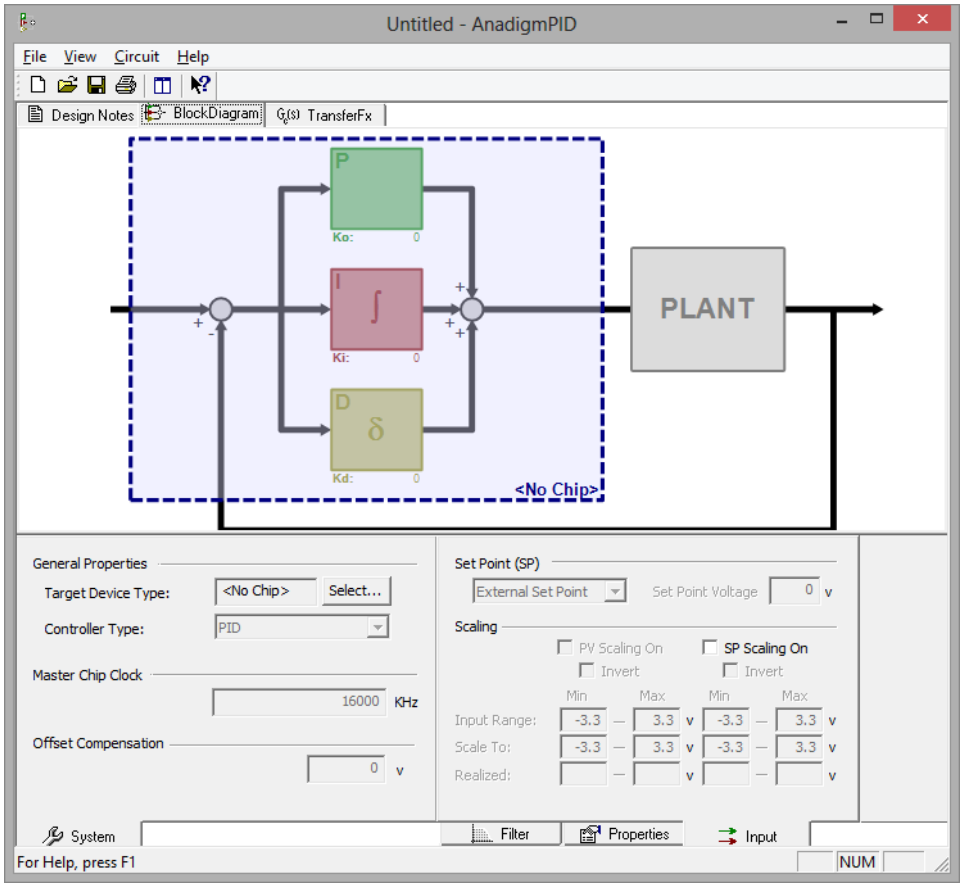


Figura 25: Tela do AnadigmPID



4. Reconfiguração Dinâmica

Run Time Reconfiguration, On The Fly Reconfiguration, In Circuit Reconfiguration ou simplesmente Reconfiguração Dinâmica, são os nomes dado a técnica que permite a reconfiguração de um dispositivo, seja ele analógico ou digital, sem a necessidade de reinicialização, evitando a interrupção do fluxo de dados/sinal (adaptado de MORAES, MESQUITA; 2001).

Para que seja possível utilizar esta técnica, é necessário que o FPAA suporte esta funcionalidade. No caso do fabricante Anadigm, os dispositivos com a nomenclatura AN2XXEXX suportam a implementação deste tipo de reconfiguração.

A atualização do circuito implementado em um FPAA em tempo real permite o desenvolvimento de sistemas inovadores que explorem essa capacidade. Algumas aplicações que podem ser citadas são a atualização dinâmica de sistemas de controle, circuitos de condicionamento de sinais com compensação de degradação e envelhecimento do sensor, ajustes de atraso de grupo, circuitos de varredura de frequência que buscam e travam em uma frequência desejada, além de outros.

Outras aplicações mais triviais também podem ser implementadas através da utilização desta técnica, como é o caso de um controle automático de ganho. Utilizando um circuito de ganho controlado é possível alterar o seu parâmetro de ganho através da reconfiguração dinâmica, bastando utilizar um processador externo para realizar o processo de reconfiguração do dispositivo.

4.1. Funcionamento

O processo de configuração de um FPAA desde a definição dos elementos de cada CAB que serão utilizados, incluindo os seus parâmetros, até o roteamento dos CABs, são realizados pela transferência de uma sequência de dados serial para o dispositivo a ser configurado. Como o FPAA armazena estes dados em uma memória *on chip*, para se realizar uma reconfiguração, basta reenviar os dados com os novos parâmetros desejados. Isto é uma tarefa simples para FPAAs baseados em memórias *SRAM*, pois estes não requerem nenhum dispositivo adicional para gravação. Contudo, para que a reconfiguração possa ocorrer sem a necessidade de

desligamento ou reinicialização do dispositivo, é necessário que o mesmo possua a capacidade de reconfigurar sua memória *on chip* dinamicamente.

Os *chips* da Anadigm que suportam esta característica, apresentam uma estrutura de memória de configuração multi-estágios. Essa estrutura é composta pela *Configuration SRAM* e a *Shadow SRAM*. A *Shadow SRAM* é a responsável por receber e armazenar a sequência de dados serial no dispositivo, enquanto a *Configuration SRAM* mantem a última configuração válida e o circuito em funcionamento. Após a completa transferência dos dados, a configuração é repassada da *Shadow SRAM* para a *Configuration SRAM* em apenas um ciclo de *clock*. Essa transferência pode ocorrer logo após o fim da transmissão da sequência de dados serial ou através da verificação de uma ocorrência interna ou externa ao dispositivo.

Além de um dispositivo FPAA que forneça suporte a técnica de reconfiguração dinâmica, é necessário a utilização de um processador externo, pois o mesmo será responsável pela codificação, gerenciamento e envio dos dados de configuração.

Nas sessões a seguir será explicado em detalhes o modo de operação em que o FPAA deve estar configurado, quais pinos precisam ser controlados e monitorados e as técnicas disponíveis para a implementação da reconfiguração dinâmica em FPAAs da Anadigm.

4.2. Modos de operação

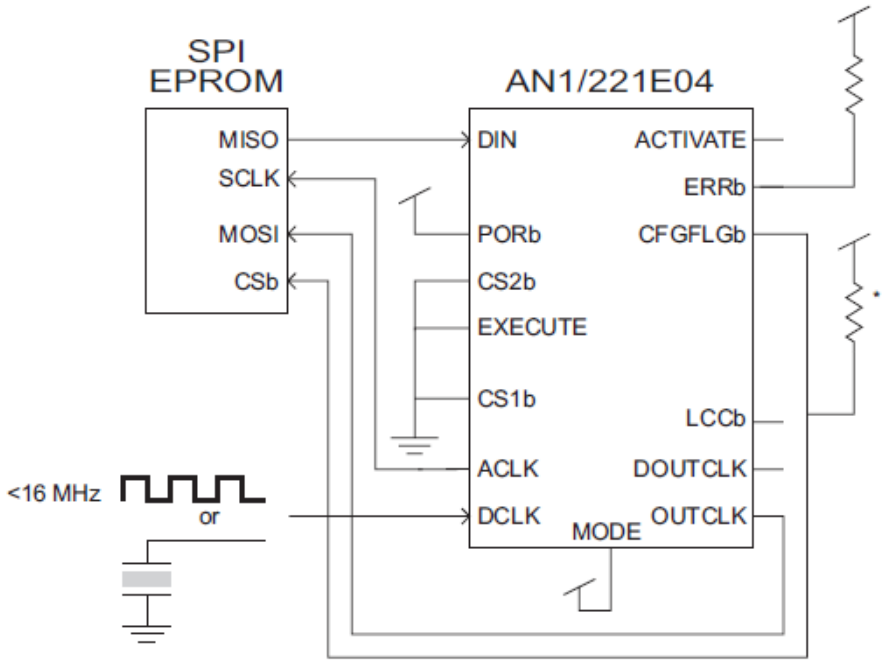
Como já mencionado anteriormente, o comportamento do FPAA é definido pelas configurações que estiverem carregadas em sua memória *Configuration SRAM*. Contudo, esta memória é volátil, ou seja, perde a sua informação sempre que houver interrupção da sua alimentação. Logo, sempre que o FPAA é inicializado, é necessário enviar a configuração desejada para a *Configuration SRAM*. Esta primeira configuração após a inicialização do dispositivo também é chamada de *Primary Configuration* (Configuração Inicial).

O dispositivo recebe esses dados de configuração através do barramento serial, o qual deve ser ligado a algum outro dispositivo que conterá essa

configuração. Os dispositivos da Anadigm podem operar em dois modos: *Master* ou *Slave.*

Quando operando no modo *Master*, pino *MODE* em nível lógico 1 conforme ilustra Figura 26, o FPAA apenas utiliza um dispositivo externo como armazenamento dos dados de configuração, por exemplo uma memória externa do tipo *SPI PROM*. Todo o processo de gerenciamento do trafego de dados e transferência dos arquivos da *Shadow* para a *Configuration SRAM* é feito pelo próprio dispositivo. Contudo, ao operar neste modo não é possível a utilização da técnica de reconfiguração dinâmica, pois o FPAA apenas consegue ler as informações da memória externa e não manipula-las.

Figura 26: FPAA configurado em modo Master



FONTE: (ANADIGM, 2003)

Para que seja possível utilizar a técnica de reconfiguração dinâmica, é necessário a utilização de um processador externo. Para ligar este processador ao FPAA, é necessário a utilização do modo *Slave*. Os FPAAs da Anadigm suportam processadores externos com as seguintes interfaces de conexão: *Synchronous Serial Interface* (SSI), *Serial Peripheral Interface* (SPI) ou interface de barramento convencional.

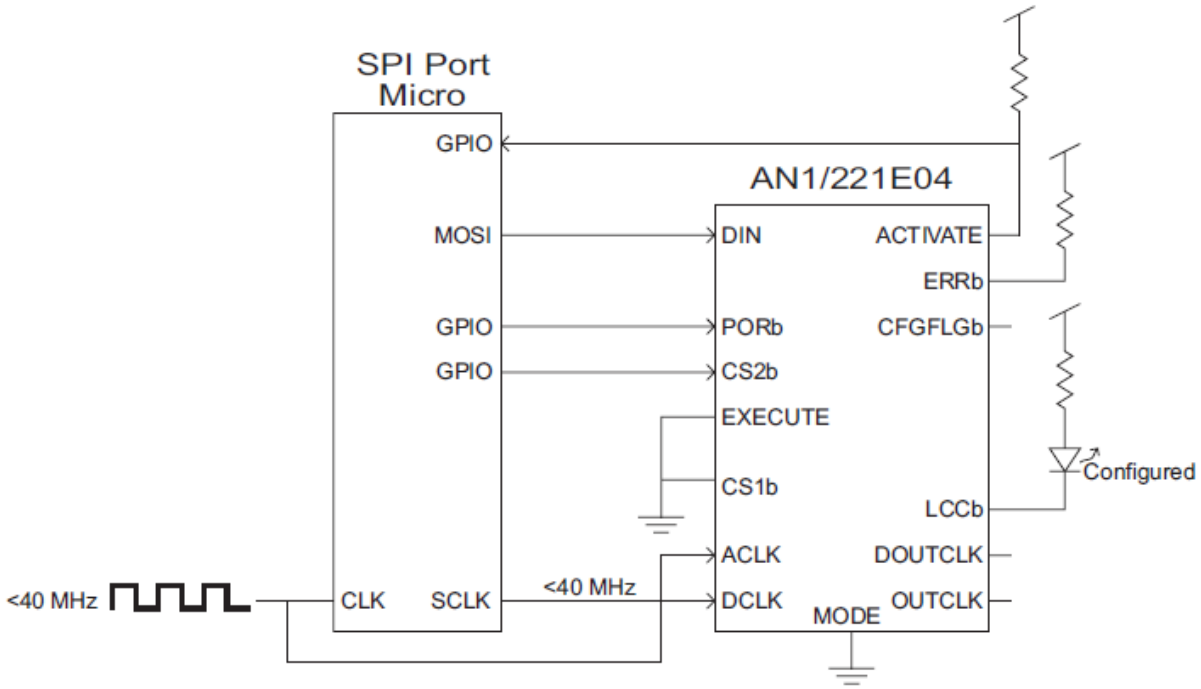
Conforme ilustram as Figuras 27 e 28, a utilização de dispositivos SPI ou SSI possuem a mesma ligação elétrica, mudando apenas o nome das conexões nos dispositivos. Cabe salientar que o pino *Execute* do FPAA possui ligações diferentes, mas isto deve-se apenas para o fato de que ao não se controlar este pino, a reconfiguração é automaticamente carregada no FPAA após a transmissão dos dados de reconfiguração. Quando este pino é controlado, é possível carregar as novas configurações para o FPAA e fazer com que elas passem a funcionar apenas em um instante desejado, através de um comando disparado por este pino.

Quanto as diferenças técnicas entre dispositivos SPI ou SSI, basicamente se devem as frequências suportadas pelos dispositivos.

Já na Figura 29, tem-se um dispositivo de barramento convencional ligado ao FPAA. Como pode ser observado, a parte de endereçamento deste dispositivo é paralelo, sendo necessário um *hardware* adicional para converter este endereço para a forma serial, e assim conecta-lo ao dispositivo.

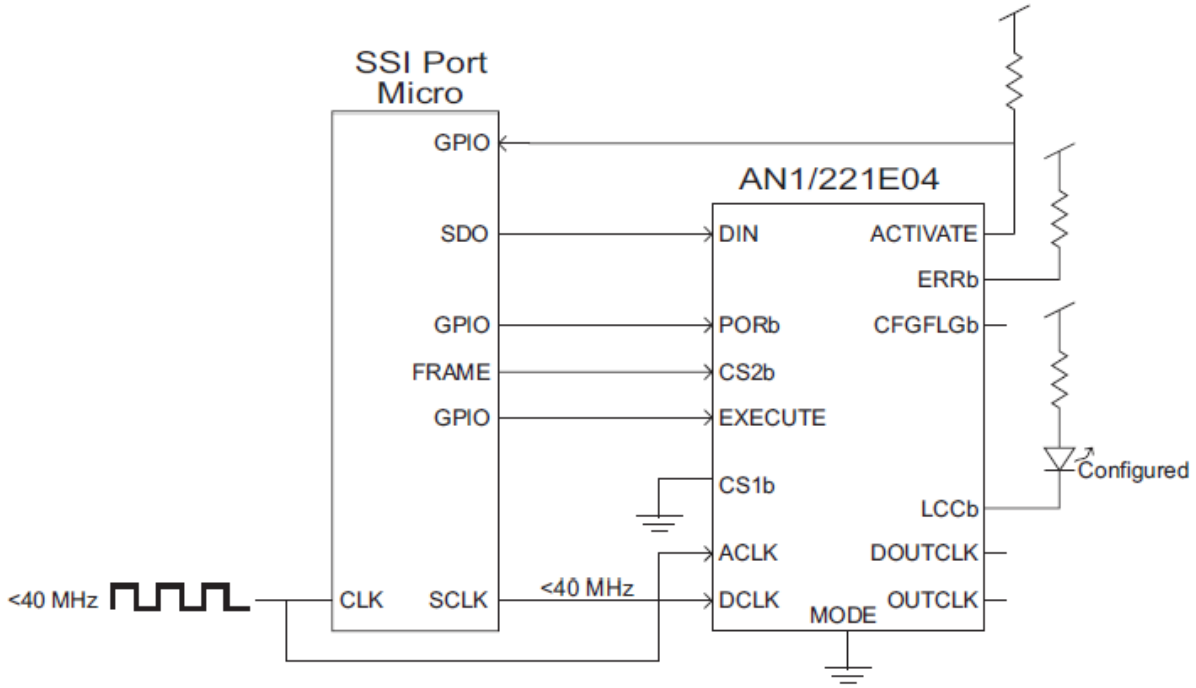
Observar que para todas as configurações em modo *Slave*, o pino *Mode* deve ser ligado ao nível lógico 0.

Figura 27: FPAA configurado em modo Slave utilizando dispositivo SPI e sem controle do pino Execute



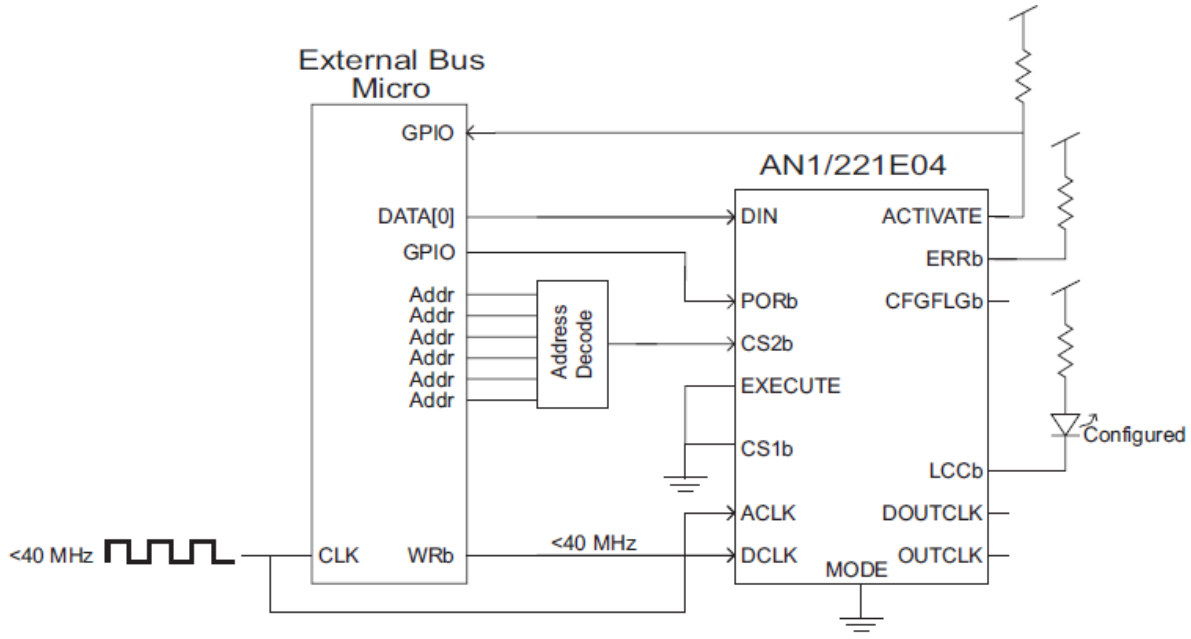
FONTE: (ANADIGM, 2003)

Figura 28: FPAA configurado em modo Slave utilizando dispositivo SSI e com controle do pino Execute



FONTE: (ANADIGM, 2003)

Figura 29: FPAA configurado em modo Slave utilizando dispositivo de barramento convencional e sem controle do pino Execute



FONTE: (ANADIGM, 2003)

4.3. Protocolo de Configuração

Independente da forma como esteja configurado os dispositivos que irão se comunicar com o FPAA, os dados enviados de forma serial precisam seguir um padrão de protocolo. Nos casos de reconfiguração dinâmica, o processador além de gerar os dados de configuração, precisa seguir este protocolo no momento de envio dos dados, caso contrário a reconfiguração não será bem sucedida.

Existem dois formatos de dados que compõem o protocolo de configuração: o *Primary Configuration Format* (Configuração Inicial), responsável pela configuração inicial do dispositivo e o *Update Format* (Reconfiguração), responsável pela atualização das configurações já estabelecidas.

4.3.1. Primary Configuration Format

Este formato de configuração é exigido sempre que o dispositivo for inicializado, sendo responsável pela primeira configuração.

Sempre que o dispositivo é ligado, a *Shadow SRAM* tem todas suas posições de memórias configuradas para nível lógico 0. A Configuração Inicial é então necessária para que seja enviado nível lógico 1 em determinadas posições da memória da *Shadow SRAM*, para que posteriormente estas informações sejam repassadas para a *Configuration* SRAM e o dispositivo esteja devidamente configurado, de acordo com o circuito programado.

A composição da Configuração Inicial é formada pelo *Header Block*, seguido de um ou mais *Data Block*. O *Header Block* é composto pelo byte de sincronização, quatro bytes de identificação do modelo do FPAA, um byte de endereçamento do FPAA e um byte de controle, conforme ilustra a Figura 30.

Um mesmo processador pode ser responsável pela configuração de mais de um dispositivo FPAA. Para que se possa selecionar qual dispositivo será configurado utiliza-se o byte de endereçamento, o *ID1 Byte*, para realizar a identificação de cada *chip*. Já o byte de controle, *Control,* é utilizado para configurar algumas funções e formas de ativação de comandos do FPAA, como exibido na Figura 31. Para que a configuração seja automaticamente carregada após o seu envio, o bit 2 do byte de controle precisa estar em nível lógico 1.

Figura 30: Estrutura do Primary Configuration

	Data	Byte Name	Description
Header Block	11010101 D5	SYNC	Synchronization byte, always D5
	10110111 B7	JTAG ID BYTE 0	Bits [7:0] of JTAG Device ID - 0x800022B7 (or 0x800012B7)
	00100010 22	JTAG ID BYTE 1	Bits [15:8] of JTAG Device ID
	00000000 00	JTAG ID BYTE 2	Bits [23:16] of JTAG Device ID
	10000000 80	JTAG ID BYTE 3	Bits [31:24] of JTAG Device ID
	XXXXXXXX	ID1	ID1 Byte
	XXXXXXXX	CONTROL	Configuration Control Byte
Data Block (first)	11XXXXXX	BYTE ADDRESS	Starting Byte Address (DATA_FOLLOWS = 1)
	XXXXXXXX	BANK ADDRESS	Starting Bank address
	XXXXXXXX	DATA COUNT	Data byte count, a value of 00 instructs 256 bytes
	XXXXXXXX	DATA 0	Data byte to write to starting address + 0
	XXXXXXXX	DATA 1	Data byte to write to starting address + 1
	Remaining data bytes go in this region...		
	XXXXXXXX	DATA n	Data byte to write to starting address + n
	00101010 2A	ERR	Error check byte
Remaining data blocks go in this region...			
Data Block (last)	10XXXXXX	BYTE ADDRESS	Starting Byte Address (DATA_FOLLOWS = 0)
	XXXXXXXX	BANK ADDRESS	Starting Bank address
	XXXXXXXX	DATA COUNT	Data byte count, a value of 00 instructs 256 bytes
	XXXXXXXX	DATA 0	Data byte to write to starting address + 0
	XXXXXXXX	DATA 1	Data byte to write to starting address + 1
	Remaining data bytes go in this region...		
	XXXXXXXX	DATA n	Data byte to write to starting address + n
	00101010 2A	ERR	Error check byte

FONTE: (ANADIGM, 2003)

O *Data Block* é formado pelos bytes *Byte Address, Bank Address, Data Count, Data* e *ERR*. O *Byte Address* e o *Bank Address* são os responsáveis pela posição na memória em que a informação será salva. Observar que caso existe mais blocos de configuração na sequência, o bit número 6 deste byte precisa estar em nível lógico 1. Caso contrário o mesmo deve estar em nível lógico 0, indicando ser o último byte de dados. O byte *Data* contém a informação e o byte *ERR* sempre irá enviar o valor hexadecimal 2A, indicando que os dados foram corretamente transmitidos. Caso o FPAA receba um valor diferente deste, irá interpretar como erro na comunicação, descartando a configuração enviada.

Figura 31: Definição dos bits do byte de controle

Bit Number								
7	6	5	4	3	2	1	0	
0	0	0	0	0	1	0	1	Default bit values as generated by AnadigmDesigner®2
								PULLUPS: **1:** Enable internal pull-ups. **0:** Disable internal pull-ups. This bit is used to enable internal pull-ups on the CFGFLGb and ACTIVATE pins. PULLUPS is sticky, i.e. Once set, it stays set until a device reset. If the pin is externally loaded then it is recommended that an external pull-up resistor be used instead of the internal. (Note - DIN pull-up is controlled by configuration data only.) (Note - ERRb always requires an external pull-up resistor. 10KΩ is the recommended value.)
								(Factory Reserved): **1:** Factory Reserved setting. **0:** Normally set to "0".
								ENDEXECUTE: **1:** At the end of the current transfer cycle, Shadow SRAM will be copied into Configuration SRAM. **0:** No action. (See below and section 3.1.14 further explanation.)
								SRESET: **1:** The device will perform a reset. **0:** No action. This bit allows the host to initiate a soft reset. The device will reset as soon this bit is latched.
								READ: **1:** Sets the device in read mode, Configuration SRAM and LUT only. **0:** Sets the device in write mode.
								STOP_READBACK: **1:** Stop any data read back from the device. **0:** Allow data read back from the device. This bit can be set during Primary Configuration or Update. If set, an internal flag is set which prevents all further data read backs. This internal flag can only be reset by re-powering the device, thereby destroying the SRAM contents. If any attempt to do a read back is made after this bit is set, then ERRb will drive low for 14 DCLK cycles and the device will be reset to a point where a Primary Configuration is required. In the AN121E04 device, this feature is superfluous.
								RESET_ALL: **1:** On an error, the ERRb output will pull low for 15 DCLK cycles and the device will be reset to a point where a Primary Configuration is required. **0:** On an error, the ERRb output will be pulsed low for a single DLCK cycle and the device will be reset to a point where only an Update is required. (Review section 3.1.7 for further explanation.)
								(Factory Reserved): **1:** Factory Reserved setting. **0:** Normally set to "0".

FONTE: (ANADIGM, 2003)

4.3.2. Update Format

Após a conclusão do processo de Configuração Inicial, o dispositivo encontra-se configurado e funcionando. Para reconfigurar o FPAA de forma dinâmica, não é necessário o reenvio completo das informações, mas apenas qual parte da memória precisa ser atualizada. Para isso, utiliza-se a Reconfiguração.

A Reconfiguração é composta por um *Header Block* reduzido, contando apenas com o byte de sincronização, byte de endereçamento e o byte de controle. O

byte de endereçamento é utilizado apenas para informar qual *chip* deve ser reconfigurado em caso de múltiplos dispositivos no mesmo barramento.

Ao contrário do que ocorre no Configuração Inicial, o *Data Block* apenas contém informação das posições de memória que serão alteradas com relação a configuração atual, mantendo praticamente o mesmo formato, como ilustra a Figura 32. Porém, é importante observar que este bloco passa a possuir a opção da utilização de código detector de erros, *Cyclic Redundancy Check* – CRC.

Figura 32: Estrutura do Update

	Data	Byte Name	Description
Header Block	11010101 D5	SYNC	Synchronization byte, always D5
	XXXXXXXX	TARGET ID	ID1, ID2, or 0xFF - Logical address of the target device(s).
	XXXXXXXX	CONTROL	Configuration Control Byte
Data Block (first)	11XXXXXX	BYTE ADDRESS	Starting Byte Address (DATA_FOLLOWS = 1)
	XXXXXXXX	BANK ADDRESS	Starting Bank address
	XXXXXXXX	DATA COUNT	Data byte count, a value of 00 instructs 256 bytes
	XXXXXXXX	DATA 0	Data byte to write to starting address + 0
	XXXXXXXX	DATA 1	Data byte to write to starting address + 1
	Remaining data bytes (if any) go in this region...		
	XXXXXXXX	DATA n	Data byte to write to starting address + n
	XXXXXXXX or 00101010 2A	CRC_MSB or ERR	Most significant byte of CRC16 error code or (depending on Bit 5 of BYTE ADDRESS) Basic error check byte
	XXXXXXXX	CRC_LSB	Least significant byte of CRC16 error code (if used)
Remaining data blocks (if any) go in this region...			
Data Block (last)	10XXXXXX	BYTE ADDRESS	Starting Byte Address (DATA_FOLLOWS = 0)
	XXXXXXXX	BANK ADDRESS	Starting Bank address
	XXXXXXXX	DATA COUNT	Data byte count, a value of 00 instructs 256 bytes
	XXXXXXXX	DATA 0	Data byte to write to starting address + 0
	XXXXXXXX	DATA 1	Data byte to write to starting address +1
	Remaining data bytes (if any) go in this region...		
	XXXXXXXX	DATA n	Data byte to write to starting address + n
	XXXXXXXX or 00101010 2A	CRC_MSB or ERR	Most significant byte of CRC16 error code or (depending on Bit 5 of BYTE ADDRESS) Basic error check byte
	XXXXXXXX	CRC_LSB	Least significant byte of CRC16 error code (if used)

FONTE: (ANADIGM, 2003)

4.4. Técnicas de Implementação

Para configurar os seus FPAAs, a Anadigm desenvolveu o AnadigmDesigner2. Trata-se de um *software* EDA, *Electronic Design Automation,* que permite a configuração dos dispositivos de forma rápida e simples.

Este *software* além de permitir a elaboração dos circuitos, permite a configuração dos dispositivos de forma estática e a implementação de técnicas de reconfiguração dinâmica. São duas técnicas disponíveis pela Anadigm para a reconfiguração dinâmica: *State Driven Method* e *Algorithmic Method*. Cada técnica possui vantagens e desvantagens, os quais serão apresentados a seguir.

4.4.1. State Driven Method

O *State Driven Method* foi desenvolvido voltado para situações em que o *hardware* de processamento externo, responsável pela reconfiguração dinâmica, seja de baixo custo, logo, apresentado pouca capacidade computacional e de memória. Este método trabalha com o conceito de pré – configurações, pois todas as possibilidades de configuração que o dispositivo possa vir a assumir necessitam ser planejadas e seus arquivos de reconfiguração gerados pelo AnadigmDesigner2.

Com os códigos previamente gerados, o processador externo executará uma rotina em que apenas necessita enviar os dados, de forma serial, para o FPAA e gerenciar qual bloco de reconfiguração deve ser enviando e em que instante.

Sua vantagem fica por conta da já mencionada necessidade de *hardware* externo de baixa capacidade computacional e facilidade de geração dos blocos de configuração. Este método também permite que a reconfiguração implemente um novo circuito analógico, e não apenas alteração de parâmetros do circuito já existente. A sua desvantagem fica por conta da limitação de a reconfiguração ser restrita à pré – configurações já previamente planejadas

4.4.2. Algorithmic Method

O *Algorithmic Method* é um método baseado no uso de códigos na linguagem de programação C. Ao contrário do *State Driven Method*, este método requer um *hardware* externo com maior capacidade de processamento e memória.

No *software* AnadigmDesigner2, cada CAM tem associado uma função em código C que foi desenvolvida para programar os seus parâmetros. Utilizando este método, essas funções podem ser exportadas para serem utilizadas no processador que irá realizar a reconfiguração dos *chips*.

O código C que é gerado contém informações de baixo nível dos componentes utilizados para a implementação do circuito, permitindo assim a geração de códigos de reconfiguração com a alteração dos parâmetros desejados em cada CAM, sem a necessidade de recorrer a pré – configurações.

Contudo, essas funções não são suficientes para reprogramar o dispositivo FPAA, sendo necessário elaborar uma rotina em C que faça uso dessas funções para gerar os dados de configurações verificados na sessão anterior.

Além de gerar esses dados, o processador segue responsável pelo envio dos dados para o dispositivo serial.

Por fim, a grande vantagem desse método está na livre reconfiguração dos parâmetros do circuito implementando, respeitando as limitações das CAMs utilizadas no circuito. A sua desvantagem fica por conta da necessidade de um processador melhor, a possibilidade de apenas alterar parâmetros do circuito e não reconfigurar um circuito diferente do já em uso e o fato de requerer um maior conhecimento em linguagem C para desenvolver o código para o correto funcionamento.

Esta técnica foi a utilizada neste trabalho e será melhor explicada no próximo capítulo, com exemplos de implementação.

5. Implementação e Análise dos Resultados

Para verificar o funcionamento da reconfiguração dinâmica através da técnica de *Algorithmic Method* nos dispositivos FPAA da Anadigm Company, realizou-se a implementação de dois circuitos analógicos: um circuito amplificador de tensão não inversor e um circuito controlador PID.

Foram utilizados durante esta etapa um computador rodando sistema Windows, compilador DEV-C++ 5.7.1, *software* AnadigmDesigner2, kit de desenvolvimento AN221D04 da Anadigm, osciloscópio e circuitos analógicos quando necessários. O computador foi utilizado para rodar o código gerado pelo compilador DEV-C++ em tempo real, através da comunicação RS-232, fazendo a função de processamento externo para a reconfiguração dinâmica do FPAA.

5.1. Sistema I – Circuito Amplificador de Tensão Não Inversor

O primeiro sistema escolhido para ser implementado trata-se de um circuito amplificador de tensão não inversor. O objetivo desse circuito é permitir a saída de um nível de tensão DC que varie entre 0,03 V e 3 V, tendo o seu ajuste de tensão realizado através do ajuste do ganho do amplificador. O limite inferior de tensão deve-se a limitações de ajustes do CAM utilizado, enquanto o limite superior foi definido para fins de projeto.

A implementação desse circuito teve o objetivo de familiarização com a técnica de reconfiguração dinâmica por se tratar de um circuito simples, facilitando a implementação, e ser um circuito que auxiliará o ensaio do sistema de controle do circuito do Sistema II. Além disso, esse circuito permitiu verificar alguns parâmetros do FPAA utilizado, sendo eles o *Slew Rate* do *chip* e o tempo que o dispositivo leva para ser reconfigurado.

5.1.1. Funcionamento

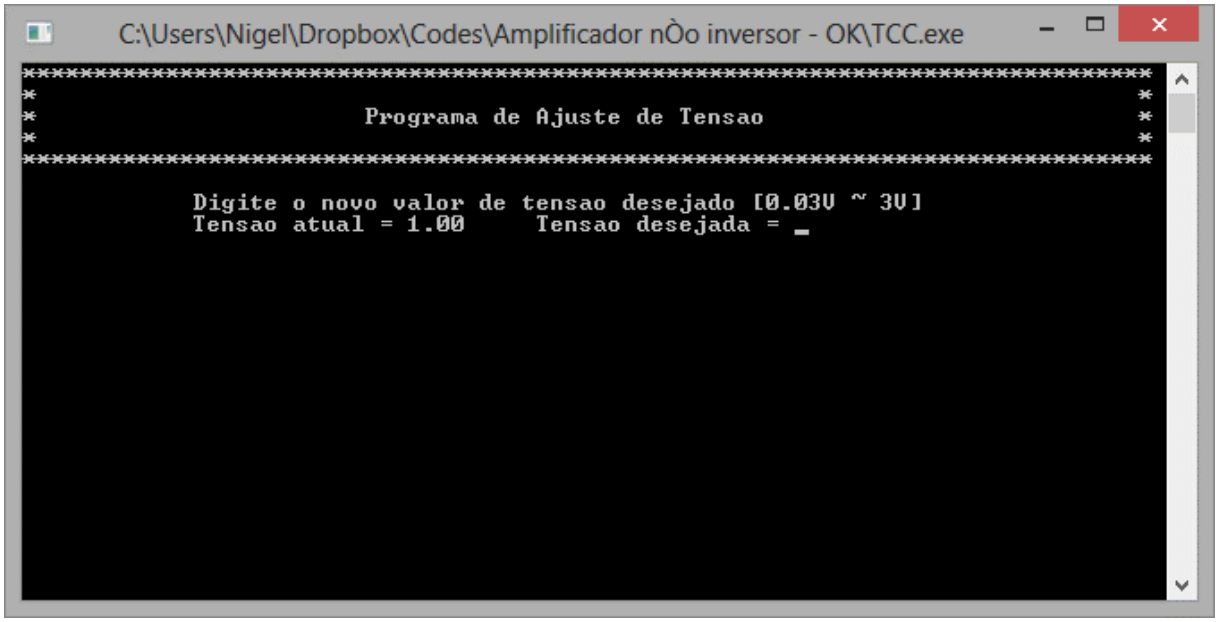
O sistema é composto uma fonte de tensão 3 VDC e um amplificador de tensão não inversor. Este sistema não requer nenhum *hardware* externo adicional

para o seu funcionamento, além do computador que irá rodar o programa de reconfiguração dinâmica.

Como condição inicial, o amplificador é ajustado para trabalhar com um ganho de 0,333, o que resulta em uma tensão de saída de aproximadamente 1 V. O *software* de reconfiguração dinâmica é executado no computador e foi programado para que o usuário informe o nível de tensão desejada, sendo o ganho necessário no amplificador calculado pela rotina. A Figura 33 demonstra a tela de ajuste pelo usuário.

Sempre que o usuário entrar com um novo valor de tensão, o FPAA é reconfigurado e a tensão de saída ajustada para o nível solicitado. Caso o valor informado esteja abaixo do limite de 0,03 V, o sistema será reconfigurado para uma saída de 0,03 V. Caso o valor seja acima de 3 V, o sistema irá se ajustar para a tensão máxima de 3 V.

Figura 33: Tela de ajuste de tensão do Sistema I

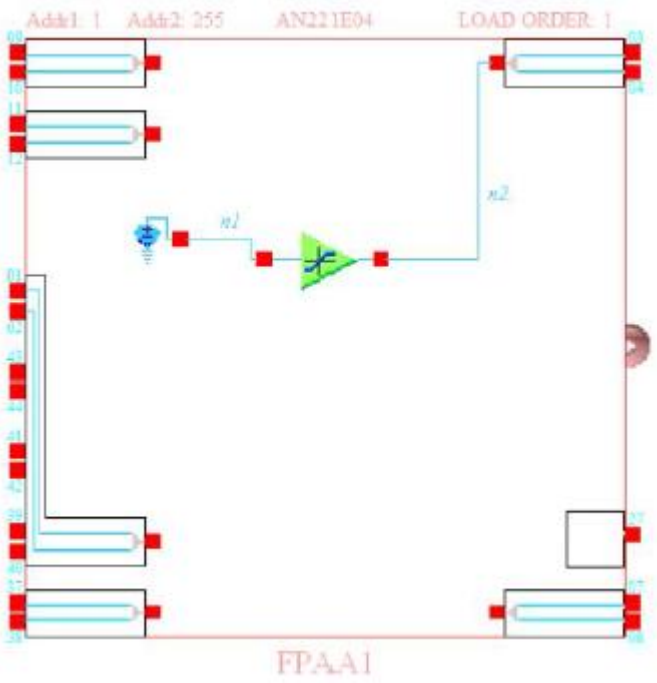


5.1.2. Implementação

Para a implementação deste sistema, utilizou-se dois CAMs: *Voltage*, que implementa a fonte de 3 VDC e *GainLimiter*, que implementa o amplificador de tensão não inversor com limite de tensão máxima de saída. A tensão de saída máxima do amplificador foi ajustada para 3 V. A máxima tensão que poderia ser

obtida com o FPAA é uma tensão de 4 V, devido as limitações de *hardware* do chip. O *clock* geral utilizado foi de 16 MHz e o dos CAMs foi de 4 MHz. Os pinos para a saída diferencial do nível de tensão ajustado são pelos pino O1P (pino 3) e O1N (pino 4). O circuito configurado no *software* AnadigmDesigner2 é demonstrado na Figura 34.

Figura 34: Sistema I no AnadigmDesigner2



5.1.3. Código para Reconfiguração Dinâmica

Para que o sistema funcione corretamente e possa ser reconfigurado dinamicamente, é necessário desenvolver uma rotina que se comunique com o FPAA quando necessário. A comunicação entre o computador e o kit de desenvolvimento se dará por meio de uma comunicação serial RS-232.

Essa rotina, que foi desenvolvida em linguagem C, pode ser dividida em três partes:

- Cálculo dos parâmetros de reconfiguração do circuito;
- Geração dos arquivos de reconfiguração;
- Envio dos dados para o dispositivo a ser configurado via comunicação serial.

Para a parte de comunicação serial, utilizou-se como sub rotina um *software* criado por Teunis van Beelenwas e disponibilizado gratuitamente. Para maiores detalhes de versão utilizada e como obter o *software*, consultar o cabeçalho da rotina desenvolvida, que encontra-se em anexo neste trabalho.

Os comandos utilizados dessa sub rotina são:

- **RS232_OpenComport(int comport_number, int baudrate)**. Utilizado para inicializar a porta COM (*Communication Port*) no computador. O parâmetro *int comport_number* recebe um número referente a porta COM a ser utilizada e *int baudrate* recebe um número inteiro referente a velocidade definida para a comunicação.

- **RS232_SendByte(int comport_number, unsigned char byte)**. Utilizado para o envio de caracteres. O parâmetro *int comport_number* recebe um número referente a porta COM utilizada e *unsigned char byte* recebe o caractere a ser enviado. Pode ser utilizado o seu representante em forma inteira ou hexadecimal, de acordo com a tabela ASCII.

Na etapa de geração dos arquivos de reconfiguração, é necessário a utilização das funções em linguagem C que são utilizadas pelos CAMs no sistema a ser implementado. Para se ter acesso a essas funções, utiliza-se o *software* AnadigmDesigner2 para a geração destes arquivos. Após o circuito previamente criado e configurado, basta ir na aba *Dynamic Config* e selecionar a opção *Algorithmic Method*. Irá surgir a tela da Figura 35.

Como pode se observar, serão gerados quatro arquivos. Os arquivos *ApiCode.c, ApiCode.h, CAMCode.c* e *CAMCode.h.* Os arquivos *.c* fazem referência as funções, enquanto os *.h* fazem referência ao *header* (cabeçalho). O *ApiCode* controla funções do FPAA, como a própria geração do arquivo de reconfiguração, enquanto o *CAMCode* apenas altera os parâmetros dos CAMs presentes no circuito a ser implementado.

Contudo, recomenda-se que antes de clicar em *Generate*, escolha-se a opção *CAM Functions*, onde será definido quais funções CAM deseja se controlar. O objetivo é criar um arquivo *CAMCode* mais enxuto, deixando o mais leve (para processadores com problemas de limitação de memória) e de fácil compreensão quando for ser utilizado. Para isso, basta seguir a instrução da Figura 36, deixando

marcado em *CAM Type* apenas o parâmetro que deseja-se controlar, que neste exemplo refere-se ao *AnadigmVortex\GainLimiter*.

Figura 35: Tela de geração dos arquivos de reconfiguração

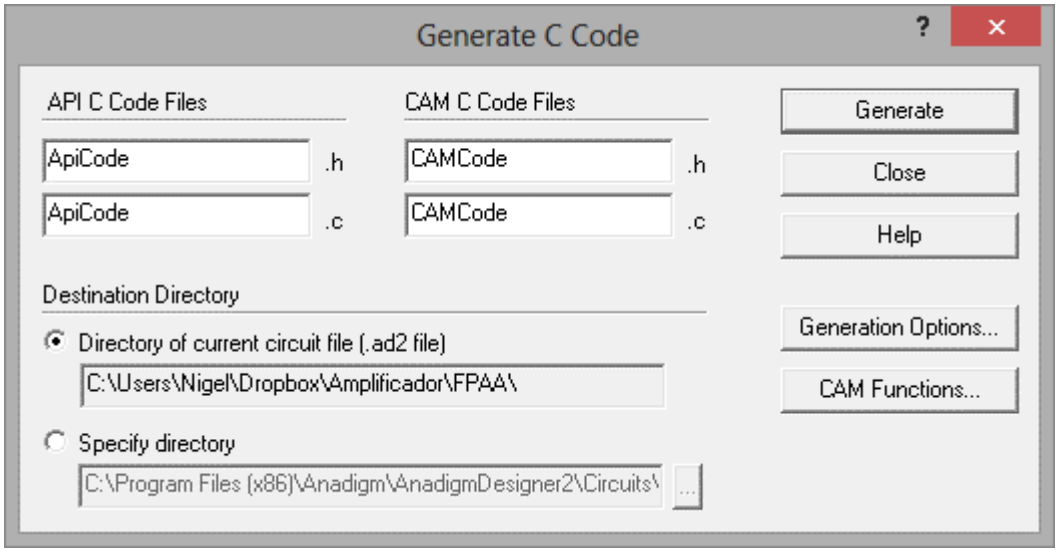
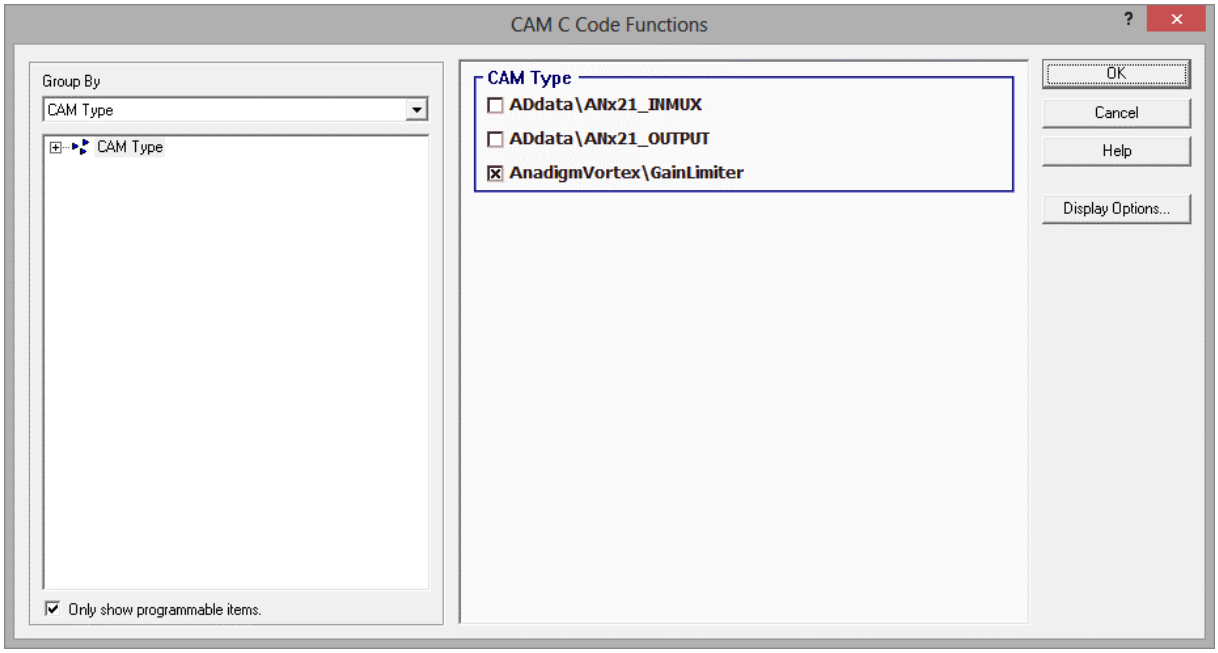


Figura 36: Tela de seleção dos parâmetros CAM a serem reconfigurados



Os comandos utilizados dessa sub rotina são:

- **an_GetPrimaryConfigData(an_Chip chip, int* count)**. Utilizado para gerar o arquivo de Configuração Inicial. O parâmetro *an_Chip chip* identifica o FPAA a ser configurado e *int* count* funciona como um ponteiro, recebendo o tamanho do

arquivo de reconfiguração. Este comando gera informação que precisa ser salva em um vetor do tipo *char*.

- **an_InitializeVortexReconfigData(an_Chip chip)**. Utilizado para inicializar o *buffer* de criação do arquivo de Reconfiguração. O parâmetro *an_Chip chip* identifica o FPAA a ser reconfigurado.

- **an_setGainLimiter(an_CAM cam, double G, double VL)**. Utilizado para configurar os parâmetros do CAM *GainLimiter.* O parâmetro *an_CAM cam* informa qual CAM de qual FPAA será reconfigurado, enquanto *double G* requer a informação do ganho a ser implementado e *double VL* é referente ao limite máximo de tensão de saída do CAM.

- **an_GetVortexReconfigData(an_Chip chip, int* count).** Utilizado para gerar o arquivo de Reconfiguração. O parâmetro *an_Chip chip* identifica o FPAA a ser configurado e *int* count* funciona como um ponteiro, recebendo o tamanho do arquivo de reconfiguração. Este comando gera informação que precisa ser salva em um vetor do tipo *char*.

- **an_ShutdownVortexReconfigData(chip_id)**. Utilizado para encerrar o *buffer* do arquivo de Reconfiguração. Utiliza-se para liberar memória no processamento.

Por fim, a rotina implementada faz o cálculo dos parâmetros necessários a serem reconfigurados e o controle das funções acima mencionadas. Para este exemplo, é necessário calcular o ganho necessário tendo em vista que o usuário irá informar o nível de tensão de saída desejado. O cálculo é bem simples, bastando dividir o valor de tensão desejado pelo número três. Esse número três deve-se ao fato de que caso o ganho do amplificar esteja ajustado para um, o valor de tensão de saída será de 3 V, devido ao valor de tensão da fonte implementada no circuito.

Para que o circuito entre em funcionamento, basta executar a rotina desenvolvida, fazendo com que o circuito inicie com a saída ajusta para 1 V, sofrendo alteração assim que o usuário informar o novo nível de tensão desejado.

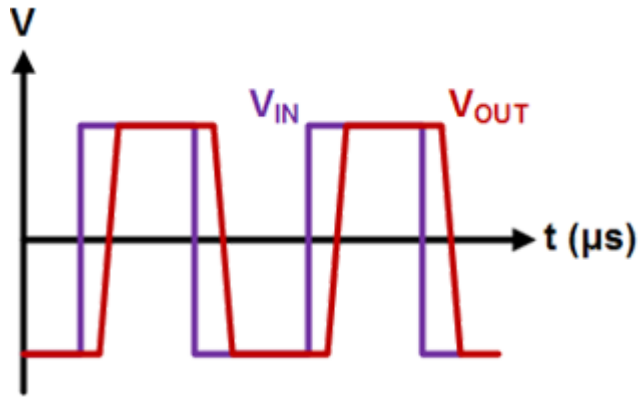
Maiores detalhes podem ser obtidos analisando-se o código em anexo ao fim deste trabalho.

5.1.4. Ensaios e Resultados

Para a verificação do funcionamento do sistema implementado, basta apenas monitorar a saída diferencial e verificar se o nível de tensão produzido condiz com o nível selecionado no programa. Contudo, além deste ensaio, aproveitou-se para fazer o levantamento de mais duas informações a respeito da implementação deste sistema, sendo elas o *Slew Rate* e o tempo de reconfiguração dinâmica.

O *Slew Rate* é definido como a máxima taxa de variação de tensão de saída por unidade de tempo, sendo normalmente expressão em *Volts* por segundo. A Figura 37 demonstra o efeito do *Slew Rate* na saída de um amplificador operacional quando submetido a necessidade de uma variação de tensão de saída em forma de onda quadrada.

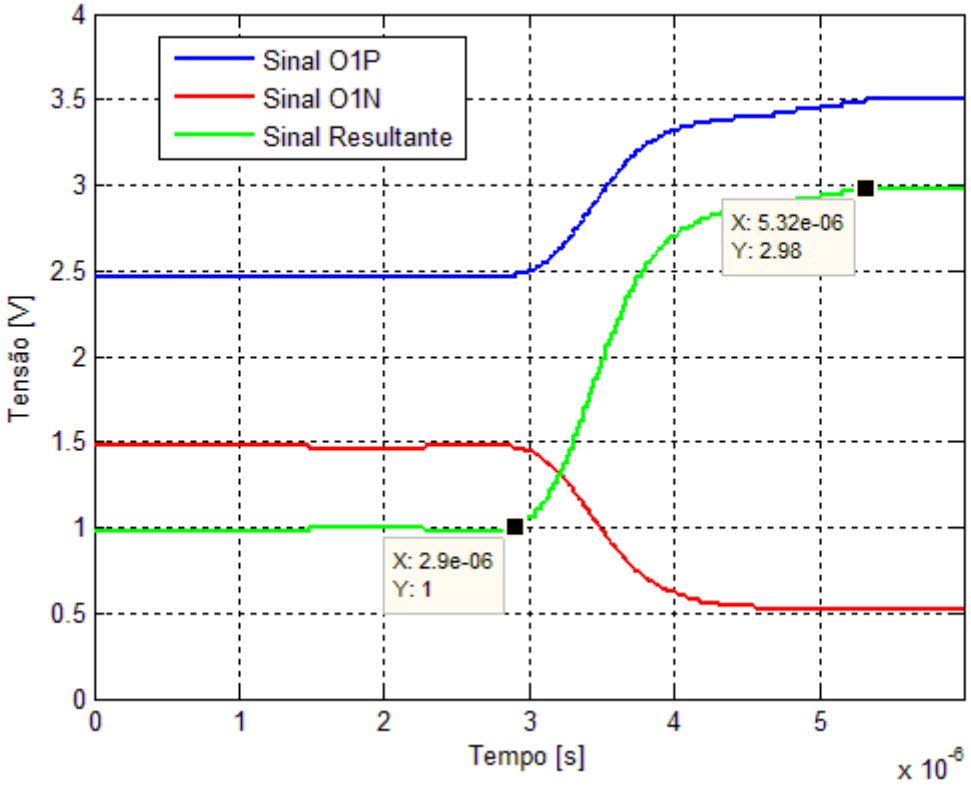
Figura 37: Slew Rate



Para a execução deste ensaio, utilizou-se um osciloscópio digital com duas ponteiras, sendo uma ligada a saída O1P e a outra ao O1N do FPAA, com ambas as ponteiras referenciadas ao terra. As informações coletadas foram exportadas em um arquivo do tipo .csv para possibilitar o adequado tratamento dos dados, gerando figuras de melhor visualização para apresentação dos dados obtidos.

A Figura 38 demonstra o resultado obtido ao se alterar a tensão desejada na saída do sistema de 1 V para 3 V.

Figura 38: Ensaio variando tensão de saída de 1 V para 3 V

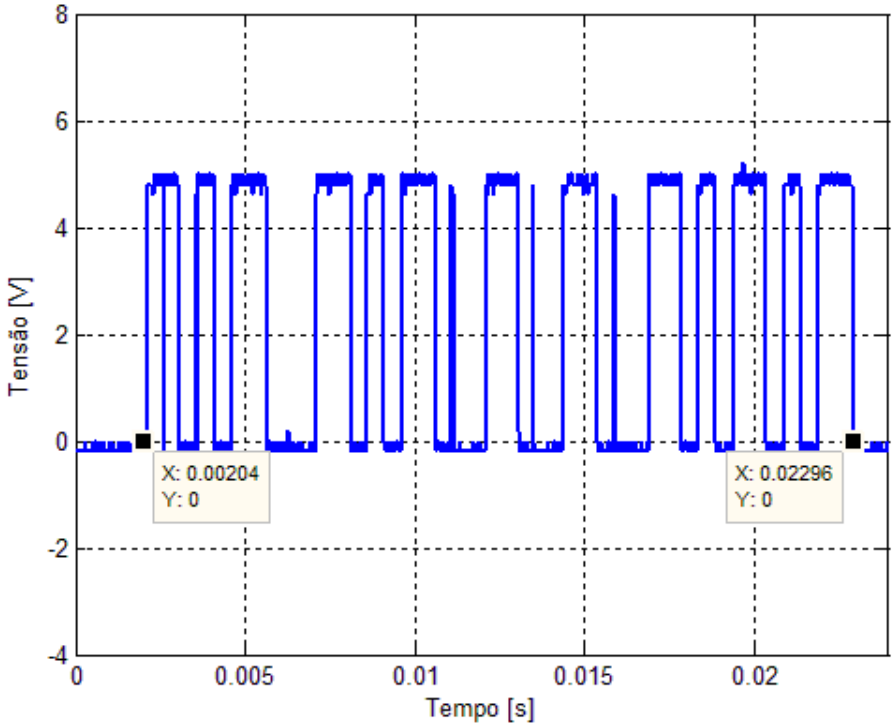


Como pode ser verificado, o sinal de saída do dispositivo é diferencial. O FPAA trabalha com o nível de tensão 2 V como referencial interno para a geração de sinais diferenciais. Pode-se verificar que o nível de tensão é alterado sem a interrupção do sinal de saída, o que indica que a reconfiguração dinâmica foi executada com êxito. A mesma imagem também nos permite verificar que para uma variação no sinal de saída de 2 V, o dispositivo levou o tempo de aproximadamente 2,42 µs, o que resulta em um *Slew Rate* de aproximadamente 0,83 V/µs. Apenas para efeito de comparação, o amplificador operacional LM741 costuma apresentar uma taxa de *Slew Rate* de 0,5 V/µs.

Para o ensaio de tempo de reconfiguração dinâmica, foram realizados dois ensaios. O primeiro monitorou-se apenas o sinal no pino *DIN* do FPAA, para verificar-se o tempo que os dados de reconfiguração demoram para serem transferidos para o FPAA. Na verificação do tempo de reconfiguração interna do FPAA, monitorou-se novamente o pino *DIN*, para verificar o término do envio dos

dados, e o pino O1P, para verificação de quando a saída foi alterada. Os resultados são demonstrados nas Figuras 39 e 40.

Figura 39: Ensaio do tempo de envio dos dados de reconfiguração



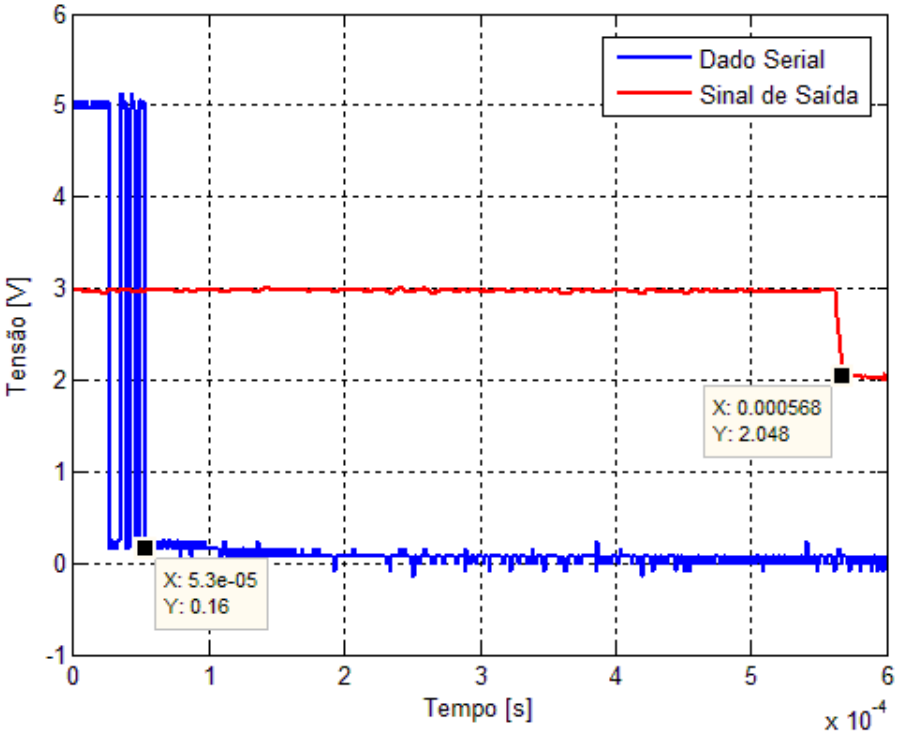
Neste ensaio, a comunicação serial estava ajustada para a velocidade de 57600 *bps*. Como pode ser observado na Figura 39, transferência dos dados via comunicação serial RS-232 levou aproximadamente 21 ms.

Já para o ensaio do tempo de reconfiguração dinâmica, o FPAA estava ajustado com o *clock* principal na frequência de 16 MHz, enquanto o *clock* interno do CAM estava ajustado para a frequência de 4 MHz.

Conforme demonstra a Figura 40, o tempo transcorrido do término do envio dos dados de reconfiguração dinâmica até o momento em que a saída sofreu alteração foi de aproximadamente 515 µs. Com isso pode-se concluir que o tempo total para reconfiguração dinâmica deste sistema leva quase 22 ms para ocorrer, sendo o envio dos dados o principal responsável por este valor. O tempo de envio dos dados serial está atrelado ao *baudrate* da comunicação serial, o qual não pode ser alterada devido ao sistema ABK que apenas funciona em 57600 *bps*.

Dessa forma, conclui-se que a reconfiguração dinâmica funcionou perfeitamente para este sistema, partindo-se para a implementação do próximo sistema, um controlador analógico PID.

Figura 40: Ensaio do tempo de reconfiguração dinâmica do FPAA



5.2. Sistema II – Circuito Controlador Analógico PID

O controlador PID (Proporcional – Integral – Derivativo) é capaz de eliminar erros de regime permanente através da sua ação integral, bem como antecipar o comportamento do processo, graças à ação derivativa. A ação proporcional, por sua vez, faz com que o sistema reaja ao erro presente, conferindo ao sistema uma reação imediata e, portanto, rápida à ação de perturbações ou variações de referência de magnitudes significativas. De certa maneira, as três ações presentes no controlador PID permitem emular, matematicamente, o comportamento da mente do operador ao controlar um processo manualmente (BAZZANELA, DA SILVA JR., 2005).

Com a conclusão da implementação do primeiro sistema, e consequentemente o conhecimento de funcionamento da técnica de reconfiguração

dinâmica, partiu-se para a implementação de um circuito controlador analógico PID. O objetivo desse circuito é que através do ajuste dinâmico dos parâmetros, seja possível fazer com que sinais de referência sejam seguidos com erro nulo e reduzindo o tempo necessário para que isso ocorra.

Para que fosse possível o ensaio deste sistema, foi necessário a utilização de um circuito eletrônico externo, visando simular um processo térmico simples. Para isso, utilizou-se um circuito RC série, o qual será melhor detalhado posteriormente.

5.2.1. Funcionamento

O objetivo deste sistema é de controlar o nível de tensão com que o capacitor de um circuito RC (Resistivo – Capacitivo) série deverá permanecer carregado, tomando como referência o nível de tensão ajustado no controlador, simulando um processo do tipo térmico.

O sistema é composto por um gerador de sinal de referência, um controlador analógico PID e um circuito RC. O gerador de sinal de referência fornece um sinal de tensão contínua variando entre 0,03 V e 3 V, igual ao sistema anterior. O controlador analógico PID é formado por três blocos, sendo um com ganho proporcional, um com ganho integral e o terceiro com ganho derivativo, conforme os circuitos de controladores PID. A topologia implementada foi a de um controlador PID paralelo. O sinal de saída do controlador é ligado ao circuito externo RC, sendo responsável pela carga do capacitor. Para que o controlador funcione adequadamente, o nível de tensão no capacitor é monitorado pelo controlador PID.

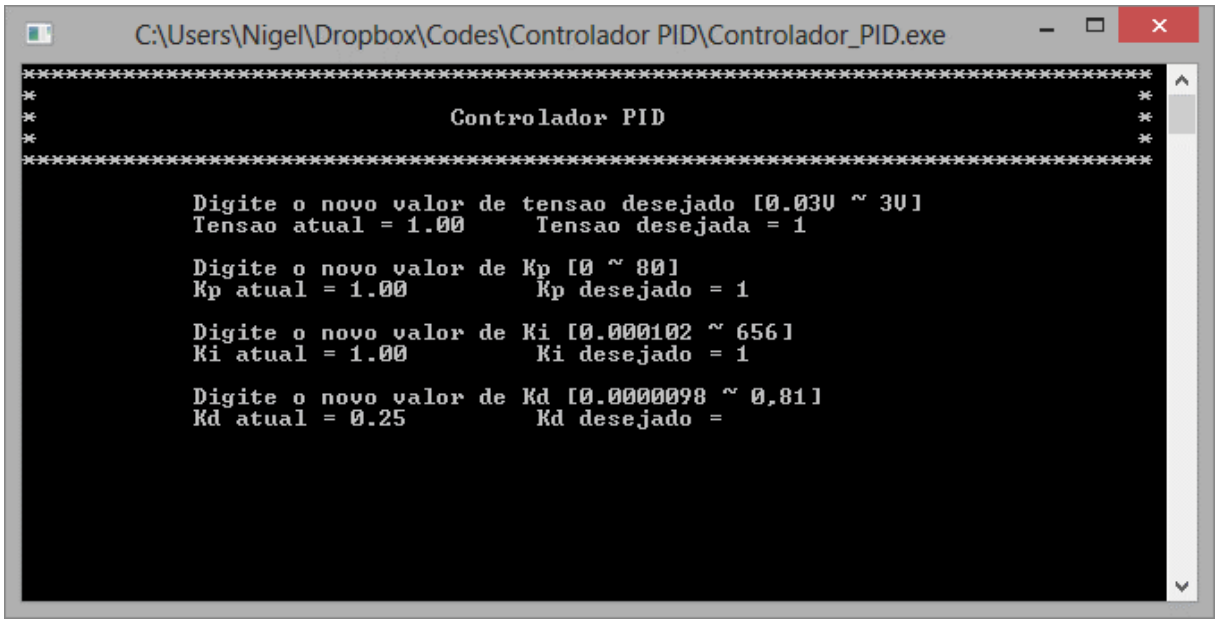
Além do ajuste da tensão de referência, este sistema permite a variação do ajuste dos ganhos dos controladores proporcional, *Kp*, integral, *Ki* e derivativo, *Kp*. Os valores que os ganhos podem atingir variam de acordo com o *clock* selecionado para a implementação do sistema, que neste exemplo foi adotado 16 MHz para o FPAA e 100 kHz para os CAMs. Além disso, devido a forma de implementação do sistema, os ganhos proporcional e derivativo estão relacionados um ao outro. Assim, deu-se prioridade para que seja ajustado o ganho proporcional, fazendo com que caso não seja possível implementar ambos os ganhos desejados, manter o ganho

proporcional e recalcular o ganho derivativo para o valor mais próximo possível do valor desejado. Respeitando essas limitações, as variações possíveis de ganho são:

- *Kp:* de 0 até 80;
- *Ki:* de 0,000102/µs até 656/µs;
- *Kd:* de 0,0000098 ms até 0,81 ms.

Ao ser iniciado, o sistema está ajustado para que a tensão de referência seja de 1 V, *Kp* = 1, *Ki* = 1/µs e *Kp* = 0,25 ms, conforme ilustra a Figura 41. Dessa forma, o capacitor será carregado até atingir a tensão de 1 V. Qualquer um dos parâmetros pode ser alterado a qualquer momento, onde a referência irá alterar o nível de tensão no capacitor e os demais parâmetros irão alterar a velocidade com que o nível de tensão seja atingido.

Figura 41: Tela de ajuste de parâmetros do Sistema II



5.2.2. Implementação

Para a implementação deste sistema, do ponto de vista do FPAA foram utilizados sete CAMs: um *Voltage*, um *GainLimiter*, dois *SumDiff,* um *Integrator,* um *GainHalf* e um *Sample and Hold.* Na saída foi ativado o filtro passa baixas.

O CAM *Voltage* implementa a fonte de 3 VDC e o *GainLimiter* implementa o amplificador de tensão, sendo estes iguais ao sistema anterior e gerando o sinal de referência.

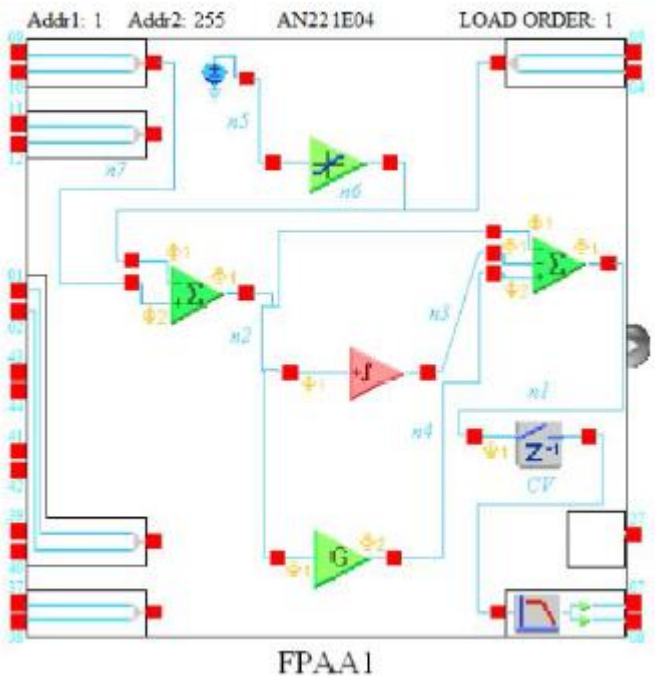
O primeiro CAM *SumDiff*, funciona como um circuito somador/subtrator. É utilizado para o cálculo do erro, diferença entre o valor de referência com o valor atual do processo. O segundo para a soma das componentes proporcional, integral e derivativa. O CAM *SumDiff* também possui ajuste de ganho, com isso, ele também faz parte da implementação dos ganhos dos controladores proporcional, integral e derivativo.

O controlador proporcional tem o seu ganho ajustado no CAM *SumDiff*, contudo o seu valor sofre interferência do ganho do controlador derivativo, conforme será abordado na sequência.

O controlador integral é composto pelo CAM *Integrator* e pelo ajuste de ganho realizado no *SumDiff* também. O CAM *Integrator* como o próprio nome já diz, implementa um circuito integrador.

O controlador derivativo é implementado de uma maneira diferente, visando minimizar a utilização de recursos. Como pode ser verificado na Figura 42, os sinais são executados na fase 1 ou 2 do *clock*. A fase 1 corresponde quando o *clock* está com o nível lógico em 1, enquanto a fase 2 corresponde ao *clock* com o nível lógico em 0. Como pode ser verificado, os caminhos do controlador proporcional e integral foram implementados de maneira que o sinal chegue ao CAM *SumDiff* na fase 1. Já o sinal do controlador derivativo passa pelo CAM *GainHalf* (circuito amplificador), que além de ajustar o ganho do sinal, provoca um atraso de fase, fazendo com o que o sinal chegue no *SumDiff* durante a fase 2. Este atraso, que está diretamente ligado ao *clock* dos circuitos acaba por se tornar o ganho derivativo.

Figura 42: Sistema II no AnadigmDesigner2



Em virtude dessa forma construtiva, os ganhos passam a ter a seguinte relação:

- *Kp = G1 – (G * G3);*
- *Ki = K * G2;*
- *Kd = (G * G3) / fc.*

G refere-se ao ganho do CAM *GainHalf, G1, G2* e *G3* referem-se aos ganhos no CAM *SumDiff*, *K* refere-se ao ganho do CAM *Integrator* e *fc* ao *clock* selecionado para os CAMs.

Por fim, tem-se o CAM *Sample and Hold* que apenas tem a função de manter o nível de tensão amostrado durante a fase 1 na saída durante a fase 2.

Contudo, para que fosse possível realizar o ensaio deste sistema, foi necessário a utilização de um circuito RC série externo para a simulação de um processo do tipo térmico. Um circuito RC série possui a sua função de transferência de acordo com a Equação 7.

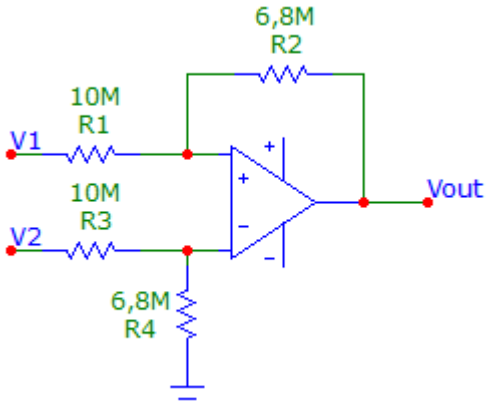
$$G(s) = \frac{(1/RC)}{s + (1/RC)} \tag{7}$$

Ao se adotar um resistor de 1 kΩ e um capacitor de 1 mF, a função de transferência passa a ser regida pela Equação 8.

$$G(s) = \frac{1}{s+1} \tag{8}$$

Porém, para que se possa ser notável a necessidade de um controlador para se atingir o nível de tensão desejado no capacitor, optou-se por fazer que o sistema tivesse um ganho em regime permanente menor do que 1. Para isso, utilizou-se um circuito amplificador subtrator diferencial com um ganho de 0,68. Este circuito, como demonstra a Figura 43, tem a função de implementar um ganho de 0,68, transformar o sinal diferencial proveniente do FPAA em um sinal referenciado ao terra e de isolar o circuito externo do FPAA, funcionando como um *buffer*. Isso se faz necessário devido à baixa capacidade de fornecer corrente diretamente dos operacionais internos do FPAA.

Figura 43: Amplificador subtrator diferencial

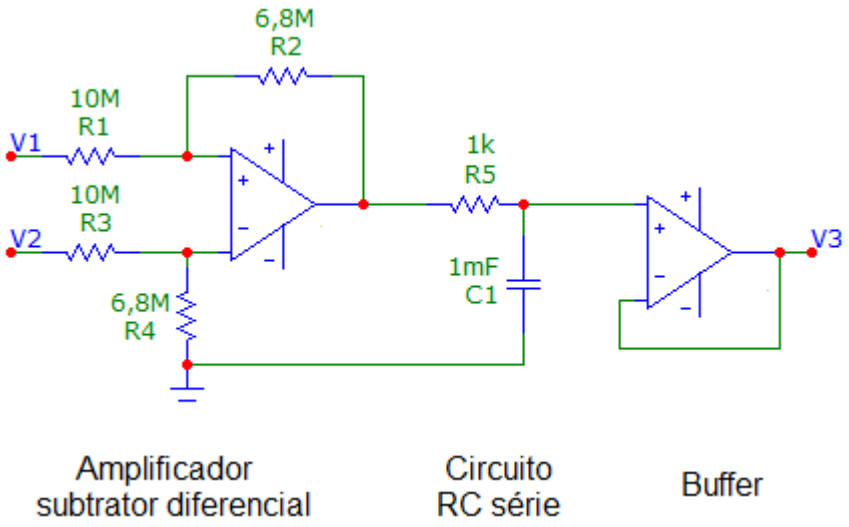


Considerando que o ganho implementado pelo circuito acima seja parte do processo a ser simulado a função de transferência do processo passa a ser regida pela Equação 9:

$$G(s) = \frac{0{,}68}{s+1} \tag{9}$$

Para que o FPAA possa fazer a leitura do nível de tensão com que o capacitor está carregado, é necessário a utilização de um circuito de *Buffer*, visando a proteção do FPAA. Desta forma, o circuito externo completo utilizado é o demonstrado na Figura 44.

Figura 44: Circuito externo utilizado para o Sistema II



O CI utilizado para os amplificadores operacionais foi o LM324 e sua alimentação foi realizada com 5 VDC. Essa alimentação teve como objetivo limitar a tensão de alimentação máxima do capacitor, simulando a limitação do atuador de um processo térmico.

No que diz respeito as ligações do FPAA ao circuito externo, as ligações foram realizadas da seguinte forma:

- Leitura do valor de tensão no capacitor: pino I1P (pino 9) do FPAA no V3 do circuito externo e pino I1N (pino 10) ao terra;

- Monitoramento do sinal de referência: pino O1P (pino 3) e pino O1N (pino 4);

- Saída do sinal do controlador ao processo: pino O2P (pino 7) ao V1 e pino O2N (pino 8) ao V2.

5.2.3. Código para Reconfiguração Dinâmica

Assim como no sistema anterior, é necessário o desenvolvimento de uma rotina em linguagem C para que o FPAA possa ser reconfigurado dinamicamente. A rotina implementada segue o mesmo padrão da rotina utilizada para o Sistema I, sofrendo alteração apenas na parte do código que realiza o cálculo dos parâmetros de reconfiguração do circuito e na geração dos arquivos de reconfiguração.

Novamente é necessário a utilização das funções utilizadas pelos CAMs no sistema implementado para a geração dos arquivos de reconfiguração. O processo de geração dos arquivos *ApiCode.c, ApiCode.h, CAMCode.c* e *CAMCode.h* mantem-se o mesmo, sendo necessário apenas revisar a tela de seleção dos parâmetros CAM a serem reconfigurados, devendo ser configurado conforme ilustra a Figura 45.

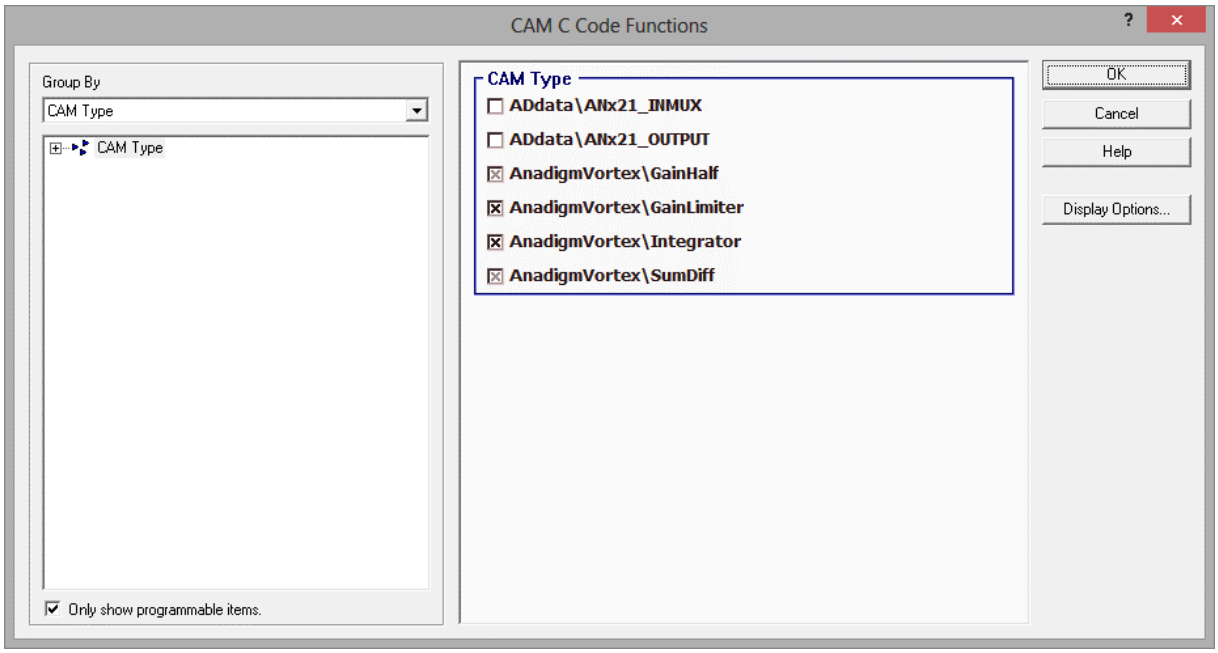
Assim, além das funções já utilizadas no Sistema I que continuaram a serem utilizadas, este novo sistema também utiliza as seguintes funções:

- **an_setGainHalf(an_CAM cam, double G)**. Utilizado para configurar os parâmetros do CAM *GainHalf.* O parâmetro *an_CAM cam* informa qual CAM de qual FPAA será reconfigurado, enquanto *double G* requer a informação do ganho a ser implementado.

- **an_setIntegrator(an_CAM cam, double K)**. Utilizado para configurar os parâmetros do CAM *Integrator.* O parâmetro *an_CAM cam* informa qual CAM de qual FPAA será reconfigurado, enquanto *double K* requer a informação da constante de integração a ser implementada.

- **an_setGainSumDiff_3in(an_CAM cam, double G1, double G2, double G3)**. Utilizado para configurar os parâmetros do CAM *SumDiff.* O parâmetro *an_CAM cam* informa qual CAM de qual FPAA será reconfigurado, enquanto *double G1, double G2* e *double G3* requerem a informação dos ganhos a ser implementados.

Figura 45: Tela de seleção dos parâmetros CAM a serem reconfigurados



Concluída a etapa de geração das funções de controle dos CAMs, a rotina em linguagem C do Sistema I foi alterada para a realização dos cálculos dos parâmetros necessários, conforme explicação na etapa de implementação, e utilização das funções acima descritas para a geração do código de reconfiguração dinâmica do FPAA. Para maiores detalhes, o código encontra-se em anexo ao fim deste trabalho.

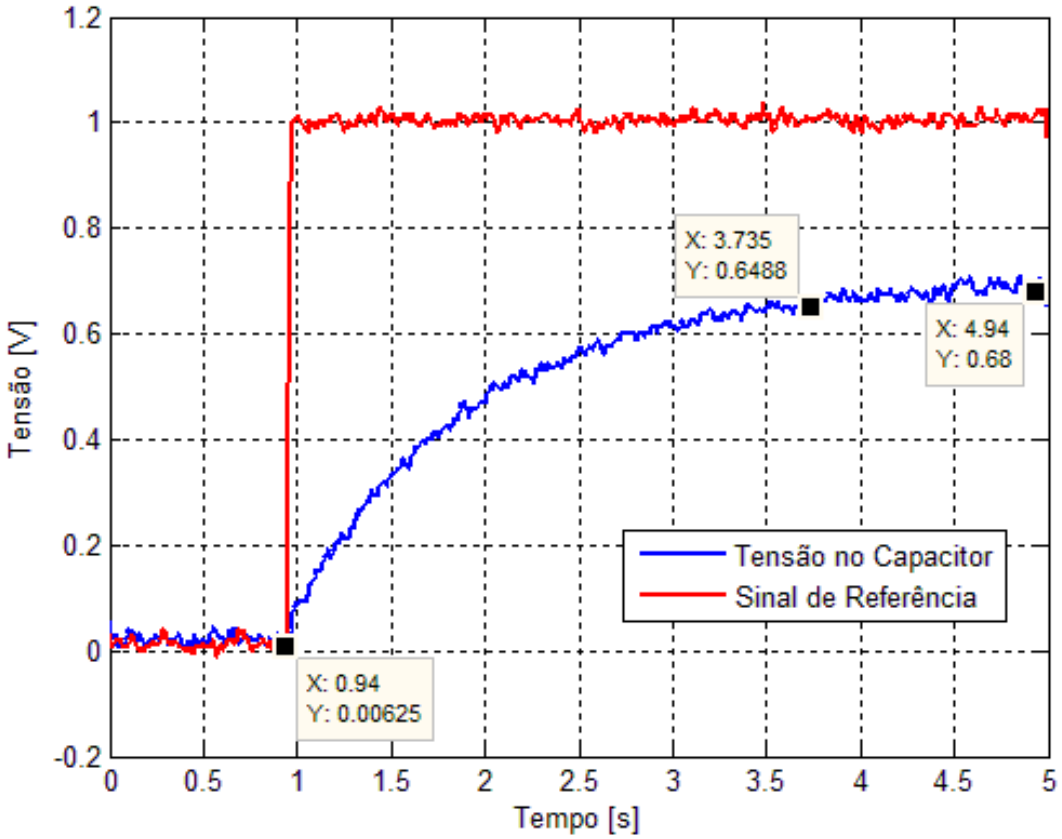
Para que o circuito entre em funcionamento, basta executar a rotina desenvolvida, fazendo com que o circuito inicie com o ajuste da tensão de referência em 1 V e com os valores dos ganhos *Kp* = 1, *Ki* = 1 e *Kd* = 0,25. Qualquer um dos parâmetros acima podem ser alterados a qualquer momento, sem a interrupção do sinal de saída.

5.2.4. Ensaios e Resultados

Foram realizados ensaios para verificar o correto funcionamento do sistema de controle implementado, assim como a mudança de comportamento conforme alteração no sinal de referência ou parâmetros do controlador.

Para verificar o comportamento do processo simulado, realizou-se um ensaio ajustando a tensão de referência em 1 V e os ganhos *Kp* = 1, *Ki* = 0 e *Kd* = 0. Essa simulação seria o equivalente a realizar um ensaio em malha aberta no processo a ser controlado. O resultado é demonstrado na Figura 46.

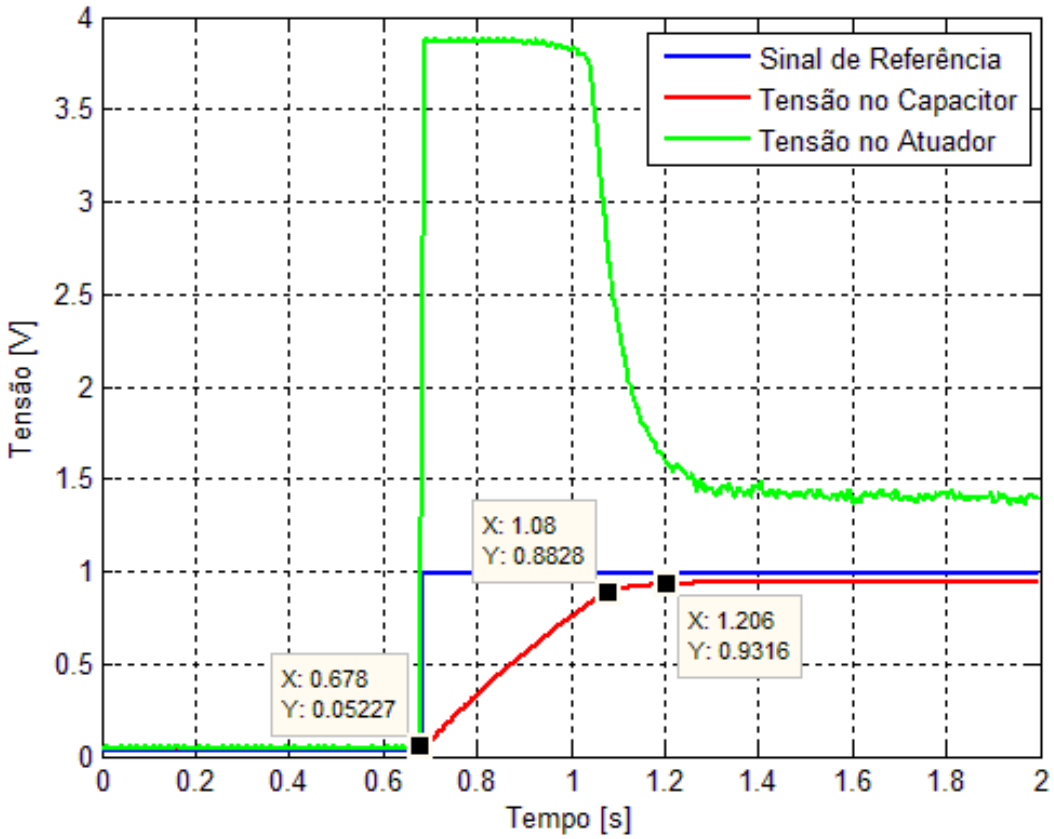
Figura 46: Ensaio em malha aberta



Como pode ser observado, o capacitor não consegue ser carregado com a tensão de 1 V, atingindo aproximadamente o valor máximo de 0,68 V. Isto deve-se ao fato do processo possuir ganho DC de 0,68, conforme mencionado anteriormente. Observa-se também que o tempo para atingir este nível de tensão é de aproximadamente quatro segundos.

Realizou-se novo ensaio, alterando apenas o valor do ganho proporcional, fazendo *Kp* = 25. A Figura 47 demonstra o resultado.

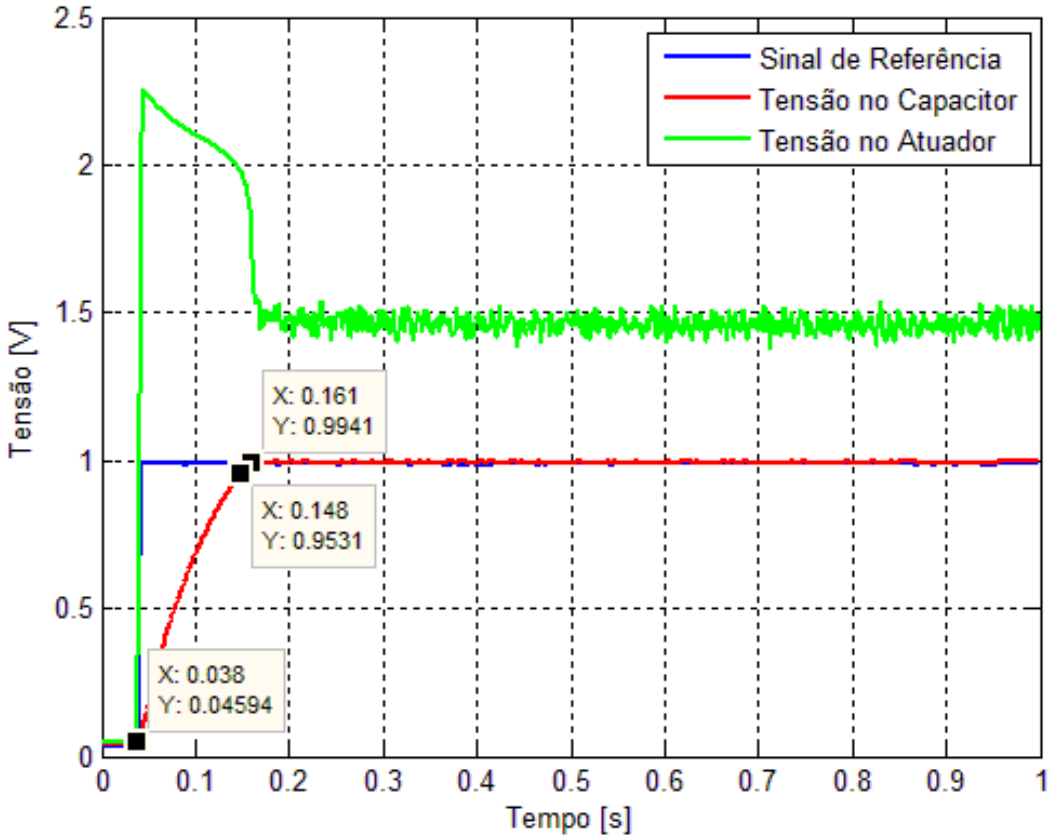
Figura 47: Ensaio com Kp = 25, Ki = 0 e Kd = 0



Como pode ser verificado, a tensão do capacitor chegou bem próximo a tensão de referência, atingindo o nível de 0,931 V. Porém, por mais que se aumente o ganho proporcional, nunca irá se obter erro nulo, pois esta é uma característica do controlador proporcional para referências do tipo salto. Contudo, pode-se verificar que o tempo de carga do capacitor foi reduzido, tendo sua tensão máxima atingida em aproximadamente 528 ms.

Para atingir o erro nulo, realizou-se outro ensaio em que o ganho integral foi ativado. O ajuste da tensão de referência permaneceu em 1 V e os ganhos configurados foram *Kp* = 0,26 e *Ki* = 0,5. O resultado é apresentado na Figura 48.

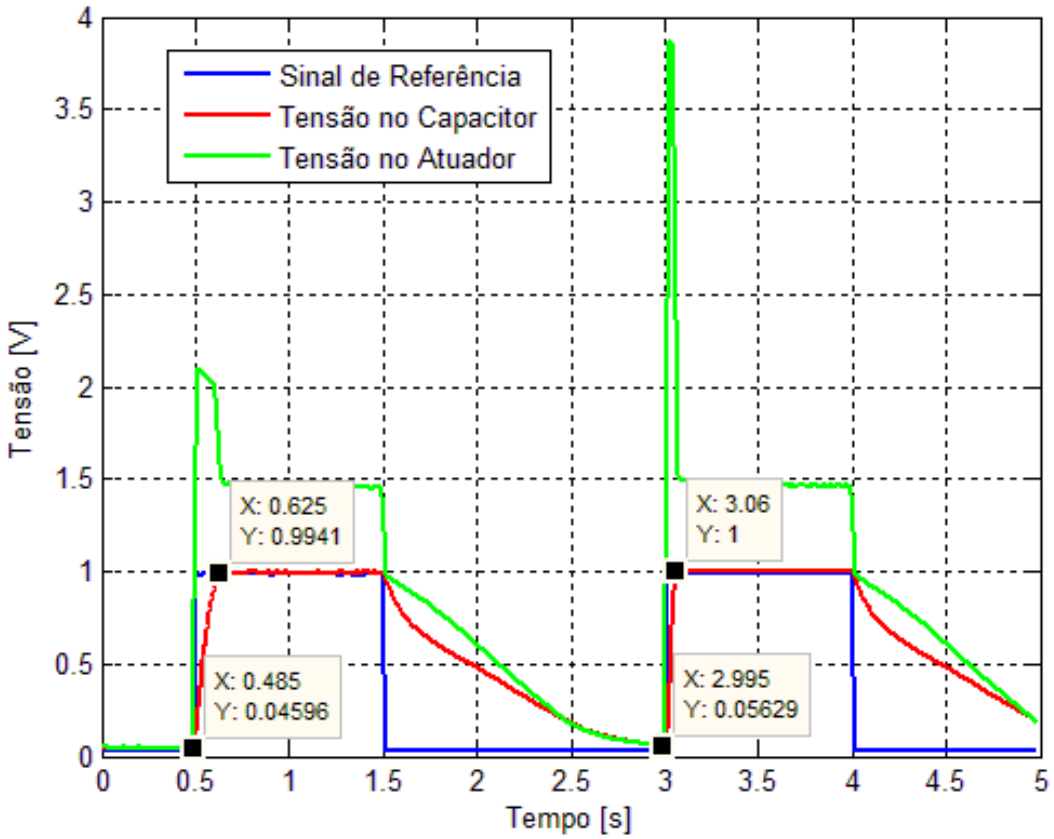
Figura 48: Ensaio com Kp = 0,26, Ki = 0,5 e Kd = 0



Como previsto, a tensão no capacitor atingiu o nível de 1 V, o que significa que o controlador atingiu erro nulo. Verifica-se também que devido a escolha dos parâmetros para o controlador, o tempo para estabilização do sistema reduziu, atingindo aproximadamente 120 ms.

Para finalizar, realizou-se um último teste, onde a tensão de referência é ajustada para 1 V e permanece até que o capacitor obtenha a tensão de 1 V, depois a tensão é alterada para 0 V e somente ajustada para 1 V novamente após a descarga do capacitor. No primeiro ciclo de carga do capacitor, os parâmetros do controlador são Kp = 0,26, Ki = 0,5 e Kd = 0. Para o segundo ciclo, os parâmetros foram alterados para Kp = 0,26, Ki = 200 e Kd = 0,05. O resultado é demonstrado na Figura 49.

Figura 49: Ensaio com variação dos parâmetros durante funcionamento



Como pode ser analisado, em ambos os casos o capacitor atingiu a tensão de referência de 1 V, contudo o tempo para que isso ocorra passou de aproximadamente 120 ms para aproximadamente 65 ms. Importante observar o esforço realizado pelo atuador, que inclusive chegou a saturar, ou seja, atingiu o seu limite de tensão. Caso o atuador tivesse a capacidade de um nível de tensão maior, o tempo para atingir a referência seria menor ainda.

Assim, conclui-se que o controlador PID analógico funciona de forma satisfatória.

6. Conclusão

Circuitos digitais sempre foram caracterizados por serem mais fáceis e rápidos de serem implementados que circuitos analógicos. Com o advento do *Field Programmable Analog Array*, este cenário tende a ser modificado.

Com o avanço da tecnologia, além da ideia principal de tornar simples e rápido o desenvolvimento de circuitos analógicos, os FPAAs passaram a apresentar uma nova função revolucionária no domínio analógico: a capacidade de reconfiguração dinâmica.

Por se tratar de uma tecnologia relativamente nova, não há muitos trabalhos a respeito do assunto. Com isso, este trabalho apresentou como objetivo a apresentação de uma solução para a implementação da técnica de reconfiguração dinâmica, visando auxiliar possíveis estudos futuros sobre o assunto.

Neste trabalho utilizou-se a técnica de *Algorithmic Method* para a demonstração de dois sistemas em funcionamento: um simples circuito amplificador de tensão não inversor e um circuito controlador analógico PID. Como verificado nos resultados obtidos, pode-se comprovar o correto funcionamento da técnica de reconfiguração dinâmica, verificando-se que este dispositivo pode oferecer soluções interessantes. No caso de controladores PID, permite a implementação de *hardwares* embarcados que ocupam pouco espaço físico e com baixo consumo de energia, podendo ser alternativa para alguns sistemas digitais.

Para trabalhos futuros, sugere-se a continuidade dos estudos desta técnica, permitindo que sistemas mais robustos sejam implementados, incluindo a implementação de *hardwares* auto calibráveis (*self tunning*) e auto recuperáveis (*self recovering*).

7. Referências

[1] GULAK, P. G. **Field-Programmable Analog Arrays: Past, Present and Future Perspectives.** In: IEEE Region 10 International Conference on Microelectronics and VLSI. Proceedings. Hong Kong, China. November 6 – 10, 1995, p. 123 – 126.

[2] LITA, IOAN; VISAN, DANIEL ALEXANDRU; CIOC, ION BOGDAN. **FPAA Based PID Controller with Applications in the Nuclear Domain.** In: 32nd International Spring Seminar on Electronics Technology. Conference Proceedings. Brno, Czech Republic. May 13 – 17, 2009.

[3] BALEN, TIAGO ROBERTO. **Teste de Dispositivos Analógicos Programáveis (FPAAs)**. 2006. 127 f. Dissertação (Mestrado em Engenharia Elétrica) – Universidade Federal do Rio Grande do Sul, Porto Alegre. 2006.

[4] SANTOS, ADELINO DE OLIVEIRA. **Aplicação de Dispositivos Analógicos Reconfiguráveis.** 2010. 121 f. Dissertação (Mestrado em Engenharia Eletrotécnica e de Computadores) – Instituto Superior de Engenharia do Porto, Porto, Portugual. 2010.

[5] HALL, TYSON S. **Field-Programmable Analog Arrays: a Floating-Gate Approach**. 2004. 115 f. Tese (Doutorado em Engenharia Elétrica e de Computação) – Georgia Institute of Technology, Atlanta, Estados Unidos. 2004.

[6] ECKSTEIN, KERSTIN; MÖHRINGER, PETER. **Dynamically Reconfiguring of the Field Programmable Analogue Array AN221E04**. Disponível em: http://www.fh-sw.de/sw/fachb/et/labinfo/v/vundv/fpaa-xxii.pdf

[7] DAMIANI, FELIPE RAFAEL DE OLIVEIRA. **Proposta de Protótipo de um Oxímetro de Pulso Empregando Tecnologia FPAA.** 2010. 79 f. Dissertação (Graduação em Engenharia Elétrica com ênfase em Eletrônica) – Universidade de São Paulo, São Carlos. 2010.

[8] SELOW, ROBERTO; LOPES, HEITOR S.; LIMA, CARLOS R. ERIG. **A Comparison of FPGA and FPAA Technologies for a Signal Processing Application**. In: 19th International Conference on Field Programmable Logic and Applications – IEEE. Czech Republic. August 31 – September 2, 2009, p. 230 – 235.

[9] LATTICE. **Programmble Analog Circuits: ispPAC handbook**. Hilsboro, USA: Lattice Semiconductor Corporation, 2000.

[10] BRATT, A.; MACBETH, I. DPAD2 – A Field Programmable Analog Array. **Analog Integrated Circuits and Signal Processing:** special issue on field programmable analog arrays, Dordrecht, The Netherlands: Kluwer Academic Publishers. V. 17, n 1 – 2, p. 67 – 89.

[11] HAYKIN, S.; VAN VEEN, B. **Sinais e Sistemas**. Rio de Janeiro, Brasil: Bookman do Brasil, 2001.

[12] MORAES, FERNANDO; MESQUITA, DANIEL. **Tendências em reconfiguração dinâmica de FPGA.** 2001. Disponível em: http://www.inf.pucrs.br/moraes/my_pubs/papers/2001/2001_scr2001_palestra.pdf

[13] ANADIGM COMPANY. **AN121E04/AN221E04 Field Programmable Analog Arrays – User Manual.** 2003. Disponível em: http://www.anadigm.com/_doc/um021200-u007.pdf

[14] ANADIGM COMPANY. **Anadigmvortex AN221D04 Evalution Board – User Manual.** 2003. Disponível em: http://www.anadigm.com/_doc/UM030300-U008.pdf

[15] ANADIGM COMPANY. **AnadigmDesigner2 – User Manual.** 2004. Disponível em: http://www.anadigm.com/_doc/UM020800-U001.pdf

[16] ANADIGM COMPANY. **Design Brief 205 – Understanding the Anadigm Boot Kernel (ABK).** 2003. Disponível em: http://www.anadigm.com/_apps/DB020800-U205.pdf

[17] BAZANELLA, ALEXANDRE SANFELICE; DA SILVA JR., JOÃO MANOEL GOMES. **Sistemas de Controle – princípios e métodos de projeto.** Porto Alegre, Brasil. Editora da UFRGS, 2005.

Apêndice I – Código Implementado Para Sistema I

```
////////////////////////////////////////////////////////////////////////////
// Amplificador.c : Código para implementação de Reconfiguração Dinâmica
//                  de um circuito amplificador não inversor no FPAA
//                  AN221E04 da Anadigm Company
//
//                   Code for Dynamic Reconfiguration of a non inversor
//                  amplifier circuit on FPAA AN221E04 from Anadigm Company
//
// Autor/Author: Nigel Foster
// Data/Date: 12/10/2014
//
// Notas: Este software deve ser usado em conjunto com os arquivos gerados
// pelo software AnadigmDesigner2.
// Estes arquivos estão com os nomes padrões de ApiCode.c, ApiCode.h,
// CAMCode.c and CAMCode.h.
// Este software usa um código de http://www.teuniz.net/RS-232/ desenvolvido
// por Teunis van Beelenwas. A versão usada é de 31 de Janeiro de 2014 e os
// arquivos utilizados são rs232.c e rs232.h
//
// Notes: This software should be used in conjunction with the software
// files generated by AnadigmDesigner2.
// These files have been given the default names ApiCode.c, ApiCode.h,
// CAMCode.c and CAMCode.h.
// This software uses a code from http://www.teuniz.net/RS-232/ by Teunis
// van Beelenwas. The version used is from January 31, 2014 and the files
// used are rs232.c and rs232.h
////////////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "rs232.h"
#include "ApiCode.h"
#include "CAMCode.h"
#include <Windows.h>

/*****************************************************************************
* Declaração de variáveis                                                    *
*****************************************************************************/
const an_Byte* PriConfigData;   //ponteiro para a configuração primária
const an_Byte* ReconfigData;    //ponteiro para a configuração de atualização
float gain,tensao;              //ganho do CAM e Tensão desejada
int i, n,n1,StreamSize,tamanho[4], //variavéis usadas nas funções
cport_nr=5,                     //variável que define a porta COM utilizada. COM6
bdrate=57600;                   //ajuste do BaudRate utilizado
unsigned char buf[4096];        //define tamanho do buffer serial

/*****************************************************************************
* Função: PrimaryConfig                                                      *
*                                                                            *
* Propósito: Esta função chama a rotina ApiCode.C para receber, preparar e   *
* enviar os dados para o FPAA. Faz uso da rotina RS232.c para envio dos dados. *
*                                                                            *
* Entrada: unsigned char chip_id - FPAA a ser configurado.                   *
*                                                                            *
* Saída: Nenhuma, mas o FPAA será configurado de acordo com a configuração   *
```

```
* primária.                                                                  *
*****************************************************************************/
void PrimaryConfig(an_Byte chip_id)
{
      PriConfigData = an_GetPrimaryConfigData(chip_id, &StreamSize);
//função da rotina ApiCode.c que gera os dados da configuração primária

      //converte o tamanho do bloco de dados de número inteiro para 4 caracteres
      //ASCII (word)
n=(StreamSize+7)*2;
      if(n>255){
            n1=n/256;
            tamanho[0]=n1/16;
            tamanho[1]=n1-(tamanho[0]*16);
            n=n-(n1*256);
            tamanho[2]=n/16;
            tamanho[3]=n-(tamanho[2]*16);
            }
      else{
            tamanho[0]=0;
            tamanho[1]=0;
            tamanho[2]=n/16;
            tamanho[3]=n-(tamanho[2]*16);
            }
      for(i=0;i<4;i++){
      tamanho[i]=tamanho[i]+48;
      if(tamanho[i]>57){
            tamanho[i]=tamanho[i]+7;
            }
      }

      //converte as informações do bloco de dados de unsigned char para caracteres
      //ASCII
      n=StreamSize;
      int temp[n],dados[2*n],j=0;
       for(i=0;i<n;i++){
      temp[i]=PriConfigData[i];
        }
      for(i=0;i<n;i++){
      dados[j]=temp[i]/16;
      dados[j+1]=temp[i]-(dados[j]*16);
      dados[j]=dados[j]+48;
      dados[j+1]=dados[j+1]+48;
      if(dados[j]>57){
            dados[j]=dados[j]+7;
      }
      if(dados[j+1]>57){
            dados[j+1]=dados[j+1]+7;
      }
      j=j+2;
      }

      //envia o comando de configuração do FPAA e cinco pads (0x00) antes dos
      //dado os cinco pads são necessários para que o FPAA já esteja inicializado
      //antes dos dados de configurações começarem a ser enviados.
      //OBS: cada informação transmitida requer dois caracteres ASCII
      RS232_SendByte(cport_nr, 0x02); //STX - indica início de comunicação com o
                                      //FPAA
```

```
    RS232_SendByte(cport_nr, 0x30); //comando para configuração que realiza reset
                                    //ao final

    //envia o tamanho do bloco de dados (contando os pads)
             for(i=0;i<4;i++){
       RS232_SendByte(cport_nr, tamanho[i]);
       }

       RS232_SendByte(cport_nr, 0x30); //início dos cinco pads
       RS232_SendByte(cport_nr, 0x30);
       RS232_SendByte(cport_nr, 0x30);
       RS232_SendByte(cport_nr, 0x30);
       RS232_SendByte(cport_nr, 0x30);
       RS232_SendByte(cport_nr, 0x30);
       RS232_SendByte(cport_nr, 0x30);
       RS232_SendByte(cport_nr, 0x30);
       RS232_SendByte(cport_nr, 0x30);
       RS232_SendByte(cport_nr, 0x30);


    //envia o bloco de configuração
       for(i=0;i<2*StreamSize;i++){
             RS232_SendByte(cport_nr, dados[i]);
       }

        //envia dois pads (0x00) antes do término dos dados de configuração e na
        //sequencia envia o comando de fim de comunicação
        //os dois pads são necessários para garantir que a configuração esteja
salva
        //antes do dispositivo reiniciar.
    RS232_SendByte(cport_nr, 0x30); //início dos dois pads
    RS232_SendByte(cport_nr, 0x30);
    RS232_SendByte(cport_nr, 0x30);
    RS232_SendByte(cport_nr, 0x30);
       RS232_SendByte(cport_nr, 0x03);  //ETX - indica o término da comunicação
com
                                        //o FPAA
}

/*************************************************************************
* Função: Configure                                                      *
*                                                                        *
* Propósito: Esta função chama a rotina ApiCode.C e CAM.c para configurar os  *
* parâmetros nas CAMs e receber, preparar e enviar os dados para o FPAA.      *
* Faz uso da rotina RS232.c para envio dos dados.                        *
*                                                                        *
* Entrada: unsigned char chip_id - FPAA a ser reconfigurado.             *
*                                                                        *
* Saída: Nenhuma, mas o FPAA será configurado de acordo com a configuração    *
* primária.                                                              *
*************************************************************************/
void Configure(an_Byte chip_id)
{
       an_InitializeVortexReconfigData(chip_id);
       //inicializa o buffer de dados
    an_setGainLimiter(an_FPAA1_GainLimiter1, gain, 3);
       //configura os parâmetros da CAM
    ReconfigData = an_GetVortexReconfigData(chip_id,&StreamSize);
//gera os arquivos de reconfig
```

```
  //converte o tamanho do bloco de dados de número inteiro para 4 caracteres
  //ASCII (word)
  n=(StreamSize+2)*2;
  if(n>255){
        n1=n/256;
        tamanho[0]=n1/16;
        tamanho[1]=n1-(tamanho[0]*16);
        n=n-(n1*256);
        tamanho[2]=n/16;
        tamanho[3]=n-(tamanho[2]*16);
        }
  else{
        tamanho[0]=0;
        tamanho[1]=0;
        tamanho[2]=n/16;
        tamanho[3]=n-(tamanho[2]*16);
        }
  for(i=0;i<4;i++){
  tamanho[i]=tamanho[i]+48;
  if(tamanho[i]>57){
        tamanho[i]=tamanho[i]+7;
        }
  }

  //converte as informações do bloco de dados de unsigned char para caracteres
  //ASCII
  n=StreamSize;
int temp[n],dados[2*n],j=0;
for(i=0;i<n;i++){
  temp[i]=ReconfigData[i];
   }
  for(i=0;i<n;i++){
  dados[j]=temp[i]/16;
  dados[j+1]=temp[i]-(dados[j]*16);
  dados[j]=dados[j]+48;
  dados[j+1]=dados[j+1]+48;
  if(dados[j]>57){
        dados[j]=dados[j]+7;
  }
  if(dados[j+1]>57){
        dados[j+1]=dados[j+1]+7;
  }
  j=j+2;
  }

  //envia o comando de configuração do FPAA e cinco pads (0x00) antes dos
  //dados os cinco pads são necessários para que o FPAA já esteja inicializado
  //antes dos dados de configurações começarem a ser enviados.
  //OBS: cada informação transmitida requer dois caracteres ASCII
  RS232_SendByte(cport_nr, 0x02); //STX - indica início de comunicação com o
                                  //FPAA
RS232_SendByte(cport_nr, 0x31); //comando para reconfiguração sem reset ao
                                //final

  //envia o tamanho do bloco de reconfiguração
        for(i=0;i<4;i++){
  RS232_SendByte(cport_nr, tamanho[i]);
  }
```

```
    //envia o bloco de reconfiguração
      for(i=0;i<2*StreamSize;i++){
            RS232_SendByte(cport_nr, dados[i]);
      }

        //envia dois pads (0x00) antes do término dos dados de configuração e na
       //sequência envia o comando de fim de comunicação
      RS232_SendByte(cport_nr, 0x30); //início dos dois pads
    RS232_SendByte(cport_nr, 0x30);
    RS232_SendByte(cport_nr, 0x30);
    RS232_SendByte(cport_nr, 0x30);
      RS232_SendByte(cport_nr, 0x03);  //ETX - indica o término da comunicação
                                       //com o FPAA

      an_ShutdownVortexReconfigData(chip_id); // encerra o buffer de dados
}

/*****************************************************************************
* Função: main                                                               *
*                                                                            *
* Propósito : Função principal do programa. Fica em loop eterno lendo o valor *
* de tensão desejado pelo usuário, calculando o ganho necessário e           *
* reconfigurando o FPAA.                                                     *
*                                                                            *
* Entrada: -                                                                  *
*                                                                            *
* Saída: -                                                                   *
*****************************************************************************/

void main()
{

//inicializa a comunicação serial
   if(RS232_OpenComport(cport_nr, bdrate))
  {
    printf("Can not open comport\n");
  }

  PrimaryConfig(an_FPAA1); //chama a rotina para o envio da configuração primária
  tensao=1; //para que apareça o valor de tensão da configuração primária

  while(1){ //loop infinito
            system("cls");  //limpar a tela

      printf("************************************************************\n");
      printf("*                                                          *\n");
      printf("*                Programa de Ajuste de Tensao              *\n");
      printf("*                                                          *\n");
      printf("************************************************************\n\n");
      printf("         Digite o novo valor de tensao desejado [0.03V ~ 3V]\n");
      printf("         Tensao atual = %.2f",tensao);
      printf("         Tensao desejada = ");
      scanf("%f",&tensao);
      //verifica se a tensão deseja está dentro dos limites permitidos
      if(tensao<0.03){
            tensao=0.03;
      }else if(tensao>3){
            tensao=3;
      }
```

```
      gain=tensao/3; //calcula o ganho
      Configure(an_FPAA1); //chama a rotina para a reconfiguração
    }
}
```

Apêndice II – Código Implementado para Sistema II

```
////////////////////////////////////////////////////////////////////////////
// Controle PID.c:  Código para implementação de Reconfiguração Dinâmica
//                  de um circuito controlador PID no FPAA AN221E04 da Anadigm
//                  Company
//
//                   Code for Dynamic Reconfiguration of a PID controller
//                   on FPAA AN221E04 from Anadigm Company
//
// Autor/Author: Nigel Foster
// Data/Date: 14/10/2014
//
// Notas: Este software deve ser usado em conjunto com os arquivos gerados
// pelo software AnadigmDesigner2.
// Estes arquivos estão com os nomes padrões de ApiCode.c, ApiCode.h,
// CAMCode.c and CAMCode.h.
// Este software usa um código de http://www.teuniz.net/RS-232/ desenvolvido
// por Teunis van Beelenwas. A versão usada é de 31 de Janeiro de 2014 e os
// arquivos utilizados são rs232.c e rs232.h
//
// Notes: This software should be used in conjunction with the software
// files generated by AnadigmDesigner2.
// These files have been given the default names ApiCode.c, ApiCode.h,
// CAMCode.c and CAMCode.h.
// This software uses a code from http://www.teuniz.net/RS-232/ by Teunis
// van Beelenwas. The version used is from January 31, 2014 and the files
// used are rs232.c and rs232.h
////////////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "rs232.h"
#include "ApiCode.h"
#include "CAMCode.h"
#include <Windows.h>

/*************************************************************************
* Declaração de variáveis                                                *
*************************************************************************/
const an_Byte* PriConfigData;   //ponteiro para a configuração primária
const an_Byte* ReconfigData;    //ponteiro para a configuração de atualização
float gain,tensão,kp,ki,g,g1,g2,g3,k;        //ganho do CAM e Tensão desejada
int i, n,n1,StreamSize,tamanho[4], //variavéis usadas nas funções
cport_nr=5,                     //variável que define a porta COM utilizada. COM6
bdrate=57600;                   //ajuste do BaudRate utilizado
fc=100;                         //clock dos CAMs
unsigned char buf[4096];        //define tamanho do buffer serial

/*************************************************************************
* Função: PrimaryConfig                                                  *
*                                                                        *
* Propósito: Esta função chama a rotina ApiCode.C para receber, preparar e     *
* enviar os dados para o FPAA. Faz uso da rotina RS232.c para envio dos dados. *
*                                                                        *
```

```
* Entrada: unsigned char chip_id - FPAA a ser configurado.                    *
*                                                                             *
* Saída: Nenhuma, mas o FPAA será configurado de acordo com a configuração    *
* primária.                                                                   *
*****************************************************************************/
void PrimaryConfig(an_Byte chip_id)
{
      PriConfigData = an_GetPrimaryConfigData(chip_id, &StreamSize);
//função da rotina ApiCode.c que gera os dados da configuração primária

      //converte o tamanho do bloco de dados de número inteiro para 4 caracteres
      //ASCII (word)
n=(StreamSize+7)*2;
      if(n>255){
            n1=n/256;
            tamanho[0]=n1/16;
            tamanho[1]=n1-(tamanho[0]*16);
            n=n-(n1*256);
            tamanho[2]=n/16;
            tamanho[3]=n-(tamanho[2]*16);
            }
      else{
            tamanho[0]=0;
            tamanho[1]=0;
            tamanho[2]=n/16;
            tamanho[3]=n-(tamanho[2]*16);
            }
      for(i=0;i<4;i++){
      tamanho[i]=tamanho[i]+48;
      if(tamanho[i]>57){
            tamanho[i]=tamanho[i]+7;
            }
      }

      //converte as informações do bloco de dados de unsigned char para caracteres
      //ASCII
      n=StreamSize;
      int temp[n],dados[2*n],j=0;
       for(i=0;i<n;i++){
      temp[i]=PriConfigData[i];
        }
      for(i=0;i<n;i++){
      dados[j]=temp[i]/16;
      dados[j+1]=temp[i]-(dados[j]*16);
      dados[j]=dados[j]+48;
      dados[j+1]=dados[j+1]+48;
      if(dados[j]>57){
            dados[j]=dados[j]+7;
      }
      if(dados[j+1]>57){
            dados[j+1]=dados[j+1]+7;
      }
      j=j+2;
      }

      //envia o comando de configuração do FPAA e cinco pads (0x00) antes dos
      //dado os cinco pads são necessários para que o FPAA já esteja inicializado
      //antes dos dados de configurações começarem a ser enviados.
      //OBS: cada informação transmitida requer dois caracteres ASCII
```

```
      RS232_SendByte(cport_nr, 0x02); //STX - indica início de comunicação com o
                                      //FPAA

    RS232_SendByte(cport_nr, 0x30); //comando para configuração que realiza reset
                                    //ao final

    //envia o tamanho do bloco de dados (contando os pads)
             for(i=0;i<4;i++){
      RS232_SendByte(cport_nr, tamanho[i]);
      }

      RS232_SendByte(cport_nr, 0x30); //início dos cinco pads
      RS232_SendByte(cport_nr, 0x30);
      RS232_SendByte(cport_nr, 0x30);
      RS232_SendByte(cport_nr, 0x30);
      RS232_SendByte(cport_nr, 0x30);
      RS232_SendByte(cport_nr, 0x30);
      RS232_SendByte(cport_nr, 0x30);
      RS232_SendByte(cport_nr, 0x30);
      RS232_SendByte(cport_nr, 0x30);
      RS232_SendByte(cport_nr, 0x30);


    //envia o bloco de configuração
      for(i=0;i<2*StreamSize;i++){
             RS232_SendByte(cport_nr, dados[i]);
      }

       //envia dois pads (0x00) antes do término dos dados de configuração e na
       //sequencia envia o comando de fim de comunicação
       //os dois pads são necessários para garantir que a configuração esteja
salva
       //antes do dispositivo reiniciar.
    RS232_SendByte(cport_nr, 0x30); //início dos dois pads
    RS232_SendByte(cport_nr, 0x30);
    RS232_SendByte(cport_nr, 0x30);
    RS232_SendByte(cport_nr, 0x30);
      RS232_SendByte(cport_nr, 0x03);  //ETX - indica o término da comunicação
com
                                       //o FPAA
}

/*************************************************************************
* Função: Configure                                                      *
*                                                                        *
* Propósito: Esta função chama a rotina ApiCode.C e CAM.c para configurar os  *
* parâmetros nas CAMs e receber, preparar e enviar os dados para o FPAA.      *
* Faz uso da rotina RS232.c para envio dos dados.                        *
*                                                                        *
* Entrada: unsigned char chip_id - FPAA a ser reconfigurado.             *
*                                                                        *
* Saída: Nenhuma, mas o FPAA será configurado de acordo com a configuração    *
* primária.                                                              *
*************************************************************************/
void Configure(an_Byte chip_id)
{     //inicializa o buffer de dados
       an_InitializeVortexReconfigData(chip_id);
       //configura os parâmetros das CAMs
       an_setGainLimiter(an_FPAA1_GainLimiter1, gain, 3);
```

```
  an_setGainSumDiff_3in(an_FPAA1_PID_Component_1, g1,g2,g3);
  an_setIntegrator(an_FPAA1_PID_Component_3, k);
  an_setGainHalf(an_FPAA1_PID_Component_4, g);
  //gera os arquivos de reconfig
  ReconfigData = an_GetVortexReconfigData(chip_id,&StreamSize);

  //converte o tamanho do bloco de dados de número inteiro para 4 caracteres
  //ASCII (word)
  n=(StreamSize+2)*2;
  if(n>255){
        n1=n/256;
        tamanho[0]=n1/16;
        tamanho[1]=n1-(tamanho[0]*16);
        n=n-(n1*256);
        tamanho[2]=n/16;
        tamanho[3]=n-(tamanho[2]*16);
        }
  else{
        tamanho[0]=0;
        tamanho[1]=0;
        tamanho[2]=n/16;
        tamanho[3]=n-(tamanho[2]*16);
        }
  for(i=0;i<4;i++){
  tamanho[i]=tamanho[i]+48;
  if(tamanho[i]>57){
        tamanho[i]=tamanho[i]+7;
        }
  }

  //converte as informações do bloco de dados de unsigned char para caracteres
  //ASCII
  n=StreamSize;
int temp[n],dados[2*n],j=0;
for(i=0;i<n;i++){
  temp[i]=ReconfigData[i];
    }
  for(i=0;i<n;i++){
  dados[j]=temp[i]/16;
  dados[j+1]=temp[i]-(dados[j]*16);
  dados[j]=dados[j]+48;
  dados[j+1]=dados[j+1]+48;
  if(dados[j]>57){
        dados[j]=dados[j]+7;
  }
  if(dados[j+1]>57){
        dados[j+1]=dados[j+1]+7;
  }
  j=j+2;
  }

  //envia o comando de configuração do FPAA e cinco pads (0x00) antes dos
  //dados os cinco pads são necessários para que o FPAA já esteja inicializado
  //antes dos dados de configurações começarem a ser enviados.
  //OBS: cada informação transmitida requer dois caracteres ASCII
  RS232_SendByte(cport_nr, 0x02); //STX - indica início de comunicação com o
                                  //FPAA
RS232_SendByte(cport_nr, 0x31); //comando para reconfiguração sem reset ao
                                  //final
```

```
      //envia o tamanho do bloco de reconfiguração
            for(i=0;i<4;i++){
      RS232_SendByte(cport_nr, tamanho[i]);
      }
    //envia o bloco de reconfiguração
      for(i=0;i<2*StreamSize;i++){
            RS232_SendByte(cport_nr, dados[i]);
      }

        //envia dois pads (0x00) antes do término dos dados de configuração e na
       //sequência envia o comando de fim de comunicação
      RS232_SendByte(cport_nr, 0x30); //início dos dois pads
      RS232_SendByte(cport_nr, 0x30);
      RS232_SendByte(cport_nr, 0x30);
      RS232_SendByte(cport_nr, 0x30);
      RS232_SendByte(cport_nr, 0x03);  //ETX - indica o término da comunicação
                                       //com o FPAA

      an_ShutdownVortexReconfigData(chip_id); // encerra o buffer de dados
}
/*****************************************************************************
* Função: main                                                               *
*                                                                            *
* Propósito : Função principal do programa. Fica em loop eterno lendo o valor *
* de tensão de referência desejado e o ganho dos controladores informados    *
* pelo usuário, calculando o ganho necessário em cada CAM e reconfigurando   *
* o FPAA.                                                                    *
*                                                                            *
* Entrada: -                                                                 *
*                                                                            *
* Saída: -                                                                   *
*****************************************************************************/

       void main()
       {

       //inicializa a comunicação serial
          if(RS232_OpenComport(cport_nr, bdrate))
         {
           printf("Can not open comport\n");
         }

         PrimaryConfig(an_FPAA1); //chama a rotina para enviar configuração
                                  //primária
         tensao=1; //para que apareça o valor de tensão da configuração primária


         kp=1,ki=1,kd=0.25; //para que apareça o valor dos ganhos da config.

       while(1){ //loop infinito
                  system("cls");  //limpar a tela

      printf("************************************************************\n");
      printf("*                                                          *\n");
      printf("*                    Controlador PID                       *\n");
      printf("*                                                          *\n");
      printf("************************************************************\n\n");
      printf("         Digite o novo valor de tensao desejado [0.03V ~ 3V]   \n");
```

```
printf("         Tensao atual = %.2f",tensao);
printf("         Tensao desejada = ");
      scanf("%f",&tensao);
      //verifica se a tensão deseja está dentro dos limites permitidos
if(tensao<0.03){
            tensao=0.03;
      }else if(tensao>3){
            tensao=3;
      }
      gain=tensao/3;
      printf("\n       Digite o novo valor de Kp [0 ~ 80]\n");
            printf("         Kp atual = %.2f",kp);
            printf("         Kp desejado = ");
      scanf("%f",&kp);
      //verifica se o ganho Kp está dentro dos limites permitidos
      if(kp==0){
            kp=0;
      }
            else if(kp<0.102){
            kp=0.102;
      }else if(kp>80){
            kp=80;
      }
      printf("\n       Digite o novo valor de Ki [0.000102 ~ 656]\n");
            printf("         Ki atual = %.2f",ki);
            printf("         Ki desejado = ");
      scanf("%f",&ki);
      //verifica se o ganho Ki está dentro dos limites permitidos
      if(kp==0){

      if(ki==0){
            ki=0;
      }else if(ki>656){
            ki=656;
      }else if(ki<0.000102){
            ki=0.000102;
      }
      printf("\n       Digite o novo valor de Kd [0.0000098 ~ 0,81]\n");
            printf("         Kd atual = %.2f",kd);
            printf("         Kd desejado = ");
      scanf("%f",&kd);
      //verifica se o ganho Kd está dentro dos limites permitidos
      if(kp==0){

      if(kd==0){
            kd=0;
      }else if(kd>0.81){
            kd=0.81;
      }else if(kd<0.0000098){
            kd=0.0000098;
      }


      //calculo dos parâmetros de cada CAM para obter os ganhos desejados
      if(kd==0){
            g=1,g3=0;
      }
      else if(kd>=0.00098){
            g=1;
```

```
                g3=(kd*fc)/g;
            }
            else{
                g=0.01;
                g3=(kd*fc)/g;
            }

            if(ki==0){
                k=1,g2=0;
            }
            else if(ki>2){ //0,8262
                k=8.1;
                g2=ki/k;
            }
            else if(ki<0.01){ //0,081
                k=0.001;
                g2=ki/k;
            }
            else{
                k=1;
                g2=ki/k;
            }

            if(kp==0){
                g1=0;
            }
            else {
            g1=(kd*fc)+kp;
            }

            if(g1>81){
                g1=81;
                kd=(g1-kp)/fc;
                if(kd>=0.00098){
                g=1;
                g3=(kd*fc)/g;
                }
                else{
                g=0.01;
                g3=(kd*fc)/g;
                }
                //exibe ganhos ajustados na tela
                printf("\n       Valor  Kd  reajustado  para  atender  Kp.  Rever
valores desejados\n");
                printf("\n       Kp=%f\n",kp);
                printf("         Ki=%f\n",ki);
                printf("         Kd=%f\n",kd);
            }
            else if(g1<0.102&&g1!=0){
                g1=0.102;
                kd=(g1-kp)/fc;
                if(kd>=0.00098){
                g=1;
                g3=(kd*fc)/g;
                }
                else{
                g=0.01;
                g3=(kd*fc)/g;
                }
```

```
                  //exibe ganhos ajustados na tela caso Kd seja recalculado
                  printf("\n      Valor Kd reajustado para atender Kp. Rever
            valores desejados\n");
                  printf("\n      Kp=%f\n",kp);
                  printf("        Ki=%f\n",ki);
                  printf("        Kd=%f\n",kd);
            }

            //exibe os parâmetros de cada CAM.
            printf("\n      G=%f\n",g);
            printf("        G1=%f\n",g1);
            printf("        G2=%f\n",g2);
            printf("        G3=%f\n",g3);
            printf("        K=%f\n",k);
            Configure(an_FPAA1); //chama rotina para reconfigurar o FPAA
            getch();
      }
}
```