

UNIVERSIDADE REGIONAL INTEGRADA DO ALTO URUGUAI E DAS MISSÕES  
URI - CAMPUS DE SANTO ÂNGELO  
DEPARTAMENTO DE ENGENHARIAS E CIÊNCIA DA COMPUTAÇÃO  
CURSO DE CIÊNCIA DA COMPUTAÇÃO

RAFFAEL BOTTOLI SCHEMMER

**DESENVOLVIMENTO DE UMA INFRA-ESTRUTURA DE SUPORTE  
VLIW PARA O PROCESSADOR AIDA-16**

Trabalho de Conclusão do Curso de Ciência da  
Computação

Prof. M.Sc. Carlos Alberto Petry

Santo Ângelo, novembro de 2009

UNIVERSIDADE REGIONAL INTEGRADA DO ALTO URUGUAI E DAS MISSÕES

– URI Campus de Santo Ângelo

Reitor: Bruno Ademar Mentges

Pró-Reitora de Ensino: Helena Confortin

Pró-Reitor de Ensino, Extensão e Pós-Graduação: Sandro Rogério Vargas Ustra

Pró-Reitor de Administração: Clóvis Quadros Hempel

Diretor Geral: Gilberto Pacheco

Diretora Acadêmica: Dinalva Agissé de Souza

Diretora Administrativa: Rosane Maria Seibert

Coordenador do Departamento de Engenharias e Ciência da Computação: Paulo Ricardo Betencourt

Coordenadora do curso de Ciência da Computação: Cristina Paludo Santos

## **AGRADECIMENTOS**

Dedico este espaço de agradecimento, em especial a minha mãe querida Fatima Bottoli Schemmer e a meu pai batalhador, Wilson Irineu Schemmer, pois através de uma vida inteira sofrida de muito trabalho, empenho, luta e dedicação, pôde me prover esta oportunidade única, que alguns tanto valorizam e outros simplesmente a jogam fora que é a possibilidade de estudar e aprender. Tive a oportunidade de vivenciar os dois modelos de estudantes, embarquei sem rumo em um curso, totalmente diferente do meu perfil, nele eu estudava por estudar, acabei precisando optar pela mudança de curso, onde em pouco tempo me encontrei, e unicamente por isso, hoje consigo escrever este papel, orgulhoso do meu feito e feliz por ter de uma forma segura, garantido a meus pais que mesmo errando uma vez, não cometi o mesmo erro novamente e cursei este curso da forma mais extrema possível, valorizando todo esforço e toda dedicação investida em mim dando o máximo em todas as disciplinas. Isso não só me fez alguém realizado como também me somou conhecimento e vivências suficientes para que hoje eu consiga olhar pra traz e sentir pena de quem eu era, e me orgulhar do feito, como olhar pra frente e sentir que ainda existe um longo caminho, que felizmente me motiva ainda mais para não só continuar caminhando, mas sim aprender a correr e se um dia conseguir, aprender a voar.

Ao longo do caminho durante a graduação, fiz muitos amigos, dos quais me orgulho em poder agradecer, pois sem eles não seria o que sou um deles, meu orientador Carlos Alberto Petry, uma das pessoas mais incríveis que tive a oportunidade única de conhecer em um momento decisivo pra mim, considero ele o responsável pela escrita deste trabalho, pois sem ele provavelmente este trabalho não teria o nível e a qualidade que tem hoje, e também devo a ele pelo incentivo que me dá em acreditar que existe possibilidade de continuar lutando para prosseguir estudando porem em um nível de pós-graduação em ciência da computação.

Gostaria também de usar esse espaço pra agradecer a pessoa mais importante ao longo de minha graduação, o mestre Luciano Lores Caimi, o qual me ensinou não somente conhecimentos sobre ciência da computação, que nem teria como citar a quantidade por falta de espaço, como também diversas lições de vida, que são estas as quais me orgulho de ser quem eu sou hoje, foi com ele que aprendi que a palavra superação é algo que nunca é o bastante pra quem quer ser alguém de valor na vida, este trabalho eu escrevo em agradecimento a ti professor Luciano.

Outra pessoa que tive a oportunidade única de conhecer pessoalmente e gostaria de agradecer por ser peça fundamental para definir o foco do meu trabalho de conclusão e por também me permitir pela oportunidade dada para isso foi o Dr. Luigi Carro do GME da UFRGS, onde após ler parte de seu livro de sistemas embarcados, e de uma breve conversa com um de seus ex-alunos do mestrado, reparei que sempre tive interesse nos tópicos que sistemas embarcados focam como base, e nisso decidi explorar um dos projetos de pesquisa bem difundidos do Luigi que é o Femtojava. Minha proposta na época era criar uma cache e avaliar se o custo e a potencia consumida valeriam a pena frente ao desempenho, porem no dia que fui a UFRGS comentar sobre minha proposta e buscar material sobre o Femtojava, fui alertado que deveria estudar VLIW e procurar aplicar esta abordagem sobre alguma coisa real, nisso nasceu o ISS e o montador VLIW para o AIDA-16, justamente para gerar pesquisa da nossa pesquisa.

Concluído os agradecimentos, resolvi deixar este ultimo parágrafo pra agradecer aos amigos Rodrigo Motta pelas noites de sono que perdeu comigo me explicando assuntos difíceis e me dando rumos para seguir em meu trabalho de conclusão de curso e na minha vida profissional, também ao meu amigo Wagner que sempre viu em mim alguém que custo a acreditar que realmente existe e como não poderia deixar de ser, gostaria de agradecer por poder contar como amigo, com o cara mais inteligente, esforçado, objetivo, dinâmico e decidido que já conheci na vida, Tales Marchesan Chaves, tenho certeza absoluta que 2009 não seria 5% do que já foi e esta sendo se não fosse você estar presente, canso de falar, mas falarei quantas vezes for você é provavelmente meu maior exemplo de superação, eu tentei e continuo tentando trabalhar no mesmo clock que você, porem nem eu<sup>3</sup> conseguiria ser metade do que você consegue ser, e falo unicamente da vida acadêmica, sucesso camarada, isso é o mínimo que posso te garantir que você terá sem sombras de duvidas daqui pra frente.

## SUMÁRIO

1	INTRODUÇÃO	15
1.1	Redefinição de objetivos e requisitos mínimos	16
1.1.1	Redefinição dos Objetivos específicos	17
1.1.2	Redefinição os Requisitos mínimos	18
1.2	Organização do documento	19
2	REFERENCIAL TEORICO E TRABALHOS RELACIONADOS	20
2.1	Estudo da especificação do processador AIDA-16	20
2.1.1	Modos de endereçamento	20
2.1.2	Formato das instruções	22
2.1.3	Conjunto e representação das instruções	24
2.1.4	Recompilando detalhes do ISA do AIDA-16	24
2.2	Técnicas para escalar desempenho em caminhos de dados	25
2.2.1	Maquinas monociclo	26
2.2.2	Maquinas multiciclo	26
2.2.3	Maquinas Pipeline	26
2.2.4	Maquinas Superescalares	27
2.2.5	Maquinas VLIW	28
2.3	Paralelismo em nível de instrução	28
2.4	Estudo da abordagem VLIW	31
2.4.1	Histórico da abordagem VLIW	33
2.4.2	Vantagens da arquitetura VLIW	34
2.4.3	Desvantagens da arquitetura VLIW	34
2.4.4	Técnicas VLIW associadas ao compilador	35
2.5	Técnicas de Encapsulamento de instruções em organização da arquitetura de processadores VLIW	36

2.5.1	Pipeline em processadores VLIW	37
2.5.2	Mecanismo de fetch	37
2.5.3	Bypassing	38
2.6	Comparativo entre a organização VLIW x Superescalar	38
2.7	Estudo introdutório sobre ISS	39
2.8	Estudo introdutório sobre Montadores	40
3	DESENVOLVIMENTO DA INFRA-ESTRUTURA DE SUPORTE VLIW	44
3.1	Especificação e implementação do montador	44
3.1.1	Estudo do ISA do processador AIDA-16	44
3.1.2	Especificação das diretivas básicas de montagem	45
3.1.3	Implementação do bloco de leitura do arquivo assembly	47
3.1.4	Implementação do bloco de análise das instruções	48
3.1.5	Implementação do bloco de análise dos dados	53
3.1.6	Implementação do bloco de código intermediário e geração de código assembly das instruções	54
3.1.7	Implementação do bloco de código intermediário e geração de código assembly dos dados	55
3.2	Especificação e Implementação do ISS	56
3.2.1	Especificação do ISS	56
3.2.2	Lendo o arquivo binário gerado pelo montador e iniciando o montador	58
3.2.3	Especificação da Interface de interação com o usuário	58
3.2.4	Implementação do funcionamento da interface do ISS	60
3.3	Especificação e Implementação do VLIW no montador	64
3.3.1	Especificação VLIW sobre o montador	64
3.3.2	Implementação VLIW sobre o montador	65
3.3.3	Formato das instruções	65
3.4	Especificação e Implementação do VLIW no ISS	77
3.4.1	Especificação do VLIW sobre o ISS	77
3.4.2	Implementação VLIW sobre o ISS	78
4	VALIDAÇÃO E RESULTADOS OBTIDOS	79
4.1	Validação do montador de acordo com a especificação do AIDA-16	79
4.1.1	Testando instruções validas do Formato – R	79
4.1.2	Testando instruções invalidas do Formato – R	80
4.1.3	Testando instruções validas do Formato – I	84
4.1.4	Testando instruções invalidas do Formato – I	85
4.1.5	Testando instruções validas do Formato – Load/Store	87
4.1.6	Testando instruções inválidas do Formato – Load/Store	88
4.1.7	Testando instruções validas do Formato – Branch	90
4.1.8	Testando instruções inválidas do Formato - Branch	91

4.2	Validação do ISS conforme a especificação do processador AIDA-16	93
4.2.1	Testando as instruções do Formato – R no ISS	93
4.2.2	Testando as instruções do Formato – I no ISS	95
4.2.3	Testando as instruções do Formato – Load/Store no ISS	95
4.2.4	Testando as instruções do Formato – Branch no ISS	97
4.3	Validação do montador VLIW para o ISA do AIDA-16	98
4.4	Validação do ISS VLIW para o ISA do AIDA-16	114
5	CONCLUSÃO E CONTRIBUIÇÕES	116
5.1	Trabalhos futuros	117
	REFERÊNCIAS BIBLIOGRÁFICAS	119
	APENDICE A – EXEMPLO DE VALIDAÇÃO DO ISS E DO MONTADOR VLIW	120
	APENDICE B – CÓDIGO ASSEMBLY RESPONSÁVEL POR VALIDAR AS INSTRUÇÕES LÓGICAS E ARITMÉTICAS DO MODO DE ENDEREÇAMENTO DIRETO	122
	APENDICE C – TABELA REVISADA DO CONJUNTO DE INSTRUÇÕES DO PROCESSADOR AIDA-16	123

## **LISTA DE ABREVIACÕES**

AIDA-16 – Arquitetura Instrucional Destinada ao Ensino  
CPI – Cycles per clock  
CPU – Central Processing Unit  
ILP – Instruction Level Parallelism  
ISA – Instruction Set Architecture  
ISS – Instruction Set Simulator  
MIPS - Microprocessor without Interlocked Pipeline Stages  
RAW – Read after write  
URI – Universidade Regional Integrada  
VHDL – VHSIC Hardware Description Language  
VHSIC - Very High Speed Integrated Circuit  
VLIW – Very Long Instruction Word  
WAR – Write after read  
WAW – Write after write  
VLIW – Very Long Instruction Word



## LISTA DE FIGURAS

Figura 2.1: Modos de endereçamento do processador AIDA-16	25
Figura 2.2: Formato – R do processador AIDA-16	26
Figura 2.3: Formato – I do processador AIDA-16	27
Figura 2.4: Formato – Load/Store do processador AIDA-16	27
Figura 2.5: Formato – Branch do processador AIDA-16	28
Figura 3.1: Exemplo de código assembly para a especificação do montador	37
Figura 2.6: Modelo de máquina VLIW ideal	52
Figura 3.2: Formato – R : Modo de endereçamento direto	53
Figura 3.3: Formato – R : Modo de endereçamento Imediato	55
Figura 3.4: Formato – I	56
Figura 3.5: Formato – Load/Store	56
Figura 3.6: Formato – Branch	57
Figura 3.7: Estrutura responsável por guardar os campos da instrução	59
Figura 3.8: Estrutura responsável por guardar os campos dos dados	60
Figura 3.9: Estrutura responsável por guardar os campos das instruções	67
Figura 3.10: Instruções Formato R (Direto) – Formato R (Direto)	71
Figura 3.11: Instruções Formato R (Direto/Indireto) e Formato R (Indireto/Indireto)	72
Figura 3.12: Instruções Formato R (Direto) - Formato R (Imediato)	73
Figura 3.13: Instruções Formato R (Imediato) – Formato R (Indireto)	73
Figura 3.14: Instruções Formato R (Imediato) – Formato R (Direto)	74
Figura 3.15: Instruções Formato R (Indireto) – Formato R (Imediato)	74
Figura 3.16: Instruções Formato R (Direto) – Formato I	75
Figura 3.17: Instruções Formato R (Indireto) – Formato I	76
Figura 3.18: Instruções Formato R (Imediato) – Formato I	76
Figura 3.19: Instruções Formato R (Direto) – Formato Load/Store	77
Figura 3.20: Instruções Formato R (Indireto) – Formato Load/Store	77
Figura 3.21: Instruções Formato R (Imediato) – Formato Load/Store	78
Figura 3.22: Instruções Formato R (Direto/Indireto/Imediato) – Formato Branch	79
Figura 3.23: Instruções Formato I – Formato I	79
Figura 3.24: Instruções Formato I – Formato Load/Store	80
Figura 3.25: Instruções Formato I – Formato Branch	80
Figura 3.26: Instruções Formato Load/Store – Formato Load/Store	81
Figura 3.27: Instruções Formato Load/Store – Formato Branch	81
Figura 3.28: Instruções Formato Branch – Formato Branch	81
Figura 3.29: Instruções válidas do formato – R	86

Figura 4.1: Saída do montador ao efetuar o bloco de instruções do Formato-R	87
Figura 4.2: Instruções inválidas do formato – R	87
Figura 4.3: Código da operação não encontrado	88
Figura 4.4: Modo de endereçamento não suportado na instrução do Formato-R	88
Figura 4.5: Registrador de destino não suportado na instrução do Formato-R	89
Figura 4.6: Falta operandos na instrução do Formato-R	89
Figura 4.7: Constante não suportada pela instrução do Formato-R	90
Figura 4.8: Registrador não suportado na instrução do Formato-R	90
Figura 4.9: Excesso de parâmetros na instrução do Formato-R	91
Figura 4.10: Instruções válidas do formato – I	91
Figura 4.11: Teste das instruções válidas do Formato – I	92
Figura 4.12: Instruções inválidas do formato – I	92
Figura 4.13: Código da operação não encontrado	93
Figura 4.14: Registrador de destino não suportado na instrução do Formato-I	93
Figura 4.15: Falta operandos na instrução do Formato-I	94
Figura 4.16: Excesso de parâmetros na instrução do Formato-I	94
Figura 4.17: Constante não suportada na instrução do Formato-I	95
Figura 4.18: Instruções válidas do formato – Load/Store	95
Figura 4.19: Teste das instruções válidas do Formato – Load/Store	96
Figura 4.20: Instruções inválidas do formato – Load/Store	96
Figura 4.21: Código da operação não encontrado	97
Figura 4.22: Registrador não suportado na instrução do Formato – Load/Store	97
Figura 4.23: Falta parâmetros na instrução do Formato – Load/Store	98
Figura 4.24: Excesso de parâmetros na instrução do Formato – Load/Store	98
Figura 4.25: Instruções válidas do formato – Branch	99
Figura 4.26: Teste das instruções válidas do Formato – Branch	99
Figura 4.27: Instruções inválidas do formato – Branch	100
Figura 4.28: Código da operação não encontrado	100
Figura 4.29: Constante não suportada na instrução do Formato – Branch	101
Figura 4.30: Falta parâmetros na instrução do Formato – Branch	101
Figura 4.31: Falta parâmetros na instrução do Formato – Branch	102
Figura 4.32: Equação que valida as instruções lógicas e aritméticas	103
Figura 4.33: Validando instruções lógicas e aritméticas no modo de end.direto	103
Figura 4.34: Instruções que validam o formato – I	104
Figura 4.35: Validando as instruções li e lui do formato – I	105
Figura 4.36: Instruções que validam o formato – Load/Store	106
Figura 4.37: Validando as instruções do Formato Load/Store	107
Figura 4.38: Bloco de instruções que valida o formato – Branch	108
Figura 4.39: Validando as instruções do Formato Branch	109
Figura 4.31: Equação que valida as instruções lógicas e aritméticas	109
Figura 4.40: Validando a dependência do tipo RAW em R (Direto) – R (Direto)	110
Figura 4.41: Validando a dependência do tipo WAR em R (Direto) – R (Direto)	110
Figura 4.42: Validando a dependência do tipo WAW em R (Direto) – R (Direto)	110
Figura 4.43: Validando que não existe dependência do tipo RAW em R (Direto) – R (Imediato)	111
Figura 4.44: Validando a dependência do tipo WAR em R (Direto) – R (Imediato)	111
Figura 4.45: Validando a dependência do tipo WAW em R (Direto) – R (Imediato)	112
Figura 4.46: Validando a saída das instruções R (Imediato) – R (Indireto)	112
Figura 4.47: Validando a dependência do tipo RAW em R (Imediato) – R (Direto)	112
Figura 4.48: Validando que não existe dependência do tipo WAR em R (Imediato) – R (Direto)	113

Figura 4.49: Validando que não existe dependência do tipo WAR em R (Imediato) – R (Direto)	113
Figura 4.50: Validando que existem dependências do tipo RAW e WAW nos formatos R(Imediato) – R(Indireto)	113
Figura 4.51: Validando que não existem dependências do tipo WAR nas instruções do tipo R(Imediato) – R(Indireto)	114
Figura 4.52: Validando a dependência do tipo RAW das instruções R (Direto) – I	114
Figura 4.53: Validando a dependência do tipo WAW das instruções R (Direto) – I	114
Figura 4.54: Validando a dependência do tipo WAW das instruções R (Direto) – I	115
Figura 4.56: Validando a dependência do tipo WAR/WAW das instruções R (Indireto) – I	115
Figura 4.57: Validando que não existem dependências do tipo RAW nas instruções do tipo R(Imediato) – I	115
Figura 4.58: Validando que não existem dependências do tipo WAR nas instruções do tipo R(Imediato) – I	116
Figura 4.59: Validando a dependência do tipo WAW das instruções R (Imediato) – I	116
Figura 4.60: Validando a dependência do tipo RAW das instruções R (Direto) – Load/Store	117
Figura 4.61: Validando a dependência do tipo WAR das instruções R (Direto) – Load/Store	117
Figura 4.62: Validando a dependência do tipo WAR/WAW das instruções R (Direto) – Load/Store	117
Figura 4.63: Validando a dependência do tipo RAW/WAR/WAW das instruções R (Indireto) – Load/Store	118
Figura 4.64: Validando a dependência do tipo RAW das instruções R (Imediato) – I	118
Figura 4.65: Validando que não existe dependências nas instruções R (Imediato) – I	119
Figura 4.66: Validando a dependência do tipo WAW das instruções R (Imediato) – I	119
Figura 4.67: Validando a dependência de controle das instruções R (Direto) – Branch	120
Figura 4.68: Validando a dependência de controle das instruções R (Imediato) – Branch	120
Figura 4.69: Validando a dependência de controle das instruções R (Indireto) – Branch	120
Figura 4.70: Validando que não existem dependências RAW entre instruções I – I	121
Figura 4.71: Validando que não existem dependências WAR entre instruções I – I	121
Figura 4.72: Validando a dependência do tipo WAW entre instruções I – I	121
Figura 4.73: Validando a dependência do tipo RAW entre instruções I - Load/Store	122
Figura 4.74: Validando que não existem dependências WAR entre instruções I – Load/Store	122
Figura 4.75: Validando a dependência do tipo WAW entre instruções I – Load/Store	122
Figura 4.76: Validando que não existem dependências entre instruções I – Branch	123
Figura 4.77: Validando a dependência do tipo RAW entre instruções Load/Store-Load/Store	123
Figura 4.78: Validando a dependência do tipo WAR entre instruções Load/Store-Load/Store	123
Figura 4.79: Validando a dependência do tipo WAW entre instruções Load/Store-Load/Store	124
Figura 4.80: Validando que não existem dependências entre instruções Load/Store – Branch	124
Figura 4.81: Validando a dependência de controle entre as instruções Branch – Branch	124
Figura 4.82: Bloco de busca de instrução VLIW	125
Figura 4.83: Bloco de decodificação de instrução VLIW	125
Figura 4.84: Bloco de execução de instrução VLIW	126

## **LISTA DE TABELAS**

Tabela 2.1: Conjunto de instruções do processador AIDA-16	29
---	----

## **RESUMO**

Este trabalho de conclusão de curso tem como objetivo principal o desenvolvimento de uma infra-estrutura com suporte à VLIW para o conjunto de instruções (ISA) do processador AIDA-16 composta de um montador (Assembler) para o código de montagem (Assembly) e um simulador do conjunto de instruções (ISS) para executar este código em formato de código de máquina. Tal infra-estrutura se caracteriza pelo paradigma de execução não temporizada (untimed), permitindo assim, a execução de código AIDA-16 nativo segundo a abordagem VLIW. A principal motivação para tal implementação é disponibilizar um novo modelo de arquitetura de processamento para o processador já existente, descrito em nível de VHDL. Adicionalmente objetiva-se disponibilizar ferramentas que permitam realizar a execução do código nativo do processador servindo como ferramenta de ensino tanto para professores, em disciplinas como Arquitetura de Computadores. No caso dos alunos, estas ferramentas devem auxiliar na consolidação dos conceitos aprendidos, a partir da montagem e simulação do código nativo considerando exemplos práticos.

**Palavras chave:** AIDA-16, VLIW, ISS, Montador, ferramentas de simulação.

## **ABSTRACT**

This final year project aims to development a VLIW infra-structure in order support the instruction set architecture (ISA) of AIDA-16 processor, which consists of an Assembler for the assembly code and a simulator instruction set (ISS) that executes this code in machine format. The VLIW infra-structure is characterized by the untimed execution paradigm, which allows the AIDA-16 code execution according to the native VLIW approach. The focus of this implementation is to provide a new type of processor architecture for a well know and existing processor, described in VHDL. Additionally, provides tools that will allow processor native code execution, which can be used by teachers in disciplines, such as Computer Architecture. In the students context, these tools should be used to reinforce learned concepts from compiling and simulating a native code, considering practical examples.

**Keywords:** AIDA-16, VLIW, ISS, Assembler,Simulation Tools

## 1 INTRODUÇÃO

Assim como em diversas áreas do conhecimento, na área da computação existem disciplinas com assuntos inter-relacionados que exigem uma ampla gama de conhecimentos e pré-requisitos para que os assuntos possam ser entendidos. Sistemas digitais e arquitetura de computadores fazem parte deste, exigindo não somente o entendimento de temas específicos como também o seu relacionamento com outras áreas, tais como sistemas operacionais, compiladores, linguagem de programação, entre outros. No intuito de auxiliar os professores no ensino destas disciplinas e facilitar o processo de aprendizado por parte dos alunos, foi desenvolvido um microprocessador multiciclo chamado AIDA-16 (Arquitetura Instrucional Destinada ao Ensino) [MOT05]. O trabalho disponibilizou a arquitetura do conjunto de instruções, os modos de endereçamento, os tipos de instruções e seus formatos, e a organização interna do processador.

O resultado do projeto AIDA-16 disponibilizou uma arquitetura possível de ser utilizada como ferramenta de ensino, servindo como forma alternativa ao ensino da programação assembly, neste caso baseado na ISA do AIDA-16, além de permitir a assimilação dos conceitos básicos usados na arquitetura deste processador. Esta estratégia de ensino foi consagrada em outras universidades, onde diversos processadores foram especificados em diferentes graus de complexidade destinados ao ensino da organização e arquitetura de computadores [WEB01].

Processadores mais avançados usam várias técnicas para explorar paralelismo em nível de instrução, implementando outros modelos de organização interna como o uso de pipeline, unidades superescalar, entre outros. Atualmente tem se verificado que técnicas como VLIW têm ganhado força na área de sistemas embarcados [FIS05], motivadas pelo crescente aumento do nível de complexidade destes sistemas. Além disto, o uso de interfaces gráficas exige maior carga de processamento por parte do sistema. Processadores VLIW trazem como proposta implementar em software o tratamento das dependências e possíveis perigos que daí podem decorrer devido à execução paralela das instruções,

característica desta abordagem de paralelismo. Assim, o escalonamento das instruções a serem executadas de forma paralela é realizado pelo processo de compilação caracterizando o escalonamento estático das instruções. O resultado do escalonamento permite encapsular diversas instruções que ao serem emitidas para o hardware serão executadas de forma paralela. Desta forma é possível manter o processador compacto reduzindo o custo do chip, o consumo de energia e melhorando o desempenho, visto que a complexidade inerente à resolução dos perigos decorrentes das dependências já foram solucionadas, consequentemente o hardware pode se concentrar na execução das instruções. Todos os fatores anteriormente citados justificam a crescente utilização das técnicas VLIW em sistemas embarcados [FIS05], onde o baixo consumo de potência e o custo são de alta relevância e impõem limitantes ao projeto de processadores embarcados.

O desenvolvimento de uma versão do processador AIDA-16 fazendo uso da arquitetura VLIW corrobora com o principal objetivo do projeto, o ensino acadêmico, disponibilizando uma versão cujos conceitos mais avançados de arquitetura de computadores pode colaborar ainda mais com as disciplinas relacionadas, como Sistemas Digitais e Arquitetura de Computadores, servindo tanto de modelo de estudos como ferramenta prática para aplicar os conceitos aprendidos.

Este trabalho busca desenvolver um simulador do conjunto de instruções AIDA-16 usando paradigma não temporizado (untimed), que será referenciado a partir deste ponto como ISS, e um montador VLIW para o mesmo conjunto. O montador tem por objetivo verificar se o código assembly escrito pelo desenvolvedor está correto em termos sintáticos e léxicos e dentro do possível também quanto à sua semântica. Adicionalmente o montador também identificará problemas relacionados às dependências entre instruções indicando o local e tipo de dependência, a fim de garantir que o código de máquina possa ser executado conforme a abordagem VLIW. O encapsulamento considerado para a instrução VLIW conterá duas instruções AIDA-16 para cada emissão. O ISS disponibilizado implementa execução VLIW a partir das duas instruções contidas no encapsulamento.

### **1.1 Redefinição de objetivos e requisitos mínimos**

Esta seção do trabalho de conclusão de curso, referenciado a partir deste ponto como TCC, tem por objetivo redefinir objetivos gerais, objetivos específicos e requisitos



mínimos definidos no projeto de conclusão de curso, PCC, devido à necessidade de ajustes ocorridos durante o desenvolvimento do TCC.

O objetivo geral do PCC, definido no semestre passado, visava o desenvolvimento de uma especificação e implementação VLIW do processador AIDA-16, utilizando uma linguagem de descrição de comportamento de hardware, VHDL. A escolha desta linguagem ocorreu devido à possibilidade de prototipação em hardware tipo FPGA do processador e permitir a execução de código AIDA-16 neste ambiente.

Durante o período de realização do TCC, o autor constatou que o tempo necessário para a realização desta tarefa, no nível proposto, precisaria ser maior que o tempo previsto para o referido trabalho de conclusão, devido à grande necessidade de aprendizado de novos conceitos, ferramentas e bibliotecas de desenvolvimento, além da complexidade envolvendo diversas técnicas VLIW que deveriam ser estudadas para serem aplicadas ao processador AIDA-16 em versão VLIW. Diante disto, foi feita alterações nos objetivos de forma que ao invés de realizar a especificação e a descrição do processador em linguagem VHDL e posterior prototipação em FPGA para teste, se trabalharia com a abordagem de simulação em nível funcional fazendo uso de linguagens de programação de alto nível como C. O resultado produziria um ISS capaz de executar por simulação a funcionalidade do conjunto de instruções AIDA-16 segundo o paradigma VLIW.

Outro objetivo geral definido foi à especificação e a implementação do montador com suporte à VLIW. Este objetivo permanece inalterado conforme definido no PCC

#### 1.1.1 Redefinição dos Objetivos específicos

Segue abaixo a lista dos objetivos específicos definidos durante o desenvolvimento do PCC.

Especificar a plataforma AIDA com base na especificação VLIW a ser desenvolvida e no AIDA-16 já conhecidos, considerando a seguintes etapas:

- ❖ Estudar os conceitos iniciais da abordagem VLIW;
- ❖ Estudar a arquitetura do conjunto de instruções AIDA-16;
- ❖ Estudar os modelos de projeto a serem usados no encapsulamento VLIW;

- ❖ Estudar os impactos da abordagem VLIW em relação à organização da arquitetura do processador;
- ❖ Estudar as técnicas VLIW e quais suas implicações no projeto do montador do processador;
- ❖ Realizar a especificação das instruções VLIW utilizando com base a ISA do processador AIDA-16;
- ❖ Realizar a especificação do caminho de dados e de controle da nova versão de processador baseado em VLIW;
- ❖ Realizar a especificação do montador e das técnicas VLIW.

Abaixo segue a lista redefinida dos objetivos específicos:

- ❖ Estudar a especificação da ISA do processador AIDA-16;
- ❖ Estudar os conceitos da abordagem VLIW;
- ❖ Especificar e implementar o montador AIDA-16 com suporte VLIW;
- ❖ Especificar e implementar o ISS AIDA-16 com suporte VLIW;
- ❖ Prover mecanismos e cenários de teste para validar o montador e o ISS.

#### 1.1.2 Redefinição os Requisitos mínimos

Pelo PCC esperava-se obter como requisito mínimo, a especificação do encapsulamento das instruções da ISA do processador AIDA-16, implementando ao menos uma técnica VLIW no montador; especificação do caminho dos dados e do controle de forma a obter um circuito lógico capaz de executar código nativo AIDA-16 segundo a abordagem VLIW.

Devido às mudanças realizadas nos objetivos gerais e específicos no TCC, foi definido outro requisito mínimo a serem atendido, embora tenha na prática tido pouca alteração se observada a mudança no nível de abstração imprimida ao presente trabalho. A saber: especificação e a implementação de um montador com suporte VLIW, capaz de atender a pelo menos uma técnica VLIW, gerando código de máquina apenas quando livre de dependências; especificar e implementar um simulador do conjunto de instruções AIDA-16 com suporta a execução VLIW, o qual deverá carregar e executar as instruções geradas pelo montador; desenvolver cenários de teste a fim de permitir a corretude do processo de

montagem com suporte VLIW e validar a simulação funcional das instruções da ISA considerada.

## **1.2 Organização do documento**

O restante deste TCC está estruturado conforme detalhado nos itens abaixo.

- ❖ Capítulo 2: apresenta a compilação do referencial teórico consultado como base para do presente TCC, bem como uma seção com um breve resumo de trabalhos relacionados, previamente desenvolvidos nesta universidade.
- ❖ Capítulo 3: constitui o principal capítulo deste trabalho, seguido do Capítulo 4. Apresenta os procedimentos de especificação e implementação do montador e do simulador do conjunto de instruções considerando duas etapas: (i) sem suporte à abordagem VLIW e (ii) com suporte à VLIW. Apresenta o trabalho realizado com base nos objetivos propostos e no estudo apresentado no Capítulo 2.
- ❖ Capítulo 4: apresenta o processo de validação dos requisitos atribuídos ao montador bem como do simulador do conjunto de instruções. São apresentados os resultados que comprovam que os requisitos definidos foram atendidos.
- ❖ Capítulo 5: por fim, são apresentadas as conclusões finais do trabalho de conclusão e a sugestões para trabalhos futuros que poderão ser explorados a partir das implementações geradas por este trabalho.

## **2 REFERENCIAL TEORICO E TRABALHOS RELACIONADOS**

Esta seção do trabalho de conclusão de curso dá uma introdução à especificação do processador AIDA-16 e compila assuntos utilizados para o desenvolvimento dos próximos capítulos do trabalho.

### **2.1 Estudo da especificação do processador AIDA-16**

AIDA-16, acrónimo de Arquitetura Instrucional Destinada ao Ensino, é um processador multiciclo de 16bits que trabalha com arquitetura híbrida, contendo um conjunto de instruções complexas (CISC) e características da organização RISC como um grande banco de registradores e também sendo baseado em arquitetura load/store. O AIDA-16 é um processador que foi especificado e implementado em uma linguagem de descrição de hardware (VHDL), como prototipado em um FPGA para propósito de validação [MOT05].

#### **2.1.1 Modos de endereçamento**

Como o ISA do AIDA foi desenvolvido baseado em arquiteturas de load/store é preciso fornecer diversas maneiras ao programador de manipular os dados contidos no banco de registradores. Para isso o AIDA implementa três modos de endereçamento, que são selecionadas pelas instruções do formato - R a partir do campo am (address memory). Na figura abaixo temos os códigos binários que representam respectivamente cada um dos modos de endereçamento suportados pelo processador AIDA-16:

Código Binário (campo <i>rd</i> )	Modo de Endereçamento
00	Modo Direto
01	Modo Indireto
10	Modo Imediato
11	Reservado

*Figura 2.1: Modos de endereçamento do processador AIDA-16*

### 1. Modo direto

Nesse modo de endereçamento o valor do campo *am* na instrução do formato - R contém o valor 00 e no campo *rs* e *rt* referentes aos registradores da instrução contém o número do registrador no banco de registradores que contém o dado que deve ser feita a operação e que posteriormente deve armazenar o resultado da operação da instrução.

### 2. Modo indireto

Neste modo de endereçamento o valor do campo *am* na instrução do formato - R contém o valor 01 e no campo *rs* e *rt* referentes aos registradores da instrução contém o número do registrador no banco de registradores, que contém o número que será o registrador a ser manipulado pela instrução, e que posteriormente deverá armazenar o resultado da operação da instrução.

### 3. Modo imediato

Neste modo de endereçamento o valor do campo *am* na instrução do formato - R contém o valor 10 e no campo *rs* e *rt* referentes aos registradores da instrução contém a constante numérica que deverá ser feita a operação, não sendo necessário efetuar leituras no banco de registrador para conhecer os dados que estarão explicitamente declarados no corpo da instrução. O registrador que irá receber o dado será conhecido por *rd*, que se estiver (*rd* = 0) o valor deverá ir para *\$t0* e se estiver em (*rd* = 1) o valor deverá ir para *\$t1*.

Este processador nasceu a partir de um estudo onde foi verificado a falta de uma plataforma de ensino capaz de servir como base para experimentos práticos a disciplinas

como sistemas digitais onde o hardware se faz presente e arquitetura de computadores que se baseia totalmente nos conceitos da ISA com programação assembly. Um dos objetivos do AIDA-16 era prover algo completo, capaz de abstrair a complexidade de uma ferramenta paga, como seus custos de aquisição [MOT05].

### 2.1.2 Formato das instruções

#### 1) Formato - R

O formato - R tem contempla as instruções lógicas e aritméticas, responsáveis pelas operações lógicas e matemáticas do processador AIDA-16. Neste modo existem 16 instruções contempladas no AIDA-16, listadas abaixo:

- ✓ (Instruções implementadas com base no formato - R: nop, add, sub, inc, dec, mul, div, mod, shr, shl, ror, rol, or, and, xor, not). Abaixo a representação gráfica da distribuição dos campos do formato - R:



*Figura 2.2: Formato – R do processador AIDA-16*

A seguir é descrito o significado de cada um dos campos do formato - R:

- ❖ op : Campo responsável por informar qual é o código da operação a ser executada
- ❖ am : Seleciona qual o modo de endereçamento que deverá ser utilizado como explicado anteriormente
- ❖ rd : Campo responsável por informar se o registrador que deve receber o resultado da operação é rs(rd = 0) ou rt(rd = 1)
- ❖ rs : Campo responsável por implementar o registrador 0
- ❖ rt : Campo responsável por implementar o registrador 1

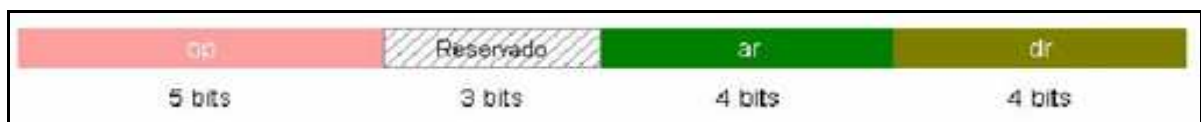
#### 2) Formato – I



*Figura 2.3: Formato – I do processador AIDA-16*

- ❖ op : Campo responsável por informar qual é o código da operação a ser executada
- ❖ reservado: Área ignorada pelo processador
- ❖ rd : Campo responsável por indicar se a constante vai ser escrita em \$t0(rd = 0) ou \$t1(rd = 1)
- ❖ v : Campo que informa a constante a ser carregada no registrador

### 3) Formato - Load/Store



*Figura 2.4 : Formato – Load/Store do processador AIDA-16*

- ❖ op : Campo responsável por informar qual é o código da operação a ser executada
- ❖ reservado : Área ignorada pelo processador
- ❖ ar : Campo responsável por indicar o registrador onde o endereço da célula de memória deve ser escrita
- ❖ rd : Campo responsável por indicar o registrador onde se encontra o dado a ser escrito

### 4) Formato – Branch



*Figura 2.5 : Formato – Branch do processador AIDA-16*

- ❖ op : Campo responsável por informar qual é o código da operação a ser executada
- ❖ d : Indica o numero de instruções para onde o programa deverá ser desviado

### 2.1.3 Conjunto e representação das instruções

O processador AIDA-16 possui um total de 30 instruções, onde o campo op de 5bits é presente em todos os formatos de instruções que permite que até 32 instruções sejam implementadas no AIDA-16, porem somente 30 delas são implementadas. O conjunto de instruções do AIDA-16 pode ser dividido em 5 categorias :

- ❖ 8 Instruções aritméticas responsáveis por prover capacidade computacional para processar dados numéricos
- ❖ 8 Instruções lógicas que realizam operações bit a bit nos operados
- ❖ 7 Instruções de transferência de dados que permitem carregar constantes de dados da memória de dados para os registradores e movimentar valores entre os registradores
- ❖ 6 Instruções de desvio condicional que são usadas para alterar o curso de um programa através de uma determinada condição
- ❖ 1 Instrução de desvio incondicional que altera o fluxo de controle seqüencial de um programa

Abaixo a tabela que compila todos os detalhes do conjunto de instruções do processador AIDA-16 :

### 2.1.4 Recompilando detalhes do ISA do AIDA-16

Ao realizar a especificação do montador e o ISS VLIW, verificou-se que deveria existir uma instrução responsável por indicar no código fonte onde o simulador deveria parar de executar o código. Ao verificar os números dos códigos de operação das instruções existentes no AIDA-16, foi especificado que a nova instrução, irá se chamar halt e deverá ter o numero 30 no conjunto de instruções, que em binário representa 11110. Verificou-se que na tabela antiga do conjunto de instruções do processador AIDA-16 existiam alguns erros que eram explicados corretamente em suas respectivas seções porem descritos errados na tabela de referencia do conjunto de instruções. A fim de corrigir os erros, foi criada uma



nova tabela com a função de corrigir os erros citados acima e contemplar a nova instrução, que esta anexada ao apêndice C deste trabalho.

*Tabela 2.1 : Conjunto de instruções do processador AIDA-16*

Categoria	Instr.	Exemplo	Significado	Flag <sup>1</sup>	Descrição
NA	<i>nop</i>	<i>nop</i>	NA	NA	nenhuma operação
Aritmética	<i>add</i>	<i>add 0, 00, \$r0, \$r1</i>	$Sr_0 \leftarrow Sr_0 + Sr_1$	c, o, z	adição
	<i>sub</i>	<i>sub 0, 00, \$r0, \$r1</i>	$Sr_0 \leftarrow Sr_0 - Sr_1$	c, o, z	subtração
	<i>inc</i>	<i>inc 0, 00, \$r0</i>	$Sr_0 \leftarrow Sr_0 + 1$	c, o, z	incremento
	<i>dec</i>	<i>dec 0, 00, \$r0</i>	$Sr_0 \leftarrow Sr_0 - 1$	c, o, z	decremento
	<i>mul</i>	<i>mul 0, 00, \$r0, \$r1</i>	$Sr_0 \leftarrow Sr_0 \times Sr_1$	c, o, z	multiplicação
	<i>div</i>	<i>div 0, 00, \$r0, \$r1</i>	$Sr_0 \leftarrow Sr_0 \div Sr_1$	c, o, z	divisão
	<i>mod</i>	<i>mod 0, 00, \$r0, \$r1</i>	$Sr_0 \leftarrow Sr_0 \bmod 1$	c, o, z	resto da divisão
Lógica	<i>shr</i>	<i>shr 0, 00, \$r0</i>	$Sr_0 \leftarrow shr(Sr_0)$	c, z	desloc. para a direita
	<i>shl</i>	<i>shl 0, 00, \$r0</i>	$Sr_0 \leftarrow shl(Sr_0)$	c, z	desloc. para a esquerda
	<i>ror</i>	<i>ror 0, 00, \$r0</i>	$Sr_0 \leftarrow ror(Sr_0)$	z	rotação para a direita
	<i>rol</i>	<i>rol 0, 00, \$r0</i>	$Sr_0 \leftarrow rol(Sr_0)$	z	rotação para a esquerda
	<i>or</i>	<i>or 0, 00, \$r0</i>	$Sr_0 \leftarrow (Sr_0) or (Sr_1)$	z	or lógico <i>bit-a-bit</i>
	<i>and</i>	<i>and 0, 00, \$r0</i>	$Sr_0 \leftarrow (Sr_0) and (Sr_1)$	z	and lógico <i>bit-a-bit</i>
	<i>xor</i>	<i>xor 0, 00, \$r0</i>	$Sr_0 \leftarrow (Sr_0) xor (Sr_1)$	z	xor lógico <i>bit-a-bit</i>
	<i>not</i>	<i>not 0, 00, \$r0</i>	$Sr_0 \leftarrow not(Sr_0)$	z	complemento
Transferência de dados	<i>li</i>	<i>li 0, 50</i>	$St_0 \leftarrow 50$	NA	carga baixa de reg.
	<i>lui</i>	<i>lui 0, 500</i>	$St_0 \leftarrow 500$	NA	carga alta de reg.
	<i>lw</i>	<i>lw \$r0, \$r1</i>	$Sr_0 \leftarrow memory[St_1]$	NA	busca <i>word</i> na memória
	<i>sw</i>	<i>sw \$r0, \$r1</i>	$memory[St_0] \leftarrow Sr_1$	NA	carrega <i>word</i> na memória
	<i>lb</i>	<i>lb \$r0, \$r1</i>	$Sr_0 \leftarrow memory[St_1]$	NA	busca <i>byte</i> na memória
	<i>sb</i>	<i>sb \$r0, \$r1</i>	$memory[St_0] \leftarrow Sr_1$	NA	carrega <i>byte</i> na memória
	<i>mov</i>	<i>mov \$r0, \$r1</i>	$Sr_1 \leftarrow Sr_0$	NA	copiar entre registradores
Desvio Condicional	<i>juv</i>	<i>juv 100</i>	<i>if v = 1 then \$pc ← 100 × 2 + \$pc</i>	NA	desvia se <i>overflow</i>
	<i>jnv</i>	<i>jnv 100</i>	<i>if v = 0 then \$pc ← 100 × 2 + \$pc</i>	NA	desvia se não <i>overflow</i>
	<i>jz</i>	<i>jz 100</i>	<i>if z = 0 then \$pc ← 100 × 2 + \$pc</i>	NA	desvia se zero
	<i>jnz</i>	<i>jnz 100</i>	<i>if n = 1 then \$pc ← 100 × 2 + \$pc</i>	NA	desvia se não zero
	<i>jc</i>	<i>jc 100</i>	<i>if c = 1 then \$pc ← 100 × 2 + \$pc</i>	NA	desvia se <i>carry</i>
	<i>jnc</i>	<i>jnc 100</i>	<i>if c = 0 then \$pc ← 100 × 2 + \$pc</i>	NA	desvia se não <i>carry</i>
Desvio Incondicional	<i>jmp</i>	<i>jmp 100</i>	$\$pc \leftarrow 100 \times 2 + \$pc$	NA	desvia para endereço

<sup>1</sup> Indica os *flags* de sinalização que são afetados pela instrução

## 2.2 Técnicas para escalar desempenho em caminhos de dados

Esta seção no Capítulo 2 tem como objetivo documentar os modelos de organização arquitetural proposto pelos projetistas de hardware ao longo da história dos microprocessadores, a fim de escalar mais desempenho. Abaixo são descritos os principais modelos de máquina, dando início ao monociclo que foi o primeiro deles, e chegando ao modelo VLIW que é considerado como um dos mais recentes da atualidade [FIS05].

### 2.2.1 Maquinas monociclo

O modelo de maquina monociclo se caracteriza como o primeiro modelo utilizado na fabricação dos primeiros microprocessadores da história [HEN03]. A idéia chave de uma maquina monociclo esta relacionada diretamente com o ciclo de clock onde todas as instruções, independente aquelas que levam menos tempo, deverão ser adaptadas à instrução que levar o maior tempo para ser executada no caminho de dados do processador [HEN03]. Este modelo de maquina apresenta a desvantagem da perda de desempenho na execução das instruções que levam menos tempo, pois como o ciclo de clock é único para todas as instruções, o tempo de execução total das instruções do programa assembly será diretamente afetado.

### 2.2.2 Maquinas multiciclo

Em uma proposta de evoluir o modelo monociclo frente a suas limitações, surgiu o modelo multiciclo, que tem como idéia chave a introdução de registradores em pontos estratégicos do caminho dos dados, com o objetivo de armazenar a informação que por ali passar [HEN03], até que o ciclo de clock não seja alterado, provendo assim a capacidade do microprocessador em poder trabalhar com um ciclo de clock menor, dando a possibilidade que cada instrução execute baseado em seu numero de ciclos necessários para caminhar por entre o caminho de dados e realizar sua operação [HEN03]. Sendo assim, o problema encontrado no modelo monociclo é eliminado, pois instruções que executam em menos tempo agora podem terminar antes, e disponibilizar o caminho de dados para que, no tempo em que o processador ficasse totalmente ocioso, outra instrução possa estar utilizando o processador.

### 2.2.3 Maquinas Pipeline

O conceito de pipe veio para atender a alta demanda de desempenho, os projetistas verificaram que pela simples inclusão de barreiras temporais, através de registradores, era possível quebrar a execução de uma instrução em estágios, divididos através das barreiras temporárias [HEN03]. Assim poderiam existir instruções sobrepostas, executando de forma concorrente pelos recursos do processador [HEN03]. Com esta técnica, quase todo o caminho dos dados é utilizado por varias instruções concorrentes a cada ciclo de clock, mantendo em teoria todas as unidades operacionais ocupadas.

Porem como as instruções manipulam dados na memória de dados e no banco de registrador, e estruturas físicas como a própria memória de dados ou a unidade lógica, como também estruturas de controle como o contador de programa, diversas instruções sobrepostas executando de maneira concorrente pelos recursos citados acima, podem causar certos conflitos em tempo de execução, que alteram a lógica do código assembly que esta sendo executado. Para evitar ou tratar estas dependências, os projetistas utilizam diversos mecanismos, alguns implementados via software, e outros via hardware, que serão explorados com maior profundidade nas próximas seções deste trabalho.

#### 2.2.4 Maquinas Superescalares

Uma maquina superescalar nada mais é que um conjunto de pipelines onde existe somente um contador de programa em comum que busca um conjunto de instruções capaz de prover instruções suficientes para que ambos os pipes consigam a cada ciclo de clock buscar e entregar uma instrução. Buscar e entregar uma instrução a cada ciclo, caracteriza tanto para um modelo pipeline como para um modelo superescalar como sendo o contexto ideal, que dificilmente é obtido pelas dependências citadas anteriormente, que podem acontecer tanto em um pipeline que executa instruções sobrepostas, como em dois pipes justapostos executando duas ou mais instruções paralelas.

A primeira tecnologia que foi encontrada pelos projetistas, e aquela que causa a maior perda de desempenho é colocar uma instrução de não operar "nop" com a finalidade de corrigir a dependência, pois a nop faz com que exista uma distancia temporal de uma instrução entre as duas instruções que contem a dependência, garantindo que não ira existir mais dependência entre as instruções, o que permite que o programa consiga ser executado corretamente executando de forma concorrente dentro de cada pipe e paralelamente entre os pipes no modelo superescalar. Como o numero de dependências entre as instruções são altas, tanto na parte dos dados quanto na parte do controle, é facilmente entendido que o aumento de complexidade envolvendo um modelo de maquina superescalar traria pouco desempenho em aplicações quando feito a execução na ordem original do código assembly.

Nas próximas seções serão exploradas quais estratégias foram encontradas para aumentar o paralelismo em nível de instrução, explorando a execução paralela de instruções em blocos de código assembly mesmo existindo dependências, buscados em ordem, executados fora de ordem e entregues em ordem.

### 2.2.5 Maquinas VLIW

De uma forma geral uma maquina VLIW se assemelha a uma maquina superescalar, porem como será explicado posteriormente na próxima seção, é uma maquina que realiza o escalonamento das instruções estaticamente em tempo de montagem do código intermediário gerado pelo compilador da linguagem nativa do programa fonte, diferente de uma maquina superescalar que realiza o processo de escalonamento de instruções em tempo de execução. A principal vantagem da maquina VLIW é poder contar com diversos algoritmos de predição de desvios e especulação de dados executados, onde mesmo não tendo à mesma flexibilidade que uma maquina dinâmica tem, contem grande percentual de acerto baseado na simulação da especulação de desvios e da reordenação do código feita em tempo de compilação. Já no modelo superescalar, existem varias técnicas, porem somente algumas poderão ser implementadas, muitas vezes por questão de custo, desempenho e do aumento da complexidade em nível de projeto para realizar a predição dos desvios e o tratamento da dependência de dados.

## 2.3 Paralelismo em nível de instrução

Paralelismo em nível de instrução ou ILP nada mais é que a resultante da sobreposição de instruções em um pipeline ou a paralelização em um modelo superescalar para melhorar o desempenho [HEN03]. De acordo com os autores a quantidade de paralelismo disponível dentro de um bloco básico, sem desvios é ainda assim bastante pequena onde em comparação com o ISA do processador MIPS, sobre os programas típicos escritos para ele, à frequência de desvios esta sempre entre 15 a 25% do código. Com isso o autor conclui que é necessário explorar o paralelismo através de blocos, para que uma implementação pipeline ou superescalar seja aproveitada na pratica.

Uma das necessidades para determinar o quão paralelo é o código que esta a ser executado é determinar a forma como uma instrução pode depender da outra e se obter paralelismo sobre estas instruções será ou não possível [HEN03]. Existem 3 tipos diferentes de dependências que são explicadas separadamente abaixo:

- ❖ Dependência de dados: Esta dependência também conhecida na literatura por “dependência verdadeira”, e referenciada neste Tcc como RAW (Read after write), que significa que existe uma dependência sempre que uma instrução anterior efetua uma

escrita e a instrução posterior uma leitura sobre um mesmo dado, que no caso do AIDA-16 poderia ser tanto uma posição da memória de dados como do banco de registrador.

- ❖ Dependência de nomes: Uma dependência de nome acontece quando duas instruções usam um mesmo registrador ou uma mesma posição de memória, onde se denomina nome mas não existe nenhum fluxo de dados entre as instruções associadas com esse nome [HEN03]. Há dois tipos de dependências de nome entre instruções:
  - Anti-dependencia: Esta dependência ocorre quando a instrução grava um registrador ou uma posição de memória que a lê [HEN03]. Esta dependência é referenciada neste Tcc como WAR (Write after read), que significa que existe uma dependência sempre que uma instrução anterior efetua uma leitura e a instrução posterior uma escrita sobre o mesmo dado, que no caso do AIDA-16 poderia ser tanto uma posição de memória e dados como do banco de registrador.
  - Dependência de saída: Esta dependência ocorre quando ambas as instruções gravam no mesmo registrador ou na mesma posição de memória [HEN03]. Esta dependência é referenciada neste Tcc como WAW (Write after write), que significa que existe uma dependência sempre que uma instrução anterior efetua uma escrita e a instrução posterior uma escrita sobre o mesmo dado, que no caso do AIDA-16 poderia ser tanto uma posição de memória de dados como do banco de registrador.

Tanto a Anti-dependencia como a dependência de saída são dependências de nome, em oposição às dependências de dados verdadeiras, pois não existe nenhum valor sendo transmitido entre as instruções, sendo que as instruções envolvidas em uma dependência de nome podem ser executadas ao mesmo tempo ou serem reordenadas se o nome (numero do registrador no banco ou endereço da memória de dados) usado nas instruções for alterado de forma que as instruções não entrem em conflito onde o renomeamento pode ser feito em tempo de compilação ou em tempo de execução [HEN03].

- ❖ Dependência de controle: Uma dependência de controle determina que só seja conhecido para onde o contador de programa deverá buscar as próximas instruções depois que a instrução for executada e com base em seus resultados decisões possam

ser tomadas. Todas as instruções que tiverem dependência de controle devem ser preservadas em sua ordem original para manter a ordem correta do programa [HEN03].

Para superar estas dependências, existem duas abordagens que podem ser utilizadas: a abordagem baseada em escalonamento dinâmico e a abordagem baseada em escalonamento estático, que são explicadas separadamente abaixo:

❖ Escalonamento dinâmico de instruções e previsão dinâmica de desvios

Esta técnica consiste no hardware realizar a reorganização da execução de instruções para reduzir as paradas "nops", tentando com isso manter ao máximo possível o fluxo de instruções executando sem dependências e o comportamento contínuo da execução. O escalonamento dinâmico oferece diversas vantagens como o tratamento de casos em que às dependências são desconhecidas em tempo de compilação, podendo simplificar o compilador, porém encarecendo o hardware. Para obter ganhos, o escalonamento é feito com base na busca em ordem e na execução fora de ordem, o que implica em concluir fora de ordem, que poderá causar outros conflitos, resolvidos através do renomeamento de registradores em tempo de execução. Uma abordagem fundamental, utilizada com sucesso para realizar o escalonamento dinâmico no passado e nas máquinas de alto desempenho atuais é o conceito chave de controlar as dependências através do algoritmo de Tomasulo criado por Robert Tomasulo, que funciona controlando quando os operandos estão disponíveis a fim de minimizar os conflitos de dados introduzindo o renomeamento de registradores com o objetivo de controlar as dependências de nome [HEN03].

Outra técnica importante é a previsão de desvios implementada dinamicamente em hardware que tem como objetivo tratar as dependências de controle que prejudicam a exploração do ILP através dos blocos de dados [HEN03]. A ideia é utilizar o hardware para prever dinamicamente o resultado de um desvio previamente buscado onde a previsão feita será feita uma pré busca de dados para o endereço apontado pelo desvio, que se caso executado e a previsão tiver acertado, fará com que o pipeline ou o processador superescalar não perca tempo buscando as instruções, pois estas já estão fora buscadas antes do desvio concluir. Caso a previsão não acerte, o hardware deverá desfazer as instruções que buscou nesse intervalo onde o ganho envolvendo essa previsão será nulo [HEN03].

#### ❖ Escalonamento estático de instruções e previsão estática

Esta técnica consiste no compilador em realizar a verificação de quaisquer dependências entre instruções no encapsulamento a ser emitido, bem como entre qualquer emissão candidata. Todo ganho de desempenho deve estar significativamente associado ao compilador, que em contraste a um superescalar com escalonamento dinâmico está associado ao hardware. Com um compilador preparado para verificar as dependências é possível não apenas tratar das dependências, mas também formatar as instruções em encapsulamentos potenciais, de forma que o hardware fique isento de verificar as dependências realizando reordenamento de registradores e troca de instruções sem alterar a lógica do programa, eliminando as potenciais dependências [HEN03].

Fora isso, o compilador também terá capacidade de especular os desvios efetuando uma simulação de diversas variações nos dados das instruções, prevendo a media que um determinado desvio pode acontecer, marcando isto a um flag no campo do encapsulamento das instruções, que será lido pelo hardware e assumido como resultado do desvio, que se caso acontecer de executar e a especulação estática der errada, o hardware deverá implementar mecanismos de recuperar o contexto desde o momento em que o desvio aconteceu[HEN03].

## **2.4 Estudo da abordagem VLIW**

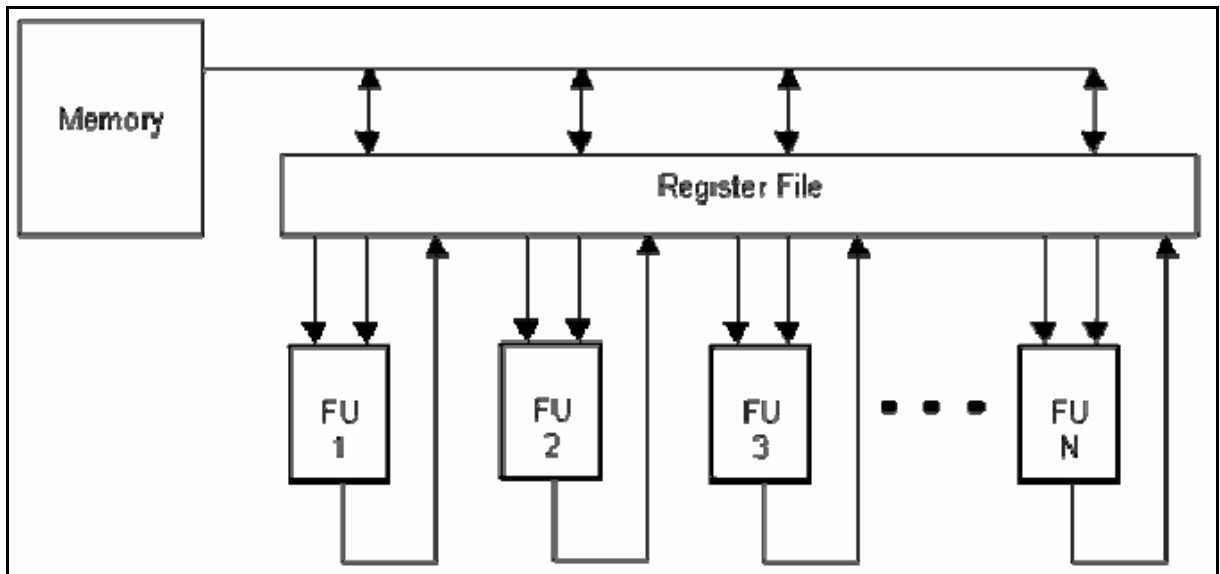
Durante vários anos acreditou-se que a quantidade de paralelismo em nível de instrução que poderia ser explorado nos programas existentes não justificaria os esforços para prover recursos para suportar a execução simultânea de diversas instruções [ITO98]. Entretanto, estudos apresentados por [FIS05] mostraram que a utilização de certas técnicas tanto em hardware como em software podem aumentar a quantidade de paralelismo a ser explorado. Maquinas superpipeline e superescalares têm sido concebidas para explorar esse tipo de paralelismo, obtendo resultados consideráveis, entretanto, apesar dos avanços obtidos, atingir alto CPI ainda é um desafio. Uma abordagem alternativa é a VLIW (Very Long Instruction Word) que conta com sua estrutura de controle extremamente simplificada, não possuindo suporte algum para escalonamento dinâmico de instruções e tratamento de dependências. A filosofia VLIW vai contra ao que é utilizado normalmente, propondo a utilizar técnicas de escalonamento estático em nível de compilador, provendo emissão

múltipla de instruções escalonadas em ordem, onde não existam dependências entre as instruções.

A abordagem VLIW basicamente é apoiada sobre dois grandes conceitos, que são o micro código horizontal provido pelas longas instruções e processamento superescalar provido por um caminho de dados capaz de executar diversas instruções em um único ciclo de máquina. A organização da máquina VLIW é composta por um grande banco de registradores compartilhado entre as unidades funcionais, responsáveis por realizar a execução simultaneamente de todas as instruções que são sincronizadas unicamente em uma longa instrução. A ideia aqui é explorar o paralelismo em nível de instrução, de forma que seja possível reduzir o número de instruções e assim reduzir o tempo total para executar o programa.

Podemos classificar de acordo com a classificação de flynn que a abordagem VLIW é um meio termo entre SIMD e MIMD [ITO98], pois vindo por SIMD ela permite a execução simultânea de diferentes operações a partir de um único fluxo de instruções, e vindo por MIMD, pois possui diversas unidades de execução executando diferentes operações ao mesmo tempo. Porém como foi comentado é um meio termo, pois no modelo SIMD a cada momento, uma operação é executada nos diversos elementos de processamento sobre diversos dados, e no modelo MIMD cada unidade de processamento é um processador independente com seu próprio contador de programa, de forma que não há sincronismo no processamento das instruções. Por ter características diferentes de todas as classificações existentes por flynn em [RAU89] a arquitetura VLIW é classificada como SIMOMD (Single Instruction, Multiple Operation, Multiple Data). Para tirar proveito de todas as unidades funcionais disponíveis é necessário um compilador para realizar o escalonamento estático de forma que as instruções sejam independentes e não causem conflitos e mantenham a semântica do programa durante a execução. Abaixo é demonstrado um modelo de máquina VLIW ideal:





*Figura 2.6: Modelo de máquina VLIW ideal*

#### 2.4.1 Histórico da abordagem VLIW

Oficialmente a história das máquinas VLIW inicia-se no final da década de 70, mas precisamente no ano de 1979 na universidade de Yale, quando Joseph Fisher descreveu a técnica de trace scheduling, que posteriormente foi utilizado para compilação de programas escritos em linguagens de programação convencionais para máquinas com longas palavras de instruções. No ano de 1984 Bob R. Rau e outros fundaram a companhia Cydrome, Inc para construir máquinas utilizando a filosofia VLIW e com suporte arquitetural para loops escalonados pela técnica de software pipeline. Neste mesmo ano a multiflow computer, Inc foi fundada com intuito de também construir máquinas baseadas na abordagem VLIW. Em 87 a cydrome entrega o cydra 5 que possuía um simples processador numérico VLIW com encapsulamento de 256 bits podendo disparar até 7 instruções por ciclo de máquina. Também em 87 a multiflow entrega o Trace/200 capaz de empacotar diversas instruções com palavras longas entre 256 até 1024bits. Ambas as empresas fecham suas portas, a cydrome em 88 e a multiflow em 90. Embora não obtendo sucesso comercial, a experiência na construção da organização e das técnicas em compiladores não foram perdidas, a multiflow foi comparada pela companhia HP e as pesquisas nesta área de compiladores continuaram sendo lideradas por Rau e Fisher agora na Hp.

#### 2.4.2 Vantagens da arquitetura VLIW

- ❖ Arquitetura regular e dependente do compilador, gerando pouca restrição no acesso aos recursos do processador, permitindo que o escalonamento estático das instruções seja feito com grande liberdade a fim de obter possíveis otimizações por parte de desempenho.
- ❖ Capacidade de realizar escalonamento que permita obter emissão múltipla de instruções, atingindo alto CPI.
- ❖ Mantêm o controle do hardware o mais simples possível, permitindo que se obtenha em teoria máquinas mais simples, que consuma e dissipe menos energia, e que possam prover nesse percentual de área um controle quase que inexistente, mas unidades funcionais capazes de prover mais CPI.
- ❖ Como o compilador reconhece as características do processador (Clock global do sistema/latência de cada unidade funcional), é possível obter o efeito de determinadas modificações na ordem e na estrutura da execução das instruções, e assim estar apto a resolver conflitos e dependências em tempo de compilação.

#### 2.4.3 Desvantagens da arquitetura VLIW

- ❖ Como a previsão dos desvios é feita em tempo de compilação, existem técnicas capazes de especular para onde possivelmente o desvio deverá apontar porém estas técnicas são especulativas, fazendo com que se tenham grandes perdas de desempenho quando a previsão é incorreta.
- ❖ Dependendo a situação de falha ou interrupção, o pior caso (parar o(s) pipes) sempre deverá ser tomado, pois como o compilador não sabe das eventuais falhas de execução, o processador não terá mecanismos dinâmicos para estudar as falhas.
- ❖ Exibe problemas de compatibilidade de código quando o clock global é modificado ou a latência das unidades funcionais é alterada.
- ❖ Apresenta pior densidade de código quando muitos NOPs são codificados na instrução longa, levando a uma má utilização da memória.
- ❖ Requer uma cache capaz de prover o mesmo número de portas para todas as unidades funcionais.

#### 2.4.4 Técnicas VLIW associadas ao compilador

O grande diferencial da abordagem VLIW para se obter desempenho é explorar o paralelismo em nível de instrução em máquinas com múltiplas unidades funcionais afim de que tais permaneçam a maior parte do tempo ocupadas. Este tem sido o principal desafio dos projetistas atualmente, do qual originaram uma série de técnicas tanto em software (escalonamento estático) quanto em hardware (bypassing). No caso dos processadores VLIW a arquitetura é exposta ao compilador onde a tal foi acrescida toda a responsabilidade de escalonar as instruções, para manter as unidades do processador trabalhando o máximo possível em tempo de execução. Abaixo algumas técnicas VLIW são exploradas mais a fundo:

##### ❖ Loop Unrolling

Para manter o *pipeline* cheio é necessário encontrar instruções independentes para serem colocadas a executar no *pipeline* sem ocasionar conflitos. Instruções independentes podem ser sobrepostas, desde que separadas por uma distância em ciclos que corresponda à latência da instrução fonte. O loop unrolling consiste em diminuir o número de interações estendendo o código contido em seu corpo, diminuindo o overhead de controle do loop e permitindo escalonar instruções que pertenciam a interações distintas. Para implementar loop unrolling é necessário conhecer o número de interações em tempo de compilação, bem como em ter registradores disponíveis para armazenar os resultados intermediários da interação. Em uma máquina VLIW o fator unrolling deverá ser bem maior para obter ganhos significativos.

##### ❖ Trace Scheduling

Esta técnica foi proposta por Joseph Fisher em 81 para compactação de operações de programas em micro instruções horizontais, tornando-se uma das principais técnicas de compactação de código utilizada em máquinas VLIW. O conceito de compactação nada mais é que a conversão de micro código seqüencial em micro código paralelo. Existem várias técnicas de compactação, sendo as locais as que atuam sobre blocos básicos e as globais as que atuam sobre trechos que extrapolam os limites básicos. O uso de trace scheduling é conveniente quando o loop unrolling não é suficiente para extrair ILP suficiente para manter as unidades funcionais trabalhando.

Em resumo o trace scheduling consiste em tomar o trace com maior probabilidade de execução dentre os traces não compactados, construindo um gráfico de direção sem loops e compactado. Durante a compactação operações podem ser movidas ou inseridas em outros traces não compactados, como forma de manter a semântica do programa ou como código de compensação. Sendo assim o trace possibilita explorar paralelismo muito além daquele contido em blocos básicos já que constitui em uma técnica de compactação global.

#### ❖ Software Pipeline

A técnica de software pipeline consiste em reorganizar a estrutura do loop selecionando instruções de diferentes iterações para diminuir conflitos e permitir o despacho de um numero maior de instruções. Essa técnica tem como seu correspondente em hardware o algoritmo de Tomasulo implementado em maquinas como o IBM 360/91. Em muitos casos o software pipeline é utilizado em conjunto com o loop unrolling onde cada uma elimina um tipo de overhead. O loop unrolling reduz o overhead das instruções de controle do loop enquanto que o software pipeline reduz o numero de vezes em que o loop não esta com o overlap máximo de instruções para apenas duas situações, correspondente ao inicio e o fim do loop. Utilizando software pipeline o tamanho do código tende ao aumentar em relação ao original, entretanto este código deve ser incluído para tratar casos de um numero muito pequeno de interações e que não é conhecido em tempo de compilação.

## **2.5 Técnicas de Encapsulamento de instruções em organização da arquitetura de processadores VLIW**

A organização de uma arquitetura VLIW é bem simples, entretanto um processador pode ter suas unidades funcionais totalmente pipelinezadas, podendo implementar diferentes tipos de codificação de palavra e ter suporte ao tratamento de exceções e implementar bypassing. Entretanto a idéia é ter um hardware o mais "simples" possível, deixando com que toda complexidade de controle fique a cargo do compilador e não do processador, porem a sua incorporação pode trazer diversos benefícios e solucionar eventuais problemas. Abaixo alguns mecanismos da organização da arquitetura de um processador VLIW

### 2.5.1 Pipeline em processadores VLIW

O pipeline em processadores VLIW tem características particulares em relação às maquina superescalares, embora ambos tenham a capacidade de emissão múltipla de instruções. Em um pipeline VLIW a busca é feita através de um único endereço, porém cada instrução longa buscada contém diversas outras instruções. A decodificação é simples, onde cada opcode de cada instrução especifica a operação que deverá ser executada sobre os dados. Na execução as operações são despachadas para as unidades funcionais correspondentes não havendo mecanismos que detectem ou trate as dependências, que fica estaticamente a cargo de um compilador, que tenha conhecimentos necessários sobre a organização do processador, como o numero de estágios do pipeline, o numero de unidades funcionais, a latência de cada unidade funcional e o clock global do sistema.

Toda eficiência global do processador VLIW fica atrelada à eficiência do escalonamento do compilador. A densidade de código gerada por uma maquina VLIW é pior do que as maquinas superescalares, pois nesta técnica o formato das instruções longas precisa codificar NOPs quando instruções não podem ser despachadas simultaneamente, o que em maquinas superescalares, todo código contem somente instruções a serem executadas. Outro serio problema é a incompatibilidade de código objeto entre maquinas VLIW de outras gerações, pois na medida em que se altera um aspecto da organização da maquina (numero de estágios) ou clock global do sistema não poderemos reaproveitar código escritos anteriormente para outro processador VLIW para essa nova organização [FIS05].

### 2.5.2 Mecanismo de fetch

Maquinas VLIW trabalham com longas palavras de instruções, e para isso requerem mecanismos de fetch de instruções que suportem elevada largura de banda. A importância de ter um mecanismo de fetch que atenda a essa exigência é clara, sendo que o limite da largura de banda das caches de instruções pode passar a ser um gargalo nesse tipo de arquitetura. Basicamente as instruções VLIW podem ser classificadas em relação a sua codificação em:

- ❖ Uncompresses encoding: Formato de instruções com tamanho fixo em bits codificando explicitamente NOPs quando uma operação não pode ser escalonada na instrução, possuindo como vantagem um mecanismo mais simples para fetch de instruções.
- ❖ Compressed encoding: Formato de instruções com tamanho variável, não escalonando NOPs na instrução, possuindo como vantagem instruções longas mais compactadas em detrimento da memória e exigindo em algumas situações menos largura de banda, porem requer uma unidade de fetch mais complexa por parte do processador [ITO98].

### 2.5.3 Bypassing

Esta técnica, também conhecida como forwarding é bastante utilizada em processadores RISC para resolver dependências de dados. É considerada uma relação custo x benefício aceitável, pois traz grandes benefícios causando pouco overhead para ser implementada. Para inserir forwarding em maquinas VLIW a complexidade aumenta, pois temos n unidades funcionais, o que demanda maior complexidade para rotear a saída das n unidades funcionais para as suas entradas respectivas [ITO98].

## 2.6 Comparativo entre a organização VLIW x Superescalar

Esta seção tem como objetivo efetuar um breve resumo comparativo entre um modelo de maquina VLIW que como já foi comentado anteriormente, que tem como base trabalhar com emissão múltipla de instruções e realizar o escalonamento destas instruções em tempo de compilação, e de uma maquina superescalar que também trabalha com emissão múltipla de instruções, porem implementando o escalonamento de instruções dinamicamente em tempo de execução [HEN03].

Maquina VLIW :

Realiza a emissão de instruções em longos encapsulamentos, onde toda detecção das dependências é feita estaticamente em tempo de compilação através do escalonamento estático das instruções nos encapsulamentos. O principal objetivo de uma maquina VLIW é prover nenhuma dependência no encapsulamento de instruções, simplificando ao extremo o caminho de dados e o controle do microprocessador VLIW, como também especular através de técnicas os desvios condicionais no código assembly [HEN03].

Maquina Superescalar:

Em uma maquina superescalar a emissão de instrução se dá dinamicamente onde toda detecção das dependências é feita dinamicamente por estruturas do hardware que trabalhando com base em escalonamento dinâmico de instruções é capaz de aplicar técnicas como o algoritmo de Tomasulo que realiza a execução fora de ordem, e o outro objetivo é especular, através de técnicas implementadas em hardware, os desvios condicionais no código assembly [HEN03]. O principal objetivo de uma maquina superescalar é prover o máximo de instruções despachadas a cada ciclo de clock, mantendo o CPI sempre igual ao numero de pipes do caminho de dados.

## 2.7 Estudo introdutório sobre ISS

ISS ou Instruction Set Simulator é um programa com a capacidade de executar o comportamento de um conjunto de instruções de um processador podendo executar programas escritos ou compilados para estes processadores que não existem na pratica ou ainda não foram construídos por questões de c, e por isso são primeiramente simulados. Um ISS é normalmente escrito em C e modela todas as funcionalidades do processador no nível de instruções. Todos os registradores internos são simulados corretamente, porém o tempo não pode se considerado. Estes modelos apresentam um grande desempenho, sendo ideais para fases iniciais da co-simulação [WAG99].

Basicamente existem dois modelos simulação existentes:

- Usando a implementação real
- Usando um modelo em nível de transistor RTL do processador
- Implementação através de uma linguagem de alto nível

Estes métodos têm desvantagens porque como já comentado acima a implementação real tem um custo associado e em nível de RTL existe o overhead da simulação, onde para

resolver estes problemas são aplicados os ISS implementado em linguagens de alto nível como C com a função de representar a funcionalidade do processador e não mapear os detalhes internos da arquitetura, simulando entre 4 a 5 vezes mais rápido comparado ao RTL [MIR09].

## **2.8 Estudo introdutório sobre Montadores**

Um montador é uma ferramenta para o desenvolvimento de programas que devam acessar os recursos da arquitetura de um processador sendo responsável por traduzir as instruções em linguagem de montagem para as instruções binárias do processador, permitindo que os programas sejam expressos em uma representação muito mais amigável que a representação de 0s e 1s da máquina [PAT00].

O principal papel da linguagem de montagem é dispor de uma linguagem que seja possível escrever programas sem recorrer a codificação binária. Porém esta linguagem acabou sendo substituída, uma vez que atualmente a maioria dos programadores prefere escrever seus programas em linguagem de alto nível em função de atualmente não existir fortes restrições de memória e processador para grande parte das aplicações, mesmo as de alta complexidade desenvolvidas atualmente [PAT00].

A linguagem de montagem é uma representação simbólica da codificação binária de um processador sendo mais legível que a linguagem de máquina pois utiliza símbolos em vez de bits onde os símbolos são associados a padrões de bits da ocorrência freqüente, tanto dos códigos de operação e dos identificadores de registradores de modo que as pessoas leiam essas informações e possam lembrar-se delas depois [PAT00].

A linguagem de montagem permite que os programadores utilizem labels para identificar e dar nomes a determinadas palavras de memória que guardem instruções ou dados. Atualmente poucas pessoas conhecem a linguagem de montagem, que ainda é utilizada quando precisamos escrever programas nos quais o tamanho e a velocidade sejam parâmetros críticos da aplicação, como também precisar explorar características do hardware que não são acessíveis aos programadores pelas linguagens de alto nível.

A linguagem de montagem deverá ser utilizada nos seguintes momentos:

- No tempo de aprendizado de disciplinas como organização e arquitetura de computadores, para que o estudante tenha conhecimento da existência da linguagem de



montagem como através dela aprenda de fato como o conjunto de instruções do processador é utilizado na prática

- Quando requisitos não funcionais como tamanho do código ou velocidade de execução são fatores críticos para o sucesso de um projeto, sendo esta a principal razão para se trabalhar em linguagem de montagem em vez de se utilizar uma linguagem de alto nível, capaz de abstrair a dificuldade da programação em nível de montagem frente à complexidade da aplicação.

A fim de obter o melhor da programação em nível de montagem e da programação em alto nível, normalmente os projetos são sub-divididos em duas partes críticas: uma considerando diretamente sobre a linguagem de montagem e outra considerando a linguagem de alto nível, uma vez que esta possui menos restrições críticas [PAT00].

Normalmente os compiladores são mais eficientes que os programadores em nível de montagem, na medida em que produzem código compactado, de alta qualidade e otimizado ao longo de todo o programa. Porém como todo algoritmo é artesanal, o compilador ainda assim não substitui a figura do programador, sendo somente ele capaz de adaptar detalhes específicos dos requisitos da aplicação ao código assembly [PAT00].

Uma última porém significativa razão para se trabalhar com linguagem de montagem, que traduz exatamente o contexto deste TCC é o de se utilizar de linguagem de montagem para gerar código para o processador quando não existe linguagem de alto nível para este conjunto de instruções [PAT00]. Como o processador AIDA-16 ainda é recente, a linguagem de montagem é um primeiro degrau para que linguagens de alto nível sejam concebidas futuramente a fim de aumentar o poder de desenvolvimento de software para o conjunto de instruções AIDA-16.

Abaixo são citadas algumas desvantagens do uso de uma linguagem de montagem:

- Programas escritos em uma linguagem de montagem são específicos para uma determinada máquina, onde o algoritmo deverá ser completamente reescrito para poder executar em uma máquina contendo uma organização diferente.
- Atualmente as arquiteturas de computadores tendem a se tornar obsoletas rapidamente, o que exige que o algoritmo não seja atrelado à arquitetura da máquina.
- Programas escritos em uma linguagem de montagem normalmente são maiores que os seus equivalentes de linguagem de alto nível, e é comprovado que os programadores tem a mesma capacidade de escrever o mesmo número de linhas, tanto em uma

linguagem de alto nível como em uma linguagem de montagem, que resulta em uma gigantesca perda de produtividade em se trabalhando em linguagem de montagem comparado a uma linguagem de alto nível [PAT00].

- Outro aspecto negativo é que construções muito utilizadas por linguagens de alto nível como estruturas de seleção e de repetição devem ser construídas em linguagem de montagem com a utilização de desvios condicionais e desvios incondicionais, resultado em programas de difícil leitura e de depuração em caso de erros.

O processo de tradução do código assembly ocorre com a procura às diretivas de montagem onde depois é chamado o ligador que ira combinar com um conjunto de arquivos objeto e rotinas da biblioteca de programas, formando um programa executável.

O primeiro passo depois de ler o arquivo é ler cada uma das linhas do arquivo de instruções e dividindo-as em unidades léxicas os campos da instrução. Depois de fazer este processo em todas as linhas ele ira gerar o arquivo objeto que contem o cabeçalho do arquivo-objeto, o segmento dos dados, as informações sobre Relocação, a tabela de símbolos e informações sobre análise de erros [PAT00].

Algumas das facilidades encontradas com um montador são as diretivas de montagem, também conhecidas como pseudo instruções ou comandos do montador, sendo usadas como :

- ✓ Posição inicial do código objeto
- ✓ Fim do código objeto
- ✓ Posição inicial das constantes de memória
- ✓ Declaração de constantes de dados

Estas diretivas de montagem não geram código executável e não devem ser confundidas, nem pelo programador nem pelo desenvolvedor do montador com instruções do processador, servindo unicamente como comandos para informar ao montador algumas opções do programador para facilitar a escrita do código assembly.

Todo código assembly contem duas áreas de código, uma para declarações das instruções do programa fonte e outra para a declaração das constantes de dados, sendo separadas através das diretivas de montagem como já comentadas acima [PAT00].

O montador também pode reconhecer alguns tipos de dados, sendo elas as constantes as variáveis e os labels, onde uma constante é um nome associado a um numero sem qualquer atributo, uma variável é uma identificação a um objeto manipulável capaz de formar os operandos para as instruções de manipulação de dados e os labels capazes de identificar os operandos das instruções de saltos condicionais, incondicionais e chamadas de sub-rotinas.

### **3 DESENVOLVIMENTO DA INFRA-ESTRUTURA DE SUPORTE VLIW**

Este capítulo tem por objetivo documentar o desenvolvimento da infra-estrutura de suporte VLIW para o processador AIDA-16. Abordada em duas etapas do desenvolvimento: (i) apresenta o processo de especificação e implementação do montador e simulador tendo por base a arquitetura convencional do processador; (ii) estende o desenvolvimento realizado na primeira etapa de forma a disponibilizar o suporte VLIW tanto para o montador quanto para o simulador do conjunto de instruções.

#### **3.1 Especificação e implementação do montador**

A primeira etapa do presente TCC se inicia a partir do desenvolvimento do montador para a arquitetura convencional do processador, onde inicialmente foram realizados estudos sobre a arquitetura do conjunto de instruções de forma a identificar quais os requisitos necessários para implementação do montador para suportar a ISA do processador AIDA-16. Este estudo se deu a partir dos seguintes tópicos que seguem abaixo.

##### **3.1.1 Estudo do ISA do processador AIDA-16**

A partir de um breve estudo feito na especificação do processador AIDA-16 foi possível obter as seguintes informações mais significativas para o montador:

- ❖ a ISA do AIDA-16 possui 30 instruções;
- ❖ estas instruções estão divididas em 4 classes de instruções que são:
  - ✓ Formato – R: é a primeira classe de instruções, contém 15 instruções que variam conforme o modo de endereçamento, sendo responsáveis por implementar as operações lógicas e aritméticas do AIDA-16;
  - ✓ Formato – I: é a segunda classe de instruções, contém duas instruções que executam a carga de constantes para os registradores do banco de registradores.

- ✓ Formato - Load/Store: é a terceira classe de instruções, contendo 5 instruções responsáveis por manipular a memória de dados;
- ✓ Formato – Branch: é a quarta classe de instruções, contém 7 instruções responsáveis por operações de desvios, condicionais e incondicionais, do processador AIDA-16;
- ❖ O processador AIDA-16 possui um banco de registradores contendo 15 registradores de propósito geral, denominados de \$r0 a \$r13 e outros dois registradores, \$t0 \$t1, também de propósito geral porém, adicionalmente utilizados por algumas instruções como registradores específicos de para armazenar resultados das operações;
- ❖ todas as operações efetuadas no AIDA-16 são realizadas diretamente fazendo uso do banco de registradores, assim operações que envolvam dados existentes em memória devem ser precedidas por instruções de carga da memória, operando sobre dados carregados em registradores.

Identificadas estas características pode-se dar início à especificação e desenvolvimento do montador.

### 3.1.2 Especificação das diretivas básicas de montagem

Depois de feito o estudo do AIDA-16, foi necessário definir quais seriam as diretivas de montagem do código assembly. Verificando em [WEB04], foi constatado que diretivas de montagem deveriam existir afim de que o programador explicitamente demarcasse no código fonte qual seria o bloco de instruções, e qual seria o bloco de dados do código. Para isso foi especificado duas diretivas de montagem, descritas abaixo:

- ❖ .text: diretiva usada para informar ao montador à região que contém o código de assembly, delimitada pela ocorrência desta diretiva até a ocorrência da diretiva data;
- ❖ .data: diretiva usada para informar ao montador a região onde estão declarados os espaços para variáveis e as constantes, delimitada pela ocorrência da diretiva até o final do arquivo.

Para que um programador consiga escrever código assembly válido para o montador AIDA-16, é necessário considerar as seguintes regras durante a programação:

- ❖ todo arquivo contendo código assembly deverá conter a extensão de arquivo .asm.
- ❖ o código assembly deve conter explicitamente as diretivas .text e .data, mesmo não utilizados pelo programador.
- ❖ a diretiva .text deverá ser declarada antes da diretiva .data, caso contrario o montador ira indicar uma condição de erro.
- ❖ todo programa assembly deverá conter uma ou mais instruções halt, caso contrario o montado indicará uma condição de erro.
- ❖ comentários são aceitos pelo uso da diretiva de montagem # (cerquilha), onde todo o texto após este símbolo (inclusive) até a ocorrência de uma salto de linha serão considerados como comentários e portanto ignorados no processo de montagem.
- ❖ Uma ou mais linhas em branco são ignoradas no processo de montagem, servindo como forma de aumentar a legibilidade do código assembly para o programador.

Como consta na documentação do AIDA-16, a memória dos dados é endereçada a byte, como tradicionalmente é feito nas arquiteturas de computadores [WEB04]. Devido à palavra do processador AIDA-16 ser de 16bits é necessário criar duas diretivas de montagem para dados: uma para manipular bytes e outra para manipular palavras de 16 bits (2bytes). Abaixo estão listadas as duas diretivas de dados especificadas para prover suporte à declaração de dados:

- ❖ .byte <valor>: diretiva de montagem responsável por reservar e definir valores em memória de um byte;
- ❖ .word <valor>: diretiva de montagem responsável por reservar e definir valores em memória para palavras (2bytes).

Todos os detalhes comentados acima estão inseridos na Figura 3.1, constituindo um exemplo sintético de como as diretivas devem estar dispostas em um arquivo contendo código assembly para AIDA-16.

```

# Arquivo de exemplo de como escrever um programa para o montador AIDA-16 (exemplo.asm)
# Eu sou um comentário
.text # Responsável por indicar que logo abaixo estarão às instruções do código fonte
add 00,0,$r0,$r1 # Instrução responsável por somar o registrador $r0 com o registrador $r1 e
armazenar o resultado no registrador $r0
halt
.data # Responsável por indicar que logo abaixo estão às declarações de dados do código fonte
.word 5 # Instrução que aloca o dado 5 na célula 0 e na célula 1 da memória do ISS
.byte 2 # Instrução que aloca o dado 2 a célula 2 da memória do ISS

```

*Figura 3.1 : Exemplo das diretivas do montador usadas no código assembly do processador AIDA-16.*

As subseções seguintes descrevem os passos executados para o desenvolvimento do montador, considerando os diversos aspectos a serem considerados durante o processo de montagem, os quais foram divididos em blocos de acordo com a característica ou função.

### 3.1.3 Implementação do bloco de leitura do arquivo assembly

A implementação inicial do montador se dá através das funções responsáveis por efetuar a leitura do arquivo assembly, que é feita através do nome do arquivo passado como parâmetro na linha de comando. Após se obter o nome, o primeiro passo, feito pelo montador, é verificar se o arquivo existe, caso exista é verificado se a extensão do arquivo coincide com .asm, conforme determinam o requisito. Feito estas verificações e estando tudo correto, o montador verifica se existem as diretivas de montagem: *.text* e *.data*; e se *.text* ocorre antes de *.data*, além de verificar se o arquivo assembly possui a instrução *halt*. Caso todas as condições citadas acima forem válidas, o montador então segmenta o arquivo em linhas iniciadas após *.text* e finalizada antes da diretiva *.data*, onde cada uma deve possuir uma instrução assembly. O mesmo é realizado para as diretivas contidas a partir de *.data* gerando linhas referentes aos dados declarados pelo programador. São ignorados os comentários, os espaços e linhas em branco do código assembly. Todas as exceções que invalidem os procedimentos citados acima são relatadas via prompt da linha de comando informando a respectiva situação ocorrida e causando o fim do processo de montagem. Para passar desta etapa o programador deve corrigir as situações informadas podendo então novamente submeter o programa ao montador.

### 3.1.4 Implementação do bloco de análise das instruções

Após o processo de leitura de o arquivo assembly ter sido concluído corretamente, o montador ira executar o processo responsável pela análise léxica de todas as instruções do arquivo. O analisador léxico das instruções lê instrução por instrução e verifica o primeiro campo da instrução, mnemônico, com a finalidade de verificar se este campo contém um identificador de instrução que pertença a um dos 31 códigos de instruções reconhecidos processador AIDA-16. Caso o campo for válido é realizado o procedimento de análise sintática, caso contrário será informado ao usuário que instrução considerada não é válida e o processo de montagem é finalizado.

O próximo passo ocorre caso a instrução seja valida, sendo a análise sintática, que é realizada pelo montador tomando por base o formato de instrução através, identificado durante o reconhecimento da instrução. Reconhecido o formato da instrução é possível saber quantos e quais são os campos existentes na instrução e quais faixas de valores cada campo poderá ter. Uma vez cada formato da ISA do AIDA-16 tem campos e funções diferentes, optou-se por detalhar como é feita a análise sintática de cada formato distinto, iniciado pelo formato R.

#### ❖ Analise sintática do formato - R

O formato R é considerado quando o parâmetro informado pelo analisador léxico for zero, ele então inicia a varredura das instruções a partir do último elemento lido pelo analisador léxico buscando os seguintes elementos referenciados pela instrução do formato R conforme exemplo mostrado abaixo, que servirá para exemplificar o processo:



*Figura 3.2 : Formato – R : Modo de endereçamento direto*

- ❖ **add** :opcode;
- ❖ **am** : modo de endereçamento;
- ❖ **rd** : registrador de destino;
- ❖ **rs** : registrador de origem 1;



❖ **rt** : registrador de origem 2.

O primeiro campo a ser buscado pelo analisador sintático do formato R é o modo de endereçamento. Este campo é obtido pela leitura do segundo campo, logo após o opcode até o separador ','. Identificado o modo de endereçamento, este é verificado semanticamente onde caso for diferente de 00, 01 ou 10, que são os três modos de endereçamento suportados pela instrução do formato R, o montador deverá informar que o modo de endereçamento contido na linha é inválido. Caso o modo de endereçamento lido for válido, o terceiro campo, que é o registrador de destino, será buscado da mesma forma que o modo de endereçamento, através da leitura a contar do separador ',' do modo de endereçamento até o próximo separador ','. Após ler o registrador de destino, é verificado se o valor é igual a 0 ou 1 caso for diferente o montador deverá informar que o registrador de destino é inválido.

O quarto campo, registrador de origem 1, é lido a partir do separador ',' do registrador destino até o próximo separador ','. Após ler este registrador, o montador efetua a verificação para verificar se o campo inicia pelos prefixos '\$r' ou '\$t', onde '\$r' é seguido dos valores 0 a 13 e '\$t' seguido dos valores 0 ou 1 indicando os registradores 14 e 15. Caso os valores forem diferentes de qualquer uma destas combinações o montador irá informar uma condição de erro, registrador inválido, para esta instrução. O quinto e último campo da instrução do formato R é o registrador de origem 2, que será obtido após o último separador ',' que segue o registrador de origem 1. Após ler este registrador, o montador irá efetuar as mesmas verificações semânticas feitas para o registrador de origem. Como se pode concluir, é possível existir falta ou excesso de campos na instrução, exceções que também são verificadas a cada passo descrito anteriormente e gerados os devidos erros: falta ou excesso de campos na instrução do formato R.

Caso o modo de endereçamento for igual a 10, que representa o modo de endereçamento imediato tendo como característica conter a constante no corpo da instrução, e não do endereço de um registrador, o analisador sintático deverá se comportar de forma diferente, verificando o mesmo número de campos na instrução porém, efetuando análises semânticas diferentes. Abaixo é representado um exemplo do tipo imediato do formato R, onde são explicadas as diferentes envolvendo esse caso especial:



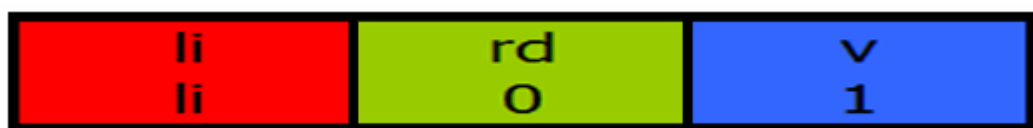
*Figura 3.3 : Formato – R : Modo de endereçamento Imediato*

- ❖ **add** : opcode
- ❖ **am** : modo de endereçamento
- ❖ **rd** : registrador de destino
- ❖ **rs** : registrador 1
- ❖ **rt** : registrador 2

Como é possível verificar no exemplo acima, os campos do registrador de origem 1 e 2 são compostos por constantes e não por registradores. Quando o bloco de análise sintática verificar que a instrução do formato R contém o modo de endereçamento imediato, uma estrutura no montador será responsável por selecionar uma função que trata desta condição especial, que irá efetuar testes nos registradores de origem 1 e 2 verificando se as constantes estão dentro da faixa de permitidos: 0 a 15 (4 bits) para cada um dos campos. Caso as constantes sejam maiores que a faixa, o montador informa uma condição de erro e finalizar sua execução. Caso houver falta ou excesso de parâmetros o montador também sinaliza tal condição e finaliza sua execução.

- ❖ Análise sintática do formato - I

O formato I é considerado quando o parâmetro informado pelo analisador léxico for um, ele então inicia a varredura das instruções a partir do último elemento lido pelo analisador léxico buscando os seguintes elementos referenciados pela instrução do formato I conforme exemplo mostrado abaixo, que servirá para exemplificar o processo:



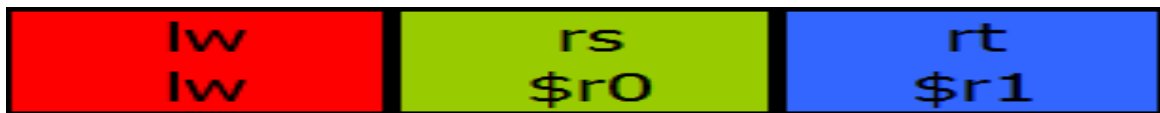
*Figura 3.4 : Formato – I*

- ❖ **li** : opcode;
- ❖ **rd** : registrador de destino;
- ❖ **v** : constante numérica;

Identificado o primeiro campo, opcode, o próximo campo a ser avaliado pelo analisador sintático do formato I é o registrador de destino. Este campo é lido a partir do opcode até o separador ','. Depois de lido é verificado semanticamente, onde se o registrador de destino for diferente de 0 ou de 1 o montador ira informar que o registrador de destino informado esta fora da especificação do AIDA-16. O próximo campo a ser verificado é a constante numérica v, que será lida a partir do separador ',' do registrador de destino até o final da instrução. Esta constante é verificada se esta dentro da faixa de valores permitida para o campo, 8 bits. Caso não estiver o montador ira informar que a constante esta fora da faixa de representação como especificado no AIDA-16. Também é possível que exista falta ou excesso de campos neste tipo de instrução, exceções também avaliadas em cada um dos passos descritos anteriormente, e caso sejam encontradas serão gerados os devidos erros.

- ❖ Analise sintática do formato - load/store

O formato load/store é considerado quando o parâmetro informado pelo analisador léxico for dois, ele então inicia a varredura das instruções a partir do último elemento lido pelo analisador léxico buscando os seguintes elementos referenciados pela instrução do formato I conforme exemplo mostrado abaixo, que servirá para exemplificar o processo:



*Figura 3.5 : Formato – Load/Store*

- ❖ **lw** : opcode;
- ❖ **rs** : registrador 1;
- ❖ **rt** : registrador 2.

O segundo campo a ser avaliado pelo analisador sintático será o registrador de origem, ele é lido a contar do primeiro elemento depois do opcode até o separador ','. Depois de lido ele é verificado quanto à semântica para verificar se o registrador de origem contém os prefixos '\$r' ou '\$t', onde '\$r' é seguido dos valores 0 a 13 e '\$t' seguido dos valores 0 ou 1 indicando os registradores 14 e 15. O próximo passo é buscar o registrador de destino, que é feito a partir da posição atual de leitura até o final da linha. O registrador de destino é verificado exatamente na mesma forma como o registrador de origem, onde se existir algum erro, será informado pelo montador. Também é possível que exista falta ou excesso de campos nas instruções, onde estas exceções também são verificadas em cada um dos passos descritos anteriormente, e caso encontrados serão gerados os devidos erros informando que na linha foi detectado falta ou excesso de campos na instrução do formato-load/store.

❖ Analise sintática do formato - Branch

O formato branch é considerado quando o parâmetro informado pelo analisador léxico for três, ele então inicia a varredura das instruções a partir do último elemento lido pelo analisador léxico buscando os seguintes elementos referenciados pela instrução do formato branch conforme exemplo mostrado abaixo, que servirá para exemplificar o processo:



*Figura 3.6 : Formato – Branch*

❖ **jmp** : opcode;

❖ **v** : constante contendo o deslocamento em relação ao PC.

Além do opcode, o único campo a ser considerado pelo analisador sintático será a constante v, ela é lida a contar do primeiro elemento depois do opcode. Depois de lido ele é verificado semanticamente para saber se a constante esta dentro da faixa de valores representados pelos 11 bits que constituem o campo da constante. Caso o valor não seja na faixa representável pelos 11 bits o montador ira informar que a constante esta fora da faixa de representação como especificado no AIDA-16. Como se pode perceber, também é

possível que exista falta ou excesso de campos nas instruções, onde estas exceções também são verificadas e caso sejam encontradas será gerado o devido erro informando que na linha foi detectada falta ou excesso de campos na instrução do formato branch.

Finalizados os procedimentos acima e nenhum erro venha a ser identificado o montador está apto a seguir a conversão do bloco de código assembly para seu formato em código de máquina, uma vez que os analisadores léxicos, sintáticos e semânticos verificaram o bloco de código quanto as especificação da ISA do processador AIDA-16. Porém como todo código assembly é formado de dados além das instruções, é necessário efetuar as mesmas análises feitas sobre as instruções no bloco dos dados do código assembly, para só então poder certificar que o código está livre de erros e pode realmente ser convertido em código de máquina. Na seção abaixo é explorado como foi implementado o bloco que realiza as análises na parte dos dados do código assembly.

#### 3.1.5 Implementação do bloco de análise dos dados

Após verificar o bloco de instruções do código assembly, o montador passa a realizar a análise léxico do bloco de dados, implementada a partir da leitura de cada linha da área de dados (.data). Caso o campo lido for igual a uma das duas diretivas de montagem de dados validas (.word e .byte), a análise sintática da linha segue normalmente, caso contrario será informado que o tipo de dado declarado na respectiva linha que foi avaliada não é valido conforme a especificação da ISA do processador AIDA-16.

O próximo passo será efetuar a análise sintática, que é implementada efetuando-se a verificação da constante presente após a diretiva de dados. A leitura é feita a partir do ultimo elemento lido na linha pelo analisador léxico, que irá ler a constante até encontrar o fim de linha. Neste momento o analisador semântico irá verificar se a constante lida está dentro da faixa de valores representável pelo número de bits previsto pela diretiva, sendo 8 para .byte e 16 para .word, previamente conhecida. O fato de também poder existir excesso ou falta de campos no bloco de dados também é algo que o montador avalia e, casos encontrados são gerados os devidos erros informando que na linha foi detectada falta ou excesso de campos.

### 3.1.6 Implementação do bloco de código intermediário e geração de código assembly das instruções

Enquanto o montador realizava a análise das instruções e dos dados, a cada campo das instruções e dos dados lidos eram sendo armazenados em estruturas com a finalidade de armazenar os dados que posteriormente seriam convertidos para dados binários. Abaixo é mostrada a estrutura que armazena os campos de uma instrução válida:

```
struct instrucao
{
    unsigned short int formato;
    unsigned short int parametro;
    unsigned short int opcode;
    unsigned short int modo_end;
    unsigned short int reg_dest;
    unsigned short int reg0;
    unsigned short int reg1;
    unsigned short int cont;
    unsigned short int desl;
}i[TAM];
```

*Figura 3.7: Estrutura responsável por guardar os campos da instrução*

A fim de detalhar qual a função de cada variável da estrutura, abaixo é feita uma breve síntese da função de cada uma das variáveis:

**formato** : Armazena o formato da instrução

**parâmetro** : Numero de parâmetros da instrução

**opcode** : Código da operação da instrução

**mod\_end** : Modo de endereçamento da instrução

**reg\_dest** : Registrador de destino da instrução

**reg0** : Registrador 1 da instrução

**reg1** : Registrador 2 da instrução

**const** : Constante de 8 bits do formato-i

**desl** : Constante de 11 bits do formato-branch

Como o leitor já deve ter visualizado, a estrutura visa através de o campo formato contemplar todos os formatos de instruções em uma única estrutura, com a finalidade de facilidade de modelagem e programação. Em primeiro plano o autor através de leituras

superficiais sobre técnicas VLIW avançadas verificou que ter um código objeto poderia futuramente trazer vantagens para contemplar tais conceitos, que mais a frente nesta primeira oportunidade de TCC sobre VLIW foram pouco explorados. Independente isto, a função responsável por gerar o código binário de saída acabou sendo muito simplificada, pois foi utilizado neste momento, técnicas que manipulam diretamente os bits de um tipo de dado primitivo da linguagem C onde através de deslocamento de bits, de uma variável unsigned short int, foi possível gerar através dos campos da estrutura variáveis unsigned short int contemplando as instruções contidas no código intermediário e escrevendo o bloco de instruções em um arquivo de extensão .bin contendo o mesmo nome do arquivo .asm informado ao iniciar o montador. O montador então irá efetuar a escrita dos primeiros 16 bits do arquivo inserindo o número de instruções do arquivo binário de saída, e outros 16bits contendo o número de dados de memória, onde posteriormente a isso, todas as instruções serão escritas no arquivo, e após executado este bloco, a memória de dados será escrita como explicado na seção abaixo.

### 3.1.7 Implementação do bloco de código intermediário e geração de código assembly dos dados

Dá mesma forma que o montador efetuou o processo de análise para as instruções e armazenou seus resultados em estruturas, os dados também sofreram o mesmo processo, que é feito com base na estrutura listada abaixo:

```
struct celula
{
    unsigned short int tipo;
    unsigned short int valor_word;
    unsigned char valor_byte;
}m[TAM];
```

*Figura 3.8: Estrutura responsável por guardar os campos dos dados*

A fim de detalhar qual a função de cada tópico, abaixo é feita uma breve síntese da função de cada uma das variáveis da estrutura:

**tipo** : Armazena o tipo de dado, sendo 0 para byte e 1 para word

**valor\_word** : Variável de 16bits de tamanho que armazena a word

**valor\_byte** : variável de 8bits de tamanho que armazena o byte

Esta estrutura tem a mesma finalidade descrita para o bloco de instruções, onde na parte dos dados, as diretivas de montagem não são escritas no arquivo binário por não fazerem parte dos dados, servindo unicamente como marcadores que informam ao montador se o dado deverá ser armazenado ou em uma célula ou em duas células. Após escrever todos os elementos contidos na estrutura do bloco dos dados, o montador fecha o arquivo de saída e finaliza a função responsável por escrever os dados no arquivo binário de saída, dando por encerrada a montagem do arquivo e finalizando o montador.

### **3.2 Especificação e Implementação do ISS**

Para desenvolver o ISS, o autor deste trabalho necessitou realizar primeiramente um levantamento superficial sobre alguns dos simuladores aplicados na atualidade, podendo só a partir dali, enxergar como seria feita a então implementação do ISS do processador AIDA-16. Como o ISS foi proposto ser feito após a implementação do montador, e um estudo do processador AIDA-16 já tinha sido realizado, dando plenas capacidades do autor não precisar rever conceitos importantes por já estar bem adaptado com o conjunto de instruções do AIDA-16, este precisava unicamente especificar diretivas básicas que o ISS seria fundamentado, para só a partir daí implementar as estruturas do ISS. Para isso foi especificado alguns tópicos importantes, discutidos na próxima seção deste Capítulo.

#### **3.2.1 Especificação do ISS**

Como discutido superficialmente no Capítulo 2 sobre os tipos de ISS existentes foi optado implementar o ISS em uma linguagem de alto nível, que por razões da linguagem, não prove mecanismos que facilitem a modelagem de estruturas capazes de simular comportamento a partir de sinais ou flags sincronizados por um relógio global de tempo, obrigando o autor deste trabalho em optar por especificar a organização arquitetural do ISS como monociclo.

Como comentado acima, o ISS deu preferência por implementar um modelo de máquina monociclo, onde cada instrução será simulada como um todo, onde mesmo o montador operando em apenas um ciclo, a modelagem do projeto seguiu a idéia de que toda instrução sofre uma busca na memória de instruções, é decodificada para se conhecer qual operação a unidade lógica deverá operar, e deve-se conhecer quais são seus parâmetros e de onde seus dados deverão vir para executar no processador, que irá



executar a partir da operação conhecida no momento de decodificação, que após executar os dados deverá armazená-los em seu respectivo destino como será constado nos campos da instrução.

O modelo monociclo deverá executar as seguintes fases, descritas abaixo :

- ❖ **Buscar instrução :** Este bloco do modelo monociclo deverá cuidar da busca da operação contida na memória de instruções, apontada pelo contador de programa, que deve ser responsável por buscar a célula apontada pelo PC + a célula posterior à apontada pelo PC, pois como a memória de instruções do ISS será endereçada a byte, e cada instrução do processador AIDA-16 tem 16bits de tamanho, o contador de programa deverá buscar de duas em duas células para buscar instrução por instrução, e na hora de atualizar deverá fazer o PC mais duas unidades a cada instrução a ser buscada.
- ❖ **Decodificar e buscar os operados:** Este bloco tem a responsabilidade de ler a instrução buscada pela função anterior, e realizar a decodificação da instrução, conhecendo qual é o código da instrução a ser executada, como seus respectivos operadores, e dependendo o tipo da instrução, seu modo de endereçamento e seu registrador destino respectivo.
- ❖ **Executar e armazenar os operados:** O bloco de execução tem como responsabilidade efetuar a execução da instrução decodificada pelo bloco de decodificação, que se baseia nos campos decodificados anteriormente, e ira buscar no local especificado no corpo da instrução os dados a serem executados. Após executar, novamente os campos da instrução serão verificados para saber onde o resultado deverá ser armazenado.

Como é feito no projeto de um processador em nível de hardware, é necessário especificar quais serão as estruturas responsáveis por implementar as unidades básicas como citadas anteriormente ao explicar o fluxo que o caminho dos dados terá no ISS. Abaixo são listadas as estruturas principais que deverão ser contempladas em tempo de implementação para dar suporte o que foi citado acima:

- ❖ Duas memórias endereçadas a byte sendo uma para dados e outra para instruções, seguindo o modelo da arquitetura de memória Harvard
- ❖ Uma unidade lógica capaz de processar 30 operações

- ❖ Um banco de registradores com suporte a 16 registradores de propósito geral
- ❖ Um contador de programa (PC) de 16bits responsável por ser que armazena o ultima instrução buscada na memória de instruções

Após conhecido qual será o modelo de organização do ISS e definido quais serão suas estruturas e seus detalhes principais, como também em poder contar com o arquivo binário provido do montador, o ISS será implementado a fim de validar tanto o arquivo gerado pelo montador como também o ISS que é um dos objetivos do trabalho.

### 3.2.2 Lendo o arquivo binário gerado pelo montador e iniciando o montador

O primeiro passo de implementação do ISS foi efetuar a leitura do arquivo binário passado por parâmetro pelo usuário para iniciar o ISS. O ISS então verifica se o arquivo contem a extensão .bin, onde se conter será um arquivo valido, senão fará com que o ISS não execute e somente informe uma mensagem de erro. Caso o arquivo informado esteja correto, o ISS ira ler os primeiros 32bits do arquivo, onde 16bits deles serão armazenados em uma variável unsigned short int responsável por conhecer qual tamanho que a memória de instruções deverá ter, e os outros 16bits serão armazenados em outra variável unsigned short int responsável por conhecer qual o tamanho que a memória de dados deverá ter. A partir disso, é feito a carga da memória de instruções através de tantas leituras de 16bits ao arquivo binário de entrada, que é feito a partir da informação contida na variável que guarda o tamanho da memória de instruções, e outra leitura, porem agora de 8bits, porem armazenando os resultados na memória de dados, finalizando a carga até que o numero de leituras feitas seja igual ao tamanho da variável que representa o tamanho da memória de dados. Feito isso o ISS deve iniciar com zero o banco de registradores como também o contador de programa, que ira chamar a função de interface do ISS para que a partir dos comandos feitos pelo usuário, seja possível buscar e efetuar a simulação do código fonte armazenado na memória de instrução e dos dados na memória de dados.

### 3.2.3 Especificação da Interface de interação com o usuário

A fim de prover interação, pois como ferramenta de ensino, um dos tópicos chave do trabalho é desenvolver algo que alem de funcional deve ser de fácil utilização. Para isso uma interface de linha de comando foi desenvolvida, onde através de opções o usuário tenha as seguintes funcionalidades disponíveis, listadas cada uma separadamente abaixo:

### 1) (E) Executar numero X de fases:

Nesta opção, o usuário deve entrar com o numero de quantas fases o simulador deverá executar sem o usuário precisar entrar com comandos, ficando com a única responsabilidade de visualizar a instrução sendo executada que irá executar modificações o sobre o banco de registradores ou sobre a memória de dados. Porém como visualizado em alguns ISS, o autor resolveu permitir que o usuário informasse qual o tempo desejado em segundos para cada fase realizada pelo ISS, podendo variar de acordo com a necessidade de visualizar algo mais detalhadamente ou simplesmente verificar se o código que se esta simulando esta corretamente funcional. Caso após executar o numero de fases informado o programa assembly ainda não for concluído, o usuário tem a opção de pressionar enter para executar novamente, ou qualquer outra tecla para voltar ao menu principal

### 2) (C) Executar fase a fase

Nesta opção o usuário irá acompanhar fase a fase que só será executada quando o usuário pressionar a tecla enter, facilitando por exemplo para que o professor em uma aula didática consiga demonstrar qual a funcionalidade da instrução, que ao pressionar o enter irá ser executada. Nesta opção, caso o usuário pressionar qualquer outra tecla, ela irá voltar ao menu principal do simulador.

### 3) (M) Mostrar end word memória dado

Nesta opção o usuário tem a possibilidade de visualizar um intervalo que deve ser informado, pelo endereço de origem e pelo endereço de destino de uma faixa de endereços da memória representado em palavras (words), que são nada mais que duas células de memória ordenadas em formato little-endian como já foi explicado porque e como funciona neste Capítulo.

### 4) (I) Mostrar cont end memória inst

Nesta opção é possível visualizar o conteúdo de uma instrução através do seu respectivo endereço que deve ser informado ao selecionar esta opção. Sua função é de dar liberdade para se visualizar instruções que estão bem distantes da janela que outras opções do ISS dão de visualizar a memória de instrução, podendo servir para visualizar qual a

instrução esta presente em um endereço que poderá ser o endereço resultante de um desvio que poderia estar prestes a desviar o PC.

#### 5) (Y) Mostrar inter mem instrução

Nesta opção é possível visualizar um determinado intervalo da memória de instruções a partir de um endereço inicial e de um endereço final que devem ser informados pelo usuário. Sua função é de dar liberdade de se visualizar intervalos de instruções distantes da janela que outras opções do ISS oferecem a memória de instrução, servindo para se conhecer instruções posteriores que um desvio condicional poderia fazer o ISS buscar após efetuar um desvio.

#### 6) (D) Mostrar inter mem dados

Nesta opção é possível visualizar um determinado intervalo da memória de dados que deve ser informado, pelo endereço de origem e pelo endereço de destino, onde sua função é de dar liberdade para se visualizar células de memória que estão bem distantes da janela que outras opções do ISS dão de visualizar a memória de dados.

#### 7) (B) Mostrar conteúdo BR MI MD

Esta opção tem a finalidade de mostrar o banco de registradores, a memória de instruções a partir do endereço apontado pelo contador de programa e os 15 primeiros endereços da memória de dados. É uma facilidade extra para não ser necessário chamar as opções D e E que executam uma fase a cada vez que são chamadas, servindo unicamente para visualizar o atual conteúdo das memórias e do banco de registrador do ISS.

#### 8) (S) Sair do simulador

Esta opção tem como única finalidade finalizar a execução do ISS

### 3.2.4 Implementação do funcionamento da interface do ISS

Depois de especificada a interface do ISS e de já ter implementado o bloco de leitura e inicialização do ISS, o autor deu início a implementação das funções responsáveis

por implementar as opções da interface do ISS. Abaixo é explicado como foi implementado cada uma das funcionalidades da interface:

#### 1) (C) Executar numero X de fases

Após o usuário informar o numero de fases e o tempo, é criada uma estrutura de repetição que ira executar até que o numero de fases informado seja igual ao contador da estrutura. Internamente a estrutura existe um bloco que mostra o conteúdo do banco de registrador, da memória de instruções a iniciar do PC, indo até 15 instruções à frente, a fim de montar uma tabela que tenha o mesmo numero de linhas que o banco de registradores, como também a memória de dados, porem da célula 0 até a célula 15. Nesta estrutura é chamado as funções de busca de instrução que efetua a busca das duas células de memória que contem a instrução apontada pelo PC e depois é chamada a função decodifica instrução, que ira efetuar a decodificação da instrução através de operações de deslocamento de bits e mascaras de bits sobre a variável unsigned short int que contem a instrução, e a partir do código da operação será possível conhecer quais são os campos da instrução e serão carregados para a seguinte estrutura mostrada abaixo:

```
struct instrucao
{
    unsigned short int opcode;
    unsigned short int formato;
    short int desl;
    unsigned short int reg0;
    unsigned short int reg1;
    short int cost;
    unsigned short int regdest;
    unsigned short int modo;
}
```

*Figura 3.9: Estrutura responsável por guardar os campos das instruções*

Os campos da instrução são explicados abaixo:

**opcode** : Responsável por armazenar o código da operação

**formato** : Responsável por armazenar o tipo da instrução

**desl** : Responsável por armazenar o campo de deslocamento caso o formato fizer necessário

**reg0** : Responsável por armazenar o registrador 1 da instrução caso o formato fizer necessário

**reg1** : Responsável por armazenar o registrador 2 da instrução caso o formato fizer necessário

**cost** : Responsável por armazenar a constante caso o formato fizer necessário

**regdest** : Responsável por armazenar o registrador de destino caso o formato fizer necessário

**modo** : Responsável por armazenar o modo de endereçamento caso o formato fizer necessário

Como se pode perceber, a estrutura segue o mesmo caminho que as outras estruturas utilizadas pelo montador que contempla todos os campos das instruções em somente uma estrutura de dados. Esta metodologia de implementação como já discutido anteriormente, visa simplicidade de código, reduzindo o aumento da complexidade para implementar a unidade de execução.

Após decodificar a instrução, a função que implementa a unidade de execução é chamada, onde é feito à leitura do código da operação na estrutura e com base no seu respectivo formato, a unidade lógica irá verificar o que deve ser feito, e onde os respectivos dados se encontram, efetuando a busca e a operação, escrevendo posteriormente os resultados no destino já conhecido através dos campos da instrução.

Seguindo já comentado, a estrutura de repetição desta opção da interface irá fazer estes passos no tempo informado pelo usuário até que seu contador seja igual ao número de fases especificado pelo usuário. Pode acontecer de ao estar efetuando os passos à função que implementa a busca de instruções efetuar a busca uma instrução de halt, que informa ao ISS que a execução deve ser abordada neste instante. O ISS após ler esta instrução informa com uma mensagem que o código foi executado e este está sendo encerrado, onde ao passar alguns segundos finaliza o ISS. Caso a instrução halt não seja encontrada no intervalo e o ISS execute todas as fases uma estrutura condicional solicita se o usuário deseja executar novamente o mesmo número de fases naquela velocidade pressionando enter ou qualquer outra tecla para voltar ao menu principal.

## 2) (C) Executar fase a fase

Nesta opção, o mesmo processo citado anteriormente é feito, com a diferença de que não existe uma estrutura de repetição que executa até que um contador seja igual ao número de fases desejado, mas somente uma estrutura condicional que após efetuar todo o

processo anterior é feita uma pergunta se o usuário deseja executar mais uma fase pressionando enter ou qualquer outra tecla para voltar ao menu inicial. A principal função desta opção é dar a liberdade que o usuário demore quanto tempo achar necessário para visualizar detalhadamente o que a instrução efetuou, podendo ser utilizada tanto para ensinar passo a passo um grupo de instruções ou até mesmo para validar instruções com grau maior de complexidade, que devem se dedicar mais tempo para serem entendidas em tempo de simulação.

### 3) (M) Mostrar ende word memória dado

Esta opção foi implementada através de uma estrutura de repetição que lista a memória de dados de duas em duas células, representando neste intervalo palavras (words) de memória, que são buscadas desde o endereço inicial informado, até o endereço final, com a função de listar agora palavras e não células de memória.

### 4) (Y) Mostrar inter mem instrução

Esta opção foi implementada através de uma estrutura de repetição que lista a memória de instruções a partir do endereço inicial informado pelo usuário até o endereço final, com a função de que fosse possível conhecer endereços de instruções distantes do PC para saber quais seriam as instruções buscadas por um desvio condicional, que se caso acontecesse poderia cair neste intervalo de instrução.

### 5) (D) Mostrar inter mem dados

Esta opção foi implementada através de uma estrutura de repetição que lista a memória de dados a partir do endereço inicial informado pelo usuário até o endereço final, com a função de que seja possível conhecer endereços de dados distantes que dificilmente seriam mostrados pelas visualizações das opções C D e B.

### 6) (B) Mostrar conteúdo BR MI MD

Esta opção foi implementada através de uma estrutura de repetição que mostra o conteúdo do banco de registradores, da memória de instrução a contar do endereço inicial apontado por PC e da primeira posição da memória de dados. Esta opção tem a função de

mostrar o conteúdo do banco de registradores, da memória de instruções e da memória de dados em um respectivo instante de tempo, afim de que não fosse necessário chamar as opções C ou E listadas acima que a cada chamada executam o código pelo menos uma vez.

#### 7) (S) Sair do simulador

Esta opção foi implementada com uma função de sair (exit) da linguagem de programação onde na linha anterior é feita a limpeza da tela antes de sair. Sua função é de finalizar a execução do ISS.

### 3.3 Especificação e Implementação do VLIW no montador

A fim de dar suporte ao VLIW sobre o montador, um estudo teve que ser feito, a fim de descobrir conforme já foi citado no Capítulo 2, quais técnicas seriam exploradas e qual seria o encapsulamento de instruções a ser definido para a organização arquitetural monociclo VLIW proposta no trabalho. Todas estas informações foram especificadas e depois foram implementadas, abaixo é descrita a especificação feita para suporte VLIW no montador.

#### 3.3.1 Especificação VLIW sobre o montador

Para especificar o montador VLIW foi definido o seguinte:

- ❖ O encapsulamento de instrução será dado em duas instruções do processador AIDA-16 sem flags adicionais
- ❖ O encapsulamento será capaz de comportar qualquer combinação de formato de instrução, onde o montador VLIW ira tratar qualquer tipo de dependência envolvendo todas as combinações de formatos possíveis.
- ❖ Não serão efetuados tratamentos para os conflitos, somente será indicado onde existe tal dependência, ficando a cargo do usuário em colocar uma instrução de nop entre as instruções que existirem dependência
- ❖ O suporte VLIW a ser dado em tempo de montagem será do tratamento das seguintes dependências:
  - ✓ RAW (Dependência de dados) : Esta dependência ira acontecer quando na primeira linha houver uma escrita em um registrador e na linha seguinte houver uma leitura



neste mesmo registrador. Ao encontrar esse tipo de erro o montador deverá parar a montagem e informar que existe uma dependência entre as duas instruções.

- ✓ WAR (Anti-dependencia): Esta dependência ira acontecer quando na primeira linha houver uma leitura em um registrador e na linha seguinte houver uma escrita neste mesmo registrador. Ao encontrar esse tipo de erro o montador deverá parar a montagem e informar que existe uma dependência entre as duas instruções
- ✓ WAW (Dependência de saída) : Esta dependência ira acontecer quando na primeira linha houver uma escrita em um registrador e na linha seguinte houver uma escrita neste mesmo registrador. Ao encontrar esse tipo de erro o montador deverá parar a montagem e informar que existe uma dependência entre as duas instruções.

A partir desta especificação é possível dar inicio ao desenvolvimento da função no montador que ficará responsável por detectar e informar se em um intervalo de duas instruções a partir da primeira linha da memória de instruções existe algum tipo de dependência, que deverá informar ao usuário e parar a montagem até que o erro seja corrigido.

### 3.3.2 Implementação VLIW sobre o montador

Para dar suporte VLIW ao montador, o autor fez uso do montador antigo, que foi neste instante já estava especificado, implementado e validado onde foi verificado que deveria ser criada uma função, que pegando a memória de instrução original, porem em um estagio anterior à geração do código binário de saída, que é um momento em que o código já é considerado valido e esta pronto para ser gerado seu código binário de saída, onde para um modelo monociclo tradicional estaria correto, porem no caso do VLIW poderiam existir dependências de dados, ou de controle, como de estrutura, que como citadas anteriormente no Capítulo 2 devem ser tratadas, e se faz necessário à inclusão de uma função para tratar estas dependências. A função implementada tem como objetivo tratar as dependências, que é feita a partir da leitura sempre de duas em duas instruções, onde é verificado algumas condições que são exploradas a partir da combinação dos formatos de instruções.

### 3.3.3 Formato das instruções

O formato é a primeira referencia que informa a função quais serão as possíveis combinações que devem ser verificadas e quais serão as dependências que podem ser encontradas. O segundo passo é conhecer os campos de cada um dos formatos, onde serão

neles que as dependências poderão ser detectadas. Abaixo é demonstrado quais são as possíveis combinações de formatos e quais os campos relacionados que podem causar dependências:

#### 1. Formato R – Formato R

Nesta combinação é necessário primeiramente verificar quais são os modos de endereçamento das instruções do formato-r, para só então efetuar a leitura dos campos dos registradores e verificar se existe algum tipo de dependência. Abaixo é demonstrado quais são as combinações dos modos de endereçamento possíveis e o que deve ser verificado:

##### ❖ Formato R (Direto) – Formato R (Direto)

<pre>1 add 00,0,\$r0,\$r1 2 add 00,0,\$r0,\$r1</pre>
--

*Figura 3.10: Instruções Formato R (Direto) – Formato R (Direto)*

Neste caso podem acontecer as dependências do tipo:

- ✓ RAW: SIM (escreve em \$r0 na linha 1 como verificado por 0 do registrador de destino e lê \$r0 na linha 2)
- ✓ WAR: SIM (lê em \$r0 na linha 1 e escreve em \$r0 como verificado por 0 do registrador de destino da linha 2)
- ✓ WAW : SIM (escreve em \$r0 tanto na linha 1 como na linha 2 como verificado pelo registrador destino 0 na linha 1 e na linha 2)

Como pode ser verificado, nessa combinação de formatos de instrução utilizando o modo de endereçamento direto na linha 1 e na linha 2 faz com que seja possível existir as 3 dependências entre as instruções.

##### ❖ Formato R (Direto) – Formato R (Indireto)

<pre>1 add 00,0,\$r0,\$r1 ou 1 add 01,0,\$r0,\$r1 2 add 01,0,\$r0,\$r1 ou 2 add 01,0,\$r0,\$r1</pre>
--

*Figura 3.11: Instruções Formato R (Direto/Indireto) e Formato R (Indireto/Indireto)*

Neste caso podem acontecer as dependências do tipo:

- ✓ RAW: IMPOSSIVEL SABER (Não é possível verificar pois se desconhece em tempo de montagem o endereço apontado por \$r0 e por \$r1 da linha 2)
- ✓ WAR: IMPOSSIVEL SABER (Não é possível verificar pois se desconhece em tempo de montagem o endereço apontado por \$r0 e por \$r1 da linha 2)
- ✓ WAW: IMPOSSIVEL SABER (Não é possível verificar pois se desconhece em tempo de montagem o endereço apontado por \$r0 e por \$r1 da linha 2)

Como pode ser verificado, caso existir estes tipos de situação entre registradores iguais o montador ira informar ao usuário que não é possível tratar as dependências das instruções obrigando o usuário a também tratar a execução paralela das duas instruções com um nop como já explicado acima.

❖ Formato R (Direto) – Formato R (Imediato)

<pre>add 00,0,\$t0,\$t0 add 10,0,1,2</pre>
--

*Figura 3.12: Instruções Formato R (Direto) - Formato R (Imediato)*

Neste caso podem acontecer as dependências do tipo :

- ✓ RAW: NÃO (Não acontece)
- ✓ WAR: SIM (Acontece quando o registrador \$t0 é lido na linha 1 e escrito na linha 2)
- ✓ WAW: SIM (Acontece pois a linha 1 e a linha 2 estão escrevendo no mesmo registrador destino \$t0)

Como pode ser verificado acima, caso existir a primeira situação o encapsulamento não terá dependência, porém nos outros casos o montador irá informar ao usuário que não é possível tratar as dependências WAR e WAW das instruções obrigando o usuário a também tratar a execução paralela das duas instruções com um nop como já explicado acima.

❖ Formato R (Imediato) – Formato R (Indireto)

1	add	10,0,1,1
2	add	01,0,\$r0,\$r1

*Figura 3.13: Instruções Formato R (Imediato) – Formato R (Indireto)*

Neste caso podem acontecer as dependências do tipo:

- ✓ RAW: IMPOSSIVEL SABER (Não é possível verificar pois se desconhece em tempo de montagem o endereço apontado por \$r0 e por \$r1 da linha 2)
- ✓ WAR: NÃO (Não acontece)
- ✓ WAW: IMPOSSIVEL SABER (Não é possível verificar pois se desconhece em tempo de montagem o endereço apontado por \$r0 e por \$r1 da linha 2)

Como pode ser verificado acima, caso existir a segunda situação o encapsulamento não terá dependência, porém nos outros casos o montador irá informar ao usuário que não é possível tratar as dependências RAW e WAW das instruções obrigando o usuário a também tratar a execução paralela das duas instruções com um nop como já explicado acima.

❖ Formato R (Imediato) – Formato R (Direto)

1	add	10,0,1,1
2	add	00,0,\$t0,\$r1

*Figura 3.14: Instruções Formato R (Imediato) – Formato R (Direto)*

Neste caso podem acontecer as dependências do tipo :

- ✓ RAW: SIM (Acontece pois a linha 1 efetua uma escrita no registrador de destino 0 representado por \$t0 que é lido na instrução da linha 2)
- ✓ WAR: NÃO (Não acontece)
- ✓ WAW: SIM (Acontece pois a linha 1 efetua uma escrita no registrador de destino 0 representado por \$t0 que é lido na instrução da linha 2)

Como pode ser verificado acima, caso existir a segunda situação o encapsulamento não terá dependência, porem-nos outros casos o montador ira informar ao usuário que não é possível tratar as dependências RAW e WAW das instruções obrigando o usuário a também tratar a execução paralela das duas instruções com um nop como já explicado acima.

❖ Formato R (Indireto) – Formato R (Imediato)

<pre>1 add 01,0,\$r0,\$r1 2 add 10,0,1,1</pre>
--

*Figura 3.15: Instruções Formato R (Indireto) – Formato R (Imediato)*

Neste caso podem acontecer as dependências do tipo :

- ✓ RAW : IMPOSSIVEL SABER (Não é possível verificar pois se desconhece em tempo de montagem o endereço apontado por \$r0 e por \$r1 da linha 1)
- ✓ WAR : NÃO (Não acontece)
- ✓ WAW : IMPOSSIVEL SABER (Não é possível verificar pois se desconhece em tempo de montagem o endereço apontado por \$r0 e por \$r1 da linha 1)

Como pode ser verificado acima, caso existir a segunda situação o encapsulamento não terá dependência, porem-nos outros casos o montador ira informar ao usuário que não é possível tratar as dependências RAW e WAW das instruções obrigando o usuário a também tratar a execução paralela das duas instruções com um nop como já explicado acima.

## 2. Formato R – Formato I

Nesta combinação é necessário verificar o modo de endereçamento da instrução do formato-r e o registrador de destino da instrução do formato-i. Abaixo é demonstrado quais são as combinações possíveis destas instruções :

❖ Formato R (Direto) – Formato I

1	add	00,0,\$t0,\$r1
2	li	0,0

*Figura 3.16: Instruções Formato R (Direto) – Formato I*

Neste caso podem acontecer as dependências do tipo :

- ✓ RAW: NÃO (Não acontece)
- ✓ WAR: SIM (Acontece pois a linha 1 lê o registrador \$t0 e a linha 2 escreve em \$t0)
- ✓ WAW: SIM (Acontece pois tanto a linha 1 como a linha 2 escrevem em \$t0)

Como pode ser verificado dependências do tipo RAW não acontecem, porem nos tipo WAR e WAW podem acontecer quando na linha 1 houver leitura ou escrita sobre \$t0 ou \$t1

❖ Formato R (Indireto) – Formato I

1	add	01,0,\$r0,\$r1
2	li	0,0

*Figura 3.17: Instruções Formato R (Indireto) – Formato I*

Neste caso podem acontecer as dependências do tipo :

- ✓ RAW: NÃO (Não acontece)
- ✓ WAR: IMPOSSIVEL SABER (Não é possível verificar pois se desconhece em tempo de montagem o endereço apontado por \$r0 e por \$r1 da linha 1)

- ✓ WAW: IMPOSSIVEL SABER (Não é possível verificar pois se desconhece em tempo de montagem o endereço apontado por \$r0 e por \$r1 da linha 1)

Como pode ser verificado, dependências do tipo RAW não acontecem, porem nos tipos WAR E WAW podem acontecer, porem não é possível saber em tempo de montagem, obrigando o usuário a tratar a execução paralela das duas instruções com um nop como já explicado acima.

❖ Formato R (Imediato) – Formato I

1	add	10,0,1,2
2	li	0,0

*Figura 3.18: Instruções Formato R (Imediato) – Formato I*

Neste caso podem acontecer as dependências do tipo :

- ✓ RAW: NÃO (Não acontece)
- ✓ WAR: NÃO (Não acontece)
- ✓ WAW: SIM (Acontece pois o registrador de destino \$t0 da linha 1 é o mesmo registrador de destino \$t0 da linha 2)

Como pode ser verificado, as dependências do tipo RAW e WAR não acontecem nesse tipo de combinação de formatos de instrução, podendo acontecer somente WAW quando os registradores de destino forem iguais.

### 3. Formato R – Formato Load/Store

❖ Formato R (Direto) – Formato Load/Store

1	add	00,0,\$r0,\$r1
2	lw	\$r0,\$r0

*Figura 3.19: Instruções Formato R (Direto) – Formato Load/Store*

Neste caso podem acontecer as dependências do tipo :

- ✓ RAW: SIM (Acontece caso houver escrita no registrador 1 da linha 1 e leitura tanto no registrador 1 ou registrador 2 da linha 2)
- ✓ WAR: SIM (Acontece caso houver leitura no registrador 1 ou no registrador 2 da linha 1 e escrita no registrador 1 da linha 2)
- ✓ WAW: SIM (Acontece caso houver escrita no registrador 1 da linha 1 e no registrador 1 da linha 2)

Como pode ser verificado, nessa combinação de formatos de instrução utilizando o modo de endereçamento direto na linha 1 faz com que seja possível existir as 3 dependências entre as instruções.

❖ Formato R (Indireto) – Formato Load/Store

<pre>1 add 01,0,\$r0,\$r1 2 lw \$r0,\$r0</pre>
--

*Figura 3.20: Instruções Formato R (Indireto) – Formato Load/Store*

Neste caso podem acontecer as dependências do tipo :

- ✓ RAW: IMPOSSIVEL SABER (Não é possível verificar pois se desconhece em tempo de montagem o endereço apontado por \$r0 e por \$r1 da linha 1)
- ✓ WAR: IMPOSSIVEL SABER (Não é possível verificar pois se desconhece em tempo de montagem o endereço apontado por \$r0 e por \$r1 da linha 1)
- ✓ WAW: IMPOSSIVEL SABER (Não é possível verificar pois se desconhece em tempo de montagem o endereço apontado por \$r0 e por \$r1 da linha 1)

Como pode ser verificado, caso existir estes tipos de situação entre registradores iguais o montador ira informar ao usuário que não é possível tratar as dependências das instruções obrigando o usuário a também tratar a execução paralela das duas instruções com um nop como já explicado acima.



## ❖ Formato R (Imediato) – Formato Load/Store

1	add	10,0,1,2
2	lw	\$t0,\$t0

*Figura 3.21: Instruções Formato R (Imediato) – Formato Load/Store*

Neste caso podem acontecer as dependências do tipo :

- ✓ RAW: SIM (Acontece caso o registrador de destino \$t0 da linha 1 for igual a um dos registrador 2 da linha 2)
- ✓ WAR: NÃO (Não acontece)
- ✓ WAW: SIM (Acontece caso o registrador destino \$t0 da linha 1 for igual ao registrador 1 da linha 2)

Como pode ser verificado, acontecem dependências do tipo RAW e WAW quando há combinação destes formatos com o modo imediato do formato-r, porem não há dependência do tipo WAR.

## 4. Formato R – Formato Branch

## ❖ Formato R (Direto/Indireto/Imediato) /Formato Branch

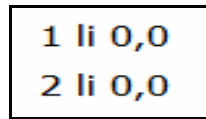
1	add	00,0,\$r0,\$r1	ou	1	add	01,0,\$r0,\$r1	ou	1	add	10,0,1,2
2	jz	2	ou	2	jz	2	ou	2	jz	2

*Figura 3.22: Instruções Formato R (Direto/Indireto/Imediato) – Formato Branch*

Neste caso podem acontecer as dependências do tipo:

- ✓ RAW/WAR/WAW: Nesse caso não acontece dependência de dados, porem o flag de zero da operação de soma só será descoberto depois que add executar, isso implica que add e jz não podem executar ao mesmo tempo e o usuário precisa colocar um nop de distancia para primeiro a soma ser efetuada.

## ❖ Formato I – Formato I



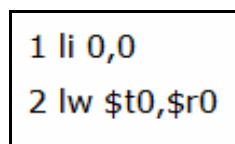
*Figura 3.23: Instruções Formato I – Formato I*

Neste caso podem acontecer as dependências do tipo :

- ✓ RAW: NÃO (Não acontece)
- ✓ WAR: NÃO (Não acontece)
- ✓ WAW: SIM (Acontece quando o registrador de destino da linha 1 for igual ao registrador de destino da linha 2)

Como pode se verificar, as dependências de dados tipo RAW e WAR não ocorrem nessa combinação de formatos de instrução, podendo ocorrer somente à dependência do tipo WAW caso os registradores de destino da linha 1 e 2 forem iguais.

❖ Formato I – Formato Load/Store



*Figura 3.24: Instruções Formato I – Formato Load/Store*

Neste caso podem acontecer as dependências do tipo:

- ✓ RAW: SIM (Acontece quando a instrução da linha 1 efetua uma escrita no registrador de destino \$t0 e uma leitura no mesmo registrador na linha 2)
- ✓ WAR: NÃO (Não acontece)
- ✓ WAW: SIM (Acontece quando a instrução da linha 1 escreve no registrador de destino \$t0 e efetua uma escrita no mesmo registrador \$t0 na linha 2)

Como pode ser verificado, as dependências do tipo RAW e WAW podem acontecer se o registrador da linha 2 for igual ao registrador de destino da linha 1 onde a dependência do tipo WAR não acontece nessa combinação de instruções.

❖ Formato I – Formato Branch

1	li	0,0
2	bnz	2

*Figura 3.25: Instruções Formato I – Formato Branch*

Neste caso podem acontecer as dependências do tipo:

- ✓ RAW/WAR/WAW: Nesta condição não ocorre nem conflito de dados nem de controle pois a instrução li opera sobre os flags de carry/zero/overflow podendo operar ao mesmo tempo com instruções do tipo branch.

❖ Formato Load/Store – Formato Load/Store

1	lw \$r0,\$r1 ou lb \$r0,\$r1
2	lw \$r1,\$r0 ou lb \$r1,\$r0

*Figura 3.26: Instruções Formato Load/Store – Formato Load/Store*

Neste caso podem acontecer as dependências do tipo :

- ✓ RAW: SIM (acontece quando a instrução da linha 1 efetua uma leitura do registrador \$r1 e na linha 2 uma escrita em \$r1)
- ✓ WAR: SIM (acontece quando a instrução da linha 1 efetua uma leitura do registrador \$r0 e na linha 2 uma escrita em \$r0)
- ✓ WAW: SIM (acontece quando a instrução da linha 1 e a instrução da linha 2 escrevem no mesmo registrador)

Neste caso podem acontecer todo tipo de dependência que precisa ser tratado caso existirem registradores iguais causando dependência.

❖ Formato Load/Store – Formato Branch

```

1 lw $r0,$r1
2 bnz 2

```

*Figura 3.27: Instruções Formato Load/Store – Formato Branch*

Neste caso podem acontecer as dependências do tipo :

- ✓ RAW/WAR/WAW : Nesta condição não acontece conflito porque instruções do tipo load/store não operam sobre os flags de carry/zero/overflow podendo operar ao mesmo tempo com instruções do tipo branch.

❖ Formato Branch – Formato Branch

```

1 bnz 2
2 bnz 2

```

*Figura 3.28: Instruções Formato Branch – Formato Branch*

Nesta condição ocorre dependência de controle onde ambas as instruções podem fazer uso do contador de programa, não sendo possível atualiza-lo ao mesmo tempo, onde o montador VLIW ira informar ao usuário que um nop deverá ser colocado entre as duas instruções.

Todas as combinações listadas acima são parte da função que verifica as dependências entre as instruções, e através delas tem autonomia para informar ao usuário se entre aquelas instruções existem ou não dependências, onde caso existir será informado em tela qual dependência e em quais instruções e será cancelada a montagem até que o usuário corrija explicitamente inserindo um nop entre as instruções com a finalidade de eliminar a dependência entre as instruções. O código só será montado corretamente no momento em que não for encontrada nenhuma dependência no código, o que caracteriza ao montador que todo código gerado por ele estará livre de dependências, garantindo que o

hardware não precisará se preocupar em tempo de execução se as instruções que estão sendo buscadas no encapsulamento de duas instruções contem algum tipo de dependência associada entre elas.

### **3.4 Especificação e Implementação do VLIW no ISS**

#### **3.4.1 Especificação do VLIW sobre o ISS**

Para especificar o ISS VLIW a partir das definições do montador VLIW foi definido o seguinte:

- ❖ A função de busca do ISS VLIW deverá ser estendida para efetuar uma busca de uma janela de 32bits que represente uma busca de duas instruções AIDA-16 de 16bits cada uma
- ❖ O bloco de decodificação deverá prover duas unidades capazes de decodificar "sequencialmente" as duas instruções buscadas da função anterior, onde ao ser passada o encapsulamento com a longa instrução para a unidade de decodificação, esta deverá ter o suporte de decodificar simultaneamente as duas instruções encapsuladas.
- ❖ A unidade lógica deverá ser replicada para prover suporte VLIW na execução "seqüencial" das duas instruções decodificadas no bloco anterior, o que ira caracterizar um comportamento de uma organização VLIW na estrutura de execução do ISS.
- ❖ Alguns detalhes envolvendo a interface do ISS deverão ser modificadas para prover suporte ao VLIW como a inclusão dos campos de visualização das duas instruções a serem buscadas fase a fase nas opções E e C.

A partir destes detalhes especificados é possível iniciar o desenvolvimento do ISS VLIW que ira fazer uso de todos os outros produtos gerados até este instante da confecção deste TCC.

### 3.4.2 Implementação VLIW sobre o ISS

Para dar suporte VLIW ao ISS, o autor fez uso do ISS antigo, onde foi verificado que deveria ser duplicada as unidades de decodificação e de execução, bastando reaproveitar o mesmo código já escrito tanto para a unidade de decodificação e para a unidade de execução criada para o ISS monociclo. No final, bastava modificar a função de busca, que ao invés de buscar somente a instrução apontada pelo PC, agora efetua a busca de duas instruções, uma a partir da apontada pelo PC e a outra mais a frente, totalizando duas instruções a cada fase de busca.

Na parte da interface, bastou informar que agora duas instruções estão sendo buscadas a cada fase nas opções C e D. Fora estes detalhes de duplicação das unidades de decodificação e execução como do ajuste na busca das instruções através da janela do PC, o ISS VLIW não exigiu nenhuma mudança significativa comparado ao ISS original, pelo fato de que a abordagem VLIW como verificado na literatura, não causa impactos substanciais no hardware, o que pode ser comparado diretamente no desenvolvimento do ISS VLIW, que é revertida para o montador, como pode ser verificado neste Capítulo pela complexidade no processo de montagem que verifica as dependências de dados existentes no encapsulamento das instruções.

## 4 VALIDAÇÃO E RESULTADOS OBTIDOS

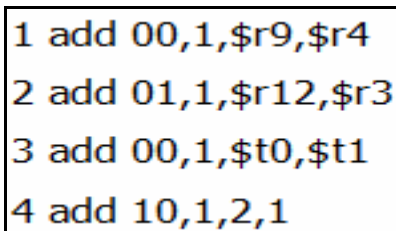
Este Capítulo tem como objetivo a validação dos produtos desenvolvidos no Capítulo anterior, através de arquivos com objetivo de testar o comportamento de instruções validas que são aquelas instruções corretas conforme definidas na especificação do processador AIDA-16 e invalidas que não corresponde a especificação do processador AIDA-16. Abaixo é feita uma avaliação de cada produto gerado durante a fase de desenvolvimento do ISS e do montador VLIW.

### 4.1 Validação do montador de acordo com a especificação do AIDA-16

Depois de concluído o desenvolvimento do montador, o autor desenvolveu um arquivo fonte chamado (montador.asm), responsável por testar o funcionamento do que foi implementado no montador. Nele foi contemplado todos os casos validos e inválidos das 4 classes de instruções, testando se o montador tinha capacidade de gerar código para as instruções corretas, e mensagens de erro para as instruções incorretas. Abaixo é visualizado por linha do arquivo.

#### 4.1.1 Testando instruções validas do Formato – R

Para testar as combinações validas mais importantes dos campos do Formato-R, foi proposto o seguinte bloco de instruções:



```
1 add 00,1,$r9,$r4
2 add 01,1,$r12,$r3
3 add 00,1,$t0,$t1
4 add 10,1,2,1
```

*Figura 4.1 : Instruções validas do formato - R*

Este mesmo bloco de instruções validas que foi aplicado na instrução add também foi replicado para todas as outras 14 instruções do Formato-R, para só assim garantir que o montador estava de acordo com a ISA do AIDA-16 para todas as instruções do Formato-R.

Como se pode visualizar no bloco de instruções acima, existem números em cada linha que são responsáveis por referenciar cada instrução neste exemplo, não fazendo parte

do corpo da instrução. Dando início ao teste de validação do bloco de instruções válidas, veremos que a função da linha 1 é de testar o modo de endereçamento direto(00) onde o registrador \$r4 irá receber o resultado da soma do registrador \$r9 com o registrador \$r4 pelo registrador de destino estar marcado como (1). A função da linha 2 é testar o modo de endereçamento indireto efetuando a soma do conteúdo do registrador apontado por \$r12 com o conteúdo do registrador apontado por \$r3 e armazenando o resultado no conteúdo do registrador apontado por \$r3 pelo registrador destino estar marcado como (1). A terceira linha faz um teste no modo de endereçamento direto, efetuando uma soma entre o registrador \$t0 e o registrador \$t1, armazenando o resultado da soma em \$t1, pois o registrador de destino está marcado como (1). A linha 4 tem como função testar o modo de endereçamento imediato, que efetua a soma entre as constantes 2 e 1 presentes no campo do primeiro e do segundo operando, onde o resultado desta operação será armazenada no registrador \$t1 pelo registrador destino estar marcado como (1).

Ao encaminhar este bloco de 4 instruções, com as demais diretivas de montagem especificadas no Capítulo 3 que validam um arquivo assembly para o processador AIDA-16 para o montador, obteve-se os seguintes resultados de montagem, como pode ser visualizado na figura abaixo :



```

spock2f@raffael: /
File Edit View Terminal Help
spock2f@raffael:/$ ./montador montador.asm
spock2f@raffael:/$

```

*Figura 4.2: Saída do montador ao efetuar o bloco de instruções do Formato-R*

Como se pode verificar na figura acima, todas as instruções de soma descritas anteriormente foram montadas com sucesso, pois o montador não gerou nenhum tipo de erro léxico, sintático ou semântico nas instruções.

#### 4.1.2 Testando instruções inválidas do Formato – R

Para testar as combinações inválidas mais importantes dos campos do Formato-R, foi proposto o seguinte bloco de instruções:



```

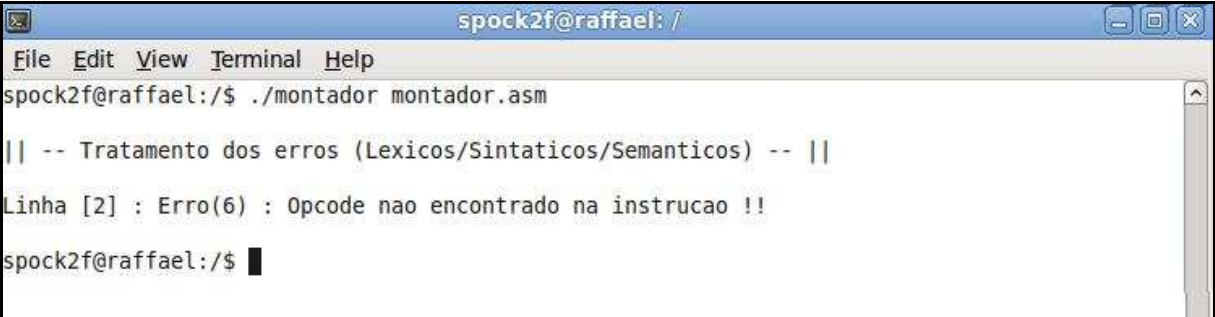
1 ad 02,1,$r9,$r4
2 add 02,1,$r9,$r4
3 add 00,3,$r9,$r4
4 add 01,1,$r3
5 add 10,1,2,16
6 add 01,1,$r12,$r16
7 add 01,1,$r12,$r16,$r17

```

*Figura 4.3 : Instruções inválidas do formato – R*

Este mesmo bloco de instruções inválidas que foi aplicado na instrução add também foi replicado para todas as outras 14 instruções do Formato-R, para só assim garantir que o montador contemple os formatos inválidos das instruções do Formato-R e gera corretamente as mensagens de erros.

Como se pode visualizar no bloco de instruções acima, existem números em cada linha que são responsáveis por referenciar cada instrução neste exemplo, não fazendo parte do corpo da instrução. Dando início ao teste de validação do bloco de instruções inválidas, veremos que a função da linha 1 é testar se o montador é capaz de reconhecer que o campo de opcode da instrução não pertence ao ISA AIDA-16. Abaixo é possível obter o resultado gerado pelo montador ao executar esta instrução :



```

spock2f@raffael: /
File Edit View Terminal Help
spock2f@raffael:/$ ./montador montador.asm

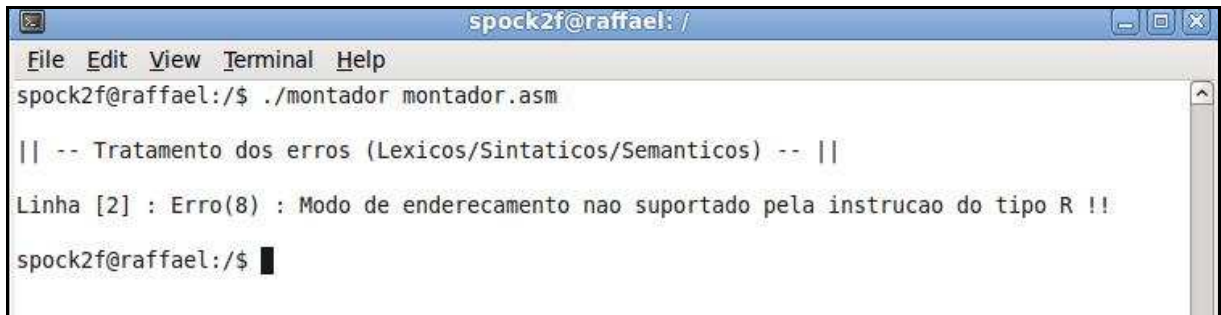
|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||
Linha [2] : Erro(6) : Opcode nao encontrado na instrucao !!
spock2f@raffael:/$ █

```

*Figura 4.4: Código da operação não encontrado*

Como se pode verificar na figura acima, o montador foi capaz de verificar que a instrução contida na linha não possuía o campo de opcode, e por isso finalizou corretamente a execução do montador informado o erro.

Na linha 2 é possível verificar que o modo de endereçamento declarado na instrução (am = 02), não faz parte da especificação do ISA AIDA-16. Abaixo é possível visualizar qual foi a saída do montador ao executar essa instrução:

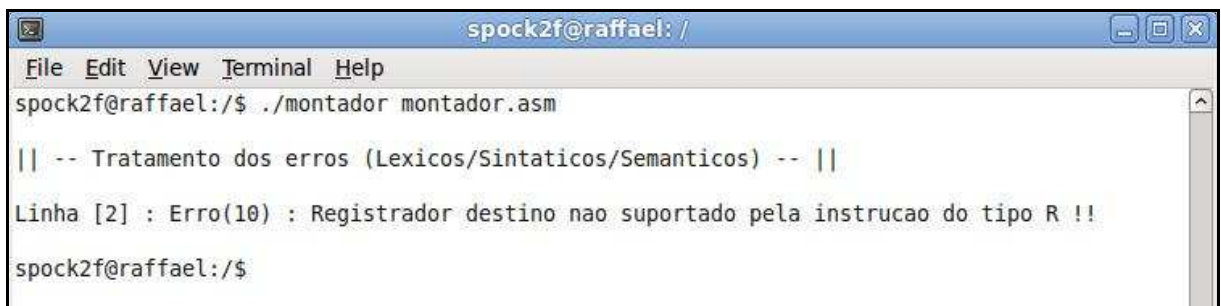


```
spock2f@raffael: /
File Edit View Terminal Help
spock2f@raffael:/$ ./montador montador.asm

|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||
Linha [2] : Erro(8) : Modo de endereçamento nao suportado pela instrucao do tipo R !!
spock2f@raffael:/$
```

*Figura 4.5: Modo de endereçamento não suportado na instrução do Formato-R*

Na linha 3 é possível verificar que o registrador destino declarado na instrução (rd = 3), não faz parte da especificação do ISA do AIDA-16. Abaixo é possível visualizar qual foi a saída do montador ao executar esta instrução.

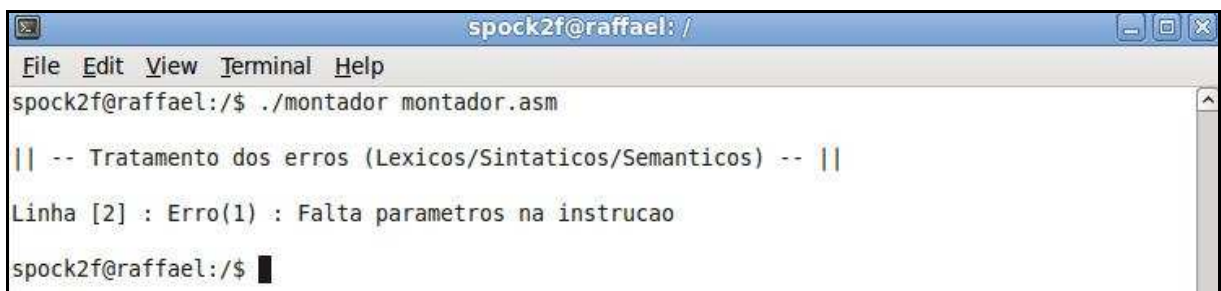


```
spock2f@raffael: /
File Edit View Terminal Help
spock2f@raffael:/$ ./montador montador.asm

|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||
Linha [2] : Erro(10) : Registrador destino nao suportado pela instrucao do tipo R !!
spock2f@raffael:/$
```

*Figura 4.6: Registrador de destino não suportado na instrução do Formato-R*

Na linha 4 é possível verificar que falta um dos operandos da instrução do formato-R, que invalida a especificação da instrução de add no modo indireto(am = 01) com somente um operando. Abaixo é possível visualizar qual foi a saída do montador ao executar esta instrução.




```
spock2f@raffael: /
File Edit View Terminal Help
spock2f@raffael:/$ ./montador montador.asm

|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||
Linha [2] : Erro(1) : Falta parametros na instrucao
spock2f@raffael:/$
```

*Figura 4.7: Falta operandos na instrução do Formato-R*

Na linha 5 é possível verificar que a instrução utiliza o modo de endereçamento imediato(am = 10) e realiza sua operação baseada nos valores das constantes presentes no campo do registrador 1 e no campo do registrador 2. Porém como é possível verificar na instrução, a capacidade máxima de representação que é de 4bits comporta constantes com valor entre (0 e 15), porém como consta na instrução da linha 5, o campo do registrador 2 contém uma constante não comportada pela instrução. Abaixo é possível visualizar qual foi a saída do montador ao executar a instrução:

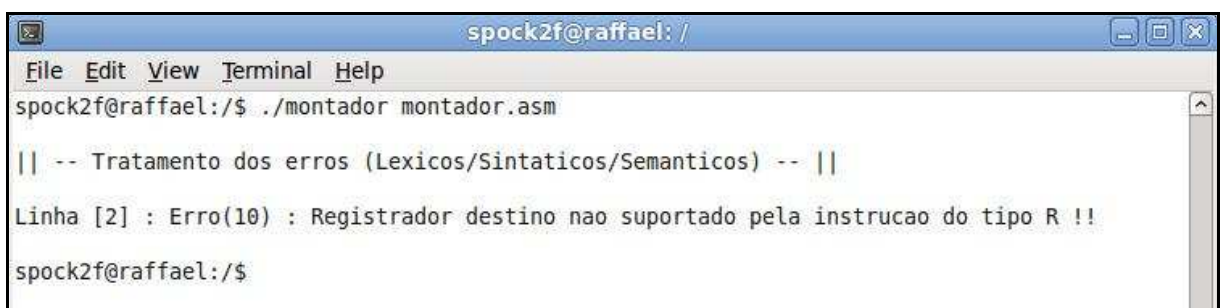
A terminal window titled 'spock2f@raffael: /' with a menu bar (File, Edit, View, Terminal, Help). The command './montador montador.asm' has been executed. The output shows a separator line '|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||' followed by the error message 'Linha [2] : Erro(13) : Constante nao suportada !!'. The prompt 'spock2f@raffael:/\$' is visible at the bottom.

```
spock2f@raffael: /
File Edit View Terminal Help
spock2f@raffael:/$ ./montador montador.asm

|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||
Linha [2] : Erro(13) : Constante nao suportada !!
spock2f@raffael:/$
```

*Figura 4.8: Constante não suportada pela instrução do Formato-R*

Na linha 6 é possível verificar que a instrução utiliza o modo de endereçamento indireto(am = 01) e realiza sua operação baseada no conteúdo apontado pelos registradores presentes no corpo da instrução, porém é possível verificar que o registrador \$r16 é um registrador não suportado pelo banco de registradores. Abaixo é possível visualizar qual foi a saída do montador ao executar a instrução:


A terminal window titled 'spock2f@raffael: /' with a menu bar (File, Edit, View, Terminal, Help). The command './montador montador.asm' has been executed. The output shows a separator line '|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||' followed by the error message 'Linha [2] : Erro(10) : Registrador destino nao suportado pela instrucao do tipo R !!'. The prompt 'spock2f@raffael:/\$' is visible at the bottom.

```
spock2f@raffael: /
File Edit View Terminal Help
spock2f@raffael:/$ ./montador montador.asm

|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||
Linha [2] : Erro(10) : Registrador destino nao suportado pela instrucao do tipo R !!
spock2f@raffael:/$
```

*Figura 4.9: Registrador não suportado na instrução do Formato-R*

Na linha 7 é possível verificar que a instrução add possui excesso de parâmetros de acordo com a especificação do ISA do AIDA-16. Abaixo é possível visualizar qual foi a saída do montador ao executar a instrução:



```

spock2f@raffael: /
File Edit View Terminal Help
spock2f@raffael:/$ ./montador montador.asm

|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||

Linha [2] : Erro(19) : Excesso de parametros !!

spock2f@raffael:/$ █

```

*Figura 4.10: Excesso de parâmetros na instrução do Formato-R*

Como se pode verificar nas figuras acima, todas as instruções de soma com erros descritas anteriormente, foram detectadas com sucesso, gerando erros léxicos, sintáticos e semânticos corretamente alertando o programador que o código do arquivo assembly de entrada não corresponde ao código como descrito na especificação do processador AIDA-16.

#### 4.1.3 Testando instruções validas do Formato – I

Para testar as combinações validas mais importantes dos campos do Formato-R, foi proposto o seguinte bloco de instruções:

1	li	0,50
2	li	1,45

*Figura 4.11 : Instruções validas do formato - I*

Este mesmo bloco de instruções validas que foi aplicado na instrução li também foi replicado para a instrução lui do Formato-I, para assim garantir que o montador esta de acordo com a ISA do AIDA-16 para todas as instruções do Formato-I.

Como se pode visualizar no bloco de instruções acima, existem números em cada linha que são responsáveis por referenciar cada instrução neste exemplo, não fazendo parte do corpo da instrução. Dando inicio ao teste de validação do bloco de instruções validas, veremos que a função da linha 1 é de efetuar a carga da constante de valor 50 na parte baixa do registrador \$t0. Já a segunda linha tem como objetivo carregar a constante 45 na parte baixa do registrador \$t1. Ao encaminhar estas duas instruções com as demais diretivas

de montagem em um arquivo assembly ao montador, obtiveram-se os seguintes resultados de montagem, como podem ser visualizados na figura abaixo:



```

spock2f@raffael: /
File Edit View Terminal Help
spock2f@raffael:/$ ./montador montador.asm
spock2f@raffael:/$
  
```

*Figura 4.12: Teste das instruções válidas do Formato – I*

Como se pode verificar na figura acima, todas as instruções de li descritas anteriormente foram montadas com sucesso, pois o montador não gerou nenhum tipo de erro léxico, sintático ou semântico nas instruções.

#### 4.1.4 Testando instruções inválidas do Formato – I

Para testar as combinações inválidas mais importantes dos campos do Formato-I, foi proposto o seguinte bloco de instruções:

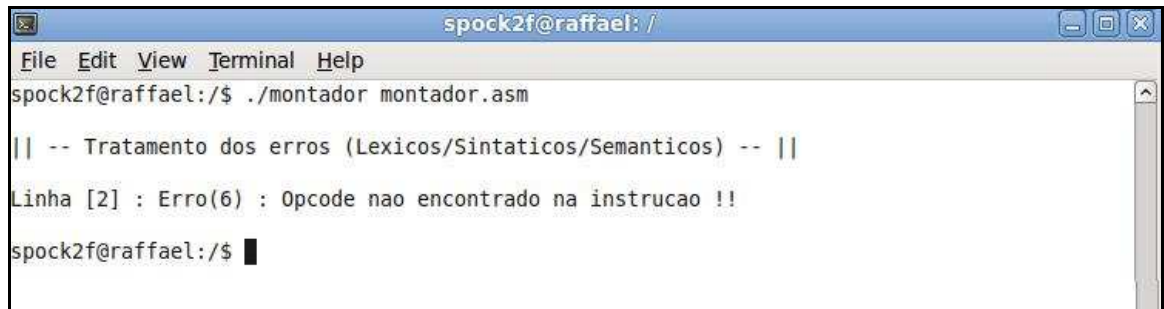
1	li	00,1,\$r3
2	li	3,\$r3
3	li	0,
4	li	1,5,\$r0
5	li	0,255

*Figura 4.13: Instruções inválidas do formato - I*

Este mesmo bloco de instruções inválidas que foi aplicado na instrução li também foi replicado para a instrução lui do Formato-I, para só assim garantir que o montador contempla os formatos inválidos das instruções do Formato-I e gera corretamente as mensagens de erros.

Como se pode visualizar no bloco de instruções acima, existem números em cada linha que são responsáveis por referenciar cada instrução neste exemplo, não fazendo parte

do corpo da instrução. Dando início ao teste de validação do bloco de instruções inválidas, veremos que a função da linha 1 é testar se o montador é capaz de reconhecer que o campo de opcode da instrução não pertence ao ISA AIDA-16. Abaixo é possível obter o resultado gerado pelo montador ao executar esta instrução:

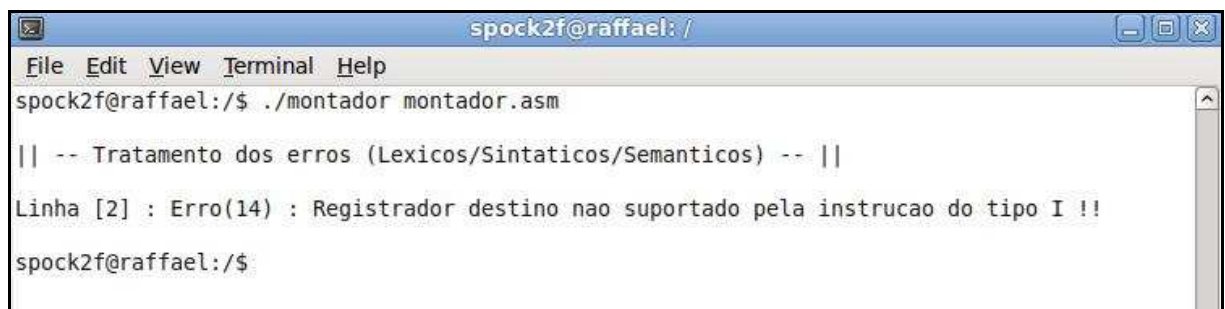


```
spock2f@raffael: /
File Edit View Terminal Help
spock2f@raffael:/$ ./montador montador.asm

|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||
Linha [2] : Erro(6) : Opcode nao encontrado na instrucao !!
spock2f@raffael:/$
```

*Figura 4.14: Código da operação não encontrado*

Na linha 2 é possível verificar que a instrução li possui o registrador de destino (rd = 3) que pela especificação do AIDA-16, não é um registrador de destino válido. Abaixo é possível visualizar qual foi a saída do montador ao executar a instrução:

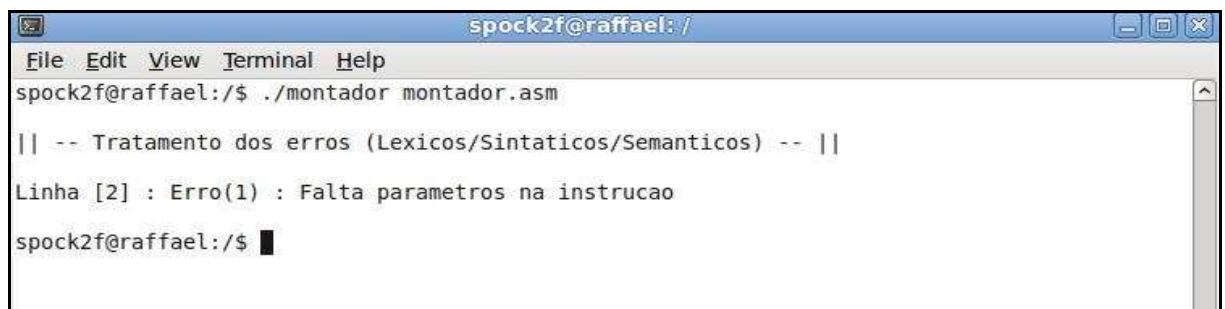


```
spock2f@raffael: /
File Edit View Terminal Help
spock2f@raffael:/$ ./montador montador.asm

|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||
Linha [2] : Erro(14) : Registrador destino nao suportado pela instrucao do tipo I !!
spock2f@raffael:/$
```

*Figura 4.15: Registrador de destino não suportado na instrução do Formato-I*

Na linha 3 é possível verificar que falta um dos parâmetros da instrução li, pelo que foi definido na especificação do AIDA-16. Abaixo é possível visualizar qual foi a saída do montador ao executar a instrução:



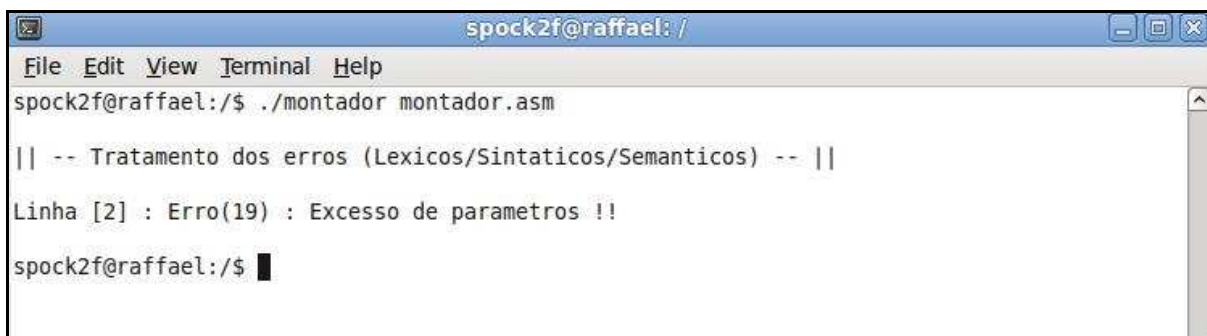
```
spock2f@raffael: /
File Edit View Terminal Help
spock2f@raffael:/$ ./montador montador.asm

|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||
Linha [2] : Erro(1) : Falta parametros na instrucao
spock2f@raffael:/$
```

*Figura 4.16: Falta operandos na instrução do Formato-I*



Na linha 4 é possível verificar que existem excesso de parâmetros na instrução li pelo que foi definido na especificação do AIDA-16. Abaixo é possível visualizar qual foi a saída do montador ao executar a instrução:

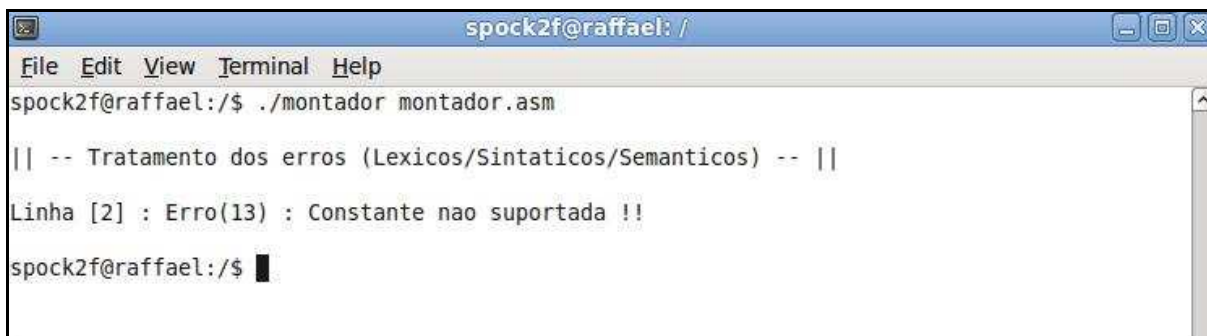
A terminal window titled 'spock2f@raffael: /' with a menu bar (File, Edit, View, Terminal, Help). The command './montador montador.asm' has been executed. The output shows a separator line '|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||' followed by the error message 'Linha [2] : Erro(19) : Excesso de parametros !!'. The prompt 'spock2f@raffael:/\$' is shown at the bottom with a cursor.

```
spock2f@raffael: /
File Edit View Terminal Help
spock2f@raffael:/$ ./montador montador.asm

|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||
Linha [2] : Erro(19) : Excesso de parametros !!
spock2f@raffael:/$
```

*Figura 4.17: Excesso de parâmetros na instrução do Formato-I*

Na linha 5 é possível verificar que a constante na instrução excede o limite de representação, baseado na definição do AIDA-16. Abaixo é possível visualizar qual foi a saída do montador ao executar a instrução:

A terminal window titled 'spock2f@raffael: /' with a menu bar (File, Edit, View, Terminal, Help). The command './montador montador.asm' has been executed. The output shows a separator line '|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||' followed by the error message 'Linha [2] : Erro(13) : Constante nao suportada !!'. The prompt 'spock2f@raffael:/\$' is shown at the bottom with a cursor.

```
spock2f@raffael: /
File Edit View Terminal Help
spock2f@raffael:/$ ./montador montador.asm

|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||
Linha [2] : Erro(13) : Constante nao suportada !!
spock2f@raffael:/$
```

*Figura 4.18: Constante não suportada na instrução do Formato-I*

Como se pode verificar nas figuras acima, todas as instruções de li com erros descritas anteriormente, foram detectadas com sucesso, gerando erros léxicos, sintáticos e semânticos corretamente alertando o programador que o código do arquivo assembly de entrada não corresponde ao código descrito na especificação do processador AIDA-16.

#### 4.1.5 Testando instruções validas do Formato – Load/Store

Para testar as combinações validas mais importantes dos campos do Formato-Load/Store, foi proposta a seguinte instrução:

```
1 lw $r0,$r1
```

*Figura 4.19: Instruções validas do formato – Load/Store*

Esta instrução valida que foi aplicada a instrução lw também foi replicado para as instruções formato-Load/Store, para assim garantir que o montador esta de acordo com a ISA do AIDA-16 para todas as instruções do Formato-Load/Store.

Dando inicio ao teste de validação do bloco de instruções validas, veremos que a função da instrução lw acima é de efetuar a carga do índice de memória apontado por \$r1 para o registrador \$r0. Ao encaminhar esta instrução com as demais diretivas de montagem em um arquivo assembly ao montador, obteve-se os seguintes resultados de montagem, como podem ser visualizados na figura abaixo:



```
spock2f@raffael: /
File Edit View Terminal Help
spock2f@raffael:/$ ./montador montador.asm
spock2f@raffael:/$
```

*Figura 4.20: Teste das instruções validas do Formato – Load/Store*

Como se pode verificar na figura acima, a instrução lw descrita anteriormente foi montada com sucesso, pois o montador não gerou nenhum tipo de erro léxico, sintático ou semântico na instrução.

#### 4.1.6 Testando instruções inválidas do Formato – Load/Store

Para testar as combinações invalidas mais importantes dos campos do formato-Load/Store, foi proposto o seguinte bloco de instruções:

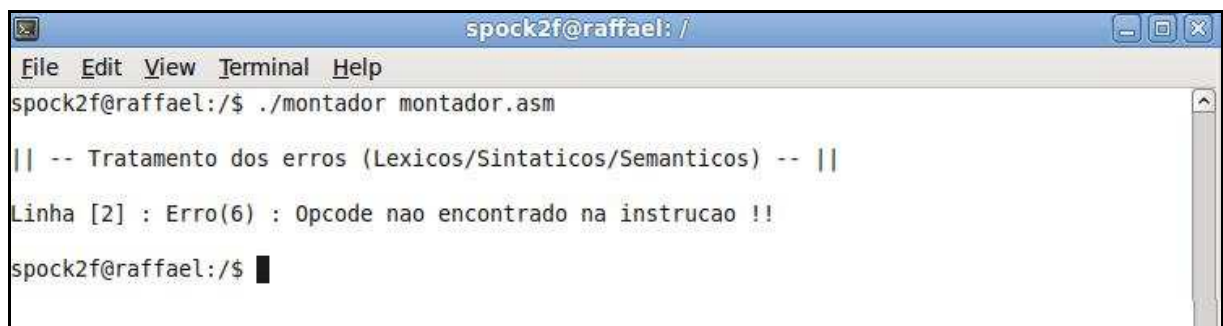
```
1 l 00,1,$r3
2 lw $r3,$r16
3 lw $r0,
4 lw $r0,$r1,$r0
```



*Figura 4.21: Instruções inválidas do formato – Load/Store*

Este mesmo bloco de instruções inválidas que foi aplicado na instrução lw também foi replicado para a instruções do formato-Load/Store, para só assim garantir que o montador contempla os formatos inválidos das instruções do formato-Load/Store e gere corretamente as mensagens de erros.

Como se pode visualizar no bloco de instruções acima, existem números em cada linha que são responsáveis por referenciar cada instrução neste exemplo, não fazendo parte do corpo da instrução. Dando inicio ao teste de validação do bloco de instruções invalidas, veremos que a função da linha 1 é testar se o montador é capaz de reconhecer que o campo de opcode da instrução não pertence ao ISA AIDA-16. Abaixo é possível obter o resultado gerado pelo montador ao executar esta instrução :

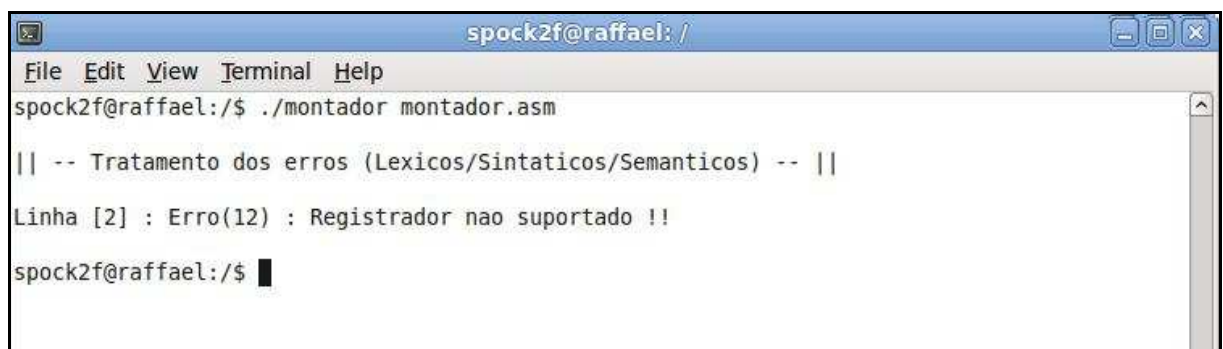
A terminal window titled 'spock2f@raffael: /' with a menu bar (File, Edit, View, Terminal, Help). The command './montador montador.asm' has been executed. The output shows a separator line '|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||' followed by the error message 'Linha [2] : Erro(6) : Opcode nao encontrado na instrucao !!'. The prompt 'spock2f@raffael:/\$' is visible at the bottom.

```
spock2f@raffael: /
File Edit View Terminal Help
spock2f@raffael:/$ ./montador montador.asm

|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||
Linha [2] : Erro(6) : Opcode nao encontrado na instrucao !!
spock2f@raffael:/$
```

*Figura 4.22: Código da operação não encontrado*

Na linha 2 é possível verificar que a instrução lw possui um registrador não suportado pelo banco de registradores que pela especificação do AIDA-16, não é um registrador valido. Abaixo é possível visualizar qual foi a saída do montador ao executar a instrução:

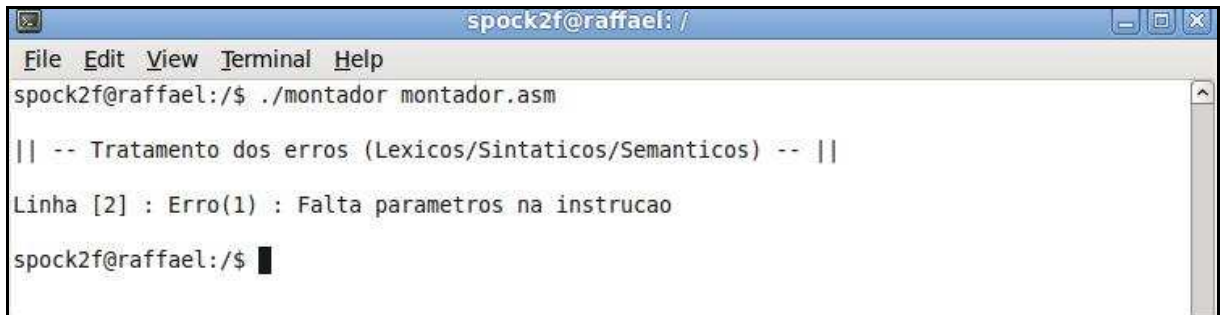
A terminal window titled 'spock2f@raffael: /' with a menu bar (File, Edit, View, Terminal, Help). The command './montador montador.asm' has been executed. The output shows a separator line '|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||' followed by the error message 'Linha [2] : Erro(12) : Registrador nao suportado !!'. The prompt 'spock2f@raffael:/\$' is visible at the bottom.

```
spock2f@raffael: /
File Edit View Terminal Help
spock2f@raffael:/$ ./montador montador.asm

|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||
Linha [2] : Erro(12) : Registrador nao suportado !!
spock2f@raffael:/$
```

*Figura 4.23: Registrador não suportado na instrução do Formato – Load/Store*

Na linha 3 é possível verificar que falta um dos parâmetros da instrução lw, pelo que foi definido na especificação do AIDA-16. Abaixo é possível visualizar qual foi a saída do montador ao executar a instrução:



```
spock2f@raffael: /
File Edit View Terminal Help
spock2f@raffael:/$ ./montador montador.asm

|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||
Linha [2] : Erro(1) : Falta parametros na instrucao
spock2f@raffael:/$
```

*Figura 4.24: Faltam parâmetros na instrução do Formato – Load/Store*

Na linha 4 é possível verificar que existem excesso de parâmetros na instrução lw pelo que foi definido na especificação do AIDA-16. Abaixo é possível visualizar qual foi a saída do montador ao executar a instrução:



```
spock2f@raffael: /
File Edit View Terminal Help
spock2f@raffael:/$ ./montador montador.asm

|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||
Linha [2] : Erro(19) : Excesso de parametros !!
spock2f@raffael:/$
```

*Figura 4.25: Excesso de parâmetros na instrução do Formato – Load/Store*

Como se pode verificar nas figuras acima, todas as instruções de lw com erros descritos anteriormente, foram detectadas com sucesso, gerando erros léxicos, sintáticos e semânticos corretamente alertando o programador que o código do arquivo assembly de entrada não corresponde ao código descrito na especificação do processador AIDA-16.

#### 4.1.7 Testando instruções validas do Formato – Branch

Para testar as combinações validas mais importantes dos campos do formato-Branch, foi proposta a seguinte instrução:

1 jv 1

*Figura 4.26 : Instruções validas do formato – Branch*

Esta instrução valida que foi aplicada a instrução jv também foi replicada para as instruções formato-Branch, para assim garantir que o montador esta de acordo com a ISA do AIDA-16 para todas as instruções do formato-Branch.

Dando inicio ao teste de validação do bloco de instruções validas, veremos que a função da instrução jv acima é de caso o valor da operação da instrução anterior der zero, o contador de programa (PC), deverá ser alterado com o incremento de uma instrução a frente. Ao encaminhar esta instrução com as demais diretivas de montagem em um arquivo assembly ao montador, obteve-se os seguintes resultados de montagem, como podem ser visualizados na figura abaixo:



```

spock2f@raffael: /
File Edit View Terminal Help
spock2f@raffael:/$ ./montador montador.asm
spock2f@raffael:/$

```

*Figura 4.27: Teste das instruções validas do Formato – Branch*

Como se pode verificar na figura acima, a instrução jv descrita anteriormente foi montada com sucesso, pois o montador não gerou nenhum tipo de erro léxico, sintático ou semântico na instrução.

#### 4.1.8 Testando instruções inválidas do Formato - Branch

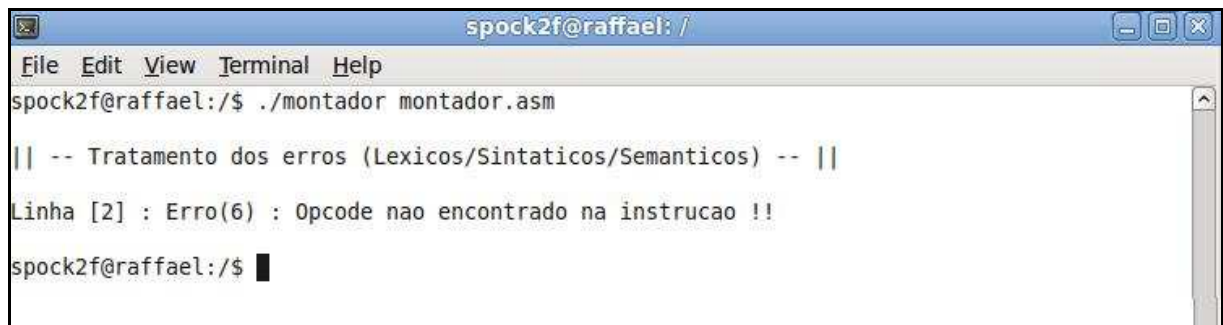
Para testar as combinações invalidas mais importantes dos campos do formato-Branch, foi proposto o seguinte bloco de instruções:

1 j 1  
2 jv 2049  
3 jv

*Figura 4.28: Instruções inválidas do formato – Branch*

Este mesmo bloco de instruções inválidas que foi aplicado na instrução `jv` também foi replicado para a instruções do formato-Branch, para só assim garantir que o montador contempla os formatos inválidos das instruções do formato-Branch e gere corretamente as mensagens de erros.

Como se pode visualizar no bloco de instruções acima, existem números em cada linha que são responsáveis por referenciar cada instrução neste exemplo, não fazendo parte do corpo da instrução. Dando inicio ao teste de validação do bloco de instruções invalidas, veremos que a função da linha 1 é verificar se o montador é capaz de reconhecer que o campo de opcode da instrução não pertence ao ISA AIDA-16. Abaixo é possível obter o resultado gerado pelo montador ao executar esta instrução:

A terminal window titled 'spock2f@raffael: /' with a menu bar (File, Edit, View, Terminal, Help). The command './montador montador.asm' has been executed. The output shows a separator line '|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||' followed by the error message 'Linha [2] : Erro(6) : Opcode nao encontrado na instrucao !!'. The prompt 'spock2f@raffael:/\$' is shown with a cursor.

```
spock2f@raffael: /
File Edit View Terminal Help
spock2f@raffael:/$ ./montador montador.asm

|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||
Linha [2] : Erro(6) : Opcode nao encontrado na instrucao !!
spock2f@raffael:/$
```

*Figura 4.29: Código da operação não encontrado*

Na linha 2 é possível verificar que a instrução `jv` possui a constante de desvio maior que a especificada no ISA do AIDA-16. Abaixo é possível visualizar qual foi a saída do montador ao executar a instrução:

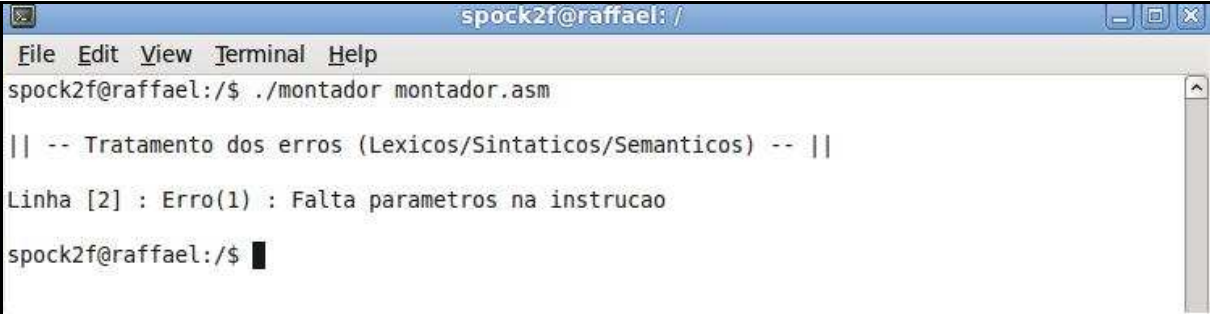
A terminal window titled 'spock2f@raffael: /' with a menu bar (File, Edit, View, Terminal, Help). The command './montador montador.asm' has been executed. The output shows a separator line '|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||' followed by the error message 'Linha [2] : Erro(13) : Constante nao suportada !!'. The prompt 'spock2f@raffael:/\$' is shown with a cursor.

```
spock2f@raffael: /
File Edit View Terminal Help
spock2f@raffael:/$ ./montador montador.asm

|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||
Linha [2] : Erro(13) : Constante nao suportada !!
spock2f@raffael:/$
```

*Figura 4.30: Constante não suportada na instrução do Formato – Branch*

Na linha 3 é possível verificar que faltam parâmetros na instrução `lv` de acordo com o especificado no ISA AIDA-16. Abaixo é possível visualizar qual foi a saída do montador ao executar a instrução:

A terminal window titled 'spock2f@raffael: /' with a menu bar (File, Edit, View, Terminal, Help). The command prompt shows 'spock2f@raffael:/\$ ./montador montador.asm'. The output displays a separator line '|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||', followed by the error message 'Linha [2] : Erro(1) : Falta parametros na instrucao', and ends with the prompt 'spock2f@raffael:/\$' and a cursor.

```
spock2f@raffael: /
File Edit View Terminal Help
spock2f@raffael:/$ ./montador montador.asm

|| -- Tratamento dos erros (Lexicos/Sintaticos/Semanticos) -- ||
Linha [2] : Erro(1) : Falta parametros na instrucao
spock2f@raffael:/$
```

*Figura 4.31: Falta parâmetros na instrução do Formato - Branch*

## 4.2 Validação do ISS conforme a especificação do processador AIDA-16

Para validar o ISS o autor desenvolveu um arquivo assembly chamado `ISS.asm`, responsável por implementar todas as operações capazes de testar todas as estruturas do simulador, validando seu funcionamento. Para que isso fosse possível, o produto anterior do trabalho, ferramenta de base para gerar o arquivo binário de uso do ISS que é o montador precisou ser implementado e verificado primeiro, para que só então o ISS depois de implementado pudesse ser testado. Afim de que seja explícito ao leitor, todos os trechos de códigos descritos nesta seção deste Capítulo, serão carregados para o montador com as diretivas de montagem necessárias para que o arquivo seja um arquivo válido em nível de montagem, e será gerado seu respectivo arquivo binário, que será a entrada do ISS para apresentar as respectivas interfaces demonstradas como exemplo de validação dos formatos de instrução do ISS. Abaixo é visualizado trechos de código responsáveis por validar os formatos de instrução do ISA AIDA-16.

### 4.2.1 Testando as instruções do Formato – R no ISS

Para testar as instruções do formato-R foi proposto a implementação de uma equação capaz de contemplar um grande número de instruções lógicas e aritméticas, que devido ao volume e ao número de variantes frente ao modo de endereçamento e aos registradores de destino, terá a mesma relevância que validar cada instrução e suas variantes separadamente, como feito no arquivo `ISS.asm`, que foi testado inúmeras vezes e valida o funcionamento do ISS.

A equação especificada para testar add, sub, inc, dec, mul, div, mod, shr, shl, ror, rol, or, and, xor, not é :

$$S = ((((((((((a - b) / (c++) + (d--) \% 2) \text{ xor } 2) \text{ shl } ) \text{ shr} ) \text{ ror} ) \text{ rol} ) \text{ and } 0) \text{ not} ) \text{ not} ) \text{ or } 1)$$

*Figura 4.32: Equação que valida as instruções lógicas e aritméticas*

Os valores de entrada são a = 11, b = 1, c = 4, d = 1

O valor de correto de saída baseado nos valores de entrada da deverá ser = 1 para o modo de endereçamento direto que será testado, onde seu código fonte esta referenciado no Apêndice B deste trabalho. Abaixo é possível visualizar nas figuras da interface do ISS que demonstram o valor de S representado no registrador \$r0, validando o funcionamento de todas as variações das instruções lógicas e aritméticas do ISS para o modo de endereçamento direto. Por motivos de simplicidade, o autor não listou neste trabalho a validação dos modos de endereçamento indireto e imediato, que foram testados apenas em tempo de desenvolvimento do ISS através do arquivo ISS.asm que contempla todas as combinações das instruções lógicas e aritméticas para os modos de endereçamento indireto e imediato.

Validando o código que consta no apêndice B das instruções lógicas e aritméticas no modo direto:

Banco Reg	Memoria de Instrucao	Mem Data
BD[ 0] : 1	MInst[27] : halt	Mdata[ 0] : 11
BD[ 1] : 1	MInst[28] :	Mdata[ 1] : 1
BD[ 2] : 5	MInst[29] :	Mdata[ 2] : 4
BD[ 3] : 1	MInst[30] :	Mdata[ 3] : 1
BD[ 4] : 0	MInst[31] :	Mdata[ 4] :
BD[ 5] : 0	MInst[32] :	Mdata[ 5] :
BD[ 6] : 0	MInst[33] :	Mdata[ 6] :
BD[ 7] : 0	MInst[34] :	Mdata[ 7] :
BD[ 8] : 0	MInst[35] :	Mdata[ 8] :
BD[ 9] : 0	MInst[36] :	Mdata[ 9] :
BD[10] : 0	MInst[37] :	Mdata[10] :
BD[11] : 0	MInst[38] :	Mdata[11] :
BD[12] : 0	MInst[39] :	Mdata[12] :
BD[13] : 0	MInst[40] :	Mdata[13] :
BD[14] : 1	MInst[41] :	Mdata[14] :
BD[15] : 2	MInst[42] :	Mdata[15] :

Instrucao[27] : halt  
Pressione uma tecla para continuar !! [ ]

*Figura 4.33: Validando instruções lógicas e aritméticas no modo de end.direto*

#### 4.2.2 Testando as instruções do Formato – I no ISS

Para testar as instruções do formato-I foi proposto um pequeno trecho de código, já que existem somente duas instruções e existem poucas variações de código possíveis de serem feitas. Abaixo é possível verificar o bloco de código proposto para a instrução li e para a instrução lui e a interface do ISS mostrando os registradores \$t0 com 1 e \$t1 com 512 validando o formato-i.

```
1 li 0,1
2 lui 1,2
```

Figura 4.34 : Instruções que validam o formato – I

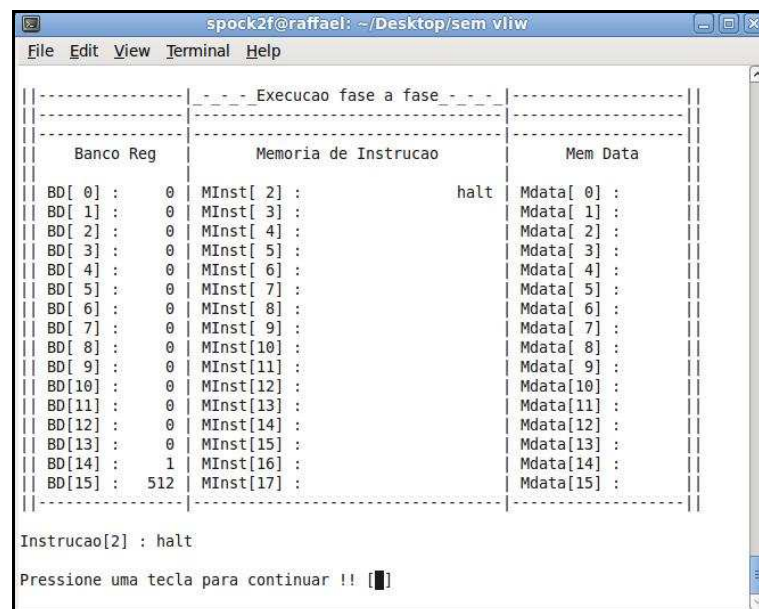


Figura 4.35: Validando as instruções li e lui do formato - I

#### 4.2.3 Testando as instruções do Formato – Load/Store no ISS

Para testar as instruções do formato-Load/Store foi proposto um pequeno trecho de código, já que existem poucas combinações de instruções do Formato Load/Store e algumas instruções para serem contempladas. Abaixo é possível verificar o bloco de código proposto para as instruções lb/sb sw/sb e mov com a interface do ISS mostrando os resultados da operação das instruções. Abaixo é possível verificar o bloco de código proposto que valida o que foi dito acima:



```

1 .text
2 li 0,0 # Carrega 0 para $t0
3 li 1,3 # Carrega 3 para $t1
4 lb $r0,$t0 # Carrega o endereço$t0(Mem[0]) para o registrador $r0
5 sb $t1,$r0 # Armazena o conteúdo do registrador 0 em $t1(Mem[1])
6 li 0,1 # Carrega 1 para $t0
7 li 1,4 # Carrega 4 para $t1
8 lw $r1,$t0 # Carrega a palavra(dois enderecos) $t0(Mem[1] e Mem[2]) para $r1
9 sw $t1,$r1 # Armazena o registrador $r1 em $t1(Mem[4] e Mem[5])
10 mov $r0,$r2 # Move o conteúdo de $r0 para $r2
11halt # Para a execução do ISS
12 .data
13 .byte 8 # Carrega a celula Mem[0] com o conteúdo 8
14 .word 9 # Carrega as células Mem[1] e Mem[2] com o conteúdo 9

```

*Figura 4.36: Instruções que validam o formato – Load/Store*

Abaixo a tela da interface que demonstra nas posições citadas nos comentários das instruções os respectivos valores ao carregar este código para o ISS:

```

spock2f@rafael: ~/Desktop/sem vliw
File Edit View Terminal Help

```

Banco Reg		Memoria de Instrucao		Mem Data	
BD[ 0 ] :	0	MInst[ 2 ] :	halt	Mdata[ 0 ] :	
BD[ 1 ] :	0	MInst[ 3 ] :		Mdata[ 1 ] :	
BD[ 2 ] :	0	MInst[ 4 ] :		Mdata[ 2 ] :	
BD[ 3 ] :	0	MInst[ 5 ] :		Mdata[ 3 ] :	
BD[ 4 ] :	0	MInst[ 6 ] :		Mdata[ 4 ] :	
BD[ 5 ] :	0	MInst[ 7 ] :		Mdata[ 5 ] :	
BD[ 6 ] :	0	MInst[ 8 ] :		Mdata[ 6 ] :	
BD[ 7 ] :	0	MInst[ 9 ] :		Mdata[ 7 ] :	
BD[ 8 ] :	0	MInst[10] :		Mdata[ 8 ] :	
BD[ 9 ] :	0	MInst[11] :		Mdata[ 9 ] :	
BD[10] :	0	MInst[12] :		Mdata[10] :	
BD[11] :	0	MInst[13] :		Mdata[11] :	
BD[12] :	0	MInst[14] :		Mdata[12] :	
BD[13] :	0	MInst[15] :		Mdata[13] :	
BD[14] :	1	MInst[16] :		Mdata[14] :	
BD[15] :	512	MInst[17] :		Mdata[15] :	

```

Instrucao[2] : halt
Pressione uma tecla para continuar !! [ ]

```

*Figura 4.37: Validando as instruções do Formato Load/Store*



A partir dos comentários feitos no código é possível verificar que o registrador \$r0 contém o conteúdo de Mem[0] como efetuado pela instrução lb e o conteúdo de Mem[3] contém o conteúdo 8 como efetuado pela instrução sb. Também é possível verificar que \$r1 possui o conteúdo de Mem[1] e Mem[2] como efetuado pela instrução lw e Mem[4] e Mem[5] como efetuado pela instrução lw. O registrador \$r2 possui o valor 8 como efetuado pela instrução mov. Com isso é possível validar que o ISS opera com sucesso o bloco de código demonstrado acima.

#### 4.2.4 Testando as instruções do Formato – Branch no ISS

Para testar as instruções do Formato – Branch foi desenvolvido o seguinte código fonte:

```
1 .text
2 add 10,0,0,0
3 jz 1
4 nop
5 add 10,0,1,1
6 jnz 3
7 add 10,0,10,10
8 halt
9 nop
10 add 10,0,0,0
11 jnc 1
12 nop
13 jc 2
14 jv 3
15 jnz 1
16 jmp - 10
17 .data
```

*Figura 4.38: Bloco de instruções que valida o formato - Branch*

Abaixo a tela da interface que demonstra na posição \$t0 o valor 20 que valida o funcionamento de todas as instruções do formato-branch:

```

spock2f@rafael: ~/Desktop/sem vliw
File Edit View Terminal Help

----- Execução fase a fase -----
-----
Banco Reg      Memoria de Instrucao      Mem Data
-----
BD[ 0] :      8 MInst[ 9] :      halt Mdata[ 0] :      8
BD[ 1] :      9 MInst[10] :      Mdata[ 1] :      0
BD[ 2] :      8 MInst[11] :      Mdata[ 2] :      9
BD[ 3] :      0 MInst[12] :      Mdata[ 3] :      8
BD[ 4] :      0 MInst[13] :      Mdata[ 4] :      0
BD[ 5] :      0 MInst[14] :      Mdata[ 5] :      9
BD[ 6] :      0 MInst[15] :      Mdata[ 6] :
BD[ 7] :      0 MInst[16] :      Mdata[ 7] :
BD[ 8] :      0 MInst[17] :      Mdata[ 8] :
BD[ 9] :      0 MInst[18] :      Mdata[ 9] :
BD[10] :      0 MInst[19] :      Mdata[10] :
BD[11] :      0 MInst[20] :      Mdata[11] :
BD[12] :      0 MInst[21] :      Mdata[12] :
BD[13] :      0 MInst[22] :      Mdata[13] :
BD[14] :      1 MInst[23] :      Mdata[14] :
BD[15] :      4 MInst[24] :      Mdata[15] :

Instrucao[9] : halt
Pressione uma tecla para continuar !!

```

Figura 4.39 : Validando as instruções do Formato Branch

### 4.3 Validação do montador VLIW para o ISA do AIDA-16

A fim de validar o bloco de tratamento de conflitos de dados do montador VLIW foi desenvolvido o arquivo vliw.asm, que contempla todas as combinações envolvendo todas as 31 instruções do ISA do AIDA-16. Abaixo é feita algumas das comparações principais que demonstram os conflitos encontrados entre a combinação dos 4 formatos de instrução, juntamente com seus campos de registrador de destino e modos de endereçamento. Se o leitor reparar, foi testado somente um conjunto de combinações capazes de gerar dependência, porem como a ISA contempla muitos campos com diversas variações, é possível obter muitas combinações, onde somente algumas foram citadas neste texto para fim de validar o funcionamento, porem o arquivo vliw.asm tem como objetivo validar todas as combinações possíveis existente, resultado em mais de 1000 linhas de código assembly que com isso, valida que o montador VLIW é capaz de detectar todas as combinações de instruções que contenham dependência.

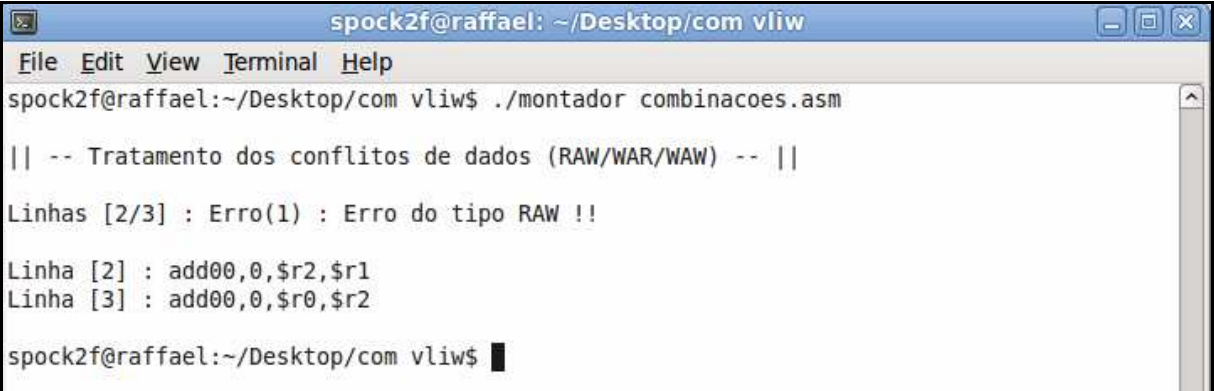
#### 1. Formato R – Formato R

Como já foi explorado no Capítulo 3 sobre quais as dependências de dados podem acontecer sobre duas instruções do formato R sobrepostas, abaixo será validado todas as combinações envolvendo os modos de endereçamento de duas instruções do tipo R.

❖ Formato R (Direto) – Formato R (Direto)

Neste caso pode acontecer RAW WAR e WAW como mostrado nas figuras abaixo :

RAW: Registrador \$r2 é escrito pela linha 2 e lido pela linha 3



```

spock2f@raffael: ~/Desktop/com vliw
File Edit View Terminal Help
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

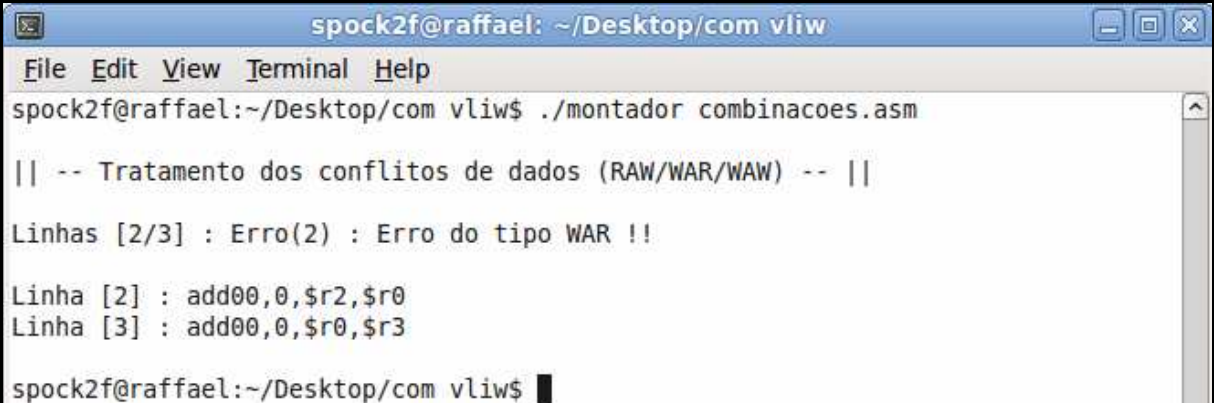
Linhas [2/3] : Erro(1) : Erro do tipo RAW !!

Linha [2] : add00,0,$r2,$r1
Linha [3] : add00,0,$r0,$r2

spock2f@raffael:~/Desktop/com vliw$
  
```

*Figura 4.40: Validando a dependência do tipo RAW em R (Direto) – R (Direto)*

WAR: Registrador \$r0 é lido pela instrução da linha 2 e escrito pela linha 3



```

spock2f@raffael: ~/Desktop/com vliw
File Edit View Terminal Help
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

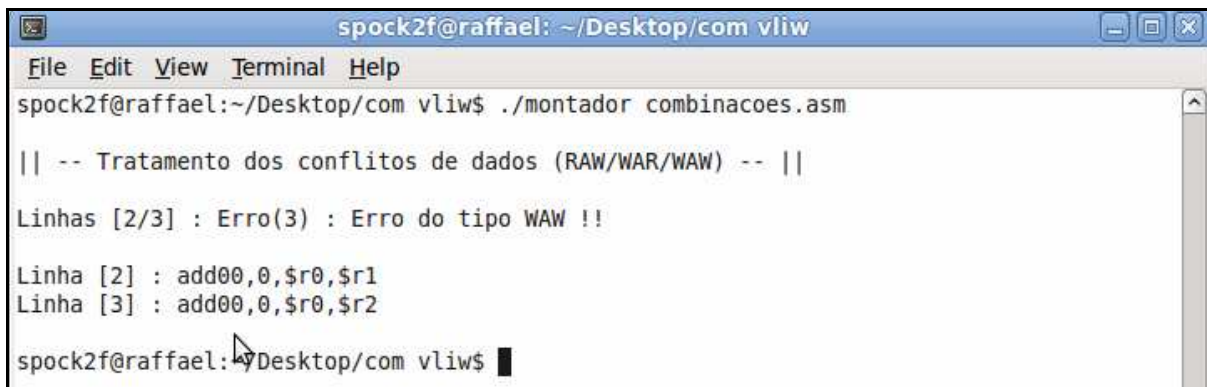
Linhas [2/3] : Erro(2) : Erro do tipo WAR !!

Linha [2] : add00,0,$r2,$r0
Linha [3] : add00,0,$r0,$r3

spock2f@raffael:~/Desktop/com vliw$
  
```

*Figura 4.41: Validando a dependência do tipo WAR em R (Direto) – R (Direto)*

WAW: Registrador \$r0 é escrito tanto pela linha 2 como pela linha 3



```

spock2f@raffael: ~/Desktop/com vliw
File Edit View Terminal Help
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(3) : Erro do tipo WAW !!

Linha [2] : add00,0,$r0,$r1
Linha [3] : add00,0,$r0,$r2

spock2f@raffael:~/Desktop/com vliw$

```

*Figura 4.42: Validando a dependência do tipo WAW em R (Direto) – R (Direto)*

❖ Formato R (Direto) – Formato R (Imediato)

Neste caso pode acontecer WAR e WAW como mostrado nas figuras abaixo :

RAW: Como pode ser verificado, não acontece RAW em Direto - Imediato



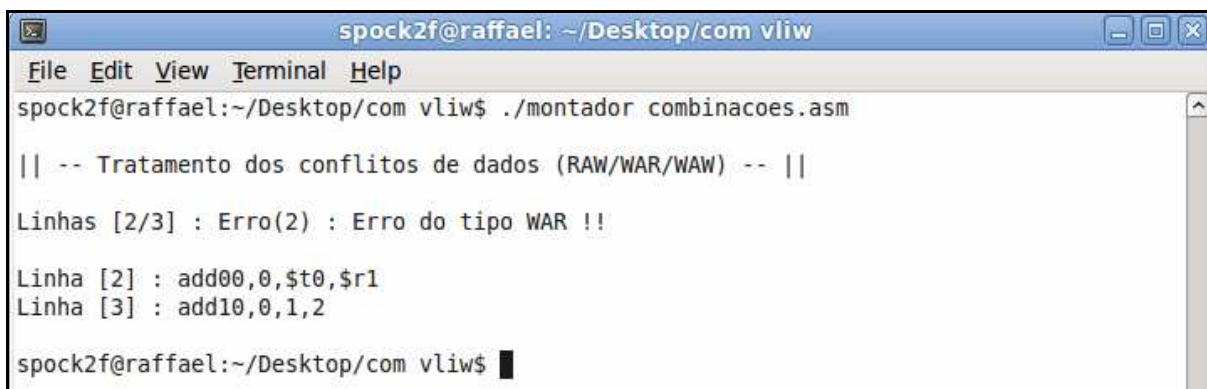
```

spock2f@raffael: ~/Desktop/com vliw
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm
spock2f@raffael:~/Desktop/com vliw$

```

*Figura 4.43: Validando que não existe dependência do tipo RAW em R (Direto) – R (Imediato)*

WAR: Registrador \$t0 é lido na linha 2 e escrito na linha 3



```

spock2f@raffael: ~/Desktop/com vliw
File Edit View Terminal Help
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(2) : Erro do tipo WAR !!

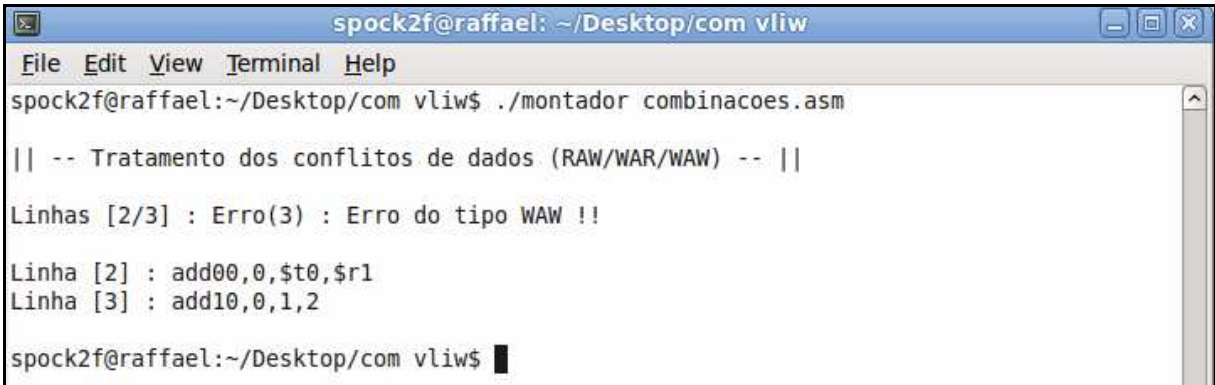
Linha [2] : add00,0,$t0,$r1
Linha [3] : add10,0,1,2

spock2f@raffael:~/Desktop/com vliw$

```

*Figura 4.44: Validando a dependência do tipo WAR em R (Direto) – R (Imediato)*

WAW: Registrador \$t0 é escrito na linha 2 e na linha 3



```

spock2f@raffael: ~/Desktop/com vliw
File Edit View Terminal Help
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(3) : Erro do tipo WAW !!

Linha [2] : add00,0,$t0,$r1
Linha [3] : add10,0,1,2

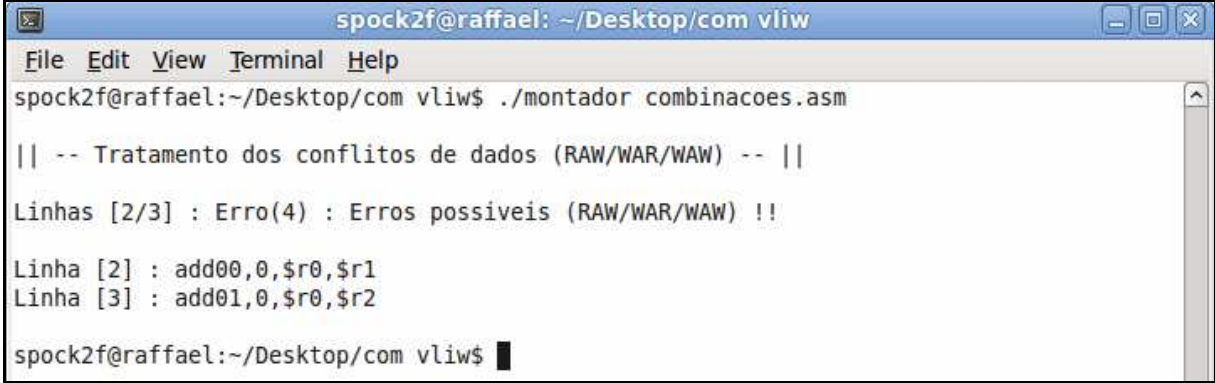
spock2f@raffael:~/Desktop/com vliw$

```

*Figura 4.45: Validando a dependência do tipo WAW em R (Direto) – R (Imediato)*

#### ❖ Formato R (Imediato) – Formato R (Indireto)

Neste caso pode acontecer tanto RAW como WAR e WAW porem como é possível visualizar abaixo, não é possível detectar o conteúdo apontado pelos registradores da linha 3 pois somente em tempo de execução eles serão conhecidos.



```

spock2f@raffael: ~/Desktop/com vliw
File Edit View Terminal Help
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(4) : Erros possiveis (RAW/WAR/WAW) !!

Linha [2] : add00,0,$r0,$r1
Linha [3] : add01,0,$r0,$r2

spock2f@raffael:~/Desktop/com vliw$


```

*Figura 4.46: Validando a saída das instruções R (Imediato) – R (Indireto)*

#### ❖ Formato R (Imediato) – Formato R (Direto)

Neste caso podem acontecer RAW e WAW como mostrado nas figuras abaixo:

RAW: Registrador \$t0 é escrito na linha 2 e lido na linha 3



```

spock2f@raffael: ~/Desktop/com vliw
File Edit View Terminal Help
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(1) : Erro do tipo RAW !!

Linha [2] : add10,0,1,2
Linha [3] : add00,0,$r0,$t0

spock2f@raffael:~/Desktop/com vliw$

```

*Figura 4.47: Validando a dependência do tipo RAW em R (Imediato) – R (Direto)*

WAR: Como pode ser verificado, não acontece WAR em Imediato - Direto




```

spock2f@raffael: ~/Desktop/com vliw
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm
spock2f@raffael:~/Desktop/com vliw$

```

*Figura 4.48: Validando que não existe dependência do tipo WAR em R (Imediato) – R (Direto)*

WAW: Registrador \$t0 é escrito tanto na linha 2 como na linha 3



```

spock2f@raffael: ~/Desktop/com vliw
File Edit View Terminal Help
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(3) : Erro do tipo WAW !!

Linha [2] : add10,0,1,2
Linha [3] : add00,0,$t0,$r1

spock2f@raffael:~/Desktop/com vliw$

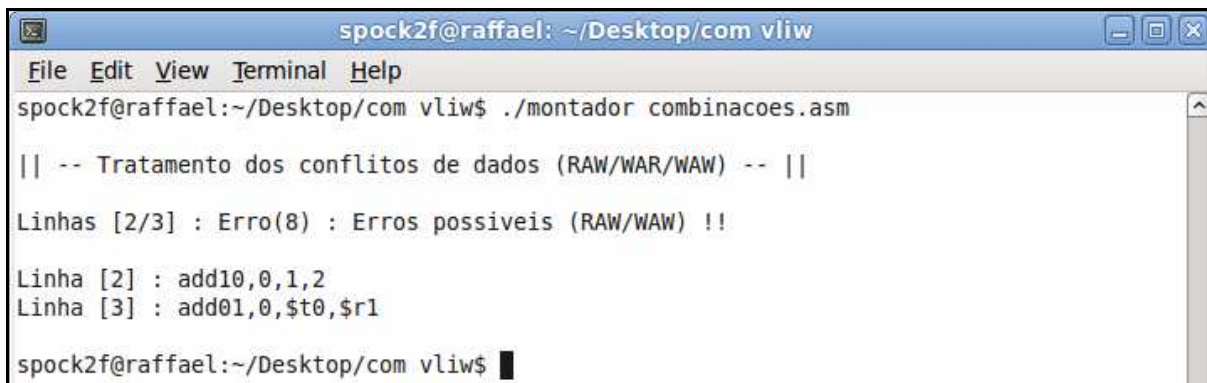
```

*Figura 4.49: Validando que não existe dependência do tipo WAR em R (Imediato) – R (Direto)*

#### ❖ Formato R (Indireto) – Formato R (Imediato)

Nesse caso não é possível saber se existe dependência WAR ou WAW em tempo de montagem pois não é possível saber para quais registradores apontam o conteúdo dos registradores do campo da instrução.





```

spock2f@raffael: ~/Desktop/com vliw
File Edit View Terminal Help
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(8) : Erros possiveis (RAW/WAW) !!

Linha [2] : add10,0,1,2
Linha [3] : add01,0,$t0,$r1

spock2f@raffael:~/Desktop/com vliw$

```

*Figura 4.50: Validando que existem dependências do tipo RAW e WAW nos formatos R(Indireto) – R(Imediato)*

WAR: Não acontece esta dependência



```

spock2f@raffael: ~/Desktop/com vliw
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm
spock2f@raffael:~/Desktop/com vliw$

```

*Figura 4.51: Validando que não existem dependências do tipo WAR nas instruções do tipo R(Indireto) – R(Imediato)*

## 2. Formato R – Formato I

### ❖ Formato R (Direto) – Formato I

Neste caso existem as dependências do tipo WAR e WAW como mostrado abaixo:

RAW : Não acontece esta dependência



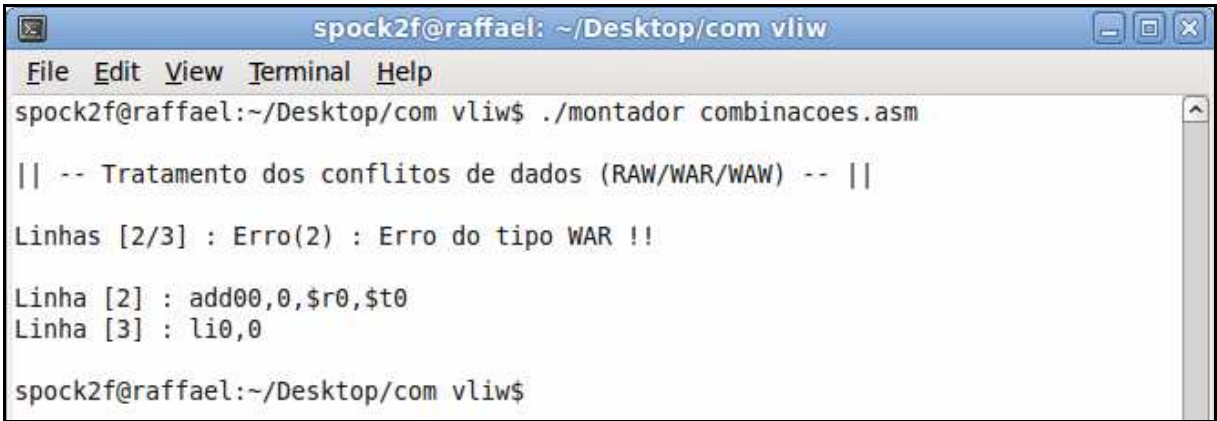
```

spock2f@raffael: ~/Desktop/com vliw
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm
spock2f@raffael:~/Desktop/com vliw$

```

*Figura 4.52: Validando a dependência do tipo RAW das instruções R (Direto) – I*

WAR: O registrador \$t0 é lido na linha 2 e escrito na linha 3



```

spock2f@raffael: ~/Desktop/com vliw
File Edit View Terminal Help
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(2) : Erro do tipo WAR !!

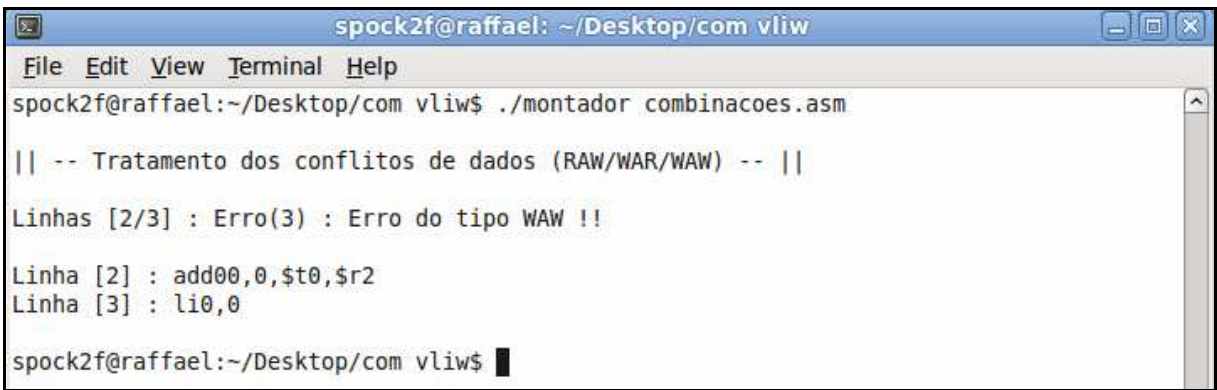
Linha [2] : add00,0,$r0,$t0
Linha [3] : li0,0

spock2f@raffael:~/Desktop/com vliw$

```

*Figura 4.53: Validando a dependência do tipo WAR das instruções R (Direto) – I*

WAW: O registrador \$t0 é escrito na linha 2 e escrito na linha 3



```

spock2f@raffael: ~/Desktop/com vliw
File Edit View Terminal Help
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(3) : Erro do tipo WAW !!

Linha [2] : add00,0,$t0,$r2
Linha [3] : li0,0

spock2f@raffael:~/Desktop/com vliw$ █

```

*Figura 4.54: Validando a dependência do tipo WAW das instruções R (Direto) – I*

❖ Formato R (Indireto) – Formato I

RAW: Não existe dependência



```

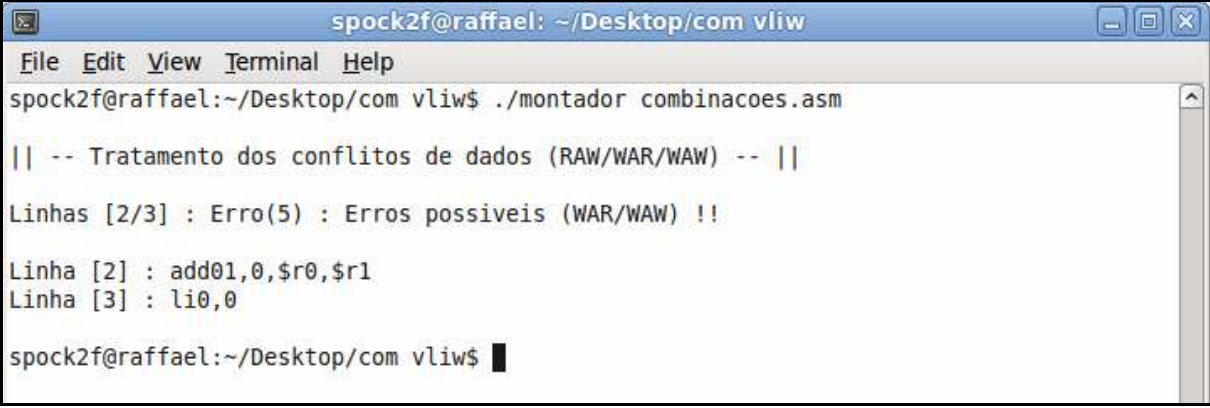
spock2f@raffael: ~/Desktop/com vliw
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm
spock2f@raffael:~/Desktop/com vliw$ █

```

*Figura 4.55: Validando que não existem dependências do tipo RAW nas instruções do tipo R(Indireto) – I*



*WAR/WAW*: Nesse caso não é possível saber se existe dependência WAR ou WAW em tempo de montagem pois não é possível saber para quais registradores apontam o conteúdo dos registradores do campo da instrução.



```

spock2f@raffael: ~/Desktop/com vliw
File Edit View Terminal Help
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(5) : Erros possiveis (WAR/WAW) !!

Linha [2] : add01,0,$r0,$r1
Linha [3] : li0,0

spock2f@raffael:~/Desktop/com vliw$

```

*Figura 4.56: Validando as dependências do tipo WAR/WAW das instruções R (Indireto) – I*

#### ❖ Formato R (Imediato) – Formato I

Neste caso existe dependência do tipo WAW como mostrado abaixo:

RAW : Não existem dependência



```

spock2f@raffael: ~/Desktop/com vliw
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm
spock2f@raffael:~/Desktop/com vliw$

```

*Figura 4.57: Validando que não existem dependências do tipo RAW nas instruções do tipo R(Imediato) – I*

WAR : Não existem dependência



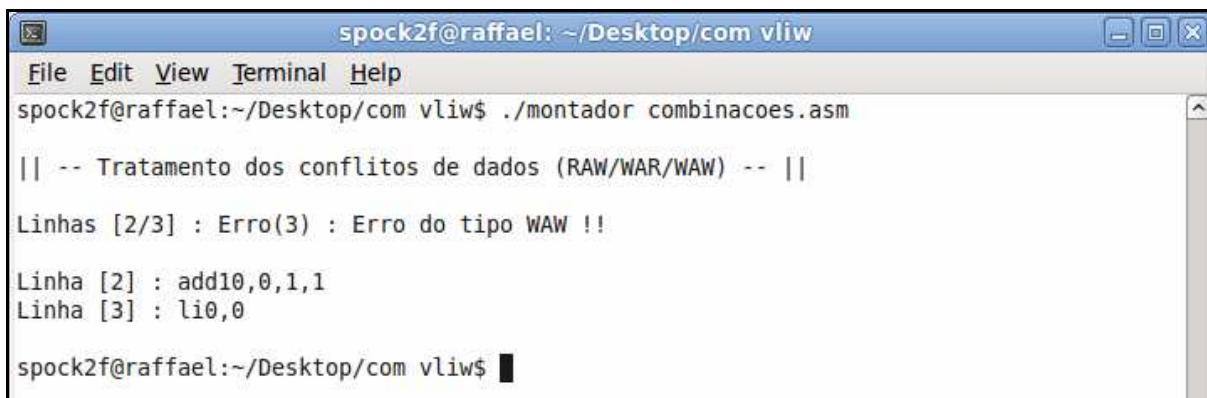
```

spock2f@raffael: ~/Desktop/com vliw
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm
spock2f@raffael:~/Desktop/com vliw$

```

*Figura 4.58: Validando que não existem dependências do tipo WAR nas instruções do tipo R(Imediato) – I*

WAW: Escrita do registrador \$t0 na linha 2 e escrita do registrador \$t0 na linha 3



```

spock2f@raffael: ~/Desktop/com vliw
File Edit View Terminal Help
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(3) : Erro do tipo WAW !!

Linha [2] : add10,0,1,1
Linha [3] : li0,0

spock2f@raffael:~/Desktop/com vliw$

```

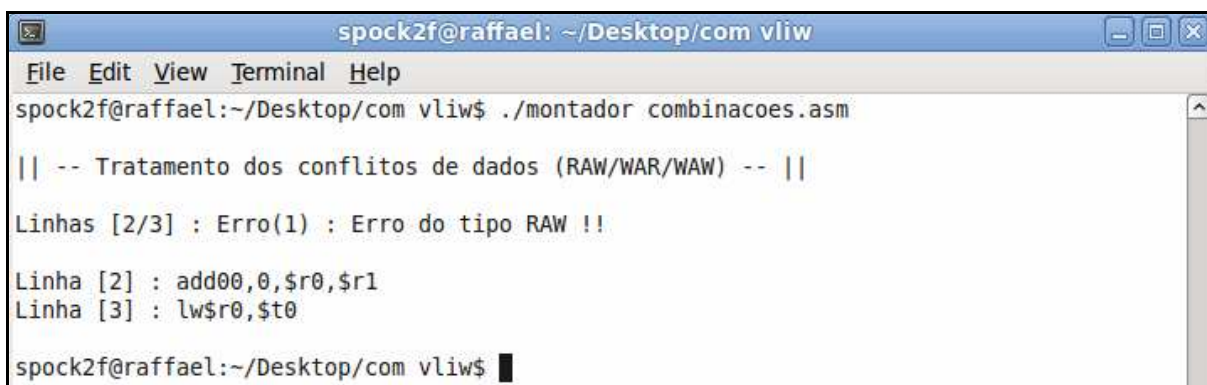
*Figura 4.59: Validando a dependência do tipo WAW das instruções R (Imediato) – I*

### 3. Formato R – Formato Load/Store

#### ❖ Formato R (Direto) – Formato Load/Store

Neste caso podem ocorrer as dependências do tipo RAW, WAR e WAW como mostrado abaixo:

RAW: Escrita do registrador \$r0 na linha 2 e leitura do registrador \$r0 na linha 3



```

spock2f@raffael: ~/Desktop/com vliw
File Edit View Terminal Help
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(1) : Erro do tipo RAW !!

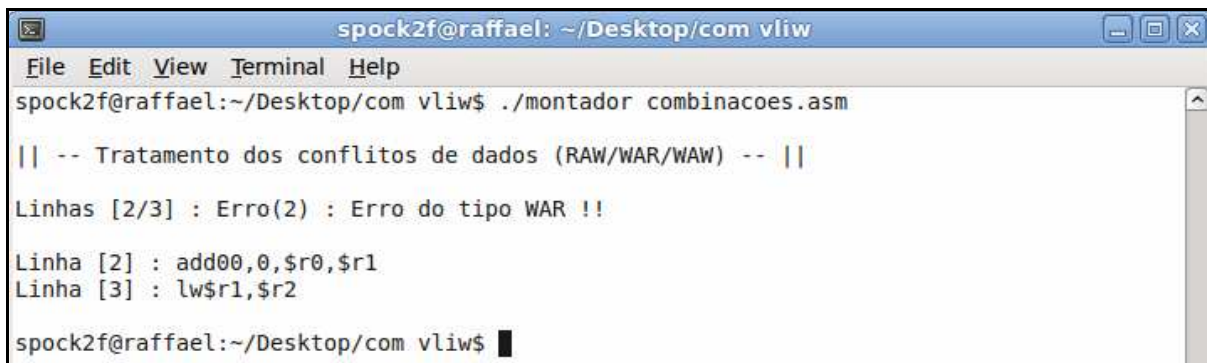
Linha [2] : add00,0,$r0,$r1
Linha [3] : lw$r0,$t0

spock2f@raffael:~/Desktop/com vliw$

```

*Figura 4.60: Validando a dependência do tipo RAW das instruções R (Direto) – Load/Store*

WAR: Leitura no registrador \$r1 na linha 2 e escrita no registrador \$r1 na linha 3



```

spock2f@raffael: ~/Desktop/com vliw
File Edit View Terminal Help
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(2) : Erro do tipo WAR !!

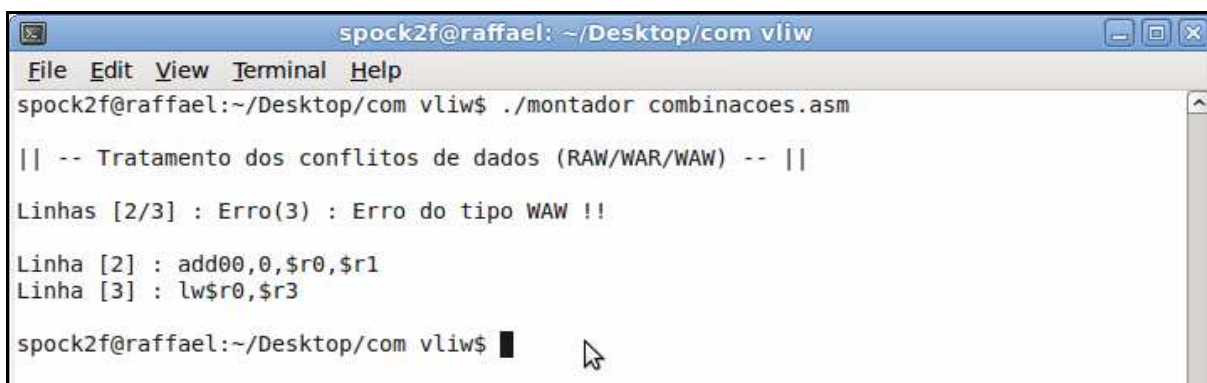
Linha [2] : add00,0,$r0,$r1
Linha [3] : lw$r1,$r2

spock2f@raffael:~/Desktop/com vliw$

```

Figura 4.61: Validando a dependência do tipo WAR das instruções R (Direto) – Load/Store

WAW: Escrita no Registrador \$r0 na linha 2 e na linha 3



```

spock2f@raffael: ~/Desktop/com vliw
File Edit View Terminal Help
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(3) : Erro do tipo WAW !!

Linha [2] : add00,0,$r0,$r1
Linha [3] : lw$r0,$r3

spock2f@raffael:~/Desktop/com vliw$

```

Figura 4.62: Validando a dependência do tipo WAW das instruções R (Direto) – Load/Store

#### ❖ Formato R (Indireto) – Formato Load/Store

Nesse caso não é possível saber se existe dependência ou não em tempo de montagem pois não é possível saber para quais registradores apontam o conteúdo dos registradores do campo da instrução.



```

spock2f@raffael: ~/Desktop/com vliw
File Edit View Terminal Help
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(4) : Erros possiveis (RAW/WAR/WAW) !!

Linha [2] : add01,0,$r0,$r1
Linha [3] : lw$r0,$t0

spock2f@raffael:~/Desktop/com vliw$

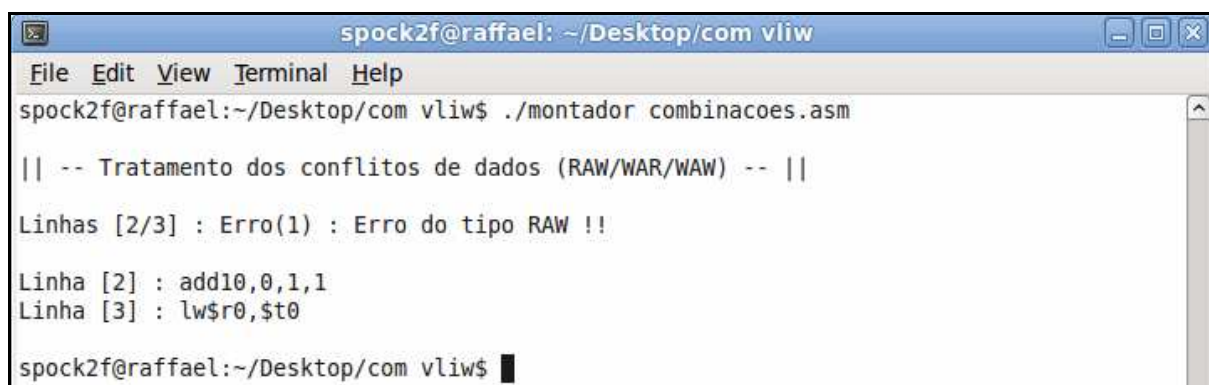
```

Figura 4.63: Validando a dependência do tipo RAW/WAR/WAW das instruções R (Indireto) – Load/Store

❖ Formato R (Imediato) – Formato Load/Store

Neste caso pode acontecer dependência do tipo RAW e WAW como mostrado abaixo:

RAW: Registrador \$t0 é escrito na linha 2 e lido na linha 3



```
spock2f@raffael: ~/Desktop/com vliw
File Edit View Terminal Help
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(1) : Erro do tipo RAW !!

Linha [2] : addi0,0,1,1
Linha [3] : lw$r0,$t0

spock2f@raffael:~/Desktop/com vliw$
```

*Figura 4.64: Validando a dependência do tipo RAW das instruções R (Imediato) – Load/Store*

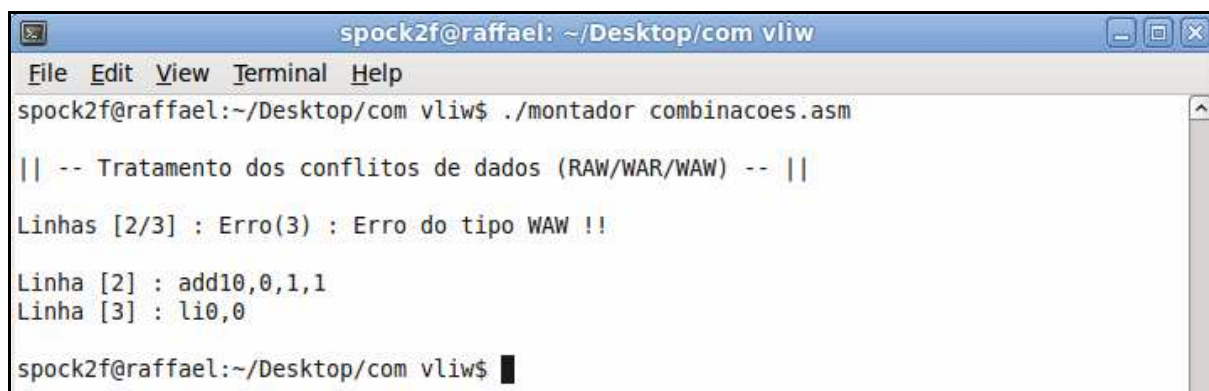
WAR: Não existe dependência



```
spock2f@raffael: ~/Desktop/com vliw
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm
spock2f@raffael:~/Desktop/com vliw$
```

*Figura 4.65: Validando que não existe dependência do tipo WAR nas instruções R (Imediato) – Load/Store*

WAW: Registrador \$t0 sendo escrito na linha 2 e na linha 3



```
spock2f@raffael: ~/Desktop/com vliw
File Edit View Terminal Help
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(3) : Erro do tipo WAW !!

Linha [2] : addi0,0,1,1
Linha [3] : li0,0

spock2f@raffael:~/Desktop/com vliw$
```

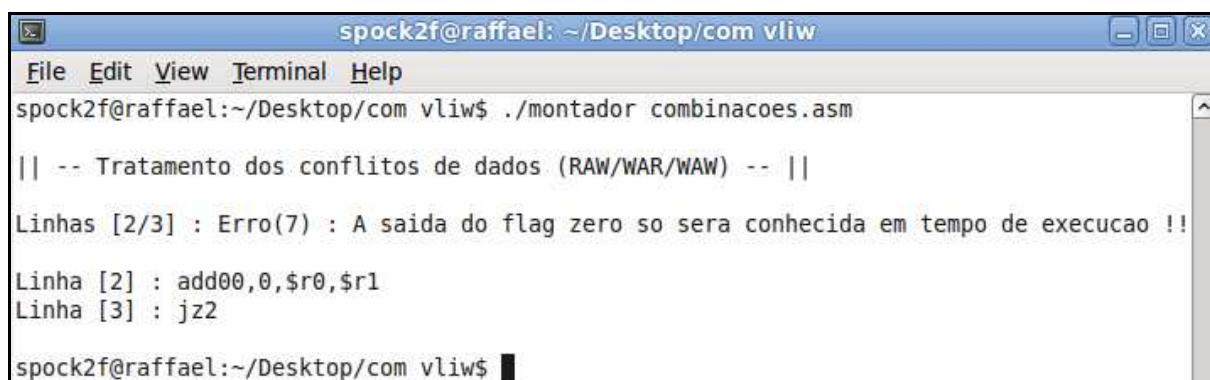
*Figura 4.66: Validando a dependência do tipo WAW das instruções R (Imediato) – Load/Store*

#### 4. Formato R – Formato Branch

##### ❖ Formato R (Direto/Indireto/Imediato) /Formato Branch

Neste caso existe dependência do resultado do flag de zero/carry/overflow da instrução o que em outras palavras quer dizer que as instruções não poderão executar ao mesmo tempo, devendo ser colocado um “nop” entre as instruções. Abaixo é demonstrado as saídas para o modo de endereçamento direto/indireto/imediato da instrução de add.

##### ✓ Formato R(Direto) – Branch



```

spock2f@raffael: ~/Desktop/com vliw
File Edit View Terminal Help
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(7) : A saida do flag zero so sera conhecida em tempo de execucao !!

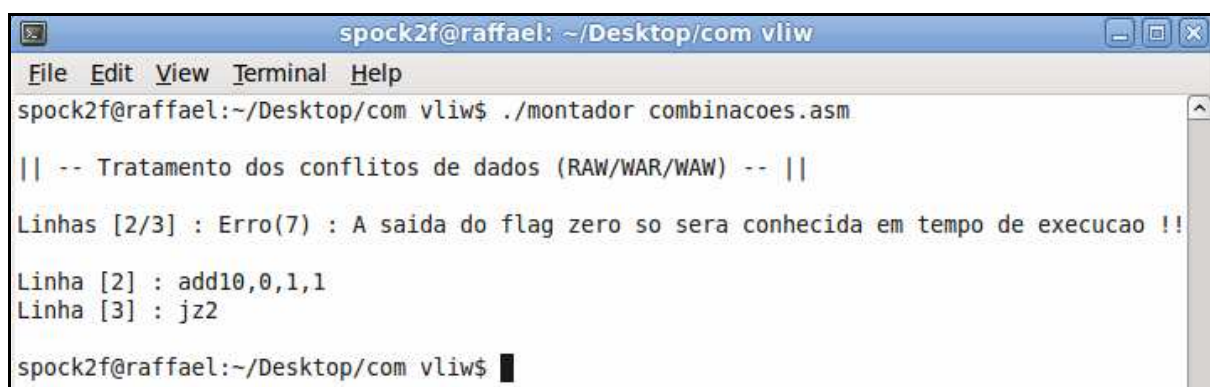
Linha [2] : add00,0,$r0,$r1
Linha [3] : jz2

spock2f@raffael:~/Desktop/com vliw$

```

*Figura 4.67: Validando a dependência de controle das instruções R (Direto) – Branch*

##### ✓ Formato R(Imediato) – Branch



```

spock2f@raffael: ~/Desktop/com vliw
File Edit View Terminal Help
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(7) : A saida do flag zero so sera conhecida em tempo de execucao !!

Linha [2] : add10,0,1,1
Linha [3] : jz2

spock2f@raffael:~/Desktop/com vliw$

```

*Figura 4.68: Validando a dependência de controle das instruções R (Imediato) – Branch*

##### ✓ Formato R(Indireto) – Branch



```

spock2f@raffael: ~/Desktop/com vliw
File Edit View Terminal Help
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(7) : A saida do flag zero so sera conhecida em tempo de execucao !!

Linha [2] : add01,0,$r0,$r1
Linha [3] : jz2

spock2f@raffael:~/Desktop/com vliw$

```

Figura 4.69: Validando a dependência de controle das instruções R (Indireto) - Branch

## 5. Formato I – Formato I

Neste caso pode acontecer dependência do tipo WAW como mostrado abaixo :

RAW: Não existe dependência

```

spock2f@raffael: ~/Desktop/com vliw
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm
spock2f@raffael:~/Desktop/com vliw$

```

Figura 4.70: Validando que não existem dependências RAW entre instruções I – I

WAW: Não existe dependência

```

spock2f@raffael: ~/Desktop/com vliw
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm
spock2f@raffael:~/Desktop/com vliw$

```

Figura 4.71: Validando que não existem dependências WAR entre instruções I – I

WAW: Registrador \$t0 é escrito na linha 2 e na linha 3

```

spock2f@raffael: ~/Desktop/com vliw
File Edit View Terminal Help
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(3) : Erro do tipo WAW !!

Linha [2] : li0,0
Linha [3] : li0,0

spock2f@raffael:~/Desktop/com vliw$

```

Figura 4.72: Validando a dependência do tipo WAW entre instruções I - I

## 6. Formato I – Formato Load/Store

Neste caso pode acontecer RAW e WAW como mostrado nas figuras abaixo :

RAW: Registrador \$t0 é escrito na linha 2 e lido na linha 3



```
spock2f@raffael: ~/Desktop/com vliw
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(1) : Erro do tipo RAW !!

Linha [2] : li0,0
Linha [3] : lw$r0,$t0

spock2f@raffael:~/Desktop/com vliw$
```

Figura 4.73: Validando a dependência do tipo RAW entre instruções I - Load/Store

WAR: Não acontece



```
spock2f@raffael: ~/Desktop/com vliw
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm
spock2f@raffael:~/Desktop/com vliw$
```

Figura 4.74: Validando que não existem dependências WAR entre instruções I – Load/Store

WAW: Registrador \$t0 é escrito na linha 2 e escrito na linha 3



```
spock2f@raffael: ~/Desktop/com vliw
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(3) : Erro do tipo WAW !!

Linha [2] : li0,0
Linha [3] : lw$t0,$r1

spock2f@raffael:~/Desktop/com vliw$
```

Figura 4.75: Validando a dependência do tipo WAW entre instruções I – Load/Store

## 7. Formato I – Formato Branch

Neste caso não acontecem dependências como mostrado abaixo:



```


spock2f@raffael: ~/Desktop/com vliw
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm
spock2f@raffael:~/Desktop/com vliw$
  
```

*Figura 4.76: Validando que não existem dependências entre instruções I - Branch*

## 8. Formato Load/Store – Formato Load/Store

Neste caso pode acontecer RAW , WAR e WAW como mostrado nas figuras abaixo :

RAW: Registrador \$r1 é escrito na linha 2 e lido na linha 3



```

spock2f@raffael: ~/Desktop/com vliw
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||


Linhas [2/3] : Erro(1) : Erro do tipo RAW !!

Linha [2] : lw$r1,$r2
Linha [3] : lw$r3,$r1

spock2f@raffael:~/Desktop/com vliw$
  
```

*Figura 4.77: Validando a dependência do tipo RAW entre instruções Load/Store-Load/Store*

WAR: Registrador \$r3 é lido na linha 2 e escrito na linha 3



```

spock2f@raffael: ~/Desktop/com vliw
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(2) : Erro do tipo WAR !!

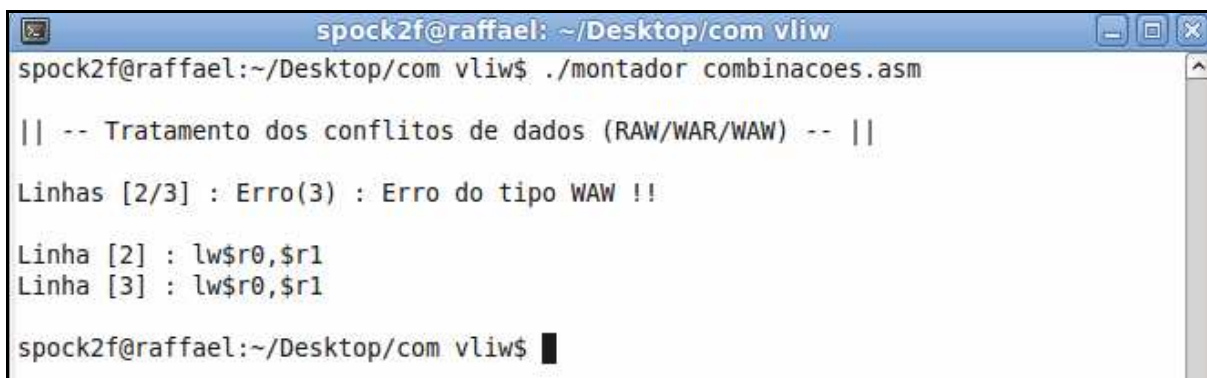
Linha [2] : lw$r3,$r2
Linha [3] : lw$r2,$r4

spock2f@raffael:~/Desktop/com vliw$
  
```

*Figura 4.78: Validando a dependência do tipo WAR entre instruções Load/Store-Load/Store*



WAW: Registrador \$r0 é escrito na linha 2 e na linha 3



```

spock2f@raffael: ~/Desktop/com vliw
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(3) : Erro do tipo WAW !!

Linha [2] : lw$r0,$r1
Linha [3] : lw$r0,$r1

spock2f@raffael:~/Desktop/com vliw$

```

Figura 4.79: Validando a dependência do tipo WAW entre instruções Load/Store–Load/Store

## 9. Formato Load/Store – Formato Branch

Neste caso não acontecem dependências como mostrado abaixo :



```

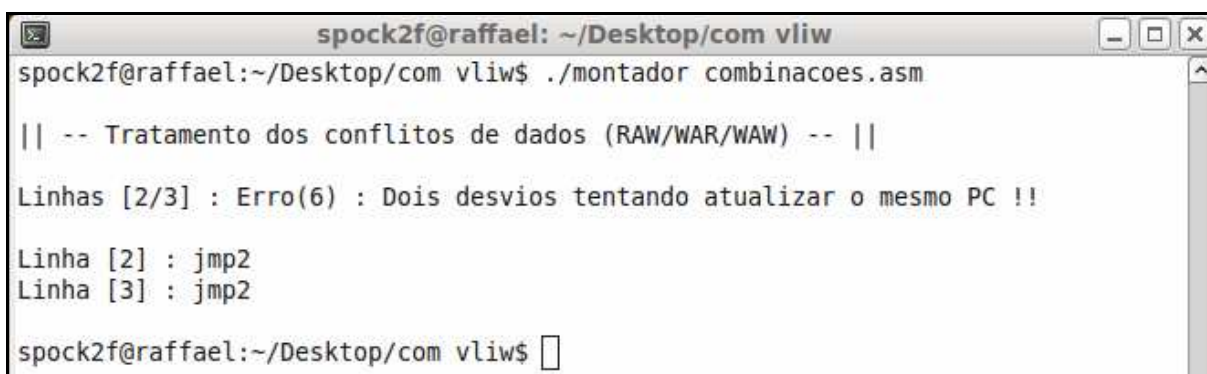
spock2f@raffael: ~/Desktop/com vliw
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm
spock2f@raffael:~/Desktop/com vliw$

```

Figura 4.80: Validando que não existem dependências entre instruções Load/Store - Branch

## 10. Formato Branch – Formato Branch

Neste tipo de contexto quando existirem duas instruções de Branch ira existir um conflito de controle pelo contador de programa, devendo ser informado pelo montador ao usuário.



```

spock2f@raffael: ~/Desktop/com vliw
spock2f@raffael:~/Desktop/com vliw$ ./montador combinacoes.asm

|| -- Tratamento dos conflitos de dados (RAW/WAR/WAW) -- ||

Linhas [2/3] : Erro(6) : Dois desvios tentando atualizar o mesmo PC !!

Linha [2] : jmp2
Linha [3] : jmp2

spock2f@raffael:~/Desktop/com vliw$

```

Figura 4.81: Validando a dependência de controle entre as instruções Branch - Branch

#### 4.4 Validação do ISS VLIW para o ISA do AIDA-16

Como o tratamento das dependências é feito em tempo de montagem, todo código binário que será carregado no ISS é livre de qualquer tipo de dependência entre as instruções. A implementação do ISS VLIW contou com a duplicação das estruturas de busca, decodificação e de execução a fim de contemplar o conceito da organização da máquina VLIW. Como as estruturas responsáveis pela busca, decodificação e execução já foram validadas na etapa anterior não é necessário efetuar novamente o processo de validação do ISS VLIW. A fim de demonstrar em nível de código de alto nível as estruturas que implementam o suporte ao VLIW, abaixo é demonstrando os blocos de busca, decodificação e execução VLIW :

```
int BUSCA_INSTRUCAO() /* Funcao responsavel por buscar as instrucoes */
{
    instracao0 = MEM_INST[PC]; /* Busca a primeira molecula (Instrucao 0 + Instrucao 1) */
    instracao1 = MEM_INST[(PC+1)];
    if(instracao0 == 61440) /* Verifica se a instrucao halt esta na Instrucao 0 - Condicao para parar */
    {
        return(1);
    }
    else if(instracao1 == 61440)
    {
        PC = PC+1; /* Verifica se a instrucao halt esta na Instrucao 1 - Ainda falta 1 instrucao */
        return(2);
    }
    PC=PC+2; /* Incrementa o PC em duas unidades */
    return(0);
}
```

*Figura 4.82: Bloco de busca de instrução VLIW*

```
int DECODIFICA_INSTRUCAO() /* Funcao responsavel por decodificar as instrucoes lidas */
{
    if(flag == 2) /* Responsavel por decodificar somente uma instrucao (Quando a segunda e halt) */
    {
        inst0 = Decodifica(instrucao0);
    }
    else /* Responsavel por decodificar duas instrucoes */
    {
        inst0 = Decodifica(instrucao0);
        inst1 = Decodifica(instrucao1);
    }
}
```

*Figura 4.83: Bloco de decodificação de instrução VLIW*

```
int EXECUTA() /* Bloco de execucao VLIW */
{
    if(flag == 2) /* Executa somente uma instrucao (Quando a outra e halt) */
    {
        Ula(inst0,1);
    }
    else /* Executa duas instrucoes */
    {
        Ula(inst0,1);
        Ula(inst1,2);
    }
}
```

*Figura 4.84: Bloco de execução de instrução VLIW*

O Apêndice A contempla o algoritmo de ordenação bubble sort, responsável por ordenar um vetor de bytes na memória de dados que esta em ordem decrescente em ordem crescente, e que serviu de exemplo pratico para validar todos os conceitos envolvidos no trabalho, passando pelo processo de montagem e simulação, utilizando tanto o montador como o ISS VLIW.

## 5 CONCLUSÃO E CONTRIBUIÇÕES

O presente trabalho apresentou a implementação de uma infra-estrutura para processamento em nível de simulação funcional não temporizada de um modelo de arquitetura baseado no paradigma VLIW para o processador AIDA-16. Para suportar esta execução foram desenvolvidas duas ferramentas fundamentais: (i) montador, que realiza a verificação léxica, sintática do código de montagem e alguns aspectos semânticos, além de identificar as dependências que possam existir entre instruções de forma a garantir a geração de código de máquina verificado quanto à execução VLIW; (ii) simulador do conjunto de instruções, que realiza a simulação funcional das instruções do código de máquina geradas pelo montador de forma não temporizada, além de permitir a busca e execução paralela destas instruções segundo o paradigma VLIW.

Além da infra-estrutura VLIW para AIDA-16, este trabalho disponibilizou também uma segunda contribuição: a implementação da mesma infra-estrutura citada anteriormente porém sem considerar a abordagem VLIW. Esta versão foi criada pensando em permitir que código normal, sem abordagens paralisáveis, pudesse ser executada como suporte ao aprendizado acadêmico de arquitetura de computadores e disciplinas correlatas tais como sistemas digitais. Com esta versão, alunos podem desenvolver código AIDA-16 para arquitetura convencional, realizar a montagem destes códigos e por fim executá-los e observar seu comportamento funcional.

O processo de montagem auxilia o desenvolvedor tanto para a arquitetura convencional como VLIW, apresentando os locais onde ocorreram erros sintáticos e léxicos, incluindo alguns erros semânticos, possíveis de serem tratados. O resultado final é a geração do código de máquina correto do ponto de vista da execução convencional ou VLIW, conforme a abordagem escolhida. Por outro lado, o simulador do conjunto de instruções (ISS) disponibilizou o ambiente necessário para que a funcionalidade das instruções pudesse ser observada, da mesma forma considerando abordagem convencional ou VLIW.

Para que a geração e execução do código de máquina fossem validadas, foi necessário gerar uma extensa quantidade de códigos para teste, de forma a tentar cobrir todas as possibilidades de configuração e combinações da ISA do AIDA-16. A geração dos programas contendo estes códigos de teste apresentou uma etapa desafiadora, uma vez que a ISA possui muitas variações possíveis, portanto gerando uma grande quantidade de combinações a serem consideradas. Os programas de teste foram gerados para atender tanto a arquitetura convencional como a VLIW permitindo testar o montador e o simulador.

Conclui-se com base no apêndice A que mesmo existindo vantagens em desenvolver uma organização de processador AIDA-16 VLIW como constado neste trabalho, devem existir técnicas mais avançadas, capazes de remover as dependências existentes entre as instruções, a fim de remover o overhead causado no aumento do código com a inserção de "NOPs" para tratar as dependências entre as instruções, que em pratica faz com que não compense aumentar o numero de unidades capazes de buscar, decodificar e executar as instruções, e também aumentar a complexidade do montador.

Fora os aspectos de desempenho, o propósito fundamental do trabalho de prover as ferramentas de ensino se consagrou verdadeiro e necessário, pois mesmo sendo realizado um pré estudo superficial do conjunto de instruções do processador em tempo de desenvolvimento dos produtos deste trabalho, os conceitos chave contidos em diversas instruções só se mostraram importantes e realmente foram entendidos no momento em que o autor deste trabalho desenvolvia códigos práticos para validar o funcionamento das ferramentas de ensino, e com isso foi possível entender a importância de se ter algo capaz de demonstrar na pratica a aplicabilidade dos conceitos teóricos aprendidos em sala de aula.

## 5.1 Trabalhos futuros

A seguir são descritas algumas sugestões para a realização de trabalhos futuros baseados nos resultados deste trabalho.

- ❖ Técnicas VLIW, como *looping unrolling*, fazem uso do código fonte de alto nível de abstração, como código em linguagem C, para obter maior grau de paralelismo ao executar instruções. Sugere-se considerar o desenvolvimento de trabalhos que contemplem as técnicas VLIW em nível de compilação, gerando uma coleção de ferramentas de compilação para AIDA-16 baseadas nas ferramentas binutils do gcc.

Desta forma poderia ser possível escrever código em linguagem C e, a partir deste, gerar o código de máquina correspondente à ISA AIDA-16. Como esta etapa pode ser muita ampla, complexa e demorada sugere-se que seja dividida em súbitas menores.

- ❖ Outra proposta de trabalho futuro seria o desenvolvimento da versão de arquitetura VLIW semelhante à proposta neste trabalho porém temporizada, ou seja, considerando ciclos de relógio na sua execução, possivelmente implementada em linguagem SystemC, disponibilizando uma infra-estrutura mais próxima à implementação RTL. Complementarmente uma versão VLIW em nível RTL também é considerada uma boa sugestão de trabalho futura, possivelmente descrita em linguagem VHDL.
- ❖ A partir das ferramentas de compilação C, seria possível buscar outras técnicas para execução VLIW, tornando possível ampliar o nível de paralelismo e permitir o estudo de novas técnicas de paralelismo em nível de instrução.

## REFERÊNCIAS BIBLIOGRÁFICAS

[CAR01] CARRO, Luigi. "Projeto e Prototipação de Sistemas Digitais". Porto Alegre: Instituto de Informática da UFRGS: Editora da UFRGS, 2001.

[FIS05] FISHER, Joseph A.; Faraboschi P.; Young C. "Embedded Computing." : Editora Morgan Kaufmann Publisher, 2005.

[HEN03] HENNESSY, J. L.; Patterson, A. "Arquitetura de Computadores uma Abordagem Quantitativa". Editora Campus, 2003.

[ITO98] ITO, Sérgio A. Arquiteturas VLIW : Uma alternativa para exploração de paralelismo a nível de instrução 1998.

[MOT05] MOTTA, Rodrigo B. AIDA-16: um microprocessador destinado ao ensino utilizando hardware programável.

[MIR09] MIREK, Lukasz. <http://www.design-reuse.com/articles/21745/interpretive-instruction-set-simulator.html>, 2009 Disponível em. Acessada dia 25 de novembro de 2009.

[PAT00] HENNESSY, J. L.; Patterson, D. A. "Interface hardware/software". Editora LTC, 2000.

[SHE09] SCHEMMER, Raffael B. Especificação de um processador e de um montador VLIW baseados no ISA da plataforma AIDA.

[TAN99] TANENBAUM, Andrew S. Structuerd Computer Organization. Prentice Hall, 1999.

[WAG99] WAGNER, Flávio R. Ambiente para cosimulação SIMOO-VSS Synopsys, 1999.

[WEB01] WEBER, Raul F. "Fundamentos de Arquitetura de Computadores." Porto Alegre: Instituto de Informática da UFRGS: Editora Sagra Luzzatto, 2001.

## APENDICE A – EXEMPLO DE VALIDAÇÃO DO ISS E DO MONTADOR VLIW

```
# Universidade Regional Integrada do Alto uruguai e das Missões
# Bubble Sort - Exemplo pratico de validação do ISS e do Assembler
# Disciplina :Trabalho de conclusão - 10 Semestre
# Professor : Mr.Carlos petry
# Aluno : Raffael Bottoli Schemmer
```

```
# Algoritmo em C traduzido para o ASM AIDA-16
# int vet[10] = {9,8,7,6,5,4,3,2,1,0};
# int contador=0,contador2,tmp;
# do
# {
#     contador2=contador;
#     do
#     {
#         if(vet[contador] > vet[contador2])
#         {
#             tmp = vet[contador];
#             vet[contador] = vet[contador2];
#             vet[contador2] = tmp;
#         }
#         contador2++;
#     }while(contador2 < 10);
#     contador++;
# }while(contador < 10);
```

```
# Comentários do bubble.asm
# $r2 contem a constante 10
# $r11 representa a variável contador2
# $r10 representa a variável contador
# $t0 e $t1 são registradores temporários de troca
# $r0 recebe sempre vet[contador]
# $r1 recebe sempre vet[contador2]
# $r3 representa a variável tmp
# jz -10 implementa a função do while(contador2 < 10)
# jz -14 implementa a função do while(contador < 10)
# jz 2 implementa a função do if
```

```
.text
li 0,10
nop
li 1,0
```



```
mov $t0,$r2
nop
li 0,0
mov $t0,$r10
mov $t0,$r11
nop
lb $r0,$r10
nop
lb $r1,$r11
nop
mov $r0,$r3
div 00,0,$r3,$r1
nop
jz 2
nop
mov $r0,$r5
sb $r10,$r1
sb $r11,$r5
nop
inc 00,0,$r10
nop
mov $r10,$r4
nop
div 00,0,$r4,$r2
jz -10
nop
inc 00,0,$r11
mov $r11,$r10
mov $r11,$r5
nop
div 00,0,$r5,$r2
jz -14
nop
halt
.data
.byte 9
.byte 8
.byte 7
.byte 6
.byte 5
.byte 4
.byte 3
.byte 2
.byte 1
.byte 0
```

**APENDICE B – CODIGO ASSEMBLY RESPONSAVEL POR VALIDAR AS  
INSTRUÇÕES LOGICAS E ARITMETICAS DO MODO DE ENDERECAMENTO  
DIRETO**

```
# Universidade Regional Integrada do Alto Uruguai e das Missões
# Exemplo de validação das instruções lógicas e aritméticas – am = Direto
# Disciplina :Trabalho de conclusão - 10 Semestre
# Professor : Mr.Carlos petry
# Aluno : Raffael Bottoli Schemmer
.text
li 0,0
li 1,1
lb $r0,$t0
lb $r1,$t1
sub 00,0,$r0,$r1
li 1,2
lb $r2,$t1
inc 00,0,$r2
div 00,0,$r0,$r2
li 1,3
lb $r3,$t1
div 00,0,$r0,$r3
add 00,0,$r0,$r3
li 1,2
mod 00,0,$r0,$t1
li 0,2
xor 00,0,$r0,$t0
shl 00,0,$r0
shr 00,0,$r0
ror 00,0,$r0
rol 00,0,$r0
li 0,0
and 00,0,$r0,$t0
not 00,0,$r0
not 00,0,$r0
li 0,1
or 00,0,$r0,$t0
halt
.data
.byte 11
.byte 1
.byte 4
.byte 1
```

## APENDICE C – TABELA REVISADA DO CONJUNTO DE INSTRUÇÕES DO PROCESSADOR AIDA-16

Tipo	Cód	Instr	Exemplo	Significado	Flag	Descrição
NA	00000	nop	nop	NA	NA	Não operar
NA	11110	halt	halt	NA	NA	Para a execução do ISS
Aritm	00001	add	add 00,0,\$r0,\$r1	$\$r0 = \$r0 + \$r1$	z*,c*,o*	Soma dois valores
Aritm	00010	sub	sub 00,0,\$r0,\$r1	$\$r0 = \$r0 - \$r1$	z,c,o	Subtrai dois valores
Aritm	00011	inc	inc 00,0,\$r0	$\$r0 = \$r0 + 1$	z,c,o	Incrementa um valor
Aritm	00100	dec	dec 00,0,\$r0	$\$r0 = \$r0 - 1$	z,c,o	Decrementa um valor
Aritm	00101	mul	mul 00,0,\$r0,\$r1	$\$r0 = \$r0 * \$r1$	z,c,o	Multiplica dois valores
Aritm	00110	div	div 00,0,\$r0,\$r1	$\$r0 = \$r0 / \$r1$	z,c,o	Divide dois valores
Aritm	00111	mod	mod 00,0,\$r0,\$r1	$\$r0 = \$r0 \text{ mod } \$r1$	z,c,o	Calcula o resto da divisão
Lógica	01000	shr	shr 00,0,\$r0	$\$r0 = \text{shr}(\$r0)$	z,c	Desloca 1 bit a direita
Lógica	01001	shl	shl 00,0,\$r0	$\$r0 = \text{shl}(\$r0)$	z,c	Desloca 1 bit a esquerda
Lógica	01010	ror	ror 00,0,\$r0	$\$r0 = \text{ror}(\$r0)$	z	Rota 1 bit a direita
Lógica	01011	rol	rol 00,0,\$r0	$\$r0 = \text{rol}(\$r0)$	z	Rota 1 bit a esquerda
Lógica	01100	or	or 00,0,\$r0	$\$r0 = \text{or}(\$r0)$	z	Executa OU sobre a instrução
Lógica	01101	and	and 00,0,\$r0	$\$r0 = \text{and}(\$r0)$	z	Executa AND sobre a instrução
Lógica	01110	xor	xor 00,0,\$r0	$\$r0 = \text{xor}(\$r0)$	z	Executa XOR sobre a instrução
Lógica	01111	not	not 00,0,\$r0	$\$r0 = \text{not}(\$r0)$	z	Executa NOT sobre a instrução
Tranf/Mem	10000	li	li 0,1	$\$t0 = 1$	NA	Carga no byte baixo do registrador
Tranf/Mem	10001	lui	lui 0,1	$\$t0 = 512$	NA	Carga no byte alto do registrador
Tranf/Mem	10010	lw	lw \$,r0,\$r1	$\$r0 = \text{Memória}[\$r1]$	NA	Carga de word da memória para o BR*
Tranf/Mem	10011	sw	sw \$,r0,\$r1	$\$r0 = \text{Memória}[\$r1]$	NA	Carga de Reg do BR para a memória
Tranf/Mem	10100	lb	lb \$,r0,\$r1	$\$r0 = \text{Memória}[\$r1]$	NA	Carga de byte da memória para o BR
Tranf/Mem	10101	sb	sb \$,r0,\$r1	$\$r0 = \text{Memória}[\$r1]$	NA	Carga de Reg do BR para a memória
Tranf/Mem	10110	mov	mov \$,r0,\$r1	$\$r1 = \$r0$	NA	Copia entre registradores
Desv/Condic	10111	juv	juv 1	se v = 1 PC = PC + 1*2	NA	Desvia se der overflow
Desv/Condic	11000	jnv	jnv 1	se v = 0 PC = PC + 1*2	NA	Desvia se não der overflow
Desv/Condic	11001	jz	jz 1	se z = 1 PC = PC + 1*2	NA	Desvia se der zero
Desv/Condic	11010	jnz	jnz 1	se z = 0 PC = PC + 1*2	NA	Desvia se não der zero
Desv/Condic	11011	jc	jc 1	se c = 1 PC = PC + 1*2	NA	Desvia se der carry
Desv/Condic	11100	jnc	jnc 1	se c = 0 PC = PC + 1*2	NA	Desvia se não der carry
Desv/InCond	11101	jmp	jmp 1	PC = PC + 1*2	NA	Desvia para o endereço na instrução

## **DESENVOLVIMENTO DE UMA INFRA-ESTRUTURA DE SUPORTE VLIW PARA O PROCESSADOR AIDA-16**

por

Raffael Bottoli Schemmer

Trabalho de Conclusão apresentado aos Professores(as):

---

Prof. M.Sc. Carlos Alberto Petry

---

Prof. M.Sc. Paulo Ricardo Betencourt

---

Prof. M.Sc. Alexandro Magno dos Santos Adário

Vista e permitida à impressão.

Santo Ângelo, 30/11/2008