Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Unidade Acadêmica de Engenharia Elétrica

Relatório de Estágio Supervisionado em Engenharia Elétrica

Italo Yure Braga Arruda

Campina Grande, Dezembro de 2009

Italo Yure Braga Arruda

IMPLEMENTAÇÃO EM FPGA DO ALGORITMO CORDIC

Relatório de estágio supervisionado submetido à Unidade Acadêmica de Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para obtenção da graduação em Engenharia Elétrica.

Orientador

Prof. Raimundo Carlos Silvério Freire

Campina Grande, Dezembro de 2009

DEDICATÓRIA

Aos meus pais, por jamais me permitirem desistir.

AGRADECIMENTOS

Agradeço a minha família pelo amor incondicional e por toda a paciência comigo.

Ao meu orientador prof. Raimundo Carlos Silvério Freire, por me proporcionar a oportunidade de trabalhar e desenvolver minhas atividades de Iniciação Científica no Laboratório de Instrumentação e Metrologia Científicas.

A todos os meus amigos que de alguma forma, direta ou indireta, contribuíram para a conclusão deste trabalho.

A minha namorada, Kelly, por todo o seu carinho e apoio.

E meus agradecimentos mais que especiais a Alan, pela sua orientação, auxílio e amizade.

RESUMO

O presente relatório de estágio apresenta os passos para a implementação de um *hardware* cuja finalidade é calcular as funções trigonométricas seno e cosseno baseado no algoritmo CORDIC. A descrição da estrutura e do comportamento do *hardware* foi feita por meio de linguagem de descrição de *hardware* (HDL - *Hardware Description Language*), HDL adotada para esse projeto foi Verilog. Os testes do *hardware* foram então realizados utilizando o FPGA (*Field Programmable Gate Array*) Cyclone II contido na placa de desenvolvimento DE2-70 da Altera.

Palavras-chave: CORDIC, FPGA, Verilog

ABSTRACT

This report presents the steps for the implementation of a hardware whose purpose is to calculate the sine and cosine trigonometric functions based on the CORDIC algorithm The hardware behavior and description was made by use of HDL (Hardware Description Language). For this project the adopted HDL was Verilog. The hardware tests were then performed using the Cyclone II FPGA (Field Programmable Gate Array) contained in the development board DE2-70 of Altera.

Keywords: CORDIC, FPGA, Verilog

Sumário

1. Introdução

Em 1959 Jack E. Volder desenvolveu um algoritmo denominado de CORDIC, acrônimo das palavras inglesas (*Coordinate Rotation Digital Computer*). Inicialmente empregado na resolução de identidades trigonométricas envolvidas na rotação de planos coordenados e na conversão de coordenadas retangulares para polares. Posteriormente, John Stephen Walther realizou generalizações desse algoritmo para rotações em sistemas circulares, lineares e hiperbólicos.

Neste relatório é apresentado o trabalho desenvolvido durante o período de estágio que consistiu na implementação de um *hardware* cuja finalidade é calcular a função trigonométrica seno e cosseno baseado no algoritmo CORDIC.

Será apresentado inicialmente o Laboratório de Instrumentação e Metrologia Científicas (LIMC), local em que foi realizado o estágio, a seguir serão apresentadas as linguagens de descrição de *hardware*, em sequência será apresentada a arquitetura típica de um FPGA (*Field Programmable Gate Array*), no item seguinte será especificado o FPGA utilizado no desenvolvimento do projeto, em seguida será descrito o funcionamento do algoritmo CORDIC, na sequência será mostrado como foi feita a implementação em FPGA, a descrição do código HDL, a verificação do *hardware* e a discussão dos resultados. Por fim, serão apresentados os comentários finais e propostas para trabalhos futuros.

2. Laboratório de Instrumentação e Metrologia Científicas

O Laboratório de Instrumentação e Metrologia Científicas (LIMC) é um dos laboratórios que compõem o núcleo de graduação e pós-graduação em Engenharia Elétrica da UFCG (Universidade Federal de Campina Grande) e está sob a coordenação do Prof. Raimundo Carlos Silvério Freire.

O Laboratório de Instrumentação e Metrologia Científica (LIMC) tem os seguinte grupos e linhas de pesquisa:

- Instrumentação Eletrônica e Eletro-Mecânica;
- Instrumentação Biomédica;
- Sensores Inteligentes;
- Aplicações de Sensores Termo-Resistivos;
- Processamento de sinais de sensores;
- Sistemas inteligentes;
- Sistemas de Aquisição e Análise de Vibrações;
- Concepção de circuitos integrados.

3. Linguagens de descrição de hardware

Com o surgimento da tecnologia VLSI (*Very Large Scale Integration*) o processo de desenvolvimento de circuitos se tornou demasiadamente complexo para que circuitos continuassem sendo testados em *protoboards* e que o leiaute fosse feito na mão e papel. Com um único *chip* contendo mais de 100 mil transistores, tornou-se então necessário que técnicas CAD (*Computer Aided Design*) evoluíssem para que pudessem auxiliar no desenvolvimento e na simulação de circuitos digitais VLSI [1].

Foi a partir dessa necessidade que surgiram as linguagens de descrição de *hardware* (HDL). Dentre as HDLs utilizadas no projeto de sistemas digitais se encontram: Verilog e VHDL (VHSIC *Hardware Description Language*, em que VHSIC significa "*Very High Speed Integrated Circuits*"). A linguagem Verilog foi originalmente desenvolvida durante 1984 como um produto proprietário da *Gateway Design Automation*. A linguagem VHDL foi desenvolvida na mesma década que a Verilog sob a responsabilidade da agência norte americana DARPA (*Defense Advanced Research Projects Agency*) [2].

Inicialmente as HDLs se tornaram populares pela verificação lógica dos circuitos, mas o desenvolvedor ainda tinha que converter o sistema descrito em HDL em um diagrama elétrico com interconexões entre as portas. A metodologia de desenvolvimento mudou radicalmente com o advento da síntese lógica. Os circuitos digitais puderam ser então descritos em nível de transferência entre registradores (RTL – *Register Transfer Level*). Dessa forma, tornou-se necessário especificar apenas o fluxo de dados entre os registradores e como o sistema processa esses dados. A partir da descrição RTL as ferramentas de síntese lógica extraiam automaticamente os detalhes das portas e de suas interconexões para implementar os circuitos [1].

Atualmente, uma linguagem de descrição de *hardware* pode ser definida como uma linguagem específica orientada à descrição da estrutura e do comportamento de um *hardware*. A descrição estrutural descreve a interconexão entre os componentes que fazem parte do circuito. A descrição comportamental descreve o funcionamento de cada um dos componentes do circuito [2].

4. FPGA

Na indústria de semicondutores moderna é possível que um engenheiro desenvolva um ASIC (*Application Specific Integrated Circuit*) com um mínimo, ou nenhum, conhecimento específico da física de semicondutores ou processos de semicondutores. Isso deve-se ao fato de que os fornecedores disponibilizam uma biblioteca de células e funções que o desenvolvedor (engenheiro) pode usar sem saber como são implementadas no silício[8].

Um tipo específico de ASIC é o que possui um arranjo de portas (*gate array*) com uma arquitetura particular que consiste de linhas e de colunas formadas por estruturas regulares de transistores. Sendo cada célula básica, ou porta, consistindo do mesmo número de transistores que inicialmente não estão conectados[8].

Os FPGAs são membros de uma classe de dispositivos chamada de FPL (*Field-Programmable Logic*), que são como dispositivos programáveis contendo repetitivos campos e pequenos elementos lógicos (LE – *Logic Element*) ou blocos lógicos[9]. Foram criados pela Xilinx Inc. e surgiram em 1985.

O projeto de um ASIC clássico requer passos adicionais do processo de fabricação de semicondutores além dos exigidos por um FPL[9]. Além disso, devido ao fato dos FPGAs serem estruturados como um ASIC formado por arranjo de portas isso os torna adequados para serem usados na prototipagem de ASICs .

Apesar da arquitetura do FPGA variar em alguns detalhes de fabricante para fabricante eles ainda conservam uma estrutura em comum, consistindo de blocos lógicos configuráveis (CLB – *Configurable Logic Block*), blocos configuráveis de Entrada/Saída (IOB – *Input/Output Block*), e matrizes de chaves de interconexão (*Switch Matrix*)[2]. Além disso, devem haver circuitos de *clock* direcionando os sinais de *clock* para cada bloco lógico e recursos adicionais como ULAs, memórias e decodificadores. Na figura 1 [2] é apresentada uma estrutura simplificada do FPGA.

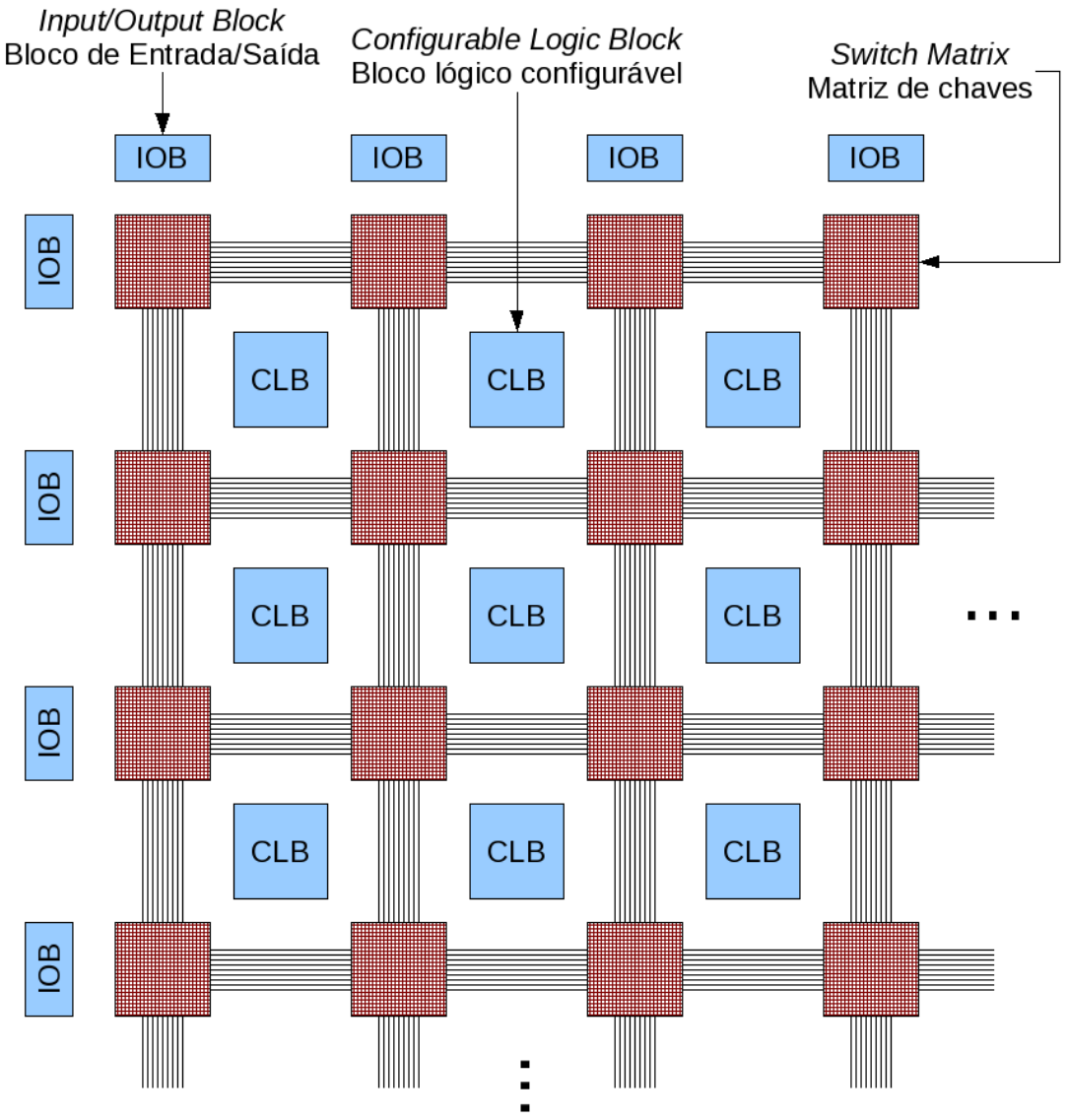

Figura 1: Estrutura simplificada do FPGA [2]

Os CLBs são a unidade lógica básica do FPGA, seu número exato e suas características variam de acordo com o dispositivo. Porém, podem ser definidos como circuitos idênticos formados por chaves de interconexão, por alguma seleção de circuito (MUX, etc) e por flip-flops. As chaves de interconexão são altamente flexíveis e podem ser configuradas para manipular lógica combinatória, registradores de deslocamento ou memória RAM. Numa arquitetura de grãos grandes eles contêm lógica suficiente para criar uma pequena máquina de estados. Numa arquitetura de grãos pequenos os

CLBs são capazes somente de armazenar uma lógica muito básica[8].

A Entrada/Saída em FPGAs é agrupada em bancos, cada um destes bancos suporta de forma independente diferentes padrões de Entrada/Saída. Um bloco configurável de entrada/saída é responsável por receber e enviar sinais do *chip*. Ele consiste de um *buffer* de entrada e um *buffer* de saída de três estados. Podemos encontrar resistores *pull up* ou resistores *pull down* na saída e ainda um flip-flop fazendo com que os sinais síncronos possam ser enviados a saída diretamente aos pinos sem encontrar atrasos significativos. Mais de uma dúzia de tipos de Entrada/Saída podem ser fornecidos pelo FPGA, permitindo a flexibilidade ao suporte de Entrada/Saída.

A interconexão em um FPGA pode ser feita de diferentes formas por meio de vias de interconexão flexíveis que roteiam os sinais entre os CLBs, entre os blocos de Entrada/Saída, ou entre os CLBs e os blocos de Entrada/Saída. Longas vias podem ser usadas para conectar CLBs que estão fisicamente distantes sem induzir muitos atrasos e ainda podem ser usadas como barramentos no interior do chip. Pequenas vias são usadas para a conexão de CLBs individuais localizados fisicamente próximos um ao outro. Várias chaves de interconexão conectam essas longas e pequenas vias juntas em caminhos específicos. Longas vias especiais, chamadas vias de *clock* global, são feitas especialmente para terem uma baixa impedância e deste modo rápida propagação de sinal, são conectadas aos *buffers* do *clock* e a cada elemento síncrono de cada CLB, deste modo, distribuindo o *clock* através do FPGA.

O grande atrativo para a utilização de sistemas baseados em FPGA é o baixo custo para o desenvolvimento de um *hardware,* combinado a isso estão três características: a reconfigurabilidade do *hardware,* a elevada capacidade computacional e um menor ciclo de projeto[2].

4.1. Placa Altera DE2-70

A Altera DE2-70 (figura 2) é uma placa educacional e de desenvolvimento, equipada com quase 70.000 LE's (Logic Element) da Altera Cyclone II 2C70. A placa oferece um rico conjunto de características possíveis de serem utilizadas em ambiente laboratorial para uma variedade de projetos, tão quanto para o desenvolvimento de sistemas digitais sofisticados.

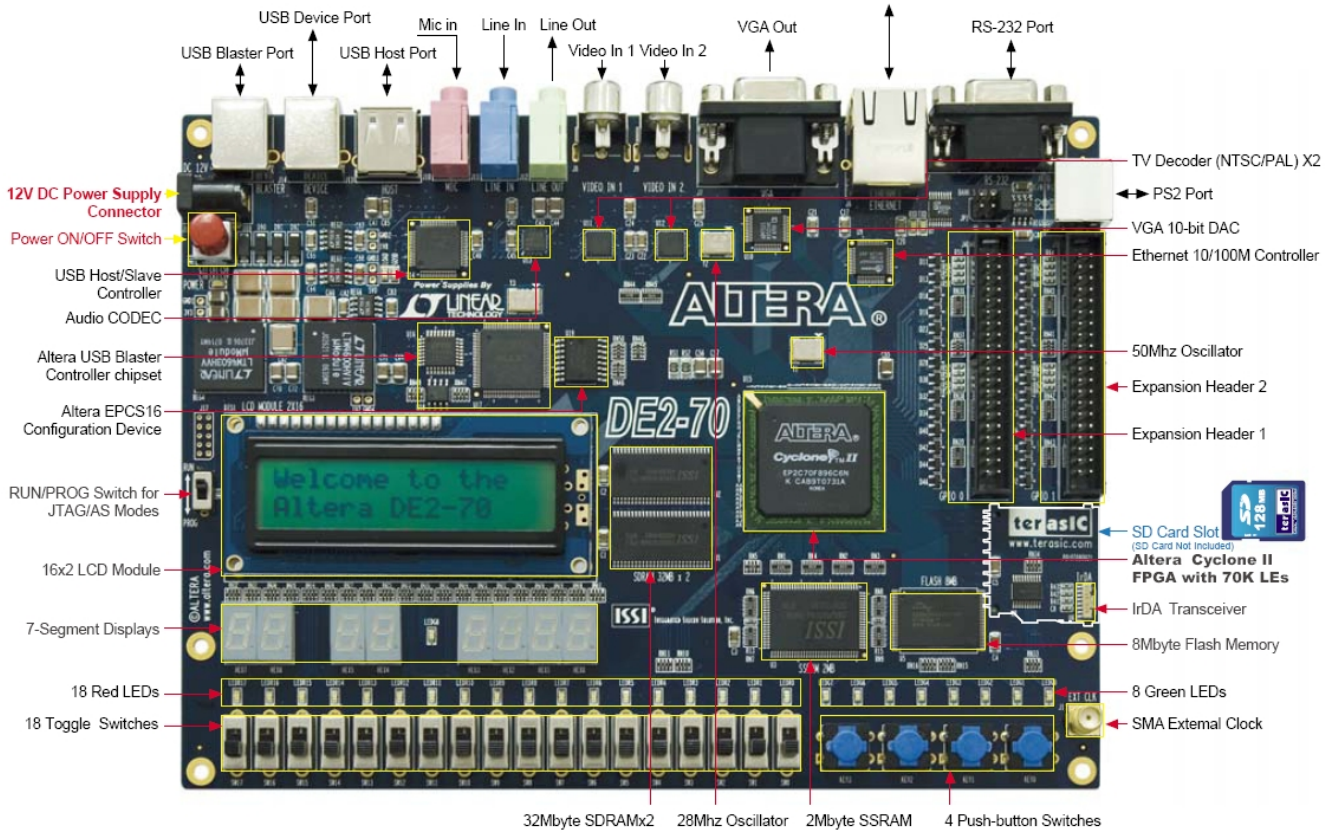

Figura 2: Placa Altera DE2-70

A placa DE2-70 possui diversos componentes periféricos que permitem ao usuário implementar circuitos simples, que utilizam apenas chaves e *LEDs* (*Light Emitting Diode*), como também circuitos que apresentem um maior grau de complexidade (transferências de dados via USB (*Universal Serial Bus*), exibição de dados em monitores VGA (*Video Graphics Array*), armazenamento de dados em

HDs (*Hard Disk*), sendo portanto adequada a uma grande variedade de circuitos e aplicações.

Os seguintes itens de *hardware* estão disponíveis na placa DE2-70 da Altera:

- Dispositivo FPGA Altera Cyclone® II 2C70;
- Dispositivo Altera de Configuração Serial – EPCS16;
- USB Blaster (*on board*) para programação e controle API;
- 2-Mbyte SSRAM;
- 32-Mbyte SDRAM;
- 8-Mbyte Memória Flash;
- Socket para Cartão SD;
- 4 botões;
- 18 chaves;
- 18 LEDs vermelhos;
- 9 LEDs verdes;
- Osciladores de 50-MHz e 28.63-MHz para funcionarem como fontes de *clock*;
- CODEC de áudio de 24 bits com qualidade de CD;
- VGA DAC (10-bit high-speed triple DACs) com conector VGA-out;
- 2 TV Decoder (NTSC/PAL/SECAM) e conector TV-in;
- 10/100 Controlador Ethernet com um conector;
- Controlador USB Host/Slave com conectores USB tipo A e tipo B;
- RS-232 transceiver e conector de 9 pinos;
- Conector PS/2 mouse/teclado;
- Transceptor IrDA;

- 1 conector SMA;
- 2 Expansion Headers de 40 pinos com proteção de diodo.

Para fornecer máxima flexibilidade ao usuário, todas as conexões são feitas diretamente no dispositivo FPGA Cyclone II. Isso permite ao usuário poder configurar o FPGA facilmente afim de implementar qualquer projeto.

5. Algoritmo CORDIC

CORDIC é o acrônimo das palavras inglesas (*Coordinate Rotation Digital Computer*). Algoritmo inicialmente desenvolvido em 1958 por Jack E. Volder no departamento de aeroeletrônica da Conair com o objetivo de substituir o computador analógico do sistema de navegação do bombardeiro B-58 [3].

Quando desenvolvido por Volder, este algoritmo realizava cálculos de rotação de vetores em um plano. Posteriormente, John S. Walther fez generalizações para rotações em sistemas circulares, lineares e hiperbólicos [2].

Este algoritmo é capaz de realizar o cálculo de funções trigonométricas, hiperbólicas, exponenciais e logarítmicas, multiplicações, divisões e raízes quadradas. Sendo as únicas operações exigidas a adição, subtração e deslocamento de bits, além disso, uma tabela com valores de constantes previamente armazenadas [4].

O algoritmo CORDIC é derivado das seguintes relações:

$$\begin{aligned} x' &= x \cdot \cos(\phi) - y \cdot \sin(\phi) \\ y' &= y \cdot \cos(\phi) + x \cdot \sin(\phi) \end{aligned} \tag{1}$$

que rotaciona um vetor em um plano cartesiano de um ângulo ϕ . Estas relações podem ser reorganizadas de modo a obter:

$$\begin{aligned} x' &= \cos(\phi)[x - y \cdot \tan(\phi)] \\ y' &= \cos(\phi)[y + x \tan(\phi)] \end{aligned} \tag{2}$$

Se os ângulos de rotação são restringidos de modo que $\tan(\phi) = \pm 2^{-i}$, a multiplicação pelo termo de tangente é reduzido a uma simples operação de deslocamento. Os ângulos são obtidos realizando uma série de pequenas rotações. Se a decisão a cada iteração, *i*, é em qual direção rotacionar, então o termo $\cos(\phi)$ torna-se uma constante (porque $\cos(\phi) = \cos(-\phi)$. Agora, pode-se expressar a rotação iterativa como

$$x_{i+1} = K_i[x_i - y_i \cdot d_i \cdot 2^{-i}]$$
$$y_{i+1} = K_i[y_i - x_i \cdot d_i \cdot 2^{-i}] \quad (3)$$

Em que:

$$K_i = \cos(\tan^{-1} 2^{-i}) = 1/\sqrt{1+2^{-2i}} \quad (4)$$

$$d_i = \pm 1 \quad (5)$$

O algoritmo de rotação possui um ganho, A_n , de aproximadamente 1,647. O ganho exato depende do número de iterações e obedece a seguinte relação

$$A_n = \prod \sqrt{1+2^{-2i}} \quad (6)$$

O produto de K_i é, portanto, um fator de ajuste ao ganho A_n podendo ser aplicado em qualquer parte do sistema. Esse produto se aproxima do valor 0,6073 quando o número de iterações tende ao infinito.

Um melhor método de conversão utiliza um somador-subtrator adicional que acumula os ângulos de rotação a cada iteração. Os valores desses ângulos são fornecidos por uma tabela. O acumulador dos ângulos introduz uma terceira equação ao algoritmo:

$$z_{i+1} = z_i - d_i \cdot \tan^{-1}(2^{-1}) \quad (7)$$

O algoritmo CORDIC é normalmente operado em dois modos, o modo de rotação e o modo de vetorização.

Utilizando o modo de rotação, o acumulador de ângulos é iniciado com o ângulo de rotação desejado. A decisão de rotação a cada iteração é fazer a magnitude do ângulo residual diminuir a cada iteração. A decisão a cada iteração baseia-se no sinal do ângulo residual a cada etapa. Portanto, no modo de rotação as equações do CORDIC são:

$$x_{i+1} = x_i - y_i \cdot d_i \cdot \tan(\alpha_i)$$
$$y_{i+1} = y_i + x_i \cdot d_i \cdot \tan(\alpha_i) \quad (8)$$
$$z_{i+1} = z_i - d_i \cdot \alpha_i$$

Em que:

- $d_i = -1$ se $z_i < 0$, $+1$ caso contrário
- α_i são valores de ângulos que produzem as rotações curtas

A figura 3 apresenta graficamente o funcionamento do algoritmo CORDIC no modo de rotação. O algoritmo inicia com a informação do valor de z_0 , sendo z_0 o ângulo o qual desejamos descobrir os valores de seno e cosseno. Um vetor é rotacionado em torno do ângulo desejado, na figura, representado pelo círculo, o valor inicial desse vetor é $x_0 = 1$ e $y_0 = 0$. Por meio de uma sequência de rotações curtas e conhecidas a cada iteração esse vetor se aproxima do ângulo fornecendo os valores do seno, indicado pelo eixo y, e cosseno, informado pelo eixo y, do ângulo desejado.

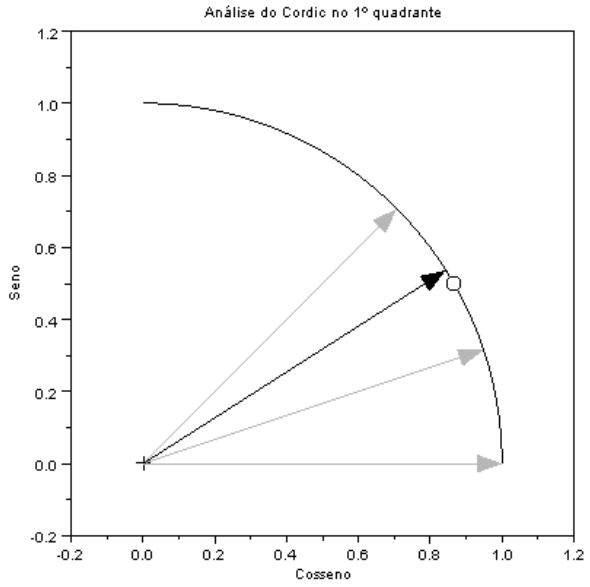

Figura 3: Funcionamento do CORDIC no modo de operação de rotação

Portanto, quanto maior o número de iterações mais exato será o valor do seno e cosseno desejado. O critério de parada escolhido foi o número de iterações, outra regra de parada poderia ser a monitoração do ângulo residual, entretanto, para que esta regra de parada fosse viável seria necessário adicionar mair elementos lógicos, aumentando assim a complexidade do sistema. Os valores de α , assim como os de sua tangente, são armazenados em uma tabela.

6. Implementação em FPGA

6.1. Fluxo de projeto

Um fluxo de projeto típico para circuitos VLSI descritos utilizando linguagem de descrição de *hardware* é apresentado na figura 4.

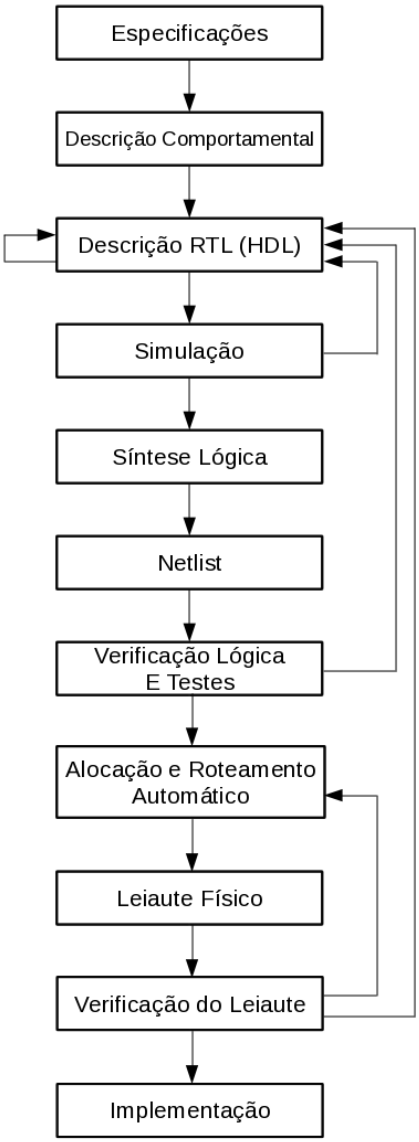

Figura 4: Fluxo de projeto típico

Existem diferentes maneiras de se descrever um circuito: por meio de diagramas elétricos, por meio de HDLs, com a combinação de ambas, entre outras. Cabe ao desenvolvedor escolher qual das abordagens irá satisfazer os requisitos exigidos pelo projeto. Uma descrição por meio de diagramas elétricos fornece uma visão mais detalhada do circuito,uma descrição em nível mais baixo. Nesse nível de descrição o desenvolvedor sabe como são feitas as interconexões entre as portas do circuito, por exemplo. Uma descrição utilizando HDL não fornece um nível de detalhamento tão elevado, deve ser utilizado quando o projeto permite um nível de abstração maior.

Qualquer fluxo de projeto inicia com as especificações. As especificações descrevem a funcionalidade, interface e arquitetura geral do circuito digital proposto, neste momento não há preocupação de como o circuito será implementado. O fluxo continua com a descrição comportamental cujo objetivo é analisar o projeto em termos de funcionalidade, desempenho, cumprimento das normas e outros problemas de nível mais elevado [1].

A descrição RTL é obtida manualmente a partir da descrição comportamental. Desse ponto em diante, o desenvolvimento do projeto é realizado com o auxílio de ferramentas CAD [1].

As ferramentas de síntese lógica convertem a descrição RTL em uma *netlist* em nível de portas Uma *netlist* em nível de portas é uma descrição do circuito em forma de portas e das conexões entre elas.

A maior parte do tempo do fluxo de projeto é consumido em verificações e testes. Antes que qualquer circuito seja enviado para a fabricação deve-se ter certeza que ele atende às especificações e qual o seu comportamento quando submetido a diferentes condições. Então, há uma atividade intensa na otimização da descrição RTL do circuito. Como é possível observar na figura 4 a cada verificação ou teste do projeto pode ser necessário voltar ao ponto do fluxo da descrição RTL do circuito.

6.2. Representação dos Dados

Para este projeto, a representação dos dados adotada segue o padrão IEEE 754 (*Standard for Binary Floating Point Arithmetic*) [5]. O padrão IEE 754 estabelece formatos de ponto flutuante usados para representar subconjuntos dos números reais [2]. O formato escolhido foi o “precisão simples” (*Single-Precision Format*). A estrutura representando esse o formato é apresentada na figura 5.

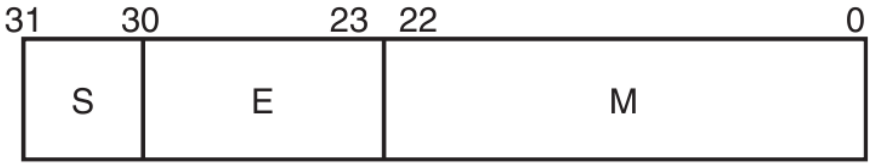

Figura 5: Representação do padrão IEEE 754 com formato em "precisão simples"

O formato “precisão simples” utiliza 32 bits para a representação do dado, sendo que:

- O bit mais significativo é o bit de sinal;
- Os 8 bits seguintes representam o expoente;
- Os 23 bits restantes correspondem à mantissa.

6.3. Descrição do código HDL

O algoritmo CORDIC foi implementado utilizando a linguagem de *hardware* Verilog. Todo o código foi desenvolvido com auxílio da ferramenta CAD Quartus® II da Altera. O Quartus® II fornece um ambiente para que o código Verilog seja escrito, verificado, compilado, e inserido no FPGA.

Um código em Verilog é formado por vários módulos (*modules)*, ou seja, um projeto em Verilog é uma hierarquia de módulos. Os módulos possuem, entradas, saídas, declarações de variáveis, fios, ou registradores, blocos de lógica sequencial ou combinacional e instâncias de outros módulos. Os módulos são capazes de se comunicarem entre si, assim as entradas e saídas dos módulos são interconectadas, isso é chamado de instanciação.

O código Verilog do CORDIC é implementado com a instanciação de vários módulos. A figura 6 apresenta a arquitetura do CORDIC, cada bloco da figura representa um módulo do código e as setas indicam como cada um desses módulos estão interconectados.

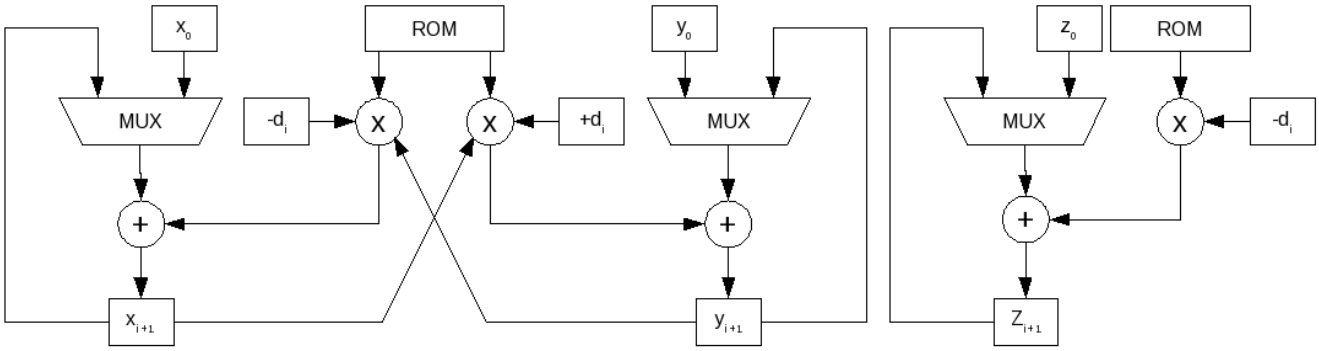

Figura 6: Arquitetura do CORDIC

A instanciação de todos os módulos é realizada em um único módulo principal, denominado de *top*, portanto, não há sub-hierarquia. Um módulo, que não está representado na figura 6, conectado a todos os demais realiza o controle do CORDIC, armazena as palavras de controle fornecendo uma nova instrução a cada ciclo de *clock*. As ROMS armazenam as tabelas que contém os valores dos ângulos e os valores das tangentes das equações (8). Os módulos de soma e multiplicação são módulos prontos

fornecidos pelo *software* Quartus® II, denominados de *megafunctions*. Esses somadores e multiplicadores são capazes de realizar as operações algébricas com os dados no formato de "precisão simples" do padrão IEEE 754. Na figura 6, os blocos $+d_i$ e $-d_i$ representam os módulos que realizam o tratamento da variável d_i das equações (8). Os blocos x_0 , y_0 e z_0 representam os módulos que armazenam os valores iniciais de *x y* e *z* das equações (8), e os blocos x_{i+1} , y_{i+1} e z_{i+1} representam os módulos que armazenam os valores *x y* e *z* durante as iterações do CORDIC. Há ainda os módulos que descrevem os multiplexadores utilizados no *hardware*.

Como a verificação desse *hardware* é feita de maneira bastante simples, apenas verificar os bits que representam o valor do seno ou cosseno desejado, optou-se por não realizar a sua simulação. Foi implementado um *hardware* para a realização dos testes e verificações do CORDIC utilizando a placa DE2-70 da Altera. Pelo fato de *megafunctions* terem sido utilizadas o projeto tornou-se dedicado a arquitetura da Altera, não podendo tornar-se ASIC.

6.4. Verificação do hardware

Os testes do *hardware* do CORDIC foram realizados utilizando o FPGA Altera DE2-70. Inserido no código Verilog do CORDIC há um código que implementa o *hardware* utilizado para a verificação e testes. Os testes foram feitos utilizando os 16 LEDs vermelhos e uma das chaves disponíveis no FPGA. Os resultados do cosseno calculado eram apresentados no formato “precisão simples” do padrão IEEE 754, cada um dos LEDs vermelhos exibia um bit do valor do cosseno, quando colocada a chave na posição 1 eram observados os dezesseis bits mais significativos do resultado, na posição 0 os 16 bits menos significativos eram exibidos. Para avaliar os valores obtidos pelo *hardware* o algoritmo do CORDIC foi desenvolvido também em *software* utilizando o Scilab. Foram realizados testes com 2 a 15 iterações para o cálculo do cosseno de um ângulo de 30 graus. Os resultados das iterações obtidas quando utilizando o *hardware* e quando utilizando o *software* são exibidos na tabela 1.

Tabela 1: Valores do cos(30°) obtidos em hardware e software

Iteração	Cos(30°)	
	Hardware	Software
2	0,9109499	0,9109500
3	0,8350374	0,8350375
4	0,9014608	0,9014609
5	0,8747728	0,8747729
6	0,8596682	0,8596683
7	0,8676476	0,8676478
8	0,8637629	0,8637630
9	0,8657318	0,8657318
10	0,8667096	0,8667097
11	0,8662223	0,8662224
12	0,8659783	0,8659784
13	0,8661004	0,8661005
14	0,8660394	0,8660395
15	0,8660699	0,8660700

6.5. Discussão dos resultados

Nos testes realizados os resultados do cálculo do cosseno de 30°,exibidos na tabela 6, forneceram valores muito próximos, idênticos até a sexta casa decimal, para a implementação do CORDIC em *hardware* e em *software*. Então, se utilizado o *software* como referência o *hardware* obtido correspondeu as especificações desejadas.

Quando comparado os resultados obtidos com o valor tabelado do cosseno de 30° ($\cos(30°)=0{,}8660254$) o erro percentual obtido ao final da 15ª iteração foi de 0,005%. Os erros absolutos obtidos em cada iteração são apresentados na figura 7.

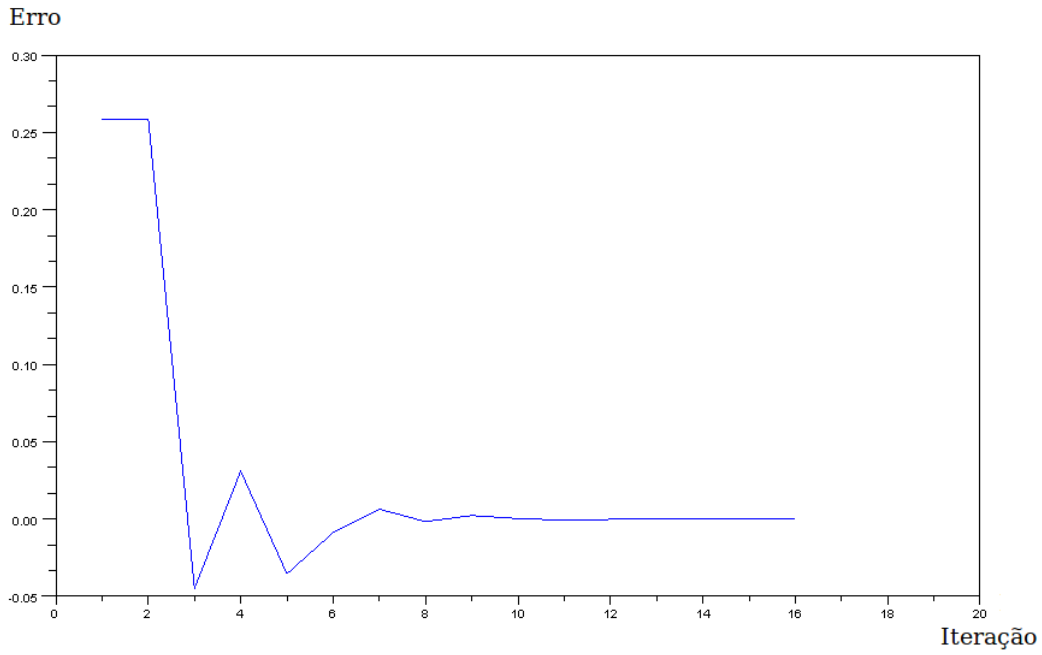

Figura 7: Erro absoluto do cálculo do cosseno de 30° obtido com o CORDIC

A fim de tornar a implementação do CORDIC em *hardware* mais simples foi utilizado um valor de K (equação 4) constante e igual a 0,6073. Isso fez com que a lógica do *hardware* fosse simplificada, mas, apesar de não ser possível visualizar na figura 7, com o aumento do número de iterações foi observado que o cálculo do cosseno não converge para o valor exato tabelado, tendo

sempre um erro associado.

Para a solução desse problema é indicado que o *hardware* do CORDIC seja modificado para que o valor correto de K seja calculado a cada iteração. Como a determinação de K envolve uma divisão e o cálculo de uma raiz quadrada (equação 4) a lógica se tornaria bastante complexa, outra solução seria utilizar os valores corretos de K para cada iteração previamente armazenados em uma memória, assim como foi feito para os ângulos α e para as tangentes do ângulo α (equação 8), já que os valores de K não dependem do ângulo a ser calculado.

7. Conclusões

Nesse relatório de estágio foi apresentado a implementação do algoritmo CORDIC em *hardware* utilizando a linguagem de descrição de *hardware* Verilog. Os resultados obtidos na verificação do *hardware* quando comparados com os valores calculados em *software* foram idênticos até a sexta casa decimal, portanto, o *hardware* correspondeu as especificações desejadas. O erro percentual obtido ao final da 15ª iteração foi de 0,005%. Como sugestões para trabalhos futuros, o *hardware* do CORDIC poderia ser modificado para que o valor correto de K seja utilizado a cada iteração, além disso, poderia ser utilizado um formato de "precisão dupla" do padrão IEEE 754 para a representação dos dados.

8. Referências

[1] PALNITKAR, Samir: “Verilog HDL: A Guide to Digital Design and Synthesis”, Prentice Hall, 1996.

[2] SÁ, Alan: “Estimação de frequência de sinais digitais baseada no paradigma conexionista”, Dissertação de mestrado, UFCG, 2009.

[3] VOLDER, Jack E.: “The Birth of CORDIC.” Journal of VLSI Signal Processing 25, pages 101–105, 2000.

[4] WALTHER, J. S.: “A unified algorithm for elementary functions.” Proceedings of the AFIPS Spring Joint Computer Conference, pages 379-385, 1971.

[5] IEEE Standard #754-2008.

[6] VOLDER, Jack E.: “The CORDIC trigonometric computing technique.” , IRE Transactions on electronic computers, pages 330-334, September 1959.

[7] ANDRAKA, Ray, ”A survey of CORDIC algorithms for FPGA based computers”, Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays, March 1998, pages 191-200.

[8] ZEIDMAN, Bob: “Introduction to CPLD and FPGA Design”.

[9] MEYER-BAESER, Uwe: “Digital signal processing with field programmable gate arrays”, Springer, 2001.

9. Anexos

9.1. Código do CORDIC implementado em software

```
// Universidade Federal de Campina Grande
// Centro de Engenharia Eletrica e Informatica
// Departamento de Engenharia Eletrica
// Relatorio de Estagio
// Aluno: Italo Yure Braga Arruda
//-----------------------------------------------

clear;
clc;
//clf;    //comando para limpar figura

// Insira aqui o valor do angulo (em graus):
theta = 30;
// Insira aqui o numero de iteracoes usadas no processo:
I = 15;

// Essas sao as variaveis de inicializacao do algoritmo
  x(1) = 1;
  y(1) = 0;
  z(1) = theta;
  d(1) = 1;
  K = 0.6073;
  x(1) = K*x(1);

for i=1:I
  if z(i) >=0
    d=+1;
  else
    d=-1;
  end
  k=1;
//  k = (1/sqrt(1+2^(-2*(i-1))));
  x(i+1) = k*(x(i) -(y(i)*d*(1/2^(i-1))));
  y(i+1) = k*(y(i) +(x(i)*d*(1/2^(i-1))));
  z(i+1) = z(i) -(d*(atan(1/2^(i-1)))*180/%pi);
end



//Valor do cosseno de pi/6 tabelado
cos(%pi/6);
```

```
//Calculo do erro
erro = cos(%pi/6) - x;

//Plotar erro
plot(erro)


// Desenhar a Area de plotagem;
plot2d(0,0,-1,"031"," ",[-0.1,-0.1,1.1,1.1])
xtitle('Analise do Cordic no 1 quadrante ','Cosseno','Seno');

//Desenhar o circulo unitario;
xarc(-1,1,2,2,0,90*64)

////Desenhar os vetores
for it = 2:14
  xpause(500000); //useg
  plot2d(cos(theta*%pi/180), sin(theta*%pi/180), -9)
  xset("clipgrf")
    xarrows([0;x(it-1)],[0;y(it-1)],1,color("grey70"))//Ref da iteração - 1
    xarrows([0;x(it)],[0;y(it)],1,1)                  //Ref da iteração
  xset("clipoff")
end
```

9.2. Código do CORDIC implementado em HDL

```
module cordic ( iSW, iKEY, iCLK_50, oLEDR, oLEDG);

	input [0:0]	iCLK_50;
	input [0:0] iSW;
	input [0:0] iKEY;

	output [15:0] oLEDR;
	output [8:0] oLEDG;

	reg [5:0] estado;
	reg [3:0] contador_base = 4'b0000;
	reg [0:0] comp_out;
	reg [15:0] buffer;
	reg [31:0] cosomega;


	//wire [0:0]		w_div;			// clock
	wire [0:0]	en_contador;		// contador
	wire [4:0]		w_contador_out;
	wire [31:0]	w_regX_out;		// registrador estático de X
	wire [31:0]	w_regY_out;		// registrador estático de Y
	wire [31:0]	w_regZ_out;		// registrador estático de Z
	wire [0:0]	en_reg;				// enable regx, regy , regz;
	wire [31:0]	w_regx_out;		// registrador dinâmico de x
	wire [31:0]	w_regy_out;		// registrador dinâmico de y
	wire [31:0]	w_regz_out;		// registrador dinâmico de z
	wire [0:0]	ct_mux;				// enable dos mux's
	wire [31:0]		w_muxx_out;		// mux's
	wire [31:0]		w_muxy_out;
	wire [31:0]	w_muxz_out;
	wire [31:0]	w_reg0_out;		// reg0
	wire [0:0]	en_comp;			// comp
	wire [0:0]		w_comp_out;
	wire [0:0]	en_t_angle;			// memoria t_angle
	wire [31:0]	w_t_angle_out;
	wire [0:0]	en_i2;				// memoria i2
	wire [31:0]	w_i2_out;
	wire [0:0]	en_mult;			// enable do multiplicador
	wire [31:0]	w_multx_out; // multiplicadores
	wire [31:0]	w_multy_out;
	wire [31:0]	w_tratadx_out;		// tratad (tratamento da direção)
```

```
	wire [31:0]	w_tratady_out;		// tratad (tratamento da direção)
	wire [31:0]	w_tratadz_out;		// tratad (tratamento da direção)
	wire [0:0]	en_addsub;			// enable adders e subs
	wire [31:0]	w_subx_out;		// saida do sub de x
	wire [31:0]	w_subz_out;		// saida do sub de z
	wire [31:0]	w_addy_out;		// saida do add de y

	wire [31:0] w_regK_out;
	wire [0:0]  cl_multk, en_multk;
	wire [31:0] w_multk_out;

	wire [9:0]		w_comandos;		// my_ucode

// ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- -----

	//module my_counter (clock, en, cont);
	my_counter	contador	(iCLK_50, en_contador, w_contador_out);

	//module registerSZ (registerSZ_out);
	registerSX	regX		(w_regX_out);
	registerSY	regY		(w_regY_out);
	registerSZ	regZ		(w_regZ_out);
	registerSK	regK		(w_regK_out);

	//module registerz (r_in, r_refresh, r_out);
	register	regx		(w_subx_out, en_reg, w_regx_out);
	register	regy		(w_addy_out, en_reg, w_regy_out);
	register	regz		(w_subz_out, en_reg, w_regz_out);

	//module my_mux2p1 (in0, in1, ctrl, mux2p1_out);
	my_mux2p1	muxx		(w_regX_out, w_regx_out, ct_mux, w_muxx_out);
	my_mux2p1	muxy		(w_regY_out, w_regy_out, ct_mux, w_muxy_out);
	my_mux2p1	muxz		(w_regZ_out, w_regz_out, ct_mux, w_muxz_out);

	//module registerS0 (registerS0_out);
	registerS0	reg0		(w_reg0_out);

	//module mf_compare (clk_en, clock, dataa, datab, ageb);
	mf_compare	comp		(en_comp, iCLK_50, w_muxz_out, w_reg0_out, w_comp_out);


	//module mf_ROM_ang (address, clock, q);
	//mf_ROM_ang	t_angle		(w_contador_out, en_t_angle, w_t_angle_out);
	//module my_ROM_ang (address,data);
	my_ROM_ang	t_angle		(w_contador_out, w_t_angle_out);

	//module mf_ROM_2i ( address, clken, clock, q);
```

```
	//mf_ROM_2i i2			(w_contador_out, en_i2, w_div, w_i2_out);
	//module my_ROM_2i (address,data);
	my_ROM_2i  i2			(w_contador_out, w_i2_out);


	//module mf_mult (clk_en, clock, dataa, datab, result);
	//module mf_mult ( aclr, clk_en, clock, dataa, datab, result);
	mf_mult		multx		(en_i2, en_mult, iCLK_50, w_muxy_out, w_i2_out,
w_multx_out);
	mf_mult		multy		(en_i2, en_mult, iCLK_50, w_muxx_out, w_i2_out,
w_multy_out);

	//module tratd ( tratd_in, tratd_ctrl, tratd_out);
	tratd		tratadx		(w_multx_out, w_comp_out, w_tratadx_out);
	tratd		tratady		(w_multy_out, w_comp_out, w_tratady_out);
	tratd		tratadz		(w_t_angle_out, w_comp_out, w_tratadz_out);

//	//module mf_sub (clk_en, clock, dataa, datab, result);
//	mf_sub		subx		(en_addsub, w_div, w_muxx_out, w_tratadx_out,
w_subx_out);
//	mf_sub		subz		(en_addsub, w_div, w_muxz_out, w_tratadz_out,
w_subz_out);

	//module mf_sub ( aclr, clk_en, clock, dataa, datab, result);
	mf_sub		subx		(en_t_angle, en_addsub, iCLK_50, w_muxx_out,
w_tratadx_out, w_subx_out);
	mf_sub		subz		(en_t_angle, en_addsub, iCLK_50, w_muxz_out,
w_tratadz_out, w_subz_out);

	//module mf_add (clk_en, clock, dataa, datab, result);
	//module mf_add (aclr, clk_en, clock, dataa, datab, result);
	mf_add		addy		(en_t_angle, en_addsub, iCLK_50, w_muxy_out,
w_tratady_out, w_addy_out);

	//module mf_mult ( aclr, clk_en, clock, dataa, datab, result);
	mf_mult		multk		(cl_multk, en_multk, iCLK_50, w_regK_out, w_regx_out,
w_multk_out);

	//module my_ucode (estado, palavra);
	my_ucode	ucodigo		(estado, w_comandos);

// ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- -----

	always@(posedge iCLK_50) begin
			if (estado <  6'b110111) estado <= estado + 1'b1; //ultimo estado que ele faz algo
+ 1, contanto que ele não faça nada, ou seja comandos  = 00000...;
			if (estado == 6'b110111) cosomega <= w_multk_out; //mesma linha do if <
```

```
                        if (estado == 6'b101111) begin  //qd chegar na linha que o regdinamico é
atualizado + 1
                                if (contador_base < 4'b1101) begin
                                        estado <= 6'b011000;            // volto para a 1 linha que tem
ctmux=1
                                        contador_base <= contador_base + 1'b1;
                                end
                        end
        end


        assign {en_contador, en_reg, ct_mux, en_comp,
                        en_t_angle, en_i2, en_mult, en_addsub,cl_multk,en_multk} = w_comandos;


        always@(*) begin
                comp_out = w_comp_out;
                case(iSW[0])
                        0:begin
                                buffer = cosomega[15:0];
                                end
                        1:begin
                                buffer = cosomega[31:16];
                                end

                        default: buffer = cosomega[31:16];
                endcase
        end
        assign oLEDR[15:0] = buffer;

        assign oLEDG[8] = comp_out;

        assign oLEDG[7:0] = {en_contador, en_reg, ct_mux, en_comp,
                                        en_t_angle, en_i2, en_mult, en_addsub};

endmodule
```

```
module my_counter ( clock, en, cont);
        input [0:0] clock;
        input [0:0] en;
        output reg [4:0] cont;

        always@(posedge clock) begin
```

```
            if (en)
            cont <= cont + 5'b00001;
    end
endmodule
```

```
module registerSX ( registerSX_out);
    output reg [31:0] registerSX_out;

    always@(*) begin
        registerSX_out = 32'b00111111100000000000000000000000;//valor 1;
    end

endmodule
```

```
module registerSY ( registerSY_out);
    output reg [31:0] registerSY_out;

    always@(*) begin
        registerSY_out = 32'b00000000000000000000000000000000;//valor 0;
    end

endmodule
```

```
module registerSZ ( registerSZ_out);
    output reg [31:0] registerSZ_out;

    always@(*) begin
        registerSZ_out = 32'b00111111000001100000101010010000;// valor %pi/6;
        //registerSZ_out = 32'b00111111010010010000111111011010;// valor %pi/4;
    end

endmodule
```

```
module registerSK ( registerSK_out);
	output reg [31:0] registerSK_out;

	always@(*) begin
		registerSK_out = 32'b00111111000110110111100000000010;// valor 0.6073;
	end

endmodule
```

```
module register ( r_in, r_refresh, r_out);
	input [31:0] r_in;
	input [0:0] r_refresh;

	output reg [31:0] r_out;

	always@(posedge r_refresh) begin
		r_out <= r_in;
	end

endmodule
```

```
module my_mux2p1 ( in0, in1, ctrl, mux2p1_out);

	input [31:0] in0;
	input [31:0] in1;
	input [0:0] ctrl;

	output reg [31:0] mux2p1_out;

	always@(*) begin
		if (ctrl)
		mux2p1_out = in1;
		else
		mux2p1_out = in0;
	end
endmodule
```

```
module registerS0 ( registerS0_out);
	output reg [31:0] registerS0_out;

	always@(*) begin
		registerS0_out = 32'b00000000000000000000000000000000;
	end

endmodule
```

```
module tratd ( tratd_in, tratd_ctrl, tratd_out);

	input [31:0] tratd_in;
	input [0:0] tratd_ctrl;

	output reg [31:0] tratd_out;


	always@(*) begin
		if (tratd_ctrl)
			tratd_out = tratd_in;
		else begin
		tratd_out[30:0] = tratd_in[30:0];
		tratd_out[31] = 1'b1;
		end

	end

endmodule
```

```
module my_ROM_2i (address,data);
	input [4:0] address;
	output reg [31:0] data;

// ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- -----


	always@(*)begin
		case (address)
			5'b00000: data = 32'b00111111100000000000000000000000;
```

```
                5'b00001: data = 32'b00111111000000000000000000000000;
                5'b00010: data = 32'b00111110100000000000000000000000;
                5'b00011: data = 32'b00111110000000000000000000000000;
                5'b00100: data = 32'b00111101100000000000000000000000;
                5'b00101: data = 32'b00111101000000000000000000000000;
                5'b00110: data = 32'b00111100100000000000000000000000;
                5'b00111: data = 32'b00111100000000000000000000000000;
                5'b01000: data = 32'b00111011100000000000000000000000;
                5'b01001: data = 32'b00111011000000000000000000000000;
                5'b01010: data = 32'b00111010100000000000000000000000;
                5'b01011: data = 32'b00111010000000000000000000000000;
                5'b01100: data = 32'b00111001100000000000000000000000;
                5'b01101: data = 32'b00111001000000000000000000000000;
                5'b01110: data = 32'b00111000100000000000000000000000;
                5'b01111: data = 32'b00111000000000000000000000000000;
                5'b10000: data = 32'b00110111100000000000000000000000;
                5'b10001: data = 32'b00110111000000000000000000000000;
                5'b10010: data = 32'b00110110100000000000000000000000;
                5'b10011: data = 32'b00110110000000000000000000000000;
                5'b10100: data = 32'b00110101100000000000000000000000;
                5'b10101: data = 32'b00110101000000000000000000000000;
                5'b10110: data = 32'b00110100100000000000000000000000;
                 default: data = 32'b00000000000000000000000000000000;
        endcase
    end
endmodule
```

```
module my_ROM_ang (address,data);
    input [4:0] address;
    output reg [31:0] data;

// ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- -----


    always@(*)begin
        case (address)
            5'b00000: data = 32'b00111111010010010000111111011010;
            5'b00001: data = 32'b00111110111011010110001100111000;
            5'b00010: data = 32'b00111110011110101101101110101000;
            5'b00011: data = 32'b00111101111111101010110111010000;
            5'b00100: data = 32'b00111101011111111010101011000000;
            5'b00101: data = 32'b00111100111111111110101010000000;
            5'b00110: data = 32'b00111100011111111111101010000000;
```

```
                5'b00111: data = 32'b00111011111111111111111000000000;
                5'b01000: data = 32'b00111011011111111111111000000000;
                5'b01001: data = 32'b00111010111111111111110000000000;
                5'b01010: data = 32'b00111010011111111111110000000000;
                5'b01011: data = 32'b00111001111111111111000000000000;
                5'b01100: data = 32'b00111001011111111110000000000000;
                5'b01101: data = 32'b00111000111111111100000000000000;
                5'b01110: data = 32'b00111000011111111000000000000000;
                5'b01111: data = 32'b00110111111111110000000000000000;
                5'b10000: data = 32'b00110111011111100000000000000000;
                5'b10001: data = 32'b00110110111111000000000000000000;
                5'b10010: data = 32'b00110110011110000000000000000000;
                5'b10011: data = 32'b00110101111100000000000000000000;
                5'b10100: data = 32'b00110101011000000000000000000000;
                5'b10101: data = 32'b00110100110000000000000000000000;
                 default: data = 32'b00000000000000000000000000000000;
            endcase
        end
endmodule
```

```
module my_ucode ( estado, palavra);
        input [5:0] estado;
        output reg [9:0] palavra;

// ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- -----


        always@(*)begin
            case (estado)
                6'b000001: palavra = 10'b0000000000;
                6'b000010: palavra = 10'b0000011000;
                6'b000011: palavra = 10'b0000001000;
                6'b000100: palavra = 10'b0000001000;
                6'b000101: palavra = 10'b0000001000;
                6'b000110: palavra = 10'b0000001000;
                6'b000111: palavra = 10'b0000001000;
                6'b001000: palavra = 10'b0000001000;
                6'b001001: palavra = 10'b0001001000;
                6'b001010: palavra = 10'b0001000000;
                6'b001011: palavra = 10'b0001000000;
                6'b001100: palavra = 10'b0001000000;
                6'b001101: palavra = 10'b0000100100;
                6'b001110: palavra = 10'b0000000100;
```

```
                6'b001111: palavra = 10'b0000000100;
                6'b010000: palavra = 10'b0000000100;
                6'b010001: palavra = 10'b0000000100;
                6'b010010: palavra = 10'b0000000100;
                6'b010011: palavra = 10'b0000000100;
                6'b010100: palavra = 10'b0000000100;
                6'b010101: palavra = 10'b0000000100;
                6'b010110: palavra = 10'b0000000100;
                6'b010111: palavra = 10'b0100000000;
                6'b011000: palavra = 10'b1010000000;
                6'b011001: palavra = 10'b0010011000;
                6'b011010: palavra = 10'b0010001000;
                6'b011011: palavra = 10'b0010001000;
                6'b011100: palavra = 10'b0010001000;
                6'b011101: palavra = 10'b0010001000;
                6'b011110: palavra = 10'b0010001000;
                6'b011111: palavra = 10'b0010001000;
                6'b100000: palavra = 10'b0011001000;
                6'b100001: palavra = 10'b0011000000;
                6'b100010: palavra = 10'b0011000000;
                6'b100011: palavra = 10'b0011000000;
                6'b100100: palavra = 10'b0010100100;
                6'b100101: palavra = 10'b0010000100;
                6'b100110: palavra = 10'b0010000100;
                6'b100111: palavra = 10'b0010000100;
                6'b101000: palavra = 10'b0010000100;
                6'b101001: palavra = 10'b0010000100;
                6'b101010: palavra = 10'b0010000100;
                6'b101011: palavra = 10'b0010000100;
                6'b101100: palavra = 10'b0010000100;
                6'b101101: palavra = 10'b0010000100;
                6'b101110: palavra = 10'b0110000000;
                6'b101111: palavra = 10'b0000000011;
                6'b110000: palavra = 10'b0000000001;
                6'b110001: palavra = 10'b0000000001;
                6'b110010: palavra = 10'b0000000001;
                6'b110011: palavra = 10'b0000000001;
                6'b110100: palavra = 10'b0000000001;
                6'b110101: palavra = 10'b0000000001;
                6'b110110: palavra = 10'b0000000001;
                6'b110111: palavra = 10'b0000000000;
                default: palavra = 10'b00000000;
            endcase
        end
    endmodule
```