Dênis Araújo da Silva

Marcos Aurélio Freitas de Almeida Costa

Desenvolvimento de um Microprocessador 8086 RISC

Itajubá - MG

30 de Maio de 2014

Dênis Araújo da Silva

Marcos Aurélio Freitas de Almeida Costa

Desenvolvimento de um Microprocessador 8086 RISC

Documento apresentando as normas gerais para o desenvolvimento e a redação do trabalho de diploma do curso de Engenharia da Computação da Universidade Federal de Itajubá.

Universidade Federal de Itajubá - UNIFEI
Instituto de Engenharia de Sistemas e Tecnologias da Informação
Engenharia da Computação

Itajubá - MG

30 de Maio de 2014

Sumário

Lista de Figuras

Lista de Tabelas

Dedicatória

Eu, Dênis Araújo da Silva, gostaria particularmente de dedicar este Trabalho Final de Graduação, primeiramente aos meus pais, Severino Belo da Silva e Cleide Aparecida de Araújo Silva por terem me proporciado condições e apoio, para que eu podesse estar aqui realizando um dos meus sonhos e agradeço muito por terem investido em minha educação desde pequeno, gostaria também de dedicar à Chayene Aguiar Rocha, por ter me ajudado tanto a superar algumas perdas durante este longo período distante e principalmente por ter se preocupado e cuidado de mim quando passei por alguns problemas de saúde e também à minha irmã, Natalia Araújo da Silva e às pessoas que infelizmente me distanciei e perdi ao longo do caminho que ajudaram a me criar e por fim, à Deus.

Eu, Marcos Aurélio Freitas de Almeida Costa, agradeço a Deus por ter sempre me amparado e iluminado meu caminho, a meus pais, Marcos Aurélio Almeida Costa e Maria da Conceição Freitas de Almeida Costa, por terem sempre me apoiado e aconselhado durante minha trajetória, a meus irmãos, Anna Karolina Freitas Almeida Costa e Luiz Eduardo Freitas Almeida Costa, pelo apoio e a todos os amigos que fiz durante este tempo.

Agradecimentos

Ao nosso orientador, Profº.Drº. Maurílio Pereira Coutinho, por ter acatado a ideia do projeto, pela sua disponibilidade, incentivo, aprendizado nas áreas de computadores digitais e microprocessadores, e pelo auxílio e direcionamento durante todo o período de realização deste trabalho.

Ao nosso co-orientador Profº.Drº. Robson Luiz Moreno, por sua disponibilidade, incentivo, aprendizado na área de Eletrônica Digital e VHDL, ter oferecido sempre material para o desenvolvimento do trabalho, facilitando o e apoiando durante todo o processo de realização do mesmo.

À Universidade Federal de Itajubá e ao Instituto de Engenharia de Sistemas e Tecnologia da Informação.

E por fim e não menos importante aos amigos que fizemos durante esta longa jornada.

Resumo

Este documento descreve o Trabalho Final de Graduação do curso de Engenharia da Computação da Universidade Federal de Itajubá. O projeto visa desenvolver um microprocessador iAPX86 de arquitetura RISC implementado em linguagem VHDL.

Originalmente o iAPX86 possui 255 operações em seu conjunto de instruções. Para adaptá-lo à filosofia RISC, foram feitas modificações no formato das instruções e escolheu-se as de objetivo básico para integrar o processador RISC aqui desenvolvido.

Assim, chegou-se em instruções de tamanho fixo, 4 bytes cada uma, e nas seguintes nove operações: ADD, ADC, SUB, SBB, AND, OR, XOR, CMP e MOV. Onde a instrução MOV é a única capaz de acessar à memória, seguindo assim a filosofia RISC de máquinas LOAD/STORE.

1 Introdução

O microprocessador, ou simplesmente CPU, é uma peça fundamental dos dispositivos eletrônicos atuais. Ela está presente em computadores pessoais, tablets, smartphones e eletrodomésticos. É responsável pela execução de operações aritméticas e lógicas requisitadas pelos programas.

O projeto de um microprocessador envolve circuitos extensos e complexos, é neste ponto que entra a lógica programável. Este recurso permite escrever um código que implemente a funcionalidade de um circuito eletrônico. VHDL é uma das linguagens que permite a escrita deste código.

2 Objetivo

O objetivo deste trabalho é adaptar o microprocessador 8086 para um dispositivo de arquitetura RISC Load/Store. O microprocessador será implementado em linguagem VHDL.

3 Fundamentação Teórica

3.1 Microprocessadores

3.1.1 Definição

Segundo (FERREIRA, 1981), um microprocessador é um circuito eletrônico composto de inúmeros transistores microscópicos num único circuito integrado. Os transistores que compõem o microprocessador são arranjados para formar diferentes circuitos dentro do chip. Entre estes circuitos pode-se destacar registradores, decodificadores, contadores, etc.

O coração de um microcomputador é sua unidade de processamento (MPU). A MPU de um microcomputador é implementada com um dispositivo VLSI (*Very Large Scale Integration*) conhecido como microprocessador, ou somente processador, sendo mais direto. Um microprocessador é uma unidade de processamento de propósito geral construído em um único circuito integrado (CI), (SINGH, 1947).

Como visto acima, o microprocessador é o coração de um sistema microprocessado, na Figura 2, defini-se as quatro partes básicas de um sistema microprocessado, que incluem um microprocessador, memória e entrada/saída que são interligados por um sistema de *buses*, que será explicado mais a frente. Um *bus* é um conjunto de fios que transmite informação entre dois ou mais dispositivos(NEWELL, 1989).

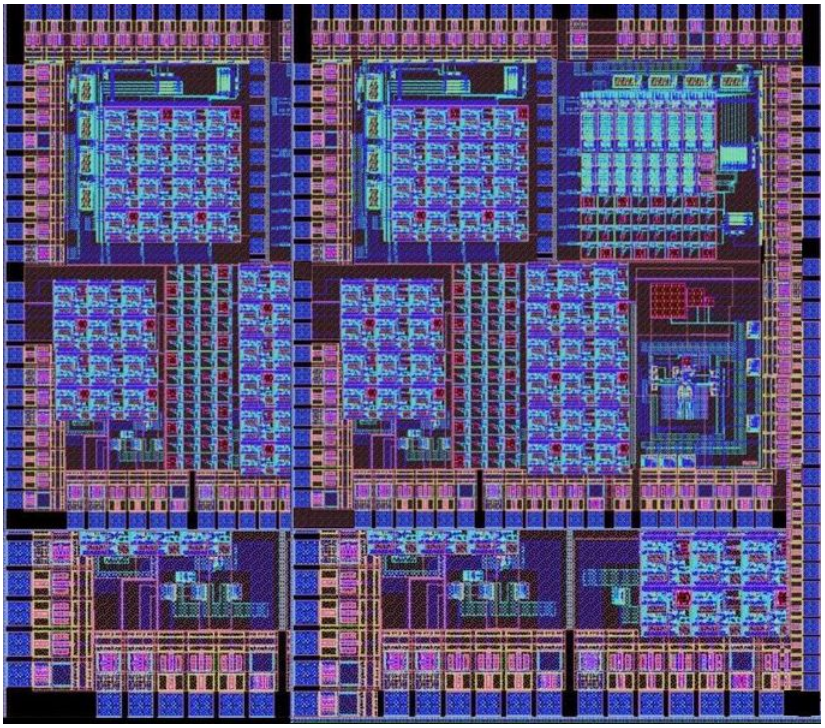

Figura 1: Microcircuito produzido pelo processo fotográfico de multicamadas. Microprocessador com frequência de 4.8GHz, utilizado para o processamento de imagens do Gyroscan 6,2 Tesla. (ANGELOLEITHOLD, 2004)

3.1.2 Funcionamento

Os microprocessadores funcionam a partir de um relógio interno, feito de quartz que quando sujeito a uma corrente elétrica, emite pulsos, chamados de "top". Tais pulsos fazem com que o microprocessador execute uma ação, ou seja, uma instrução, seja a mesma executada de forma parcial ou total. Estes pulsos também definem a potência do microprocessador, sendo esta potência definida como o número de instruções executadas por segundo e tem como unidade utilizada o MIPS (Milhões de Instruções Por Segundo) (MICROPROCESSADORES, 2013).

Existem dispositivos de entrada e saída que permitem a importação de dados para armazenamento ou processamento, a exportação dos resultados e acessos aos dados armazenados. Outros sinais importantes para o funcionamento do micro-

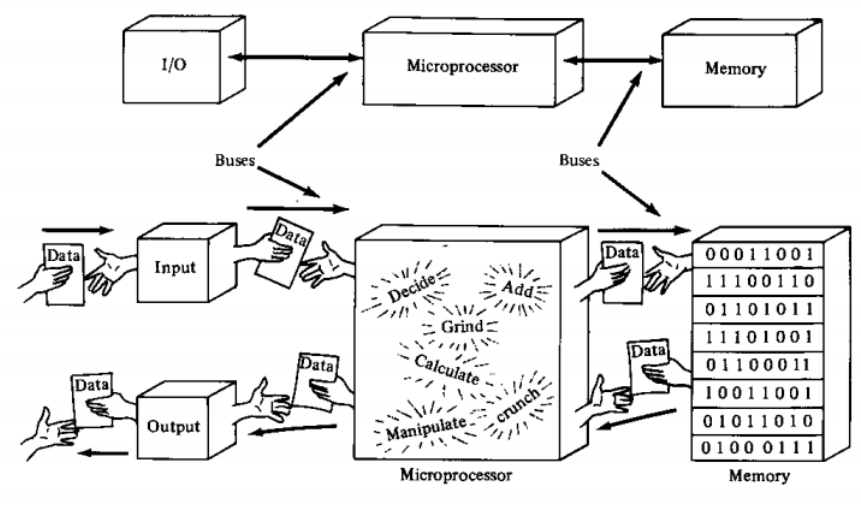

Figura 2: Diagrama de blocos de um sistema microprocessado. (NEWELL, 1989)

processador é o sinal de *reset*, que faz com que a CPU volte a um estado inicial que é definido e conhecido. Voltando a este estado o microprocessador pode começar a executar programas. O sinal de interrupção faz com que o microprocessador pare sua execução e comece a executar uma rotina pré-definida (MICROPROCESSADORES, 2013)

3.1.3 Programa de Computador

A Figura 3 é uma representação visual de um programa de computador, onde ter-se o código não é o suficiente. Para realizar a tarefa especificada pelo programa, o computador (microprocessador) necessita ler as instruções do programa, interpretá-las e executá-las. (NEWELL, 1989).

De acordo a (NEWELL, 1989) a maneira que um computador executa um programa é cíclica e segue a seguinte ordem:

1. Leitura (*Fetch*) de uma instrução
   Onde o computador lê uma instrução e a copia da memória para o seu cérebro (*Microprocessador*).

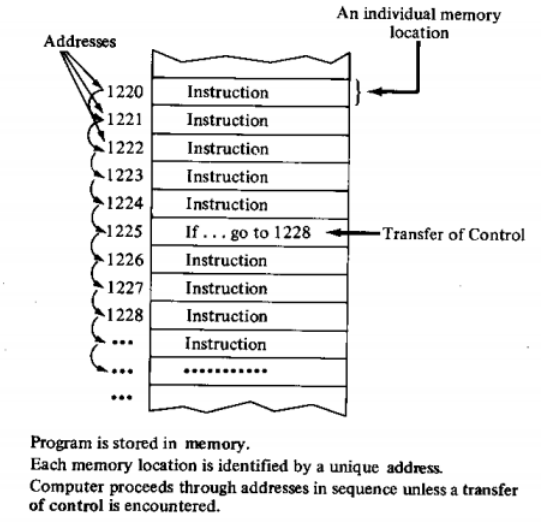

Figura 3: Organização de um programa de computador (NEWELL, 1989)

2. Interpretação (*Decode*) da instrução
   Cada número que representa uma instrução dentro do programa, possui um significado para o computador, em termos da ação que deve ser realizada.

3. Execução (*Execute*) da instrução

Para a realização deste ciclo, o microprocessador conta com uma série de circuitos internos com funcionalidades específicas, que serão descritos à seguir.

3.1.4 Registradores

Os registradores são utilizados para salvar informação binária durante o tempo de execução de um programa. Cada registro possui uma função específica associada a ele (ENGINEERING; MANAGEMENT, 2013):

O **acumulador** é um registro primário associado a *ALU* (Unidade Lógica Aritmética) e operações de entrada/saída. O **registro de instrução** guarda o código binário da instrução que está sendo executada. O **contador de programa** contém o endereço de memória da próxima instrução que deve ser tomada.

Todos estes registradores podem ser visualizados na Figura 4.

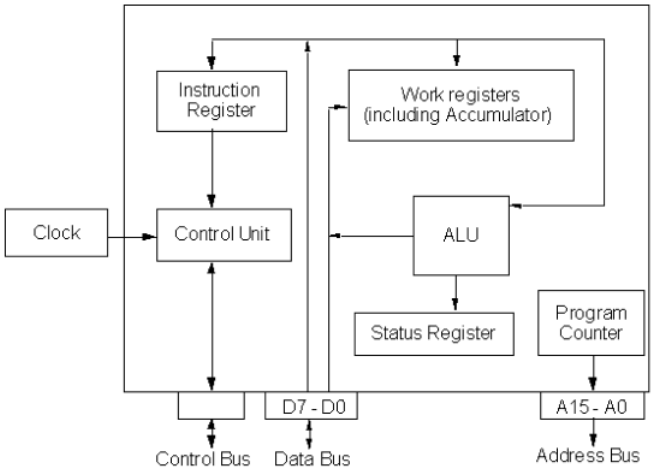

Figura 4: Organização interna de um microprocessador hipotético (ENGINEERING; MANAGEMENT, 2013)

3.1.5 Unidade Lógica Aritmética

A Unidade Lógica Aritmética, ou *Arithmetic Logic Unit* (ALU) em inglês, é o maior componente da unidade central de processamento de um sistema microprocessado. A unidade realiza todos os processos relacionados a operações aritméticas e lógicas que necessitam ser feitas nas instruções. Em alguns microprocessadores a ALU é divida em unidade aritmética (UA) e unidade lógica (UL). Uma ULA pode ser desenvolvida por engenheiros para calcular qualquer operação. Assim que as operações começam a ficar mais complexas, a ULA fica mais cara, ocupa mais espaço e dissipa mais calor. Por isto, engenheiros fazem a ULA poderosa o suficiente, para garantir que a Unidade de Processamento seja também poderosa e rápida, porém não tão complexa, que a torne proibitiva em termos de custo entre outras desvantagens (TECHOPEDIA, 2013).

ULAs normalmente realizam as seguintes operações:

- **Operações Lógicas**: Essas incluem AND, OR, NOT, XOR, NOR, NAND, etc.
- **Operações de Rotacionamento de Bits**: Pertence ao rotacionamento da posição dos bits por um certo número de vezes para direita ou esquerda.
- **Operações Aritméticas**: Refere-se, normalmente, a adição e subtração. Multiplicação e divisão as vezes são implementadas. Porém, estas são operações custosas. A adição pode ser utilizada como substituta para a multiplicação e a subtração para a divisão.

3.1.6 Unidade de Controle

A unidade de controle é composta por um controlador de sequência e um decodificador de instrução. Durante a execução, a unidade de controle, ajusta o conteúdo do contador de programa para ser posicionado nas linhas de endereçamento. Essas linhas indicam o endereço da posição de memória, que contém o código da próxima instrução a ser executada. Em seguida, a unidade de controle insere no registrador de instrução o código de instrução da posição de memória. O decodificador de instrução é habilitado e a unidade de controle ativa as linhas de controle necessárias, buscando dos resultados desejados (ENGINEERING; MANAGEMENT, 2013).

3.1.6.1 Sinais de Controle

Os sinais de controle são sinais elétricos que orquestram as diversas unidades do processador, que participam na execução de uma instrução. Os sinais de controle são distribuídos devido a um elemento chamado sequenciador. O sinal *Read/Write*, em português Leitura/Escrita, diz para a memória ou outros dispositivos que o processador quer ler ou escrever uma informação (MICROPROCESSADORES, 2013).

3.1.7 Sistema de Barramentos

No diagrama simplificado da figura 5 todos os módulos lógicos se comunicam com a Unidade Central de Processamento. Na prática, muitos modelos de interconexão podem ser usados, geralmente através de barramentos. Lembre-se de que um barramento é um meio de transmissão de informações ou sinais, distinguidos por suas funções. No caso dos sistemas baseados em microprocessador, ao menos três barramentos são fornecidos (FILHO, 2013):

- **Barramento de Dados**: Transmite dados entre as unidades. Portanto, um microprocessador de 8 bits requer um barramento de dados de 8 linhas para transmitir dados de 8 bits em paralelo. Semelhantemente, um microprocessador de 64 bits necessita de um barramento de dados de 64 linhas para transmitir dados de 64 bits em paralelo. Se o barramento de dados para um microprocessador de 64 bits fosse formado por 8 linhas, seriam necessárias oito transmissões sucessivas, tornando mais lento o sistema. O Barramento de Dados é bi-direcional, isto é, pode transmitir em ambas as direções.

- **Barramento de Endereço**: É usado para selecionar a origem ou destino de sinais transmitidos em um dos outros barramentos ou numa de suas linhas, conduzindo endereços. Uma função típica do Barramento de Endereço é selecionar um registrador em um dos dispositivos do sistema, que é usado como a fonte ou o destino do dado. O Barramento de Endereço do nosso computador padrão, tem 16 linhas e pode endereçar 2^{16} (64 K) dispositivos (1K = 1024, ou 2^{10} , no jargão de computação).

- **Barramento de Controle**: Sincroniza as atividades do sistema, conduzindo o status e a informação de controle de/para o Microprocessador. Para um Barramento de Controle ser formado, ao menos 10 (geralmente são mais) linhas de controle são necessárias, linhas para controle dos registradores, da unidade lógica aritmética, dos registros de flags, entre outros.

De acordo com (FILHO, 2013), os barramentos são implementados como linhas de comunicação reais. Eles podem ser posicionados como parte do circuito no próprio Chip (Barramentos internos) ou podem servir de comunicação externa entre os Chips (Barramentos externos). Os barramentos externos podem ser expandidos para facilitar a conexão de dispositivos especiais. Um projeto eficiente de barramentos é crucial para a velocidade do sistema.

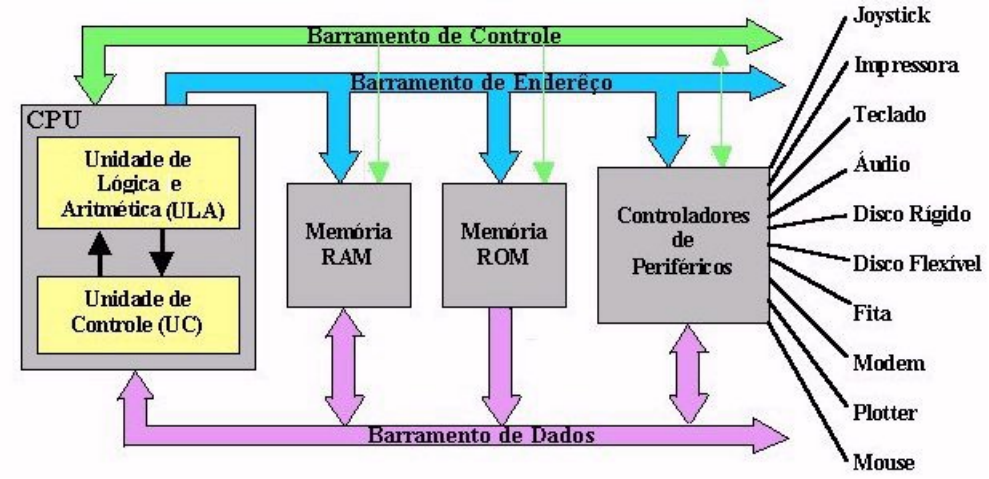

Figura 5: Diagrama simplificado de um Microcomputador (FILHO, 2013)

3.1.8 Dispositivos de Entrada/Saída

Os dispositivos de entrada/saída (E/S) ou *input/output (I/O)*, também denominados periféricos, permitem a interação do processador com o homem, possibilitando a entrada e/ou saída de dados. Porém, é necessário o módulo mais a direita da figura 5, que são os controladores de periféricos. Tais controladores possuem a tarefa de combinar as velocidades entre os dispositivos, pois a maioria dos periféricos são consideravelmente mais lentos que a unidade de processamento na conversão de dados de um formato em outro (FILHO, 2013). Alguns exemplos de periféricos de entrada: teclado, mouse, scanner, etc. Dispositivos de saída: monitor, impressora, etc.

3.1.9 Arquiteturas

De acordo com (PATTERSON, 2005), uma das mais importantes abstrações é a interface entre o hardware e o software de baixo nível. Por causa de sua importância, é dado uma nomenclatura especial: **arquitetura do conjunto de instruções** (ISA), ou simplesmente arquitetura de uma máquina. O conjunto de instruções, inclui qualquer coisa que programadores necessitam para saber como programar em linguagem de máquina corretamente, incluindo instruções, dispositivos E/S, entre outros. Tipicamente o sistema operacional irá encapsular os detalhes da realização da E/S, alocação de memória, e outras funcionalidades de baixo nível do sistema. Portanto, programadores não precisam se preocupar com estes detalhes. Dois tipos de conjuntos de instruções existentes serão explicados a diante.

3.1.9.1 CISC - Complex Instruction Set Computer

CISC é uma arquitetura de processador, que teve como princípio o uso eficiente de memória e a facilidade de programar. Cada instrução desse processador tem várias operações em seu interior ajudando o programador a implementar programas. A maioria dos projetos de microprocessadores comuns - incluindo o Intel (R) 80x86 e séries Motorola 68K - também seguem a filosofia CISC (CISC, 2013).

Os primeiros processadores utilizados para decodificar e executar instruções, principalmente para trabalhos simples, com poucos registros, funcionaram. Porém não para sistemas complexos. Assim, seus criadores construíram uma lógica simples para controlar os caminhos de dados entre os vários elementos do processador, e usou um conjunto simplificado de instruções de microcódigo para controlar a lógica do caminho de dados.

A microprogramação é uma representação simbólica do controle em forma de instruções, chamadas microinstruções, que são executadas em uma micromáquina simples (PATTERSON, 2005). Podemos ver na figura 6 o exemplo de um micro-

código.

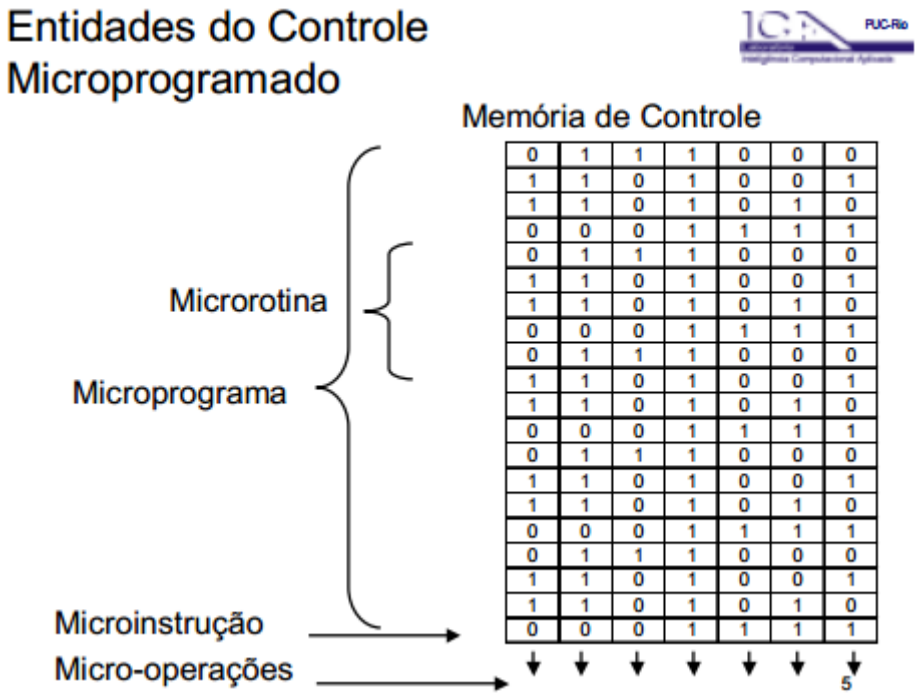

Figura 6: Entidades do Controle Microprogramado (PUC-RIO, 2013)

3.1.9.2 RISC - Reduced Instruction Set Computer

Na década de 1980, ocorreram as mudanças para uma nova arquitetura, o modelo RISC (Reduced Instruction Set Computer). As melhorias nas linguagens de programação, tecnologia de compiladores e custo de memória significaram que menos programação estava sendo feita no nível do assembly. Desta forma os conjuntos de instruções poderiam ser medidos pela forma como os compiladores os usavam, ao contrário de como os programadores em assembly os usavam. Praticamente todos os novos conjuntos de instruções desde 1982, seguiram essa filosofia RISC de tamanhos de instrução fixos, conjuntos de instrução load/store, modos de endereçamento limitados e operações limitadas. ARM, Hitachi SH, IBM PowerPC, MIPS e Sun SPARC são todos exemplos de arquiteturas RISCs (PATTERSON, 2005).

3.1.9.3 Comparação entre RISC e CISC

Como visto anteriormente, a arquitetura CISC apresenta instruções complexas executadas em vários ciclos de relógio, enquanto a arquitetura RISC possui somente instruções que são executadas em apenas um ciclo. No quesito de acesso a memória, o conjunto complexo possui vários tipos de modos de endereçamento de memória, facilitando o trabalho do programador. Já os microprocessadores RISC são considerados máquinas *load/store*, o que é possível devido uma grande quantidade de registradores dos mais variados tipos. A clara vantagem da arquitetura RISC é em questão de velocidade. Por possuir um conjunto de instruções com todas instruções com formato fixo, ocorre um uso intenso de *pipeline*. No desenvolvimento de um microprocessador CISC, a complexidade do sistema se encontra no microprograma, como visto um exemplo na figura 6, e na arquitetura RISC a complexidade se encontra no compilador. Ambas arquiteturas são muito bem aceitas no mercado e cada uma possui suas vantagens e desvantagens para serem aplicados em diversos tipos de projetos.

3.1.10 Memória Cache

De acordo com (TARNOFF, 2011), a memória cache consegue realizar a ponte entre a diferença de velocidade entre o processador e a memória. A cache é um pequeno espaço de alta velocidade que se situa entre o processador e a memória na hierarquia de memórias.

Cache, foi o nome escolhido para representar o nível na hierarquia de memória entre o processador e a memória do primeiro computador comercial a ter este nível extra, como pode ser visto na Figura 7 item (b). A razão da cache ($SRAM$) ser menor é devido a maiores decodificadores de endereço, pois são mais lentos do que menores decodificadores de endereço. Quanto maior a memória é, mais complexo é seu decodificador de endereço, e mais tempo leva para identificar o valor da posição de memória do endereço desejado (TARNOFF, 2011).

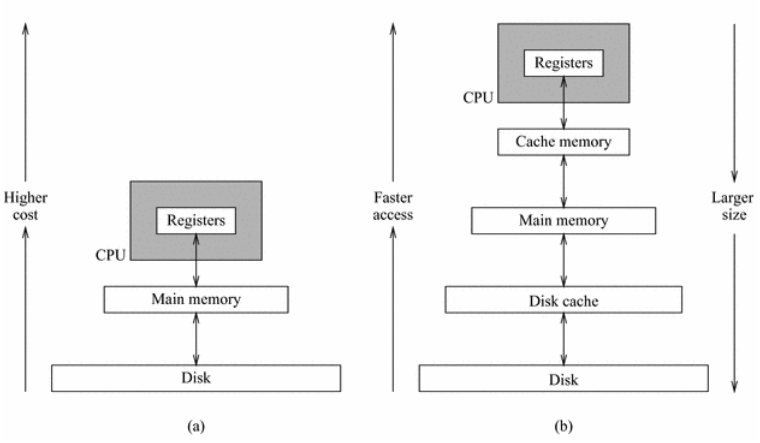

Figura 7: Hierarquia de Memórias (TARNOFF, 2011). a) Processador sem memória cache , b) Processador com memória cache

É possível utilizar este conceito e dar um passo a diante introduzindo uma *SRAM* menor entre o cache o e processador, dentro do próprio envólucro do processador, criando dois níveis de memória cache L1 e L2, como se nota na Figura 8 (TARNOFF, 2011).

Figura 8: Dois níveis de memória cache L1 e L2 (TARNOFF, 2011)

3.1.11 Pipeline

3.1.11.1 Definição

De acordo com (PATTERSON, 2005), *Pipelining* é uma técnica de implementação no qual multiplas instruções são sobrepostas durante a execução, como visto na Figura 9. Hoje em dia, *pipelining* é a chave para fazer processadores rápidos (PATTERSON, 2005).

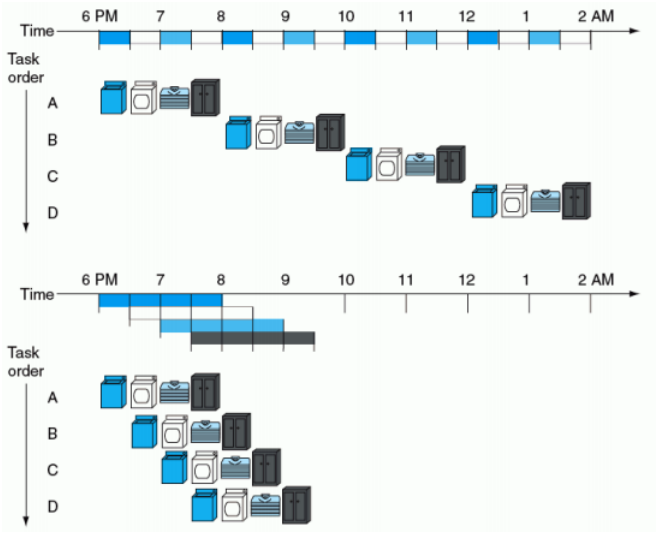

Figura 9: A analogia de uma lavanderia com o *pipeline* (PATTERSON, 2005)

Como podemos ver, a utilização de *pipeline* torna a execução muito mais rápida do que se as tarefas fossem executadas sequencialmente. Como exemplo utilizaremos o microprocessador MIPS, que possui 5 estágios de execução de uma instrução: Decodificação da Instrução (*Instruction Fetch)*, Leitura dos Registros (*Reg*),Operação de ULA (*ALU*) , Acesso ao dado e Escrita no Registro. Na figura 10 podemos ver quantitativamente a diferença do processo com utilização de *pipeline*.

3.1.11.2 Desenvolvimento de um conjunto de instrução para o Pipeline

Primeiramente, todas as instruções do MIPS possuem o mesmo comprimento, esta restrição faz com que fique mais fácil decodificar as instruções no primeiro e no segundo estágio do *pipeline*. Em um conjunto de instruções, como o do IA-32, instruções variam de 1 até 17 bytes, o *pipelining* é consideravelmente mais complicado (PATTERSON, 2005). Atualmente a arquitetura do IA-32 transforma as instruções em microinstruções, sendo essas utilizadas para a realização do *pipeline* com uma arquitetura *CISC*.

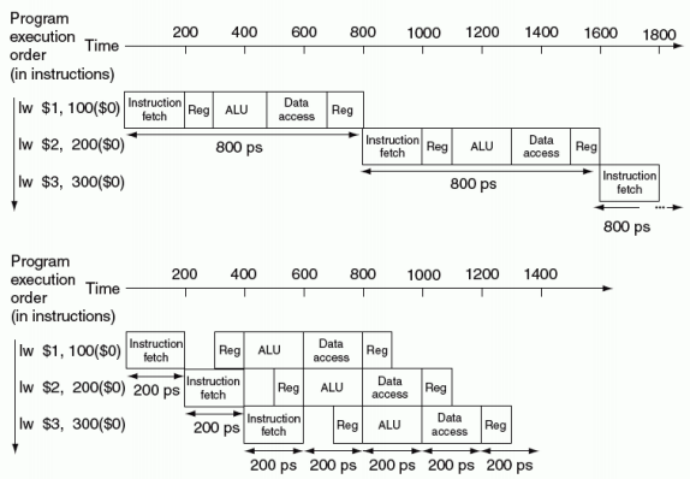

Figura 10: Comparação quantitativa da utilização de *Pipeline* utilizando a instrução *Load Word* do microprocessador MIPS. (PATTERSON, 2005)

3.1.11.3 Problemas do Pipeline

O **primeiro** tipo de problema do *pipeline* é o problema estrutural, que significa que o hardware não suporta a combinação de instruções que desejamos que sejam executadas em um único ciclo de relógio. Ao caso de termos somente uma única memória, no segundo e quarto passo, são necessários acessos a memória que não podem ser executadas de uma única vez (PATTERSON, 2005). O **segundo** tipo de problema é devido ao acesso aos dados, quando uma etapa necessita de utilizar o dado que uma outra etapa está modificando o *pipeline* fica travado, pois temos uma indefinição quanto a resultado deste dado, por exemplo, no trecho de código á seguir do MIPS, na figura 11.

```
add    $s0, $t0, $t1
sub    $t2, $s0, $t3
```

Figura 11: Trecho de código que exemplifica um problema do *pipeline* (PATTERSON, 2005).

Quando este tipo de problema ocorre, um fato chamado *bubble* aparece no meio do *pipeline*, como se fosse uma linha vazia entre dois processos do *pipeline*,

podendo ser notado na Figura 12.

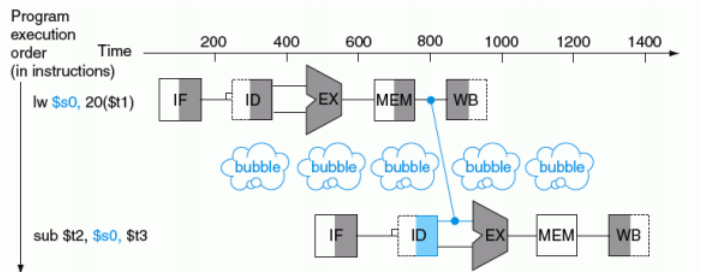

Figura 12: Fenômeno do tipo *bubble* no *pipeline* semelhante ao problema da figura 11 (PATTERSON, 2005).

Para resolver este tipo de problema, o código pode ser reescrito de uma forma que evite a dependência das informações, essa correção pode ser realizada tanto pelo compilador como pelo programador.

O terceiro tipo de problema, é chamado de problema de controle, também chamado de problema de *branch*, surge da necessidade de tomar uma decisão baseada nos resultados de uma instrução enquanto outras estão em execução (PATTERSON, 2005).

As instruções de *branch* são instruções de desvios condicionais no código, portanto a execução da próxima instrução depende se o *branch* causará desvio ou não. Caso o desvio ocorra, o fenômeno de *bubble* ocorre novamente, pois o *pipeline* necessita ficar um tempo parado esperando a tomada de decisão, semelhante a figura 13.

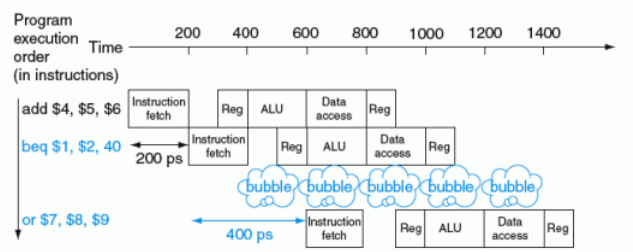

Figura 13: Fenômeno do tipo *bubble* no *pipeline* quando ocorre o problema de *branch* (PATTERSON, 2005).

Para solucionar este problema, desenvolveu-se uma estrutura dentro do próprio microprocessador, chamada *branch predictor*. O *branch predictor* é um circuito digital que tenta adivinhar qual vai ser o caminho que o branch irá seguir, para continuar preenchendo o *pipeline*. Hoje em dia, tal estrutura desempenha um papel fundamental para o desenvolvimento de processadores de alta performance.

3.1.12 Processadores Multi-Core e Hyper-Threading

De acordo com (BINSTOCK, 2013), basicamente, multi-core é um design ao qual um único processador físico, contém o núcleo lógico de mais de um processador, como pode ser visto na Figura 14. O objetivo deste design é habilitar o sistema a executar mais tarefas simultaneamente e desse modo alcançar um maior desempenho do sistema.

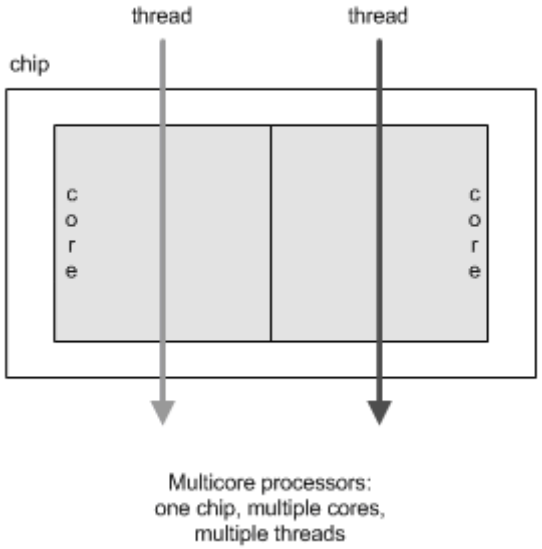

Figura 14: Processador com dois núcleos dentro de um único processador (BINSTOCK, 2013).

Programas são executados, à partir de threads, essas threads são sequências de instruções relacionadas. Nos primórdios do PC, a maioria dos programas consistia de uma única thread, o sistema operacional naquela época era capaz de executar

somente um programa por vez, tendo como resultado uma sensação dolorosa que seu PC congelava enquanto imprimia um documento ou uma folha de trabalho, o sistema era incapaz de realizar duas tarefas simultaneamente. Inovações no sistema operacional introduziram os sistemas multitarefa, no qual um programa pode ser brevemente suspendido enquanto executa outro, de uma maneira que o usuário não perceba. Realizando esta troca rapidamente, o sistema tem a aparência de estar executando os programas simultaneamente. Contudo o processador estava, de fato, executando uma única thread.

No início dos anos 2000, o design de processadores ganhou recursos adicionais, como uma lógica dedicada para operações com ponto flutuante, para suportar a execução de múltiplas instruções em paralelo. A Intel®definiu o melhor uso desses recursos empregando-as para executar duas threads simultaneamente no mesmo núcleo de processamento, figura 15. A Intel®nomeou este procedimento simultâneo como *Hyper-Threading Technology*®e lançou-a nos processadores **Intel Xeon ®** em 2003. De acordo com medidores da Intel®, aplicações que eram escritas utilizando múltiplas threads realizaram suas tarefas 30% mais rápido do que se fossem executadas utilizando a tecnologia **HT**. Para induzir o sistema operacional a reconhecer um processador como duas possibilidades de execução de *pipeline*, *chips* foram feitos para aparentar serem dois processadores lógicos.

3.2 Microprocessador 8086/8088

3.2.1 História

Em 1968 a empresa Intel foi fundada por Robert N. Noyce, Gordon E. Moore e Andrew Grove.

Em 15 de novembro de 1971 nascia o processador 4004 de apenas 4 bits e grande capacidade para realizar operações aritméticas. Esse microprocessador possuía 2.300 transistores para processar 0,06 milhões de instruções (60.000) por segundo

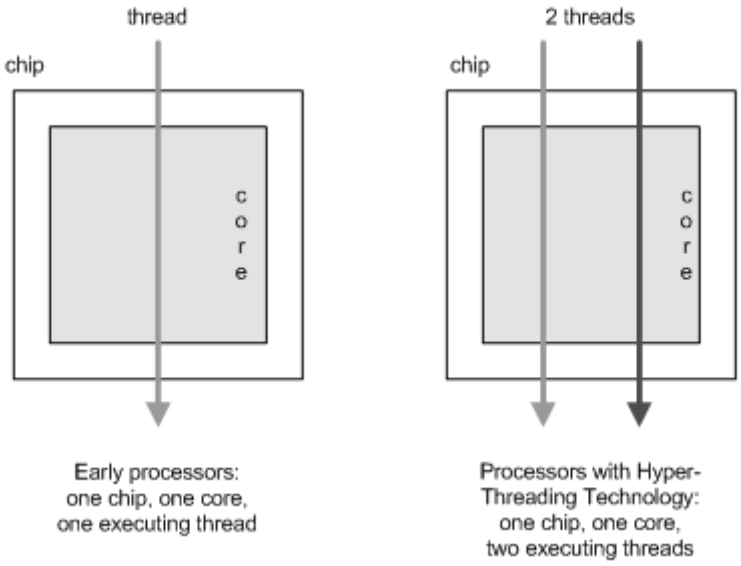

Figura 15: Exemplo de um processador com a tecnologia *Hyper-Threading* (BINSTOCK, 2013).

e não tinha o tamanho de um selo de carta. Para se ter uma idéia, o ENIAC, primeiro computador de que se tem notícia , construído em 1946 para fins bélicos, ocupava sozinho 1.000 metros quadrados e fazia o mesmo que o 4004.

O 4004 foi usado apenas para cálculos poucos complexos (4 operações). Ele era um pouco mais lento que o Eniac II mais tinha a vantagem de possuir a metade do tamanho, esquentar menos e consumir menos energia.

Em 1972 surgiu o 8008, primeiro processador de 8 bits, com capacidade de memória de 16 Kbytes (16.384 bytes), enquanto o 4004 possuía apenas 640 bytes.

Em 1974 é lançado o 8080, com desempenho seis vezes maior que o anterior. Com um relógio de 2 MHz, rodava um programa da Microsoft chamado Basic. Além de 16KB de memória ROM onde ficava o sistema, possuía 4KB de memória RAM. Seus controles eram através de botões e possuía drive de disquete 8”com capacidade de 250 KB.

O 8086 foi o primeiro processador feito pela Intel para ser usado com os PC’s.

Ele contava com um barramento de dados interno e externo de 16 bits. E foi este o motivo de não ter sido o processador mais utilizado. Inicialmente ele foi distribuído em versões de 4,77 MHz. Posteriormente vieram versões turbinadas de 8 e 10 MHz.

A história do 8086 é bem simples. Quando ele foi lançado, a maioria dos dispositivos e circuitos disponíveis eram de 8 bits. Era muito caro adaptar todo o resto do computador por causa do processador. E foi isso que acabou com o 8086. Para adaptar-se a este mercado a Intel lançou o 8088, com barramento externo mais lento, de 8 bits. Deixando a diferença de barramento externo, ambos eram idênticos.

Quando este chip, o 8086, veio a ser utilizado já era tarde demais. Ele chegou até a fazer parte de uns poucos clones do IBM PC e posteriormente em dois modelos do IBM PS/2 e de um computador Compaq. Mas sua destruição veio com um processador mais poderoso, o 80286.

Outro possível fator para a pouca aceitação deste processador pode ter sido a falta de unidades devido à demanda. Nunca havia chips suficientes para produzir computadores em grande escala.

3.2.2 Visão preliminar

Tanto o 8086 como o 8088 utilizam o conceito de fila de instruções para melhorar a velocidade do computador. Uma área no interior da pastilha denominada fila de instruções retém diversos bytes de uma instrução. Quando o computador estiver pronto para a próxima instrução, ele não precisa pegar muitos bytes na memória, uma vez que toda instrução poderá já se encontrar na fila. O conceito de fila aumenta o numero de operações realizadas por segundo uma vez que o processador vai estar utilizando o bus de dados e endereços por menor período de tempo, disponibilizando este para outros dispositivos. A fila do 8086 tem 6 bytes de largura e a do 8088 tem 4 bytes.

O 8086 pode acessar 1 megabyte de memória de leitura/escrita (2^{20} bytes). Entretanto ele utiliza um esquema de endereçamento de memória denominado segmentação, em que determinados registradores de segmento fornecem um endereço básico que é automaticamente acrescentado a cada endereço de usuário de 16 bits na máquina.

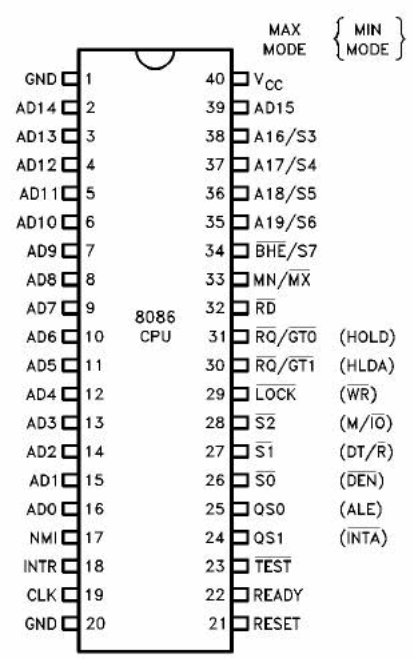

Figura 16: Pinagem do IA-PX 86 (WAITE, 1988)

A parte do endereço e todas as vias de dados são multiplexados em 16 pinos (os 16 pinos do barramento de dados é o que o classifica como um microprocessador de 16 bits). Os 4 bits restantes são implementados por quatro pinos adicionais de endereço, que também são utilizados para status (como mostra a figura 16). É requerido um relógio externo à pastilha e é utilizado um controlador de via externo à pastilha para demultiplexar a via de dados e de endereço.

O 8086 tem uma estrutura de interrupção poderosa. Quase todos os microprocessadores de 8 bits requerem pastilhas externas adicionais para permitir operações de interrupção adequadas. No 8086, cerca de 1000 bytes são colocados de lado para

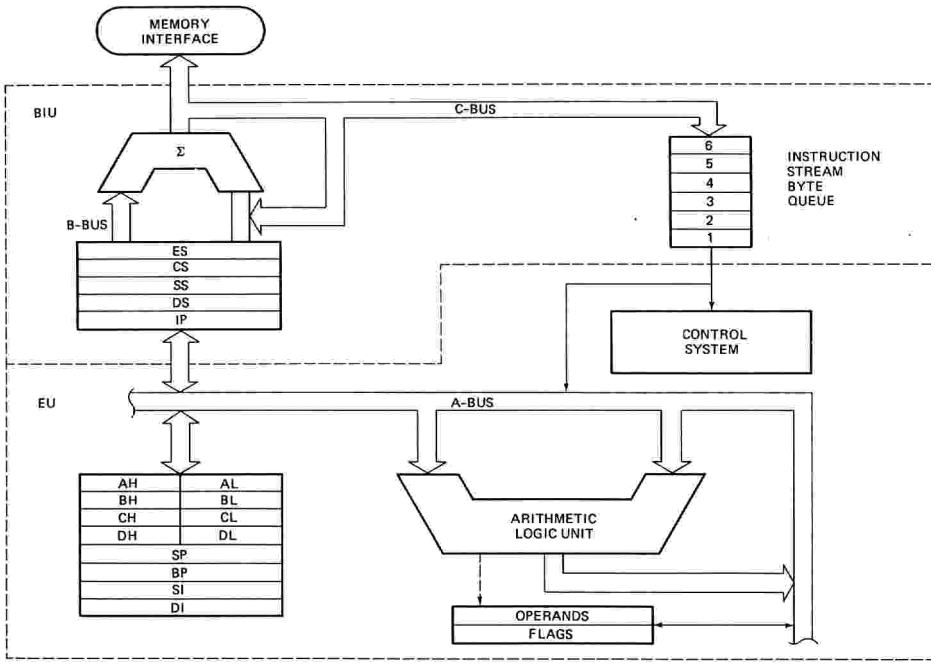

Figura 17: Arquitetura do 8086/8088 (WAITE, 1988)

conter até 265 apontadores de vetores (lembrando que cada apontador é um endereço contendo seletor:offset, ou seja, 4 bytes). O 8086 executa operações de E/S (ou I/O) em um espaço separado da memória denominado espaço de E/S. Tem um total de 64 KBytes. Para ser utilizado pelos co-processadores é fornecido um pino de entrada TEST especial (pino 23, Figura 16) para permitir ao 8086 saber quando é que o co-processador completou a tarefa. Quando uma instrução WAIT é acionada, o 8086 pára e aguarda do co-processador externo, ou qualquer outro hardware, um sinal para que continue pela alteração do pino TEST.

3.2.3 Memória

A largura de memória "vista" por um microprocessador é determinada pela quantidade de bits que o microprocessador pode acessar por vez. Esta quantidade é determinada pela largura do seu barramento de dados externo. O microprocessador 8086 possui um barramento de dados de 16 bits o que permite acessar dois bytes da

memória consecutivamente (cada referência à memória acessa 2 bytes), enquanto, o 8088 possui um barramento de dados de 8 bits, ou seja, uma referência à memória acessa 1 byte.

Os microprocessadores 8086/8088 podem acessar (ler ou escrever) bytes ou palavras (2 bytes) que se localizam tanto em endereços pares como em endereços ímpares da memória. No entanto, dependendo do microprocessador e do tamanho do dado a ser acessado, pode ser necessário que o microprocessador efetue uma ou duas referências para a memória. A Tabela 1 resume o número de referências necessárias para os microprocessadores 8086/8088 acessarem dados de 8 e 16 bits em endereços pares e ímpares da memória.

Tabela 1: Número de Referências 8086/8088

Tamanho do Dado	Endereço	Número de Referências	
		8086	8088
Byte	Par	1	1
	Impar	1	1
Palavra	Par	1	2
	Impar	2	2

O espaço de endereçamento da memória do microprocessador 8088 é organizado como um vetor linear de 1 MByte, sendo que cada localização deste espaço é referenciada por meio de um endereço único de 20 bits chamado endereço físico.

No microprocessador 8086 o espaço de endereçamento da memória é organizado em dois bancos de 512 KBytes cada, chamados bancos par e ímpar, respectivamente. O banco par é conectado ao 8086 por meio de 8 bits menos significativos do barramento de dados, já o banco ímpar é conectado por meio dos 8 bits mais significativos do barramento de dados.

3.2.4 Arquitetura do microprocessador

A unidade de execução (EU) executa as operações aritméticas e lógicas, além de controlar a maioria dos registros internos e manipular os dados. Em contraste, a unidade de interface de barramento (BIU representado por um símbolo de somatória na Figura 17) executa as operações de barramento, incluindo a transferência de dados, e controla os registros restantes do microprocessador.

As duas unidades de processamento são capazes de executar suas operações de forma independente, isto é, cada unidade pode fazer suas tarefas sem a assistência da outra unidade.

Embora a unidade de execução EU esteja isolada do barramento do sistema, pela unidade de interface de barramento (BIU), ela ainda pode acessar o barramento para certas operações. A EU ganha o acesso do barramento requisitando que a BIU temporariamente suspenda suas atividade. A EU então assume o controle da BIU de modo a poder usar o barramento para enviar ou receber informações.

Da Figura 17 nota-se que a EU consiste de duas seções principais que são os registros de propósitos gerais e a unidade aritmética e lógica - ALU. Os registros de propósito geral do 8086/8088 presentes na EU são áreas de armazenamento com capacidade de manter dados binários que foram ou serão usados pelo microprocessador. A grande vantagem destes registros se deve ao fato de se poder acessá-los com maior facilidade e de forma muito mais rápida do que uma determinada localização da memória.

Todos os registros de propósito geral possuem capacidade aritmética e lógica. Dados podem ser armazenados através da BIU ou transferidos para memória. Os registros de propósito geral são todos de 16 bits. Entretanto os bytes menos e mais significativos podem ser usados separadamente como registros de 1 byte. Dessa forma tem-se os seguintes registros: AH, AL, BH, BL, CH, CL, DH e DL.

A maioria das operações que se pode executar em um dos registros de propósito geral podem também ser executadas nos demais registros. Contudo, os registros

de propósito geral tem uso específico para algumas poucas instruções. Devido a este fato, os registros de propósito geral recebem nomes descritivos que são: Acumulador (AX), Base (BX), Contador (CX), Dado (DX), índice fonte (SI), índice destino (DI), ponteiro base (BP) e ponteiro da pilha (SP).

A unidade aritmética e lógica, ALU, recebe instruções e então executa sobre os dados especificados pela instrução uma operação aritmética, como soma ou subtração ou lógica como OR ou AND.

A seção de lógica de controle de barramento é responsável por todas as operações de barramento do microprocessador como, por exemplo, a busca de dados para a unidade aritmética e lógica (ALU). Quando necessário a seção de lógica de controle de barramento acessa localizações particulares na memória, de modo que a EU possa enviar ou receber informações para ou destas localizações.

A seção de lógica de controle de barramento também controla o sentido do fluxo de informação no barramento. Quando uma informação tiver que ser enviada à memória, esta seção assegura que os sinais de controle sejam os apropriados para a transmissão. O mesmo ocorre quando for necessário receber uma informação. A fila de instruções age como um "encanamento" onde os bytes das instruções trazidos da memória são armazenados antes do seu uso pela EU. No 8086 esta fila é composta de 6 localizações de 8 bits cada, enquanto no 8088 a fila de instruções é composta de 4 localizações de 8 bits. Pode-se dizer que estas localizações servem como áreas para o armazenamento temporário (buffer) das instruções trazidas da memória.

No microprocessador 8086/8088 é a seção lógica de controle de barramento BIU, que busca os bytes das instruções do programa na memória e os coloca na fila de instruções. Esta fila mantém estes bytes até que a EU esteja pronta para aceitá-las.

Devido as características do microprocessador 8086, a BIU sempre busca as instruções acessando palavras (16 bits) que se encontram armazenadas em endereços

pares. A única exceção ocorre quando existe um desvio (JUMP) para uma instrução que se encontra armazenada na memória em um endereço impar. Quando isso ocorre o 8086 traz para a fila de instruções um único byte da instrução e a seguir continua acessando palavras que se encontram armazenadas em endereços pares. Este fato não ocorre em um 8088 visto que seu barramento de dados é de 8 bits. É importante salientar que as instruções de um microprocessador 8086 podem ter de um a seis bytes de comprimento.

Independente do microprocessador, caso ocorra um desvio na seqüência de execução das instruções, a fila de instruções é automaticamente esvaziada e a BIU passa a buscar as instruções a partir da nova localização de memória para a qual se deu o desvio.

3.2.5 Endereçamento da memória

Ao contrário dos microprocessadores que utilizam um modelo de memória linear, ou seja, que enxergam o seu espaço de endereçamento de memória de forma sequencial, o 8086/8088 utiliza um modelo de memória denominado segmentada. Neste modelo o microprocessador enxerga o espaço de endereçamento de memória dividido em vários segmentos.

Um segmento nada mais é do que uma região continua do espaço de endereçamento de memória que é tratada pelo microprocessador como uma unidade lógica. Por serem unidades lógicas e não físicas, os segmentos podem localizar-se em qualquer parte do espaço de endereçamento linear. Conseqüentemente, dois ou mais segmentos distintos podem ser: adjacentes, parcialmente sobrepostos, totalmente sobrepostos ou desconexos.

No modelo de memória segmentada do 8086/8088, o microprocessador somente pode acessar as localizações de seu espaço de endereçamento por meio de um determinado segmento. Assim para este microprocessador, um segmento funciona como uma ”janela móvel”sobre o seu espaço de endereçamento linear, através do

qual ele acessa as localizações do seu espaço de endereçamento.

Para o microprocessador 8086/8088, os segmentos podem se localizar em qualquer parte do seu espaço de endereçamento de memória. Entretanto, devido a arquitetura deste microprocessador, um seguimento somente pode começar em uma localização do espaço de endereçamento de memória, cujo endereço físico seja múltiplo de 16 (10H).

O endereço físico do início de um segmento do 8086/8088 é designado por endereço base. Os 16 bits mais significativos do endereço base correspondem a um endereço chamado endereço de segmento ou seletor. Conseqüentemente, cada segmento do 8086/8088 é identificado por um endereço de segmento ou seletor de 16 bits.

Dentro de cada segmento o endereçamento se da de forma linear, porém relativo ao endereço de início do segmento. Cada localização do espaço de endereçamento de memória dentro do segmento é identificado por meio de um endereço de 16 bits chamado endereço efetivo (Effective Address - EA) ou endereço de offset (offset).

Devido a segmentação do espaço de endereçamento de memória e ao endereçamento relativo dentro do segmento, cada localização do espaço de endereçamento de memória do microprocessador é identificado por meio de um endereço de 32bits chamado de endereço lógico (seletor : offset).

A forma como o microprocessador converte um endereço lógico de 32 bits em um endereço físico de 20 bits, faz com que vários endereços lógicos identifiquem uma mesma localização do espaço de endereçamento de memória. A conversão de endereço lógico em endereço físico se da multiplicando o seletor por 10h e em seguida somando o offset. O endereço físico pode ser representado na forma normalizada para obter-se o endereço lógico, os 4 bits menos significativos do endereço físico correspondem ao endereço de offset e os 16 bits mais significativos do endereço físico correspondem ao endereço de segmento.

3.2.6 Conjunto de registros

Embora os registros SI, DI, BP e SP possuam capacidade aritmética e lógica de 16bits, como os demais registros de propósito geral de 16 bits, estes registros geralmente são usados para manter o endereço efetivo (offset) de localizações de memória ou para apontar estruturas de dados na memória.

Particularmente, o registro SP é utilizado para manter o endereço efetivo do topo de uma estrutura de dados na memória que funciona como uma pilha (stack), onde o último dado a ser armazenado nesta estrutura deverá ser o primeiro a ser retirado (LIFO - Last Input/first output). Uma vez que esta estrutura é de vital importância para o funcionamento do microprocessador, a utilização do registro SP em operações aritméticas ou lógicas não é aconselhada.

Como o microprocessador 8086/8088 utiliza um modelo de memória segmentada, o acesso às localizações do seu espaço de endereçamento de memória é feito através de segmentos mediante endereços lógicos de 32bits. Por isso, para poder acessar as localizações dentro de um determinado segmento é necessário que o 8086/8088 conheça o endereço deste segmento, ou seja, o seu seletor. Para isso, o microprocessador 8086/8088 dispõe de um conjunto de registros especiais chamados registro de segmentos, cuja finalidade, como o próprio nome indica, é manter endereços de segmentos.

Para acessar uma localização do espaço de endereçamento de memória que não é abrangida por um dos segmentos apontados pelos registros de segmentos, é necessário alterar o conteúdo de um dos registros de segmento, de modo que este registro aponte para um segmento que venha a abranger a localização que se deseja acessar.

Um programa, geralmente, é composto por três partes ou segmentos que são: segmento de códigos, segmento de dados e segmento de pilha. As partes ou segmentos de um programa podem residir em qualquer ordem e em qualquer lugar do espaço de endereçamento de memória que tenha memória física.

Parte do programa	Registro de segmento
Código do programa	CS
Dados do programa	DS
Pilha do programa	SS
Área extra da memória	ES

O ponteiro de instruções (IP) é um registro de 16 bits, que sempre mantém o endereço efetivo da localização de memória onde está armazenado o código de máquina da próxima instrução a ser executada. Este registro é automaticamente incrementado pelo microprocessador de acordo com o tamanho desta instrução.

A representação de um endereço lógico se dá na forma seletor:offset. Quando os conteúdos de dois registros são usados para especificar um endereço lógico, o endereço lógico geralmente é escrito na forma RS:RO onde RS corresponde ao nome do registro que mantém a parte do endereço lógico que se refere ao endereço de segmento e RO corresponde ao nome do registro que mantém a parte do endereço lógico que se refere ao endereço efetivo (offset). Os registros que podem ser utilizados para apontar localização dentro do segmento de dados são os registros BX, SI e DI.

No segmento de stack o ponteiro de stack (SP) mantém o endereço efetivo da localização de memória que corresponde ao topo da pilha. O endereço de segmento é mantido no registro de segmento SS.

O poder real de um microprocessador está na sua capacidade de tomar decisões. O 8086/8088 baseia as suas decisões no conteúdo de um registro de 16 bits chamado registro de flags. Este registro é automaticamente atualizado para manter informações a respeito da ultima operação aritmética ou lógica que o microprocessador executou. Apenas 9 flags do registro de flags são utilizados, são eles: OF - overflow, DF - direção, IF - interrupção, TF - armadilha, SF- sinal, ZF

- zero, AF - carry auxiliar, PF- paridade e CF - carry.

3.2.7 Instruções

Cada instrução (cartão de referencia do microprocessador 8086/8088 em anexo) possui uma representação binária única que é conhecida por código de máquina. Este modelo binário ou código de máquina da instrução, quando for aplicado aos circuitos internos do microprocessador faz com que ele execute uma operação particular. Uma instrução de um modo geral pode ser dividida em duas partes: código de operação (opcode) e operandos.

O código de operação (opcode) é a parte da instrução que identifica a operação básica a ser executada pelo microprocessador. Os operandos identificam os dados que devem ser utilizados.

Dependendo da instrução ela pode ter 2 operandos, 1 operando ou nenhum operando. Na representação das instruções com dois operandos, o operando destino é sempre especificado em primeiro, e este é separado do operando fonte por uma vírgula. Quando um operando se refere a um registro de 8 ou 16 bits ele é chamado de operando registro, e quando ele refere a uma localização de memória ele é chamado operando memória. Por outro lado, um operando imediato se refere a um dado de 8 ou 16 bits que é especificado na própria instrução.

Um operando pode se encontrar em um registro (operando registro), numa localização de memória (operando memória), num dispositivo periférico de entrada/saída, ou até mesmo estar codificado no próprio código de máquina da instrução (operando imediato). A maneira na qual a localização de um operando é especificada chama-se modo de endereçamento. O 8086/8088 utiliza duas categorias de endereçamento geral que são, o modo de endereçamento registro ou modo registro e o modo de endereçamento memória ou modo memória. Pode-se dizer que uma instrução do 8086/8088 utiliza o modo de endereçamento registro quando nenhum dos operandos da instrução se refere a memória. Por outro lado, uma instrução

utiliza o modo de endereçamento memória quando um dos operandos se refere a um operando memória.

Quando se trata de um operando memória até 3 valores de 16 bits podem ser somados para especificar seu endereço efetivo. Nesta soma qualquer carry que venha a ocorrer é ignorado pelo microprocessador, por um endereço efetivo (offset) no 8086/8088 é representado por um número de 16 bits.

Sobre as 2 categorias gerais de endereçamento existem 7 modos específicos de endereçamento.

3.2.7.1 Endereçamento por registro

Uma instrução utiliza o modo de endereçamento por registro quando os operandos fonte e destino forem registros.

Exemplo: MOV AX, BX.

3.2.7.2 Endereçamento imediato

Uma instrução utiliza o modo de endereçamento imediato quando o operando fonte desta instrução for imediato.

Exemplos: MOV CX, 1234h (modo registro); MOV [2011H], 1234H (modo memória).

3.2.7.3 Endereçamento Direto

Uma instrução utiliza o modo de endereçamento direto quando o operando destino ou operando fonte da instrução se refere a uma localização de memória, cujo endereço efetivo é especificado na própria instrução.

Exemplos: MOV CX, [1234H]; MOV [1234H], DX

3.2.7.4 Endereçamento indireto por registro

Uma instrução utiliza o modo de endereçamento indireto por registro quando o operando destino ou o operando fonte da instrução se refere a um operando memória, cujo endereço efetivo se encontra armazenado num registro.

Exemplo: MOV CX, [BX];

3.2.7.5 Endereçamento por base

Uma instrução utiliza o modo de endereçamento por base quando o operando destino ou operando fonte da instrução se refere a um operando memória, cujo endereço efetivo (EA) é especificado pela soma do conteúdo do registro BX ou BP com um número de 8 ou 16 bits chamado deslocamento. No caso do deslocamento de 8 bits o microprocessador estende o sinal até se obter um numero binário sinalizado de 16 bits, quando ocorrer um deslocamento de 16 bits o microprocessador interpreta como um numero absoluto de 16 bits.

Quando o conteúdo do registro BP é utilizado no cálculo, o 8086/8088 automaticamente associa o endereço efetivo do operando memória com o conteúdo do registro de segmento de stack (registro SS), de modo a formar o endereço lógico do operando memória. Neste caso, portando, o 8086/8088 acesa o operando memória no segmento da pilha.

Exemplo: MOV AX, [BX + 1000H]

3.2.7.6 Endereçamento Indexado

Uma instrução utiliza o modo de endereçamento indexado quando o operando destino ou o operando fonte da instrução se refere a um operando memória, cujo endereço efetivo é especificado pela soma do conteúdo do registro SI ou DI, com um número binário de 8 ou 16 bits chamado deslocamento. No caso do deslocamento de 8 bits o microprocessador estende o sinal até se obter um número

binário sinalizado de 16 bits, quando ocorrer um deslocamento de 16 bits o microprocessador interpreta como um número absoluto de 16 bits.

Exemplo: MOV AX, [SI + 2000H]

3.2.7.7 Endereçamento por base indexado

Uma instrução utiliza o modo de endereçamento por base indexado quando o operando destino ou o operando fonte da instrução se refere a um operando memória, cujo endereço efetivo é especificado pela soma do conteúdo do registro BX ou BP, com o conteúdo do registro SI ou DI e opcionalmente um número binário de 8 ou 16 bits chamado deslocamento.

Exemplo: MOV AX, [BX + SI + 2000H]

3.3 VHDL

A linguagem VHDL foi desenvolvida pela necessidade de um padrão para o intercâmbio de informações entre fornecedores de equipamentos para o Departamento de Defesa dos Estados Unidos. Esta linguagem é usada para descrever o comportamento de circuitos ou sistemas eletrônicos a partir de um sistema físico. É importante lembrar que esta é uma linguagem concorrente, ou seja, os comandos envolvidos em um mesmo evento acontecem simultaneamente, diferentemente de linguagens de programação de software. Além disso, é uma linguagem portável, ou seja, independe da tecnologia ou do fornecedor. A Figura 18 mostra as etapas de um projeto utilizando VHDL.

Na lógica programável, há dois dispositivos principais: CPLD (*Complex Programmable Logic Devices*) e FPGA (*Field Programmable Gate Arrays*) no campo de ASIC (*Application Specific Integrated Circuits*). A partir do código VHDL, pode-se fabricar um chip de alta complexidade ou executá-lo em um dispositivo programável.

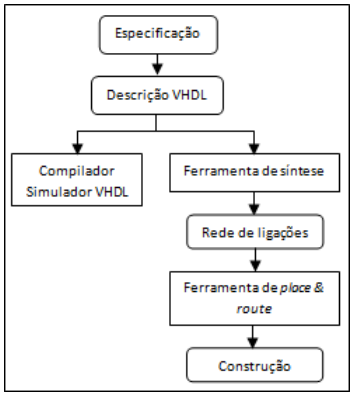

Figura 18: Etapas de Projeto Usando VHDL

O código VHDL é composto de três partes principais: Biblioteca, Entidade e Arquitetura.

1. Biblioteca (*Library*) : É composta de todas as bibliotecas usadas no projeto.
2. Entidade (*Entity*): Determina as entradas e saídas do circuito.
3. Arquitetura (*Architecture*): Contém o código VHDL que descreve a forma como o circuito deve se comportar (*function*).

3.3.1 Biblioteca

Uma biblioteca têm várias implementações de código que são úteis a outros projetos. A Figura 19 ilustra a estrutura típica de uma biblioteca. O código, normalmente, é escrito na forma de funções (*Functions*), procedimentos (*Procedures*) ou componentes (*Components*), que ficam dentro de pacotes (*Packages*) e depois é compilado na biblioteca.

3.3.2 Entidade

Uma entidade de projeto pode representar uma simples porta lógica como um sistema completo. A declaração da entidade define a interface com o ambiente

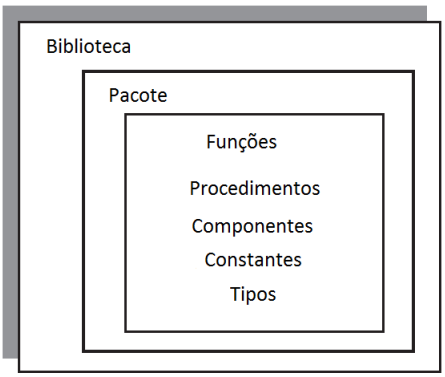

Figura 19: Estrutura de uma Library (PEDRONI, 2011)

exterior, como, por exemplo, as entradas e saídas. Os quatro modos de porta são:

1. IN : Apenas entrada.
2. OUT: Apenas saída.
3. BUFFER: Saída que controla sinal interno.
4. INOUT: Porta bidirecional

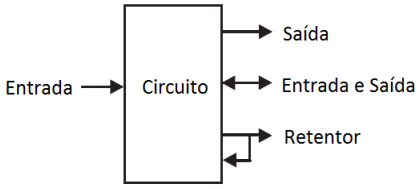

Figura 20: Tipos de Entrada e Saída (PEDRONI, 2011)

3.3.3 Arquitetura

A arquitetura contém a parte lógica da entidade utilizando suas entradas e saídas. Ainda é possível declarar sinais internos dentro da arquitetura, estes sinais são chamados classes. São elas:

1. CONSTANT - Define um objeto com valor estático.

2. VARIABLE - São objetos que podem ter o seu valor alterado, e são usadas em regiões de código seqüencial.

3. SIGNAL - São objetos que podem ter o seu valor alterado, e são usadas em regiões de código concorrente ou seqüencial. É bom lembrar que a porta de uma entidade realiza a declaração de um sinal.

```
ARCHITETURE architeture_name OF entity_name IS
        [declarations]
BEGIN
        (code)
END architeture_name;
```

Figura 21: Sintaxe de uma Architeture

A arquitetura é composta de duas partes, uma para declarações, onde sinais e constantes são declarados e outra onde fica o código. Como no caso da entidade, o nome da arquitetura pode ser qualquer nome (exceto palavras reservadas), até mesmo o nome da entidade.

4 Desenvolvimento

4.1 Requisitos de Funcionamento

4.1.1 Software

- Altera Quartus II Version 13.1 Build 162 09/02/2014 SJ Web Edition
- ModelSim Altera Starter Edition 13.1
- Sistema Operacional Windows 7/8/8.1

4.1.2 Hardware

- Microcomputador de 1GHz ou superior
- 256 MB de Memória RAM
- 4GB de espaço disponível em disco

4.1.3 Ferramentas Utilizadas no Projeto

- Software Quartus II 13.1 Web Edition
- ModelSim Altera Starter Edition 13.1
- Notebook Dell Inspiron 14R: Intel Core i7, 8GB de Memória RAM, 1TB de espaço em disco.

- Notebook Samsung RF511-SD3BR: Intel Core i7, 8GB de Memória RAM, 1TB de espaço em disco.

4.2 Definição do Conjunto de Instruções

4.2.1 Conjunto de Instruções CISC

Em uma máquina CISC, como o iAPX8086, há centenas de instruções de diversos tamanhos. Isso se justificava no passado pela velocidade lenta da memória, uma vez que após acessá-la para buscar uma instrução a execução das demais instruções normalmente não precisava de outro acesso.

No entanto, as memórias de hoje são de alto desempenho, e dispensam a precaução de evitar acessá-las. Diante disso, um programa CISC se torna complexo e demanda tempo para ser executado, uma vez que possui instruções complexas que requerem vários ciclos de relógio e raramente são utilizadas.

Um programa escrito para uma máquina CISC, possui instruções escritas de um modo menor, ou seja, estão mais simples, porém mais codificadas. Quando o processador recebe tais instruções ele tem de decodificá-las em código de máquina para que seus circuitos possam executá-las.

Os processadores de arquitetura CISC contêm uma micro-programação, ou seja, um conjunto de códigos de instruções que são gravados no processador. Desta forma, este recebe as instruções dos programas e as executa utilizando as instruções contidas em sua micro-programação.

A Figura 22 a seguir ilustra o processo de quebra do código que chega ao microprocessador, já em baixo nível, em diversas instruções mais próximas do hardware, estas por sua vez estão contidas no microcódigo do processador.

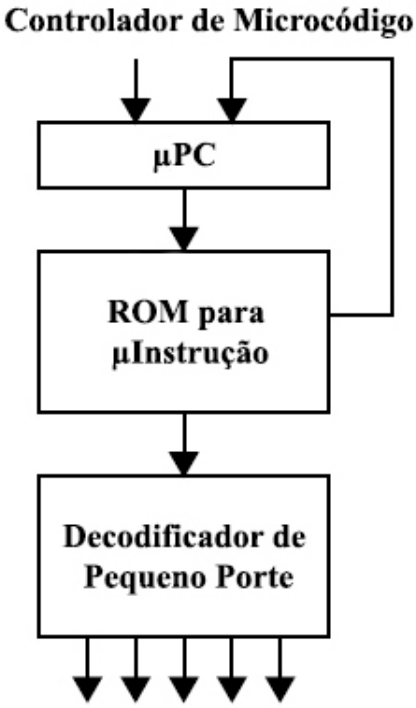

Figura 22: Sequência para Execução de Instruções CISC

4.2.2 Conjunto de Instruções RISC

Em uma máquina RISC, como a desenvolvida neste trabalho, o conjunto de instruções é reduzido. O hardware suporta um conjunto mínimo de funções, sendo estas operações aritméticas e lógicas, transferência de dados entre CPU, memória e periféricos, além de operações de controle da máquina.

Uma das principais características das instruções é que cada uma executa uma ação muito simples. Assim, uma máquina RISC tem suas instruções compiladas diretamente para código de máquina, não sendo necessária uma posterior quebra em microcódigo (como ocorre em máquinas CISC).

A Figura 23 a seguir ilustra o processo de compilação dos programas de um microprocessador de arquitetura RISC.

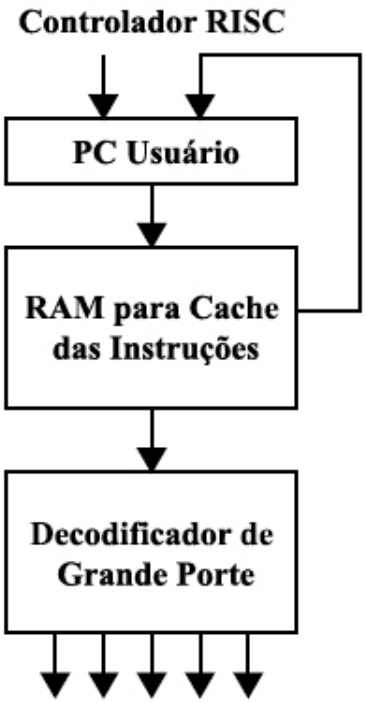

Figura 23: Sequência para Execução de Instruções RISC

A definição de um conjunto de instruções para uma máquina RISC segue as seguintes regras:

- Analisar aplicações para identificar operações-chave
- Projetar um processador que seja eficiente para executar essas operações.
- Projetar instruções que realizam as operações-chave.
- Acrescentar mais instruções necessárias, cuidando para não afetar a velocidade da máquina.

4.2.3 Instruções Escolhidas

Para a implementação de um microprocessador o primeiro item que necessita ser pensado e desenvolvido com muita atenção é o conjunto de instruções. Por-

tanto, dentre 255 instruções do microprocessador 8086, foram escolhidas somente as instruções que são de funcionamento básico de um programa de computador.

Realizando um pararelo entre os dois tipos possíveis de conjunto de instruções, o microprocessador em desenvolvimento implementará instruções semelhantes ao conjunto do Microprocessador MIPS, que possui um conjunto de instruções bem reduzido, somente com 50 instruções e todos opcodes com o tamanho fixo de 6 bits. Essas instruções elas subdividas em somente 3 tipos, que são: I,J e R. Sendo o tipo I, as operações de registro - imediato, as instruções J, que são os jumps e as instruções R, que são as instruções registro-registro.

"O design do conjunto de instruções deve ter vários objetivos, sendo o mais óbvio e útil a performance do microprocessador."(HENNESSY, 1984).

Neste momento a preocupação é desenvolver um microprocessador que tenha um funcionamento básico e muito explícito, pois o foco deste trabalho não é desenvolver uma tecnologia nova, porém compreender profundamente o desenvolvimento da arquitetura e dos componentes de um microprocessador e lembrando que, as máquinas RISC só se tornaram viáveis devido aos avanços de software no aparecimento de compiladores otimizados. (SONG, 2003).

Para adequar o 8086 a filosofia RISC, somente o modo de endereçamento imediato é possível, pois torna o microprocessador uma máquina *Load/Store*. Nenhuma operação memória-memória se adequa a filosofia. Além do conjunto de instruções característico, os microprocessadores RISC possuem normalmente uma grande quantidade de registradores, devidamente pela impossibilidade de realizar operações memória-memória, porém o microprocessador a ser desenvolvido, por seguir as características de um 8086, possuirá somente os mesmos registradores existentes no microprocessador 8086 CISC, em busca de facilitar o desenvolvimento. Além disso, com o foco de que o mesmo código gerado para um microprocessador 8086 CISC possa ser utilizado, respeitando as mesmas instruções, em sua versão RISC.

Para o devido gerenciamento de memória, o microprocessador a ser desen-

volvido não irá ter em seu conjunto os modos que fazem uso de segmentação. Portanto, a memória será enxergada como uma memória linear, como se tivessemos somente um segmento sendo utilizado. Tal definição será explicitada mais a diante. Semelhante ao funcionamento do micrprocessador 386 que possui além de um modo de memória segmentada, um modo protegido no qual a memória é vista linearmente.

Portanto, após todas as considerações tomadas para o desenvolvimento do conjunto de instruções, temos na Tabela 2 o conjunto de instruções escolhido para o desenvolvimento deste microprocessador.

Tabela 2: Instruções Escolhidas e seus devidos Opcodes

Instrução	Opcode
ADD Reg16,Imed 16	10000001 11000 R/M I16L I16H
OR Reg16,Imed16	10000001 11001 R/M I16L I16H
ADC Reg16,Imed16	10000001 11010 R/M I16L I16H
SBB Reg16,Imed16	10000001 11011 R/M I16L I16H
AND Reg16,Imed16	10000001 11100 R/M I16L I16H
SUB Reg16,Imed16	10000001 11101 R/M I16L I16H
XOR Reg16,Imed16	10000001 11110 R/M I16L I16H
CMP Reg16,Imed16	10000001 11111 R/M I16L I16H
MOV Reg16,Imed16	00000000 10111 R/M I16L I16H

Para realizar uma verificação das instruções escolhidas e determinar a sua finalidade, aproveitamos da liberação que a Microsoft realizou nos últimos dias, abrindo o código fonte do sistema MS-DOS para o Museu da História da Computação, o qual foi disponibilizado pela internet (SHUSTEK, 2014). Neste arquivo existem duas versões do MS-DOS. Como base utilizamos a versão 1.1 do MS-DOS, com os arquivos fontes escritos em Asembly. Baseado históricamente, este sistema era executado em um processador Intel®. Portanto, desenvolveu-se um código na linguagem Python, que encontra-se em anexo, para realizar uma varredura em todos os arquivos .asm para checar a quantidade de instruções semelhantes as que foram definidas como instruções do microprocessador a ser implementado. Com os resultados, obtivemos a figura 24.

Para realizar uma verificação das instruções escolhidas e determinar a sua finalidade, utilizou-se da liberação que a Microsoft realizou nos últimos dias, que foi, abrir o código fonte do sistema MS-DOS para o Museu da História da Computação, o qual foi disponibilizado pela internet (SHUSTEK, 2014), neste arquivo que pode ser feito realizado o download livremente, existem duas versões do MS-DOS, como base utilizamos a versão 1.1 do MS-DOS, com os arquivos fontes escritos em Asembly. Baseado históricamente, este sistema era executado em um processador Intel®. Portanto, desenvolveu-se um código na linguagem Python, que encontra-se em anexo, para realizar uma varredura em todos os arquivos .asm para checar a quantidade de instruções semelhantes as que foram definidas como instruções do microprocessador a ser implementado, com os resultados, se tem a Figura 24.

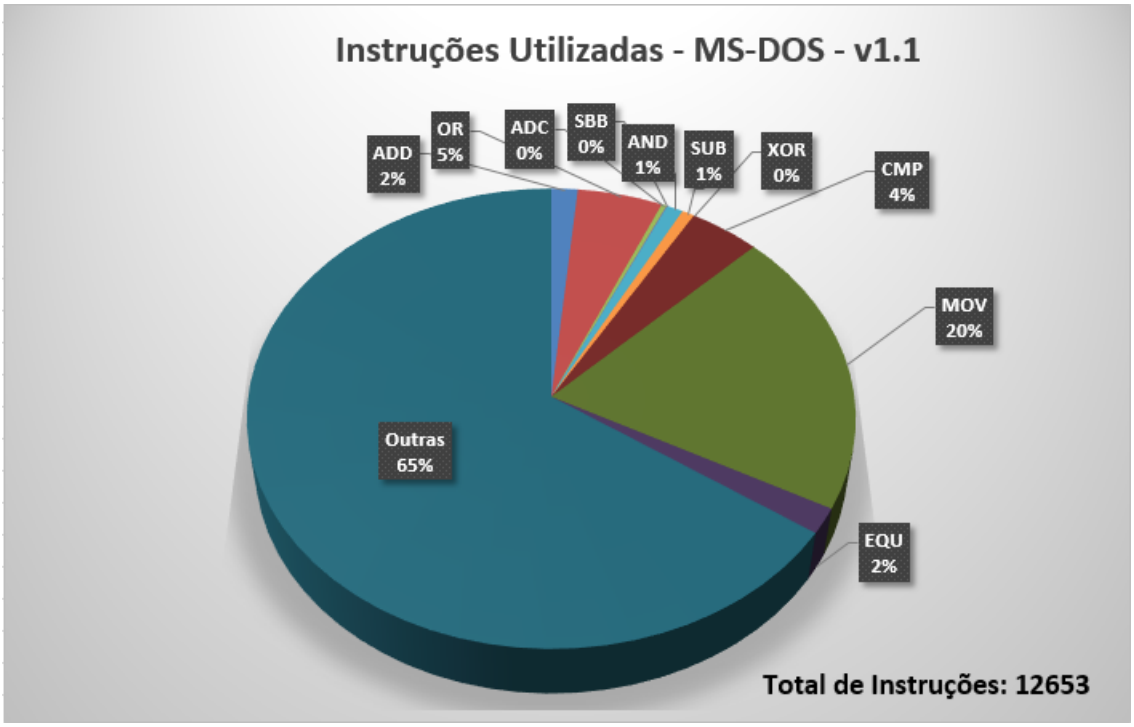

Figura 24: Análise quantitativa das instruções do código do MS-DOS(SHUSTEK, 2014)

Assim como em um projeto de pesquisa da IBM identificou que a maioria das instruções eram usadas com pouca freqüência. Cerca de 20% delas eram usadas

80% das vezes. Os próprios desenvolvedores de sistemas operacionais habituaram-se a determinados subconjuntos de instruções, tendendo a ignorar as demais, principalmente as mais complexas(**??**). No gráfico podemos ver claramente a alta utilização de instruções MOV, independente do seu tipo de endereçamento, o torna a instrução mais importante do código.

Assim como em um projeto de pesquisa da IBM identificou que a maioria das instruções eram usadas com pouca freqüência. Cerca de 20% delas eram usadas 80% das vezes. Os próprios desenvolvedores de sistemas operacionais habituaram-se a determinados subconjuntos de instruções, tendendo a ignorar as demais, principalmente as mais complexas (SONG, 2003), no gráfico se vê claramente a alta utilização de instruções MOV, independente do seu tipo de endereçamento, o que a torna essencial ao processador aqui desenvolvido.

4.2.4 Definição do Tamanho das Instruções

Na arquitetura CISC há instruções de diversos tamanhos. Um exemplo é o iAPX8086, que contém instruções que variam de 1 até 6 bytes. Desta forma, cada uma destas instruções requer uma quantidade diferente de ciclos de máquina para serem executadas.

Na arquitetura RISC todas as instruções possuem o mesmo tamanho. E, segundo a regra de ouro, todas as instruções devem ser executadas em apenas um ciclo de máquina. O problema quanto à regra de ouro são as instruções *LOAD* e *STORE* que requerem mais ciclos de relógio, uma vez que estas fazem acesso à memória.

Neste ponto, a regra de ouro tem de ser adaptada. A solução proposta é que a cada ciclo de relógio a execução de uma nova instrução seja iniciada. E nessa característica é que entra o uso da técnica de *pipelining*.

Para o microprocessador desenvolvido neste trabalho as instruções possuem 4 bytes. Este tamanho de instrução foi definido visando uma melhor administração

da memória disponível e respeitando a regra de ouro, onde todas as instruções tem o mesmo tamanho. Na Figura 25 a seguir tem-se a estrutura das instruções aritméticas e lógicas.

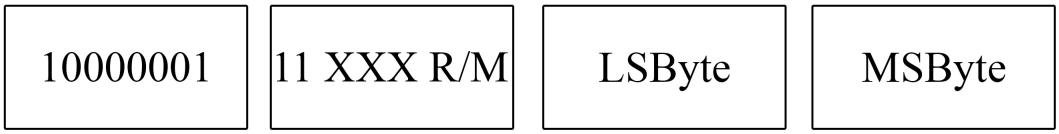

Figura 25: Bytes das Instruções Aritméticas e Lógicas

Na Figura 26 a seguir tem-se a estrutura da instrução MOV.

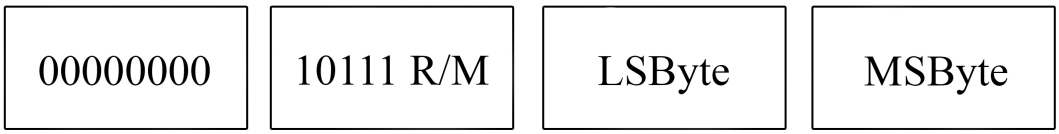

Figura 26: Bytes da Instrução MOV

4.3 Implementação em VHDL

Á seguir, será descrito todos os componentes em VHDL que foram desenvolvidos passo a passo, cada um com a sua devida validação com um *testbench* executado pelo Modelsim, todos os códigos desenvolvidos esão localizados em anexos.

4.3.1 Registro de Propósito Geral

O registrador de próposito geral tem como objetivo encapsular os registradores AX, BX, CX, DX, SP, BP, SI e DI. Como definido anteriormente, todos os registro são de 16bits, portanto além de possuir entrada e saída de 16 bits, possui uma entrada de controle para a seleção correta do registrador desejado, que é determinado no opcode pelo item R/M, que segue por padrão, assim como no 8086, a tabela 3. Além de entradas para o *relógio* e *reset* do sistema e uma entrada de

controle de leitura/escrita, sendo leitura em nível baixo e escrita em nível alto, além da entrada de habilitação do componente para o devido controle da máquina de estados do microprocessador.

Tabela 3: Códigos de controle do Registro de Propósito Geral

Registrador	**Código R/M**
AX	000
BX	011
CX	001
DX	010
SP	100
BP	101
SI	110
DI	111

Temos na Figura 27 o resultado produzido pela execução do *testbench*, podemos verificar que o dado passado no barramento de entrada de dados, denominado *entradaDados_tb*, é salvo dentro do registrador AX ou DX, dependendo do sinal do seletor. Além de verificar o funcionamento dos sinais de habilitação e de leitura e escrita, onde o ”0”determina a escrita e ”1”determina a leitura, sendo que o funcionamento é controlado pelos sinais de relógio e *reset*. Além do barramento de entrada de dados podemos verificar o funcionamento do barramento de saída quando o sinal de leitura está em ”0”.

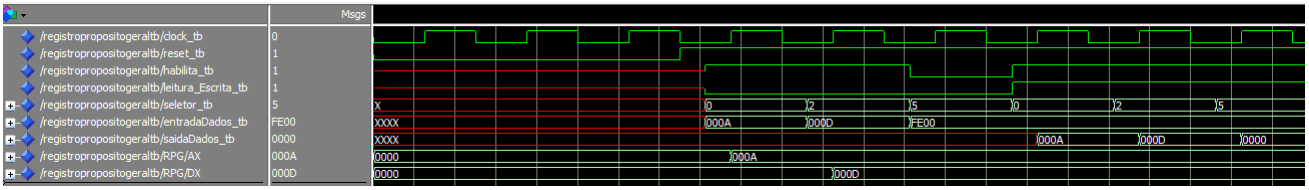

Figura 27: Resultado do *testbench* aplicado ao componente de Registro de Propósito Geral

4.3.2 Registro de Segmento

Com funcionamento semelhante do Registro de Propósito Geral, possui o objetivo de implementar os registradores que possuem o funcionamento de manipulação de memória que são, CS, ES, DS, SS, além do contador de índice IP. Diferentemente do Registro de Popósito Geral, o registro de Segmento possui uma linha de entrada denominada incrementa_IP, tal linha foi implementada a fim de facilitar o comando de incremento de IP para se caminhar continuamente na memória, sendo que o registro possui acesso direto ao registrador IP, em vez de se desenvolver uma estrutura própria para este cálculo. Tal registro também possui seu devido barramento de controle de 3 bits, que está descrito na tabela 4, porém como descrito anteriormente este barramento será fixado em "001"para que somente seja utilizado o segmento de código para o funcionamento deste microprocessador.

Tabela 4: Códigos de controle do Registro de Segmento

Registrador	**Código R/M**
ES	000
CS	001
SS	010
DS	010
IP	100
Todos em Alta Impedância	outros

Temos na Figura 28 o resultado produzido pela execução do *testbench*, semelhante ao registro de Propósito Geral, podemos verificar que o dado passado no barramento de entrada de dados, denominado *entradaDados_tb*, é salvo dentro dos registradore ES,CS,SS,DS ou IP dependendo do sinal do seletor. Além de verificar o funcionamento dos sinais de habilitação e de leitura e escrita, onde o "0"determina a escrita e "1"determina a leitura, sendo que o funcionamento é controlado pelos sinais de relógio e *reset*. Além do barramento de entrada de dados podemos verificar o funcionamento do barramento de saída quando o sinal de leitura está em "0". E diferentemente do Registro de Propósito Geral, este registro possui um sinal de incrementa_IP o qual podemos verificar seu funcionamento quando está

em nível alto, incrementando em 1 o valor do registro de índice.

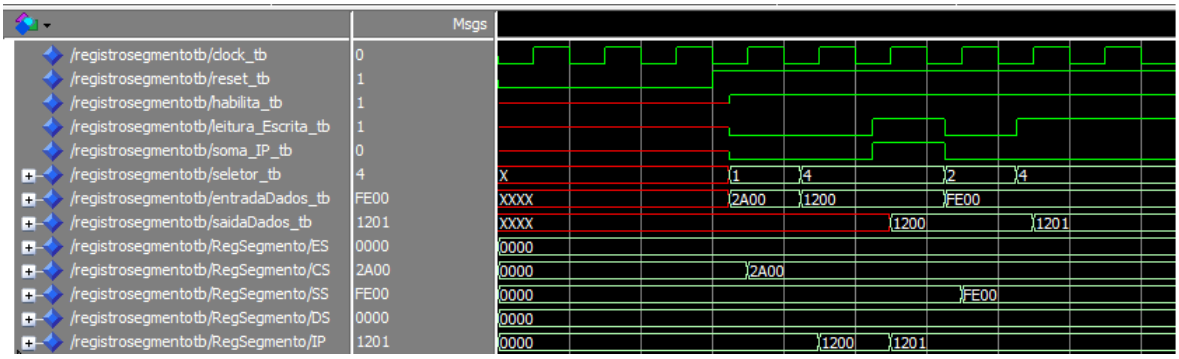

Figura 28: Resultado do *testbench* aplicado ao componente de Registro de Segmento

4.3.3 Calculadora de Endereço

Foi desenvolvido uma calculadora de endereço para realizar o cálculo do endereço relativo para o endereço físico, por exemplo, temos o seguinte endereço lógico CS:IP, sendo CS com 1200h e IP com 3450h, então, a calculado tem o papel de converter este endereço lógico, realizando a seguinte operação: 12000h + 3405h = 15405h, um endereço físico de 16 bits que é colocado no barramento de endereço do microprocessador, a fim de apontar um endereço na memória. Assim foi desenvolvido um componente simples, com duas entradas de 16 bits e uma saída de 20 bits, temos na Figura 29 a execução do exemplo dado logo a cima.

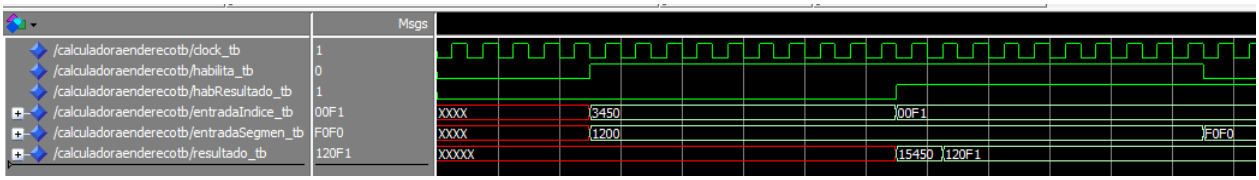

Figura 29: Resultado do *testbench* aplicado a Calculadora de Endereço

4.3.4 Demultiplexador

O Demultiplexador seleciona um dos diversos dados de entrada e o transfere para a saída. Foi implementado para possuir duas saídas de 16 bits, devido requisitos básicos do sistema.

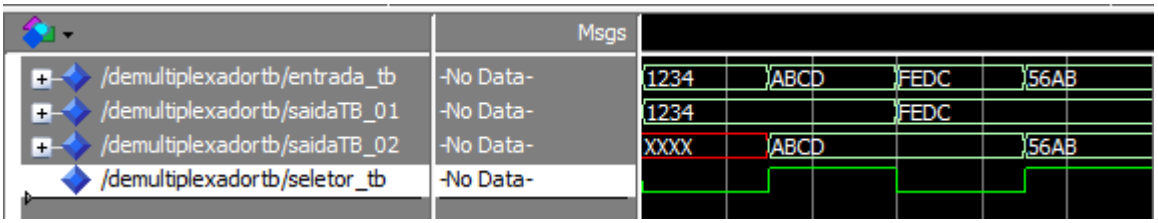

Figura 30: Resultado do *testbench* aplicado ao Demultiplexador

4.3.5 Multiplexador

O Multiplexador seleciona um dos diversos dados de entrada e o transfere para a saída. Assim como o Demultiplexador, foi implementado com duas entradas de 16 bits.

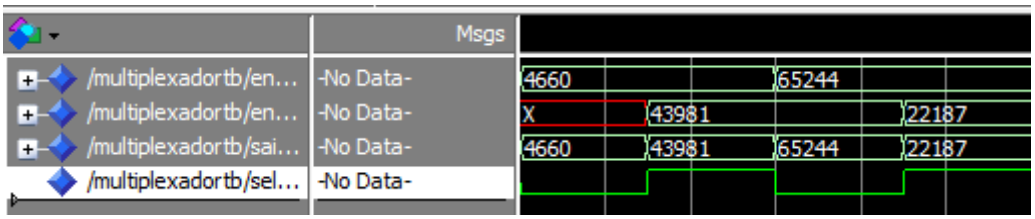

Figura 31: Resultado do *testbench* aplicado ao Multiplexador

4.3.6 Registro de Flags

O Registro de Flags tem como objetivo guardar o resultado dos flags que são calculado pela Unidade Lógica Aritmética. Funciona como um flip-flop para cada sinal de entrada, porém como saída possui um único vetor de 16 bits, para que te-

nha uma exibição semelhante à maneira de se manipular flags no microprocessador 8086, podemos verificar seu funcionamento na Figura 32.

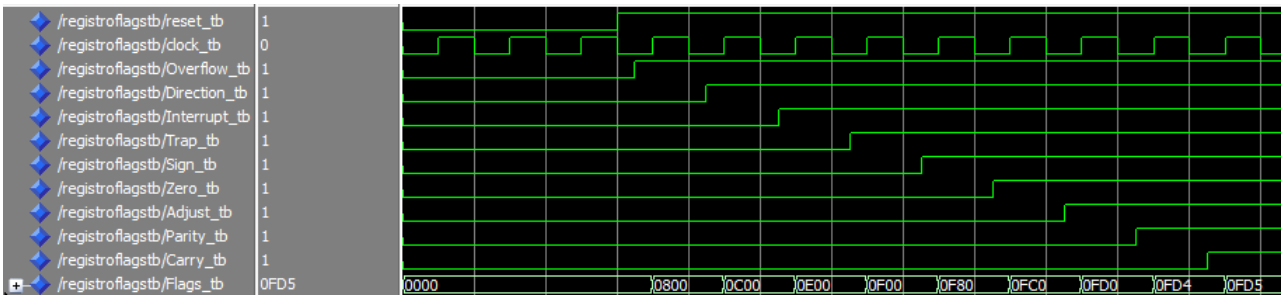

Figura 32: Resultado do *testbench* aplicado ao Registro de Flags

4.3.7 Unidade de Controle de Endereços

A Unidade de Controle de Endereços possui como objetivo controlar alguns componentes a maneira de realizar a perfeita sincronização entre eles, e que exista um fluxo consistente de dados entre as estruturas para que o cálculo que seja realizado, e que seja correto. Portanto a Unidade de Controle de Endereços é a implementação de uma máquina de estados 33, que é iniciada em reset, e estimulado por pulsos de relógio e por um sinal de habilitação advindo da Unidade de Controle, responsável pela realização da sincronia total do microprocessador.

4.3.7.1 Diagrama de Estados

O sinal de habilita existe para que exista uma espécie de domínio da Unidade de Controle sobre a Unidade de Controle de Endereços, pois o papel da controladora de endereços é fornecer á memória o endereço correto para que a unidade de controle sempre acesse no barramento de dados, um dado válido para que a máquina não trave e sempre continue processando cíclicamente, temos na Figura 35 o resultado do *testbench* realizado com a unidade.

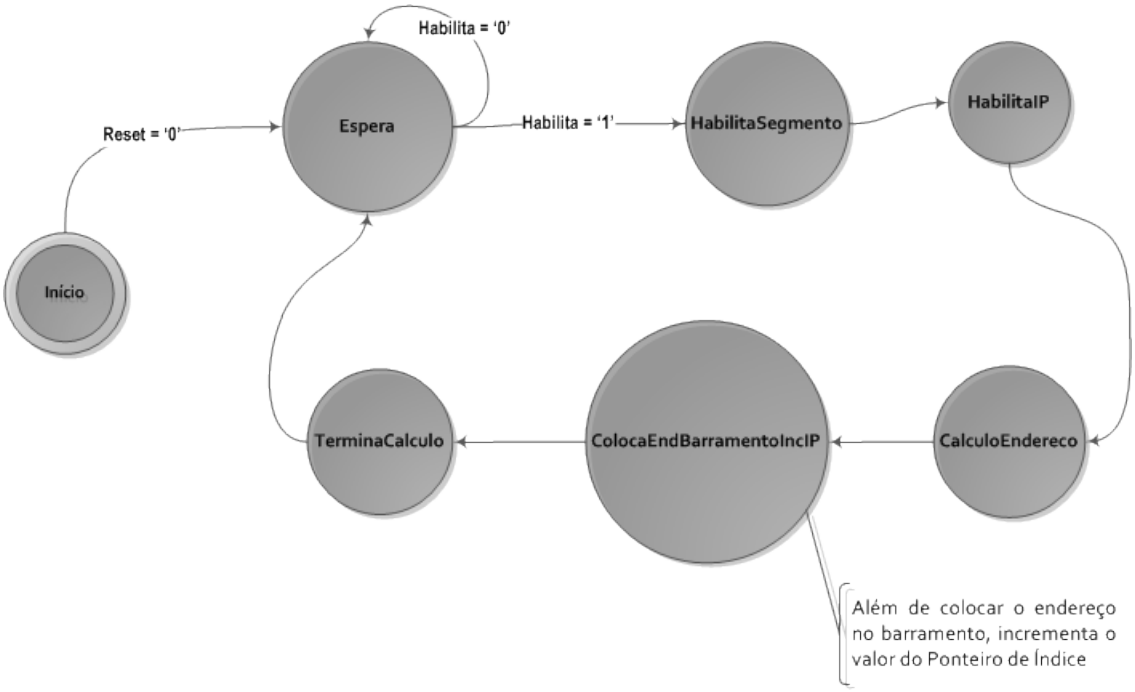

Figura 33: Diagrama de Estados da Unidade de Controle de Endereços

4.3.8 Memória ROM

A memória ROM é utilizada para armazenar os opcodes das instruções. Os valores armazenados na estrutura representam o que seria o resultado da compilação de um software. Desta forma, esta é utilizada para simulação da funcionalidade do microprocessador diante de um código assembly resultante do processo de compilação.

Na Figura 34 abaixo tem-se a simulação da funcionalidade desta estrutura:

Figura 34: Saída Memória ROM

4.3.9 Unidade Aritmética e Lógica - ULA

É a estrutura responsável por executar todas as operações aritméticas e lógicas do microprocessador. É composta por algumas estruturas auxiliares, já descritas neste documento, que implementam funcionalidades necessárias na execução bit a bit de cada instrução. Os dados chegam na ULA através do multiplexador ou direto da unidade de controle, a respectiva operação é executada e então o resultado é colocado no registrador correspondente definido pela unidade de controle.

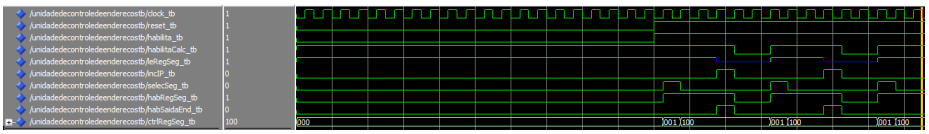

Figura 35: Resultado do *testbench* aplicado a Unidade de Controle de Endereços

4.3.9.1 Detector do Flag Auxiliar

O Detector do Flag Auxiliar é uma estrutura criada para ser utilizada na Unidade Aritmética e Lógica. Esta estrutura detecta a ocorrência de um ”vai um”do bit 3 para o bit 4 ou quando há ”vem um”do bit 4 para o bit 3, durante a execução de alguma instrução aritmética. A validação da funcionalidade da estrutura através de um *testbench* é apresentada na Figura 36 a seguir:

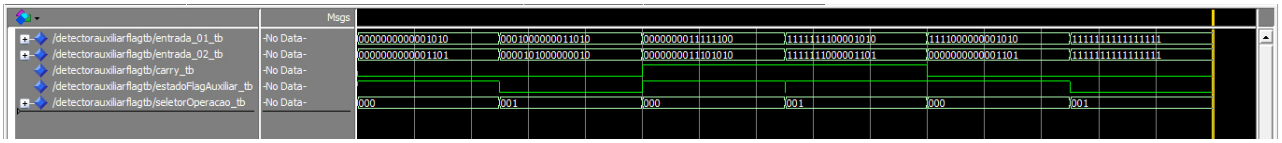

Figura 36: Saída Estrutura Detector Auxiliar Flag

4.3.9.2 Detector do Flag de Paridade

O Detector do Flag de Paridade é uma estrutura que detecta a paridade do resultado obtido em uma instrução aritmética ou lógica. É utilizado na Unidade

Aritmética e Lógica. Caso haja um número par de bits 1 no resultado da operação, o flag de paridade recebe valor 1, caso contrário, recebe valor 0. Na Figura 37 a seguir, tem-se a validação da funcionalidade da estrutura:

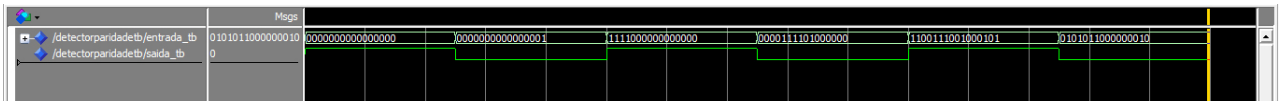

Figura 37: Saída Estrutura Detector Flag de Paridade

4.3.9.3 Detector do Zero Flag

Esta estrutura verifica o resultado obtido em uma operação aritmética ou lógica. Caso o valor final seja 0, o zero flag recebe valor 1, caso contrário, recebe valor 0. Na Figura 38 a seguir se tem a validação da funcionalidade da estrutura:

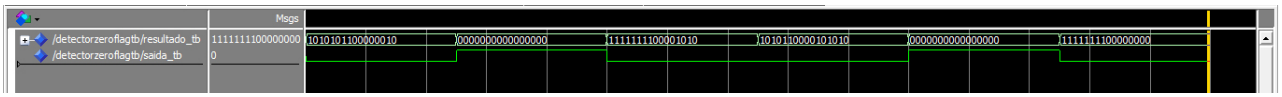

Figura 38: Saída Estrutura Detector Zero Flag

4.3.10 Unidade de Controle

A Unidade de Controle é uma das principais estruturas do microprocessador. Ela é responsável por 3 funções básicas: busca (fetch), decodificação e execução. Além disso, gera todos os sinais que controlam as unidades *escravas* à ela. A *Unidade de Controle de Endereços* é uma máquina de estados escrava à unidade de controle, portanto há um sinal de habilitação que conecta essas duas unidades que é enviado pela Unidade de Controle, tornando-a uma máquina de estados *master*. Podemos verificar o diagrama de estados implementado na Unidade de Controle, na figura 39.

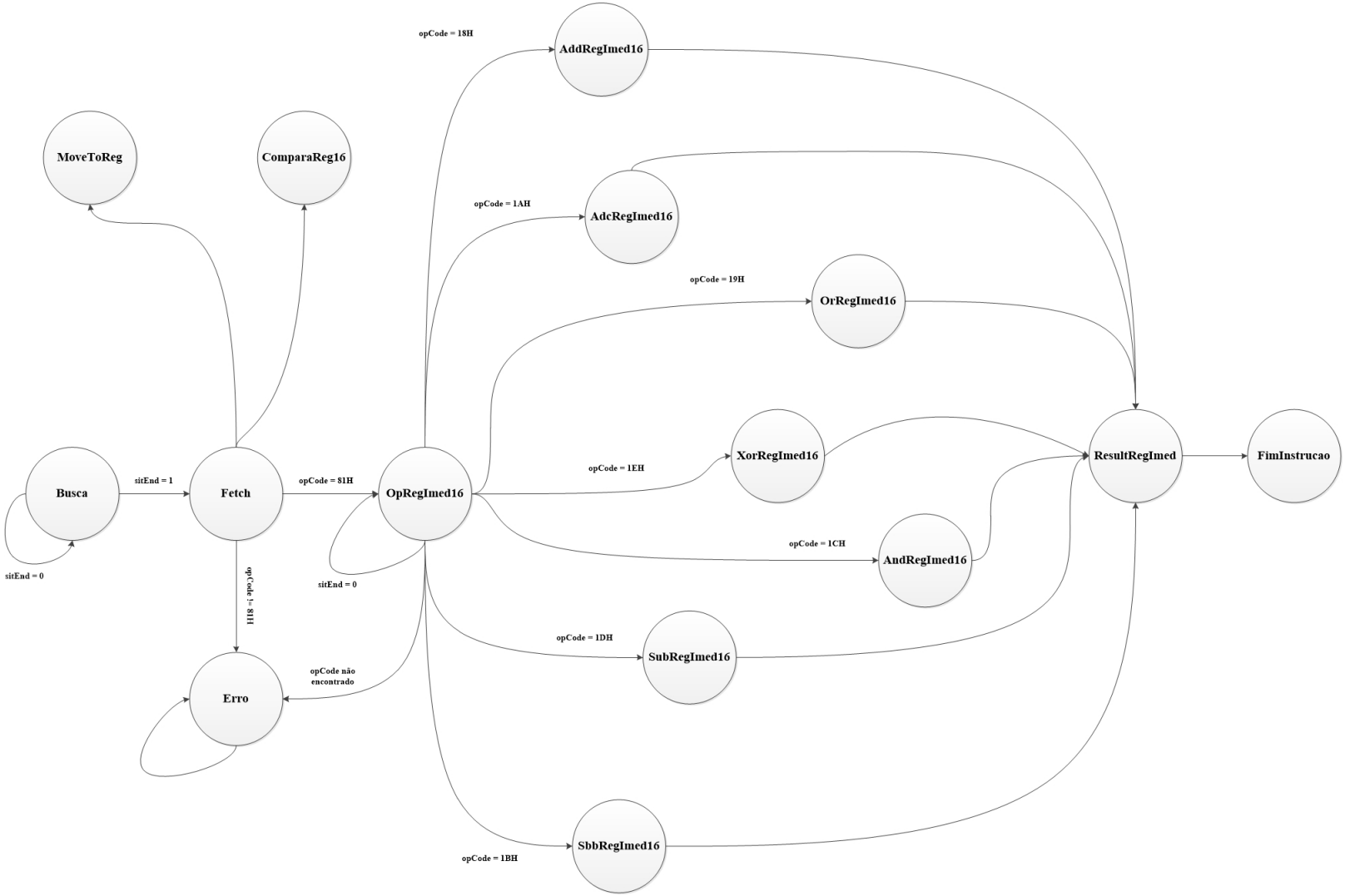

Figura 39: Diagrama de Estados da Unidade de Controle

5 Resultados

O processador implementado foi testado da seguinte maneira, foi criado um arquivo *testbench* e através dos resultados fornecidos pelo software ModelSim - Altera, selecionando somente os sinais internos de interesse da instrução, que será colocada diretamente no barramento de dados, em sua forma hexadecimal. Nas próximas seções, serão detalhadas todas as instruções implementadas, uma a uma com o seu devido resultado do *testbench*.

Temos na Figura 40, a visão RTL da estrutura de simulação que é a conexão de uma memória ROM que possui o código a ser executado e o microprocessador.

Temos na Figura 50, a visão RTL da estrutura do microprocessador completo, com todas as estruturas e ligações necessárias para o perfeito funcionamento.

5.1 ADD Reg16,Imed16

Esta instrução adiciona ao valor do registro um valor imediato de 16 bits, a instrução de teste em hexadecimal é definida como **81C0 00FF** onde 81C0h é o opcode da instrução e 00FFh é o valor imediato a ser somado no registro AX, por utilizar R/M igual a 000 como vimos anteriormente. Na figura 51 podemos ver o resultado da execução do *testbench*, com os sinais que mais nos interessam no funcionamento tanto da memória quanto do microprocessador para esta operação, as linhas de relógio e reset são responsáveis pelo funcionamento e passo a passo

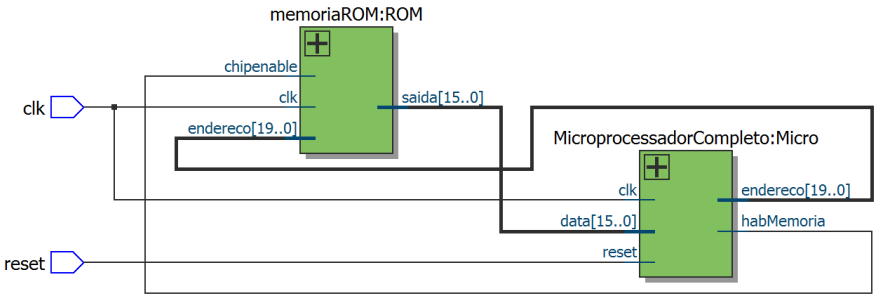

Figura 40: Visão RTL da estrutura de testes

do microprocessador, tanto que enquanto a linha de reset não fica em nível alto, o microprocessador não responde. Temos os sinais dos estados da Unidade de Controle e da Unidade de Controle de Endereço, úteis para verificar se a linha de raciocínio, presente nos grafos descritos anteriormente, está correta e respeitando a ordem de mestre e escravo imposta. Temos o endereço e a saída da memória ROM, exteriorizadas de maneira a verificar se o microprocessador está colocando no barramento de endereço o valor correto e se a memória coloca no barramento de dados o valor correto do *opcode*. Além disso, temos o enumerado descrito na Unidade Lógica e Aritmética para verificarmos qual operação é realizada dentro deste componente, podemos ver que esta linha se mantem em **op_add**, e por fim temos o registrador AX, o qual está sendo manipulado, para verificarmos a alteração em seu valor. Podemos verificar que a instrução está funcionando corretamente, pois o valor do registrador AX é alterado para **00FFh** após a execução da instrução e todos os estados são percorridos, como realmente deveriam ser em ambas unidades de controle.

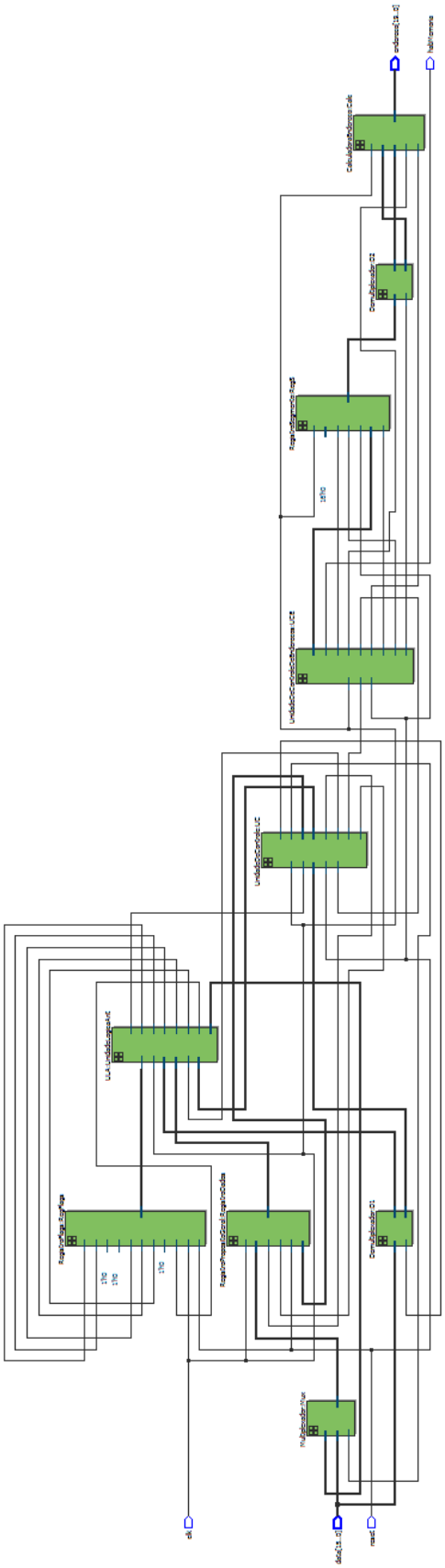

Figura 41: Visão RTL do microprocessador com todas estruturas corretas e funcionais

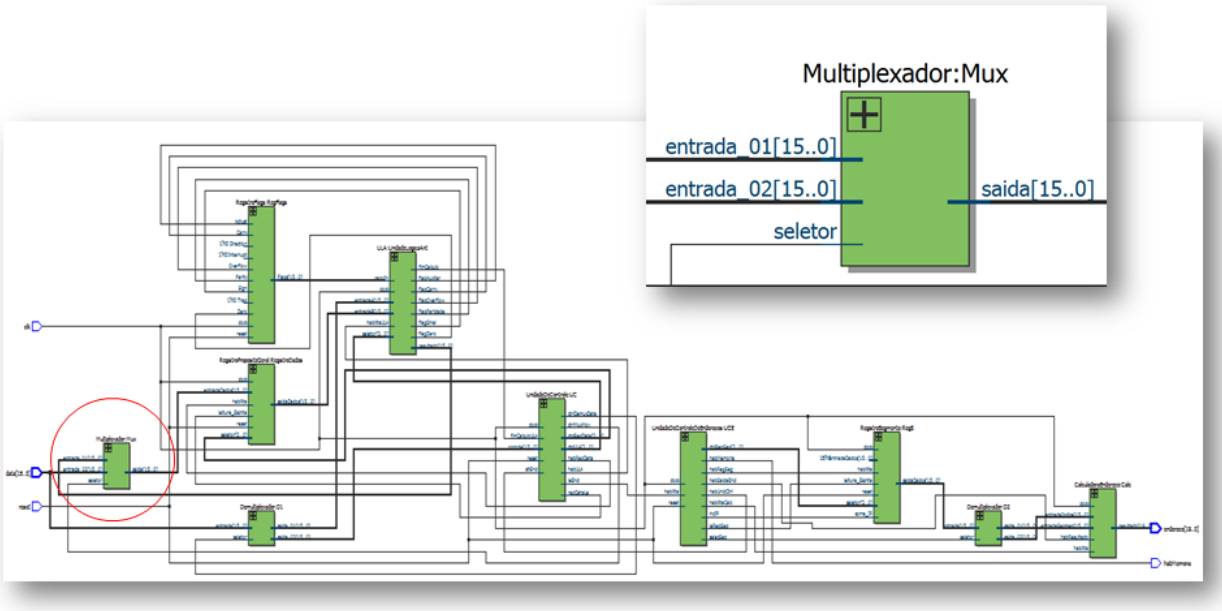

Figura 42: Visão RTL do microprocessador com foco no Multiplexador

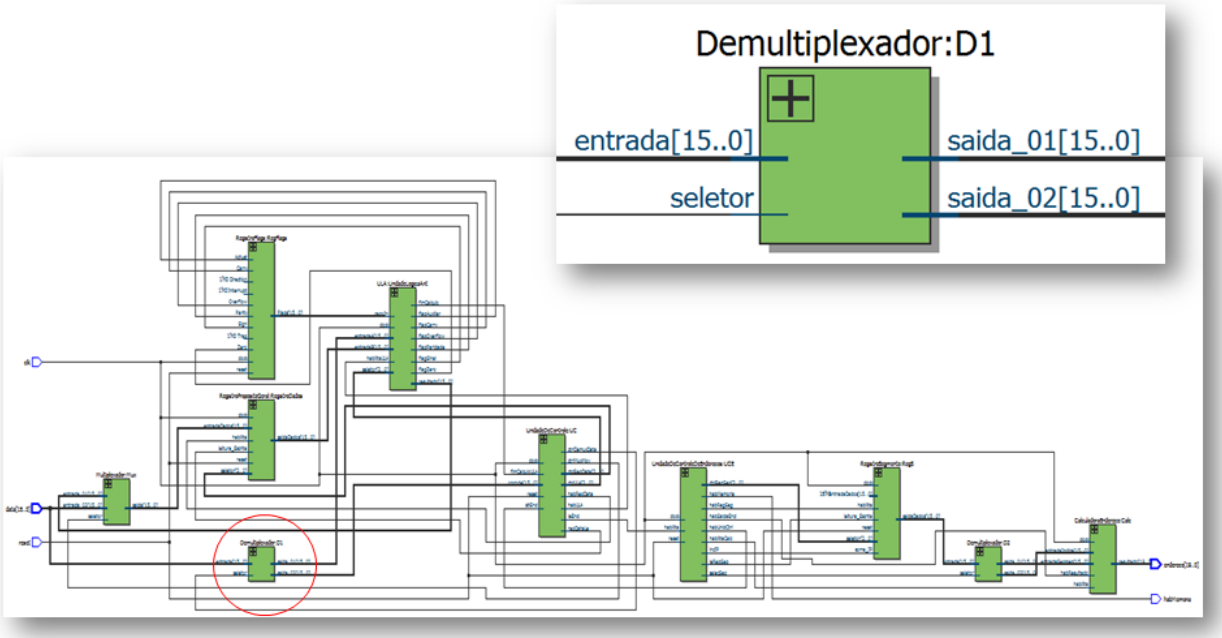

Figura 43: Visão RTL do microprocessador com foco no Demultiplexador

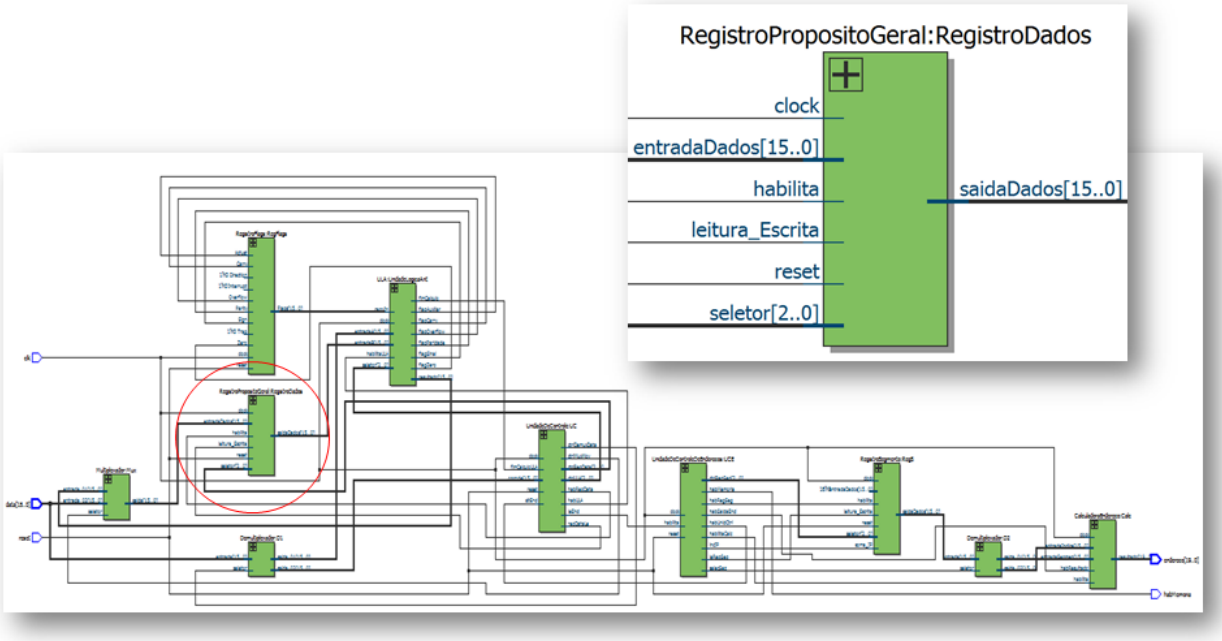

Figura 44: Visão RTL do microprocessador com foco no Registro de Dados

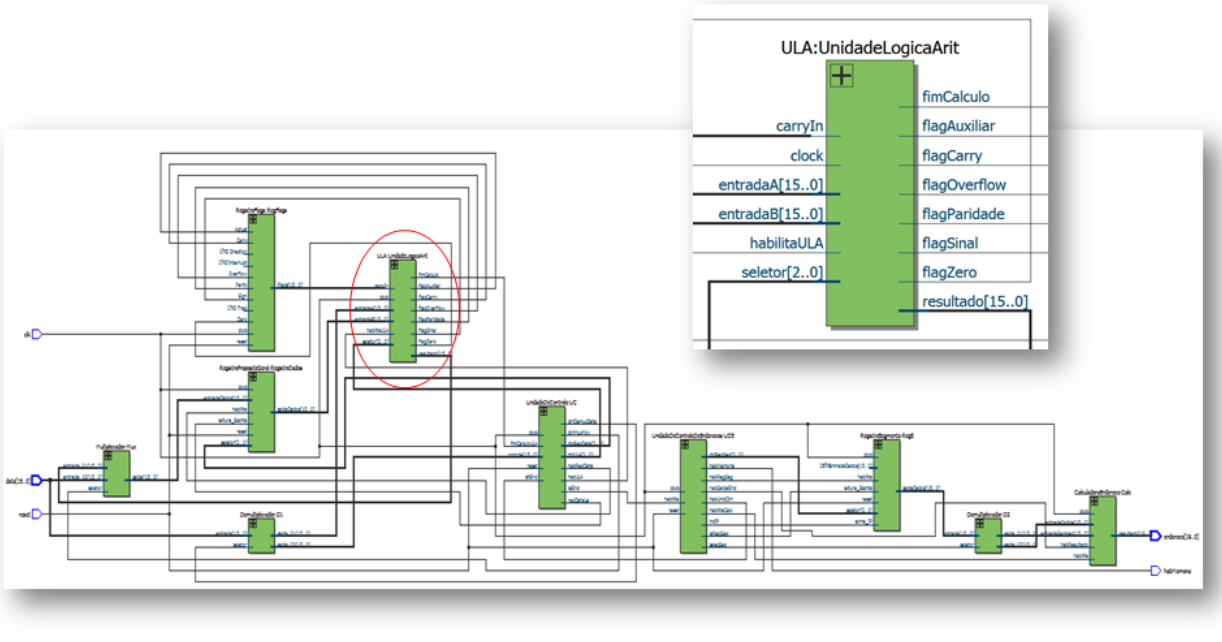

Figura 45: Visão RTL do microprocessador com foco na Unidade Lógica Aritmética

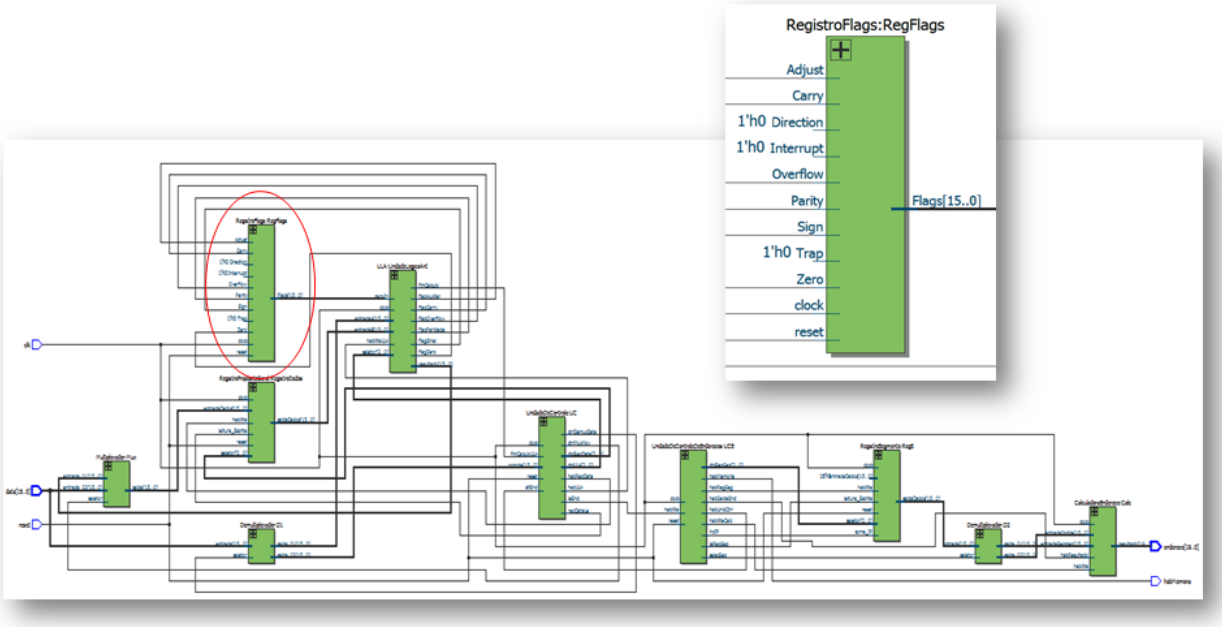

Figura 46: Visão RTL do microprocessador com foco no Registro de Flags

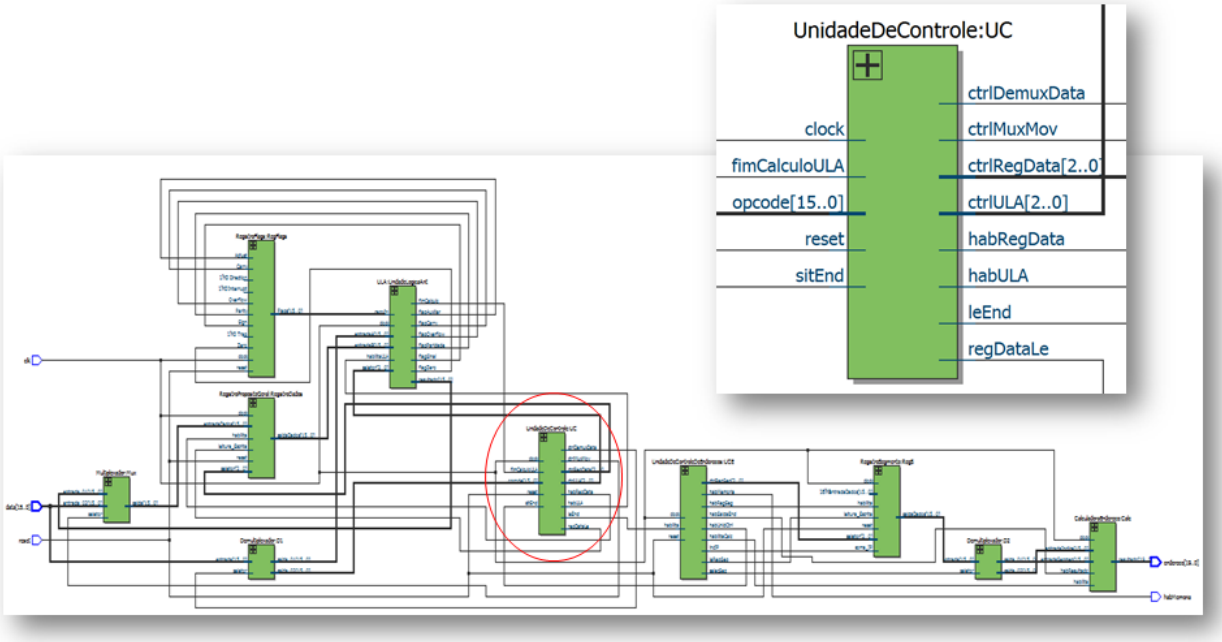

Figura 47: Visão RTL do microprocessador com foco na Unidade de Controle

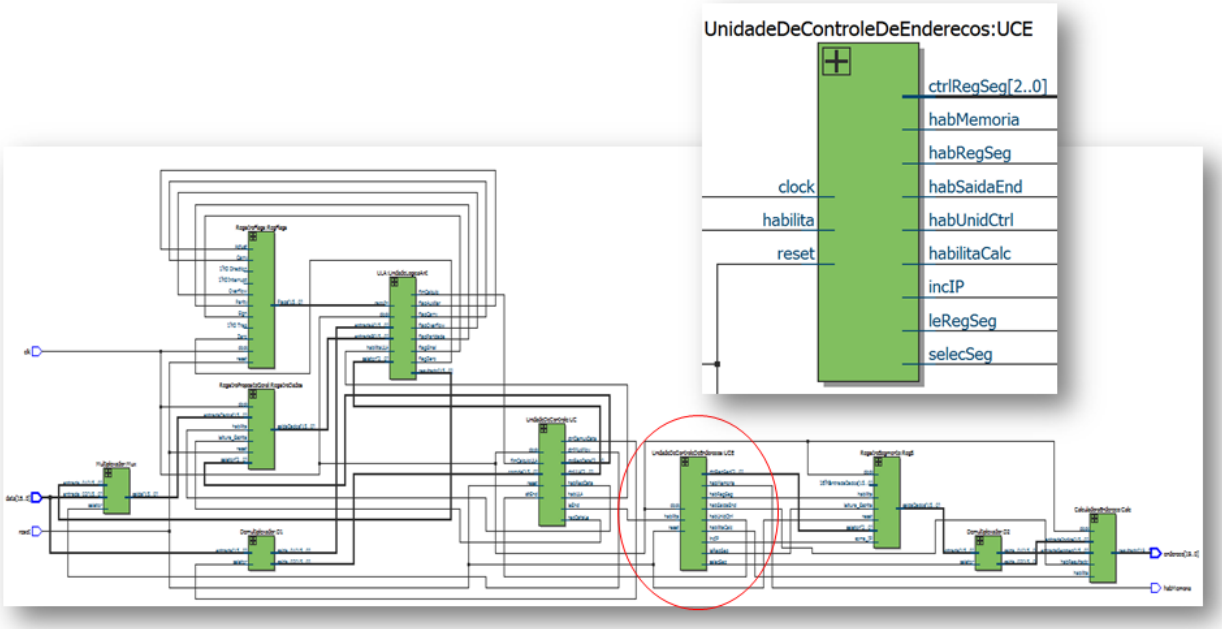

Figura 48: Visão RTL do microprocessador com foco na Unidade de Controle de Endereço

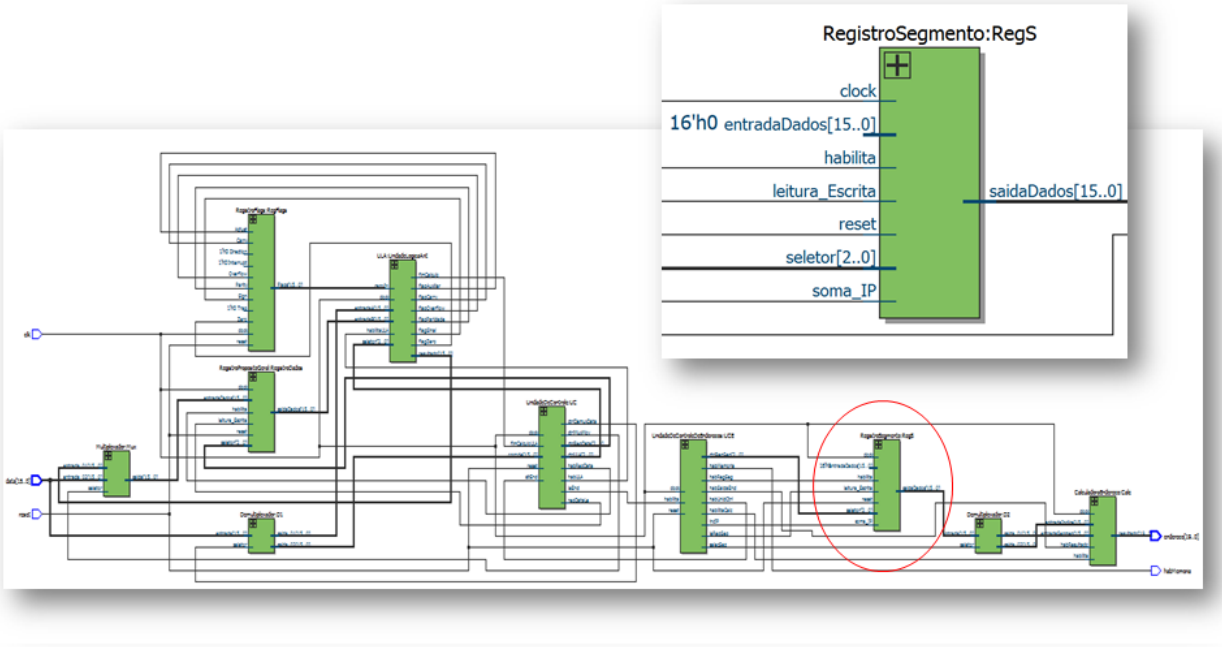

Figura 49: Visão RTL do microprocessador com foco no Registro de Segmento

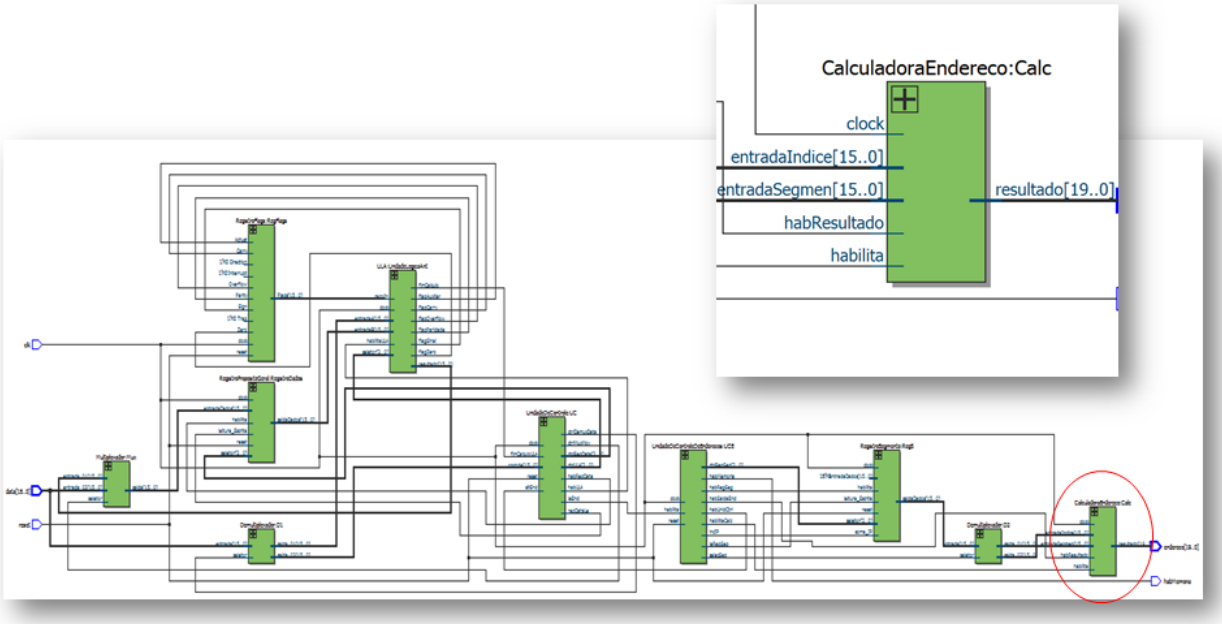

Figura 50: Visão RTL do microprocessador com foco na Calculadora de Endereço

Figura 51: Resultado Teste Operação ADD

5.2 OR Reg16,Imed16

Esta instrução realiza a operação ÖUb̈it a bit do valor do registro com um valor imediato de 16 bits, a instrução de teste em hexadecimal é definida como **81C8 1234** onde 81C8h é o opcode da instrução e 1234h é o valor imediato a ser realizado a operação ÖUño registro AX, por utilizar R/M igual a 000 como vimos anteriormente. Na figura 52 podemos ver o resultado da execução do *testbench*, com os sinais que mais nos interessam no funcionamento tanto da memória quanto do microprocessador para esta operação, as linhas de relógio e reset são responsáveis pelo funcionamento e passo a passo do microprocessador, tanto que enquanto a linha de reset não fica em nível alto, o microprocessador não responde. Temos os sinais dos estados da Unidade de Controle e da Unidade de Controle de Endereço, úteis para verificar se a linha de raciocínio, presente nos grafos descritos anteriormente, está correta e respeitando a ordem de mestre e escravo imposta. Temos o endereço e a saída da memória ROM, exteriorizadas de maneira a verificar se o microprocessador está colocando no barramento de endereço o valor correto e se a memória coloca no barramento de dados o valor correto do *opcode*. Além disso, temos o enumerado descrito na Unidade Lógica e Aritmética para verificarmos qual operação é realizada dentro deste componente, podemos ver que esta linha é alterada para **op_a_or_b**, e por fim temos o registrador AX, o qual está sendo manipulado, para verificarmos a alteração em seu valor. Podemos verificar que a instrução está funcionando corretamente, pois o valor do registrador AX é alterado para **1234h** após a execução da instrução e todos os estados são percorridos, como realmente deveriam ser em ambas unidades de controle.

Figura 52: Resultado Teste Operação OR

5.3 ADC Reg16,Imed16

Esta instrução adiciona, com carry, ao valor do registro um valor imediato de 16 bits, a instrução de teste em hexadecimal é definida como **81D0 1234** onde 81D0h é o opcode da instrução e 1234h é o valor imediato a ser somado com carry no registro AX, por utilizar R/M igual a 000 como vimos anteriormente. Na figura 53 podemos ver o resultado da execução do *testbench*, com os sinais que mais nos interessam no funcionamento tanto da memória quanto do microprocessador para esta operação, as linhas de relógio e reset são responsáveis pelo funcionamento e passo a passo do microprocessador, tanto que enquanto a linha de reset não fica em nível alto, o microprocessador não responde. Temos os sinais dos estados da Unidade de Controle e da Unidade de Controle de Endereço, úteis para verificar se a linha de raciocínio, presente nos grafos descritos anteriormente, está correta e respeitando a ordem de mestre e escravo imposta. Temos o endereço e a saída da memória ROM, exteriorizadas de maneira a verificar se o microprocessador está colocando no barramento de endereço o valor correto e se a memória coloca no barramento de dados o valor correto do *opcode*. Temos também a utilização do sinal de *carryIn* que é colocado em nível alto, em busca de verificar a diferença entre esta operação e a operação de soma sem o *carry*. Além disso, temos o enumerado descrito na Unidade Lógica e Aritmética para verificarmos qual operação é realizada dentro deste componente, podemos ver que esta linha é alterada para **op_addCarry**, e por fim temos o registrador AX, o qual está sendo manipulado, para verificarmos a alteração em seu valor. Podemos verificar que a instrução está funcionando corretamente, pois o valor do registrador AX é alterado para **1235h** após a execução da instrução e todos os estados são percorridos, como realmente deveriam ser em ambas unidades de controle.

Figura 53: Resultado Teste Operação ADC

5.4 SBB Reg16,Imed16

Esta instrução subtrai, com borrow, ao valor do registro um valor imediato de 16 bits, a instrução de teste em hexadecimal é definida como **81D8 1234** onde 81D8h é o opcode da instrução e 1234h é o valor imediato a ser subtraído com borrow no registro AX, por utilizar R/M igual a 000 como vimos anteriormente. Na figura 54 podemos ver o resultado da execução do *testbench*, com os sinais que mais nos interessam no funcionamento tanto da memória quanto do microprocessador para esta operação, as linhas de relógio e reset são responsáveis pelo funcionamento e passo a passo do microprocessador, tanto que enquanto a linha de reset não fica em nível alto, o microprocessador não responde. Temos os sinais dos estados da Unidade de Controle e da Unidade de Controle de Endereço, úteis para verificar se a linha de raciocínio, presente nos grafos descritos anteriormente, está correta e respeitando a ordem de mestre e escravo imposta. Temos o endereço e a saída da memória ROM, exteriorizadas de maneira a verificar se o microprocessador está colocando no barramento de endereço o valor correto e se a memória coloca no barramento de dados o valor correto do *opcode*. Temos também a utilização do sinal de *carryIn*, que neste caso é utilizado como borrow, é colocado em nível alto, em busca de verificar a diferença entre esta operação e a operação de subtração sem o *borrow*. Além disso, temos o enumerado descrito na Unidade Lógica e Aritmética para verificarmos qual operação é realizada dentro deste componente, podemos ver que esta linha se mantem em **op_subCarry**, e por fim temos o registrador AX, o qual está sendo manipulado, para verificarmos a alteração em seu valor. Podemos verificar que a instrução está funcionando corretamente, pois o valor do registrador AX é alterado para **EDCBh** após a execução da instrução e todos os estados são percorridos, como realmente deveriam ser em ambas unidades de controle.

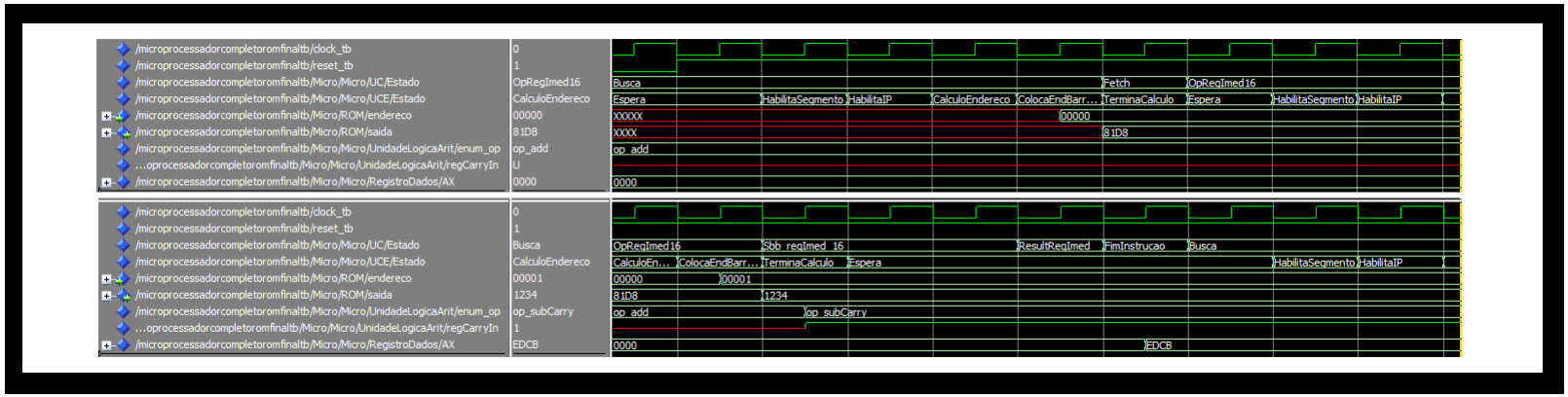

Figura 54: Resultado Teste Operação SBB

5.5 AND Reg16,Imed16

Esta instrução realiza a operação ÄNDb̈it a bit do valor do registro com um valor imediato de 16 bits, a instrução de teste em hexadecimal é definida como **81E0 1234** onde 81E0h é o opcode da instrução e 1234h é o valor imediato a ser realizado a operação ÄNDc̈om o registro AX, por utilizar R/M igual a 000 como vimos anteriormente. Na figura 55 podemos ver o resultado da execução do *test-bench*, com os sinais que mais nos interessam no funcionamento tanto da memória quanto do microprocessador para esta operação, as linhas de relógio e reset são responsáveis pelo funcionamento e passo a passo do microprocessador, tanto que enquanto a linha de reset não fica em nível alto, o microprocessador não responde. Temos os sinais dos estados da Unidade de Controle e da Unidade de Controle de Endereço, úteis para verificar se a linha de raciocínio, presente nos grafos descritos anteriormente, está correta e respeitando a ordem de mestre e escravo imposta. Temos o endereço e a saída da memória ROM, exteriorizadas de maneira a verificar se o microprocessador está colocando no barramento de endereço o valor correto e se a memória coloca no barramento de dados o valor correto do *opcode*. Além disso, temos o enumerado descrito na Unidade Lógica e Aritmética para verificarmos qual operação é realizada dentro deste componente, podemos ver que esta linha se mantem em **op_a_and_b**, e por fim temos o registrador AX, o qual está sendo manipulado, para verificarmos a alteração em seu valor. Podemos verificar que a instrução está funcionando corretamente, pois o valor do registrador AX é alterado para **0000h** após a execução da instrução e todos os estados são percorridos, como realmente deveriam ser em ambas unidades de controle.

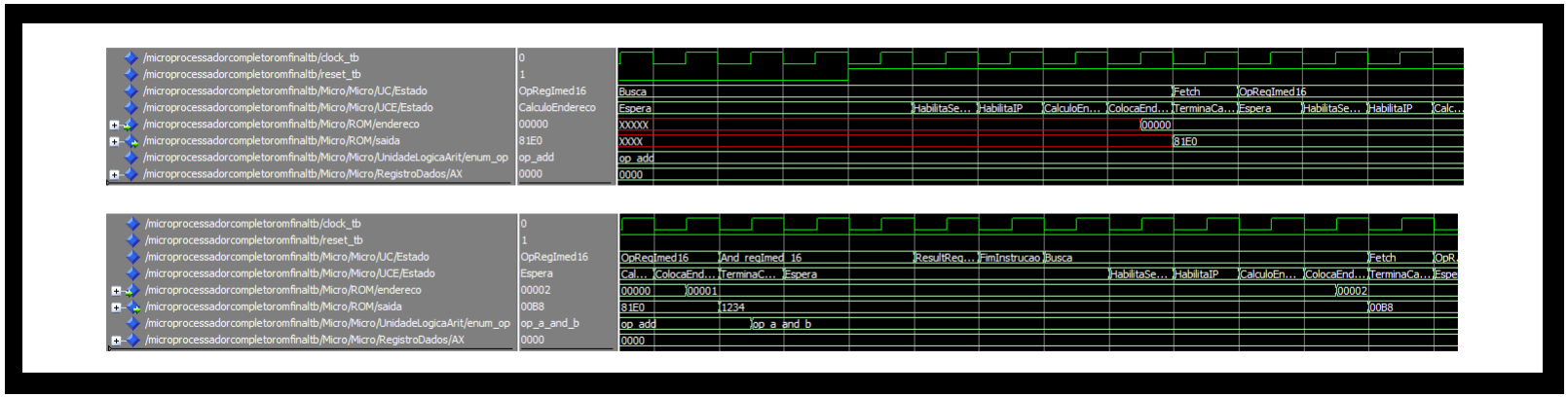

Figura 55: Resultado Teste Operação AND

5.6 SUB Reg16,Imed16

Esta instrução subtrai ao valor do registro um valor imediato de 16 bits, a instrução de teste em hexadecimal é definida como **81E8 1234** onde 81E8h é o opcode da instrução e 1234h é o valor imediato a ser subtraído no registro AX, por utilizar R/M igual a 000 como vimos anteriormente. Na figura 56 podemos ver o resultado da execução do *testbench*, com os sinais que mais nos interessam no funcionamento tanto da memória quanto do microprocessador para esta operação, as linhas de relógio e reset são responsáveis pelo funcionamento e passo a passo do microprocessador, tanto que enquanto a linha de reset não fica em nível alto, o microprocessador não responde. Temos os sinais dos estados da Unidade de Controle e da Unidade de Controle de Endereço, úteis para verificar se a linha de raciocínio, presente nos grafos descritos anteriormente, está correta e respeitando a ordem de mestre e escravo imposta. Temos o endereço e a saída da memória ROM, exteriorizadas de maneira a verificar se o microprocessador está colocando no barramento de endereço o valor correto e se a memória coloca no barramento de dados o valor correto do *opcode*. Além disso, temos o enumerado descrito na Unidade Lógica e Aritmética para verificarmos qual operação é realizada dentro deste componente, podemos ver que esta linha se mantem em **op_sub**, e por fim temos o registrador AX, o qual está sendo manipulado, para verificarmos a alteração em seu valor. Podemos verificar que a instrução está funcionando corretamente, pois o valor do registrador AX é alterado para **EDCCh** após a execução da instrução e todos os estados são percorridos, como realmente deveriam ser em ambas unidades de controle.

Figura 56: Resultado Teste Operação SUB

5.7 XOR Reg16,Imed16

Esta instrução realiza a operação ”XOR”bit a bit do valor do registro com um valor imediato de 16 bits, a instrução de teste em hexadecimal é definida como **81E8 1234** onde 81E8h é o opcode da instrução e 1234h é o valor imediato a ser realizado a operação ”XOR”com o registro AX, por utilizar R/M igual a 000 como vimos anteriormente. Temos na figura 57 o resultado da simulação.

Figura 57: Resultado Teste Operação XOR

5.8 CMP Reg16,Imed16

Esta instrução realiza a comparação ao valor do registro um valor imediato de 16 bits, a instrução de teste em hexadecimal é definida como **81F8 1234** onde 81F8h é o opcode da instrução e 1234h é o valor imediato a ser comparado com o registro AX, por utilizar R/M igual a 000 como vimos anteriormente. Temos na figura 58 o resultado da simulação.

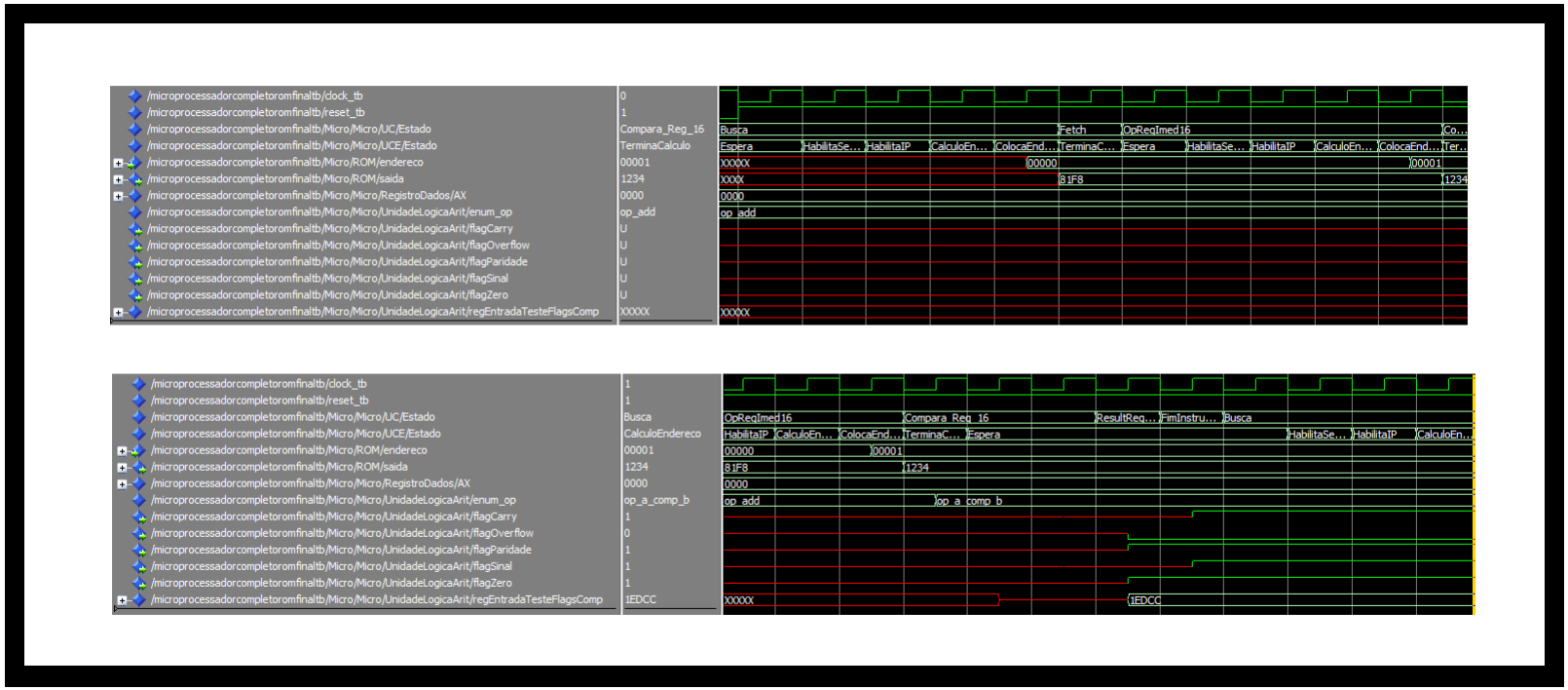

Figura 58: Resultado Teste Operação CMP

5.9 MOV Reg16,Imed16

Esta instrução move um valor imediato de 16 bits para um determinador registrador, a instrução de teste em hexadecimal é definida como **00B8 1234** onde 00B8h é o opcode da instrução e 1234h é o valor imediato a ser movido para o registro AX, por utilizar R/M igual a 000 como vimos anteriormente. Temos na figura 59 o resultado da simulação.

Figura 59: Resultado Teste Operação MOV

6 Considerações Finais

Este trabalho tinha como objetivo desenvolver um processador RISC a partir de um originalmente feito na arquitetura CISC. Sendo possível assim, analisar questões de desempenho no que diz respeito a execução dos programas e acessos à memória. Poucas instruções foram implementadas, uma vez que a arquitetura RISC possui um conjunto de instruções curto e, além disso, este é um projeto com fins didáticos sendo possível, caso necessário, a implementação de mais instruções posteriormente.

O processador desenvolvido foi testado via simulação e funcionou como esperado. As instruções implementadas desempenharam corretamente sua função e, portanto, pode-se dizer que o projeto foi concluído com êxito.

Além disso o projeto foi desenvolvido de forma modular, o que facilita a manutenção do código e posteriores melhorias. O mesmo, possui uma arquitetura de fácil entendimento o que incentiva a continuação do projeto em trabalhos futuros.

O desenvolvimento deste trabalho foi muito importante na consolidação dos conhecimentos de hardware e software. O estudo de como as diferentes arquiteturas das máquinas são implementadas foi necessário para definir o comportamento do processador aqui desenvolvido. Todo o processo de descrição do comportamento destes blocos em VHDL requer um profundo conhecimento de técnicas de depuração e desempenho na área de software e hardware.

7 Trabalhos Futuros

A máquina desenvolvida neste trabalho não teve todo o seu potencial desenvolvido. É possível desenvolver mais instruções, respeitando a arquitetura RISC, de modo a dar maior flexibilidade na hora de escrever programas para o processador executar.

Como um dos principais trabalhos futuros, fica o desenvolvimento de instruções de *jump* e *branch*, pois com essas instruções junto com as instruções aqui desenvolvidas, é possível que o microprocessador torne-se funcional.

Há também a possibilidade de aumentar o tamanho dos registros, desde que a filosofia da arquitetura utilizada seja respeitada, e melhorar o tamanho dos barramentos de comunicação, tornando o processador mais poderoso.

Além de que o microprocessador desenvolvido em nível de descrição de hardware pode ser utilizado como base para futuros trabalhos nas áreas de *pipeline*, *placement*, *routing*, entre outras pesquisas que necessitam da utilização de um microprocessador simulado de fácil compreensão. Toda a documentação deste microprocessador está em um repositório aberto no Github, https://github.com/macaufreitas/micro-risc. Assim liberamos o acesso a todos que desejam acessar qualquer documento diretamente do nosso desenvolvimento do projeto, além de liberar para a comunidade uma arquitetura de microprocessador *open source*.

Referências

ANGELOLEITHOLD. Internal integrated circuit. Http://pt.wikipedia.org/wiki/Ficheiro:InternalIntegratedCircuit2.JPG. 2004.

BINSTOCK, A. Multi-core processor architecture explained. Http://software.intel.com/en-us/articles/multi-core-processor-architecture-explained. 2013.

CISC. Cisc. Http://www.laynetworks.com/CISC.htm. 2013.

ENGINEERING, I. D. of I.; MANAGEMENT, L. Lecture 7: Microprocessor structure, assembly programming. Http://www.ielm.ust.hk/dfaculty/ajay/courses/alp/ieem110/lecs/mup/mup.html. 2013.

FERREIRA, F. B. M. e I. S. M. *Práticas com Microprocessadores*. [S.l.: s.n.], 1981.

FILHO, P. M. R. de G. N. Fundamentos de hardware. Http://www.di.ufpb.br/raimundo/ArqDI/Arq5.htm. 2013.

HENNESSY, J. L. Vlsi processor architecture. *IEEE Transactions on Computers*, C-33, p. 1221–1245, 1984.

MICROPROCESSADORES. Microprocessadores. Http://pt.kioskea.net/contents/pc/processeur.php3. 2013.

NEWELL, S. B. *Introduction to Microcomputing*. [S.l.]: John Wiley and Sons, Inc., 1989.

PATTERSON, J. L. H. D. A. *Computer Organization and Design, The Hardware/Software Interface*. Oxford, Reino Unido: Elsevier, 2005.

PEDRONI, V. A. *Eletrônica Digital Moderna e VHDL*. [S.l.]: Campus, 2011.

PUC-RIO. Puc-rio. Http://tcs.eng.br/PUC/plog/sd11-Microprogramacao.pdf. 2013.

SHUSTEK, L. Microsoft ms-dos early source code. Http://www.computerhistory.org/atchm/microsoft-ms-dos-early-source-code/. 2014.

SINGH, W. A. T. e A. *The 8088 and 8086 Microprocessors, Programming, Interfacing, Software, Hardware, and Applications*. Upper Saddle River, New Jersey, Columbus, Ohio: Pearson, 1947.

SONG, S. W. Organização de computadores - risc. Http://www.ime.usp.br/ song/mac412/oc-risc.pdf. 2003.

TARNOFF, D. *Computer Organization and Design Fundamentals*. [S.l.]: tarnoff, 2011.

TECHOPEDIA. Arithmetic logic unit (alu). Http://www.techopedia.com/definition/2849/arithmetic-logic-unit-alu. 2013.

WAITE, C. L. M. e M. *80866/8088 Manual do Microprocessador de 16 bits*. [S.l.]: McGraw-Hill, 1988.

8 Anexo - Códigos

```
-------- Bibliotecas e Pacotes --------
library ieee;
use ieee.std_logic_1164.all;
---------------------------------------

entity Multiplexador is
  port (
    entrada_01 : in  std_logic_vector (15 downto 0);
    entrada_02 : in  std_logic_vector (15 downto 0);
    saida      : out std_logic_vector (15 downto 0);
    seletor    : in  std_logic
  );
end Multiplexador;

architecture ArquiteturaMux of Multiplexador is
begin
  -- Processo que verifica mudanca na chave seletora
  process (seletor,entrada_01,entrada_02)
  begin
    if (seletor = '0') then
      saida <= entrada_01;
    else
      saida <= entrada_02;
    end if;
  end process;
end ArquiteturaMux;
```

Descrição do hardware do Multiplexador

```
-------- Bibliotecas e Pacotes --------
library ieee;
use ieee.std_logic_1164.all;
---------------------------------------

entity Demultiplexador is
  port (
    entrada : in std_logic_vector (15 downto 0);
    saida_01 : out std_logic_vector (15 downto 0);
    saida_02 : out std_logic_vector (15 downto 0);
    seletor : in std_logic
  );
end Demultiplexador;

architecture ArquiteturaDemux of Demultiplexador is
begin
  -- Processo que verifica mudanca na chave seletora
  process (seletor,entrada)
  begin
    if (seletor = '0') then
      saida_01 <= entrada;
    else
      saida_02 <= entrada;
    end if;
  end process;
end ArquiteturaDemux;
```

Descrição do hardware do Demultiplexador

```
-------- Bibliotecas e Pacotes --------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
---------------------------------------
```

```
6
entity RegistroSegmento is
8   port (
      clock           : in std_logic;
10    reset           : in std_logic;
      habilita        : in std_logic;
12    leitura_Escrita : in std_logic;
      soma_IP       : in std_logic;
14    seletor         : in std_logic_vector (2 downto 0);
      entradaDados    : in std_logic_vector (15 downto 0);
16    saidaDados      : out std_logic_vector (15 downto 0)
    );
18 end RegistroSegmento;

20 architecture ArquiteturaRS of RegistroSegmento is

22   -- Declaracao Registros
    signal ES, CS, SS, DS, IP : std_logic_vector (15 downto 0);
24
25 begin
26
    -- Processo de Reset - Escrita - Incremento IP
28  ProcessoResetEscrita : process (clock, reset, soma_IP)
    begin
30    if (reset = '0') then
        ES <= X"0000";
32      CS <= X"0000";
        SS <= X"0000";
34      DS <= X"0000";
        IP <= X"0001";
36    elsif (rising_edge (clock) and soma_IP = '1' and habilita = '1')
          then
        --Soma o IP em 4 posicoes
38      IP <= IP + X"0001";
      elsif (rising_edge (clock) and leitura_Escrita = '0' and habilita
          = '1') then
```

```
40        case seletor is
            when "000" => ES <= entradaDados;
42          when "001" => CS <= entradaDados;
            when "010" => SS <= entradaDados;
44          when "011" => DS <= entradaDados;
            when "100" => IP <= entradaDados;
46          when others =>
              ES <= (others => 'Z');
48            CS <= (others => 'Z');
              SS <= (others => 'Z');
50            DS <= (others => 'Z');
              IP <= (others => 'Z');
52        end case;
      end if;
54  end process;

56  -- Processo de Leitura
    ProcessoLeitura : process (clock)
58  begin
      if (rising_edge (clock) and leitura_Escrita = '1' and habilita =
          '1') then
60      case seletor is
            when "000" => saidaDados <= ES;
62          when "001" => saidaDados <= CS;
            when "010" => saidaDados <= SS;
64          when "011" => saidaDados <= DS;
            when "100" => saidaDados <= IP;
66          when others => saidaDados <= (others => 'Z');
        end case;
68    end if;
  end process;
70

end ArquiteturaRS;
```

Descrição do hardware do Registro de Segmento

```
1 -------- Bibliotecas e Pacotes --------
```

```
library ieee;
use ieee.std_logic_1164.all;
-----------------------------------------

entity RegistroPropositoGeral is
  port (
    clock           : in std_logic;
    reset           : in std_logic;
    habilita        : in std_logic;
    leitura_Escrita : in std_logic;
    seletor         : in std_logic_vector (2 downto 0);
    entradaDados    : in std_logic_vector (15 downto 0);
    saidaDados      : out std_logic_vector (15 downto 0)
  );
end RegistroPropositoGeral;

architecture ArquiteturaRPG of RegistroPropositoGeral is

  -- Declaracao Registros
  signal AX, BX, CX, DX, SP, BP, SI, DI : std_logic_vector (15 downto
      0);

begin

  -- Processo de Reset ou Escrita
  ProcessoResetEscrita : process (reset, clock)
  begin
    if (reset = '0') then
      AX <= X"0000";
      BX <= X"0000";
      CX <= X"0000";
      DX <= X"0000";
      SP <= X"0000";
      BP <= X"0000";
      SI <= X"0000";
      DI <= X"0000";
```

```
37      elsif (rising_edge(clock) and leitura_Escrita = '0' and habilita
           = '1') then
        case seletor is
39        when "000" => AX <= entradaDados;
          when "011" => BX <= entradaDados;
41        when "001" => CX <= entradaDados;
          when "010" => DX <= entradaDados;
43        when "100" => SP <= entradaDados;
          when "101" => BP <= entradaDados;
45        when "110" => SI <= entradaDados;
          when "111" => DI <= entradaDados;
47        when others =>
            AX <= (others => 'Z');
49          BX <= (others => 'Z');
            CX <= (others => 'Z');
51          DX <= (others => 'Z');
            SP <= (others => 'Z');
53          BP <= (others => 'Z');
            SI <= (others => 'Z');
55          DI <= (others => 'Z');
        end case;
57    end if;
  end process;
59
  -- Processo de Leitura
61  ProcessoLeitura : process (clock, leitura_Escrita)
  begin
63    if (rising_edge (clock) and leitura_Escrita = '1' and habilita =
         '1') then
        case seletor is
65        when "000" => saidaDados <= AX;
          when "011" => saidaDados <= BX;
67        when "001" => saidaDados <= CX;
          when "010" => saidaDados <= DX;
69        when "100" => saidaDados <= SP;
          when "101" => saidaDados <= BP;
```

```
71          when "110" => saidaDados <= SI;
            when "111" => saidaDados <= DI;
73          when others => saidaDados <= (others => 'Z');
        end case;
75      end if;
    end process;
77
end ArquiteturaRPG;
```

Descrição do hardware do Registro de Propósito Geral

```
-------- Bibliotecas e Pacotes -------
2 library ieee;
  use ieee.std_logic_1164.all;
4 --------------------------------------

6 entity RegistroFlags is
    port (
8       reset         : in std_logic;
        clock         : in std_logic;
10      Overflow      : in std_logic;
        Direction     : in std_logic;
12      Interrupt     : in std_logic;
        Trap          : in std_logic;
14      Sign          : in std_logic;
        Zero          : in std_logic;
16      Adjust        : in std_logic;
        Parity        : in std_logic;
18      Carry         : in std_logic;
        Flags       : out std_logic_vector(15 downto 0)
20  );
  end RegistroFlags;
22
  architecture ArquiteturaRF of RegistroFlags is
24 begin
    -- Processo de Escrita e Reset
26  ProcessoResetEscrita : process (reset, clock)
```

```
  begin
    if (reset = '0') then
      Flags <= (others => '0');
    elsif (rising_edge(clock)) then
      Flags(11) <= Overflow;
      Flags(10) <= Direction;
      Flags(9)  <= Interrupt;
      Flags(8)  <= Trap;
      Flags(7)  <= Sign;
      Flags(6)  <= Zero;
      Flags(4)  <= Adjust;
      Flags(2)  <= Parity;
      Flags(0)  <= Carry;
    end if;
  end process;
end ArquiteturaRF;
```

Descrição do hardware do Registro de Flags

```
-------- Bibliotecas e Pacotes --------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
---------------------------------------

entity CalculadoraEndereco is
  port
  (
    clock          : in std_logic;
    habilita       : in std_logic;
    habResultado   : in std_logic;
    entradaIndice  : in std_logic_vector(15 downto 0);
    entradaSegmen  : in std_logic_vector(15 downto 0);
    resultado      : out std_logic_vector(19 downto 0)
  );
```

```
20 end entity;

22 architecture rtl of CalculadoraEndereco is

24   --Declaracao dos registros
     signal regS,regI,regResult : std_logic_vector(19 downto 0) := (
         others => '0');
26
   begin
28
     ProcessoCalculo : process(clock,habilita)
30   begin
       if(rising_edge(clock) and (habilita ='1')) then
32       regI <= ("0000" & entradaIndice);
         regS <= (entradaSegmen & "0000");
34       regResult <= regS + regI;
       end if;
36   end process;

38   resultado <= regResult when habResultado = '1';

40 end rtl;
```

Descrição do hardware da Calculadora de Endereços

```
   -------- Bibliotecas e Pacotes -------
2  library ieee;
   use ieee.std_logic_1164.all;
4  use ieee.std_logic_arith.all;
   ---------------------------------------
6
   -------------------------------------------------------
8  -- seletorOperacao = 0 ==> Operacao de Adicao -------
   -- seletorOperacao = 1 ==> Operacao de Subtracao ----
10 -------------------------------------------------------
```

```
12 entity DetectorAuxiliarFlag is
     port(
14     entrada_01 : in std_logic_vector(15 downto 0);
       entrada_02 : in std_logic_vector(15 downto 0);
16     carry : in std_logic;
       seletorOperacao : in std_logic_vector(2 downto 0);
18     estadoFlagAuxiliar : out std_logic
     );
20 end DetectorAuxiliarFlag;

22 architecture ArquiteturaDAF of DetectorAuxiliarFlag is

24   -- Instancia do Somador
     component Somador
26     port(
         entradaA : in std_logic;
28       entradaB : in std_logic;
         carryIn  : in std_logic;
30       carryOut : out std_logic;
         saida    : out std_logic
32     );
     end component;
34
     -- Instancia do Subtrator
36   component Subtrator
       port(
38       entradaA  : in std_logic;
         entradaB  : in std_logic;
40       borrowIn  : in std_logic;
         borrowOut : out std_logic;
42       saida     : out std_logic
       );
44   end component;

46   -- Sinais auxiliares
     signal c0, c1, c2, c3, c4 : std_logic;
```

```
48    signal b0, b1, b2, b3, b4 : std_logic;
      signal ra0, ra1, ra2, ra3, ra4 : std_logic;
50    signal rs0, rs1, rs2, rs3, rs4 : std_logic;

52  begin

54    Add0: Somador port map(entrada_01(0), entrada_02(0), carry, c0, ra0
          );
      Add1: Somador port map(entrada_01(1), entrada_02(1), c0, c1, ra1);
56    Add2: Somador port map(entrada_01(2), entrada_02(2), c1, c2, ra2);
      Add3: Somador port map(entrada_01(3), entrada_02(3), c2, c3, ra3);
58    Add4: Somador port map(entrada_01(4), entrada_02(4), c3, c4, ra4);

60    Sub0: Subtrator port map(entrada_01(0), entrada_02(0), carry, b0,
          rs0);
      Sub1: Subtrator port map(entrada_01(1), entrada_02(1), b0, b1, rs1)
          ;
62    Sub2: Subtrator port map(entrada_01(2), entrada_02(2), b1, b2, rs2)
          ;
      Sub3: Subtrator port map(entrada_01(3), entrada_02(3), b2, b3, rs3)
          ;
64    Sub4: Subtrator port map(entrada_01(4), entrada_02(4), b3, b4, rs4)
          ;

66    process(entrada_01, entrada_02, seletorOperacao, b3, c3)
      begin
68      case seletorOperacao is
          when "000" =>
70          estadoFlagAuxiliar <= c3;
          when "001" =>
72          estadoFlagAuxiliar <= b3;
          when others =>
74          estadoFlagAuxiliar <= 'Z';
        end case;
76    end process;
```

```
78 end ArquiteturaDAF;
```

Descrição do hardware do Detector de Flag Auxiliar

```
1  -------- Bibliotecas e Pacotes --------
   library ieee;
3  use ieee.std_logic_1164.all;
   ---------------------------------------
5
   entity DetectorParidade is
7    port (
       entrada : in std_logic_vector(15 downto 0);
9      saida : out std_logic
     );
11 end DetectorParidade;

13 architecture ArquiteturaDP of DetectorParidade is
   begin
15   ProcessoParidade : process(entrada)
     begin
17     saida <= not(entrada(15) xor entrada(14) xor entrada(13) xor
           entrada(12) xor entrada(11) xor entrada(10) xor
                entrada(9) xor entrada(8) xor entrada(7) xor entrada(6)
                   xor entrada(5) xor entrada(4) xor
19              entrada(3) xor entrada(2) xor entrada(1) xor entrada(0));
     end process;
21 end ArquiteturaDP;
```

Descrição do hardware do Detector de Flag de Paridade

```
1  -------- Bibliotecas e Pacotes --------
   library ieee;
3  use ieee.std_logic_1164.all;
   ---------------------------------------
5
   entity DetectorZeroFlag is
7    port(
       resultado : in std_logic_vector(15 downto 0);
```

```
    saida : out std_logic
  );
end DetectorZeroFlag;

architecture ArquiteturaDZF of DetectorZeroFlag is
begin
  process(resultado)
  begin
    saida <= not(resultado(0) or resultado(1) or resultado(2) or
        resultado(3) or resultado(4) or
              resultado(5) or resultado(6) or resultado(7) or
                 resultado(8) or resultado(9) or
              resultado(10) or resultado(11) or resultado(12) or
                 resultado(13) or
              resultado(14) or resultado(15));
  end process;
end ArquiteturaDZF;
```

Descrição do hardware do Detector de Zero Flag

```
-------- Bibliotecas e Pacotes -------
library ieee;
library work;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
--------------------------------------

entity ULA is
  port (
    clock            : in  std_logic;
    seletor           : in  std_logic_vector(2 downto 0); -- seletor
       de operacoes
    carryIn          : in  std_logic;
    entradaA         : in  std_logic_vector(15 downto 0); -- entrada
       de dados A
```

```
16      entradaB         : in  std_logic_vector(15 downto 0); -- entrada
            de dados B
        habilitaULA      : in  std_logic;
18      flagCarry       : out std_logic;
        flagOverflow     : out std_logic;
20      flagParidade     : out std_logic;
        flagSinal        : out std_logic;
22      flagZero         : out std_logic;
        flagAuxiliar     : out std_logic;
24      resultado        : out std_logic_vector(15 downto 0);
        fimCalculo       : out std_logic
26    );
  end ULA;
28
  architecture arquiteturaULA of ULA is
30
    -- tipos possÃveis de operacoes
32    type op_type is (op_add,op_a_or_b,op_a_and_b,op_addCarry,
          op_subCarry,
                    op_sub,op_a_xor_b,op_nop, op_a_comp_b);
34
    signal enum_op : op_type;
36    signal op : std_logic_vector(2 downto 0);

38    --sinais de registro dos valores de entrada e calculado
     signal reg, regEntradaTesteFlags, regEntradaTesteFlagsComp :
          std_logic_vector(16 downto 0);
40    signal regA,regB : std_logic_vector(15 downto 0);
    signal regCarryIn : std_logic;
42
    component DetectorZeroFlag
44      port(
        resultado : in std_logic_vector(15 downto 0);
46      saida : out std_logic
      );
48    end component;
```

```
50    component DetectorParidade
        port (
52        entrada : in std_logic_vector(15 downto 0);
          saida : out std_logic
54      );
    end component;
56
    component DetectorAuxiliarFlag
58      port(
          entrada_01 : in std_logic_vector(15 downto 0);
60        entrada_02 : in std_logic_vector(15 downto 0);
          carry : in std_logic;
62        seletorOperacao : in std_logic_vector(2 downto 0);
          estadoFlagAuxiliar : out std_logic
64      );
    end component;
66
  begin
68
    --Processo que determina qual operacao deve ser realizada
70  ProcessoDetermina : process(seletor)
     begin
72    if(habilitaULA = '1') then
        regCarryIn <= '1'; -- Inicializacao do registro de carry
74      case seletor is
          when "000"  => enum_op <= op_add;
76        when "001"  => enum_op <= op_a_or_b;
          when "010"  => enum_op <= op_addCarry;
78        when "011"  => enum_op <= op_subCarry;
          when "100"  => enum_op <= op_sub;
80        when "101"  => enum_op <= op_a_xor_b;
          when "110"  => enum_op <= op_a_and_b;
82        when "111"  => enum_op <= op_a_comp_b;
          when others => enum_op <= op_nop;
84      end case;
```

```
    end if;
86  end process;

88  --Processo que efetivamente realiza as operacoes
    ProcessoCalcula : process(clock)
90  begin
      if (rising_edge(clock) and (habilitaULA = '1')) then
92    fimCalculo <= '0';
      regA <= entradaA;
94    regB <= entradaB;
        case enum_op is
96      when op_add =>
          regEntradaTesteFlags <= ('0' & regA) + regB;
98        flagCarry <= regEntradaTesteFlags(16);
          flagOverflow <= (regEntradaTesteFlags(16) xor entradaA(15)
              xor entradaB(15) xor
100                   regEntradaTesteFlags(15)) ;
          flagSinal <= regEntradaTesteFlags(15);
102       op <= "000";
        when op_sub =>
104       regEntradaTesteFlags <= ('0' & regB) - regA;
          flagCarry <= regEntradaTesteFlags(16);
106       flagOverflow <= ((not regA(15)) and regB(15) and
              regEntradaTesteFlags(15)) or
                  (regA(15) and (not regB(15)) and (not
                      regEntradaTesteFlags(15)));
108       flagSinal <= regEntradaTesteFlags(15);
          op <= "001";
110     when op_a_or_b =>
          regEntradaTesteFlags <= '0' & (regA or regB);
112       flagCarry <= '0';
          flagOverflow <= '0';
114       flagSinal <= regEntradaTesteFlags(15);
          op <= "010";
116     when op_addCarry =>
          regEntradaTesteFlags <= ('0' & regA) + regB + regCarryIn;
```

```
        --Denis
118       flagCarry <= regEntradaTesteFlags(16);
          flagOverflow <= regEntradaTesteFlags(16) xor regA(15) xor
              regB(15) xor regEntradaTesteFlags(15);
120       flagSinal <= regEntradaTesteFlags(15);
          op <= "000";
122     when op_subCarry =>
          regEntradaTesteFlags <= ('0' & regB) - regA - regCarryIn;
124       flagCarry <= regEntradaTesteFlags(16);
          flagOverflow <= ((not regA(15)) and regB(15) and
              regEntradaTesteFlags(15)) or
126                   (regA(15) and (not regB(15)) and (not
                          regEntradaTesteFlags(15)));
          flagSinal <= regEntradaTesteFlags(15);
128       op <= "001";
        when op_a_xor_b =>
130       regEntradaTesteFlags <= '0' & (regA xor regB);
          flagCarry <= '0';
132       flagOverflow <= '0';
          flagSinal <= regEntradaTesteFlags(15);
134       op <= "011";
        when op_nop =>
136       reg(15) <= '0';
        when op_a_comp_b =>
138       regEntradaTesteFlagsComp <= ('0' & regB) - regA;
          flagCarry <= regEntradaTesteFlagsComp(16);
140       flagOverflow <= ((not regA(15)) and regB(15) and
              regEntradaTesteFlagsComp(15)) or
                      (regA(15) and (not regB(15)) and (not
                          regEntradaTesteFlagsComp(15)));
142       flagSinal <= regEntradaTesteFlagsComp(15);
          op <= "100";
144       regEntradaTesteFlags <= '0' & regB; --Denis
        when op_a_and_b =>
146       regEntradaTesteFlags <= '0' & (regA and regB);
          flagCarry <= '0';
```

```
148             flagOverflow <= '0';
                flagSinal <= regEntradaTesteFlags(15);
150             op <= "010";
            when others =>
                reg <= (others => 'Z');
        end case;
154     fimCalculo <= '1';
        end if;
156   end process;

158   --atribuicao do resultado
    resultado <= regEntradaTesteFlags(15 downto 0);
160
    --atribuicao dos resultados finais
162   dParidade : DetectorParidade port map (regEntradaTesteFlags(15
        downto 0), flagParidade);
    dAuxiliarFlag : DetectorAuxiliarFlag port map (entradaA, entradaB,
        carryIn, op, flagAuxiliar);
164   dZeroFlag : DetectorZeroFlag port map (regEntradaTesteFlags(15
        downto 0), flagZero);

166 end arquiteturaULA;
```

Descrição do hardware da Unidade Lógica Aritmética - ULA

```
  library ieee;
2 use ieee.std_logic_1164.all;

4 -- Pacote auxiliar a unidade de controle de memoria, facilitar a
    escrita e visualizacao do codigo

6 package uce_aux is

8   --Possiveis estados da unidade de controle
    type Tipo_estado is ( Espera, HabilitaSegmento, HabilitaIP,
10                        CalculoEndereco, ColocaEndBarramentoIncIP,
                          TerminaCalculo, Erro);
```

```
12 end uce_aux;
```

Pacote auxiliar que descreve os estados da Unidade de Controle de Endereços

```
-------- Bibliotecas e Pacotes --------
library ieee;
use ieee.std_logic_1164.all;
use work.uce_aux.all;
---------------------------------------

entity UnidadeDeControleDeEnderecos is
  port(
      clock          : in  std_logic;
      reset          : in  std_logic;
      habilita       : in  std_logic;
      habilitaCalc   : out std_logic;
      habRegSeg    : out std_logic;
      leRegSeg       : out std_logic;
      ctrlRegSeg     : out std_logic_vector(2 downto 0);
      incIP        : out std_logic;
      selecSeg       : out std_logic;
      habSaidaEnd    : out std_logic;
      habMemoria     : out std_logic;
      habUnidCtrl    : out std_logic
  );
end UnidadeDeControleDeEnderecos;

architecture Arquitetura of UnidadeDeControleDeEnderecos is
  signal Estado          : Tipo_estado; --Estado da unidade de
     controle

begin

--Definicao do processo de borda de subida que altera os sinais
borda_subida : process(clock, reset)
begin

```

```
  if reset = '0' then
34 -- todas saidas zeradas
    habilitaCalc <= '0';
36  leRegSeg    <= '1';
    ctrlRegSeg    <= "000";
38  incIP        <= '0';
    selecSeg      <= '0';
40  habSaidaEnd  <= '0';
    habRegSeg     <= '1';
42  habMemoria    <= '0';
    habUnidCtrl  <= '0';
44

  elsif rising_edge(clock) then
46

    case Estado is
48

      when Espera =>
50    -- Estado nao modifica seus sinais, somente aguarda
        habUnidCtrl <= '0'; -- Faz a unidade de controle esperar
            tambÃ©m
52

      when HabilitaSegmento =>
54    -- Estado habilita o valor do segmento para ser calculado o
          endereco pela calculadora de enderecos
        habilitaCalc <= '1'; -- Habilita a calculado de enderecos
56      habRegSeg  <= '1'; -- Habilita o registro de segmentos
        selecSeg      <= '1'; -- Seleciona o registro de segmento para
            ir para a calculadora de endereco
58      ctrlRegSeg    <= "001"; -- Seleciona como registro de segmento
            o CS
        leRegSeg      <= '1'; -- Le o registro de segmento
60


62    when HabilitaIP =>
      --Estado que habilida o valor do registro IP para ser calculado
          o endereco pela calculadora de enderecos
```

```
64          selecSeg    <= '0'; -- seleciona a outra entrada da
                calculadora
            ctrlRegSeg <= "100"; -- seleciona o IP do registro de
                segmento
66
        when CalculoEndereco =>
68      --Estado que realiza o calculo do endereco

70      when ColocaEndBarramentoIncIP =>
        --Estado em que a unidade coloca o endereco no barramento
72          habSaidaEnd <= '1'; -- habilita a saida da calculadora de
                endereco para o IP
            leRegSeg      <= 'Z'; -- para de ler a unidade de registro
74          incIP         <= '1'; -- Envia o sinal para incrementar o IP
            habUnidCtrl <= '1'; -- Avisa a Unidade de Controle que o dado
                 que vai ser colocado no barramento de dados pela memoria
                Ã© um dado vÃ¡lido
76          habMemoria    <= '1'; -- Habilita a memoria

78      when TerminaCalculo =>
        --Estado em que termina o calculo do endereco
80          habilitaCalc <= '0';
            habSaidaEnd  <= '0';
82          incIP        <= '0';
            habRegSeg    <= '0';
84          habMemoria    <= '0';

86      when Erro =>
          -- todas saidas em alta impedancia
88          habilitaCalc <= 'Z';
             leRegSeg     <= 'Z';
90           ctrlRegSeg   <= "ZZZ";
             incIP        <= 'Z';
92          selecSeg      <= 'Z';
          --Trava a MÃ¡quina de Estados
94    end case;
```

```
    end if;
96 end process;

98 --definicao do processo de borda de descida que toma a decisao para o
       proximo estado
   borda_descida : process(clock,reset)
100 begin

102   if (reset = '0') then

104     --Mantem em estado de busca
      Estado <= Espera;
106
    elsif falling_edge(clock) then
108
      --Troca os estados
110     case Estado is

112       --Caso estado de Busca, passa para a decodificao ( Fecth )
        when Espera =>
114       if (habilita = '0') then
            Estado <= Espera;
116       else
            Estado <= HabilitaSegmento;
118       end if;

120       --Apos habilitar o Segmento, habilita o IP
        when HabilitaSegmento => Estado <= HabilitaIP;
122
        --Apos habilitar o IP, realiza o calculo do endereco
124     when HabilitaIP => Estado <= CalculoEndereco;

126     --Apos o calculo do endereco coloca o valor no barramento de
          endereco
        when CalculoEndereco => Estado <= ColocaEndBarramentoIncIP;
128
```

```
        --Apos colocar o endereco no barramento, termina o ciclo de
            calculo do Endereco
130     when ColocaEndBarramentoIncIP => Estado <= TerminaCalculo;

132     --Apos calculo terminado, retorna para o estado de espera
        when TerminaCalculo => Estado <= Espera;
134
        --Caso algum erro ocorra no percorrer da instrucao a maquina
            trava
136     when Erro    => Estado <= Erro;

138   end case;
    end if;
140 end process;

142 end Arquitetura;
```

Descrição do hardware da Unidade de Controle de Endereços

```
library ieee;
2 use ieee.std_logic_1164.all;

4 -- Pacote auxiliar a unidade de controle, facilitar a escrita e
    visualizacao do codigo

6 package uc_aux is

8   --Possiveis estados da unidade de controle
    type Tipo_estado is ( Busca, Fetch, OpRegImed16,
10                  Add_regImed_16, Sub_regImed_16, Erro,
                    Or_regImed_16, Adc_regImed_16, Sbb_regImed_16,
12                  And_regImed_16, Xor_regImed_16,
                    FimInstrucao, ResultRegImed, Compara_Reg_16,
                        Move_To_Reg, Fim_Move_To_Reg);
14 end uc_aux;
```

Pacote auxiliar que descreve os estados da Unidade de Controle

```
-------- Bibliotecas e Pacotes -------
library ieee;
use ieee.std_logic_1164.all;
use work.uc_aux.all;
--------------------------------------

entity UnidadeDeControle is
  port(
      clock          : in  std_logic;
      reset          : in  std_logic;
      opcode         : in  std_logic_vector(15 downto 0);
      sitEnd         : in  std_logic;
      leEnd          : out std_logic;
      habULA       : out std_logic;
      ctrlULA        : out std_logic_vector(2 downto 0);
      regDataLe       : out std_logic;
      habRegData      : out std_logic;
      ctrlRegData    : out std_logic_vector(2 downto 0);
      ctrlDemuxData   : out std_logic;
      fimCalculoULA : in  std_logic;
      ctrlMuxMov     : out std_logic
  );
end UnidadeDeControle;

architecture Arquitetura of UnidadeDeControle is
  signal Estado          : Tipo_estado; --Estado da unidade de
     controle
  signal opcodeFetch     : std_logic_vector(15 downto 0); --Opcode a
     ser decodificado

begin

--Definicao do processo de borda de subida que altera os sinais
borda_subida : process(clock, reset)
begin
```

```
35  if reset = '0' then

37  -- todas saidas zeradas
      leEnd        <= '0';
39    habULA       <= '0';
      ctrlULA        <= "000";
41    regDataLe    <= '0';
      habRegData     <= '0';
43    ctrlRegData  <= "000";
      ctrlDemuxData  <= '0';   --Dado vai direto para a ULA
45    ctrlMuxMov   <= '0';

47  elsif rising_edge(clock) then

49    case Estado is

51      when Busca =>
          -- Pede para a Unidade de Controle de Enderecos um opcode
              valido
53        leEnd  <= '1';

55      when Fetch =>
          -- Para de ler a fila
57        leEnd      <= '0';
          -- Direciona o dado da fila de instrucoes para a Unidade de
              Controle
59        ctrlDemuxData <= '1';

61      when OpRegImed16 =>
          -- Operacao de Reg/Imediato de 16 bits
63        -- Habilita a leitura da fila para ler mais 16 bits
          leEnd        <= '1';
65        -- Seta o controle do Demux para direcionar o dado direto
              para a ALU
          ctrlDemuxData <= '0';
67
```

```
        when Add_regImed_16 =>
69        -- Habilita a ALU para gravar o dado
          habULA         <= '1';
71        -- Operacao de adicao
          ctrlULA      <= b"000"; --ADD
73        -- Habilita o registro de Dados
          habRegData <= '1';
75        -- Seta o registro para ler o dado
          regDataLe  <= '1';
77        -- Passa para o controle do registro de dados o registro a
              ser utilizado
          ctrlRegData <= opcodeFetch(2 downto 0);
79        -- Fila para de enviar o valor
          leEnd      <= '0';
81
        when Or_regImed_16 =>
83        -- Habilita a ALU para gravar o dado
          habULA         <= '1';
85        -- Operacao de ou
          ctrlULA      <= b"001"; --OR
87        -- Habilita o registro de Dados
          habRegData <= '1';
89        -- Seta o registro para ler o dado
          regDataLe  <= '1';
91        -- Passa para o controle do registro de dados o registro a
              ser utilizado
          ctrlRegData <= opcodeFetch(2 downto 0);
93        -- Fila para de enviar o valor
          leEnd      <= '0';
95
        when Adc_regImed_16 =>
97        -- Habilita a ALU para gravar o dado
          habULA         <= '1';
99        -- Operacao de adicao com carry
          ctrlULA      <= b"010"; --ADC
101       -- Habilita o registro de Dados
```

```
            habRegData <= '1';
103         -- Seta o registro para ler o dado
            regDataLe  <= '1';
105         -- Passa para o controle do registro de dados o registro a
                ser utilizado
            ctrlRegData <= opcodeFetch(2 downto 0);
107         -- Fila para de enviar o valor
            leEnd      <= '0';
109
          when Sbb_regImed_16 =>
111         -- Habilita a ALU para gravar o dado
            habULA         <= '1';
113         -- Operacao de subtracao com borrow
            ctrlULA      <= b"011"; --SBB
115         -- Habilita o registro de Dados
            habRegData <= '1';
117         -- Seta o registro para ler o dado
            regDataLe  <= '1';
119         -- Passa para o controle do registro de dados o registro a
                ser utilizado
            ctrlRegData <= opcodeFetch(2 downto 0);
121         -- Fila para de enviar o valor
            leEnd      <= '0';
123
          when Sub_regImed_16 =>
125         -- Habilita a ALU para gravar o dado
            habULA         <= '1';
127         -- Operacao de subtracao
            ctrlULA      <= b"100"; --SUB
129         -- Habilita o registro de Dados
            habRegData <= '1';
131         -- Seta o registro para ler o dado
            regDataLe  <= '1';
133         -- Passa para o controle do registro de dados o registro a
                ser utilizado
            ctrlRegData <= opcodeFetch(2 downto 0);
```

```
135           -- Fila para de enviar o valor
              leEnd      <= '0';
137
          when Xor_regImed_16 =>
139           -- Habilita a ALU para gravar o dado
              habULA         <= '1';
141           -- Operacao de ou exclusivo
              ctrlULA      <= b"101"; --XOR
143           -- Habilita o registro de Dados
              habRegData <= '1';
145           -- Seta o registro para ler o dado
              regDataLe  <= '1';
147           -- Passa para o controle do registro de dados o registro a
                  ser utilizado
              ctrlRegData <= opcodeFetch(2 downto 0);
149           -- Fila para de enviar o valor
              leEnd      <= '0';
151
          when And_regImed_16 =>
153           -- Habilita a ALU para gravar o dado
              habULA         <= '1';
155           -- Operacao de and
              ctrlULA      <= b"110"; --AND
157           -- Habilita o registro de Dados
              habRegData <= '1';
159           -- Seta o registro para ler o dado
              regDataLe  <= '1';
161           -- Passa para o controle do registro de dados o registro a
                  ser utilizado
              ctrlRegData <= opcodeFetch(2 downto 0);
163           -- Fila para de enviar o valor
              leEnd      <= '0';
165
          when ResultRegImed =>
167           -- Captura o resultado e salva no registro
              regDataLe    <= '0';
```

```
169
        when Compara_Reg_16 =>
171       -- Habilita a ALU para gravar o dado
          habULA         <= '1';
173       -- Operacao de compara
          ctrlULA        <= b"111";
175       -- Habilita o registro de Dados
          habRegData <= '1';
177       -- Seta o registro para ler o dado
          regDataLe  <= '1';
179       -- Passa para o controle do registro de dados o registro a
              ser utilizado
          ctrlRegData <= opcodeFetch(2 downto 0);
181       -- Fila para de enviar o valor
          leEnd      <= '0';
183
        when Move_To_Reg =>
185       -- Direciona o dado para o registro diretamente
          ctrlMuxMov <= '1';
187       -- Habilita o registro de Dados
          habRegData <= '1';
189       -- Seta o registro para escrever o dado
          regDataLe  <= '0';
191       -- Passa para o controle do registro de dados o registro a
              ser utilizado
          ctrlRegData <= opcodeFetch(2 downto 0);
193       -- Fila para de enviar o valor
          leEnd      <= '0';
195
        when fimInstrucao =>
197       --Fim da interpretacao da instrucao!
          habULA       <= '0';
199       ctrlULA      <= (others => '0');
          ctrlRegData <= (others => '0');
201       ctrlMuxMov  <= '0';
```

```
203         when Fim_Move_To_Reg =>
              habRegData <= '0';
205
            when Erro =>
207           -- todas saidas zeradas
              leEnd        <= '0';
209           habULA       <= '0';
              ctrlULA        <= "000";
211           regDataLe      <= '0';
              habRegData      <= '0';
213           ctrlRegData    <= "000";
              ctrlDemuxData  <= '0';   --Dado vai direto para a ULA
215           --Trava a MÃ¡quina de Estados

217       end case;
    end if;
219 end process;

221 --definicao do processo de borda de descida que toma a decisao para o
        proximo estado
    borda_descida : process(clock,reset)
223 variable contClock : integer := 0;
    begin
225
    if (reset = '0') then
227
      --Mantem em estado de busca
229   Estado <= Busca;

231 elsif falling_edge(clock) then

233   --Troca os estados
      case Estado is
235
        --Caso estado de Busca, passa para a decodificao ( Fecth )
237     when Busca =>
```

```
        if (sitEnd = '0') then
239        Estado <= Busca;
        --elsif (sitFila = '1' and opcodeFetch /= "XXXX") then
241     else
           Estado <= Fetch;
243     end if;

245     --Caso estado de Fetch descobrir para qual estado a maquina ira
           , de acordo com os
        --16 primeiro  bits
247     --Operacao com Imediato de 16 = 1000 0001 = 81h
        when Fetch =>
249       -- Salva o opcode a ser decodificado
          opcodeFetch <= opcode;
251       case opcode(15 downto 8) is
            when x"81"  => Estado <= OpRegImed16;
253         when x"00"  => Estado <= OpRegImed16;
            when others => Estado <= Erro;
255       end case;

257     --Caso seja uma operacao de Registro,Imediato de 16 bits
        --Analisa os 5 proximos bits para verificar qual operacao a ser
            realizada
259     -- 11000 = Add
        when OpRegImed16 =>
261       if (sitEnd = '0') then
            Estado <= OpRegImed16;
263       else
            case opcodeFetch(7 downto 3) is
265           when b"11000" => Estado <= Add_regImed_16;
              when b"11001" => Estado <= Or_regImed_16;
267           when b"11010" => Estado <= Adc_regImed_16;
              when b"11011" => Estado <= Sbb_regImed_16;
269           when b"11100" => Estado <= And_regImed_16;
              when b"11101" => Estado <= Sub_regImed_16;
271           when b"11110" => Estado <= Xor_regImed_16;
```

```
            when b"11111" => Estado <= Compara_Reg_16;
            when b"10111" => Estado <= Move_To_Reg;
            when others   => Estado <= Erro;
        end case;
    end if;

    --Apos a adicao pula para o resultado de Registro Imediato e
        escreve o valor no registro
    when Add_regImed_16 =>
        if (contClock = 2) then
            Estado <= ResultRegImed;
            contClock := 0;
        else
            Estado <= Add_regImed_16;
            contClock := contClock + 1;
        end if;

    --Apos a operaÃ§Ã£o ou pula para o resultado de Registro
        Imediato e escreve o valor no registro
    when Or_regImed_16 =>
        if (contClock = 2) then
            Estado <= ResultRegImed;
            contClock := 0;
        else
            Estado <= Or_regImed_16;
            contClock := contClock + 1;
        end if;

    --Apos a adicao com carry pula para o resultado de Registro
        Imediato e escreve o valor no registro
    when Adc_regImed_16 =>
        if (contClock = 2) then
            Estado <= ResultRegImed;
            contClock := 0;
        else
            Estado <= Adc_regImed_16;
```

```
305             contClock := contClock + 1;
            end if;
307
        --Apos a subtracao com borrow pula para o resultado de Registro
            Imediato e escreve o valor no registro
309         when Sbb_regImed_16 =>
            if (contClock = 2) then
311             Estado <= ResultRegImed;
                contClock := 0;
313         else
                Estado <= Sbb_regImed_16;
315             contClock := contClock + 1;
            end if;
317
        --Apos a operacao and pula para o resultado de Registro
            Imediato e escreve o valor no registro
319         when And_regImed_16 =>
            if (contClock = 2) then
321             Estado <= ResultRegImed;
                contClock := 0;
323         else
                Estado <= And_regImed_16;
325             contClock := contClock + 1;
            end if;
327
        --Apos a subtracao pula para o resultado de Registro Imediato e
            escreve o valor no registro
329         when Sub_regImed_16 =>
            if (contClock = 2) then
331             Estado <= ResultRegImed;
                contClock := 0;
333         else
                Estado <= Sub_regImed_16;
335             contClock := contClock + 1;
            end if;
337
```

```
        --Apos a operacao ou exclusivo pula para o resultado de
            Registro Imediato e escreve o valor no registro
339     when Xor_regImed_16 =>
          if (contClock = 2) then
341         Estado <= ResultRegImed;
            contClock := 0;
343       else
            Estado <= Xor_regImed_16;
345         contClock := contClock + 1;
          end if;
347
        --Apos salvo o registro Imediato, passa para a finalizacao
349     when ResultRegImed => Estado <= FimInstrucao;

351     --Quando e o fim da instrucao volta para a Busca
        when FimInstrucao =>
353       Estado <= Busca;

355     --Caso algum erro ocorra no percorrer da instrucao a maquina
            trava
        when Erro    => Estado <= Erro;
357
        when Compara_Reg_16 =>
359       if (contClock = 2) then
            Estado <= ResultRegImed;
361         contClock := 0;
          else
363         Estado <= Compara_Reg_16;
            contClock := contClock + 1;
365       end if;

367     When Move_To_Reg => Estado <= Fim_Move_To_Reg;

369     When Fim_Move_To_Reg => Estado <= FimInstrucao;

371   end case;
```

```
  end if;
end process;

end Arquitetura;
```

Descrição do hardware da Unidade de Controle Principal

```
#-------------------------------------#
#Objetivo: Codigo para analisar instrucoes do codigo do DOS
#Desenvolvedor(es):
#   Marcos Aurelio Freitas de Almeida Costa
#   Denis Araujo da Silva
#Data: 03/04/2014
#-------------------------------------#

# Arquivos a serem abertos
asm_modulo = 'ASM.ASM'
command_modulo = 'COMMAND.ASM'
hex2bin_modulo = 'HEX2BIN.ASM'
io_modulo = 'IO.ASM'
msdos_modulo = 'MSDOS.ASM'
stddos_modulo = 'STDDOS.ASM'
trans_modulo = 'TRANS.ASM'

# Instrucoes a serem procuradas
mov_instrucao = 'MOV'
add_instrucao = 'ADD'
adc_instrucao = 'ADC'
sub_instrucao = 'SUB'
sbb_instrucao = 'SBB'
or_instrucao = 'OR'
and_instrucao = 'AND'
xor_instrucao = 'XOR'
cmp_instrucao = 'CMP'
equ_instrucao = 'EQU'

# Contadores
```

```
31 mov_contador = 0
   add_contador = 0
33 adc_contador = 0
   sub_contador = 0
35 sbb_contador = 0
   or_contador = 0
37 and_contador = 0
   xor_contador = 0
39 cmp_contador = 0
   etc_contador = 0
41 equ_contador = 0

43 # Analisa Modulo
   def analise_modulo(nome_modulo):
45     global mov_contador
       global add_contador
47     global adc_contador
       global sub_contador
49     global sbb_contador
       global or_contador
51     global and_contador
       global xor_contador
53     global cmp_contador
       global etc_contador
55     global equ_contador
       arquivo = open(nome_modulo)
57     linha = arquivo.readline()
       while(linha != ''):
59         # Quebra a linha ate o inicio dos comentarios
           codigo = linha.split(';')
61         # Busca os opcodes na linha sem os comentarios
           if(codigo[0] != ''):
63             if(mov_instrucao in codigo[0]):
                   mov_contador += 1
65             elif(add_instrucao in codigo[0]):
                   add_contador += 1
```

```
67              elif(adc_instrucao in codigo[0]):
                    adc_contador += 1
69              elif(sub_instrucao in codigo[0]):
                    sub_contador += 1
71              elif(sbb_instrucao in codigo[0]):
                    sbb_contador += 1
73              elif(or_instrucao in codigo[0]):
                    or_contador += 1
75              elif(and_instrucao in codigo[0]):
                    and_contador += 1
77              elif(xor_instrucao in codigo[0]):
                    xor_contador += 1
79              elif(cmp_instrucao in codigo[0]):
                    cmp_contador += 1
81              elif(equ_instrucao in codigo[0]):
                    equ_contador += 1
83              else:
                    etc_contador += 1
85          # le a proxima linha
            linha = arquivo.readline()
87
    # Metodo principal
89  def main():

91      print ('Modulos Analisados:')
        print ('Modulo: ' + str(asm_modulo))
93      analise_modulo(asm_modulo)
        print ('Modulo: ' + str(command_modulo))
95      analise_modulo(command_modulo)
        print ('Modulo: ' + str(hex2bin_modulo))
97      analise_modulo(hex2bin_modulo)
        print ('Modulo: ' + str(io_modulo))
99      analise_modulo(io_modulo)
        print ('Modulo: ' + str(msdos_modulo))
101     analise_modulo(msdos_modulo)
        print ('Modulo: ' + str(stddos_modulo))
```

```
103     analise_modulo(stddos_modulo)
        print ('Modulo: ' + str(trans_modulo))
105     analise_modulo(trans_modulo)

107     print ('-------------------------------------------------------------')

109     print ('Quantidade de vezes que as instrucoes foram utilizadas:')
        print ('MOV: ' + str(mov_contador))
111     print ('ADD: ' + str(add_contador))
        print ('ADC: ' + str(adc_contador))
113     print ('SUB: ' + str(sub_contador))
        print ('SBB: ' + str(sbb_contador))
115     print ('OR: ' + str(or_contador))
        print ('AND: ' + str(and_contador))
117     print ('XOR: ' + str(xor_contador))
        print ('CMP: ' + str(cmp_contador))
119     print ('EQU: ' + str(equ_contador))
        print ('ETC: ' + str(etc_contador))
121
        print ('-------------------------------------------------------------')
123 main();
```

Código em Python utilizado para a análise quantitativa do código fonte do MS-DOS