RAFAEL MARQUES CHAVES

IMPLEMENTAÇÃO EM HARDWARE DA FUNÇÃO DE ATIVAÇÃO DO NEURÔNIO ARTIFICIAL UTILIZANDO INSTRUÇÃO CUSTOMIZADA PARA O PROCESSADOR NIOS II

LAVRAS-MG

2010

RAFAEL MARQUES CHAVES

IMPLEMENTAÇÃO EM HARDWARE DA FUNÇÃO DE ATIVAÇÃO DO NEURÔNIO ARTIFICIAL UTILIZANDO INSTRUÇÃO CUSTOMIZADA PARA O PROCESSADOR NIOS II

Monografia
apresentada ao Colegiado do curso de Ciência da Computação para obtenção do título de bacharel

Orientador:
Prof. Wilian Soares Lacerda

LAVRAS-MG
2010

RAFAEL MARQUES CHAVES

IMPLEMENTAÇÃO EM HARDWARE DA FUNÇÃO DE ATIVAÇÃO DO NEURÔNIO ARTIFICIAL UTILIZANDO INSTRUÇÃO CUSTOMIZADA PARA O PROCESSADOR NIOS II

Monografia
apresentada ao Colegiado do curso de Ciência da Computação para obtenção do título de bacharel

Aprovada em: ___/___/___

Cristiano de Leite Castro UFLA

Danton Diego Ferreira UFLA

Orientador:

Prof. Wilian Soares Lacerda

LAVRAS-MG

2010

AGRADECIMENTOS

À Deus, por tudo.

A minha mãe, Sandra, importantíssima na minha vida, vai estar para sempre no meu coração. Ao meu pai, Marco, aos meus irmãos e grandes amigos, Marquinho, Kaká e Cíntia por me ajudar a concluir os meus estudos.

Aos meu amigos do brejão e agregados, Danilo, Júlio, Joede, Cesão, Deivisson, Assis, Tiaguinho, Shrek, Fernanda, Jefferson, Régis, Jaques, Miryan, Ciça, Marianne, Felipe, Susto, Mairo, João Lucas, Elaine, Luiz, Mestre, Denise, Teresa, Lili, Fabiana, Cidinha Chocolate, Chernobil pela amizade e por estarem presentes na minha vida nos momentos de descontração.

Aos meus amigos da computação, Gleisson, Ricardo, Mário Élson, Luciano, Gustavo, Foguinho, Jônatas, Ferreira, Lucas, Rodrigo, Ugliara, Ânderson, Carol, Lespa, Veio, Zezé, Trombadinha, Ariana, Jojó, Hudson, Vinicius, Dawison, Alex, Hebert, Gabriel.

Aos meus amigos de Sete Lagoas, Ivam, Beto, Victor, Maurício, Alexandre, Samuel.

À Luiz André, Elza, Bruno, Andréia, Vó e seus familiares por me receberem em Lavras tão bem.

Ao meu orientador por ter tido paciência e me ajudado na elaboração do projeto.

A todas as outras pessoas que de alguma forma contribuíram na minha formação.

LISTA DE TABELAS

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Nesse capítulo será apresentado a justificativa e objetivos do trabalho, descrevendo a importância do mesmo. Também contem a organização do texto e o que será mostrado nos capítulos posteriores.

1.1 Justificativa e motivação

A Inteligência Computacional procura reproduzir por meios computacionais, características normalmente atribuídas à inteligência humana, como: compreensão de linguagem, capacidade de aprendizagem, raciocínio, visão, soluções de problemas, indução e dedução lógica, etc. As Redes Neurais Artificiais (RNA's) compreendem uma sub-área da Inteligência Computacional.

As Redes Neurais Artificiais se caracterizam por uma forma de computação não-algorítmica que, em algum nível, relembram a estrutura do cérebro humano. A área de RNAs se caracteriza por um forte componente interdisciplinar, envolvendo cientistas nas áreas de Engenharia, Informática, Física, Biologia, Neuropsicologia, Matemática e Estatística. Devido ao fato de não ser baseada em regras, a computação por RNA's se caracteriza como uma alternativa à computação algorítmica convencional. Uma RNA é formada por um conjunto de elementos processadores simples, uma rede de interconexão e uma regra de aprendizado, sendo o processamento feito de forma paralela por cada um dos nodos da rede. Esses elementos são capazes de interagir com o meio exterior adaptando-se a novas situações, o que os tornam particularmente úteis em aplicações que necessitem tomadas de decisão em tempo real. Estas características, aliadas às novidades teóricas na área, abriram novas possibilidades para aplicações industriais e um alto fluxo de recursos nos últimos anos para financiamento de pesquisas na área. A gama de aplicações industriais das RNA's é vasta, já que estas se

caracterizam como uma ferramenta alternativa e poderosa para se fazer computação. Um exemplo da importância da utilização desta tecnologia na indústria está no grande número de artigos de aplicação apresentados nas grandes conferências internacionais da área.

Geralmente as RNA's, assim como outros métodos computacionais, são implementados em *software*, se apresentando na forma de algoritmos, operando em várias plataformas diferentes. Desta forma, o funcionamento destas estruturas fica agregado ao funcionamento de um computador. Isto significa que o desempenho das mesmas será limitado, entre outros fatores, pela capacidade da plataforma de processamento, e principalmente pelo fato dos computadores serem máquinas seriais. Estas estruturas operam com um *overhead* muito elevado, uma vez que existem sinais de controle que são importantes para o funcionamento do computador, mas não para o funcionamento direto das mesmas. A implementação de redes neurais artificiais em *hardware* tem como principal motivação a eliminação deste *overhead*, uma vez que estaríamos operando em um nível físico, bem mais robusto. A implementação em *hardware* também possibilita o funcionamento paralelo das redes neurais. Em termos práticos, isto significa desempenho e velocidade.

Neste trabalho foi feita uma implementação de uma RNA em *software*. A função sigmóide implementada no programa da RNA possui um alto custo computacional, gastando muito tempo de execução e, por isso, é necessário otimizá-la. Sendo assim, tem-se uma redução de tempo significativa em relação ao programa feito somente em *software*. Essas implementações resolveram o problema do XOR e foram testadas em um kit de desenvolvimento produzido pela ALTERA Corporations que contem uma FPGA (Field Programmable Gateway Array) e vários outros recursos, por exemplo, memória RAM, botões e LEDs. Foram feitas, antes de tudo, simulações no computador, para mostrar que a implementação em *hardware*

reduz o custo computacional da função sigmoidal, podendo, assim continuar com o projeto.

1.2 Objetivo do trabalho

O objetivo do trabalho é, com a implementação da função sigmóide em *hardware*, obter uma melhora significativa de tempo de execução da RNA implementada na FPGA.

Apesar da otimização, essa implementação em *hardware* utilizará dispositivos lógicos a mais dentro da FPGA. Outra desvantagem da programação em *hardware* é que é demanda um grande tempo de projeto. Portanto, será discutido após a apresentação dos resultados se o custo-benefício viabiliza o seu uso.

1.3 Organização do texto

No capítulo 2 é apresentada a revisão bibliográfica, em que explora o histórico, aplicações e funcionamento das RNA's.

No capítulo seguinte são mostrados conceitos e importância de RNA's em *Hardware* e FPGA. No final do capítulo são apresentadas algumas aplicações desses ramos.

No capítulo 4, será mostrada a metodologia usada, ou seja, os *softwares* utilizados, mostrará o kit de desenvolvimento e como foi projetado a função sigmóide em *hardware*.

No capítulo 5 serão mostrados resultados e discussão. Será comparada a eficiência da função sigmóide construída em *hardware* em relação a função em *software* já existente. Serão mostrados também quantos dispositivos lógicos foram gastos a mais e se o custo-benefício foi bom.

No capítulo 6 está a conclusão.

No capítulo 7 contêm as referências bibliográficas.

2 REDES NEURAIS ARTIFICIAIS

Nesse capítulo será apresentado um referencial sobre Redes Neurais Artificiais mostrando a sua definição, o histórico, *perceptron* simples, redes multicamadas, a aprendizagem de uma RNA, como também o algoritmo de treinamento Backpropagation.

Também serão abordadas nesse capítulo as mais recentes pesquisas e trabalhos divulgados nessa área.

2.1 Definição

Segundo Haykin (2001), Uma rede neural é um processador maciçamente paralelamente distribuído constituído de unidades de processamento simples, que têm a propensão natural para armazenar conhecimento experimental e torná-lo disponível para o uso. Ela se assemelha ao cérebro em dois aspectos:

1. O conhecimento é adquirido pela rede a partir de seu ambiente através de um processo de aprendizagem.
2. Forças de conexão entre neurônios, conhecidas como pesos sinápticos, são utilizadas para armazenar o conhecimento adquirido.

2.2 Histórico e Arquitetura de Redes Neurais Artificiais

O neurônio biológico, mostrado na Figura 1, tem sido alvo de pesquisas, tanto por parte de biólogos quanto de matemáticos, físicos, químicos e engenheiros. Neurônios biológicos são modelados matematicamente em um tipo de neurônio artificial. Redes neurais artificiais são compostas por vários neurônios artificiais interligados.

McCulloch e Pitts (1943) descreveram um simplificado, mas poderoso modelo artificial de um neurônio biológico (MCP), o qual ainda está em uso hoje em modelos de redes neurais artificiais. A Figura 2 mostra o modelo proposto.

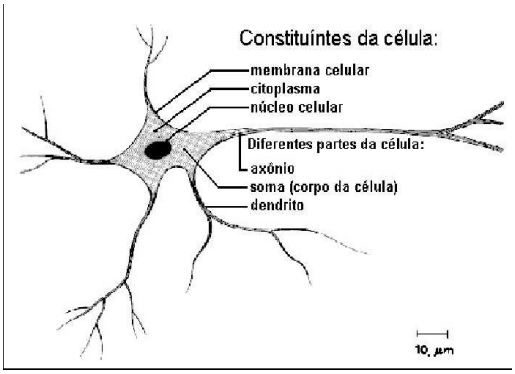


Figura 1 Neurônio biológico

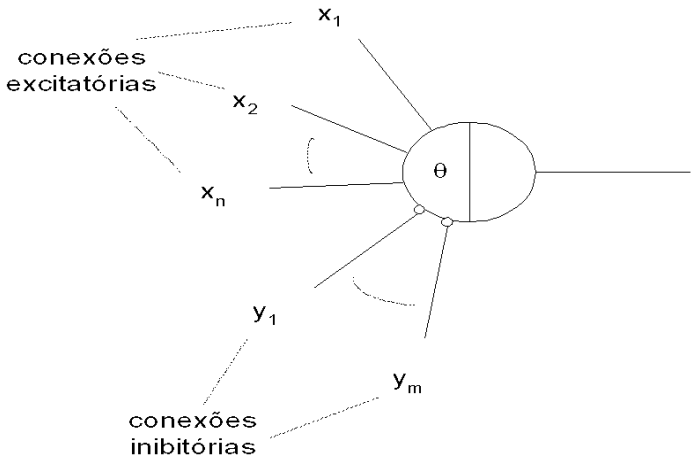


Figura 2 Neurônio artificial proposto por MacCulloch e Pitts (1943) baseado no neurônio biológico

Somente alguns anos depois que foram criados estudos a partir do trabalho de McCulloch e Pitts. O primeiro que se tem notícia, fazendo ligação direta com o aprendizado de redes artificiais, foi feito por Donald Hebb, em 1949. Ele propôs uma teoria para explicar o aprendizado em neurônios biológicos baseada no reforço das ligações sinápticas entre neurônios excitados (BRAGA, 2007, p.3).

Widrow e Hoff (1960) criaram um modelo chamado regra delta que é baseada no método do gradiente descendente – que mede a distância entre

a resposta desejada e a atual para fazer os ajustes desejados nos pesos das conexões - na tentativa de minimizar o erro na saída.

Em 1958, Frank Rosenblatt criou um modelo de neurônio chamado de *perceptron* que contem pesos alteráveis em suas sinapses. Assim, pode-se treinar a rede, alterando os parâmetros dos pesos.

Em 1969, Minsky e Paperty (Minsky, 1969) mostrou algumas situações que o perceptron de uma única camada descrito por Rosenblatt não era capaz de funcionar corretamente. Esse perceptron, que ficou conhecido como *perceptron simples*, está limitado a resolução de problemas linearmente separáveis.

Nos anos 1970, a abordagem conexionista ficou adormecida (em grande parte devido à repercussão do trabalho de Minsky e Papert), apesar de alguns poucos pesquisadores continuarem trabalhando na área. Entre eles podem ser citados Igor Aleksander (redes sem pesos), na Inglaterra, Kunihiko Fukushima (cognitron e neoxcognitron), no Japão, Steven Grossberg (sistemas auto-adptativos), nos EUA e Teuvo Kohonen (memórias associativas e modelos auto-organizáveis), na Finlândia (BRAGA, 2007, p.4).

Em 1982, John Hopfield (1982) retomou as pesquisas na área mostrando propriedades associativas da RNA e utilizando teorias recorrentes da física para estudar modelos de treinamento.

Em 1986, Rumelhart, Hinton e Williams (1986) criaram o algoritmo de treinamento Backpropagation. Esse trabalho mostrou que Minsky e Papert estavam totalmente corretos quanto à dificuldade das RNAs de aprender em problemas não-linearmente separáveis. O Backpropagation é um modelo que resolve grande parte desses problemas.

Os artigos de Hopfield e de Rumelhart foram os que mais influenciaram o ressurgimento das redes neurais (HAYKIN, 2001). De fato, após a descrição do algoritmo Backpropagation, grande parte das pesquisas em RNA foi na tentativa de propor variações desse modelo que possa ter uma maior velocidade de convergência (BRAGA, 2007).

2.3 Perceptron

Nessa seção será apresentado o modelo do *perceptron* e como ele funciona. Também será mostrado sobre alguns tipos de função de ativação muito usados.

2.3.1 Modelo do perceptron

A figura 3 mostra o perceptron criado por Rosenblatt. O valor de y, como mostra a figura, é o resultado da saída do neurônio que é calculado por uma função de ativação *f(v)*. v é obtido pela soma dos produtos do peso w pelos valores de entrada x(1 ou 0) de cada ligação. O somatório termina quando atinge o valor n que é o números de conexões que o perceptron tem. A fórmula que rege o neurônio artificial está na equação 2.1.

$$y = \sum_{j=1}^{n} Wj \times Xj \tag{2.1}$$

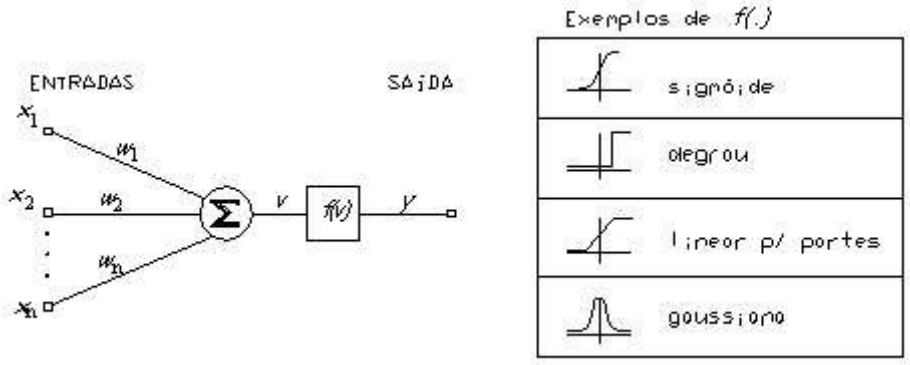


Figura 3 Modelo de perceptron criado por Rosenblatt

2.3.2 Função de ativação

Portas de limiar executam a tarefa de comparar o valor da soma v com um valor de limiar (*threshold*). A função de ativação de um neurônio MCP, é do tipo degrau deslocada do limiar de ativação em relação a origem (BRAGA, 2007).

2.3.2.1 Função de ativação limiar

Para alguns problemas, usa-se função limiar, que é da forma como mostra a equação 2.2. E a representação gráfica dessa função está na figura 4.

$$f(v) = \begin{cases} 1, \text{ se } v \geq 0; \\ 0, \text{ se } v \leq 0; \end{cases} \tag{2.2}$$

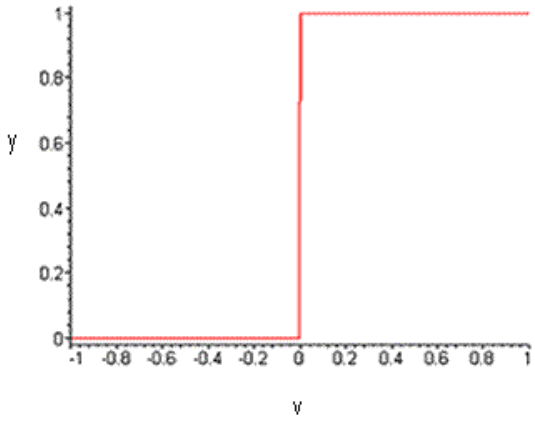


Figura 4 Gráfico da função limiar

2.3.2.2 Função de ativação sigmóide

A função sigmóide é a função geralmente usada no algoritmo Backpropagation. A sua fórmula é a equação 2.3. O gráfico que representa a fórmula está na figura 5.

$$f(v) = \frac{1}{1+e^{-v}} \tag{2.3}$$

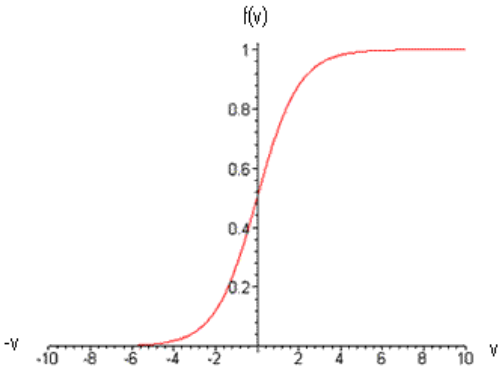


Figura 5 Gráfico da função sigmóide

2.4 Redes MLP(Multi Layer Perceptron)

Vários nós conectados formam uma rede conforme é mostrado na Figura 6. Nós MLP, que em português significa Perceptron Multi-Camadas, são hábeis para atuar em ambientes adaptativos pelo ajuste dos pesos de cada conexão. Os pesos devem ser ajustados para minimizar o erro entre a resposta do sistema e a saída desejada.

O modelo é dividido em camadas, são elas: camada de entrada, camada intermediária e camada de saída. (OMONDI, 2006).

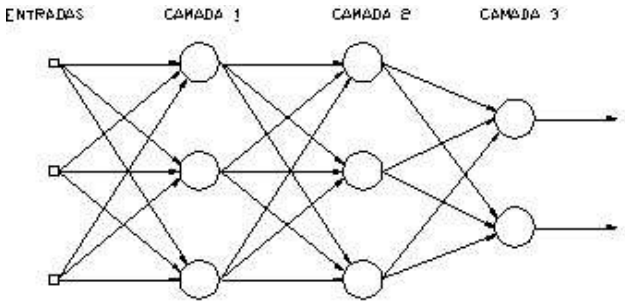


Figura 6 Rede de neurônios

– Camada de entrada

A primeira camada recebe somente os valores de entrada, não há nenhum tipo de processamento. Na figura 6, a camada de entrada é a primeira da figura da esquerda para a direita.

– Camada intermediária

É a camada que faz praticamente todo o processamento. Pode-ser ter várias camadas intermediárias, porém duas camadas são suficientes. Foi provado que mais que essa quantidade não haverá nenhuma melhora na convergência da rede. Na figura 6, as camadas intermediárias são representadas pelas camadas 1 e 2.

– Camada de saída

A camada de saída é a resposta da rede. Ela recebe os resultados da camada intermediária. Também faz o processamento, no algoritmo Backpropagation usa-se essa camada para fazer a estimativa do vetor gradiente que é necessária para retropropagação (HAYKIN, 2001). A camada de saída é a última camada representa na figura 6.

2.4.1 Número de neurônios

A definição do número de neurônios em cada uma das camadas de redes é de extrema importância para o seu desempenho, principalmente no que se refere à capacidade da rede em resolver problemas de determinada complexidade. Quanto maior o número de neurônios, maior a complexidade da rede e maior a sua abrangência em termos de soluções possíveis. A determinação do número de neurônios é, na verdade, o problema mais fundamental em aprendizado de redes neurais, e motiva boa parte das pesquisas na área. Não existe uma regra geral ou fórmula para determinar o número de neurônios que a rede neural necessita para resolver certo problema, mas há na literatura algumas propostas para estimar o tamanho da rede (BRAGA, 2007).

2.4.2 Efeitos do superdimensionamento e do subdimencionamento

Na seção anterior foi visto que a determinação do número de neurônios é muito importante para o desempenho da rede e maior é a sua abrangência em termos de soluções possíveis. Entretanto, há um limite para isso, sendo que em determinados problemas um número muito alto de neurônios pode causar mais erros, como também o contrário, menos neurônios na rede dificulta a convergência para a resposta certa. Segundo Braga (2007), quanto maior o número de parâmetros, mas difícil é, na realidade, a busca pelas soluções que se aproximam da função geradora dos dados.

Portanto, aumentando-se o número de neurônios na camada escondida aumenta-se a capacidade de mapeamento não-linear da rede. No entanto, quando esse número for muito grande, o modelo pode se sobre-ajustar aos dados, na presença de ruído nas amostras de treinamento. Diz-se que a rede está sujeito ao sobre-treinamento (*overfitting*).

Por outro lado, uma rede com poucos neurônios na camada escondida pode não ser capaz de realizar o mapeamento desejado, o que é denominado de *underfitting*. O *underfitting* também pode ser causado quando o treinamento é interrompido de forma prematura.

Nas figuras 7, 8, 9 são mostrados o ajuste ideal e o erro causado por underfitting e overfitting, respectivamente. Nas figuras, a linha tracejada mostra o gráfico desejado e a outra linha mostra o obtido. Como se vê na figura 7, o ajuste ideal tem alguns erros, mas isso é totalmente normal, nem sempre a RNA acertará 100% das respostas. O que não pode ocorrer é o que acontece nas figuras 8 e 9. Devido aos erros muito grandes, erram por muito o valor de y para uma dada entrada x.

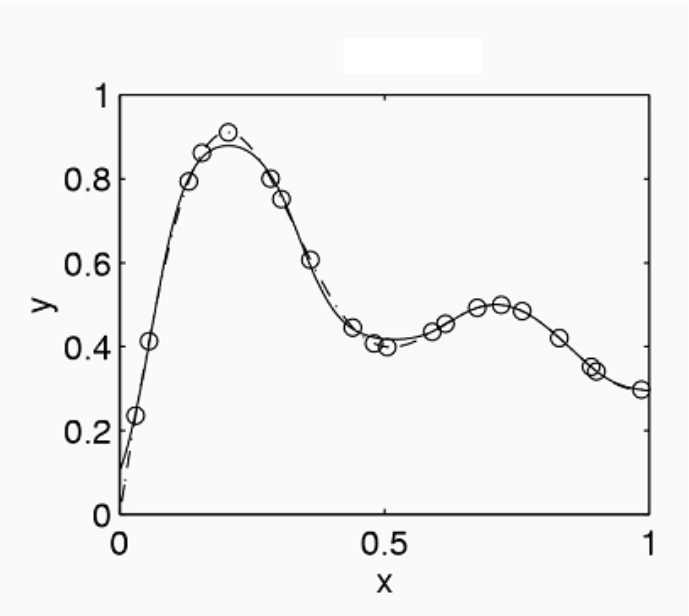


Figura 7 Ajuste ideal da função

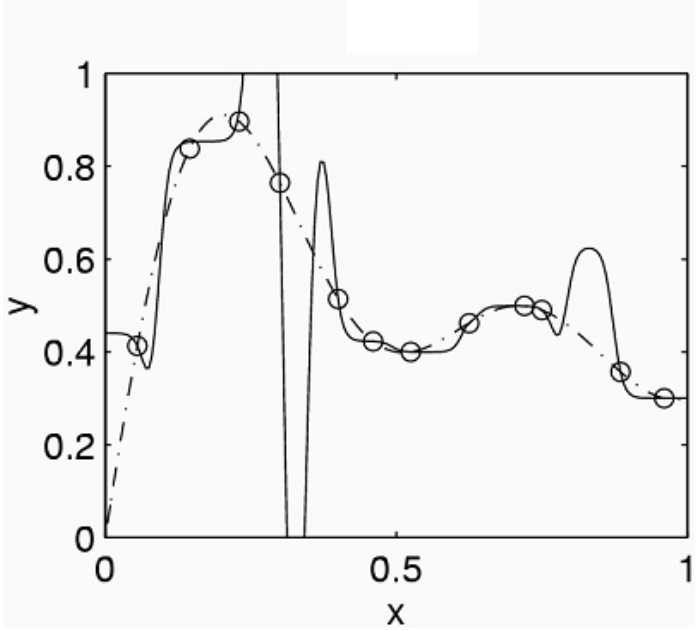


Figura 8 Overfitting

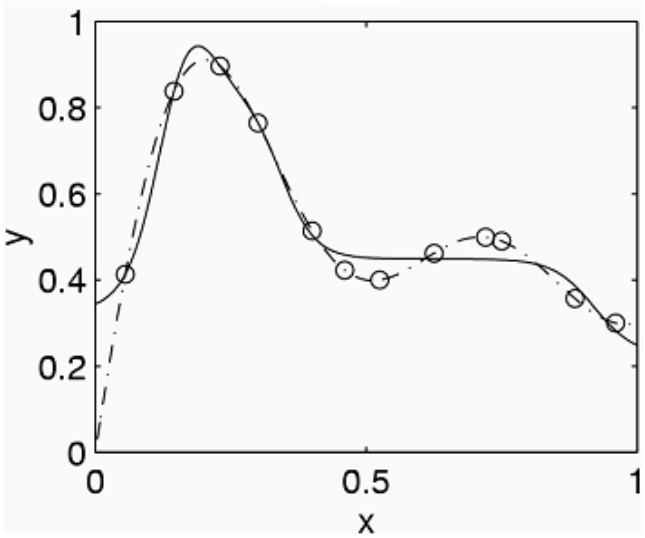


Figura 9 underfitting

2.5 Aprendizagem de uma RNA

"Para um funcionamento correto, RNA's devem passar por um processo de aprendizagem antes deles serem capazes de fazer corretamente suas tarefas" (Granado et al¸ 2006).

Segundo Haykin (2001), "Aprendizagem é um processo no qual os parâmetros livres de uma rede neural são adaptados através de um processo de estimulação do meio-ambiente no qual a rede está inserida."

Outro conceito é de Hassoun (1995), que disse que a "aprendizagem é vista como processo de otimização. Isto é, aprendizagem consiste em um processo de busca de solução em um espaço multidimensional de parâmetros, em geral os pesos das conexões, que otimize uma dada função objetivo".

2.5.1 Tipos de aprendizado

Segundo Haykin(2001), "O tipo de aprendizagem é determinado pela maneira que ocorrem as mudanças nos parâmetros".

Podem ser divididos em supervisionado ou não supervisionado (GRANADO et al, 2006):

– Aprendizado supervisionado

O usuário dispõe de um comportamento de referência preciso que ele deseja ensinar a rede. Sendo assim, a rede deve ser capaz de medir a diferença entre seu comportamento atual e o comportamento de referência, e então corrigir os pesos de maneira a reduzir este erro (desvio de comportamento em relação aos exemplos de referência) (OSÓRIO, 1995).

– Aprendizado não supervisionado

As redes com aprendizado não supervisionado utilizam um algoritmo auto-organizável com o objetivo de descobrir características ou padrões significativos em um conjunto de dados de entrada. Este tipo de aprendizado consiste na repetida modificação dos pesos sinápticos da rede em resposta a padrões de ativação.

2.5.2 Treinamento Backpropagation

A arquitetura de RNA mais comum é o Backpropagation - um algoritmo de treinamento de RNA para redes multi-camadas (RUMELHART, 1986). Ele tem duas fases, a primeira é a propagação e depois retropropagação. Segue a explicação (MENDES E OLIVEIRA, 2009b) nas duas seções seguintes.

2.5.2.1 Propagação

É apresentado, primeiramente, um padrão de entrada e logo depois a resposta de uma unidade é propagada como entrada para as unidades na camada seguinte, até a camada de saída, onde se calcula o erro e a resposta da rede é obtida. A figura 10 ilustra a fase de propagação em uma rede de neurônios.

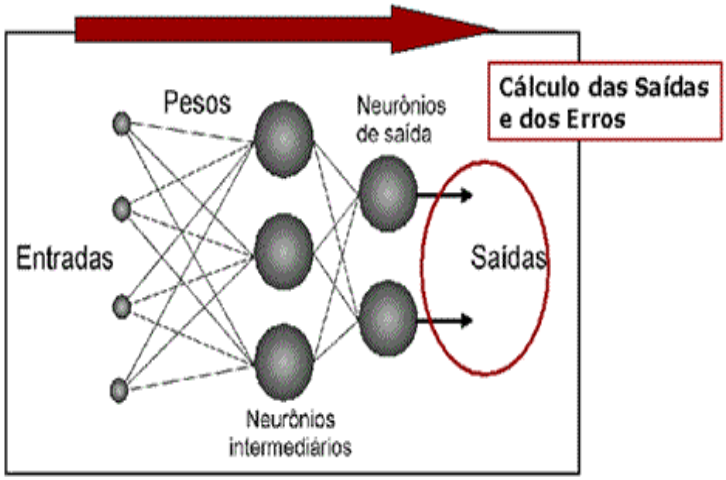


Figura 10 A fase feedfoward (Propagação)
Fonte: (MENDES E OLIVEIRA, 2009b).

A propragação é regida de acordo com a equação 2.4. Nessa figura, u_k é a saída de cada neurônio k da camada por uma soma de produtos, em que w_{kj} é a matriz de pesos em que j é uma ligação do neurônio, x_j é o valor da entrada e b_k é o bias do neurônio k. Depois de calculado u_k, é necessário o uso de uma função de ativação do neurônio, no caso uma função sigmoidal, y_k que tem como parâmetro de entrada u_k.

$$y_k = f(u_k) = f\left(\sum_{j=1}^{m} w_{kj} x_j + b_k \right) \qquad (2.4)$$

O cálculo de $f(u_k) = f(v)$ é mostrado na equação 2.5, ou seja, chamamos u_k de v.

$$f(v) = \frac{1}{1+ e^{-v}} \qquad (2.5)$$

Após o cálculo da saída y_k é feito o cálculo do erro, de acordo com a equação 2.6. $d_k(t)$ é o valor desejado e $e_k(t)$ é o valor do erro, na iteração k, com o parâmetro t.

$$e_k(t) = d_k(t) - y_k(t) \tag{2.6}$$

Este sinal de erro será utilizado para computar os valores dos erros das camadas anteriores e fazer as correções necessárias nos pesos sinápticos.

2.5.2.2 Retropropagação ("backpropagation")

Desde a camada de saída até a camada de entrada, são feitas alterações nos pesos sinápticos. Isso se faz calculando erros locais chamado de gradiente do erro(δ), para cada unidade da camada de saída até a camada de entrada. As fórmulas do gradiente são as equações 2.7 (para a camada de saída) e 2.8 (para as demais camadas).

$$\delta_k(t) = e_k(t) f(u_k)(1 - f(u_k)) \tag{2.7}$$

$$\delta_k(n) = f(u_k)(1 - f(u_k)) \sum \delta_k w_{jk} \tag{2.8}$$

Em que:

- δ_k é o erro das unidades da camada anterior conectadas a unidade j.
- w_{jk} - são os pesos das conexões com a camada anterior.

Após o cálculo dos erros de cada unidade, é calculado o ajuste dos pesos de cada conexão segundo a regra delta generalizada e são, então, atualizados os pesos segundo a equação 2.9 e a equação 2.10.

$$\Delta w_{kj}(n+1) = \alpha w_{kj}(n) + \eta \delta_j y_j \tag{2.9}$$

$$w(n+1) = w(n) + \Delta w_{kj}(n) \tag{2.10}$$

Onde:

- α - é a constante de momentun, quando $\alpha = 0$, esta função funciona como a regra delta comum;
- η- é a taxa de aprendizado;
- δ_j - é o erro da unidade;
- *y*j - é a saída produzida pela unidade j;

A figura 11 ilustra a fase de retropropagação, mostrando que primeiramente ajusta os pesos da camada de saída e logo depois das camadas intermediárias até atingir a camada de entrada.

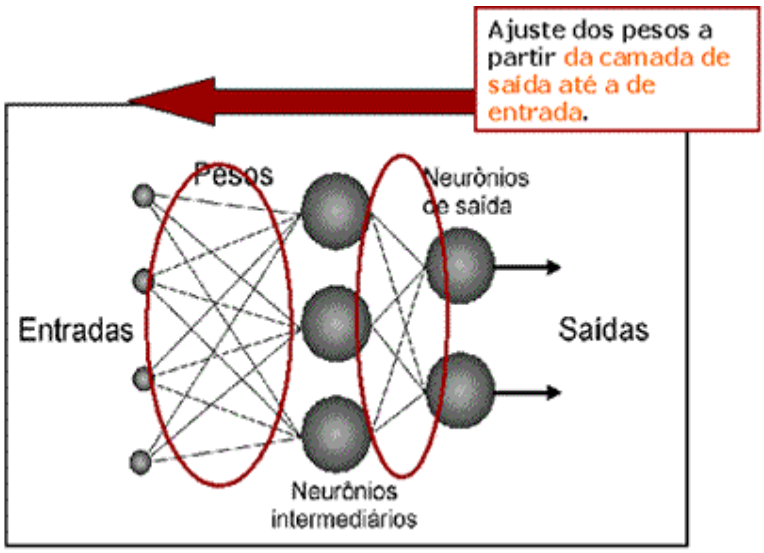


Figura 11 A fase de retropropagação
Fonte: (Mendes e Oliveira, 2009).

2.6 Trabalhos Realizados nos Últimos Anos

Em particular, a tecnologia decorrente das redes neurais artificiais tem gerado aplicações e produtos em diversas áreas. Em controle de processos industriais, área pioneira, as primeiras experiências datam de 1975. Desde então, com o desenvolvimento da eletrônica digital, as aplicações são bastante diversificadas. Exemplos típicos incluem produtos de

consumo tais como geladeiras, ar condicionado, câmeras de vídeo, máquinas de lavar roupas, fornos de microondas, aspirador de pó, e outros. Na indústria automotiva destacam-se transmissões automáticas, injeção eletrônica, suspensão ativa, freios anti-bloqueantes.

Granado (2006) acrescenta que "RNA são explicitamente configuradas para diversas aplicações como reconhecimento de padrões, classificações de dados e muitos outros".

Mendes e Oliveira (2009a) aplicaram RNA em Bioinformática com o objetivo de criar ferramentas úteis para extração de informações na área de medicina, farmacologia e outras relacionadas com a biologia molecular. No caso, a RNA é usada mais especificamente em análise de seqüências de nucleotídeos ou aminoácidos.

Nos dias atuais, o uso cada vez mais freqüente de sistemas de informação e a premente necessidade de aumento da segurança, trazem a necessidade de se identificar, autenticar e controlar os usuários de forma segura (JAIN et al, 2002). Na maioria dos sistemas computacionais, a autenticação de usuários ocorre através de senhas alfanuméricas, que representam um sério problema de segurança quando acabam parando em mãos erradas. Para evitar este problema, várias formas de autenticação de usuários baseadas em características biométricas físicas vêm sendo desenvolvidas (MILTON ROBERTO E FERNANDO SANTOS, 2005). Porém, segundo Gupta (1997), gera muito desconforto essa tecnologia, pois precisa de um alto grau de instrução e também o custo de *hardware* é muito alto.

Por causa desse problema, Milton Roberto e Fernando Santos (2005) propôs um sistema de autenticação de assinaturas manuscritas, usando RNAS e obteve 99,6% de acertos e com um custo menor.

Lopes e Wilhelm (2009) criaram um sistema de Previsão de Vendas do Sistema de Informação do Jogo de Empresas Virtual (JE) usando RNAs. Uma das principais deficiências encontradas em programas de ensino

baseados em JE tem sido a falta de mecanismos para auxiliar no processo decisório. Em suas aplicações, os JE dão muita ênfase ao jogo e pouco, ou quase nada, aos usuários (WILHELM, 1995). Assim, os usuários eram obrigados a fazer cálculos complicados e demorados ao invés de se preocupar com a abrangência do processo decisório, e suas questões táticas e estratégicas (LOPES E WILHELM, 2009). Para contornar esse tipo de problema é que foi proposto esse sistema. Apesar de resultados ruins, pois há poucos estudos nessa área já feitos, o sistema pode servir de base para várias áreas de gestão empresarial.

Santos e Souza (2008) fizeram uma classificação de distúrbios na rede elétrica. Em sua tese ela mostra que recorreu a técnicas de processamento de sinais, a fim de automatizar o diagnóstico sobre os tipos de distúrbio presentes nos sinais registrados usando RNAs e transformada wavelet.

Júnior (2008) usou RNA para classificar e detectar arritmias cardíacas. As arritmias ou ritmos anormais do coração são distúrbios cardíacos comuns e podem causar sérios riscos à vida das pessoas; sendo uma das principais causas de óbito. Muitos destes óbitos poderiam ser evitados se fosse realizado um monitoramento prévio dessas arritmias; a partir do Eletrocardiograma (ECG). O monitoramento contínuo e a detecção automática de arritmias do coração podem auxiliar o médico em um diagnóstico mais rápido e preciso. Entretanto, muitos dados podem ser gerados quando se faz a monitoração contínua de um paciente no decorrer de um dia (aproximadamente 2Mb/h com apenas 1 derivação) devendo estes ainda serem analisados por um especialista. Este trabalho levanta a hipótese de que é possível classificar as arritmias cardíacas utilizando a transformada Wavelet em conjunto com redes neurais artificiais (RNAs) auto-organizáveis e um algoritmo de pós-processamento. Esta RNA permite que a qualquer momento possam ser adicionados outros tipos de arritmias sem a necessidade de um novo treinamento da rede neural. Com este método

incremental o tempo de treinamento para novas arritmias diminui. Neste trabalho classificou-se os batimentos normais as contrações prematuras atriais (CPA) e as contrações prematuras ventriculares (CPV). Ao final adicionou-se dois novos tipos de arritmias *left bundle branch block* (LBBB) e *right bundle branch block* (RBBB) para verificar a propriedade desta RNA de não perder o conhecimento adquirido.

Muitos outros estudos também têm sido feitos em RNA, como Redes Neurais para aplicações em Lógica Fuzzy (WEBER, 2003), aplicação de redes neurais para o diagnóstico diferencial da doença meningocócica (MARTINS, s.d).

3 REDES NEURAIS ARTIFICIAIS EM HARDWARE

O desenvolvimento de uma rede neural pode ser realizado tanto em *software* quanto em *hardware*, havendo vantagens e desvantagens entre ambos. Para o desenvolvimento em *software,* as vantagens recaem sobre a facilidade e o tempo gasto para implementação da rede, já as desvantagens estão relacionadas à lentidão dos dados, por serem processados sequencialmente. Para a implementação em *hardware*, as vantagens estão relacionadas ao paralelismo intrínseco das redes neurais, enquanto que as desvantagens ficam a cargo de se alcançar um equilíbrio razoável na precisão de *bits*. Dentre as características inerentes das redes neurais, duas chamam a atenção para o desenvolvimento em *hardware*, são elas: **não-linearidade,** por permitir a resolução de problemas que não sejam linearmente separáveis; **processamento paralelo,** que é a capacidade de receber múltiplas informações e testá-las ao mesmo tempo (HASSAN et al, 2008, LUDWIG E COSTA, 2007, VALENÇA, 2005).

Nas próximas seções será discutido os tipos de implementação de RNA em *hardware* e a dificuldade de suas implementações. Também será apresentado o conceito de FPGA e suas vantagens e limitações. Além disso, na última seção será abordado trabalhos de outros autores sobre RNA em *hardware*.

3.1 Tipos de implementação de treinamento de RNA

Várias alternativas de *hardware* amigável tem sido propostas por muitas regras de aprendizado de redes neurais, especialmente com o objetivo de habilitar aprendizado on-chip. Dependendo de onde é realizado o treinamento, a rede pode ser classificada como (MOERLAND, 1997):

- Off-chip learning
- Chip-in-the-loop learning

- On-chip learnig

3.1.1 Off-chip learning

Neste caso o *hardware* não está envolvido no processo de treinamento, o qual é realizado em um computador de alta precisão. Os pesos resultantes do processo de treinamento são quantizados e então carregados no chip para fazer somente a propagação forward.

3.1.2 Chip-in-the-loop learning

Neste caso a rede neural em *hardware* é usada durante treinamento, mas apenas em propagação forward. O cálculo do novo peso é feito on-chip em um computador, o qual carrega os pesos no chip depois de cada iteração de treinamento.

3.1.3 On-chip learnig

O treinamento da rede neural é feito inteiramente on-chip o qual oferece a possibilidade de treinamento contínuo. Isto significa que os valores dos pesos são representados com apenas uma precisão limitada. Simulações têm mostrado que o popular algoritmo de retropropagação é altamente sensível ao uso de pesos de precisão limitada e que o treinamento falha quando a precisão do peso é menor que 16 *bits*. Isto ocorre principalmente por causa da atualização dos pesos serem menores que o passo de quantização, o que impede a troca do peso. Para reduzir a área do chip necessária para armazenamento do peso e superar o ruído do sistema, é desejado um favorecimento na redução do número de valores de pesos permitidos.

3.2 Dificuldades para implementações de RNA em *hardware*

Nas seções seguintes, são apresentadas algumas das influências das limitações para implementações em *hardware* de redes neurais.

3.2.1 Efeito da quantização

Para implementação digital com alta precisão numérica é necessário uma grande área de chip, enquanto em implementação analógica o ruído de sistema impede alta precisão. Então implementações em *hardware* de redes neurais tipicamente usam uma representação de parâmetros da rede com uma precisão limitada, por exemplo: 16 *bits* de peso são usados durante o processo de treinamento e apenas 4 *bits* ou 8 *bits* de peso são empregados durante a fase forward. Um exemplo de implementação em eletrônica analógica, a implementação pode ser comparada com a resolução de apenas 7 *bits*. Desde que implementações em *hardware* são caracterizadas por uma precisão baixa, é essencial estudar os seus efeitos na fase forward e no treinamento dos vários modelos de redes neurais. Projetistas de neurocomputadores digitais preferem uma precisão do peso para treinamento backpropagation em torno de 16 *bits* (MOERLAND, 1997).

3.2.2 Treinamento

Para toda iteração através do conjunto de treinamento, aleatoriamente reparticionamos o conjunto de dados em um grande número de blocos balanceados para usar para treinamento e um pequeno número para ser usado no teste (SAVRAN E UNSAL, 2003).

A realização de grandes redes backpropagation em *hardware* analógico possui sérios problemas por causa da necessidade de separar ou bidirecionar circuitos para o passo de backward do algoritmo. Outros problemas são a necessidade de precisão da derivada da função de ativação e

o cascateamento de multiplicadores no passo de backward (ALIPPI E NIGRI, 1991). Uma alternativa seria a idéia geral de algoritmos de perturbação, a qual obtém uma estimativa direta do gradiente por uma perturbação aleatória de alguns parâmetros da rede, usando o passo forward da rede para medir o erro resultante. Então, esta técnica de treinamento *on-chip* não apenas elimina o passo complexo de *backward*, mas também torna mais robusto a não idealidades ocorridas em *hardware*. As duas principais variantes desta classe de algoritmo são perturbação do nó o qual é baseado na perturbação do valor de entrada do neurônio, e perturbação do peso. A principal desvantagem destes algoritmos de perturbação é que sua natureza sequencial, como oposto ao cálculo da atualização do peso no algoritmo de retropropagação o qual pode em princípio ser feito em paralelo (MOERLAND, 1996).

A implementação de regras de aprendizado pode ser grandemente simplificada se apenas usa informação que é localmente avaliada. Esta característica minimiza o montante de ação e comunicação. Comparado ao estado da arte em implementações digitais de redes neurais, o projeto de implementação analógica com não idealidades como componentes não uniformes, respostas não ideais, e ruído de sistema, está ainda em estado experimental (MOERLAND, 1997).

3.2.3 Função de ativação

A função sigmoidal é a função de ativação não linear tradicional usada em redes neurais. A função sigmoidal não é apropriada para implementação digital direta, já que consiste de uma série exponencial infinita. Muitas implementações usam uma tabela para aproximar a função sigmoidal. Entretanto a quantidade de *hardware* necessário para esta tabela

pode ser muito grande especialmente se uma aproximação razoável é desejada. Uma simples função não linear de segunda ordem pode ser usada como uma função sigmoidal. Esta função não linear pode ser implementada diretamente usando técnicas digitais (BLAKE ET AL, 1998).

3.3 FPGA (Field Programmable Gate Array)

Antes do advento da lógica programável, circuitos lógicos eram construídos em placas utilizando componentes padrões, ou pela integração de portas lógicas (FABBRYCIO E CARDOSO, 2007). E para simulações que precisam de chips personalizados, os circuitos ficariam muitos caros, pois um circuito seria desperdiçado, toda vez que fosse feito um teste, até alcançar o desejado. Atualmente, tem-se a (WEBER, 2003) "[...] opção de usar chips programáveis, chamados de FPGAs (field-programmable gate arrays) ou, mais raramente, de LCAs (logic-cell arrays). Como o nome sugere, eles são chips compostos por um enorme número de chaves programáveis, que podem ser configurados para simular o comportamento de qualquer outro circuito. Um único FPGA pode simular não apenas um processador simples, mas também outros circuitos de apoio, como o controlador de vídeo, uma interface serial e assim por diante. Os modelos recentes incluem inclusive uma pequena quantidade de memória RAM e circuitos de apoio, de forma que você pode ter um sistema completo usando apenas um chip FPGA previamente programado, um chip de memória EPROM (ou memória flash) com o *software*, a placa de circuito com as trilhas e conectores e uma bateria ou outra fonte de energia."

Atualmente os FPGAs apresentam novas e interessantes características tais como (Nacer, s.d):

• Reprogramação dinâmica total ou parcial: que permite a implementação do conceito de *hardware* adaptativo1. Esta abordagem implementa a virtualização do *hardware* através do conceito de *cache de hardware* e que funciona *semelhante* a cachê de *software*. Apenas o *hardware* necessário está mapeado no FPGA, o restante reside em memória na forma de vetores

de configuração. Quando se fazem necessárias novas funções de *hardware* é feita a carga dos novos vetores no FPGA, modificando regiões ou mesmo reconfigurando-o totalmente. Tudo é feito de forma rápida e sem perda de informações.

• Memória SRAM distribuída: não se fazendo necessário utilizar células do FPGA para memórias. Estas memórias distribuídas podem ser utilizadas para guardar resultados intermediários e/ou informações.

• Novos arranjos de células e funções primitivas: que permitem a implementação de algoritmos de processamento de sinais que são normalmente implementados em DSPs.

• Aumento da densidade de portas: atualmente encontra-se no mercado FPGAs com densidade de 50K a 1 milhão de portas.

3.4 Trabalhos Realizados nos Últimos Anos

Na implementação por *hardware*, o tempo de desenvolvimento de um protótipo é maior comparado ao tempo utilizado em uma implementação por *software*. Contudo, devido a alta taxa de processamento paralelo que pode ser obtida, torna-se ideal para aplicações que envolvam o processamento de sinais em tempo real (MOLZ, 1999).

Assim, vários trabalhos foram feitos e estão sendo feitos em *hardware*, com o intuito de aperfeiçoar a técnica de RNA em *hardware* para ser usada na indústria e produtos específicos, como sensores, que serão usadas em atividades de campo ou não.

Molz (1999) implementou redes neurais em um ambiente de *codesign* em uma placa reconfigurável na intenção de aperfeiçoar a técnica em *hardware*. Foi notado que o cálculo da função de ativação era muito complexo para FPGA. A solução para este problema foi fazer uma tabela de valores previamente definidos contendo os valores de entrada u e suas respectivas saídas da rede, removendo, assim, a função sigmoidal. O

problema dessa solução é que trouxe muitos erros à rede, pois foi aproximado o valor de saída, demorando mais a convergência da rede neural.

Nacer (s.d) propôs uma arquitetura neural em *hardware*, usando uma FPGA, eficiente para serem usados em reconhecimento de padrões de sensores de gás. Foi usado o algoritmo de retropropagação em VHDL (Linguagem de descrição de hardware) e os resultados foram considerados adequados, pois a taxa de erros na classificação dos padrões pode ser comparável a um processador comum.

Soares (2006) fez uma implementação do controlador "sensorless" do motor de indução trifásico, utilizando como base redes neurais artificiais em FPGA. Para a tarefa de representação da função de ativação sigmoidal dos neurônios em *hardware*, fez-se uso da técnica de interpolação "spline". Como resultado final do trabalho, foi construído um protótipo do controlador do motor de indução trifásico, composto por um FPGA gerenciado por um DSP (processador digital de sinais), controlando circuitos de potência para o acionamento do motor.

Borja (2007) investiga algumas vantagens do uso dos sistemas de refrigeração com compressor de velocidade variável e válvula de expansão acionadas eletronicamente. O uso de sistemas de controle inteligente em processos industriais vem aumentado rapidamente nas ultimas décadas, principalmente em sistemas de difícil modelagem matemática. Nos ciclos de refrigeração a eficiência dos sistemas está diretamente ligada à capacidade de manter as temperaturas e pressões correspondentes às exigidas pelo processo. As temperaturas de condensação e evaporação possuem uma grande influência quanto ao consumo de energia e desempenho do sistema de refrigeração, sendo influenciado pelas perturbações externas. As influencias da mudança de velocidade e abertura/fechamento da válvula sob o ponto de vista de redução de consumo de energia são tratados através de um método de identificação e controle do sistema, utilizando Redes Neurais Artificiais como modelo Black-Box.

Dou, Xia, Jiang (2009) faz predição de estrutura secundária usando SCFG (stochastic context-free grammars, em português, gramáticas estocásticas livres de contexto) em FPGA. Nesse trabalho, os autores colocam o algoritmo CYK em chips para aumentar a eficiência do paralelismo que esse algoritmo pode trazer. O algoritmo CYK mostra as complexas dependências de dados, em que a distância da dependência é variável. Então, os autores propuseram uma solução para isso, fazendo uma estrutura de arrays sistólicas.

Widrow, Rumelhart, Lehr (1994) mostra em seu artigo, alguns trabalhos já desenvolvido nas áreas de RNA em *hardware* em aplicações na indústria, na área comercial e na ciência. Abaixo estão listados alguns deles:

- Telecomunicações;
- Controle de sons e vibrações, em sistemas automotivos e de ar-condicionado;
- Controle de feixes do acelerador de partículas;
- Aprovação de empréstimos;
- Exploração de petróleo;
- Detecção de fraudes de cartão de crédito;
- Controle de qualidade na manufatura;
- Detecção de explosivos em bagagens em aeroportos.

4 METODOLOGIA

A metodologia da implementação da função de ativação em *hardware* é aqui descrita.

Primeiramente, para provar que realmente a implementação em *hardware* atingirá o resultado esperado, foram feitas simulações em um computador. Essas simulações basearam-se em criar funções sigmóides discretizadas no lugar da função sigmóide contínua. Isso será abordado mais detalhadamente na seção 4.1.

Depois de feito as simulações, foi implementado duas funções sigmóides em *hardware* para se comparar quantas vezes cada função é mais eficiente que a função recursiva em *software*. A idéia é melhorar o desempenho da função sigmóide na execução da RNA.

4.1 Simulação da função sigmóide discretizada implementada em *software* no computador

A função sigmóide implementada no algoritmo de RNA usa a função recursiva contínua exponencial. Essa função é encontrada dentro da biblioteca das linguagens C e C++ . Para as simulações e a posterior implementação em *hardware* foram feitas algumas discretizações dessa função.

Para a criação da função sigmóide discreta, alguns pontos (u, y) da função sigmóide contínua foram selecionados a fim de criar um tabelamento desses pontos.

Foram feitas cinco simulações:

- Simulação usando a função contínua recursiva.
- Simulação discretizada com 90 pontos.
- Simulação discretizada com 60 pontos.
- Simulação discretizada com 42 pontos.

- Simulação discretizada com 20 pontos.

4.1.1 Discretização da função sigmóide

Para implementar em *hardware* a função sigmóide foi necessário fazer, primeiramente, uma discretização dessa função. Discretização é o tabelamento, com valores previamente escolhidos, de uma função contínua. Por exemplo, em uma das simulações foram retirados da função sigmóide 60 pontos (u,y) sobre o gráfico para fazer a discretização. Porém, não foram retirados ao acaso, pois alguns trechos da função sigmóide que crescem muito mais rápido que outros. Nos trechos em que u varia entre -0.5 e 0.5 a curva da função recebe uma variação maior dos valores de y – então dizemos que esse trecho da função cresce mais rápido. A figura 12 mostra a função sigmóide e a figura 13 mostra os pontos obtidos sobre o eixo u . Nota-se, nessa figura, que há mais pontos no intervalo propriamente dito. Portanto, é preciso pegar mais pontos das partes em que os valores de y crescem mais aceleradamente e menor número de pontos das que são mais lentas. Para determinar isso, foi criada a fórmula 4.1 para determinar a variação de um ponto para outro no eixo x (valores de entrada da função, também conhecido como u), de modo que a variação entre os valores de y não fossem maior que 2% e esses fossem praticamente o mesmo. A idéia aqui é pegar o menor número de pontos sobre a curva, de modo que não possa atrapalhar a convergência da RNA. Essa variação dos valores de u é denotada por Δu.

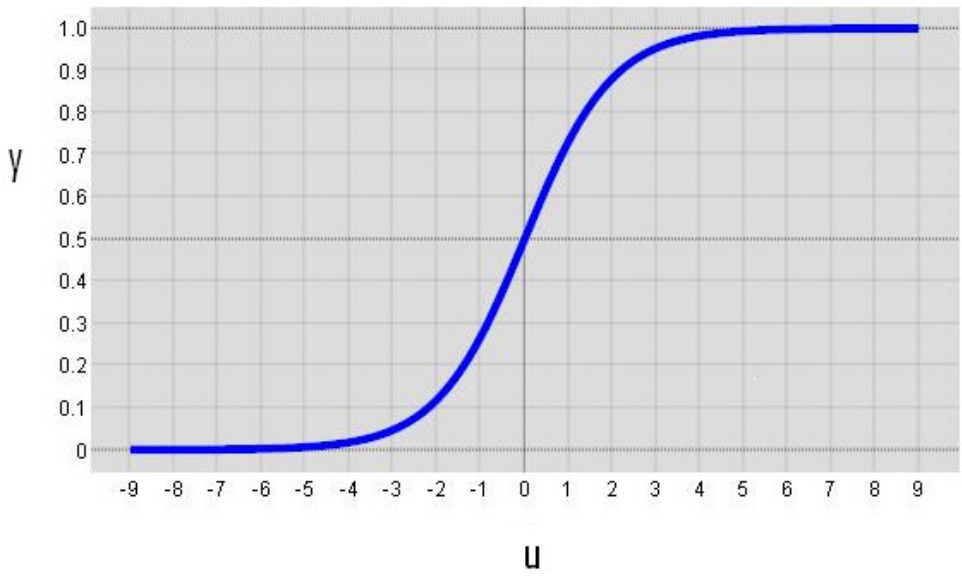


Figura 12 Gráfico da função sigmoidal contínua

$$\Delta u_{ij} = k/f'(i) \quad (4.1)$$

Em que:

- Δu_{ij} é a variação calculada valor de i(valor de u do ponto atual) para j(valor de u do ponto a ser descoberto).
- k é uma constante de variação. Seu valor é 0.018 para 60 pontos.
- f´ (i) é a derivada da função sigmoidal de i.

O valor de u do próximo ponto então será o valor de u do ponto atual mais a variação Δu (fórmula 4.2 para os positivos e fórmula 4.3 para os negativos).

$$j = i + \Delta u \quad (4.2)$$

$$j = i - \Delta u \quad (4.3)$$

em que:

- i = valor de u do ponto atual
- j = valor de u do ponto a ser descoberto.
- Δu Variação calculada na fórmula 4.1

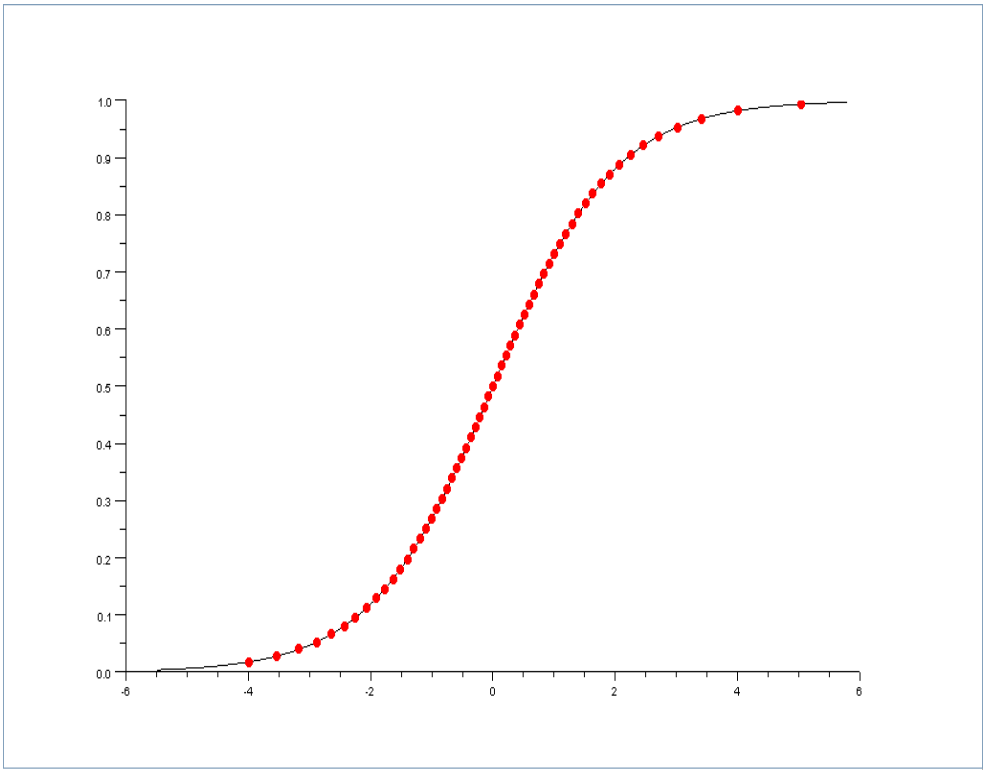


Figura 13 Gráfico da função mostrando os pontos encontrados sobre a curva sigmoidal.

Por exemplo, começa-se do ponto onde x =0, o cálculo do Δu é 0,072. Então o próximo ponto terá x = 0 + 0,072 = 0,072. O valor de y (valor da função sigmóide) correspondente será 0,517992. E assim foram achados todos os pontos.

O valor de y a ser descoberto é o valor da função sigmoidal. A tabela 1 mostra os valores encontrados para u (1ª e 4ª coluna), para y (2ª e 5ª coluna) e para Δu (3ª e 6ª coluna).

O cálculo começou em u = 0. O gráfico da função com os pontos (u,y) descobertos está na figura 13. Os pontos vermelhos na figura são os pontos descobertos e a curva preta é a curva da função. Nota-se que os pontos estão

em menor quantidade quando estão nos trechos em que a função cresce menos (ou seja, os valores de y estão mais próximos para os mesmos intervalos de u) da função e vice-versa. A figura 14 mostra o mesmo gráfico, porém em formas de escadas, para destacar a variação da distância de um ponto a outro (degrau). Será chamada no texto de degraus a partir de agora, ao invés de pontos, como referência a gráficos de escada.

Tabela 1 Valores de u encontrados

Valor de u	Valor de y	Valor de Δu	Valor de u	Valor de y	Valor de Δu
0,072	0,517992	0,072093	0	0,5	0,072
0,144093	0,535961	0,072374	-0,072	0,482008	0,072093
0,216468	0,553907	0,072847	-0,14409	0,464039	0,072374
0,289314	0,571828	0,073517	-0,21647	0,446093	0,072847
0,362832	0,589726	0,074396	-0,28931	0,428172	0,073517
0,437227	0,607598	0,075496	-0,36283	0,410274	0,074396
0,512724	0,625445	0,076837	-0,43723	0,392402	0,075496
0,58956	0,643264	0,07844	-0,51272	0,374555	0,076837
0,668	0,661055	0,080335	-0,58956	0,356736	0,07844
0,748335	0,678816	0,082559	-0,668	0,338945	0,080335
0,830894	0,696544	0,085159	-0,74834	0,321184	0,082559
0,916053	0,714237	0,088191	-0,83089	0,303456	0,085159
1,004244	0,731892	0,091731	-0,91605	0,285763	0,088191
1,095975	0,749505	0,095874	-1,00424	0,268108	0,091731
1,191848	0,767071	0,100743	-1,09597	0,250495	0,095874
1,292591	0,784585	0,106502	-1,19185	0,232929	0,100743
1,399093	0,80204	0,11337	-1,29259	0,215415	0,106502
1,512463	0,819426	0,121649	-1,39909	0,19796	0,11337
1,634112	0,836732	0,13176	-1,51246	0,180574	0,121649
1,765872	0,853943	0,144319	-1,63411	0,163268	0,13176
1,910191	0,87104	0,160244	-1,76587	0,146057	0,144319
2,070434	0,887996	0,180979	-1,91019	0,12896	0,160244
2,251413	0,904772	0,208915	-2,07043	0,112004	0,180979
2,460328	0,921313	0,248293	-2,25141	0,095228	0,180979
2,708621	0,937533	0,307353	-2,43239	0,080736	0,208915

3,015974	0,953291	0,404243	-2,64131	0,066527	0,24253
3,420217	0,96833	0,586956	-2,88384	0,052958	0,28985
4,007173	0,98214	1,026166	-3,17369	0,040168	0,358896
5,033339	0,993525	2,798109	-3,53258	0,028399	0,46687
			-3,99945	0,017996	0,652345

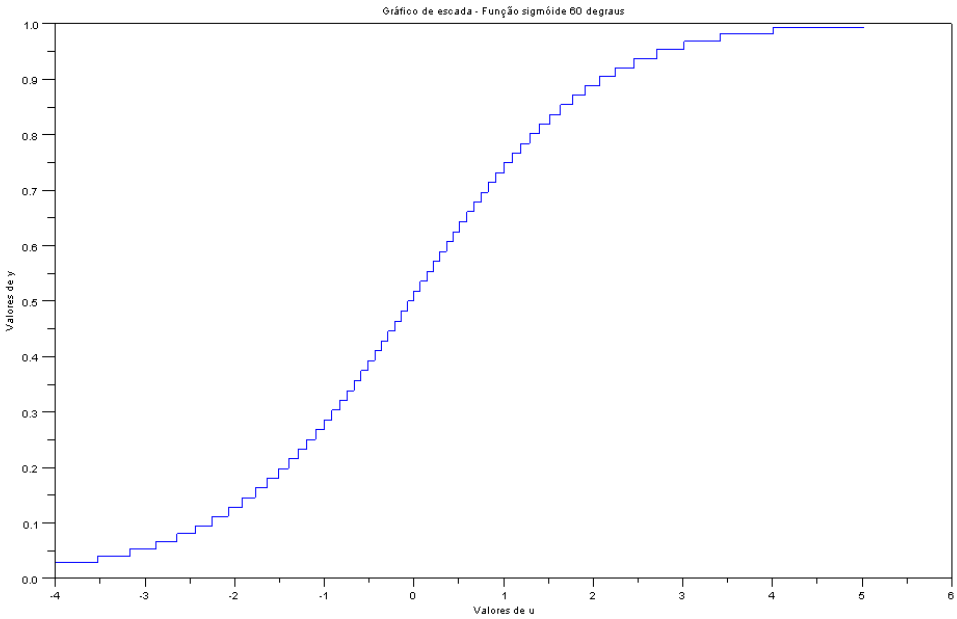


Figura 14 Gráfico da função sigmóide discretizada em 60 degraus

4.1.2 Teste do algoritmo implementado

As simulações serão testadas para resolver o problema da função lógica XOR de 4 *bits*, ou seja, a RNA vai aprender a função XOR e ela terá que responder corretamente os dados de testes. Essa função foi escolhida

porque é um problema não linearmente separável, então não pode ser resolvido somente por um neurônio e sim por uma rede que contenha pelo menos uma camada oculta.

No perceptron simples não há camadas ocultas sendo suficiente para resolver problemas linearmente separáveis. Porém, problemas não-linearmente separáveis ocorre com freqüência, sendo necessário mais de uma camada. Um desses problemas é o problema do XOR ou OR exclusivo. O padrão é de acordo com a tabela 2.

Tabela 2 O problema do XOR

A	B	A XOR B
0	0	0
1	0	1
0	1	1
1	1	0

Para este tipo de problema com dois *bits* é usado no mínimo três neurônios organizados no modelo de multicamadas (MLP). Sendo dois neurônios na camada de entrada, 3 na camada de oculta e um na camada de saída e dois valores de entrada para cada neurônio. Se aumentar o número de bits aumenta o número de neurônios.

Como já dito no início da seção será usado este problema com quatro *bits* para o problema tomar uma complexidade maior e testar se o algoritmo realmente pode ser usado em padrões muitos complexos. A tabela 3 mostra os valores da função lógica XOR com 4 *bits* e sua saída esperada.

Tabela 3 Os valores da função lógica XOR com 4 bits e sua saída esperada

A	B	C	D	Saída esperada
0	0	0	0	0
0	0	0	1	1

0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Para este problema foi necessário três camadas na rede, uma de entrada, uma escondida e uma de saída. A camada de entrada tem 4 neurônios, a camada oculta tem 16 neurônios e a camada de saída tem um. A figura 15 mostra a ilustração da rede.

O número de iterações utilizados foi cem mil. A taxa de aprendizado é 0.35 e o momentum é igual a 0.65.

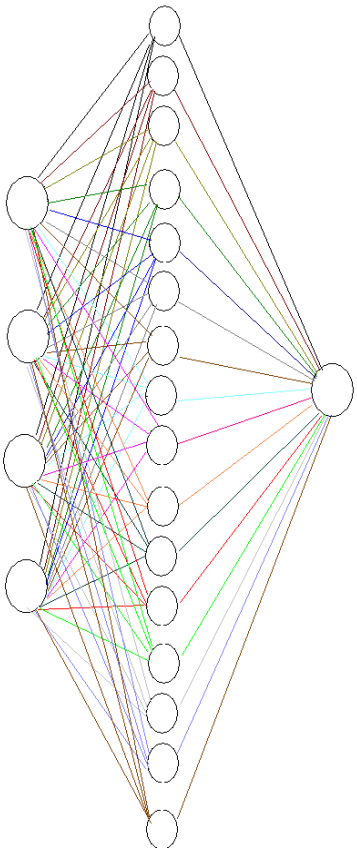


Figura 15 Rede de neurônios usado para resolver o problema do XOR

4.2 Implementação da função de ativação em *hardware*

A função em *hardware* tem sua chamada no *software* para ser executada em qualquer programa. No programa foi usado a linguagem C++ para a criação do *software* e a linguagem de descrição de *hardware* VHDL para implementação em *hardware*. É criada a função em *hardware* no processador e a chamada dessa função criada é feita no programa em C++.

As duas implementações em *hardware* são:

- Criação da função sigmóide em *hardware* com 20 degraus.
- Criação da função sigmóide em *hardware* com 60 degraus.

Foi usado o mesmo algoritmo de RNA, o mesmo problema do XOR e o mesmo tabelamento das simulações, ou seja, a mesma metodologia foi usada para a implementação em *hardware*. O que foi acrescentado no projeto para que seja feito no *hardware* vai ser mostrado nesse capítulo.

Para fazer essas tarefas de implementação foi utilizado o Kit de Desenvolvimento NIOS produzido pela Altera Corporation que contêm uma FPGA. O modelo desse kit é a Stratix II ROHS EPS2S60F672C3N e possui (ALTERA, 2009a):

- 32 MB de memória DDR SDRAM;
- 16 MB de memória Flash;
- 2 MB de memória SRAM;
- Cristal oscilador de 50 MHz;
- Possui 4 botões tipo *push-bottom;*
- 8 LED's;
- USB-blaster que conecta o kit ao computador.
- Dois display de 7 segmentos
- Interface ethernet MAC/PHY

Na figura 16 é apresentado o esquemático simplificado do kit de desenvolvimento utilizado, mostrando as conexões dos elementos citados acima com a FPGA.

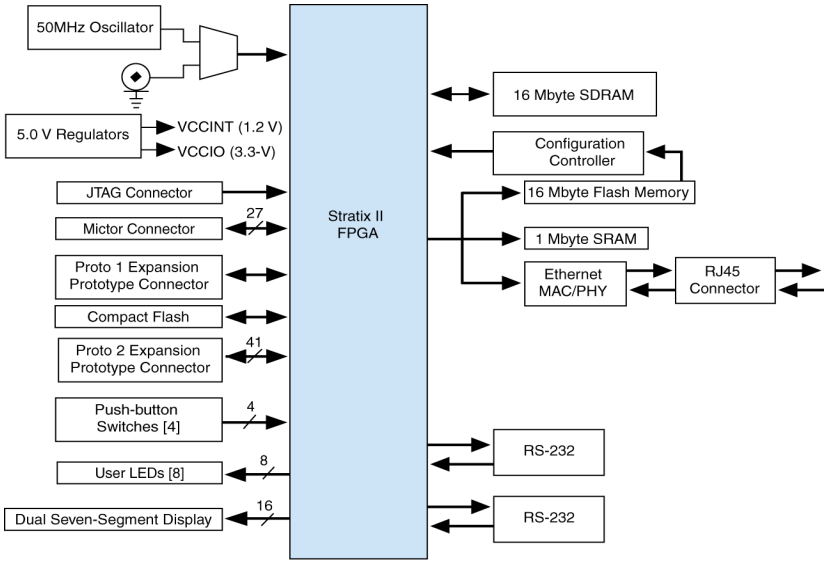


Figura 16 O modelo geral Stratix II e seus periféricos
Fonte: Altera Corporations

Para o desenvolvimento do projeto de *hardware*, foi utilizado o *software* Quartus II integrado com SOPC Builder (ALTERA, 2009b), para desenvolvimento da configuração da FPGA do kit. O SOPC Builder foi utilizado para desenvolvimento do sistema microprocessado baseado no padrão NIOS II. A configuração e o sistema microprocessado são programados no kit. O computador utilizado tem que estar conectado ao kit por um cabo USB-BLASTER, como mostrado na figura 17.

O *software* NIOS II IDE (ALTERA, 2009c) foi utilizado para desenvolvimento do programa em linguagem C/C++ que, depois de compilado e transmitido ao kit, é executado na CPU NIOS implementada na FPGA. O *software* gravador (integrado ao Quartus e NIOS IDE) grava a configuração na FPGA utilizando o cabo USB-BLASTER.

Figura 17 O kit de desenvolvimento da Altera se conecta ao PC por um cabo USB blaster.

Portanto, primeiro é preciso configurar o *hardware* que será utilizado na FPGA, usando os programas SOPC Builder e Quartus II, depois é gravado o *software* criado no NIOS II IDE. Nas seções 4.2.1 e 4.2.2 será abordado um pouco mais sobre esses *softwares*.

4.2.1 O *software* Quartus II integrado com SOPC Builder

É preciso "programar" o *hardware* do kit, colocando quais componentes serão utilizados. Isso é feito usando o Quartus II e o SOPC Builder. Nesses *softwares* contem um conjunto de componentes lógicos que auxiliam nessa tarefa. Por exemplo, se a implementação precisar somente de memória RAM, *push-bottons* e LED´s, não será necessário colocar outros dispositivos. Essa opção é muito interessante, pois se pode assim chegar perto do exemplo de uma placa real, que tem alguns componentes e outros não. Nas figuras 18 e 19 são mostradas as interfaces principais do Quartus II e SOPC Builder, respectivamente.

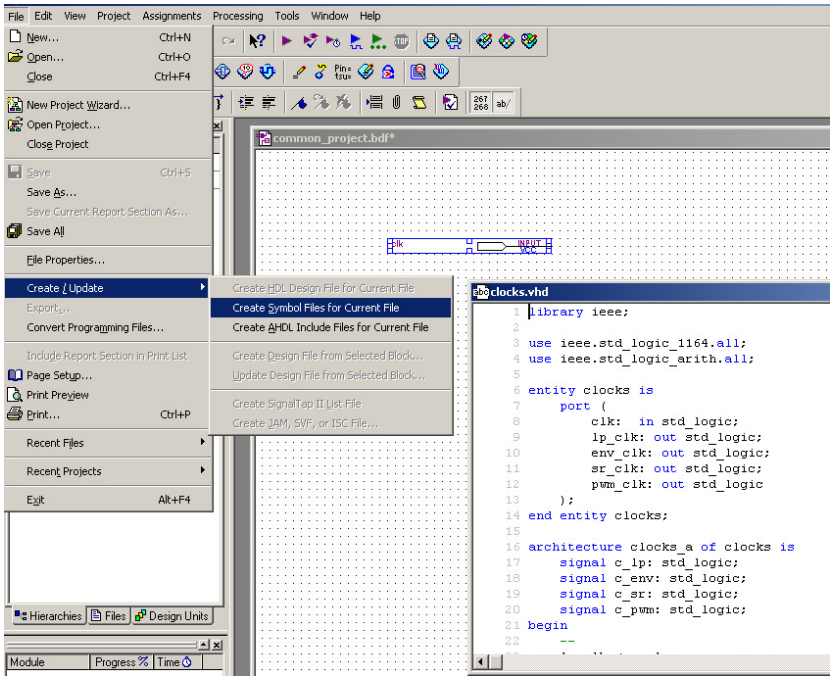


Figura 18 Software Quartus II
Fonte: Altera Corporations

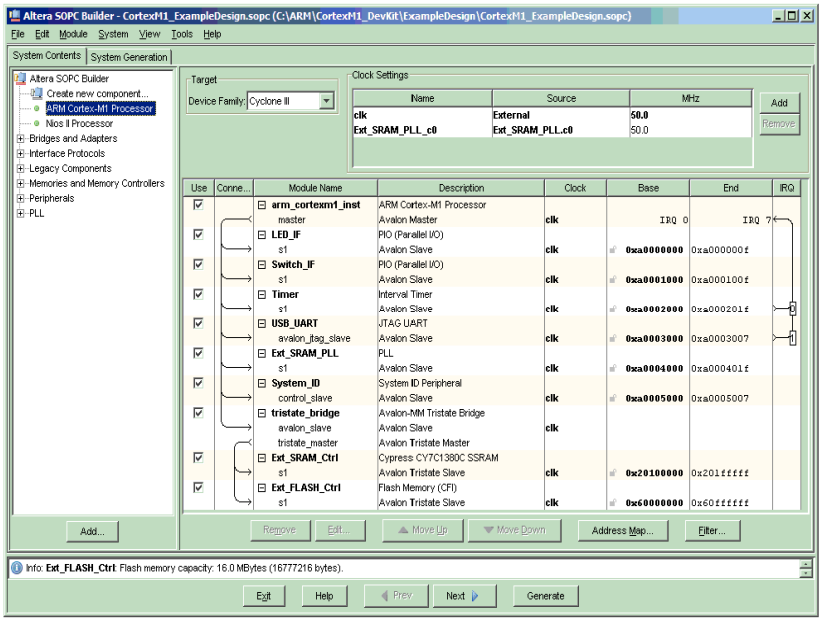


Figura 19 Software SOPC Builder
Fonte: Altera Corporations

Além disso, podem-se alterar as configurações dos componentes do kit, como, valor de oscilação do clock, criar novas instruções dentro do processador NIOS e muitas outras. Os tutoriais de como usar o *software* e outras informações estão disponíveis em (Altera, 2009).

4.2.2 NIOS IDE

Ao terminar a programação em *hardware*, cria-se o programa em C ou C++ que depois de compilado será executado no kit de desenvolvimento. Esse programa necessita de uma IDE (Integrated Development Environment). Será usado o NIOS IDE criado também pela Altera Corporations (2009c).

O NIOS IDE é uma poderosa ferramenta que auxilia o programador na criação, na compilação, ao gravar o programa no kit e na sua execução. A figura 20 mostra a interface do *software*.

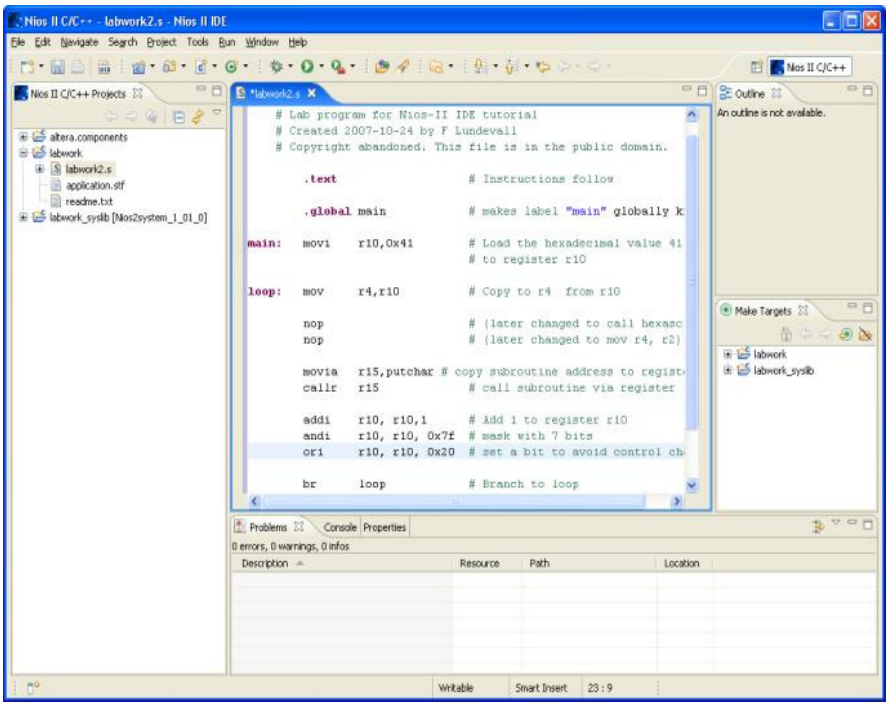


Figura 20 Software NIOS IDE
Fonte: Altera Corporations

4.2.3 Criação da instrução sigmóide em *hardware*

A instrução criada foi feita usando a linguagem de descrição em *hardware* (VHDL) e os *softwares* já citados nas seções anteriores. A figura 21 ilustra o conceito de criação de instruções dentro do processador embarcado NIOS II. Além das funções de soma, subtração, shift e várias outras dentro da ALU, também foi acrescentado a instrução de cálculo sigmoidal. Essa função acrescentada é chamada de Instrução customizada (em inglês, *Custom Logic Instruction*).

Em Chactor (2009) pode ser acessado o tutorial de como criar instruções customizadas no NIOS II.

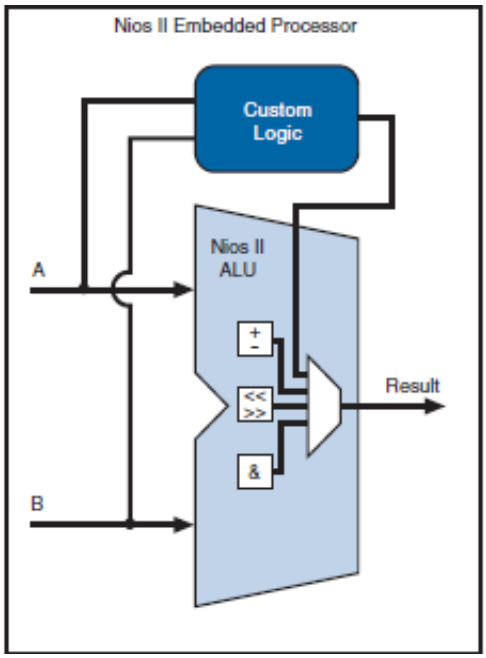


Figura 21 Instrução de hardware acrescentada ao processador NIOS II
Fonte: Altera Corporation (2009a)

O algoritmo da criação da função customizada em VHDL está no apêndice A e no apêndice C encontra-se o esquemático dessa função.

A implementação em VHDL usa os valores em binário, portanto, a primeira parte a ser feita é passar os valores da tabela 3.1 para ponto-flutuante (PF) de acordo com a norma IEEE 754. Esta norma recomendada pelos institutos ANSI (*American National Standard Institute*) (1985) e IEEE (*Institute of Electrical and Eletronic Engineers)*, estabelece o padrão a ser seguido pelos fabricantes de computadores e construtores de compiladores de linguagens científicas, ou de bibliotecas de funções matemáticas, na utilização da aritmética binária para números de PF. O apêndice B tem a tabela completa com os valores de PF usados a partir dos dados da tabela 4.1.

A IEEE 754 adota ponto flutuante para precisão simples e precisão dupla. Neste trabalho foi utilizado somente com precisão simples, pois foi usado PF para 32 *bits*. Na precisão simples divide-se a representação da seguinte maneira: usa-se 1 bit para sinal, ou seja, se é positivo ou negativo, 8 *bits* para expoente e 23 para a Mantissa. A figura 22 faz a representação dessa divisão. Como se faz a padronização pode ser encontrada em ANSI/IEEE(1985).

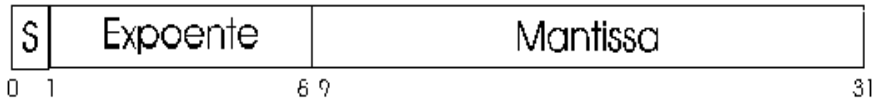


Figura 22 Campos usados para PF na IEEE 754

Foram usados comparadores de *menor que* para situar o ponto u dentro de qual faixa de valores esse valor está. Por exemplo, se passado por parâmetro o valor 1.5, o comparador procuraria na tabela (seria uma memória dentro da ALU que contêm todos os valores de u) os valores acima e abaixo dele. No caso, 1.5 está entre 1.3990 e 1.5124 (Ver tabela 1). Então, o registrador indica qual dado na outra memória que contêm o valor de y correspondente (Também está na tabela 1). E esse valor de y é a saída da

função. Na figura 23 mostra um comparador que recebe um valor em dataa que foi passado por parâmetro da função sigmóide e datab que recebe os valores de u, para comparação, da memória. É então comparado se dataa é menor que datab e a resposta - 1 ou 0 - é retornado em alb.

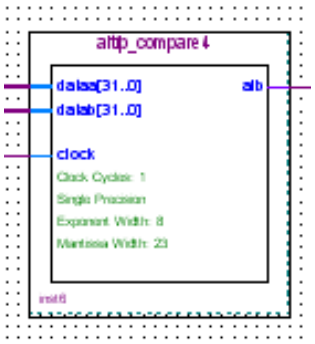


Figura 23 Comparador menor que

4.2.4 Teste do algoritmo implementado

Foi primeiramente proposto como que seria exigido do *hardware*, uma forma do usuário "perguntar" e este responder. Então, a solução foi colocar os 4 botões da FPGA como operandos de entrada – A,B,C,D da tabela 3- e somente 1 LED responder – fazendo o papel da saída esperada do neurônio da tabela 3. Os dados de treinamento chegam por um arquivo que está no PC que conectou com a placa por meio de um conector USB-BLASTER e os dados de teste vieram dos *push-bottom*. Se o botão está apertado, o valor é 1, senão o valor é 0. A saída esperada aparece no LED, se é 1, é porque está aceso, senão está apagado. Esse sistema de LEDs e botões mostrarão se a RNA está mostrando os resultados corretos. O esquema da figura 24 mostra a interação dos botões com a FPGA e com o LED.

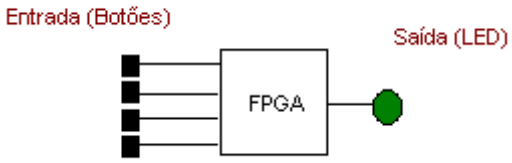


Figura 24 Esquema representando os 4 botões de entrada e o LED integrado com a FPGA com a RNA implementada.

5 RESULTADOS E DISCUSSÃO

Nesse capítulo serão mostrados os resultados obtidos com as simulações e com as três implementações no processador NIOS II – dois com instrução customizada e uma sem a instrução customizada - abordando se elas foram eficazes e se as funções customizadas realmente deixaram o treinamento mais eficiente. Se sim, quais delas foram as mais rápidas. Será feito um quadro comparativo mostrando quanto tempo gastou cada uma das implementações (foram chamadas nesse texto de Sistema sem a instrução customizada e Sistema com a instrução customizada) e quantos registradores e dispositivos lógicos necessitou cada uma. Nas próximas seções estão os resultados do projeto.

5.1 Simulações em *software* no computador

Foram feitas simulações com 20, 42, 60 e 90 degraus. Todas elas tiveram a mesma exatidão ao resolver o problema do XOR, como era de se esperar. Além disso, quanto menor for o número de degraus tem a função simulada, menos tempo gasta-se na execução do algoritmo. Isso pode ser visto na figura 25 no gráfico Tempo(s) X Época. O eixo horizontal é o número de épocas e o eixo vertical representa o tempo em segundos. Nota-se que a reta em preto, que representa nesse gráfico a função sigmóide recursiva (número de degraus tendendo a infinito), está com uma inclinação muito maior que as outras simulações, que seguem o princípio de que, menor o degrau, menor é o tempo gasto no treinamento da rede. Na figura 25, a simulação com 20 degraus, está em vermelha e foi a que gastou menos tempo em 10 épocas. Seguido dela, tem-se a simulação com 42 degraus, em verde, a simulação com 60 degraus, em azul e a simulação com 90 degraus, em amarelo.

A figura mostra que a função sigmóide implementada com valores discretos gasta menos tempo que com valores contínuos.

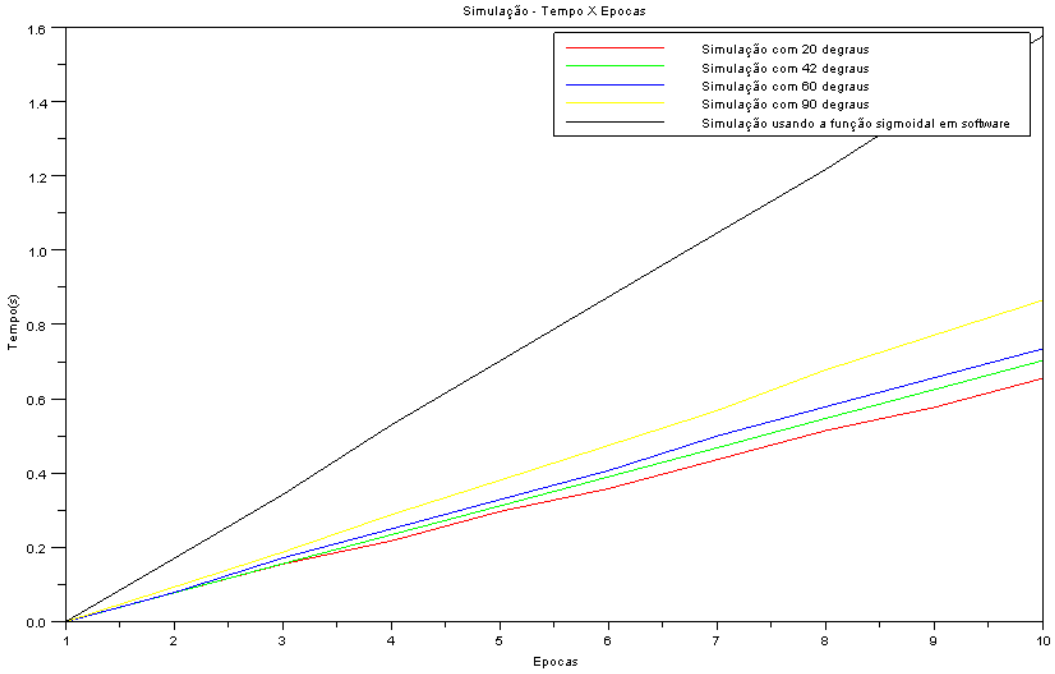


Figura 25 Várias simulações feitas - Gráfico Tempo X Época

Se o tempo fica menor usando funções com degrau menor, por outro lado, o erro médio quadrado, é claro, fica maior quando discretizamos um número menor de pontos para o domínio da função. Na figura 26 mostra o erro médio quadrado em função do número de épocas. O sistema de cores usado foi o mesmo da figura 25 e será utilizado para todos os gráficos de simulações nesse capítulo. O número de épocas está no eixo horizontal e MSE está no eixo vertical. Nota-se, primeiramente, que a simulação da função sigmóide recursiva convergiu rápido, em duas épocas. Ampliando o gráfico da segunda época em diante – o que é observado no gráfico 27 – pode ser observado várias coisas.

O primeiro ponto a ser observado é que as simulações com 90 e 42 degraus convergiram na época 3. Outro ponto é, na terceira época tem-se a impressão de que o sistema com 20 degraus também convergiu, porém o

gráfico deu um salto e o erro aumentou novamente. E depois da época 5 ele converge totalmente. A simulação com 60 degraus convergiu somente na quarta época. Portanto, a RNA treinada com a função sigmóide contínua tem o seu erro próximo de zero em menos épocas que as outras simulações.

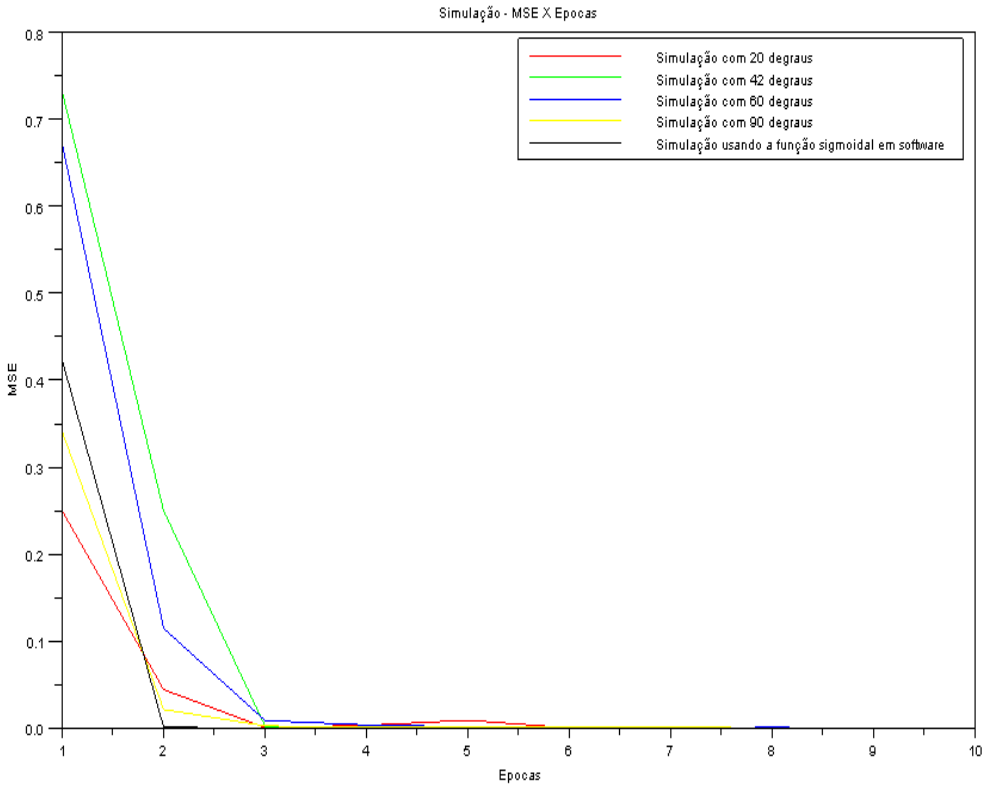


Figura 26 Várias simulações feitas – Gráfico MSE X Época

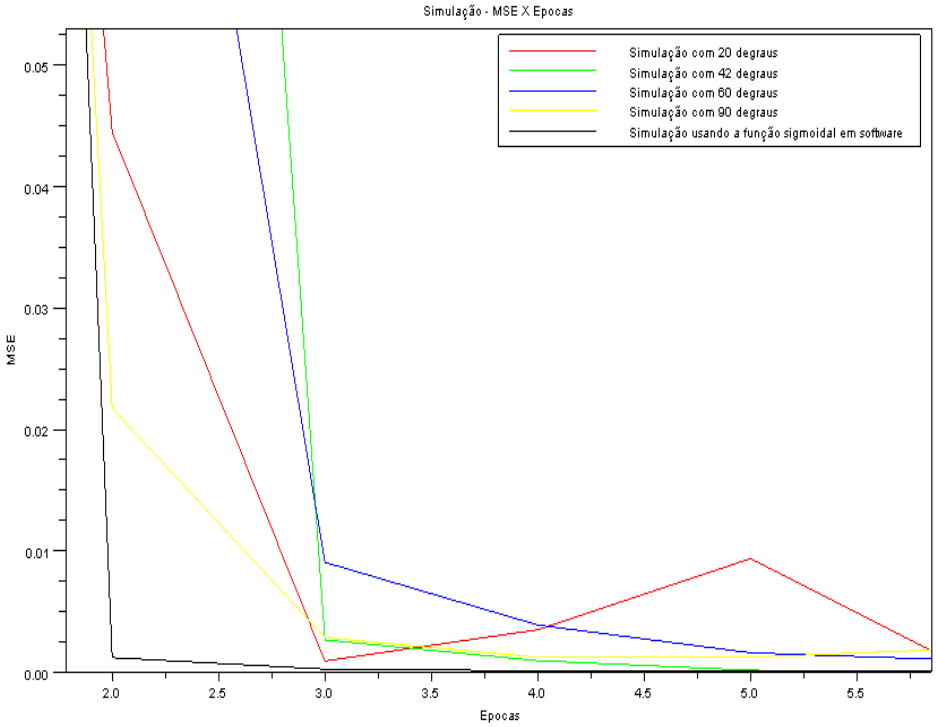
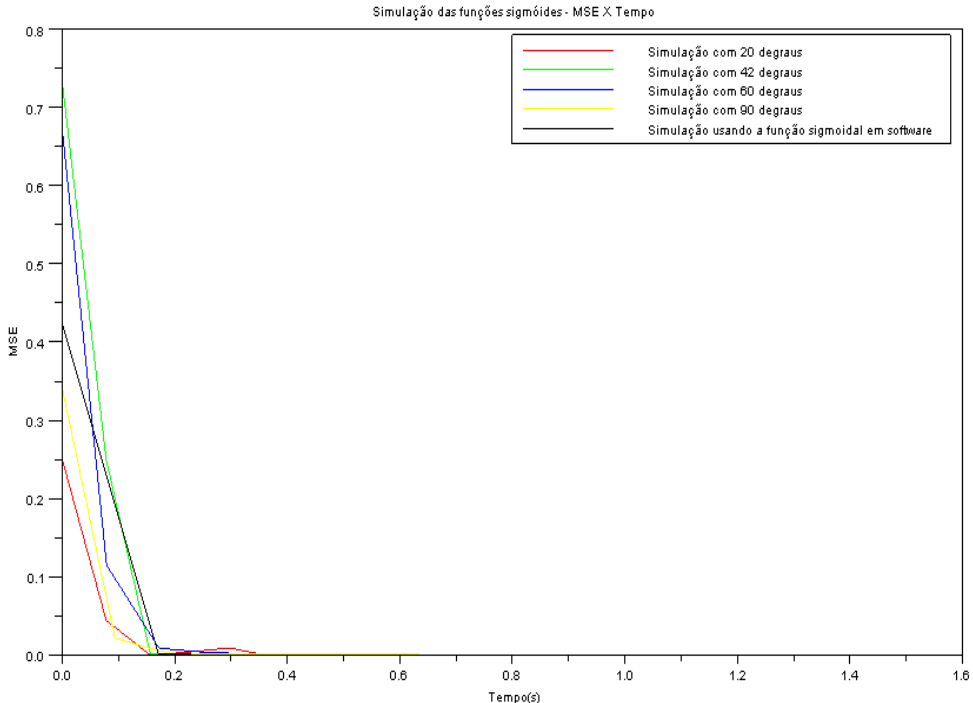


Figura 27 Detalhe ampliado do gráfico da figura 26 a partir da segunda época.

Para visualizar melhor os resultados, foi feito também a figura 28



que mostra o MSE durante o tempo da simulação.

Figura 28 Gráfico MSE X Tempo(s) das simulações

No gráfico pode-se observar que a simulação usando a função sigmoidal contínua não foi a que gastou menos tempo que todas as outras para convergir. Se ampliar a partir do instante de tempo 0.1s, como na figura 29, pode-se interpretar que:

- A simulação com 20 degraus conseguiu chegar antes de todos as outras simulações no MSE menor que 0.01, porém, ela deu um ressalto e a sua curva subiu acima de 0.01. Somente conseguiu voltar abaixo desse valor alguns segundos depois.
- A simulação com 42 degraus realmente convergiu primeiro, se considerar MSE menor que 0.01 seja satisfatório para RNA estar convergido, que todas as outras e se estabilizou sem ressaltos. Seguido delas, a ordem cronológica de simulações que convergiram é, a implementação contínua, a implementação com 90 degraus, com 60 degraus e por último a com 20 degraus, respectivamente.
- Entretanto, se considerado o MSE menor que 0.03, a simulação com 90 degraus é mais rápida. Se for considerada MSE menor que 0.05, a melhor simulação em termos de menor tempo é a simulação usando a função sigmóide com 20 degraus.

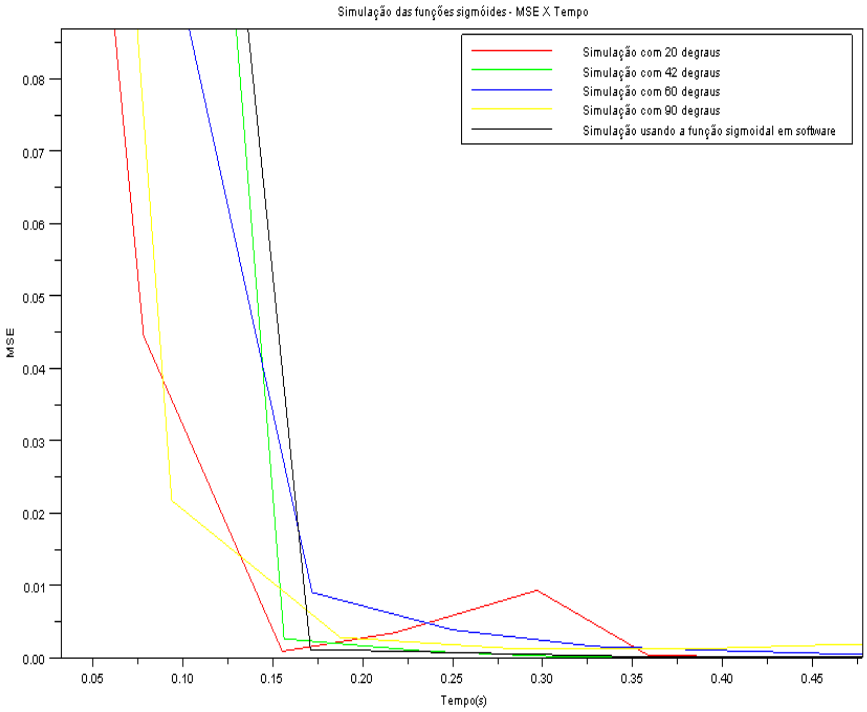


Figura 29 Detalhe ampliado do gráfico da Figura 28

Portanto, os gráficos das simulações provam que discretizando a função de ativação sigmoidal, há convergência da RNA e que pode ser obtido um tempo menor. Então, a implementação em *hardware* pode ser feita.

5.2 Sistema implementado no NIOS II

Nessa seção será mostrado os resultados obtidos com as implementações em *hardware* da função de ativação sigmoidal. Na simulação foram feitos várias funções sigmóides com número de degraus variados. No NIOS II, será feito somente as implementações em *hardware* sem a instrução customizada, e as com instrução customizada com 20 degraus e 60 degraus. Isso porque não é necessário fazer todos para mostrar

que a simulação está certa e também, porque o tempo de projeto aumenta muito, pois gasta-se muito tempo implementando em *hardware*.

A seguir aparece a figura 30 com o gráfico Tempo X Época.

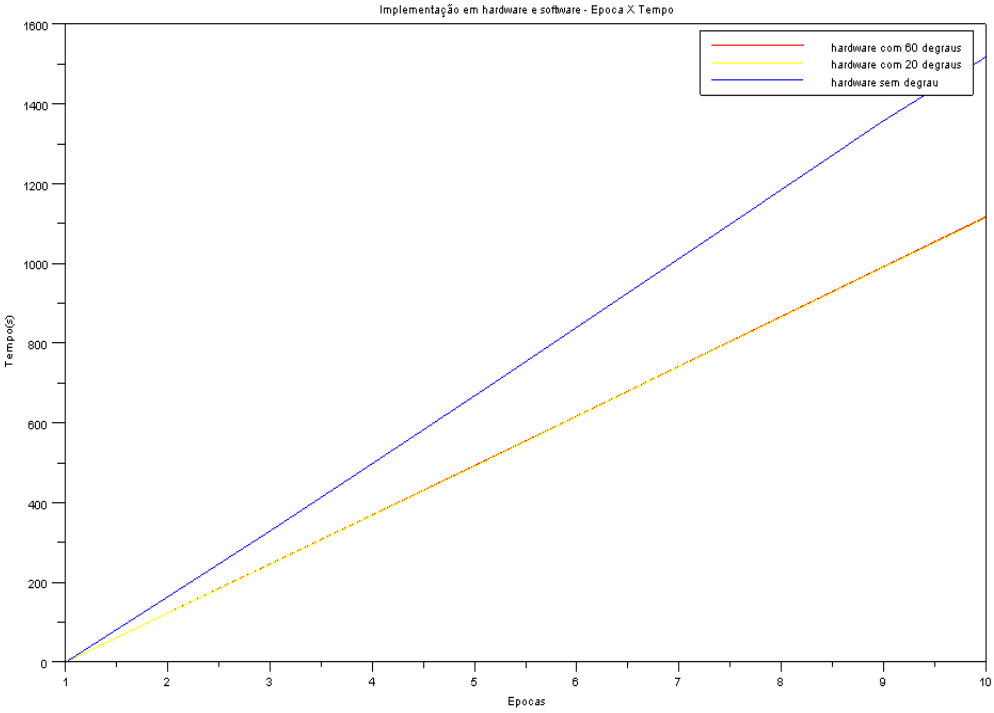
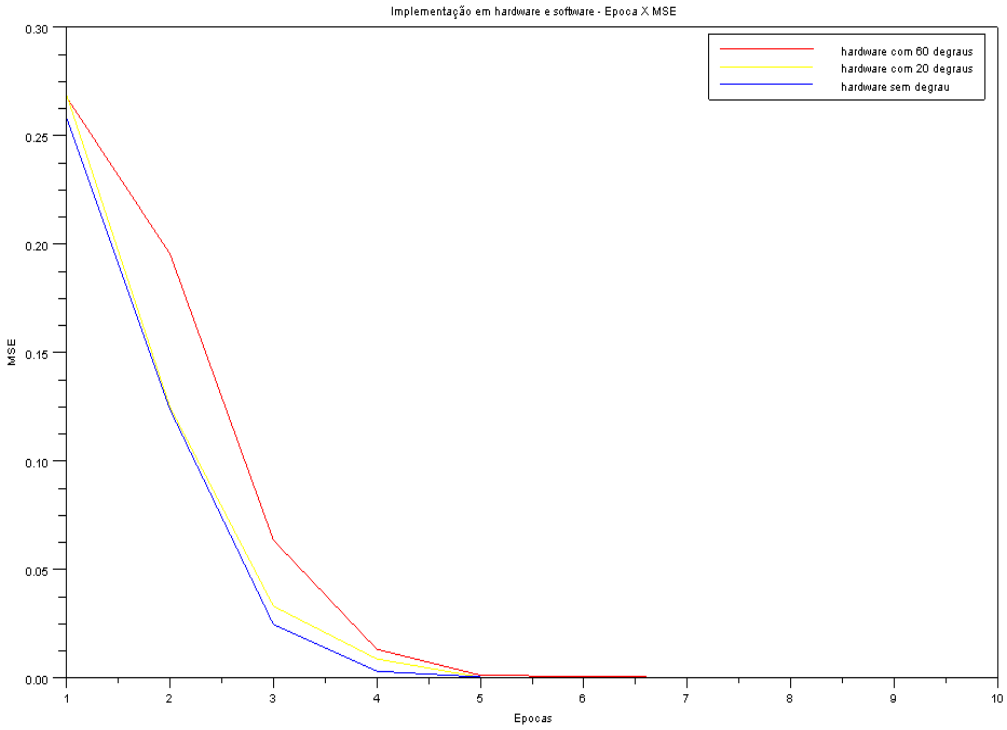


Figura 30 Gráfico da implementação em hardware mostrando o tempo em relação as épocas

Como já foi falado, o programa da RNA foi feita para durar 10 épocas, mas a sua convergência já pode se vista sobre 3 épocas apenas, porém para análise de MSE é interessante ver o que acontece se para outro problema que dure mais épocas é implementado essa RNA. No gráfico 30, mostra-se que a implementação em *hardware* discretizada em 10 épocas, gastou-se menos tempo, cerca de 6 minutos a menos. O interessante é que a simulação com 20 degraus praticamente durou o mesmo tempo que a com 60 degraus, cerca de alguns segundos de diferença, ao contrário da simulação que pode ser visto uma diferença maior entre as duas retas, vide gráfico da figura 25.

No gráfico 31, pode ser observado o que acontece com o MSE de cada implementação durante as épocas. Como é de se observar, a função contínua converge antes de todas outras funções discretizadas, porém há uma diferença do que acontece durante as simulações. A instrução de *hardware* com 20 degraus tem um MSE bem menor que o de 60 degraus até a quarta



época.

Figura 31 Gráfico da implementação em hardware mostrando o MSE em relação as épocas

Ampliando na área do gráfico em que está acontecendo a convergência, temos a figura 32. Pode-se observar o seguinte nessa figura:

- Se o MSE desejado for menor que 0.01, tanto a implementação contínua em *hardware* quanto a discretizada com 20 degraus, convergem na 4ª época,

enquanto a com 60 degraus atinge a convergência na 5ª época.

- Se o MSE for menor que 0.03, somente a implementação em *hardware* sem degrau converge na terceira época.
- Se o MSE for igual a 0.05, todos atingem convergência na 3ª época.

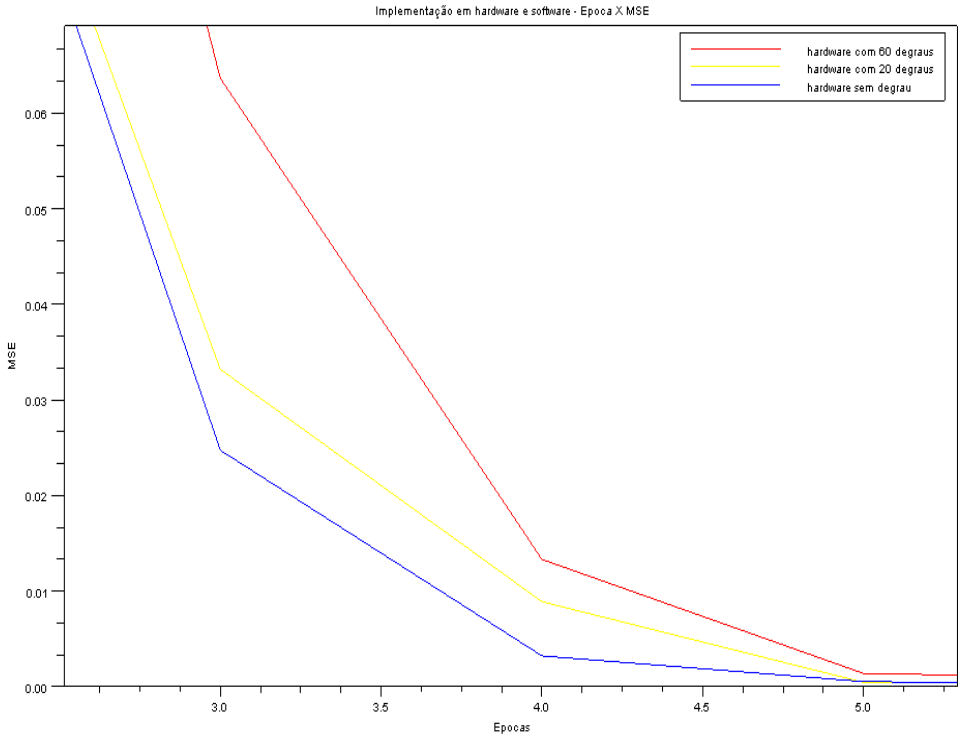


Figura 32 Detalhe ampliado do gráfico da figura 31

Nas figuras 31 e 32 foram mostrados qual implementação convergiu em menos épocas. Um outro tipo de análise deve ser feito agora, o de MSE X Tempo. Assim, mostrará qual realmente convergiu primeiro. O gráfico da figura 33 mostra esta análise.

Pode-se observar, primeiramente, que a implementação com 20 degraus convergiu em menos tempo, qualquer que for o MSE mínimo. As outras implementações teve os gráficos entrelaçados em vários pontos. Para

vê-los melhor, foi ampliado o gráfico na área em que começa a acontecer a convergência. Isso é mostrado da figura 34. Pode-se interpretar do gráfico que:

- A implementação sem degrau demorou alguns segundos a menos que o de 60 degraus para chegar ao MSE igual a 0.03.
- Se considerar o MSE igual a 0.01, a função customizada com 60 degraus gasta menos tempo - somente alguns segundos, para convergir.
- Um ponto interessante a analisar é que em *hardware*, a implementação com 20 degraus não houve sobressaltos que ocorreu nas simulações.
- Apesar da implementação da função sigmóide em *hardware* acontecer em número menor de épocas, essas duram mais tempo que a implementação discreta.

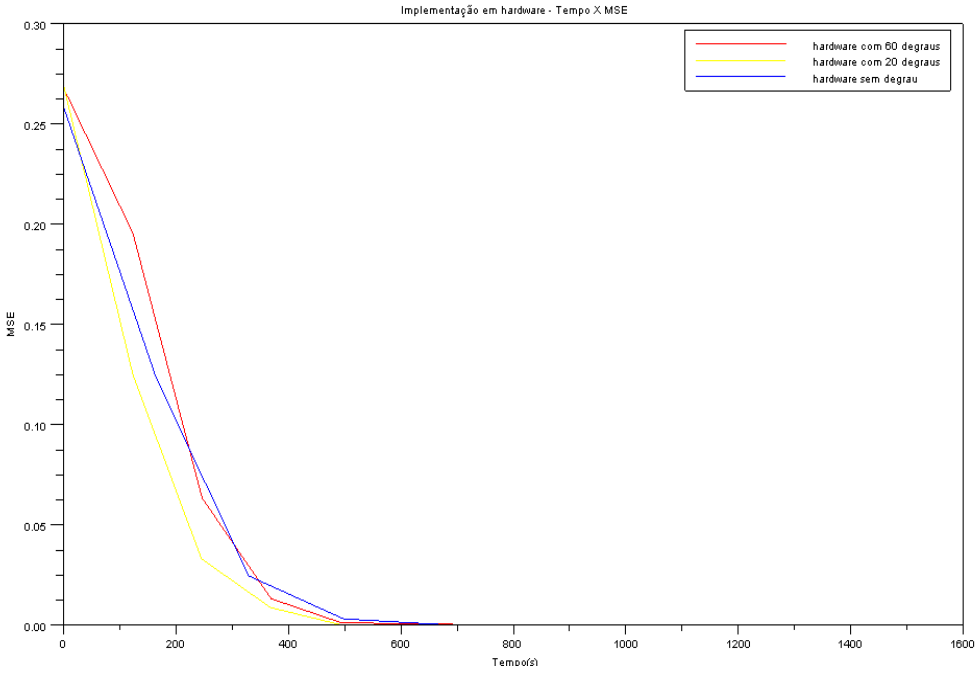


Figura 33 Gráfico da implementação em hardware TEMPO X MSE

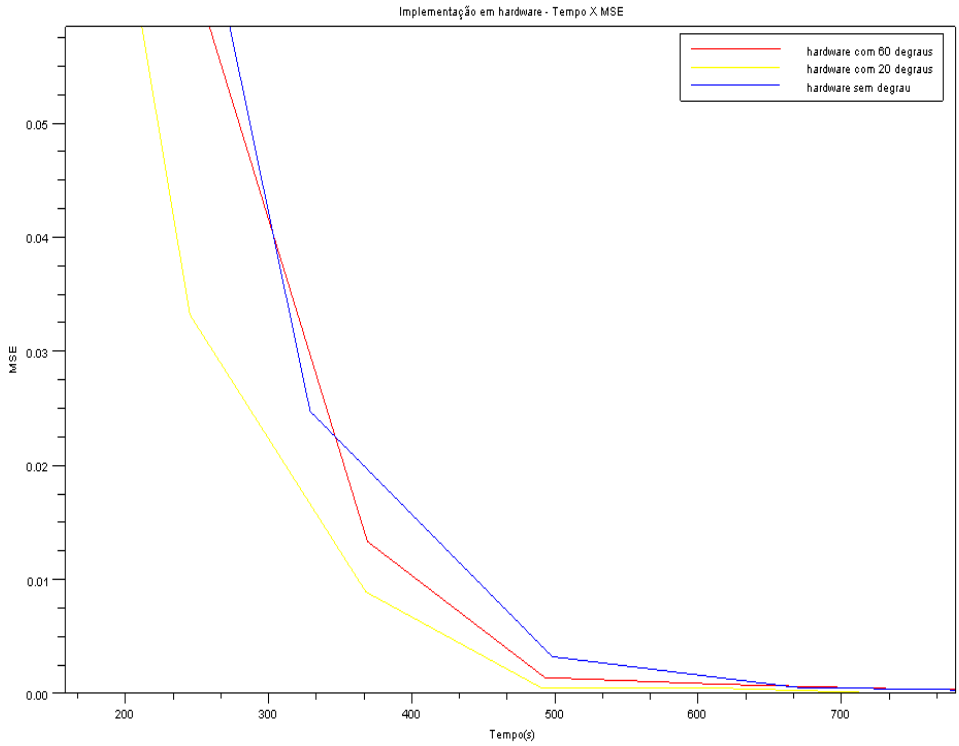
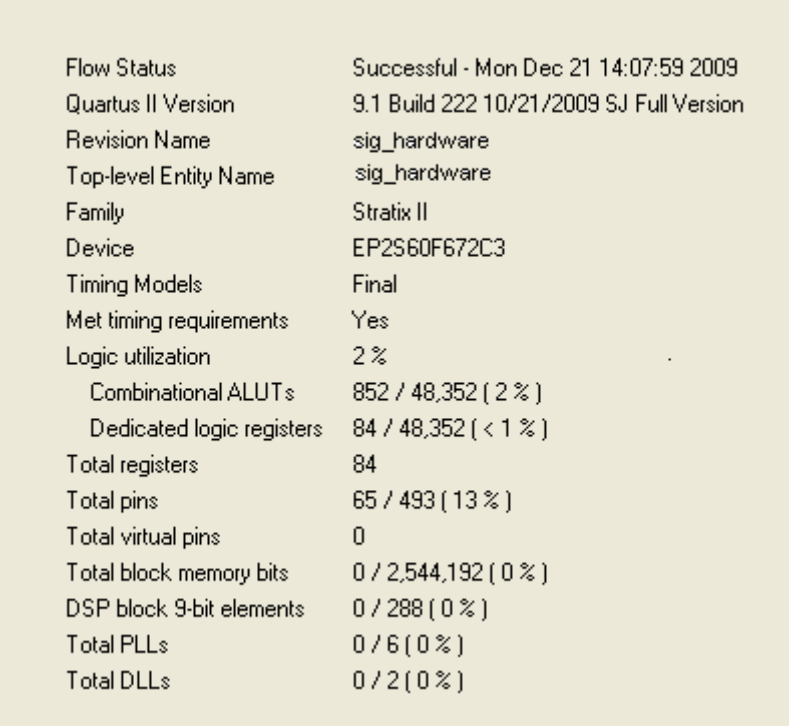


Figura 34 Detalhe ampliado do gráfico da figura 33

Portanto, observando os gráficos anteriores e considerando que a convergência do algoritmo de treinamento da RNA acontece quando o MSE for menor que 0.01, o treinamento da RNA com a função de ativação sigmóide contínua dura 560 segundos, equivalente a 9 minutos e 20 segundos, enquanto o treinamento da mesma RNA, porém usando função de ativação sigmóide discreta com 20 degraus , gastou 355 segundos, equivalente a 5 minutos e 55 segundos. Isso mostra que essa função gastou 3 minutos e 25 segundos a menos que aquela, durante as 4 épocas necessárias para a convergência da rede nos dois casos.

Com a criação dessas funções em *hardware* são gastos mais dispositivos lógicos, como é de se esperar, porém os resultados para isso foram satisfatórios. A função sigmoidal com 60 degraus gastou somente 2% a mais de dispositivos lógicos da FPGA utilizada. Somente foram usados 84 registradores e 852 ALUTs combinacionais a mais. Isso pode ser visto pela

figura 35 no relatório de saída do Quartus II, que apresenta quantos dispositivos lógicos foram gastos somente para a função sigmóide (cujo nome no programa é sig_hardware). Ou seja, além dos dispositivos lógicos que estavam sendo usados no sistema, com a adição da função sig_hardware, foram utilizados 2% a mais. A função sigmoidal com 20 degraus utilizou somente 65 registradores 353 ALUTs combinacionais a mais, cerca de 1% de dispositivos lógicos da FPGA utilizada para o projeto.



Flow Status	Successful - Mon Dec 21 14:07:59 2009
Quartus II Version	9.1 Build 222 10/21/2009 SJ Full Version
Revision Name	sig_hardware
Top-level Entity Name	sig_hardware
Family	Stratix II
Device	EP2S60F672C3
Timing Models	Final
Met timing requirements	Yes
Logic utilization	2 %
Combinational ALUTs	852 / 48,352 (2 %)
Dedicated logic registers	84 / 48,352 (< 1 %)
Total registers	84
Total pins	65 / 493 (13 %)
Total virtual pins	0
Total block memory bits	0 / 2,544,192 (0 %)
DSP block 9-bit elements	0 / 288 (0 %)
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 2 (0 %)

Figura 35 Relatório do Quartus II da função de hardware criada (sig_hardware)

Para comprovar a eficiência da função sigmóide em *hardware*, foram feitos testes separados do treinamento da RNA. Primeiramente, foi calculado o tempo médio gasto para que uma função seja executada uma vez apenas. A função sem a instrução customizada gastou 0.0004361s para executar a função. O sistema com a instrução customizada com 20 degraus gastou 80,81 x 10^{-9}s e a com 60 degraus, 87,81 x 10^{-9}s. A função

customizada gastou muito menos tempo que a função recursiva. Para mostrar que realmente foi otimizado, foi executado essas mesmas funções 100 mil vezes com valores de entradas diferentes. A partir disso, pode ser verificado que a função em *hardware* gastou muito menos tempo do que a em *software*. Esses tempos calculados para cada situação e a quantidade de dispositivos lógicos gastos por cada um pode ser verificado na Tabela 4.

Tabela 4 Resultados do tempo de execução obtido e da utilização dos elementos lógicos para cada tipo de implementação.

Implementação	Tempo gasto em uma iteração	Tempo gasto em 100 mil iterações	Número de Registradores Utilizados	Número de ALUT combinacionais
Sistema sem a instrução customizada	0,0004361s	43,61s	2901	3868
Sistema com a instrução customizada com 20 degraus	80,81 x 10^{-9}s	8,081 x 10^{-3} s	2966	4191
Sistema com a instrução customizada com 60 degraus	87,81 x 10^{-9}s	8,781 x 10^{-3} s	2985	4720

Portanto, tanto a implementação com instrução customizada e sem a instrução customizada realizaram a mesma tarefa com sucesso: executar a função lógica XOR de 4 entradas. Ou seja, o treinamento da rede neural foi alcançado. Porém, a implementação com a instrução customizada gasta menos tempo de execução que a função recursiva contínua. A vantagem de utilizar a implementação sem a instrução customizada é que essa vai utilizar menos dispositivos lógicos que aquela.

6 CONCLUSÃO E TRABALHOS FUTUROS

Foi mostrado nesse trabalho como foi implementada a função sigmóide em *hardware*, toda a metodologia usada e a importância das FPGA's que contribuíram – no sentido de facilitar – a criação da função em *hardware*. Foram apresentados trabalhos na área de Redes Neurais Artificiais e seu campo vasto: em sensores, na prevenção de arritmias cardíacas, previsão de vendas, biometria, distúrbios da rede elétrica e vários outros.

O objetivo desse trabalho, que era otimizar a RNA em *hardware*, foi atingido. A implementação da função customizada sigmóide – como foi visto, muito importante para a convergência da RNA – no processador NIOS melhorou o tempo de execução da RNA, com a mesma exatidão dos resultados da função sem customização e com poucos dispositivos lógicos a mais que o mesmo.

Como já foi criada a função sigmóide em *hardware*, pode-ser dar a chance de outros projetos irem mais a fundo na implementações da RNA em *hardware*, por exemplo, implementar todo o neurônio artificial.

Como já dito no capítulo 2, usa-se RNA na indústria automotiva. Algumas peças necessitam de uma maior velocidade, pois, por exemplo, freios anti-bloqueantes, transmissões automáticas e injeção eletrônica precisam ser usados em um curto espaço de tempo. Seria ideal a sua implementação em *hardware*. Para uso em outros tipos de sistemas embarcados também seriam interessantes a criação de instruções em *hardware* de RNA, pois estas costumam usar pouca memória e processamento, como no caso de celulares, sensores e outros tipos de aplicações.

Com menor número de ciclos de clocks gastos na função customizada criada nesse projeto pode-se inferir que há uma redução de gasto de energia desses sistemas embutidos. Sensores e celulares que usam

baterias poderiam ser recarregados um número menor de vezes. Como não é do trabalho medição de gasto de energia não foi mostrado nenhum dado ou teste preocupado com essa área.

Portanto, para os próximos trabalhos, há algumas etapas de RNA importantes para serem alcançadas:

- Implementação de todo o modelo do neurônio artificial em *hardware*, para obter maiores ganhos no tempo de execução da RNA.
- Novas técnicas de criação de RNA em *hardware*.
- Usar as metodologias e resultados desse projeto para criar alguma aplicação importante.
- Comparar se a utilização de funções customizadas em *hardware* reduz ou aumenta o gasto de energia em sistemas embarcados.

7 REFERÊNCIAS BIBLIOGRÁFICAS

Alippi, Cesare; Nigri, Meyer. **Hardware requirements for digital vlsi implementation of neural networks**. IEEEXplore, pages 1873 - 1878, 1991.

Altera Corporation. **Stratix II Device Handbook**. San Jose, 2009.

Altera Corporation. Quartus II Handbook v9.1. San Jose, 2009.

Altera Corporation. **Nios II Software Developer's Handbook.** San Jose, 2009.

American National Standards Institute / Institute of Electrical and Electronics Engineers: **IEEE Standard for Binary Floating-Point Arithmetic***,* ANSI/IEEE Std 754-1985, New York, 1985.

Blake, J. J., Maguire, L. P., McGinnity, T. M., Roche, B. and McDaid, L. J.. **The implementation of fuzzy systems, neural networks and fuzzy neural networks using fpgas**. Information Sciences, 112:151 - 168, 1998.

Borja, Jose Antonio Tumialan. **Automatização e controle inteligente on-line de sistemas de refrigeração utilizando redes neurais artificiais.** Tese de doutorado. Universidade Federal de Uberlândia/ENGENHARIA MECÂNICA, 2007.

Braga, Antônio de Pádua; de Carvalho, André Ponce de Leon F; Ludemir, Teresa Bernarda. **Redes Neurais Artificiais: Teoria e Aplicações.** 2ª edição, LTC, 2007.

Chactor. Exemplo de projeto que cria instrução personalizada original do NIOS II. Disponível em: http://blog.ednchina.com/chactor/193212/message.aspx. Consultado em 28/10/2009.

Dou, Yong; Xia, Fei; Jiang, Jingfei. **Fine-grained Parallel Application Specific Computing for RNA Secondary Structure Prediction Using SCFGs on FPGA.** Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems. National Laboratory for Parallel & Distributed Processing. National University of Defence Technology. China. p. 107-116. 2009.

Fabbrycio A.C.M; Cardoso, Marcelo Augusto Costa Fernandes. **Tópicos em Comunicação. FPGA e Fluxo de projeto.** 2007. Disponível em: http://www.decom.fee.unicamp.br/~cardoso/ie344b/Introducao_FPGA_Fluxo_de_Projeto.pdf. Consultado em 16/10/2009.

Granado,J. M.; Vega, M. A., Pérez, R., Sánchez, J. M., Gómez, J. A. **Using FPGAs to Implement Artificial Neural Networks**. IEEEXplore, 2006.

Gupta, G.; Mccabe, A. **A review of dynamic handwritten signature verification**. Technical Report - James Cook University, Australia, 1997.

Hassan, A. A.; Elnakib, M.; Abo-Elsoud, M. FPGA-Based Neuro-Architecture Intrusion Detection System, **Proceedings of the Internatinal Conference on Computer Engineering & Systems**, Cairo, pp. 268-273, 2008.

Hassoun, M. H. **Fundamentals of Artificial Neural Networks**. Cambridge: The MIT Press. 1995.

Haykin, Simon. **Redes Neurais: Princípios e Prática**. Porto Alegre: Bookman, 2ª edição, 2001, 900 p.

Hopfield, J.J. Neural Networks and physical systems with emergent collective properties. **Proceedings of the Nat.Acad.Sci**., 79: p. 2554-8, 1982.

Jain, A. K.; Griess, F. D.; Connell, S. D. **On-line signature verification**. Pattern Recognition (In Press, Uncorrected Proof). Elsevier Pres,. Jan. 2002.

Júnior, Sérgio Renato Rogal**. Detecção e classificação de arritmias cardíacas utilizando redes neurais artificiais auto-organizávies.** Dissertação de Mestrado. PUC/PR/Informática, 2008.

Lopes, Maurício Capobianco; Wilhelm, Pedro Paulo Hugo. **Aplicação de Redes Neurais Artificiais para Previsão de Vendas do Sistema de Informação do Jogo de Empresas Virtual.** Disponível em: http://members.lycos.co.uk/Dablium/artigo07.htm. Consultado em 15/10/2009.

Ludwig Jr., O.; Costa, Eduard Montgomery M.. **Redes Neurais: Fundamentos e Aplicações com Programas em C**. Rio de Janeiro: Editora Ciência Moderna LTDA, 2007.

Martins, Leila Weitzel; Assis, Joaquim Teixeira; Monat, André Soares **Aplicação de redes neurais para o diagnóstico diferencial da doença meningocócica.** Universidade Federal do Rio de Janeiro (UFRJ) e Universidade Estadual do Rio de Janeiro (UERJ) – Brasil.

McCULLOCH, W. H.; PITTS, W. S. **A logical calculus immanent in nervous activity. Bulletin of Mathematical Biophysics**, pages 115-133, 1943.

Mendes, Daniele Quintela; Oliveira, Marcio Ferreira da Silva. **Aplicações em Bioinformática.** Disponível em: http://www.lncc.br/~labinfo/tutorialRN/. Consultado em 15/10/2009.

Mendes, Daniele Quintela; Oliveira, Marcio Ferreira da Silva. **O Algoritmo "Backpropagation".** Disponível em: http://www.lncc.br/~labinfo/tutorialRN/frm4_backpropagation.htm. Consultado em 15/10/2009.

Milton Roberto, Heinen; Fernando Santos, Osório. **Autenticação de assinaturas utilizando algoritmos de Aprendizado de Máquina.** XXV Congresso da Sociedade Brasileira de Computação. UNISINOS, São Leopoldo – RS – Brazil. 22 -29 de Julho de 2005.

Minsky, M; Papert, S..Perceptrons: **A introduction to computational geometry**. MIT Press, Massachusetts, 1969.

Moerland, Perry; Fiesler, Emile. **Neural network adaptations to hardware implementations.** Technical Report RR 97-17, Dalle Molle Institute for Perceptive Artificial Intelligence, Martigny, Valais, Switzerland, January 1997.

Moerland, Perry; Fiesler, Emile. Hardware-friendly learning algorithms for neural networks: An overview**. In Proceedings of MicroNeuro '96, pages 117 - 124, 1996.**

Molz, Rolf F; Engel, Paulo M.; Moraes, Fernando G. Uso de um Ambiente Codesign para a Implementação de Redes Neurais. **In Proceedings of the IV Brazilian Conference on Neural Networks - IV Congresso Brasileiro de Redes Neurais pp. 013-018, July 20-22, 1999.**

Nacer, Carlos Henrique; Baratto, Giovani. **Rede Neural Artificial em Hardware Reconfigurável.** Universidade Federal de Santa Maria – UFSM.

Omondi, Amos R.; Rajapakse, Jagath C. **FPGA Implementations of Neural Networks.** 1ª edição. Springer, 380 p. 2006.

Osório, Fernando Santos. **Um Estudo sobre Reconhecimento Visual de Caracteres através de Redes Neurais**. Dissertação de Mestrado, CPGCC, UFRGS, Porto Alegre - Brasil. Outubro 1991.

Rosenblatt, Frank. The perceptron: a probabilistic model for information storage and organization in the brain. **Physcol Rev.**, 65: 386-408, 1958.

Rumelhart, D.E; Hinton, G. E.; Williams, R. J. **Learning internal representations by error propagation**. In D. E. Rumelhart and J. L. McClelland, editors, Parallel Distributed Processing, volume 1. MIT Press, Cambridge, MA, 1986.

Santos, Crisluci; Souza, Karina. **Classificação de distúrbios na rede elétrica usando redes neurais e wavelets.** Tese de doutorado. Universidade Federal do Rio Grande do Norte (UFRN) /ENGENHARIA ELÉTRICA, 2008.

Savran, A.; Ünsal, S.: **Hardware Implementation of a Feed forward Neural Network Using FPGAs**, Lecture Notes in Computer Science, p. 1105-1112 2003, Bursa, Turkey.

Soares, André Muniz. **Implementação digital de redes neurais artificiais para o controle de motor de indução. Tese de Mestrado em engenharia elétrica. Pós-graduação CAPES, 2006.**

Valença, Mêuser. **Aplicando Redes Neurais: Um Guia Completo.** Olinda,PE: Ed. do Autor, 2005.

WEBER, LEO; KLEIN, PEDRO ANTONIO TRIERWEILER. **Aplicação da Lógica Fuzzy em Software e Hardware**, 2003.

Widrow, B.; Hoff, M.E. Adaptative switching circuits. Institute of Radio Engineers, **Proceedings of the Western Eletronic Show and Convention**, 1960.

Widrow, Bernard; Rumelhart, David E.; Lehr, Michael A. Neural networks: applications in industry, business and science. Journal of Communications of the ACM. P. **93 – 105. 1994.**

WILHELM, Pedro Paulo Hugo; LOPES, Maurício Capobianco, et al. **Sistema inteligente de apoio à decisão**. *Revista De Negócios*. Blumenau, v.1, n. 1, dez. 1995

ANEXO Algoritmo de RNA em C++

Arquivo NeuralNet.cpp

```
#include "BackProp.h"
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include <time.h>

int main()
{

clock_t inicio, fim;

volatile int botao; //armazena o valor do botão
   // dados de treinamento XOR
   //Tipo: A XOR B XOR C XOR D
   /*double data[][5]={
            0,0,0,0,0,
            0,0,0,1,1,
            0,0,1,0,1,
            0,0,1,1,0,
            0,1,0,0,1,
```

```
            0,1,0,1,0,
            0,1,1,0,0,
            0,1,1,1,1,
            1,0,0,0,1,
            1,0,0,1,0,
            1,0,1,0,0,
            1,0,1,1,1,
            1,1,0,0,0,
            1,1,0,1,1,
            1,1,1,0,1,
            1,1,1,1,0
            };*/

double data[16][5];
// prepare test data
double testData[1][4];

int numLayers = 3, lSz[3] = {4,16,1};

double beta = 0.1, alpha = 0.65, Thresh =  0.00001;

long volatile num_iter = 100000;
//Ler arquivo
FILE*   fp_ascii =  NULL;
//char buffer[16][6];

   fp_ascii  = fopen ("/mnt/host/dados.txt", "r");
   if (fp_ascii == NULL)
   {
      printf ("Cannot open file hostfs_read_ascii.txt.\n");
      exit (1);
   }

double k = 0;

int n = 0;
int v = 0;

while(true){
   char* b;
   fgets(b, sizeof(b), fp_ascii);
```

```
        double dado = atof(b);

        //indicador de fim de arquivo
        if (dado==2)
           break;
        //colocando a linha do arquivo dentro de data, ja transformando o
dado em double
        data[n][v] = dado;
        printf("%f", data[n][v]);

        v = v + 1;

        if(v==5){
           v = 0;
           n = n + 1;
        }

        k = k + 1;
        cout<<endl;
        //printf("valor de k: %f", k);

     }

     fclose (fp_ascii);

     CBackProp *bp = new CBackProp(numLayers, lSz, beta, alpha);

     cout<< endl <<  "treinando a rede...." << endl;
     long i;
     inicio = clock();

     for (i=0; i<num_iter ; i++)
     {

        bp->bpgt(data[i%16], &data[i%16][4]);

        if( bp->mse(&data[i%16][4]) < Thresh) {
           cout << endl << "Rede treinada. Valores alcancados em " << i
<< " iteracoes." << endl;
           cout << "MSE:  " << bp->mse(&data[i%16][4])
                <<  endl <<  endl;
           break;
```

```
        }
        if ( i%(num_iter/10) == 0 ){
            cout<<  endl <<  "MSE:  " << bp->mse(&data[i%16][4])
                << "... Treinando..." << endl;
            fim= clock();
            //double difTempo1 = (double)((fim - inicio)/);
            cout<< "Intervalo 1: "<< bp->difTempo<<" "<<endl;
            inicio = clock();
        }

    }

    if ( i == num_iter ){

        cout << endl << i << " iteracoes alcancadas..."
        << "MSE: " << bp->mse(&data[(i-1)%16][4]) << endl;

    }

    //cout<< "Intervalo 2: "<< bp->difTempo<<" segundos"<<endl;

    double saida = 0;
    int saidaBin = 0;
    int num1 = 0;

    while(true){

        //ler o botao
        botao =
IORD_ALTERA_AVALON_PIO_DATA(BUTTON_PIO_BASE);

        //delay bem pequeno
        //usleep(100000);

        //se os botões 1 e 2 foram apertados juntos então a entrada
        //do neurônio é {1 1 0 0}
    if(botao == 12){
      testData[0][0] = 1;
      testData[0][1] = 1;
      testData[0][2] = 0;
      testData[0][3] = 0;
    }
```

```
//se só o primeiro botão for apertado entao a entrada é {1,0,0,0}
else if (botao == 14){
  testData[0][0] = 1;
  testData[0][1] = 0;
  testData[0][2] = 0;
  testData[0][3] = 0;
}

//se só o segundo botão for apertado entao a entrada é {0,1,0,0}
else if (botao == 13){
  testData[0][0] = 0;
  testData[0][1] = 1;
  testData[0][2] = 0;
  testData[0][3] = 0;
}

//se nenhum botão for apertado entao a entrada é {0,0,0,0}
else if(botao == 15){
  testData[0][0] = 0;
  testData[0][1] = 0;
  testData[0][2] = 0;
  testData[0][3] = 0;
}
//se botão 3 for apertado entao a entrada é {0,0,1,0}
else if(botao == 11){
  testData[0][0] = 0;
  testData[0][1] = 0;
  testData[0][2] = 1;
  testData[0][3] = 0;
}
//se botão 1 e 3 for apertado entao a entrada é {1,0,1,0}
else if(botao == 10){
  testData[0][0] = 1;
  testData[0][1] = 0;
  testData[0][2] = 1;
  testData[0][3] = 0;
}
//se botão 2 e 3 for apertado entao a entrada é {0,1,1,0}
else if(botao == 9){
  testData[0][0] = 0;
  testData[0][1] = 1;
```

```
  testData[0][2] = 1;
  testData[0][3] = 0;
}//se botão 1,2 e 3 for apertado entao a entrada é {1,1,1,0}
else if(botao==8){
  testData[0][0] = 1;
  testData[0][1] = 1;
  testData[0][2] = 1;
  testData[0][3] = 0;
}//se botão 4 for apertado entao a entrada é {0,0,0,1}
else if(botao==7){
  testData[0][0] = 0;
  testData[0][1] = 0;
  testData[0][2] = 0;
  testData[0][3] = 1;
}//se botão 4 e 1for apertado entao a entrada é {1,0,0,1}
else if(botao==6){
  testData[0][0] = 1;
  testData[0][1] = 0;
  testData[0][2] = 0;
  testData[0][3] = 1;
}
//se botão 4 e 2 for apertado entao a entrada é {0,1,0,1}
else if(botao==5){
  testData[0][0] = 0;
  testData[0][1] = 1;
  testData[0][2] = 0;
  testData[0][3] = 1;
}//se botão 4,2 e 1 for apertado entao a entrada é {1,1,0,1}
else if(botao==4){
  testData[0][0] = 1;
  testData[0][1] = 1;
  testData[0][2] = 0;
  testData[0][3] = 1;
}//se botão 4 e 3 for apertado entao a entrada é {0,0,1,1}
else if(botao==3){
  testData[0][0] = 0;
  testData[0][1] = 0;
  testData[0][2] = 1;
  testData[0][3] = 1;
}
//se botão 4,3 e 1 for apertado entao a entrada é {1,0,1,1}
else if(botao==2){
```

```
    testData[0][0] = 1;
    testData[0][1] = 0;
    testData[0][2] = 1;
    testData[0][3] = 1;
  }
  //se botão 4,3 e 2 for apertado entao a entrada é {0,1,1,1}
  else if(botao==1){
    testData[0][0] = 0;
    testData[0][1] = 1;
    testData[0][2] = 1;
    testData[0][3] = 1;
  }//se todos os botoes forem apertado entao a entrada é {1,1,1,1}
  else{
    testData[0][0] = 1;
    testData[0][1] = 1;
    testData[0][2] = 1;
    testData[0][3] = 1;
  }

  for ( int i = 0 ; i < 1 ; i++ )
  {
     bp->ffwd(testData[i]);
     saida = bp->Out(0);

     //Menos que 0.5 é considerado próximo de zero
     if (saida <=0.50)
       saidaBin = 0;
     else {
         saidaBin = 1;
         num1 = num1 + 1;
     }
     //se a saída é 1 entao acende led senão apaga
   if(saidaBin == 1)

IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE,0x1);
    else
      IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE,2
& 0x1);
  }
     cout<< "  " << bp->Out(0)<<" Em binario: " << saidaBin <<endl;

  }
```

```
   system("PAUSE");
   return 0;

}
```

Arquivo BackProp.cpp

```
#include "backprop.h"
#include <time.h>
#include <stdlib.h>


//  construtor
CBackProp::CBackProp(int nl,int *sz,double b,double
a):beta(b),alpha(a)
{

   //  set no of layers and their sizes
   numl=nl;
   lsize=new int[numl];

   for(int i=0;i<numl;i++){
      lsize[i]=sz[i];
   }

   //  allocate memory for output of each neuron
   out = new double*[numl];

   for(int i=0;i<numl;i++){
      out[i]=new double[lsize[i]];
   }

   //  allocate memory for delta
   delta = new double*[numl];

   for(int i=1;i<numl;i++){
      delta[i]=new double[lsize[i]];
   }

   //  allocate memory for weights
   weight = new double**[numl];
```

```
   for(int i=1;i<numl;i++){
      weight[i]=new double*[lsize[i]];
   }
   for(int i=1;i<numl;i++){
      for(int j=0;j<lsize[i];j++){
         weight[i][j]=new double[lsize[i-1]+1];

      }
   }

   //  allocate memory for previous weights
   prevDwt = new double**[numl];

   for(int i=1;i<numl;i++){
      prevDwt[i]=new double*[lsize[i]];

   }
   for(int i=1;i<numl;i++){
      for(int j=0;j<lsize[i];j++){
         prevDwt[i][j]=new double[lsize[i-1]+1];
      }
   }

   //  seed and assign random weights
   srand((unsigned)(time(NULL)));
   for(int i=1;i<numl;i++)
      for(int j=0;j<lsize[i];j++)
         for(int k=0;k<lsize[i-1]+1;k++)
            weight[i][j][k]=(double)(rand())/(RAND_MAX/2) -
1;//32767

   //  initialize previous weights to 0 for first iteration
   for(int i=1;i<numl;i++)
      for(int j=0;j<lsize[i];j++)
         for(int k=0;k<lsize[i-1]+1;k++)
            prevDwt[i][j][k]=(double)0.0;

// Note that the following variables are unused,
//
// delta[0]
// weight[0]
```

```
// prevDwt[0]

//  I did this intentionaly to maintains consistancy in numbering the
layers.
//  Since for a net having n layers, input layer is refered to as 0th layer,
//  first hidden layer as 1st layer and the nth layer as output layer. And
//  first (0th) layer just stores the inputs hence there is no delta or
weigth
//  values corresponding to it.
}



CBackProp::~CBackProp()
{
   //  free out
   for(int i=0;i<numl;i++)
      delete[] out[i];
   delete[] out;

   //  free delta
   for(int i=1;i<numl;i++)
      delete[] delta[i];
   delete[] delta;

   //  free weight
   for(int i=1;i<numl;i++)
      for(int j=0;j<lsize[i];j++)
         delete[] weight[i][j];
   for(int i=1;i<numl;i++)
      delete[] weight[i];
   delete[] weight;

   //  free prevDwt
   for(int i=1;i<numl;i++)
      for(int j=0;j<lsize[i];j++)
         delete[] prevDwt[i][j];
   for(int i=1;i<numl;i++)
      delete[] prevDwt[i];
   delete[] prevDwt;

   //  free layer info
```

```
   delete[] lsize;
}

//  sigmoid function
double CBackProp::sigmoid(double in)
{
       return (double)(1/(1+exp(-in)));
}

//  mean square error
double CBackProp::mse(double *tgt) const
{
   double mse=0;
   for(int i=0;i<lsize[numl-1];i++){
      mse+=(tgt[i]-out[numl-1][i])*(tgt[i]-out[numl-1][i]);
   }
   return mse/2;
}


//  returns i'th output of the net
double CBackProp::Out(int i) const
{

   return out[numl-1][i];
}

// feed forward one set of input
void CBackProp::ffwd(double *in)
{
   double sum;

   //  assign content to input layer
   for(int i=0;i<lsize[0];i++)
      out[0][i]=in[i];  // output_from_neuron(i,j) Jth neuron in Ith
Layer

   //  assign output(activation) value
   //  to each neuron usng sigmoid func
   for(int i=1;i<numl;i++){              // For each layer
      for(int j=0;j<lsize[i];j++){       // For each neuron in current layer
         sum=0.0;
```

```
            for(int k=0;k<lsize[i-1];k++){      // For input from each neuron
in preceeding layer
                sum+= out[i-1][k]*weight[i][j][k];  // Apply weight to inputs
and add to sum
            }
            sum+=weight[i][j][lsize[i-1]];      // Apply bias
            out[i][j]=sigmoid(sum);             // Apply sigmoid function
        }
    }
}

//  backpropogate errors from output
//  layer uptill the first hidden layer
void CBackProp::bpgt(double *in,double *tgt)
{
    double sum;
    inicio = clock();
    //  update output values for each neuron
    ffwd(in);

    //  find delta for output layer
    for(int i=0;i<lsize[numl-1];i++){
        delta[numl-1][i]=out[numl-1][i]*
        (1-out[numl-1][i])*(tgt[i]-out[numl-1][i]);
    }

    //  find delta for hidden layers
    for(int i=numl-2;i>0;i--){
        for(int j=0;j<lsize[i];j++){
            sum=0.0;
            for(int k=0;k<lsize[i+1];k++){
                sum+=delta[i+1][k]*weight[i+1][k][j];
            }
            delta[i][j]=out[i][j]*(1-out[i][j])*sum;
        }
    }

    //  apply momentum ( does nothing if alpha=0 )
    for(int i=1;i<numl;i++){
        for(int j=0;j<lsize[i];j++){
            for(int k=0;k<lsize[i-1];k++){
                weight[i][j][k]+=alpha*prevDwt[i][j][k];
```

```
            }
            weight[i][j][lsize[i-1]]+=alpha*prevDwt[i][j][lsize[i-1]];
        }
    }

    //  adjust weights usng steepest descent
    for(int i=1;i<numl;i++){
        for(int j=0;j<lsize[i];j++){
            for(int k=0;k<lsize[i-1];k++){
                prevDwt[i][j][k]=beta*delta[i][j]*out[i-1][k];
                weight[i][j][k]+=prevDwt[i][j][k];
            }
            prevDwt[i][j][lsize[i-1]]=beta*delta[i][j];
            weight[i][j][lsize[i-1]]+=prevDwt[i][j][lsize[i-1]];
        }
    }

    fim = clock();
    difTempo = ( (fim - inicio));
}
```

Arquivo BackProp.h

```
////////////////////////////////////////////
//  Fully connected multilayered feed      //
//  forward artificial neural network using //
//  Backpropogation algorithm for training. //
////////////////////////////////////////////


#ifndef backprop_h
#define backprop_h

#include<assert.h>
#include<iostream.h>
#include<stdio.h>
#include<math.h>


class CBackProp{
```

```
// output of each neuron
   double **out;

// delta error value for each neuron
   double **delta;

// vector of weights for each neuron
   double ***weight;

// no of layers in net
// including input layer
   int numl;

// vector of numl elements for size
// of each layer
   int *lsize;

// learning rate
   double beta;

// momentum parameter
   double alpha;

// storage for weight-change made
// in previous epoch
   double ***prevDwt;

// squashing function
   double sigmoid(double in);

//para calcular tempo em segundos
clock_t inicio, fim;

//a diferenca de tempo
public: double difTempo;

public:

   ~CBackProp();

// initializes and allocates memory
   CBackProp(int nl,int *sz,double b,double a);
```

```
// backpropogates error for one set of input
   void bpgt(double *in,double *tgt);

// feed forwards activations for one set of inputs
   void ffwd(double *in);

// returns mean square error of the net
   double mse(double *tgt) const;

// returns i'th output of the net
   double Out(int i) const;
}; #endif
```

APÊNDICE A Função simoidal em VHDL

```
-- Copyright (C) 1991-2007 Altera Corporation
```

```
-- Your use of Altera Corporation's design tools, logic
functions
-- and other software and tools, and its AMPP partner
logic
-- functions, and any output files from any of the
foregoing
-- (including device programming or simulation files),
and any
-- associated documentation or information are expressly
subject
-- to the terms and conditions of the Altera Program
License
-- Subscription Agreement, Altera MegaCore Function
License
-- Agreement, or other applicable license agreement,
including,
-- without limitation, that your use is for the sole
purpose of
-- programming logic devices manufactured by Altera and
sold by
-- Altera or its authorized distributors.  Please refer
to the
-- applicable agreement for further details.

-- PROGRAM "Quartus II"
-- VERSION "Version 7.2 Build 151 09/26/2007 SJ Full
Version"

LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

ENTITY funcaoteste IS
      port
      (

            clk :  IN  STD_LOGIC;
            signal reset : IN STD_LOGIC; -- CPU's master
asynchronous reset <required for multi-cycle>
            signal clk_en: IN STD_LOGIC;  -- Clock-
qualifier <required for multi-cycle>
            signal start: IN STD_LOGIC;   -- True when
this instr. issues <required for multi-cycle>
            a :  IN  STD_LOGIC_VECTOR(31 downto 0);
            saida :  OUT  STD_LOGIC_VECTOR(31 downto 0)
      );
END funcaoteste;

ARCHITECTURE bdf_type OF funcaoteste IS
```

```
component deltau
      PORT(clk : IN STD_LOGIC;
             u0 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u1 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u10 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u11 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u12 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u13 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u14 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u15 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u16 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u17 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u18 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u19 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u2 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u20 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u21 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u22 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u23 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u24 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u25 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u26 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u27 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u28 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u29 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u3 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u30 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u31 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u32 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u33 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u34 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u35 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u36 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u37 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u38 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u39 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u4 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u40 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u41 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u42 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u43 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u44 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u45 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u46 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u47 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u48 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u49 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u5 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u50 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u51 : OUT STD_LOGIC_VECTOR(31 downto 0);
             u52 : OUT STD_LOGIC_VECTOR(31 downto 0);
```

```
            u53 : OUT STD_LOGIC_VECTOR(31 downto 0);
            u54 : OUT STD_LOGIC_VECTOR(31 downto 0);
            u55 : OUT STD_LOGIC_VECTOR(31 downto 0);
            u56 : OUT STD_LOGIC_VECTOR(31 downto 0);
            u57 : OUT STD_LOGIC_VECTOR(31 downto 0);
            u58 : OUT STD_LOGIC_VECTOR(31 downto 0);
            u6 : OUT STD_LOGIC_VECTOR(31 downto 0);
            u7 : OUT STD_LOGIC_VECTOR(31 downto 0);
            u8 : OUT STD_LOGIC_VECTOR(31 downto 0);
            u9 : OUT STD_LOGIC_VECTOR(31 downto 0)
      );
end component;

component valory
      PORT(ent1 : IN STD_LOGIC;
            ent2 : IN STD_LOGIC;
            ent3 : IN STD_LOGIC;
            ent4 : IN STD_LOGIC;
            ent5 : IN STD_LOGIC;
            ent6 : IN STD_LOGIC;
            ent7 : IN STD_LOGIC;
            ent8 : IN STD_LOGIC;
            ent9 : IN STD_LOGIC;
            ent10 : IN STD_LOGIC;
            ent11 : IN STD_LOGIC;
            ent12 : IN STD_LOGIC;
            ent13 : IN STD_LOGIC;
            ent14 : IN STD_LOGIC;
            ent15 : IN STD_LOGIC;
            ent16 : IN STD_LOGIC;
            clk : IN STD_LOGIC;
            ent17 : IN STD_LOGIC;
            ent18 : IN STD_LOGIC;
            ent19 : IN STD_LOGIC;
            ent20 : IN STD_LOGIC;
            ent21 : IN STD_LOGIC;
            ent22 : IN STD_LOGIC;
            ent23 : IN STD_LOGIC;
            ent24 : IN STD_LOGIC;
            ent25 : IN STD_LOGIC;
            ent26 : IN STD_LOGIC;
            ent27 : IN STD_LOGIC;
            ent28 : IN STD_LOGIC;
            ent29 : IN STD_LOGIC;
            ent30 : IN STD_LOGIC;
            ent31 : IN STD_LOGIC;
            ent32 : IN STD_LOGIC;
            ent33 : IN STD_LOGIC;
            ent34 : IN STD_LOGIC;
            ent35 : IN STD_LOGIC;
            ent36 : IN STD_LOGIC;
```

```
            ent37 : IN STD_LOGIC;
            ent38 : IN STD_LOGIC;
            ent39 : IN STD_LOGIC;
            ent40 : IN STD_LOGIC;
            ent41 : IN STD_LOGIC;
            ent42 : IN STD_LOGIC;
            ent59 : IN STD_LOGIC;
            ent58 : IN STD_LOGIC;
            ent57 : IN STD_LOGIC;
            ent56 : IN STD_LOGIC;
            ent55 : IN STD_LOGIC;
            ent54 : IN STD_LOGIC;
            ent53 : IN STD_LOGIC;
            ent52 : IN STD_LOGIC;
            ent51 : IN STD_LOGIC;
            ent50 : IN STD_LOGIC;
            ent49 : IN STD_LOGIC;
            ent48 : IN STD_LOGIC;
            ent47 : IN STD_LOGIC;
            ent45 : IN STD_LOGIC;
            ent46 : IN STD_LOGIC;
            ent44 : IN STD_LOGIC;
            ent43 : IN STD_LOGIC;
            ent60 : IN STD_LOGIC;
            saida : OUT STD_LOGIC_VECTOR(31 downto 0)
      );
end component;

component altfp_compare94
      PORT(clock : IN STD_LOGIC;
            dataa : IN STD_LOGIC_VECTOR(31 downto 0);
            datab : IN STD_LOGIC_VECTOR(31 downto 0);
            alb : OUT STD_LOGIC
      );
end component;

component altfp_compare96
      PORT(clock : IN STD_LOGIC;
            dataa : IN STD_LOGIC_VECTOR(31 downto 0);
            datab : IN STD_LOGIC_VECTOR(31 downto 0);
            alb : OUT STD_LOGIC
      );
end component;

component altfp_compare98
      PORT(clock : IN STD_LOGIC;
            dataa : IN STD_LOGIC_VECTOR(31 downto 0);
            datab : IN STD_LOGIC_VECTOR(31 downto 0);
            alb : OUT STD_LOGIC
      );
end component;
```

```
component altfp_compare100
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
		datab : IN STD_LOGIC_VECTOR(31 downto 0);
		alb : OUT STD_LOGIC
	);
end component;

component altfp_compare102
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
		datab : IN STD_LOGIC_VECTOR(31 downto 0);
		alb : OUT STD_LOGIC
	);
end component;

component altfp_compare8
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
		datab : IN STD_LOGIC_VECTOR(31 downto 0);
		alb : OUT STD_LOGIC
	);
end component;

component altfp_compare104
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
		datab : IN STD_LOGIC_VECTOR(31 downto 0);
		alb : OUT STD_LOGIC
	);
end component;

component altfp_compare106
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
		datab : IN STD_LOGIC_VECTOR(31 downto 0);
		alb : OUT STD_LOGIC
	);
end component;

component altfp_compare108
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
		datab : IN STD_LOGIC_VECTOR(31 downto 0);
		alb : OUT STD_LOGIC
	);
end component;

component altfp_compare110
	PORT(clock : IN STD_LOGIC;
```

```
            dataa : IN STD_LOGIC_VECTOR(31 downto 0);
            datab : IN STD_LOGIC_VECTOR(31 downto 0);
            alb : OUT STD_LOGIC
      );
end component;

component altfp_compare112
      PORT(clock : IN STD_LOGIC;
            dataa : IN STD_LOGIC_VECTOR(31 downto 0);
            datab : IN STD_LOGIC_VECTOR(31 downto 0);
            alb : OUT STD_LOGIC
      );
end component;

component altfp_compare114
      PORT(clock : IN STD_LOGIC;
            dataa : IN STD_LOGIC_VECTOR(31 downto 0);
            datab : IN STD_LOGIC_VECTOR(31 downto 0);
            alb : OUT STD_LOGIC
      );
end component;

component altfp_compare116
      PORT(clock : IN STD_LOGIC;
            dataa : IN STD_LOGIC_VECTOR(31 downto 0);
            datab : IN STD_LOGIC_VECTOR(31 downto 0);
            alb : OUT STD_LOGIC
      );
end component;

component altfp_compare1
      PORT(clock : IN STD_LOGIC;
            dataa : IN STD_LOGIC_VECTOR(31 downto 0);
            datab : IN STD_LOGIC_VECTOR(31 downto 0);
            alb : OUT STD_LOGIC
      );
end component;

component altfp_compare10
      PORT(clock : IN STD_LOGIC;
            dataa : IN STD_LOGIC_VECTOR(31 downto 0);
            datab : IN STD_LOGIC_VECTOR(31 downto 0);
            alb : OUT STD_LOGIC
      );
end component;

component altfp_compare12
      PORT(clock : IN STD_LOGIC;
            dataa : IN STD_LOGIC_VECTOR(31 downto 0);
            datab : IN STD_LOGIC_VECTOR(31 downto 0);
            alb : OUT STD_LOGIC
```

```
	);
end component;

component altfp_compare14
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
		datab : IN STD_LOGIC_VECTOR(31 downto 0);
		alb : OUT STD_LOGIC
	);
end component;

component altfp_compare16
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
		datab : IN STD_LOGIC_VECTOR(31 downto 0);
		alb : OUT STD_LOGIC
	);
end component;

component altfp_compare18
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
		datab : IN STD_LOGIC_VECTOR(31 downto 0);
		alb : OUT STD_LOGIC
	);
end component;

component altfp_compare20
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
		datab : IN STD_LOGIC_VECTOR(31 downto 0);
		alb : OUT STD_LOGIC
	);
end component;

component altfp_compare22
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
		datab : IN STD_LOGIC_VECTOR(31 downto 0);
		alb : OUT STD_LOGIC
	);
end component;

component altfp_compare24
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
		datab : IN STD_LOGIC_VECTOR(31 downto 0);
		alb : OUT STD_LOGIC
	);
end component;
```

```
component altfp_compare26
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
		datab : IN STD_LOGIC_VECTOR(31 downto 0);
		alb : OUT STD_LOGIC
	);
end component;

component altfp_compare28
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
		datab : IN STD_LOGIC_VECTOR(31 downto 0);
		alb : OUT STD_LOGIC
	);
end component;

component altfp_compare30
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
		datab : IN STD_LOGIC_VECTOR(31 downto 0);
		alb : OUT STD_LOGIC
	);
end component;

component altfp_compare32
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
		datab : IN STD_LOGIC_VECTOR(31 downto 0);
		alb : OUT STD_LOGIC
	);
end component;

component altfp_compare34
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
		datab : IN STD_LOGIC_VECTOR(31 downto 0);
		alb : OUT STD_LOGIC
	);
end component;

component altfp_compare36
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
		datab : IN STD_LOGIC_VECTOR(31 downto 0);
		alb : OUT STD_LOGIC
	);
end component;

component altfp_compare3
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
```

```
            datab : IN STD_LOGIC_VECTOR(31 downto 0);
            alb : OUT STD_LOGIC
      );
end component;

component altfp_compare38
      PORT(clock : IN STD_LOGIC;
            dataa : IN STD_LOGIC_VECTOR(31 downto 0);
            datab : IN STD_LOGIC_VECTOR(31 downto 0);
            aleb : OUT STD_LOGIC
      );
end component;

component altfp_compare40
      PORT(clock : IN STD_LOGIC;
            dataa : IN STD_LOGIC_VECTOR(31 downto 0);
            datab : IN STD_LOGIC_VECTOR(31 downto 0);
            alb : OUT STD_LOGIC
      );
end component;

component altfp_compare42
      PORT(clock : IN STD_LOGIC;
            dataa : IN STD_LOGIC_VECTOR(31 downto 0);
            datab : IN STD_LOGIC_VECTOR(31 downto 0);
            alb : OUT STD_LOGIC
      );
end component;

component altfp_compare44
      PORT(clock : IN STD_LOGIC;
            dataa : IN STD_LOGIC_VECTOR(31 downto 0);
            datab : IN STD_LOGIC_VECTOR(31 downto 0);
            aleb : OUT STD_LOGIC
      );
end component;

component altfp_compare46
      PORT(clock : IN STD_LOGIC;
            dataa : IN STD_LOGIC_VECTOR(31 downto 0);
            datab : IN STD_LOGIC_VECTOR(31 downto 0);
            alb : OUT STD_LOGIC
      );
end component;

component altfp_compare48
      PORT(clock : IN STD_LOGIC;
            dataa : IN STD_LOGIC_VECTOR(31 downto 0);
            datab : IN STD_LOGIC_VECTOR(31 downto 0);
            alb : OUT STD_LOGIC
      );
```

```
end component;

component altfp_compare50
      PORT(clock : IN STD_LOGIC;
             dataa : IN STD_LOGIC_VECTOR(31 downto 0);
             datab : IN STD_LOGIC_VECTOR(31 downto 0);
             alb : OUT STD_LOGIC
      );
end component;

component altfp_compare52
      PORT(clock : IN STD_LOGIC;
             dataa : IN STD_LOGIC_VECTOR(31 downto 0);
             datab : IN STD_LOGIC_VECTOR(31 downto 0);
             alb : OUT STD_LOGIC
      );
end component;

component altfp_compare54
      PORT(clock : IN STD_LOGIC;
             dataa : IN STD_LOGIC_VECTOR(31 downto 0);
             datab : IN STD_LOGIC_VECTOR(31 downto 0);
             alb : OUT STD_LOGIC
      );
end component;

component altfp_compare56
      PORT(clock : IN STD_LOGIC;
             dataa : IN STD_LOGIC_VECTOR(31 downto 0);
             datab : IN STD_LOGIC_VECTOR(31 downto 0);
             alb : OUT STD_LOGIC
      );
end component;

component altfp_compare4
      PORT(clock : IN STD_LOGIC;
             dataa : IN STD_LOGIC_VECTOR(31 downto 0);
             datab : IN STD_LOGIC_VECTOR(31 downto 0);
             alb : OUT STD_LOGIC
      );
end component;

component altfp_compare58
      PORT(clock : IN STD_LOGIC;
             dataa : IN STD_LOGIC_VECTOR(31 downto 0);
             datab : IN STD_LOGIC_VECTOR(31 downto 0);
             alb : OUT STD_LOGIC
      );
end component;

component altfp_compare60
```

```
      PORT(clock : IN STD_LOGIC;
             dataa : IN STD_LOGIC_VECTOR(31 downto 0);
             datab : IN STD_LOGIC_VECTOR(31 downto 0);
             alb : OUT STD_LOGIC
      );
end component;

component altfp_compare62
      PORT(clock : IN STD_LOGIC;
             dataa : IN STD_LOGIC_VECTOR(31 downto 0);
             datab : IN STD_LOGIC_VECTOR(31 downto 0);
             aleb : OUT STD_LOGIC
      );
end component;

component altfp_compare64
      PORT(clock : IN STD_LOGIC;
             dataa : IN STD_LOGIC_VECTOR(31 downto 0);
             datab : IN STD_LOGIC_VECTOR(31 downto 0);
             alb : OUT STD_LOGIC
      );
end component;

component altfp_compare66
      PORT(clock : IN STD_LOGIC;
             dataa : IN STD_LOGIC_VECTOR(31 downto 0);
             datab : IN STD_LOGIC_VECTOR(31 downto 0);
             alb : OUT STD_LOGIC
      );
end component;

component altfp_compare70
      PORT(clock : IN STD_LOGIC;
             dataa : IN STD_LOGIC_VECTOR(31 downto 0);
             datab : IN STD_LOGIC_VECTOR(31 downto 0);
             alb : OUT STD_LOGIC
      );
end component;

component altfp_compare72
      PORT(clock : IN STD_LOGIC;
             dataa : IN STD_LOGIC_VECTOR(31 downto 0);
             datab : IN STD_LOGIC_VECTOR(31 downto 0);
             alb : OUT STD_LOGIC
      );
end component;

component altfp_compare68
      PORT(clock : IN STD_LOGIC;
             dataa : IN STD_LOGIC_VECTOR(31 downto 0);
             datab : IN STD_LOGIC_VECTOR(31 downto 0);
```

```
            alb : OUT STD_LOGIC
      );
end component;

component altfp_compare74
      PORT(clock : IN STD_LOGIC;
            dataa : IN STD_LOGIC_VECTOR(31 downto 0);
            datab : IN STD_LOGIC_VECTOR(31 downto 0);
            alb : OUT STD_LOGIC
      );
end component;

component altfp_compare76
      PORT(clock : IN STD_LOGIC;
            dataa : IN STD_LOGIC_VECTOR(31 downto 0);
            datab : IN STD_LOGIC_VECTOR(31 downto 0);
            alb : OUT STD_LOGIC
      );
end component;

component altfp_compare78
      PORT(clock : IN STD_LOGIC;
            dataa : IN STD_LOGIC_VECTOR(31 downto 0);
            datab : IN STD_LOGIC_VECTOR(31 downto 0);
            alb : OUT STD_LOGIC
      );
end component;

component altfp_compare80
      PORT(clock : IN STD_LOGIC;
            dataa : IN STD_LOGIC_VECTOR(31 downto 0);
            datab : IN STD_LOGIC_VECTOR(31 downto 0);
            alb : OUT STD_LOGIC
      );
end component;

component altfp_compare82
      PORT(clock : IN STD_LOGIC;
            dataa : IN STD_LOGIC_VECTOR(31 downto 0);
            datab : IN STD_LOGIC_VECTOR(31 downto 0);
            alb : OUT STD_LOGIC
      );
end component;

component altfp_compare6
      PORT(clock : IN STD_LOGIC;
            dataa : IN STD_LOGIC_VECTOR(31 downto 0);
            datab : IN STD_LOGIC_VECTOR(31 downto 0);
            alb : OUT STD_LOGIC
      );
end component;
```

```
component altfp_compare118
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
		datab : IN STD_LOGIC_VECTOR(31 downto 0);
		alb : OUT STD_LOGIC
	);
end component;

component altfp_compare86
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
		datab : IN STD_LOGIC_VECTOR(31 downto 0);
		alb : OUT STD_LOGIC
	);
end component;

component altfp_compare88
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
		datab : IN STD_LOGIC_VECTOR(31 downto 0);
		alb : OUT STD_LOGIC
	);
end component;

component altfp_compare90
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
		datab : IN STD_LOGIC_VECTOR(31 downto 0);
		alb : OUT STD_LOGIC
	);
end component;

component altfp_compare92
	PORT(clock : IN STD_LOGIC;
		dataa : IN STD_LOGIC_VECTOR(31 downto 0);
		datab : IN STD_LOGIC_VECTOR(31 downto 0);
		alb : OUT STD_LOGIC
	);
end component;

signal	SYNTHESIZED_WIRE_294 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_1 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_2 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_3 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_4 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_5 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_6 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_7 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_8 :  STD_LOGIC;
signal	SYNTHESIZED_WIRE_9 :  STD_LOGIC;
```

```
signal      SYNTHESIZED_WIRE_10 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_11 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_12 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_13 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_14 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_15 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_16 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_17 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_18 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_19 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_20 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_21 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_22 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_23 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_24 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_25 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_26 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_27 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_28 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_29 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_30 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_31 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_32 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_33 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_34 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_35 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_36 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_37 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_38 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_39 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_40 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_41 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_42 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_43 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_44 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_45 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_46 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_47 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_48 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_49 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_50 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_51 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_52 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_53 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_54 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_55 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_56 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_57 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_58 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_59 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_295 :  STD_LOGIC;
```

```
signal      SYNTHESIZED_WIRE_61 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_296 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_63 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_297 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_65 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_298 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_67 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_299 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_69 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_300 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_71 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_72 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_301 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_74 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_302 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_76 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_78 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_79 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_303 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_81 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_82 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_83 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_84 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_85 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_304 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_87 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_305 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_89 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_306 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_91 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_307 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_94 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_96 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_98 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_308 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_100 :  STD_LOGIC;
```

```
signal      SYNTHESIZED_WIRE_309 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_102 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_310 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_104 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_311 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_106 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_312 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_108 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_313 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_110 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_314 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_112 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_315 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_114 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_115 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_316 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_117 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_317 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_119 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_318 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_121 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_319 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_123 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_320 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_125 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_321 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_127 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_322 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_129 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_323 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_131 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_324 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_133 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_325 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_136 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_326 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_138 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_327 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_140 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_328 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_142 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_329 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_144 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_330 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_146 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_331 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_148 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_332 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_150 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_333 :  STD_LOGIC;
```

```
signal      SYNTHESIZED_WIRE_152 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_334 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_154 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_335 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_156 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_157 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_336 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_159 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_337 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_161 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_338 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_163 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_339 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_165 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_340 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_167 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_341 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_169 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_342 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_171 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_343 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_173 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_344 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_175 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_345 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_178 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_346 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_180 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_347 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_182 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_348 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_184 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_349 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_186 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_350 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_188 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_351 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_190 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_192 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_194 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_196 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_198 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_199 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_201 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_203 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_352 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_207 :  STD_LOGIC_VECTOR(31
downto 0);
```

```
signal      SYNTHESIZED_WIRE_209 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_211 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_213 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_215 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_218 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_220 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_222 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_224 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_226 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_227 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_229 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_231 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_233 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_235 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_237 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_240 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_242 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_244 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_246 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_248 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_249 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_251 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_253 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_255 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_257 :  STD_LOGIC_VECTOR(31
downto 0);
```

```
signal      SYNTHESIZED_WIRE_259 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_264 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_266 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_268 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_271 :  STD_LOGIC;
signal      SYNTHESIZED_WIRE_273 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_275 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_277 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_279 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_281 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_283 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_284 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_286 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_288 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_290 :  STD_LOGIC_VECTOR(31
downto 0);
signal      SYNTHESIZED_WIRE_292 :  STD_LOGIC_VECTOR(31
downto 0);


BEGIN

b2v_inst : deltau
PORT MAP(clk => clk,
             u0 => SYNTHESIZED_WIRE_83,
             u1 => SYNTHESIZED_WIRE_227,
             u10 => SYNTHESIZED_WIRE_209,
             u11 => SYNTHESIZED_WIRE_211,
             u12 => SYNTHESIZED_WIRE_213,
             u13 => SYNTHESIZED_WIRE_215,
             u14 => SYNTHESIZED_WIRE_218,
             u15 => SYNTHESIZED_WIRE_220,
             u16 => SYNTHESIZED_WIRE_222,
             u17 => SYNTHESIZED_WIRE_224,
             u18 => SYNTHESIZED_WIRE_226,
             u19 => SYNTHESIZED_WIRE_229,
             u2 => SYNTHESIZED_WIRE_249,
             u20 => SYNTHESIZED_WIRE_231,
```

```
              u21 => SYNTHESIZED_WIRE_233,
              u22 => SYNTHESIZED_WIRE_235,
              u23 => SYNTHESIZED_WIRE_237,
              u24 => SYNTHESIZED_WIRE_240,
              u25 => SYNTHESIZED_WIRE_242,
              u26 => SYNTHESIZED_WIRE_244,
              u27 => SYNTHESIZED_WIRE_246,
              u28 => SYNTHESIZED_WIRE_248,
              u29 => SYNTHESIZED_WIRE_251,
              u3 => SYNTHESIZED_WIRE_283,
              u30 => SYNTHESIZED_WIRE_253,
              u31 => SYNTHESIZED_WIRE_255,
              u32 => SYNTHESIZED_WIRE_257,
              u33 => SYNTHESIZED_WIRE_259,
              u34 => SYNTHESIZED_WIRE_268,
              u35 => SYNTHESIZED_WIRE_264,
              u36 => SYNTHESIZED_WIRE_266,
              u37 => SYNTHESIZED_WIRE_273,
              u38 => SYNTHESIZED_WIRE_275,
              u39 => SYNTHESIZED_WIRE_277,
              u4 => SYNTHESIZED_WIRE_71,
              u40 => SYNTHESIZED_WIRE_279,
              u41 => SYNTHESIZED_WIRE_281,
              u42 => SYNTHESIZED_WIRE_284,
              u43 => SYNTHESIZED_WIRE_286,
              u44 => SYNTHESIZED_WIRE_288,
              u45 => SYNTHESIZED_WIRE_290,
              u46 => SYNTHESIZED_WIRE_292,
              u47 => SYNTHESIZED_WIRE_61,
              u48 => SYNTHESIZED_WIRE_63,
              u49 => SYNTHESIZED_WIRE_65,
              u5 => SYNTHESIZED_WIRE_84,
              u50 => SYNTHESIZED_WIRE_67,
              u51 => SYNTHESIZED_WIRE_69,
              u52 => SYNTHESIZED_WIRE_72,
              u53 => SYNTHESIZED_WIRE_74,
              u54 => SYNTHESIZED_WIRE_76,
              u55 => SYNTHESIZED_WIRE_78,
              u56 => SYNTHESIZED_WIRE_79,
              u57 => SYNTHESIZED_WIRE_81,
              u58 => SYNTHESIZED_WIRE_82,
              u6 => SYNTHESIZED_WIRE_114,
              u7 => SYNTHESIZED_WIRE_156,
              u8 => SYNTHESIZED_WIRE_198,
              u9 => SYNTHESIZED_WIRE_207);

b2v_inst1 : valory
PORT MAP(ent1 => SYNTHESIZED_WIRE_294,
             ent2 => SYNTHESIZED_WIRE_1,
             ent3 => SYNTHESIZED_WIRE_2,
             ent4 => SYNTHESIZED_WIRE_3,
```

```
ent5 => SYNTHESIZED_WIRE_4,
ent6 => SYNTHESIZED_WIRE_5,
ent7 => SYNTHESIZED_WIRE_6,
ent8 => SYNTHESIZED_WIRE_7,
ent9 => SYNTHESIZED_WIRE_8,
ent10 => SYNTHESIZED_WIRE_9,
ent11 => SYNTHESIZED_WIRE_10,
ent12 => SYNTHESIZED_WIRE_11,
ent13 => SYNTHESIZED_WIRE_12,
ent14 => SYNTHESIZED_WIRE_13,
ent15 => SYNTHESIZED_WIRE_14,
ent16 => SYNTHESIZED_WIRE_15,
clk => clk,
ent17 => SYNTHESIZED_WIRE_16,
ent18 => SYNTHESIZED_WIRE_17,
ent19 => SYNTHESIZED_WIRE_18,
ent20 => SYNTHESIZED_WIRE_19,
ent21 => SYNTHESIZED_WIRE_20,
ent22 => SYNTHESIZED_WIRE_21,
ent23 => SYNTHESIZED_WIRE_22,
ent24 => SYNTHESIZED_WIRE_23,
ent25 => SYNTHESIZED_WIRE_24,
ent26 => SYNTHESIZED_WIRE_25,
ent27 => SYNTHESIZED_WIRE_26,
ent28 => SYNTHESIZED_WIRE_27,
ent29 => SYNTHESIZED_WIRE_28,
ent30 => SYNTHESIZED_WIRE_29,
ent31 => SYNTHESIZED_WIRE_30,
ent32 => SYNTHESIZED_WIRE_31,
ent33 => SYNTHESIZED_WIRE_32,
ent34 => SYNTHESIZED_WIRE_33,
ent35 => SYNTHESIZED_WIRE_34,
ent36 => SYNTHESIZED_WIRE_35,
ent37 => SYNTHESIZED_WIRE_36,
ent38 => SYNTHESIZED_WIRE_37,
ent39 => SYNTHESIZED_WIRE_38,
ent40 => SYNTHESIZED_WIRE_39,
ent41 => SYNTHESIZED_WIRE_40,
ent42 => SYNTHESIZED_WIRE_41,
ent59 => SYNTHESIZED_WIRE_42,
ent58 => SYNTHESIZED_WIRE_43,
ent57 => SYNTHESIZED_WIRE_44,
ent56 => SYNTHESIZED_WIRE_45,
ent55 => SYNTHESIZED_WIRE_46,
ent54 => SYNTHESIZED_WIRE_47,
ent53 => SYNTHESIZED_WIRE_48,
ent52 => SYNTHESIZED_WIRE_49,
ent51 => SYNTHESIZED_WIRE_50,
ent50 => SYNTHESIZED_WIRE_51,
ent49 => SYNTHESIZED_WIRE_52,
ent48 => SYNTHESIZED_WIRE_53,
```

```
            ent47 => SYNTHESIZED_WIRE_54,
            ent45 => SYNTHESIZED_WIRE_55,
            ent46 => SYNTHESIZED_WIRE_56,
            ent44 => SYNTHESIZED_WIRE_57,
            ent43 => SYNTHESIZED_WIRE_58,
            ent60 => SYNTHESIZED_WIRE_59,
            saida => saida);

SYNTHESIZED_WIRE_94 <= NOT(SYNTHESIZED_WIRE_295);


b2v_inst100 : altfp_compare94
PORT MAP(clock => clk,
            dataa => a,
            datab => SYNTHESIZED_WIRE_61,
            alb => SYNTHESIZED_WIRE_347);

SYNTHESIZED_WIRE_192 <= NOT(SYNTHESIZED_WIRE_296);


b2v_inst102 : altfp_compare96
PORT MAP(clock => clk,
            dataa => a,
            datab => SYNTHESIZED_WIRE_63,
            alb => SYNTHESIZED_WIRE_348);

SYNTHESIZED_WIRE_194 <= NOT(SYNTHESIZED_WIRE_297);


b2v_inst104 : altfp_compare98
PORT MAP(clock => clk,
            dataa => a,
            datab => SYNTHESIZED_WIRE_65,
            alb => SYNTHESIZED_WIRE_349);

SYNTHESIZED_WIRE_196 <= NOT(SYNTHESIZED_WIRE_298);


b2v_inst106 : altfp_compare100
PORT MAP(clock => clk,
            dataa => a,
            datab => SYNTHESIZED_WIRE_67,
            alb => SYNTHESIZED_WIRE_350);

SYNTHESIZED_WIRE_199 <= NOT(SYNTHESIZED_WIRE_299);


b2v_inst108 : altfp_compare102
PORT MAP(clock => clk,
            dataa => a,
            datab => SYNTHESIZED_WIRE_69,
```

```
            alb => SYNTHESIZED_WIRE_351);

SYNTHESIZED_WIRE_201 <= NOT(SYNTHESIZED_WIRE_300);


b2v_inst11 : altfp_compare8
PORT MAP(clock => clk,
            dataa => a,
            datab => SYNTHESIZED_WIRE_71,
            alb => SYNTHESIZED_WIRE_306);

b2v_inst110 : altfp_compare104
PORT MAP(clock => clk,
            dataa => a,
            datab => SYNTHESIZED_WIRE_72,
            alb => SYNTHESIZED_WIRE_296);

SYNTHESIZED_WIRE_203 <= NOT(SYNTHESIZED_WIRE_301);


b2v_inst112 : altfp_compare106
PORT MAP(clock => clk,
            dataa => a,
            datab => SYNTHESIZED_WIRE_74,
            alb => SYNTHESIZED_WIRE_297);

SYNTHESIZED_WIRE_59 <= NOT(SYNTHESIZED_WIRE_302);


b2v_inst114 : altfp_compare108
PORT MAP(clock => clk,
            dataa => a,
            datab => SYNTHESIZED_WIRE_76,
            alb => SYNTHESIZED_WIRE_298);

SYNTHESIZED_WIRE_271 <= NOT(SYNTHESIZED_WIRE_294);


b2v_inst116 : altfp_compare110
PORT MAP(clock => clk,
            dataa => a,
            datab => SYNTHESIZED_WIRE_78,
            alb => SYNTHESIZED_WIRE_299);

b2v_inst118 : altfp_compare112
PORT MAP(clock => clk,
            dataa => a,
            datab => SYNTHESIZED_WIRE_79,
            alb => SYNTHESIZED_WIRE_300);

SYNTHESIZED_WIRE_96 <= NOT(SYNTHESIZED_WIRE_303);
```

```
b2v_inst120 : altfp_compare114
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_81,
             alb => SYNTHESIZED_WIRE_301);

b2v_inst122 : altfp_compare116
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_82,
             alb => SYNTHESIZED_WIRE_302);

b2v_inst129 : altfp_compare1
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_83,
             alb => SYNTHESIZED_WIRE_294);

b2v_inst13 : altfp_compare10
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_84,
             alb => SYNTHESIZED_WIRE_295);

SYNTHESIZED_WIRE_2 <= SYNTHESIZED_WIRE_85 AND
SYNTHESIZED_WIRE_304;

SYNTHESIZED_WIRE_3 <= SYNTHESIZED_WIRE_87 AND
SYNTHESIZED_WIRE_305;

SYNTHESIZED_WIRE_4 <= SYNTHESIZED_WIRE_89 AND
SYNTHESIZED_WIRE_306;

SYNTHESIZED_WIRE_5 <= SYNTHESIZED_WIRE_91 AND
SYNTHESIZED_WIRE_295;

SYNTHESIZED_WIRE_98 <= NOT(SYNTHESIZED_WIRE_307);


SYNTHESIZED_WIRE_6 <= SYNTHESIZED_WIRE_94 AND
SYNTHESIZED_WIRE_303;

SYNTHESIZED_WIRE_7 <= SYNTHESIZED_WIRE_96 AND
SYNTHESIZED_WIRE_307;

SYNTHESIZED_WIRE_8 <= SYNTHESIZED_WIRE_98 AND
SYNTHESIZED_WIRE_308;
```

```
SYNTHESIZED_WIRE_9 <= SYNTHESIZED_WIRE_100 AND
SYNTHESIZED_WIRE_309;

SYNTHESIZED_WIRE_10 <= SYNTHESIZED_WIRE_102 AND
SYNTHESIZED_WIRE_310;

SYNTHESIZED_WIRE_11 <= SYNTHESIZED_WIRE_104 AND
SYNTHESIZED_WIRE_311;

SYNTHESIZED_WIRE_12 <= SYNTHESIZED_WIRE_106 AND
SYNTHESIZED_WIRE_312;

SYNTHESIZED_WIRE_13 <= SYNTHESIZED_WIRE_108 AND
SYNTHESIZED_WIRE_313;

SYNTHESIZED_WIRE_14 <= SYNTHESIZED_WIRE_110 AND
SYNTHESIZED_WIRE_314;

SYNTHESIZED_WIRE_15 <= SYNTHESIZED_WIRE_112 AND
SYNTHESIZED_WIRE_315;

b2v_inst15 : altfp_compare12
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_114,
             alb => SYNTHESIZED_WIRE_303);

SYNTHESIZED_WIRE_16 <= SYNTHESIZED_WIRE_115 AND
SYNTHESIZED_WIRE_316;

SYNTHESIZED_WIRE_17 <= SYNTHESIZED_WIRE_117 AND
SYNTHESIZED_WIRE_317;

SYNTHESIZED_WIRE_18 <= SYNTHESIZED_WIRE_119 AND
SYNTHESIZED_WIRE_318;

SYNTHESIZED_WIRE_19 <= SYNTHESIZED_WIRE_121 AND
SYNTHESIZED_WIRE_319;

SYNTHESIZED_WIRE_20 <= SYNTHESIZED_WIRE_123 AND
SYNTHESIZED_WIRE_320;

SYNTHESIZED_WIRE_21 <= SYNTHESIZED_WIRE_125 AND
SYNTHESIZED_WIRE_321;

SYNTHESIZED_WIRE_22 <= SYNTHESIZED_WIRE_127 AND
SYNTHESIZED_WIRE_322;

SYNTHESIZED_WIRE_23 <= SYNTHESIZED_WIRE_129 AND
SYNTHESIZED_WIRE_323;
```

```
SYNTHESIZED_WIRE_24 <= SYNTHESIZED_WIRE_131 AND
SYNTHESIZED_WIRE_324;

SYNTHESIZED_WIRE_25 <= SYNTHESIZED_WIRE_133 AND
SYNTHESIZED_WIRE_325;

SYNTHESIZED_WIRE_100 <= NOT(SYNTHESIZED_WIRE_308);


SYNTHESIZED_WIRE_26 <= SYNTHESIZED_WIRE_136 AND
SYNTHESIZED_WIRE_326;

SYNTHESIZED_WIRE_27 <= SYNTHESIZED_WIRE_138 AND
SYNTHESIZED_WIRE_327;

SYNTHESIZED_WIRE_28 <= SYNTHESIZED_WIRE_140 AND
SYNTHESIZED_WIRE_328;

SYNTHESIZED_WIRE_29 <= SYNTHESIZED_WIRE_142 AND
SYNTHESIZED_WIRE_329;

SYNTHESIZED_WIRE_30 <= SYNTHESIZED_WIRE_144 AND
SYNTHESIZED_WIRE_330;

SYNTHESIZED_WIRE_31 <= SYNTHESIZED_WIRE_146 AND
SYNTHESIZED_WIRE_331;

SYNTHESIZED_WIRE_32 <= SYNTHESIZED_WIRE_148 AND
SYNTHESIZED_WIRE_332;

SYNTHESIZED_WIRE_33 <= SYNTHESIZED_WIRE_150 AND
SYNTHESIZED_WIRE_333;

SYNTHESIZED_WIRE_34 <= SYNTHESIZED_WIRE_152 AND
SYNTHESIZED_WIRE_334;

SYNTHESIZED_WIRE_35 <= SYNTHESIZED_WIRE_154 AND
SYNTHESIZED_WIRE_335;

b2v_inst17 : altfp_compare14
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_156,
             alb => SYNTHESIZED_WIRE_307);

SYNTHESIZED_WIRE_36 <= SYNTHESIZED_WIRE_157 AND
SYNTHESIZED_WIRE_336;

SYNTHESIZED_WIRE_37 <= SYNTHESIZED_WIRE_159 AND
SYNTHESIZED_WIRE_337;
```

```
SYNTHESIZED_WIRE_38 <= SYNTHESIZED_WIRE_161 AND
SYNTHESIZED_WIRE_338;

SYNTHESIZED_WIRE_39 <= SYNTHESIZED_WIRE_163 AND
SYNTHESIZED_WIRE_339;

SYNTHESIZED_WIRE_40 <= SYNTHESIZED_WIRE_165 AND
SYNTHESIZED_WIRE_340;

SYNTHESIZED_WIRE_41 <= SYNTHESIZED_WIRE_167 AND
SYNTHESIZED_WIRE_341;

SYNTHESIZED_WIRE_58 <= SYNTHESIZED_WIRE_169 AND
SYNTHESIZED_WIRE_342;

SYNTHESIZED_WIRE_57 <= SYNTHESIZED_WIRE_171 AND
SYNTHESIZED_WIRE_343;

SYNTHESIZED_WIRE_55 <= SYNTHESIZED_WIRE_173 AND
SYNTHESIZED_WIRE_344;

SYNTHESIZED_WIRE_56 <= SYNTHESIZED_WIRE_175 AND
SYNTHESIZED_WIRE_345;

SYNTHESIZED_WIRE_102 <= NOT(SYNTHESIZED_WIRE_309);


SYNTHESIZED_WIRE_54 <= SYNTHESIZED_WIRE_178 AND
SYNTHESIZED_WIRE_346;

SYNTHESIZED_WIRE_53 <= SYNTHESIZED_WIRE_180 AND
SYNTHESIZED_WIRE_347;

SYNTHESIZED_WIRE_52 <= SYNTHESIZED_WIRE_182 AND
SYNTHESIZED_WIRE_348;

SYNTHESIZED_WIRE_51 <= SYNTHESIZED_WIRE_184 AND
SYNTHESIZED_WIRE_349;

SYNTHESIZED_WIRE_50 <= SYNTHESIZED_WIRE_186 AND
SYNTHESIZED_WIRE_350;

SYNTHESIZED_WIRE_49 <= SYNTHESIZED_WIRE_188 AND
SYNTHESIZED_WIRE_351;

SYNTHESIZED_WIRE_48 <= SYNTHESIZED_WIRE_190 AND
SYNTHESIZED_WIRE_296;

SYNTHESIZED_WIRE_47 <= SYNTHESIZED_WIRE_192 AND
SYNTHESIZED_WIRE_297;
```

```
SYNTHESIZED_WIRE_46 <= SYNTHESIZED_WIRE_194 AND
SYNTHESIZED_WIRE_298;

SYNTHESIZED_WIRE_45 <= SYNTHESIZED_WIRE_196 AND
SYNTHESIZED_WIRE_299;

b2v_inst19 : altfp_compare16
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_198,
             alb => SYNTHESIZED_WIRE_308);

SYNTHESIZED_WIRE_44 <= SYNTHESIZED_WIRE_199 AND
SYNTHESIZED_WIRE_300;

SYNTHESIZED_WIRE_43 <= SYNTHESIZED_WIRE_201 AND
SYNTHESIZED_WIRE_301;

SYNTHESIZED_WIRE_42 <= SYNTHESIZED_WIRE_203 AND
SYNTHESIZED_WIRE_302;

SYNTHESIZED_WIRE_85 <= NOT(SYNTHESIZED_WIRE_352);


SYNTHESIZED_WIRE_104 <= NOT(SYNTHESIZED_WIRE_310);


b2v_inst21 : altfp_compare18
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_207,
             alb => SYNTHESIZED_WIRE_309);

SYNTHESIZED_WIRE_106 <= NOT(SYNTHESIZED_WIRE_311);


b2v_inst23 : altfp_compare20
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_209,
             alb => SYNTHESIZED_WIRE_310);

SYNTHESIZED_WIRE_108 <= NOT(SYNTHESIZED_WIRE_312);


b2v_inst25 : altfp_compare22
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_211,
             alb => SYNTHESIZED_WIRE_311);
```

```
SYNTHESIZED_WIRE_110 <= NOT(SYNTHESIZED_WIRE_313);


b2v_inst27 : altfp_compare24
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_213,
             alb => SYNTHESIZED_WIRE_312);

SYNTHESIZED_WIRE_112 <= NOT(SYNTHESIZED_WIRE_314);


b2v_inst29 : altfp_compare26
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_215,
             alb => SYNTHESIZED_WIRE_313);

SYNTHESIZED_WIRE_87 <= NOT(SYNTHESIZED_WIRE_304);


SYNTHESIZED_WIRE_115 <= NOT(SYNTHESIZED_WIRE_315);


b2v_inst31 : altfp_compare28
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_218,
             alb => SYNTHESIZED_WIRE_314);

SYNTHESIZED_WIRE_117 <= NOT(SYNTHESIZED_WIRE_316);


b2v_inst33 : altfp_compare30
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_220,
             alb => SYNTHESIZED_WIRE_315);

SYNTHESIZED_WIRE_119 <= NOT(SYNTHESIZED_WIRE_317);


b2v_inst35 : altfp_compare32
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_222,
             alb => SYNTHESIZED_WIRE_316);

SYNTHESIZED_WIRE_121 <= NOT(SYNTHESIZED_WIRE_318);
```

```
b2v_inst37 : altfp_compare34
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_224,
             alb => SYNTHESIZED_WIRE_317);

SYNTHESIZED_WIRE_123 <= NOT(SYNTHESIZED_WIRE_319);


b2v_inst39 : altfp_compare36
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_226,
             alb => SYNTHESIZED_WIRE_318);

b2v_inst4 : altfp_compare3
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_227,
             alb => SYNTHESIZED_WIRE_352);

SYNTHESIZED_WIRE_125 <= NOT(SYNTHESIZED_WIRE_320);


b2v_inst41 : altfp_compare38
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_229,
             aleb => SYNTHESIZED_WIRE_319);

SYNTHESIZED_WIRE_127 <= NOT(SYNTHESIZED_WIRE_321);


b2v_inst43 : altfp_compare40
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_231,
             alb => SYNTHESIZED_WIRE_320);

SYNTHESIZED_WIRE_129 <= NOT(SYNTHESIZED_WIRE_322);


b2v_inst45 : altfp_compare42
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_233,
             alb => SYNTHESIZED_WIRE_321);

SYNTHESIZED_WIRE_131 <= NOT(SYNTHESIZED_WIRE_323);
```

```
b2v_inst47 : altfp_compare44
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_235,
             aleb => SYNTHESIZED_WIRE_322);

SYNTHESIZED_WIRE_133 <= NOT(SYNTHESIZED_WIRE_324);


b2v_inst49 : altfp_compare46
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_237,
             alb => SYNTHESIZED_WIRE_323);

SYNTHESIZED_WIRE_89 <= NOT(SYNTHESIZED_WIRE_305);


SYNTHESIZED_WIRE_136 <= NOT(SYNTHESIZED_WIRE_325);


b2v_inst51 : altfp_compare48
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_240,
             alb => SYNTHESIZED_WIRE_324);

SYNTHESIZED_WIRE_138 <= NOT(SYNTHESIZED_WIRE_326);


b2v_inst53 : altfp_compare50
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_242,
             alb => SYNTHESIZED_WIRE_325);

SYNTHESIZED_WIRE_140 <= NOT(SYNTHESIZED_WIRE_327);


b2v_inst55 : altfp_compare52
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_244,
             alb => SYNTHESIZED_WIRE_326);

SYNTHESIZED_WIRE_142 <= NOT(SYNTHESIZED_WIRE_328);


b2v_inst57 : altfp_compare54
PORT MAP(clock => clk,
             dataa => a,
```

```
            datab => SYNTHESIZED_WIRE_246,
            alb => SYNTHESIZED_WIRE_327);

SYNTHESIZED_WIRE_144 <= NOT(SYNTHESIZED_WIRE_329);


b2v_inst59 : altfp_compare56
PORT MAP(clock => clk,
            dataa => a,
            datab => SYNTHESIZED_WIRE_248,
            alb => SYNTHESIZED_WIRE_328);

b2v_inst6 : altfp_compare4
PORT MAP(clock => clk,
            dataa => a,
            datab => SYNTHESIZED_WIRE_249,
            alb => SYNTHESIZED_WIRE_304);

SYNTHESIZED_WIRE_146 <= NOT(SYNTHESIZED_WIRE_330);


b2v_inst61 : altfp_compare58
PORT MAP(clock => clk,
            dataa => a,
            datab => SYNTHESIZED_WIRE_251,
            alb => SYNTHESIZED_WIRE_329);

SYNTHESIZED_WIRE_148 <= NOT(SYNTHESIZED_WIRE_331);


b2v_inst63 : altfp_compare60
PORT MAP(clock => clk,
            dataa => a,
            datab => SYNTHESIZED_WIRE_253,
            alb => SYNTHESIZED_WIRE_330);

SYNTHESIZED_WIRE_150 <= NOT(SYNTHESIZED_WIRE_332);


b2v_inst65 : altfp_compare62
PORT MAP(clock => clk,
            dataa => a,
            datab => SYNTHESIZED_WIRE_255,
            aleb => SYNTHESIZED_WIRE_331);

SYNTHESIZED_WIRE_152 <= NOT(SYNTHESIZED_WIRE_333);


b2v_inst67 : altfp_compare64
PORT MAP(clock => clk,
            dataa => a,
```

```
            datab => SYNTHESIZED_WIRE_257,
            alb => SYNTHESIZED_WIRE_332);

SYNTHESIZED_WIRE_154 <= NOT(SYNTHESIZED_WIRE_334);


b2v_inst69 : altfp_compare66
PORT MAP(clock => clk,
            dataa => a,
            datab => SYNTHESIZED_WIRE_259,
            alb => SYNTHESIZED_WIRE_333);

SYNTHESIZED_WIRE_91 <= NOT(SYNTHESIZED_WIRE_306);


SYNTHESIZED_WIRE_157 <= NOT(SYNTHESIZED_WIRE_335);


SYNTHESIZED_WIRE_159 <= NOT(SYNTHESIZED_WIRE_336);


SYNTHESIZED_WIRE_161 <= NOT(SYNTHESIZED_WIRE_337);


b2v_inst73 : altfp_compare70
PORT MAP(clock => clk,
            dataa => a,
            datab => SYNTHESIZED_WIRE_264,
            alb => SYNTHESIZED_WIRE_335);

SYNTHESIZED_WIRE_163 <= NOT(SYNTHESIZED_WIRE_338);


b2v_inst75 : altfp_compare72
PORT MAP(clock => clk,
            dataa => a,
            datab => SYNTHESIZED_WIRE_266,
            alb => SYNTHESIZED_WIRE_336);

SYNTHESIZED_WIRE_165 <= NOT(SYNTHESIZED_WIRE_339);


b2v_inst77 : altfp_compare68
PORT MAP(clock => clk,
            dataa => a,
            datab => SYNTHESIZED_WIRE_268,
            alb => SYNTHESIZED_WIRE_334);

SYNTHESIZED_WIRE_167 <= NOT(SYNTHESIZED_WIRE_340);
```

```
SYNTHESIZED_WIRE_169 <= NOT(SYNTHESIZED_WIRE_341);


SYNTHESIZED_WIRE_1 <= SYNTHESIZED_WIRE_271 AND
SYNTHESIZED_WIRE_352;

b2v_inst80 : altfp_compare74
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_273,
             alb => SYNTHESIZED_WIRE_337);

SYNTHESIZED_WIRE_171 <= NOT(SYNTHESIZED_WIRE_342);


b2v_inst82 : altfp_compare76
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_275,
             alb => SYNTHESIZED_WIRE_338);

SYNTHESIZED_WIRE_173 <= NOT(SYNTHESIZED_WIRE_343);


b2v_inst84 : altfp_compare78
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_277,
             alb => SYNTHESIZED_WIRE_339);

SYNTHESIZED_WIRE_175 <= NOT(SYNTHESIZED_WIRE_344);


b2v_inst86 : altfp_compare80
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_279,
             alb => SYNTHESIZED_WIRE_340);

SYNTHESIZED_WIRE_178 <= NOT(SYNTHESIZED_WIRE_345);


b2v_inst88 : altfp_compare82
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_281,
             alb => SYNTHESIZED_WIRE_341);

SYNTHESIZED_WIRE_180 <= NOT(SYNTHESIZED_WIRE_346);
```

```
b2v_inst9 : altfp_compare6
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_283,
             alb => SYNTHESIZED_WIRE_305);

b2v_inst90 : altfp_compare118
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_284,
             alb => SYNTHESIZED_WIRE_342);

SYNTHESIZED_WIRE_182 <= NOT(SYNTHESIZED_WIRE_347);


b2v_inst92 : altfp_compare86
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_286,
             alb => SYNTHESIZED_WIRE_343);

SYNTHESIZED_WIRE_184 <= NOT(SYNTHESIZED_WIRE_348);


b2v_inst94 : altfp_compare88
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_288,
             alb => SYNTHESIZED_WIRE_344);

SYNTHESIZED_WIRE_186 <= NOT(SYNTHESIZED_WIRE_349);


b2v_inst96 : altfp_compare90
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_290,
             alb => SYNTHESIZED_WIRE_345);

SYNTHESIZED_WIRE_188 <= NOT(SYNTHESIZED_WIRE_350);


b2v_inst98 : altfp_compare92
PORT MAP(clock => clk,
             dataa => a,
             datab => SYNTHESIZED_WIRE_292,
             alb => SYNTHESIZED_WIRE_346);

SYNTHESIZED_WIRE_190 <= NOT(SYNTHESIZED_WIRE_351);
```

```
END;
```

APÊNDICE B Valores em ponto-flutuante de acordo com a IEEE 754

Posição do u	Valor de u	Valor de y	u em ponto flutuante	y em ponto flutuante
U0	-3,99945	0,017996	11000000011111111111011011111101	00111100100100110110110001011000
U1	-3,53258	0,028399	11000000011000100001010111001010	00111100111010001010010100000101
U2	-3,17369	0,040168	11000000010010110001110110111100	00111101001001001000011100110011
U3	-2,88384	0,052958	11000000001110001001000011010101	00111101010110001110101001111100
U4	-2,64131	0,066527	11000000001010010000101100111001	00111101100010000011111101001110
U5	-2,43239	0,080736	11000000000110111010110001000111	00111101101001010101100011101010
U6	-2,25141	0,095228	11000000000100000001011100011001	00111101110000110000011011100101
U7	-2,07043	0,112004	11000000000001001000000111101100	00111101111001010110001001011010
U8	-1,91019	0,12896	10111111111101001000000100011011	00111110000001000000111000010111
U9	-1,76587	0,146057	10111111111000100000100000000111	00111110000101011000111111110111

U10	-1,63411	0,163268	10111111110100010010101010000100	00111110001001110010111110111010
U11	-1,51246	0,180574	10111111110000011001100001001010	00111110001110001110100001100100
U12	-1,39909	0,19796	10111111101100110001010101100001	00111110010010101011011000000110
U13	-1,29259	0,215415	10111111101001010111001110010110	00111110010111001001010110111111
U14	-1,19185	0,232929	10111111100110001000111010001010	00111110011011101000010011110000
U15	-1,09597	0,250495	10111111100011000100100010111110	00111110100000000100000011100001
U16	-1,00424	0,268108	10111111100000001000101011101111	00111110100010010100010101110011
U17	-0,91605	0,285763	10111111011010101000001001000000	00111110100100100100111110000111
U18	-0,83089	0,303456	10111111010101001011010100110101	00111110100110110101111010010101
U19	-0,74834	0,321184	10111111001111111001001100110101	00111110101001000111001000111010
U20	-0,668	0,338945	10111111001010110000001000001100	00111110101011011000101000110010
U21	-0,58956	0,356736	10111111000101101110110101100111	00111110101101101010011000011001
U22	-0,51272	0,374555	10111111000000110100000110011110	00111110101111111100010110101100

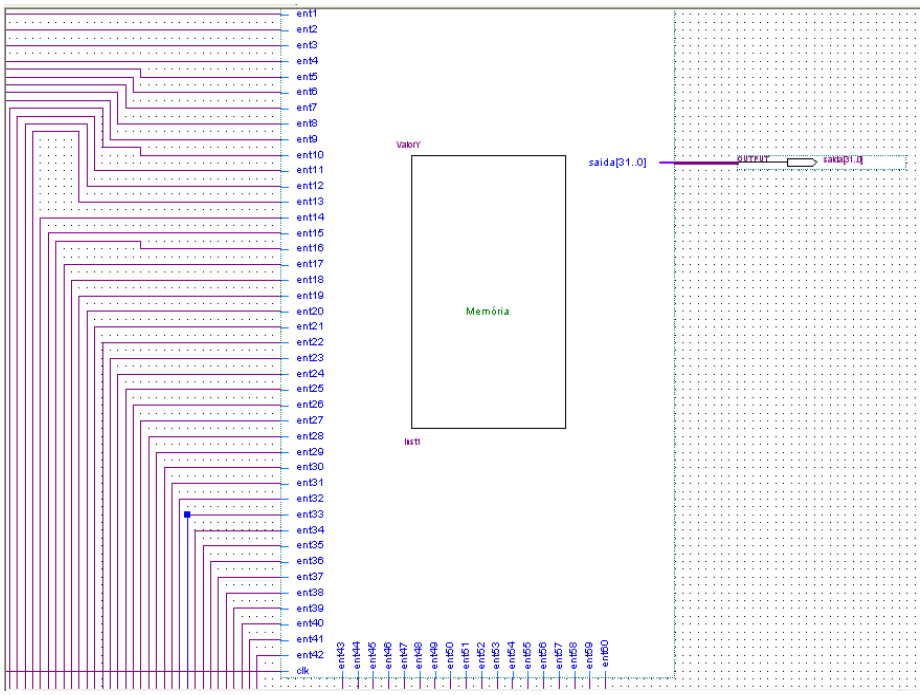
U23	-0,43723	0,392402	10111110110111111101110010011100	00111110110010001110100011101010
U24	-0,36283	0,410274	10111110101110011100010011011010	00111110110100100000111101101111
U25	-0,28931	0,428172	10111110100101000010000001110000	00111110110110110011100101011100
U26	-0,21647	0,446093	10111110010111011010101001001111	00111110111001000110011001001101
U27	-0,14409	0,464039	10111110000100111000110001010100	00111110111011011001011010000101
U28	-0,072	0,482008	10111101100100110111010010111100	00111110111101101100100111000000
U29	0	0,5	00000000000000000000000000000000	00111110000000000000000000000000
U30	0,072	0,517992	00111101100100110111010010111100	00111111000001001001101100011111
U31	0,144093	0,535961	00111110000100111000110100011101	00111111000010010011010010111101
U32	0,216468	0,553907	00111110010111011010100110101000	00111111000011011100110011011001
U33	0,289314	0,571828	00111110100101000010000011110110	00111111000100100110001101010001
U34	0,362832	0,589726	00111110101110011100010100011101	00111111000101101111100001001000

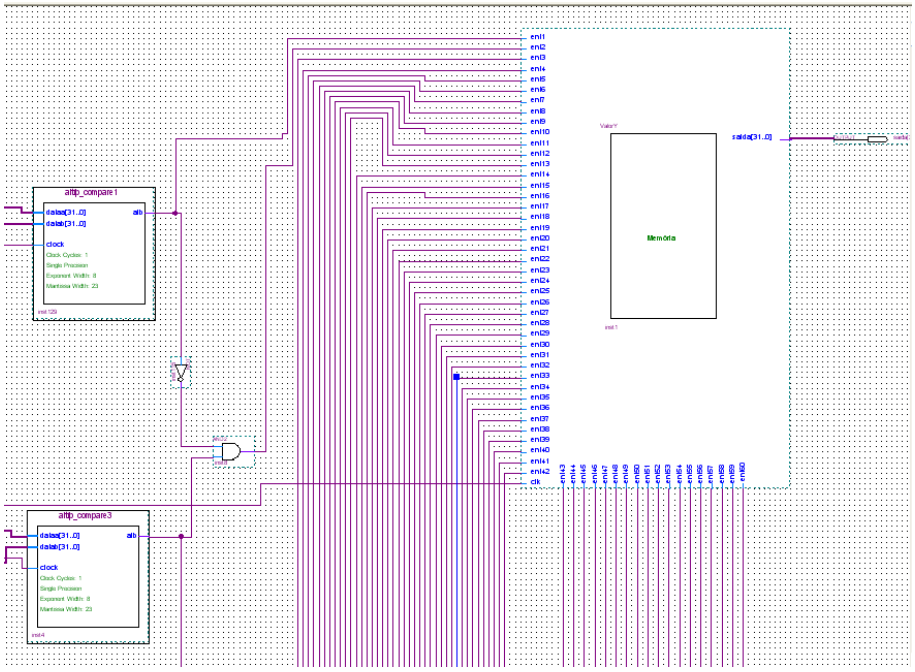
U35	0,437227	0,607598	00111110110111111101110000110111	00111111000110111000101110001010
U36	0,512724	0,625445	00111111000000110100000111100001	00111111001000000001110100101001
U37	0,58956	0,643264	00111111000000110100000111100001	00111111001001001010110011110011
U38	0,668	0,661055	00111111001010110000001000001100	00111111001010010011101011100110
U39	0,748335	0,678816	00111111001111111001001011100001	00111111001011011100011011100010
U40	0,830894	0,696544	00111111010101001011010101111000	00111111001100100101000010110101
U41	0,916053	0,714237	00111111011010101000001001110011	00111111001101101101100000111100
U42	1,004244	0,731892	00111111100000001000101100010001	00111111001110110101110101000110
U43	1,095975	0,749505	00111111100011000100100011101000	00111111001111111101111110001111
U44	1,191848	0,767071	00111111100110001000111001111001	00111111010001000101111011000011
U45	1,292591	0,784585	00111111101001010111001110011111	00111111010010001101101010010000
U46	1,399093	0,80204	00111111101100110001010101111010	00111111010011010101001001111110
U47	1,512463	0,819426	00111111110000011001100001100011	00111111010100011100010111100110

U48	1,634112	0,836732	00111111110100010010101010010100	00111111010101100011010000010001
U49	1,765872	0,853943	00111111111000100000100000010111	00111111010110101001110000000010
U50	1,910191	0,87104	00111111111101001000000100100011	00111111010111101111110001111010
U51	2,070434	0,887996	01000000000001001000000111111101	00111111011000110101001110110100
U52	2,251413	0,904772	01000000000100000001011100100110	00111111011001111001111100100011
U53	2,460328	0,921313	01000000000111010111011000000011	00111111011010111101101100101011
U54	2,708621	0,937533	01000000001011010101101000001011	00111111011100000000001000101001
U55	3,015974	0,953291	01000000010000010000010110110111	00111111011101000000101011100001
U56	3,420217	0,96833	01000000010110101110010011010101	00111111011101111110010001111001
U57	4,007173	0,98214	01000000100000000011101011000010	00111111011110110110110110000110
U58	5,033339	0,993525	01000000101000010001000100011100	00111111011111100101011110100111

APÊNDICE C Esquemático da função criada

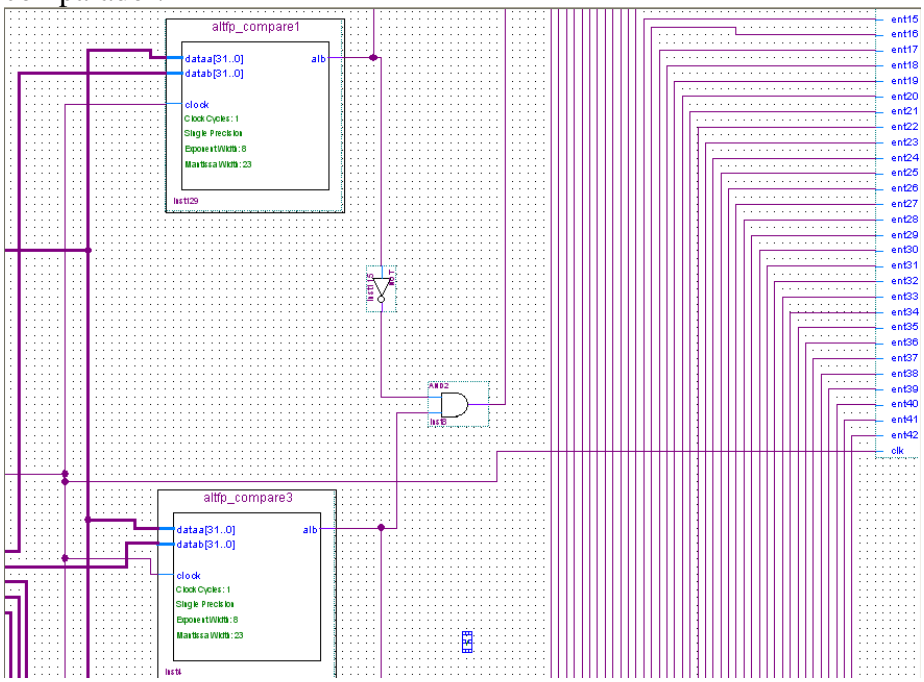
Memória que contem os valores de y correspodentes aos valores de u, cada entrada diferente está armazenado a resposta que passará para *saída*



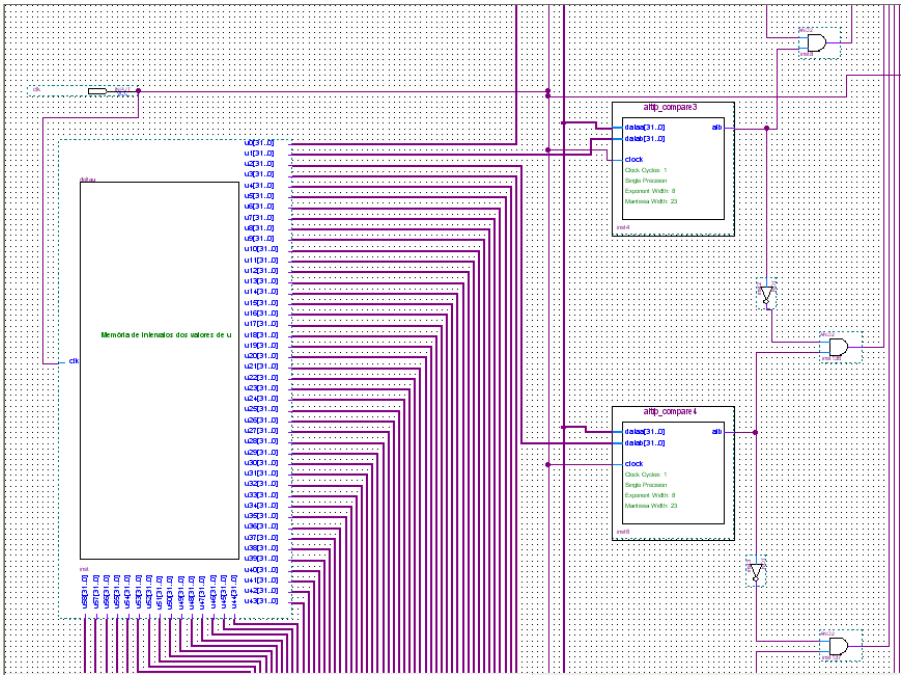


Comparadores conectados com memória, a imagem mostra somente dois dos sessenta comparadores existentes.

Um pequeno zoom nos comparadores. Entrada dataa recebe o valor de u que deseja calcular o y. Entrada datab recebe o valor de outra memória que tem os valores de u já tabelados pela tabela 3.1. A saída alb retorna 1 se dataa é menor que datab e 0 pelo contrário. A porta not indica que o valorssss é maior ou igual. Então, a porta and somente é ativada, ou seja, valor 1, se o valor for maior ou igual que o datab do primeiro comparador e menor que o datab do outro comparador.



Memória que contêm os valores de delta u e suas conexões com os comparadores



Conexão da memória com os valores de delta u, comparadores e memória com os valores de y.