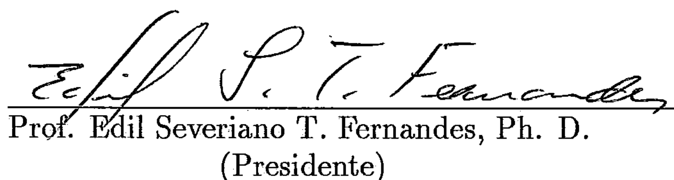
Avaliando os Parâmetros de uma Arquitetura VLIW

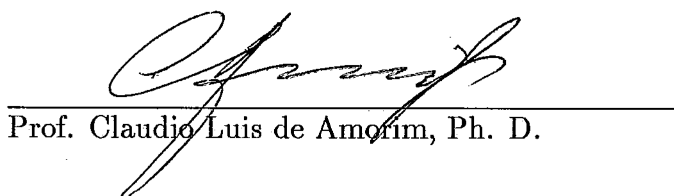
Alberto Ferreira de Souza

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

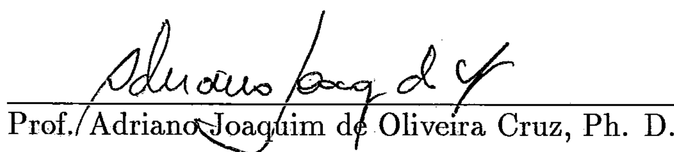
Aprovada por:



Prof. Edil Severiano T. Fernandes, Ph. D.
(Presidente)



Prof. Claudio Luis de Amorim, Ph. D.



Prof. Adriano Joaquim de Oliveira Cruz, Ph. D.

RIO DE JANEIRO, RJ - BRASIL
ABRIL DE 1993

SOUZA, ALBERTO FERREIRA DE

Avaliando os Parâmetros de uma Arquitetura VLIW [Rio de Janeiro]
1993

XIV, 94 p., 29.7 cm, (COPPE/UFRJ, M. Sc., ENGENHARIA DE SISTEMAS E COMPUTAÇÃO, 1993)

TESE – Universidade Federal do Rio de Janeiro, COPPE

1 – Máquinas VLIW 2 – Compactação de Código 3 – Arquiteturas Paralelas

I. COPPE/UFRJ II. Título (Série).

Ao Professor Edil

Agradecimentos

Ao Professor Edil, pelo seu paciente trabalho de orientação.

Ao pessoal do Laboratório e aos que já não estão mais lá, por toda a ajuda que deram.

Ao Arcileu e ao José Maria, pela compreensão na reta final.

Aos meus amigos, sempre presentes.

A todos aqueles que, de alguma forma, contribuiram para realização deste trabalho.

Resumo da Tese apresentada à COPPE como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M. Sc.)

Avaliando os Parâmetros de uma Arquitetura VLIW

Alberto Ferreira de Souza

Abril de 1993

Orientador: Edil Severiano Tavares Fernandes

Programa: Engenharia de Sistemas e Computação

Máquinas VLIW (*Very Long Instruction Word machines*) são arquiteturas paralelas que oferecem uma alternativa ao uso de Processadores Vetoriais e Multiprocessadores. Incorporando diversas unidades funcionais que podem operar em paralelo, essas máquinas podem ser caracterizadas pelas suas instruções, que incluem campos distintos para controlar diretamente cada um dos recursos do *hardware* subjacente. Elas são capazes de executar várias instruções procedentes de um mesmo programa de aplicação simultaneamente, interpretando uma instrução muito longa durante cada ciclo de processador.

Esse trabalho descreve os experimentos que permitiram avaliar o efeito de importantes parâmetros arquiteturais no volume do paralelismo de baixo nível que pode ser extraido de programas de aplicação. Através da interpretação do código objeto de um processador comercial, derivado de uma bateria de programas de teste, determinou-se a configuração ideal da máquina VLIW capaz de atingir a taxa de aceleração máxima de cada programa de teste.

Abstract of Thesis presented to COPPE as partial fulfillment of the requirements for the degree of Master of Science (M. Sc.)

A VLIW Architecture Parametrization

Alberto Ferreira de Souza

April, 1993

Thesis Supervisor: Edil Severiano Tavares Fernandes

Department: Programa de Engenharia de Sistemas e Computação

Very Long Instruction Word (VLIW) machines are highly parallel architectures that can be used as an alternative to Multiprocessors and Vector Processors. These machines incorporate a large number of functional units that can be activated in parallel and can be characterized by their instructions, which include separated fields exercising direct control to the resources of underlying hardware. They allow the concurrent execution of multiple machine instructions, proceeding from the same aplication program, through the interpretation of a very long instruction word during each processor cycle.

This work describes the experiments that have been carried out to assess the impact of important architectural parameters on the volume of low level parallelism that can be extracted from the application programs. By interpreting the actual object code derived from a suite of test programs, we have determined the ideal VLIW machine configuration leading to the maximum speedup that can be achieved by each test program.

Índice

I INTRODUÇÃO 1

II ARQUITETURAS PARALELAS E MÁQUINAS VLIW 4

II.1 Introdução 4

II.2 Arquiteturas SIMD 6

II.3 Arquiteturas MIMD 8

II.4 A Alternativa VLIW 9

II.5 Outras Arquiteturas Paralelas 12

III O AMBIENTE DE AVALIAÇÃO 14

III.1 Introdução 14

III.2 O Projeto do Experimento 15

III.3 O *Assembly* do Microprocessador i860 15

III.4 Os Programas de Teste 16

III.5 As Ferramentas de Avaliação 17

III.6 Técnica de Compactação Escolhida 19

IV COMPACTAÇÃO DE CÓDIGO 20

IV.1 Introdução 20

IV.2 Dependência de Dados 21

IV.3 Modelando a Máquina 23

IV.4 Dependência de Recursos 25

IV.5 O Algoritmo de Compactação 26

IV.6 As Características da Máquina VLIW 27

IV.7 Dependência de Dados e *Pipeline* 31

IV.8 Renomeação de Registradores 33

IV.9 Os Parâmetros do Modelo 33

IV.10 O Programa Compact 34

V EXPERIMENTOS E ANÁLISE DOS RESULTADOS 38

V.1 Introdução 38

V.2 Experimento 1: Livermore Loop 24 39

V.3 Experimento 2: Quick Sort 43

V.4 Experimento 3: Busca Binária 47

V.5 Experimento 4: Bubble Sort 51

V.6 Experimento 5: Decomposição LU 55

V.7 Experimento 6: Integracão Numérica 61

V.8 Análise dos Resultados 67

VI CONCLUSÃO 71

A Programas Exemplo 73

B Ferramentas de Avaliação **80**

B.1 O Sim860 80

B.1.1 Os Parâmetros de Entrada de Sim860 80

B.1.2 Usando o Sim860 81

B.1.3 Instruções Não Implementadas 84

B.2 Scode 84

B.2.1 Os Parâmetros de Entrada de Scode 84

B.3 Compact 85

B.3.1 Os Parâmetros de Entrada de Compact 85

B.4 O Formato dos Arquivos 86

Lista de Figuras

III.1 Modelo Básico de VLIW 14

IV.1 Máquina VLIW em Estudo 24

IV.2 Algorítmo FCFS 28

V.1 *Speedup* versus ULAI: LIVERMOR.C 40

V.2 *Speedup* versus BLRI: LIVERMOR.C 41

V.3 *Speedup* versus BERI: LIVERMOR.C 41

V.4 *Speedup* versus SE: LIVERMOR.C 41

V.5 *Speedup* versus UAM: LIVERMOR.C 42

V.6 *Speedup* versus RIPR: LIVERMOR.C 42

V.7 *Speedup* versus ULAI: QUICK.C 44

V.8 *Speedup* versus BLRI: QUICK.C 44

V.9 *Speedup* versus BERI: QUICK.C 45

V.10 *Speedup* versus SE: QUICK.C 45

V.11 *Speedup* versus UAM: QUICK.C 45

V.12 *Speedup* versus RIPR: QUICK.C 46

V.13 *Speedup* versus ULAI: BINARIA.C 48

V.14 *Speedup* versus BLRI: BINARIA.C 48

V.15 *Speedup* versus BERI: BINARIA.C 49

V.16 *Speedup* versus SE: BINARIA.C 49

V.17 *Speedup* versus UAM: BINARIA.C 49

V.18 *Speedup* versus RIPR: BINARIA.C 50

V.19 *Speedup* versus ULAI: BOLHA.C 52

V.20 *Speedup* versus BLRI: BOLHA.C 52

V.21 *Speedup* versus BERI: BOLHA.C 53

V.22 *Speedup* versus SE: BOLHA.C 53

V.23 *Speedup* versus UAM: BOLHA.C 53

V.24 *Speedup* versus RIPR: BOLHA.C 54

V.25 *Speedup* versus ULAI: LU.C 56

V.26 *Speedup* versus BLRI: LU.C 56

V.27 *Speedup* versus BERI: LU.C 57

V.28 *Speedup* versus SE: LU.C 57

V.29 *Speedup* versus UAM: LU.C 57

V.30 *Speedup* versus SPF: LU.C 58

V.31 *Speedup* versus MPF: LU.C 58

V.32 *Speedup* versus BLRPF: LU.C 58

V.33 *Speedup* versus BERPF: LU.C 59

V.34 *Speedup* versus BTER: LU.C 59

V.35 *Speedup* versus RIPR: LU.C 59

V.36 *Speedup* versus RPFPR: LU.C 60

V.37 *Speedup* versus ULAI: INTEGRAL.C 62

V.38 *Speedup* versus BLRI: INTEGRAL.C 62

V.39 *Speedup* versus BERI: INTEGRAL.C 63

V.40 *Speedup* versus SE: INTEGRAL.C 63

V.41 *Speedup* versus UAM: INTEGRAL.C 63

V.42 *Speedup* versus SPF: INTEGRAL.C 64

V.43 *Speedup* versus MPF: INTEGRAL.C 64

V.44 *Speedup* versus BLRPF: INTEGRAL.C 64

V.45 *Speedup* versus BERPF: INTEGRAL.C 65

V.46 *Speedup* versus BTER: INTEGRAL.C 65

V.47 *Speedup* versus RIPR: INTEGRAL.C 65

V.48 *Speedup* versus RPFPR: INTEGRAL.C 66

B.1 Tela do Programa Sim860 82

B.2 Formato do Registro do <*arquivo-scode*> 88

B.3 Arquivo de Parâmetros de Compact 89

Lista de Tabelas

IV.1 Latência das Unidades Funcionais da Máquina VLIW em Estudo . . . 32

IV.2 Informações Contidas no Arquivo de Saída do Programa Compact . . 35

V.1 Compactação com Recursos Ilimitados: LIVERMOR.C 39

V.2 Parâmetros Ideais para o Programa LIVERMOR.C 42

V.3 Compactação com Recursos Ilimitados: QUICK.C 44

V.4 Parâmetros Ideais para o Programa QUICK.C 46

V.5 Compactação com Recursos Ilimitados: BINARIA.C 48

V.6 Parâmetros Ideais para o Programa BINARIA.C 50

V.7 Compactação com Recursos Ilimitados: BOLHA.C 52

V.8 Parâmetros Ideais para o Programa BOLHA.C 54

V.9 Compactação com Recursos Ilimitados: LU.C 56

V.10 Parâmetros Ideais para o Programa LU.C 60

V.11 Compactação com Recursos Ilimitados: INTEGRAL.C 62

V.12 Parâmetros Ideais para o Programa INTEGRAL.C 66

V.13 Relação Entre o Valor de Cada Parâmetro e o Número de ULAIs . . . 68

V.14 *Speedup* da Máquina TRACE com cada programa de teste 69

V.15 Parâmetros da Máquina TRACE 70

@

Capítulo I

INTRODUÇÃO

As máquinas VLIW (*Very Long Instruction Word*) [21] formam uma classe de processadores caracterizados por possuir:

- um único fluxo de controle (um contador de programa e uma unidade de controle);
- uma palavra de instrução longa (*long instruction word*), contendo bits para controlar, direta e independentemente (durante cada ciclo de processador), cada unidade funcional da máquina;
- um grande número de caminhos de dados e de unidades funcionais, cujo controle é determinado em tempo de compilação (isto é, cuja ativação é determinada previamente). Máquinas VLIW não têm árbitros de barramento, filas ou outros mecanismos de sincronização por *hardware*.

Diferente dos Processadores Vetoriais, não é necessário uma grande regularidade no código do usuário para que se possa explorar o paralelismo disponível no *hardware* das VLIW. Diferente dos Multiprocessadores, não há penalidade imposta pela sincronização ou pela comunicação. Todas as unidades funcionais operam completamente sincronizadas e são controladas pelo compilador.

O uso de unidades funcionais *pipelined* e a ausência de *hardware* para gerência de conflitos, tornam as máquinas VLIW simples e rápidas. Uma unidade

funcional nunca precisa esperar pelo resultado de outra, o que permite que o *hardware* opere sempre na sua velocidade máxima.

Apesar de suas características extremamente promissoras, até recentemente não se acreditava ser possível atingir um *speedup* superior a 2 ou 3 com máquinas com este tipo de arquitetura. Experimentos haviam mostrado que devido a demasiada presença de desvios condicionais nos programas, o volume de paralelismo que poderia ser explorado com este tipo de máquina era muito pequeno [37].

A partir do desenvolvimento da técnica de compactação global de código, conhecida como *Trace Scheduling* [20], a limitação imposta pelos desvios condicionais foi superada. Hoje, outras técnicas vêm sendo desenvolvidas, ampliando o horizonte de desempenho das máquinas VLIW.

Neste trabalho são apresentados experimentos que foram realizados para determinar quais as dimensões que uma máquina VLIW deve ter, para executar uma classe de programas de aplicação com máximo desempenho. Para isso foram desenvolvidas três ferramentas que, juntas, formam um ambiente para determinar os parâmetros ideais de uma máquina VLIW.

A primeira destas ferramentas é um simulador do processador i860. Através desse simulador é possível gerar um *trace* da execução de um conjunto de programas de teste (*benchmark*). A segunda ferramenta é um programa capaz de produzir, a partir do código objeto de cada programa de teste, uma descrição detalhada de cada instrução presente neste código objeto. A terceira ferramenta é um compactador de código, que produz um código paralelo mapeado em uma máquina VLIW (cuja configuração pode ser determinada através de parâmetros) a partir do *trace* e da descrição do código objeto de um programa de teste. As características da máquina VLIW (isto é, seus parâmetros) são passadas para o compactador sob a forma de um arquivo.

Através da compactação de cada um dos componentes da bateria de testes para máquinas VLIW com diferentes características, foi possível obter o perfil da VLIW que melhor se adequa a cada programa de teste.

Esta tese está dividida em seis capítulos. Após esta introdução, é apresentada uma breve discussão sobre os diversos tipos de arquiteturas paralelas existentes hoje, comparando-os com a arquitetura VLIW. Para cada tipo, são apontadas as características, qualidades, deficiências e são também citadas algumas implementações. No Capítulo 3, são apresentados os passos seguidos para a construção do ambiente de avaliação utilizado na realização dos experimentos. No Capítulo 4, o conceito de compactação de código é formalmente apresentado, assim como o algoritmo de compactação utilizado nos experimentos e o modelo de máquina VLIW que foi utilizado. No Capítulo 5 são descritos os experimentos e apresentados os resultados. E, finalmente, no Capítulo 6, são listadas as conclusões deste trabalho.

@

Capítulo II

ARQUITETURAS PARALELAS E MÁQUINAS VLIW

II.1 Introdução

O desempenho de um computador pode ser medido pelo tempo que ele requer para executar programas. Quanto menor este tempo maior o desempenho. O tempo de execução de um programa, T, pode ser expresso por:

$$T = N \text{x} C \text{x} S$$

onde N é o número total de instruções executadas, C é a média do número de ciclos por instrução e S é o número de segundos por ciclo. Em uma primeira aproximação, N, C, e S são afetados primariamente pela capacidade de otimização do compilador; pela arquitetura do processador e pelo seu conjunto de instruções; e pela tecnologia empregada na implementação da máquina.

Muito tem sido feito para reduzir a influência de cada um dos parâmetros que afetam T. As técnicas utilizadas para tal têm sido as mais variadas. Para processadores de uso geral a abordagem tradicional é diminuir N com o custo de um pequeno aumento em C. Utilizando o paralelismo existente em máquinas microprogramadas horizontais, pode-se construir instruções complexas de modo a diminuir N sem um aumento equivalente de C. Essa abordagem é hoje conhecida como CISC (*Complex Instruction Set Computer*).

Em contraste, a abordagem RISC (*Reduced Instruction Set Computer*) [1] busca reduzir T através do uso de instruções muito simples, *pipelined*, freqüentemente implementadas diretamente no *hardware* e na sua maioria executadas em um único ciclo ($C = 1$). Devido à simplicidade das instruções, a implementação física do processador permite também uma redução de S. O aumento de N ocasionado por esta mesma simplicidade pode ser minimizado por técnicas de otimização de código embutidas no compilador.

Embora ambas as abordagens tenham obtido sucesso em tempos diferentes e sob diferentes circunstâncias da história da computação, o desempenho máximo obtido por cada uma delas é limitado, em última análise, pelos componentes utilizados na construção do processador. O constante avanço da tecnologia de fabricação de circuitos integrados permite a construção de computadores cada vez mais velozes mas, assumindo o uso da mais rápida tecnologia disponível, qualquer incremento de desempenho só pode ser obtido através da exploração do paralelismo existente nos programas.

Existem duas formas básicas de paralelismo: paralelismo de granularidade fina e paralelismo de granularidade grossa [2]. O paralelismo de granularidade fina é aquele que pode ser encontrado em trechos de programa onde é feita avaliação de expressões, por exemplo; é o paralelismo a nível de operações individuais.

.

.

$$e := (a + b) * (c + d);$$

.

.

Na linha de programa acima, a soma de escalares $(a + b)$ pode ser avaliada ao mesmo tempo que a soma de escalares $(c + d)$.

O paralelismo de granularidade grossa está presente em operações sobre grandes estruturas de dados, como matrizes; é o paralelismo a nível de subrotinas.

$$E := (A + B) * C^{-1};$$

Na linha de programa acima a soma de matrizes $(A + B)$ pode ser feita ao mesmo tempo que a inversão da matriz C.

Nos últimos anos diversas arquiteturas paralelas foram propostas e avaliadas [3,4]. Cada uma delas foi concebida para, de alguma maneira, explorar o paralelismo existente nos programas. A seguir, neste capítulo, serão apresentadas algumas destas arquiteturas, incluindo as máquinas VLIW. Elas serão comparadas para que se possa verificar suas diferenças, vantagens e desvantagens.

II.2 Arquiteturas SIMD

Segundo Flynn [5] existem quatro categorias básicas de processadores:

SISD - (*Single Instruction stream, Single Data stream*) - são os computadores seqüenciais;

MISD - (*Multiple Instruction stream, Single Data stream*) - seria o equivalente a diversos processadores executando diferentes instruções sobre os mesmos dados. Não parece haver aplicação prática para este tipo de arquitetura e não existe exemplo de computador pertencente a esta categoria;

SIMD - (*Single Instruction stream, Multiple Data stream*) - tipo de arquitetura onde diversos processadores executam a mesma instrução sobre diferentes conjuntos de dados;

MIMD - (*Multiple Instruction stream, Multiple Data stream*) - neste tipo de arquitetura, diversos processadores executam, de forma autônoma, diferentes instruções sobre diferentes conjuntos de dados.

Embora esta classificação seja elegante e prática, ela não é precisa o suficiente para classificar a grande diversidade de máquinas existente hoje. Por isso, diversos pesquisadores fizeram novas propostas de classificação para complementar a proposta original de Flynn [6,4,2].

O termo SIMD caracteriza aquelas arquiteturas que empregam uma unidade de controle, várias unidades de processamento - ou processadores - e uma rede de comunicação interconectando os processadores e a memória. A unidade de controle transfere uma única instrução por vez para todos processadores. Estes a executam, usando dados locais a cada um deles. A rede de interconexão permite que dados calculados em um processador possam ser transferidos a outros para serem usados como operandos pelas operações subseqüentes. Processadores podem também ser habilitados ou inabilitados durante a execução de uma instrução.

As máquinas SIMD tentam reduzir o tempo total de processamento dos programas, T, através da execução de mais de uma instrução por ciclo ($C < 1$).

Para tirar proveito da potencialidade de máquinas com arquitetura SIMD é preciso que um grande número de dados sofra o mesmo tipo de operação, o que ocorre apenas com uma pequena classe de problemas. Quando isso não ocorre, mesmo que haja um grande volume de paralelismo entre as instruções do programa de aplicação, ele não pode ser explorado por estas máquinas.

Apesar desta visível desvantagem, máquinas com esse tipo de arquitetura foram construídas. Isto foi motivado pela existência de problemas de grande interesse, tais como o processamento de imagens ou a solução da equação de Poisson no modelo contínuo de computação [10], que são facilmente mapeáveis em máquinas SIMD. Como exemplos de máquinas com este tipo de arquitetura poderiam ser citadas a Conection Machine [11]e o ELLIAC IV [10], entre outras.

As máquinas com arquitetura SIMD mais conhecidas são os Processadores Vetorias ou *Vector Processors. Vector Processors* reduzem N através do uso de instruções muito complexas (instruções vetoriais). Uma única instrução vetorial pode fazer uma determinada operação sobre um grande conjunto de dados, fazendo o trabalho de muitas instruções escalares. Nestas máquinas, unidades funcionais são

fortemente otimizadas para implementar, com um alto grau de paralelismo, as instruções vetoriais. Cray-1 [7], Fujitsu VP-200 [8] e NEC SX-2 [9] são alguns exemplos de *Vector Processors.*

Se as partes mais executadas de um programa podem ser expressas de forma vetorial um *Vector Processor* será eficiente na execução do mesmo. Contudo, existe um grande número de problemas que não podem ser codificados na forma de operações vetoriais. Por isso, as máquinas vetoriais operam eficientemente para uma classe de problemas apenas. Para usar seu paralelismo potencial precisa-se vetorizar os programas, ou seja, encontrar dentro destes, trechos onde a mesma operação seja executada sobre diferentes conjuntos de dados e codificar tais trechos de forma vetorial. Existem hoje compiladores capazes de, a partir de programas seqüenciais, gerar código vetorizado, apesar de, para se obter uma maior eficiência, seja necessário a ajuda do programador.

II.3 Arquiteturas MIMD

As máquinas com arquitetura MIMD são compostas por um conjunto de unidades de processamento independentes, ligadas entre si por uma rede de interconecção, ou por trechos de memória compartilhada, ou mesmo com a totalidade de sua memória compartilhada. Nestas máquinas, também conhecidas como Multiprocessadores ou Processadores Paralelos (*Parallel Processors*), as diversas unidades de processamento cooperam para executar um programa. Para isso o mesmo é dividido em tarefas, sendo atribuída uma ou mais tarefas a cada uma das unidades de processamento que compõem a máquina. A rede de interconecção - ou a memória compartilhada - permite a comunicação e sincronização de tarefas, alocadas em unidades diferentes.

A atribuição de tarefas deve ser feita de modo a minimizar a comunicação entre as unidades de processamento, já que esta passará a influir no tempo total de execução do programa. Além da comunicação, o tempo gasto para distribuir as tarefas e para sincronizar o fluxo de dados e de controle também devem ser considerados. Devido a estas características, os Processadores Paralelos são próprios para explorar o paralelismo de granularidade grossa presente nos programas.

As máquinas MIMD minimizam T executando mais de uma instrução por ciclo ($C < 1$). Em um dado instante de tempo t, ou em um hipotético ciclo global c, cada um dos seus processadores pode estar executando uma instrução.

A necessidade de sincronização, distribuição de tarefas e de comunicação são algumas das desvantagens das arquiteturas do tipo MIMD. A estas deve-se somar ainda a dificuldade de se programar tais máquinas. O estado da arte em compiladores ainda não pode, exceto em situações limitadas, receber programas codificados em FORTRAN, ou numa outra linguagem seqüencial e particioná-los automaticamente em múltiplas tarefas paralelas. Cabe aos programadores esta missão. Este problema tem sido atacado de duas formas diferentes: através da criação de linguagens paralelas e através da elaboração de algoritmos paralelos para os velhos problemas.

Com o grande avanço da tecnologia VLSI, pode-se construir hoje processadores de alto desempenho a baixo custo. Estes processadores podem ser usados como *building blocks* para construção de máquinas MIMD, o que torna este tipo de arquitetura muito atraente, apesar das desvantagens mencionadas.

Diversas máquinas com arquitetura MIMD foram construídas, tendo cada uma delas características próprias quanto à unidade de processamento e rede de interconexão. Como exemplos de MIMDs poderiam ser citadas a Cosmic Cube [12], a Butterfly [13], a Cedar [14], o Intel iPSC [15], a Máquina Paralela Híbrida (MPH) [16,17,18] e mais recentemente o NCP-1 da COPPE/UFRJ [19].

II.4 A Alternativa VLIW

As máquinas VLIW (*Very Long Instruction Word*) [21] são arquiteturas altamente paralelas que oferecem uma alternativa aos Processadores Vetoriais e Multiprocessadores. Elas se assemelham muito a máquinas microprogramadas horizontais e são caracterizadas por:

- Uma unidade de controle central que inicia uma única instrução longa (*long instruction*) por ciclo.

- Um grande número de *data paths* e unidades funcionais, sendo o controle destas planejado em tempo de compilação.
- Cada instrução longa inicia simultaneamente diversas operações nas diversas unidades funcionais que compõem a máquina.
- Cada operação requer um pequeno e estaticamente previsível número de ciclos para ser executada.
- Não existem árbitros de barramento, filas ou outros mecanismos de sincronização por *hardware* na unidade de controle.
- Algumas operações são *pipelined.*

As VLIW não se enquadram em nenhuma das duas classificações apresentadas anteriormente. Não são MIMD nem SIMD. Por permitirem a execução de diferentes instruções em um mesmo ciclo não podem ser classificadas como SIMD e, por seu fluxo de instruções ser centralizado, não podem ser classificadas também como MIMD. Apesar disso, do mesmo modo que os Processadores Vetoriais e os Processadores Paralelos, as VLIW reduzem o tempo total de execução dos programas, T, tornando $C < 1$, sendo que nestas isto é feito de forma muito fina - diversas instruções são empacotadas em uma mesma *long instruction.*

Diferente dos Processadores Vetoriais, as VLIW não requerem uma grande regularidade no código para fazer uso efetivo da potencialidade do seu *hardware.* E diferente dos Multiprocessadores, não há nelas perdas por sincronização ou comunicação. Todas as unidades funcionais executam seu codigo completamente sincronizadas e diretamente controladas, em cada ciclo do relógio, pelo compilador.

Processadores Vetoriais e Processadores Paralelos não possuem compiladores que possam produzir bom código paralelo para um grande número de programas seqüenciais. Já as máquinas VLIW, por se assemelharem muito a máquinas microprogramáveis horizontais, são, como estas, quase impossíveis de programar manualmente de forma eficiente. A tarefa de produzir código para elas deve ficar a cargo do compilador. Usando técnicas como *Trace Scheduling* [20,22,23] ou *Percolation Scheduling* [24], um compilador para VLIW liberta o programador da

difícil tarefa de localizar o paralelismo e explorá-lo dentro da estrutura do *hardware*. Assim, as VLIW podem explorar fortemente o paralelismo de granularidade fina e, com as técnicas citadas de compilação e otimização de código, o paralelismo de granularidade grossa existente nos programas. Isso torna o uso de máquinas VLIW a mais prática maneira de se obter *speedup* para a grande maioria das aplicações computacionais.

Muitas máquinas similares a VLIWs foram contruídas no passado (antes do surgimento destas) mas não foram capazes de prover grandes quantidades de paralelismo. Neste grupo poderia-se incluir apenas máquinas microprogramadas horizontais, como o CDC 6600 e muitos dos seus sucessores, o IBM 360/91, assim como a porção escalar do Cray-1. Estas máquinas eram programadas no seu nível de microcódigo utilizando-se compactação de código: técnica de conversão de microcódigo seqüencial em microcódigo paralelo equivalente, através do empacotamento de várias microoperações em uma microinstrução larga [25].

Experimentos com compactação de código demonstraram que o paralelismo dentro de blocos básicos (um bloco básico de código não tem desvios levando a ele, exceto no início, e não tem desvios deixando-o, exceto no final) é muito limitado [26]. Não se poderia esperar *speedup* superior a dois ou três devido ao paralelismo disponível no *hardware*.

O advento da técnica de compactação de código *Trace Scheduling* [20], viabilizou o surgimento das máquinas VLIW. *Trace Scheduling* é uma técnica de compactação global de código - compactação além dos limites dos blocos básicos que compõem os programas.

O compilador com compactação *Trace Scheduling* analisa o programa como um todo. Após fazer um conjunto completo de otimizações clássicas, incluindo movimentação de *loop-invariantes*, eliminação de subexpressões comuns e de variáveis de indução, o compilador constrói um grafo de fluxo do programa, com cada uma das operações representada por um nó. Usando estimativas de direção de desvio obtidas automaticamente através de heurísticas, o compilador seleciona o caminho, ou "*trace*", mais usado pelo programa durante a execução. Este *trace*

é tratado como se fosse livre de desvios condicionais e é apresentado ao gerador de código. O gerador de código escalona as operações dentro de palavras de instrução longas (*long instruction word*), levando em consideração a precedência de dados, o ótimo escalonamento de unidades funcionais, a utilização dos registradores e os acessos à memoria e aos barramentos, emitindo *long instructions* como código objeto. Mas o escalonamento feito sem levar em consideração desvios condicionais pode levar a inconsistências quando um desvio para fora do *trace* ocorre. Por isso, o compilador insere código de compensação no grafo do programa nos trechos que contêm os desvios para fora do *trace*, corrigindo as inconsistências geradas e restaurando, assim, a semântica do programa. Este processo permite ao compilador quebrar o gargalo dos desvios condicionais, provendo grandes trechos de código onde buscar paralelismo.

Experimentos demonstraram que o paralelismo disponível a máquinas VLIW é muito grande, ficando, em muitos casos, limitado apenas pelo tamanho dos dados [27]. Mas qual deveria ser o "tamanho" das máquinas VLIW para atender com bom desempenho aos programas mais típicos? É certo que não se deve projetá-las para casos particulares, com um grande número de unidades funcionais, já que elas seriam subutilizadas em situações onde não houvesse tanto paralelismo disponível. Por outro lado, projetando-as com poucas unidades, poderíamos estar perdendo a oportunidade de aumentar o *speedup* em muitos casos típicos. Assim, realizar medidas para se saber qual é a configuração mais apropriada para as máquinas VLIW se torna necessário. Alguns experimentos com este objetivo foram feitos pelo autor deste trabalho e estão aqui apresentados.

II.5 Outras Arquiteturas Paralelas

Além das arquiteturas paralelas apresentadas aqui, muitas outras foram e vêm sendo desenvolvidas. Uma discussão bastante aprofundada das mesmas (datada de 1985) foi feita por Gajski [2]. Um *survey* apresentando a maioria das arquiteturas paralelas desenvolvidas até 1990 pode se encontrado em [3].

O próximo capítulo é dedicado à descrição do ambiente desenvolvido

para determinar a configuração, isto é, número de unidades funcionais, barramentos, canais de comunicação com a memória, tamanho do banco de registradores, etc, que uma máquina VLIW deve ter para casos típicos.

@

Capítulo III

O AMBIENTE DE AVALIAÇÃO

III.1 Introdução

Uma máquina VLIW pode ser vista como um conjunto de unidades de processamento capazes de executar operações simples, como somas ou multiplicações, conectadas a um banco de registradores e a memória (Figura III.1). Elas se assemelham a um conjunto de processadores RISC com um contador de programa único e com um banco de registradores e memória comum. Fazer bom uso desta máquina consiste em manter o maior número possível dessas unidades fazendo trabalho útil durante a execução dos programas, transferindo para cada unidade uma operação para ser executada (ou um NOP quando não for possível utilizá-la), a cada ciclo de *clock*. Uma instrução para esta máquina deve incluir um número suficiente de campos para: comandar cada uma das unidades de processamento; controlar os acessos à

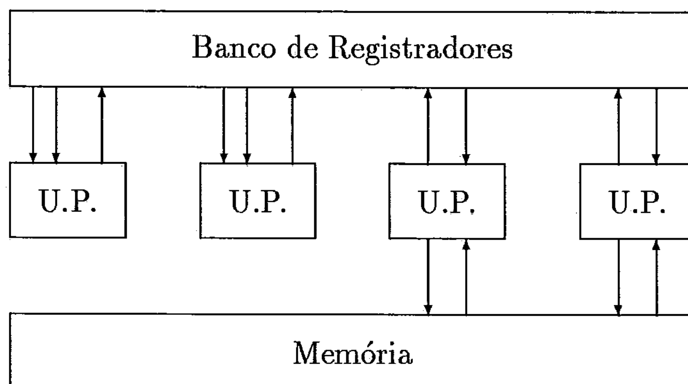


Figura III.1: Modelo Básico de VLIW

memória; e controlar o seqüenciamento das instruções.

O objetivo deste trabalho é determinar, para casos típicos, o número e o tipo de unidades de processamento (somadores, multiplicadores, etc), quantos canais entre estas unidades de processamento e o banco de registradores e quantos canais destes com a memória devem ser incorporados numa arquitetura VLIW.

III.2 O Projeto do Experimento

Dado um programa, qual deve ser a configuração de uma máquina VLIW para explorar seu paralelismo latente ao máximo? Esta foi a pergunta que se buscou responder com esta tese através de experimentos. Não se desejava que a limitação imposta pelos desvios condicionais interferisse nas medidas - existe muito paralelismo disponível além dos blocos básicos. Mas como evitar esta interferência? A solução foi supor que a máquina VLIW era capaz de prever o endereço resultante de um desvio condicional. Isto, embora seja um suposição muito forte, é verossímil, já que a máquina com esta capacidade seria equivalente a uma DataFlow com recursos ilimitados de *hardware* [28].

Para realizar os experimentos os seguintes estágios foram executados:

- escolha de um nível de linguagem onde o paralelismo pudesse ser encontrado;
- escolha de programas de teste dentro dos quais o paralelismo latente pudesse ser detectado e explorado pela máquina sendo avaliada;
- desenvolvimento de uma ferramenta que permitisse detectar o paralelismo dos programas e, para as diferentes configurações da máquina em estudo, avaliar o volume desse paralelismo.

III.3 O *Assembly* do Microprocessador i860

O nível de linguagem escolhido para detectar paralelismo foi o *Assembly.* No nível de *Assembly* é possível, através de procedimentos relativamente simples, fazer análises

do código, o que em níveis mais altos se torna bastante complicado. A linguagem *Assembly* escolhida foi a do microprocessador Intel i860.

O microprocessador Intel i860 define uma arquitetura completa contendo unidades que realizam operações com inteiros, com reais e operações gráficas [29,30]. Suas aplicações típicas incluem estações de trabalho (*workstations*) e computação científica. Sua arquitetura paralela obtém alto desempenho através de técnicas de projeto RISC, unidades de processamento *pipelined*, caminhos de dados largos e memórias *cache* para instruções e dados *on-chip*.

As operações do i860 são especificadas numa única palavra. Isso, somado ao fato das instruções serem RISC, permite identificar claramente que recursos do processador estão sendo utilizados em cada ciclo de máquina. Por estas características o *Assembly* do microprocessador i860 foi escolhido como nível de linguagem onde o paralelismo dos programas de teste seria detectado.

III.4 Os Programas de Teste

As máquinas VLIW foram consideradas, no passado, mais apropriadas para executar código científico. Dadas as suas dimensões e características, só aplicações científicas pareciam justificar sua construção. Com o avanço da tecnologia de fabricação de circuitos integrados, que permite hoje a fabricação de dispositivos com centenas de milhares de gates, as máquinas VLIW já podem ser construídas para uso geral. Assim, os programas de teste escolhidos para realização dos experimentos foram programas típicos de máquinas de uso geral. São eles:

Livermore Loop 24 - algoritmo para determinação do menor componente de um vetor de dimensão n [31];

Quick Sort - algoritmo de ordenação [32];

Busca Binária - algoritmo de busca [33];

Bubble Sort - algoritmo de ordenação, método da bolha [32];

Decomposição LU - método para solução de equações lineares baseado na Eliminação Gaussiana [34];

Integração numérica - algoritmo de integração numérica segundo o método do trapézio [35].

Estes programas foram escritos em linguagem C e posteriormente traduzidos para o *Assembly* do i860 pelo compilador HighC, da Metaware [36]. Os programas de teste estão listados no Apêndice A.

III.5 As Ferramentas de Avaliação

Se o *hardware* de uma máquina for capaz de prever precisamente o resultado de cada desvio condicional, a exploração do paralelismo disponível fica limitada apenas à dependência de dados existente nos programas. No entanto, desenvolver uma máquina com esta capacidade é um tarefa muito difícil. Uma possível abordagem seria executar o mesmo programa em diversas máquinas simultaneamente. Para cada desvio teríamos duas máquinas, cada uma delas executando as instruções de um dos dois caminhos. O processamento da que escolheu o caminho errado seria posteriormente descartado, retendo-se o trabalho da que escolheu o caminho correto de cada desvio. Apesar de se conseguir *speedups* significativos com esta técnica, experimentos mostraram que ela é impraticável. Para se conseguir um *speedup* de 10 teríamos que começar com $O(2^{16})$ máquinas [37]!

Para obter as medidas de desempenho deste trabalho era preciso uma ferramenta que permitisse prever precisamente a direção tomada por cada desvio condicional, já que numa máquina VLIW é essencial explorar o paralelismo além dos blocos básicos. Por outro lado, a inclusão de um mecanismo de previsão dinâmica de desvios numa arquitetura VLIW é uma decisão de projeto que contraria radicalmente um dos princípios que caracterizam processadores desse tipo (isto é, a complexidade do *hardware* deve ser minimizada de modo a aumentar a velocidade de processamento da máquina resultante) e por essa razão, o algoritmo de escalonamento das instruções que serão executadas simultaneamente durante cada ciclo da máquina

VLIW é implementado numa fase que precede a interpretação do código objeto. Esse algoritmo de escalonamento usualmente é implementado (através de *software*) pelo compilador, durante a fase de otimização de código. Deste modo, decidiu-se produzir um *trace* dos programas da bateria de testes e em seguida, através de um algoritmo de compactação de código, foi medido o paralelismo existente em cada um dos componentes da bateria de testes.

O *trace* é uma ferramenta de avaliação muito poderosa, pois permite visualizar globalmente a execução do programa, exatamente o que é necessário para realizar as medidas almejadas. No *trace*, todos os desvios têm o seu destino determinado, e por esse motivo, o programa todo é transformado em um único bloco básico!

Para se obter o *trace* dos programas de teste foi desenvolvido um simulador do processador i860, o Sim860 (vide Apêndice B). Os programas de teste foram processados pelo compilador HighC da Metaware e posteriormente simulados no Sim860. Durante a execução, o Sim860 gera um arquivo contendo o *trace* do programa que está sendo simulado. Todas as características do i860 foram implementadas no simulador e também todas as instruções, com exceção de algumas instruções gráficas e de controle do barramento.

Além do Sim860 foi contruída outra ferramenta para realização dos experimentos: um gerador de código estático. Este programa, denominado Scode (vide Apêndice B), gera um arquivo contendo uma descrição detalhada de cada instrução utilizada no programa, a partir do executável criado pelo compilador. De posse deste arquivo e do arquivo de *trace*, é possível alimentar o programa na máquina VLIW sendo avaliada. Aqui, alimentar o programa na VLIW sendo avaliada significa escalonar cada operação presente no *trace*, de acordo com os recursos existentes na máquina VLIW, e obedecendo as restrições impostas pelas dependências de dados existente no programa, ou seja, compactar o código. Como este trabalho foi feito utilizando-se um *trace*, a tarefa de compactação se resume na compactação local (compactação dentro de blocos básicos apenas), já que um *trace*, do ponto de vista da compactação, se comporta como um único bloco básico.

Para realizar a tarefa de compactação de código foi desenvolvida uma terceira ferramenta: um compactador de código. Esse compactador, denominado Compact (vide Apêndice B), gera um código paralelo mapeado no *hardware* da máquina VLIW em estudo, a partir do arquivo de *trace* e do arquivo de código estático. Ele gera também, estatísticas para a avaliação dos resultados obtidos com cada configuração de máquina VLIW em estudo.

III.6 Técnica de Compactação Escolhida

Compactação de código é um problema NP-completo [38]. Assim, não se pode esperar que algoritmos que não levam tempo exponencial com o número de instruções a compactar, produzam resultados ótimos. Como nos experimentos relatados aqui o número de instruções é necessariamente elevado, ficaria impossível realizá-los com um algoritmo que produzisse resultados ótimos (o tempo necessário aos experimentos seria proibitivo). Existem diversos métodos de compactação local de código. Alguns ótimos e outros que nem sempre produzem resultados ótimos e, para realização deste trabalho, um deles teve que ser escolhido.

Muitos trabalhos foram feitos, por diversos pesquisadores, sobre compactação local e global de código [39,40,41,42,43]. Vários algoritmos de compactação local foram profundamente avaliados por David Landskov e Scott Davidson nos trabalhos [25,26]. Através dessas avaliações, foi escolhido o algoritmo FCFS (*First-Come First-Served*), pelo seu bom desempenho e por ser apropriado ao problema.

No próximo capítulo o conceito de compactação de código é formalmente apresentado, assim como o algoritmo FCFS. Apresenta-se, também, uma descrição do modelo de máquina VLIW em estudo.

@

Capítulo IV

COMPACTAÇÃO DE CÓDIGO

IV.1 Introdução

A expressão "Compactação de Código" foi inicialmente utilizada para especificar uma tarefa relacionada à microprogramação de máquinas microprogramáveis horizontais.

Um microprograma é uma seqüência de microinstruções. Eles são armazenados em uma memória especial, chamada memória de controle, e as microinstruções que os compõem são executadas uma de cada vez (normalmente uma a cada ciclo de máquina). Durante a execução, as microinstruções são as palavras de controle da máquina. Cada atividade do processador, especificada dentro de uma microinstrução, é chamada de microoperação. Máquinas microprogamáveis horizontais são caracterizadas por possuirem microinstruções largas, nas quais diversas microoperações podem ser alocadas. Compactar o código é escolher, dentre os diversos arranjos possíveis de microoperações, aquele que minimize o tamanho do microprograma e, conseqüentemente, seu tempo de execução.

Nesta tese, compactação tem significado equivalente. Onde se tinha microinstruções, temos Instruções Longas (IL) e onde se tinha microoperações, temos Instruções (IN). O problema de compactação pode então ser definido como:

Compactação - Para uma determinada máquina VLIW é dado um programa definido por uma seqüência de INs. Compactar este programa

consiste em alocar estas INs nas ILs, de modo que o tempo de execução do programa seja minimizado. Essa alocação deve respeitar duas restrições:

1. A máquina deve possuir recursos (unidades funcionais) disponíveis para executar as INs alocadas em cada IL.
2. A seqüência resultante de ILs deve ser semanticamente equivalente à seqüência original de INs.

Por "semanticamente equivalente" deve-se entender que, se ambas as seqüências são executadas com a mesma entrada, a saída deve ser sempre a mesma.

Para se garantir que não haja alteração semântica no programa, devem ser observadas as "dependências de dados" entre as INs durante a compactação.

IV.2 Dependência de Dados

A maioria das INs de uma máquina opera sobre registradores. Um registrador cujo valor é usado por uma IN é chamado de registrador de entrada ou operando de entrada. Analogamente, um registrador cujo valor é alterado por uma IN é chamado de registrador de saída ou operando de saída. Uma IN pode ter também, como operandos de entrada ou saída, posições de memória ou flags.

Dado um programa para ser compactado, a lista final de ILs obtida após a compactação deve ser semanticamente equivalente ao programa original. Para produzir um código eficiente, o processo de compactação modifica a ordem original das INs, movimentando-as ao longo da lista de ILs. Como conseqüência dessa movimentação, a ordem de execução das INs é alterada: o processo de compactação pode antecipar (ou retardar) o início da execução de algumas INs. Em algumas situações, o algoritmo de compactação precisa manter o seqüenciamento especificado pelo programador de modo a preservar a integridade do programa de aplicação. A ordem de execução de duas INs não pode ser alterada se elas interagirem. A interação de duas INs (denominada também Interação de Dados) pode ser definida como:

Interação de Dados - Ocorre entre duas INs, IN_i e IN_j, onde IN_i ocorre antes que IN_j no programa original, satisfazendo as seguintes condições:

1. Um operando de saída de IN_i é usado como operando de entrada de IN_j.
2. Um operando de entrada de IN_i atua como operando de saída de IN_j.
3. Um operando de saída de IN_i também é um operando de saída de IN_j.

Se IN_j preceder IN_i no programa compactado, o seguinte pode ocorrer:

- Se a primeira condição for violada, IN_j pode utilizar um valor antigo como operando de entrada ao invés do novo valor gerado por IN_i.
- Se a segunda condição for violada, IN_j pode destruir um valor antes que IN_i o tenha utilizado.
- Caso a terceira condição seja violada, o valor final de um operando (após a execução de IN_i e IN_j), que deveria ser avaliado por IN_j, recebe o valor produzido por IN_i. Se este valor não for mais utilizado como operando no programa, esta condição pode ser desconsiderada.

A lista de ILs resultante da compactação será semanticamente equivalente à lista original de INs se, para cada par de INs na lista final de ILs que apresente interação de dados, a IN ocorrendo antes na lista original de INs, termine de usar cada operando causando interação de dados antes que a outra comece a utilizá-los. Isto leva diretamente às seguintes definições:

Ordem Preservada - Considere IN_i e IN_j, duas INs de um programa, com IN_i ocorrendo antes que IN_j. As duas INs têm ordem preservada se, para cada operando causando interação de dados entre elas, IN_i liberar esses operandos antes que IN_j comece a utilizá-los.

Dependência de Dados Direta - Considere IN_i e IN_j, duas INs de um programa, IN_i ocorrendo antes que IN_j. IN_j possui dependência de dados direta com IN_i, ou IN_i **ddd** IN_j, se as duas INs apresentarem interação de dados e se não existir uma seqüência de INs com dependência direta de dados entre elas. Isto é, não existir um seqüência de INs, IN_{m1}, IN_{m2}, ..., IN_{mn}, $n \geq 1$ tal que IN_i **ddd** IN_{m1}, IN_{m1} **ddd** IN_{m2}, ..., $IN_{m(n-1)}$ **ddd** IN_{mn}, IN_{mn} **ddd** IN_j.

Dependência de Dados - Considere novamente IN_i e IN_j, duas INs de um programa, IN_i ocorrendo antes que IN_j. IN_j apresenta dependência de dados com IN_i, ou IN_i **dd** IN_j, se IN_i **ddd** IN_j ou se existe uma IN, IN_k, tal que IN_i **ddd** IN_k e IN_k **dd** IN_j. Dependência de dados é o fechamento transitivo da dependência de dados direta.

Duas INs apresentando dependência de dados devem ter sua ordem preservada durante a compactação para que o programa compactado seja semanticamente equivalente ao programa original.

IV.3 Modelando a Máquina

Um modelo de máquina é necessário porque qualquer esquema de compactação deve produzir código levando em consideração as características da máquina onde ele será executado. Duas INs não podem ser alocadas na mesma IL se ambas necessitam de controle exclusivo de um mesmo recurso. Por exemplo: se existe uma única unidade lógica na máquina, apenas uma IN do tipo AND, ou qualquer outra operação lógica, pode ser alocada em cada IL.

O modelo de máquina usado nesse estudo pode ser visto na Figura IV.1 . Neste modelo a máquina é formada por componentes de 12 tipos:

1. U.L.A.I. - Unidade lógica e aritmética inteira;
2. S.E. - Somador de Endereços (responsável pelo cálculo dos desvios);

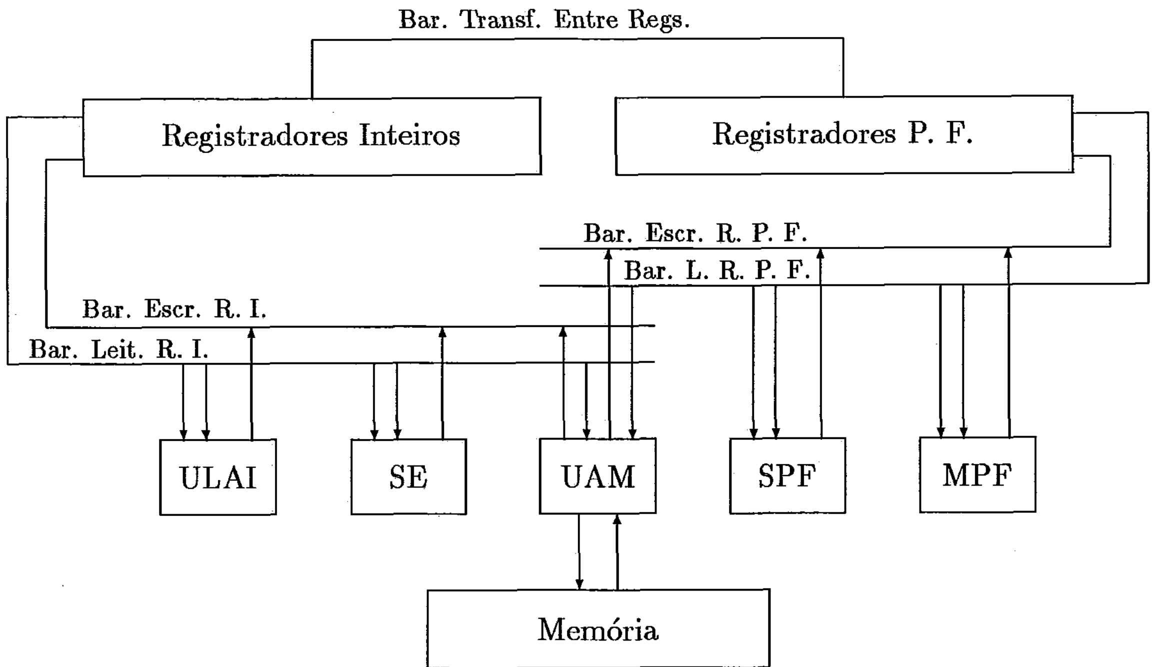


Figura IV.1: Máquina VLIW em Estudo

3. U.A.M. - Unidade de Acesso à Memória (responsável pelas leituras e escritas na memória);
4. S.P.F. - Somador de Ponto Flutuante;
5. M.P.F. - Multiplicador de Ponto Flutuante;
6. Banco de Registradores Inteiros;
7. Banco de Registradores de Ponto Flutuante;
8. Barramento de Leitura em Registradores Inteiros;
9. Barramento de Escrita em Registradores Inteiros;
10. Barramento de Leitura em Registradores de Ponto Flutuante;
11. Barramento de Escrita em Registradores de Ponto Flutuante;
12. Barramento de Transferência entre Registradores Inteiros e Registradores de Ponto Flutuante.

Esta máquina pode possuir mais de uma unidade funcional de cada tipo, um banco de registradores inteiros e um outro de registradores de ponto flutuante. Todas as unidades podem operar em paralelo. Uma IL para esta máquina tem campos para controlar cada uma das unidades, e cada campo deve ser preenchido por uma IN (ou um NOP). As INs podem ser de vários tipos e podem consumir mais que um campo, ou seja, utilizar mais que uma unidade funcional para sua execução.

A semântica de uma IN pode ser representada por uma t-upla com seis elementos:

NOME - Identificador da operação;

E - Conjunto dos recursos de entrada cujo conteúdo é lido pela IN (registradores e Flags por exemplo);

S - Conjunto de recursos de saída onde a IN escreve resultados;

U - Conjunto de unidades funcionais utilizadas pela IN durante sua execução;

CI - Número de ciclos de máquina entre o início da IN (leitura dos operandos) e o seu término (escrita dos operandos);

CA - Conjunto de campos de IL consumidos por IN.

Utilizando-se desta representação para as INs, conflitos de recursos entre elas podem ser detectados, durante a compactação, através do teste de interseção não vazia entre os vários conjuntos. Quando a interseção não é vazia, diz-se que existe dependência de recursos.

IV.4 Dependência de Recursos

Cada campo de uma IL está relacionado a uma unidade funcional da máquina VLIW. Os campos são especializados, embora possa existir mais de um campo com mesma finalidade, já que a máquina pode possuir réplicas de uma mesma unidade funcional.

Durante a compactação, ao se tentar alocar uma IN em uma IL, pode não haver campos disponíveis por já terem sido ocupados anteriormente por outras

INs. Neste caso não é possível alocar a IN. Assim, uma IN não pode ser alocada em uma IL quando ela satisfaz à seguinte definição:

> **Dependência de Recursos** - Existe entre uma IN e uma IL quando não houver campo disponível em IL para alocar IN, por já ter sido, ou terem sido, ocupados por outras INs.

IV.5 O Algoritmo de Compactação

O algoritmo de compactação utilizado foi o FCFS (*Fisrt-Come Fisrt-Served*), descrito inicialmente por Dasgupta e Tartar em [44]. Ele opera sobre a lista de INs que compõem o programa original, tomando uma IN de cada vez, na ordem em que elas aparecem no programa, e adicionando-as a uma lista (inicialmente vazia) de ILs.

A busca por uma IL, na qual a IN correntemente sendo examinada possa ser adicionada, inicia com uma análise de dependência de dados. Começando no fim da lista de ILs e seguindo para o início, vão sendo examinadas cada uma das ILs. A busca segue para uma IL mais próxima do início se a IN considerada não possuir dependência de dados com nenhuma IN na IL corrente. Se a IN tem dependência de dados com alguma outra IN na $i-ésima$ IL, a IN corrente não pode ser alocada em qualquer IL_j com $0 \leq j \leq i$. O objetivo desta busca é encontrar a IL de menor índice onde a IN possa ser alocada. Este índice é denominado *limite superior*.

O próximo passo para alocar a IN consiste em examinar os conflitos de recursos. Para ser alocada em uma IL, a IN corrente não pode apresentar conflitos de recursos com a mesma. Assumindo que o *limite superior*, i, foi encontrado, deve-se seguir em direção ao fim da lista de ILs, começando em IL_i, em busca de uma IL onde a IN corrente possa ser alocada. Quando esta IL for encontrada, ocorre uma inclusão. Se esta IL não for encontrada deve-se adicionar IN ao final da lista de ILs, através da criação de uma nova IL para receber IN.

Se não for encontrado um *limite superior*, a IN corrente não tem dependência de dados com nenhuma IN contida na lista de ILs e pode ser alocada

em qualquer IL com a qual não tenha dependência de recursos. Neste caso, deve-se seguir do topo da lista em direção à ultima IL, em busca de um lugar para IN. Se esta busca falhar, cria-se uma nova IL no topo da lista para receber IN. Colocar esta IN no topo da lista irá evitar que ela bloqueie as INs subseqüentes que tenham dependência de dados com ela.

A Figura IV.2 apresenta o algoritmo FCFS implementado na linguagem C.

IV.6 As Características da Máquina VLIW

O nível de linguagem escolhido para detectar o paralelismo dos programas de teste foi o *Assembly* e o *Assembly* escolhido foi o do microprocessador i860. Através do estudo da arquitetura interna do mesmo foram caracterizadas as unidades funcionais usadas no modelo de máquina VLIW em estudo.

O microprocessador i860 é formado por 9 blocos:

1 - Unidade Central de Execução - *CORE*;

2 - Unidade de Controle de Ponto Flutuante;

3 - Somador de Ponto Flutuante;

4 - Multiplicador de Ponto Flutuante;

5 - Unidade Gráfica;

6 - Unidade de Paginação de Memória;

7 - *Cache* de Instruções;

8 - *Cache* de Dados;

9 - Unidade de Controle do Barramento e dos *Caches*.

O *CORE* controla todas as operações do i860. É responsável também pela execução das instruções inteiras, de *load*, *store*, operações de manipulação de

```
/* Prototypes */
ITEM_LISTA_DE_ILS *Coloca_Nova_IL_no_Fim_da_Lista (ITEM_LISTA_DE_ILS *);
ITEM_LISTA_DE_ILS *Coloca_Nova_IL_no_Inicio_da_Lista (ITEM_LISTA_DE_ILS *);
void Coloca_IN_em_IL (INSTRUCAO_LONGA, INSTRUCAO);
BOOLEAN Teste_Dependencia_de_Dados (INSTRUCAO_LONGA, INSTRUCAO);
BOOLEAN Teste_Dependencia_de_Recursos (INSTRUCAO_LONGA, INSTRUCAO);

/* Variaveis Globais */
ITEM_LISTA_DE_ILS *VAZIA = (ITEM_LISTA_DE_ILS *) NULL;
ITEM_LISTA_DE_ILS *inicio_lista = (ITEM_LISTA_DE_ILS *) NULL;
ITEM_LISTA_DE_ILS *fim_lista, *i, *LIMITE_SUPERIOR_MAIS_UM;

void
Compact (IN)
INSTRUCAO IN;
{
        if (inicio_lista == VAZIA)
        {
                fim_lista = inicio_lista = Coloca_Nova_IL_no_Inicio_da_Lista (inicio_lista);
                Coloca_IN_em_IL (inicio_lista- >IL, IN);
        }
        else
        {
                i = fim_lista;
                LIMITE_SUPERIOR_MAIS_UM = (ITEM_LISTA_DE_ILS *) NULL;
                do
                {
                        if (Teste_Dependencia_de_Dados (i- >IL, IN) == FALSE)
                                i = i- >anterior;
                        else
                                LIMITE_SUPERIOR_MAIS_UM = i;
                }
                while (i != LIMITE_SUPERIOR_MAIS_UM);

                if (i != (ITEM_LISTA_DE_ILS *) NULL)
                {
                        i = i- >proximo;
                        while ( (i != (ITEM_LISTA_DE_ILS *) NULL) &&
                                (Teste_Dependencia_de_Recursos (i- >IL, IN) == TRUE) )
                                i = i- >proximo;

                        if (i != (ITEM_LISTA_DE_ILS *) NULL)
                                Coloca_IN_em_IL (i- >IL, IN);
                        else
                        {
                                fim_lista = Coloca_Nova_IL_no_Fim_da_Lista (fim_lista);
                                Coloca_IN_em_IL (fim_lista- >IL, IN);
                        }
                }
                else
                {
                        i = inicio_lista;
                        while ( (i != (ITEM_LISTA_DE_ILS *) NULL) &&
                                (Teste_Dependencia_de_Recursos (i- >IL, IN) == TRUE) )
                                i = i- >proximo;

                        if (i != (ITEM_LISTA_DE_ILS *) NULL)
                                Coloca_IN_em_IL (i- >IL, IN);
                        else
                        {
                                inicio_lista = Coloca_Nova_IL_no_Inicio_da_Lista (inicio_lista);
                                Coloca_IN_em_IL (inicio_lista- >IL, IN);
                        }
                }
        }
}
```

Figura IV.2: Algorítmo FCFS

bits, de desvio e pelo *fetch* das instruções destinadas às unidades de ponto flutuante. O i860 possui um banco com 32 registradores de 32 bits de uso geral ligado ao *CORE*, para o armazenamento de dados inteiros. Instruções de *load* e *store* podem mover dados de 8, 16 e 32 bits entre a memória e estes registradores.

No modelo de máquina VLIW em estudo, o *CORE* não entrou no conjunto de unidades funcionais como sendo uma única unidade. O *CORE* é muito complexo para ser tratado aqui como um único bloco básico. Assim, foram definidas 3 unidades funcionais para realizar as instruções executadas pelo *CORE*. São elas:

- Unidade Lógica e Aritmética Inteira - responsável pelas instruções inteiras;
- Somador de Endereços - responsável pela avaliação dos endereços alvejados pelos desvios;
- Unidade de Acesso à Memória - responsável pelos *loads* e *stores*.

No i860, o *hardware* de ponto flutuante está conectado a um conjunto exclusivo de registradores de ponto flutuante, que podem ser acessados como 32 registradores de 32 bits, ou como 16 registradores de 64 bits. Instruções de *load* e *store* especiais podem também acessar estes registradores como se fossem 8 elementos de 128 bits. Todas as instruções de ponto flutuante usam estes registradores como operandos fonte e destino. A Unidade de Controle de Ponto Flutuante controla o Somador de Ponto Flutuante e o Multiplicador de Ponto Flutuante, transferindo instruções e tratando as exceções de operando fonte e de resultados. O Somador e o Multiplicador podem operar em paralelo, produzindo até dois resultados por *clock*.

O Somador de Ponto Flutuante faz adição, subtração, comparação e conversão de valores de 64 e 32 bits. As instruções do Somador são executadas em 3 *clocks*, mas quando o i860 está operando no modo *pipeline*, um novo resultado pode ser gerado a cada *clock*. O Somador de Ponto Flutuante é um dos blocos básicos da máquina VLIW em estudo e é tratado nos experimentos como se estivesse sempre operando em modo *pipeline*.

O Multiplicador de Ponto Flutuante realiza a multiplicação inteira, de ponto flutuante e também o recíproco em ponto flutuante, de valores de 32 ou

64 bits. As instruções do Multiplicador são executadas em 2 *clocks* para operandos com 64 bits e em 3 *clocks* para operandos de 32 bits, contudo, quando em modo *pipeline*, um novo resultado pode ser gerado a cada *clock*. O Multiplicador de Ponto Flutuante é um dos blocos básicos do modelo de máquina VLIW e é tratado nos experimentos como se estivesse sempre operando em modo *pipeline*.

A Unidade Gráfica possui lógica inteira capaz de suportar desenhos trí-dimensionais, com sombreamento a cores e eliminação de superfícies escondidas. Esta unidade manipula *pixels* de 8, 16 ou 32 bits. Ela pode computar, individualmente, a intensidade de vermelho, verde e azul dentro de um *pixel*, fazendo isso em paralelo com outras operações, aproveitando-se do tamanho interno de palavra de 64 bits e do barramento externo do mesmo tamanho. Não foram implementadas todas as instruções da Unidade Gráfica no Simulador do i860 (Sim860). Apenas aquelas presentes no código objeto dos programas de teste, gerados pelo compilador.

A Unidade de Paginação de memória implementa memória virtual paginada com proteção em dois níveis (supervisor e usuário), empregando um TLB (*Translation Lookaside Buffer*) *four-way*, *set-associative* com 64 entradas. Esta unidade usa o TLB para fazer a tradução do endereço lógico para o endereço físico e para detectar a ocorrencia de violações de acesso. O Sim860 implementa integralmente a Unidade de Paginação de Memória.

O *Cache* de Instruções é *two-way*, *set-associative* com 4 Kbytes e blocos (ou linhas) de 32 bytes. O *Cache* de Dados é *two-way*, *set-associative* com 8 Kbytes organizados como blocos de 32 bytes. O i860 normalmente usa *writeback caching*, isto é, a escrita é feita no *cache*. A informação atualizada só é levada para a memória numa eventual troca de blocos, determinada pela política de substituição de blocos do *cache*. O Sim860 implementa integralmente ambos os *caches* do i860.

A Unidade de Controle do Barramento e dos *Caches* é responsável pelos acessos de dados e instruções do *CORE*. Ela recebe requisições do *CORE*, trata *data-cache miss* e *instruction-cache miss*, controla as traduções de endereço feitas pelo TLB e provê interface com o barramento externo. Neste trabalho, todos estudos foram feitos sem levar em consideração os *caches* e a comunicação com os

barramentos externos. Durante as simulações os *caches* foram tratados pelo Sim860 como se estivessem ativados, mas o tempo, ou ciclos necessários a recuperação de um *cache miss* (de dados ou de instruções) não foi levado em conta. Para efeito dos experimentos feitos aqui, o tempo de um acesso à memória é fixo e igual a 6 *clocks*. O funcionamento de uma máquina VLIW com *caches* foi, até hoje, pouco estudado e é um promissor campo para pesquisa.

IV.7 Dependência de Dados e *Pipeline*

Em uma arquitetura pode-se explorar o paralelismo de duas maneiras: espacial e temporalmente. O paralelismo espacial é aquele obtido pela execução de diversas operações em paralelo em unidades funcionais diferentes. Paralelismo temporal é aquele obtido durante o processamento de diversas operações concorrentemente em uma mesma unidade funcional, aproveitando-se de sua estrutura *pipelined*. Máquinas VLIW são exemplos de arquiteturas provendo paralelismo espacial e também (embora não obrigatorimente) paralelismo temporal.

Como algumas unidades funcionais da máquina VLIW em estudo são *pipelined*, este tipo de paralelismo também foi explorado. Assim, no processo de compactação, as características *pipelined* destas unidades funcionais foram levadas em conta. Para tal, alguns cuidados foram tomados de modo a não subverter a análise de dependência de dados - como ela foi apresentada só se aplica a INs que são executadas em um único ciclo.

Uma operação pode ser transferida, a cada ciclo de relógio, para uma unidade funcional *pipelined*. No ciclo em que a operação é iniciada são lidos os operandos de entrada. Após este primeiro ciclo a operação caminhará pelo *pipe* até que, completados os ciclos de latência, o resultado seja escrito no operando de saída. Uma IN que utiliza uma unidade funcioanal *pipelined* não pode ser considerada concluída enquanto seus operandos de saída não forem escritos. Assim, uma outra IN que tenha dependência de dados com esta, só pode ser alocada em uma IL que venha ser executada após o tempo de latência da unidade funcional *pipelined* utilizada. Isto pode ser facilmente atingido na análise de dependência de dados

Unidade Funcional	Latência (ciclos)
Unidade Lógica e Aritmética Inteira	1
Somador de Endereços	2
Unidade de Acesso à Memória	6
Somador de Ponto Flutuante	2
Multiplicador de Ponto Flutuante	3 (ou 2)
Banco de Registradores Inteiros	0
Banco de Registradores de P. F.	0
Barramento de Escrita em Reg. I.	0
Barramento de Leitura em Reg. I.	0
Barramento de Escrita em Reg. P. F.	0
Barramento de Leitura em Reg. P. F.	0
Barramento de Transferência	0

Tabela IV.1: Latência das Unidades Funcionais da Máquina VLIW em Estudo

utilizando-se instruções "*dummy*".

Uma IN que seria executada num *pipeline* com n estágios é substituída por n outras INs: uma com características idênticas (utilizando os mesmos recursos de entrada, saída, unidades funcionais, campos, etc...) não *pipelined* e $n-1$ INs *dummy*, com operandos de entrada iguais aos de saída e estes iguais aos de saída da IN que as originou. Assim, pelas regras de dependência apresentadas anteriormente, este conjunto de INs permanecerá unido na compactação, uma IN por cada IL de uma seqüência de ILs com tamanho n. Fica também garantido que INs com dependência de dados com uma IN *pipelined* não serão alocadas indevidamente, já que, também terão dependência de dados com as INs *dummy*.

A t-upla que especifica as INs *dummy* tem como elementos não vazios apenas o conjunto de recursos de entrada, **E**, e o conjunto de recursos de saída, **S** (operandos de entrada e de saída), assim, estas INs não terão dependência de recursos com outras INs.

Na Tabela IV.1 estão listadas as latências de cada unidade funcional da máquina VLIW em estudo.

IV.8 Renomeação de Registradores

Os compiladores, no processo de geração de código, podem associar variáveis a registradores. Diferentes variáveis podem ser associadas a um mesmo registrador; para isso, basta que estas variáveis estejam em diferentes procedimentos (ou subrotinas). Compiladores mais "espertos" podem até fazer a associação de mais de uma variável a um mesmo registrador dentro de um único procedimento.

Na compactação do *trace* estes casos gerarão falsas dependências de dados. Para contornar este problema foi utilizada a técnica de renomeação de registradores. Esta técnica consiste em trocar o nome de um registrador quando ele apresentar dependência de dados na compactação de uma IN, por satisfazer a 2ª ou a 3ª condição de interação de dados (vide Seção IV.2).

Ao banco de registradores inteiros e de ponto flutuante da máquina em estudo foram adicionados registradores extras para renomeação. Para otimizar o uso dos mesmos é feita análise de variáveis vivas (*live-variable analysis*) [45]. Nesta análise, uma variável pode estar viva ou morta em um determinado ponto do programa.

Variável Viva - Uma variável está viva em algum ponto na execução do programa se ela vai ser lida em algum outro ponto, posterior a este, antes de uma nova escrita na mesma.

Variável Morta - Se, em um determinado ponto do programa uma variável não está viva, então, ela está morta neste ponto.

No processo de compactação os registradores são marcados como vivos ou mortos. Registradores mortos podem ser utilizados para renomeação.

IV.9 Os Parâmetros do Modelo

Ao longo dos experimentos para avaliar o efeito (no desempenho do modelo de VLIW) causado pela mistura dos diversos recursos do "*hardware*", várias confi-

gurações derivadas do modelo básico foram especificadas. Tendo em vista que o modelo básico é "parametrizado", cada conjunto distinto de parâmetros, resulta numa diferente configuração. Os parâmetros do modelo que foram investigados estão listados em seguida.

- Unidade lógica e aritmética inteira;
- Somador de Endereços;
- Unidade de Acesso à Memória;
- Somador de Ponto Flutuante;
- Multiplicador de Ponto Flutuante;
- Barramento de Leitura em Registradores Inteiros;
- Barramento de Escrita em Registradores Inteiros;
- Barramento de Leitura em Registradores de Ponto Flutuante;
- Barramento de Escrita em Registradores de Ponto Flutuante;
- Barramento de Transferência entre Registradores Inteiros e Registradores de Ponto Flutuante;
- Registradores Inteiros para Renomeação;
- Registradores de Ponto Flutuante para Renomeação.

IV.10 O Programa Compact

O programa Compact (vide Apêndice B) implementa o algoritmo de compactação FCFS e, durante sua execução, realiza diversas medidas. Este programa recebe como entrada três arquivos: um contendo um *trace* de um programa de teste (gerado pelo programa Sim860); um outro contendo o código estático deste programa, com cada uma de suas instruções detalhada em seus elementos - sua t-upla - (gerado pelo

Resultado do Programa: LU.C					
Speedup	Instruções	Ciclos	I/C	Ciclos C.	I/C C.
23.188	18110	26063	0.695	1124	16.112

Porcentagem de Cada Tipo de Instrução			
Inteiras	Ponto F.	Desvio	Load/Store
0.677	0.112	0.059	0.149

Utilização de Recurso Por Ciclo				
U.L.A.I.	B.L.R.I.	B.E.R.I.	Som. End.	U.A.M.
10.915	18.065	11.023	3.348	2.399
Som. P.F.	Mul. P.F.	B.L.R.P.F.	B.E.R.P.F.	Bar. T.
0.880	0.920	4.109	3.318	0.059

Tabela IV.2: Informações Contidas no Arquivo de Saída do Programa Compact

programa Scode); e um terceiro contendo o valor dos parâmetros especificando uma configuração da VLIW.

Durante sua execução, o programa Compact faz, na verdade, uma simulação da execução do programa de teste em uma configuração da máquina VLIW. Como saída, o programa Compact gera uma tabela como a Tabela IV.2 . Onde:

Speedup - É a razão entre o número de ciclos requeridos pela execução do programa de teste em uma máquina seqüencial de referência e em uma máquina VLIW com código compactado. A máquina de referência é uma arquitetura constituída por uma unidade funcional de cada tipo. As latências de cada uma dessas unidades são iguais às das unidades do modelo de máquina VLIW. A máquina de referência executa uma instrução por vez e suas unidades funcionais não são *pipelined*;

Instruções - Número de instruções executadas na máquina seqüencial de referência;

Ciclos - Número de ciclos da máquina seqüencial de referência necessário à execução do programa não compactado;

I/C - Instruções / Ciclos;

Ciclos C. - Número de ciclos requeridos pela execução do programa compactado numa configuração de máquina VLIW;

I/C C. - Instruções / Ciclos C.;

Inteiras - Número de instruções lógicas e aritméticas inteiras;

Ponto F. - Número de instruções de soma, multiplicação, recíproco e comparação em ponto flutuante;

Desvio - Número de instruções de desvio (*jumps*, *jumps* condicionais, *calls*, etc);

Load/Store - Número de instruções de leitura e escrita na memória;

U.L.A.I. - Número de Unidades Lógicas e Aritméticas Inteiras;

B.L.R.I. - Número de Barramentos de Leitura em Registradores Inteiros;

B.E.R.I. - Número de Barramentos de Escrita em Registradores Inteiros;

Som. End. - Número de Somadores de Endereços;

U.A.M. - Número de Unidades de Acesso à Memória;

Som. P.F. - Número de Somadores de Ponto Flutuante;

Mul. P.F. - Número de Multiplicadores de Ponto Flutuante;

B.L.R.P.F. - Número de Barramentos de Leitura em Registradores de Ponto Flutuante;

B.E.R.P.F. - Número de Barramentos de Escrita em Registradores de Ponto Flutuante;

Bar. T. - Número de Barramentos de Transferência entre registradores inteiros e de ponto flutuante.

Executando-se diversas vezes o programa Compact tendo como entrada os arquivos de um mesmo programa de teste e diversos arquivos contendo parâmetros, pode-se traçar o perfil da máquina VLIW que melhor se ajusta à execução deste programa de teste.

No próximo capítulo são descritos os experimentos com cada programa de teste e é feita, também, a análise de seus resultados.

@

Capítulo V

EXPERIMENTOS E ANÁLISE DOS RESULTADOS

V.1 Introdução

Para a realização dos experimentos o seguinte procedimento foi seguido:

1. Tradução de um programa de teste com o compilador C da Metaware;
2. Geração de um arquivo de *trace* (através do programa Sim860) e de um arquivo de código estático (através do programa Scode), a partir do programa de teste compilado;
3. Execução do programa Compact utilizando os arquivos de *trace* e código estático gerados no passo anterior, e com arquivo de parâmetros indicando que a VLIW possui recursos ilimitados. Neste passo é determinado o *speedup* máximo que o programa de teste pode atingir;
4. Para cada parâmetro, execução do programa Compact várias vezes, variando o valor do parâmetro de um mínimo, até um ponto em que o *speedup* se iguale ao *speedup* máximo. Os demais parâmetros são considerados ilimitados;

Ao final deste procedimento, obtem-se a configuração da máquina VLIW que melhor se ajusta à execução do programa de teste.

Este procedimento foi utilizado para avaliação de máquinas VLIW para cada um dos programas de teste. Foram feitos também, gráficos mostrando

Resultado do Programa: LIVERMOR.C					
Speedup	Instruções	Ciclos	I/C	Ciclos C.	I/C C.
17.404	13007	17508	0.743	1006	12.929

Porcentagem de Cada Tipo de Instrução			
Inteiras	Ponto F.	Desvio	Load/Store
0.692	0.000	0.154	0.154

Utilização de Recurso Por Ciclo				
U.L.A.I.	B.L.R.I.	B.E.R.I.	Som. End.	U.A.M.
8.952	13.929	10.940	3.977	1.988
Som. P.F.	Mul. P.F.	B.L.R.P.F.	B.E.R.P.F.	Bar. T.
0.000	0.000	0.000	0.000	0.000

Tabela V.1: Compactação com Recursos Ilimitados: LIVERMOR.C

o efeito causado pela variação de cada parâmetro, no *speedup* das diversas configurações da máquina VLIW. Estes gráficos e avaliações são apresentados a seguir.

V.2 Experimento 1: Livermore Loop 24

O Livermore Loop 24 é um algoritmo para a determinação do elemento de menor valor de um vetor de dimensão n. O valor escolhido para n foi 1000. A listagem da implementação deste algoritmo está no Apêndice A, com o nome LIVERMOR.C. A Tabela V.1 mostra o resultado da execução de Compact para uma VLIW com recursos ilimitados. O *speedup* máximo alcançado foi 17.4.

A Figura V.1 apresenta o gráfico da variação do *speedup* com o número de U.L.A.I.s. A partir de 10 U.L.A.I.s o *speedup* não se altera mais, por isso, 10 é o número ideal de U.L.A.I.s para este programa de teste.

Os gráficos apresentados nas figuras seguintes, indicam o valor ideal de cada um dos outros parâmetros. Na Tabela V.2 estão sumarizados os parâmetros da máquina VLIW que mais se adequa a este programa de teste. Comparando os valores dessa tabela com os da Tabela V.1, pode-se observar que os parâmetros ideais para este programa, indicados na Tabela V.2, se aproximam da média de utilização por ciclo de cada recurso correspondente, indicada na Tabela V.1. Isso mostra que

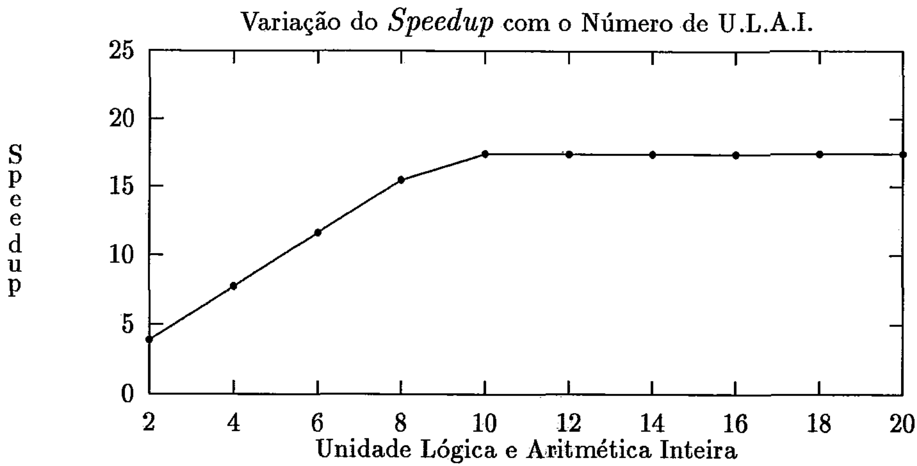


Figura V.1: *Speedup* versus ULAI: LIVERMOR.C

a máquina VLIW ótima para esta implementação do algoritmo Livermor Loop 24, apresenta pouca ociosidade (menor que 15%) de seus recursos durante a execução deste programa, o que é muito desejável. Fazendo-se esta mesma comparação entre as tabelas dos outros experimentos (equivalentes às tabelas V.1 e V.2), pode-se constatar que nem sempre isso acontece. No próximo experimento serão apresentadas possíveis causas de ociosidade de recursos da máquina VLIW durante a execução de programas de aplicação.

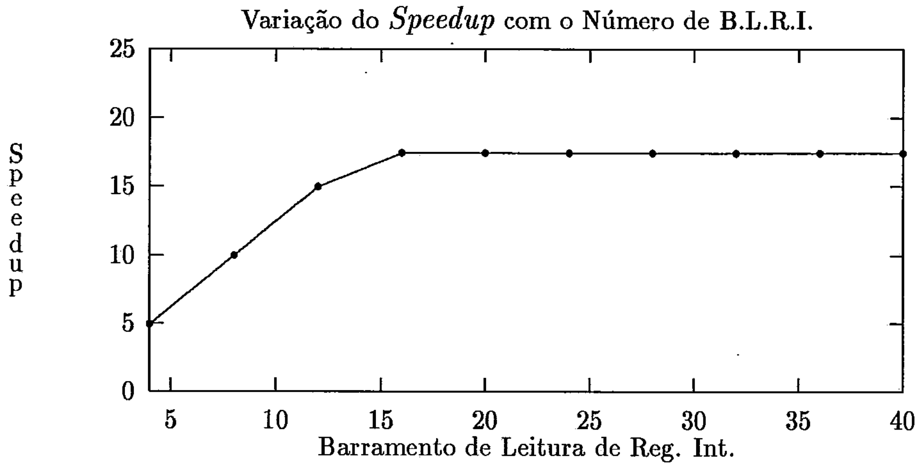


Figura V.2: *Speedup* versus BLRI: LIVERMOR.C

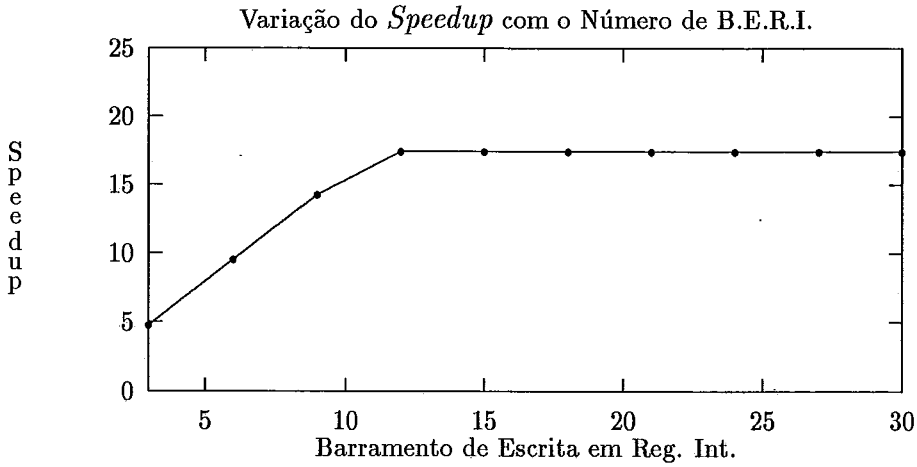


Figura V.3: *Speedup* versus BERI: LIVERMOR.C

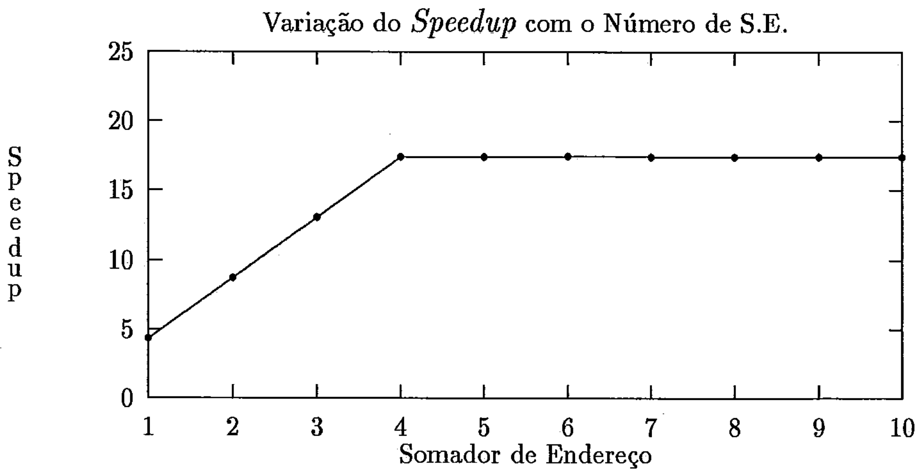


Figura V.4: *Speedup* versus SE: LIVERMOR.C

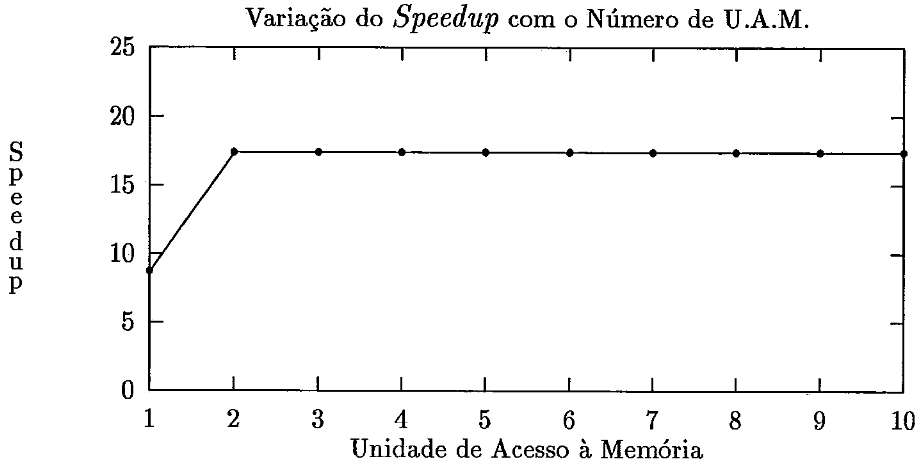


Figura V.5: *Speedup* versus UAM: LIVERMOR.C

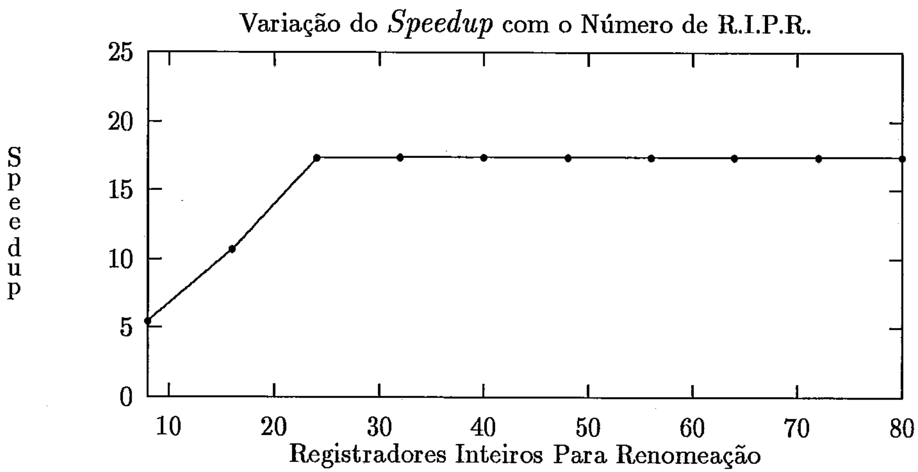


Figura V.6: *Speedup* versus RIPR: LIVERMOR.C

Parâmetro	Quantidade
Unidade Lógica e Aritmética Inteira	10
Barramento de Leitura em Reg. I.	16
Barramento de Escrita em Reg. I.	12
Somador de Endereços	4
Unidade de Acesso à Memória	2
Registradores Inteiros Para Renomeação	24

Tabela V.2: Parâmetros Ideais para o Programa LIVERMOR.C

V.3 Experimento 2: Quick Sort

A implementação do algoritmo de ordenação Quick Sort utilizada nos experimentos está no Apêndice A, com a denominação QUICK.C. O vetor a ser ordenado tem dimensão 200. A Tabela V.3 mostra o resultado da execução de Compact, numa configuração de VLIW com recursos ilimitados. O *speedup* máximo obtido foi 93.1.

A partir dos gráficos apresentados nas Figuras V.7, V.8, V.9, V.10, V.11 e V.12, determinou-se os parâmetros ideais da máquina VLIW para esta implementação do algoritmo Quick Sort. Na Tabela V.4 estão sumarizados estes parâmetros.

Diferente do experimento Livermor Loop 24, apresentado anteriormente, os valores ideais para os parâmetros da máquina VLIW são sensivelmente maiores que a média de utilização por ciclo de cada recurso correspondente, neste experimento. Através do estudo do código compactado é possível observar que isso acontece porque durante a execução ocorre, em alguns trechos, forte concentração de instruções de um mesmo tipo. O *loop do-while* de Quick Sort pode ter muitas ou poucas iterações, de acordo estágio da execução do programa, por exemplo. Para que o desempenho seja máximo, o número de unidades funcionais responsável pela execução dessas instruções deve atender a este caso particular, o que faz com que muitos desses recursos permaneçam ociosos fora dos trechos onde são muito utilizados.

Resultado do Programa: QUICK.C					
Speedup	Instruções	Ciclos	I/C	Ciclos C.	I/C C.
93.056	24428	33035	0.739	355	68.811

Porcentagem de Cada Tipo de Instrução			
Inteiras	Ponto F.	Desvio	Load/Store
0.653	0.000	0.148	0.199

Utilização de Recurso Por Ciclo				
U.L.A.I.	B.L.R.I.	B.E.R.I.	Som. End.	U.A.M.
44.938	83.741	54.839	23.873	13.721
Som. P.F.	Mul. P.F.	B.L.R.P.F.	B.E.R.P.F.	Bar. T.
0.000	0.000	0.000	0.000	0.000

Tabela V.3: Compactação com Recursos Ilimitados: QUICK.C

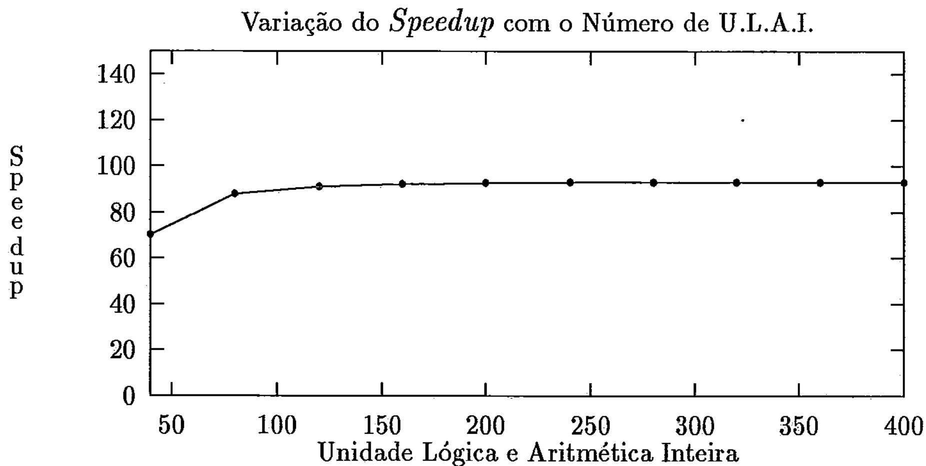


Figura V.7: *Speedup* versus ULAI: QUICK.C

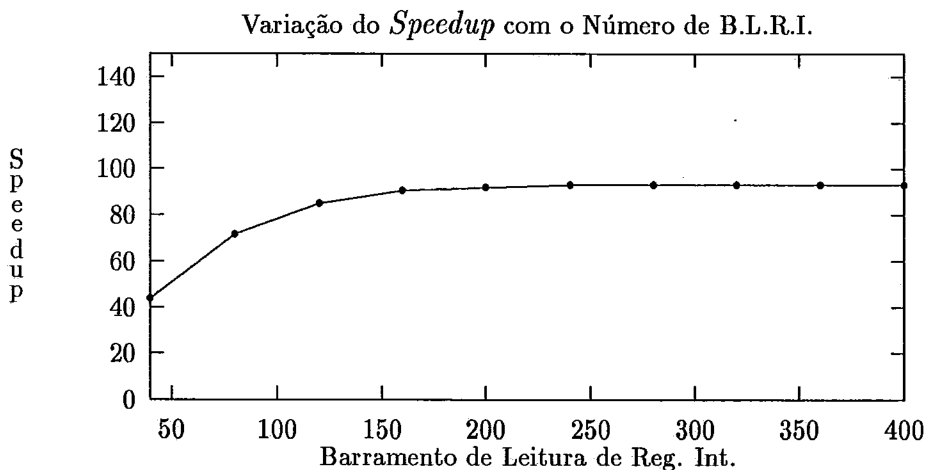


Figura V.8: *Speedup* versus BLRI: QUICK.C

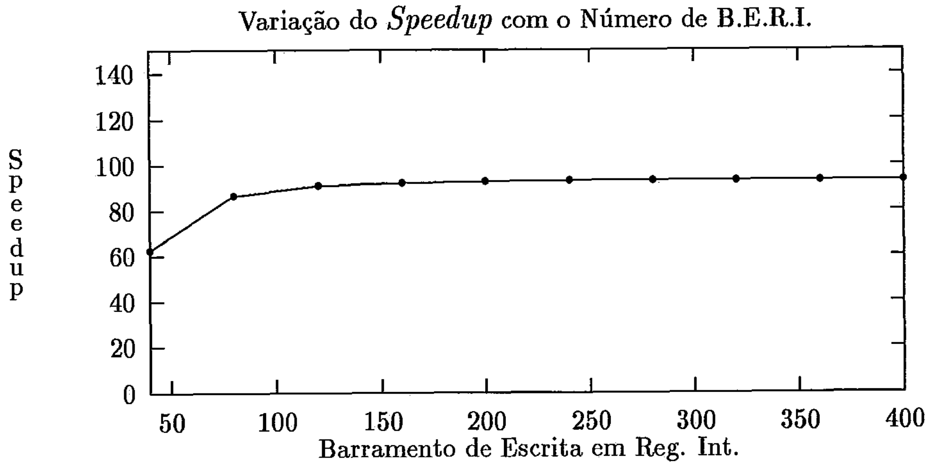


Figura V.9: *Speedup* versus BERI: QUICK.C

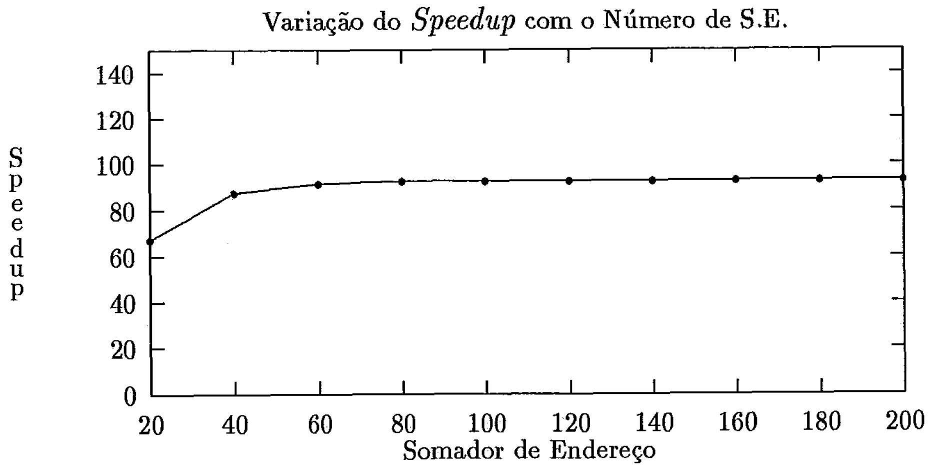


Figura V.10: *Speedup* versus SE: QUICK.C

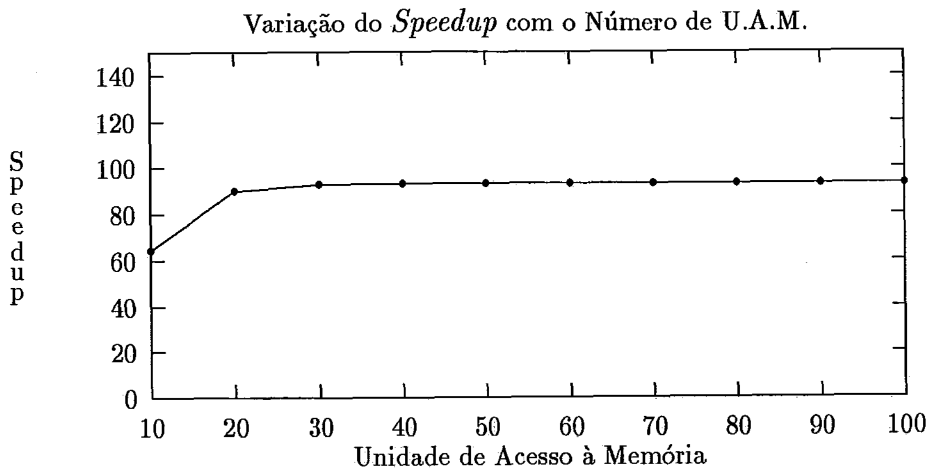


Figura V.11: *Speedup* versus UAM: QUICK.C

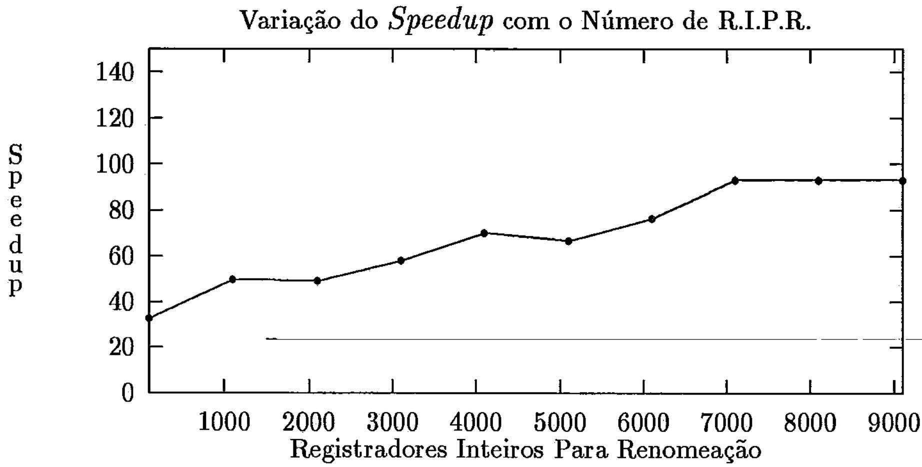


Figura V.12: *Speedup* versus RIPR: QUICK.C

Parâmetro	Quantidade
Unidade Lógica e Aritmética Inteira	240
Barramento de Leitura em Reg. I.	240
Barramento de Escrita em Reg. I.	240
Somador de Endereços	160
Unidade de Acesso à Memória	30
Registradores Inteiros Para Renomeação	7000

Tabela V.4: Parâmetros Ideais para o Programa QUICK.C

V.4 Experimento 3: Busca Binária

A implementação do algoritmo de Busca Binária utilizada nos experimentos está listada no Apêndice A, com o nome BINARIA.C. Nesta implementação o algoritmo é uma função que é chamada pelo programa principal. O programa principal chama esta função dez vezes, cada uma delas para fazer a busca de um dado diferente, em um vetor de dimensão 1000. Através dessa repetição, aumentou-se o número de instruções executadas.

Na configuração com recursos ilimitados, o *speedup* obtido foi 47.8. Os dados de saída de Compact estão indicados na Tabela V.5 . Os gráficos com o efeito da variação de cada parâmetro no *speedup*, podem ser vistos nas Figuras V.13, V.14, V.15, V.16, V.17 e V.18. A Tabela V.6 apresenta os valores ideais para cada parâmetro.

Resultado do Programa: BINARIA.C					
Speedup	Instruções	Ciclos	I/C	Ciclos C.	I/C C.
47.764	1856	2627	0.707	55	33.745

Porcentagem de Cada Tipo de Instrução			
Inteiras	Ponto F.	Desvio	Load/Store
0.677	0.000	0.194	0.129

Utilização de Recurso Por Ciclo				
U.L.A.I.	B.L.R.I.	B.E.R.I.	Som. End.	U.A.M.
26.527	44.455	26.927	10.891	4.345
Som. P.F.	Mul. P.F.	B.L.R.P.F.	B.E.R.P.F.	Bar. T.
0.000	0.000	0.000	0.000	0.000

Tabela V.5: Compactação com Recursos Ilimitados: BINARIA.C

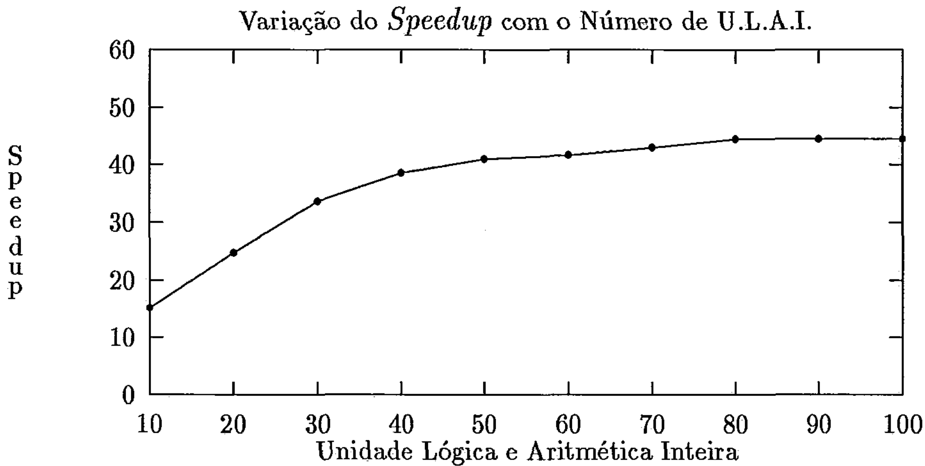


Figura V.13: *Speedup* versus ULAI: BINARIA.C

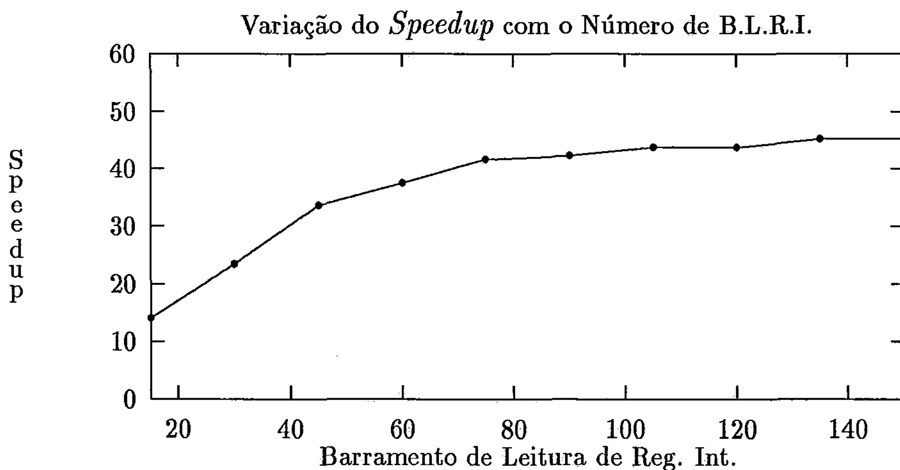


Figura V.14: *Speedup* versus BLRI: BINARIA.C

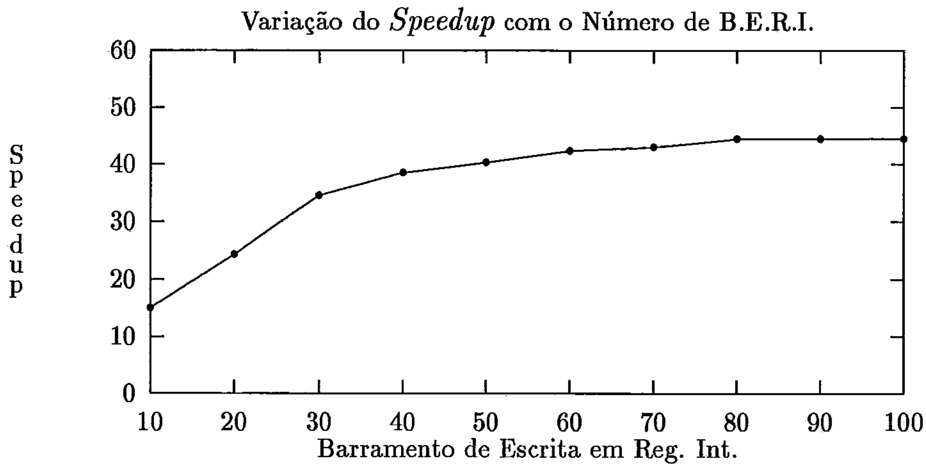


Figura V.15: *Speedup* versus BERI: BINARIA.C

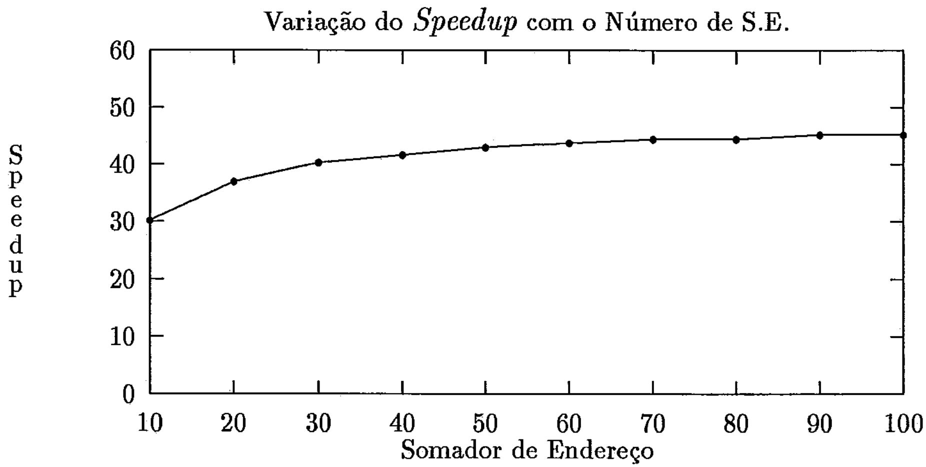


Figura V.16: *Speedup* versus SE: BINARIA.C

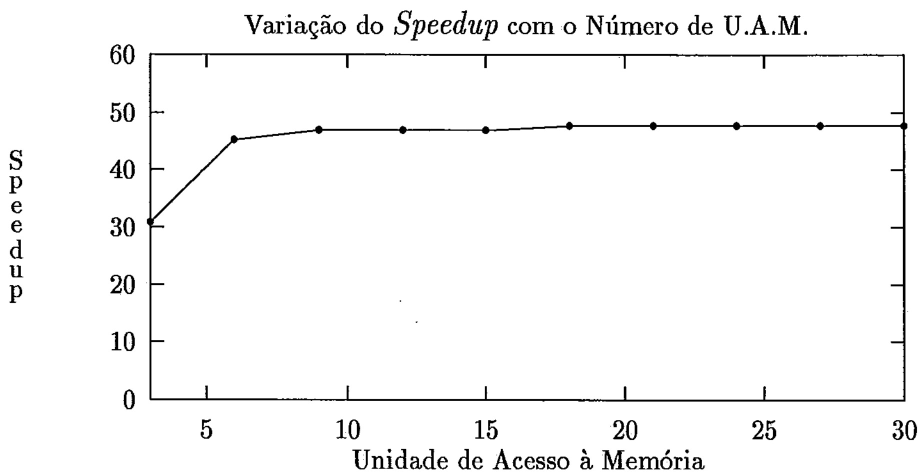


Figura V.17: *Speedup* versus UAM: BINARIA.C

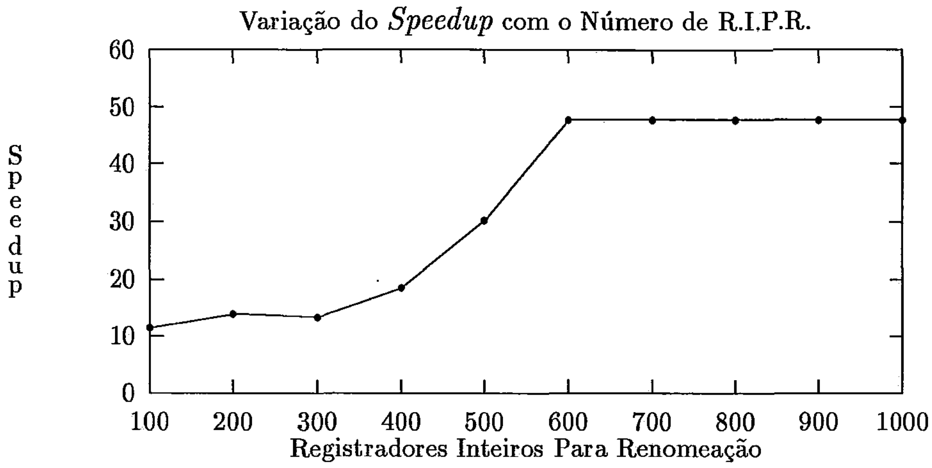


Figura V.18: *Speedup* versus RIPR: BINARIA.C

Parâmetro	Quantidade
Unidade Lógica e Aritmética Inteira	80
Barramento de Leitura em Reg. I.	135
Barramento de Escrita em Reg. I.	80
Somador de Endereços	90
Unidade de Acesso à Memória	18
Registradores Inteiros Para Renomeação	570

Tabela V.6: Parâmetros Ideais para o Programa BINARIA.C

V.5 Experimento 4: Bubble Sort

A listagem da implementação do algoritmo de ordenação Bubble Sort utilizada nos experimentos está no Apêndice A, com o nome BOLHA.C. Nesta implementação é feita a ordenação de um vetor de tamanho 60.

Na Tabela V.7 estão os resultados da execução de Compact na configuração com recursos ilimitados. As Figuras V.19, V.20, V.21, V.22, V.23 e V.24, mostram os gráficos da variação do *speedup* com cada parâmetro. A Tabela V.8, lista os valores ideais de cada parâmetro neste experimento.

Resultado do Programa: BOLHA.C					
Speedup	Instruções	Ciclos	I/C	Ciclos C.	I/C C.
99.332	21419	29303	0.731	295	72.607

Porcentagem de Cada Tipo de Instrução			
Inteiras	Ponto F.	Desvio	Load/Store
0.633	0.000	0.151	0.215

Utilização de Recurso Por Ciclo				
U.L.A.I.	B.L.R.I.	B.E.R.I.	Som. End.	U.A.M.
46.125	83.149	56.705	26.617	15.631
Som. P.F.	Mul. P.F.	B.L.R.P.F.	B.E.R.P.F.	Bar. T.
0.000	0.000	0.000	0.000	0.000

Tabela V.7: Compactação com Recursos Ilimitados: BOLHA.C

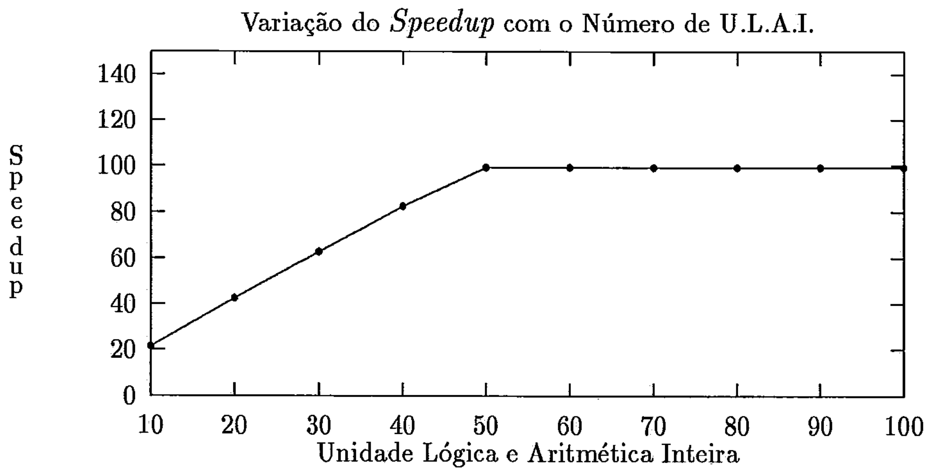


Figura V.19: *Speedup* versus ULAI: BOLHA.C

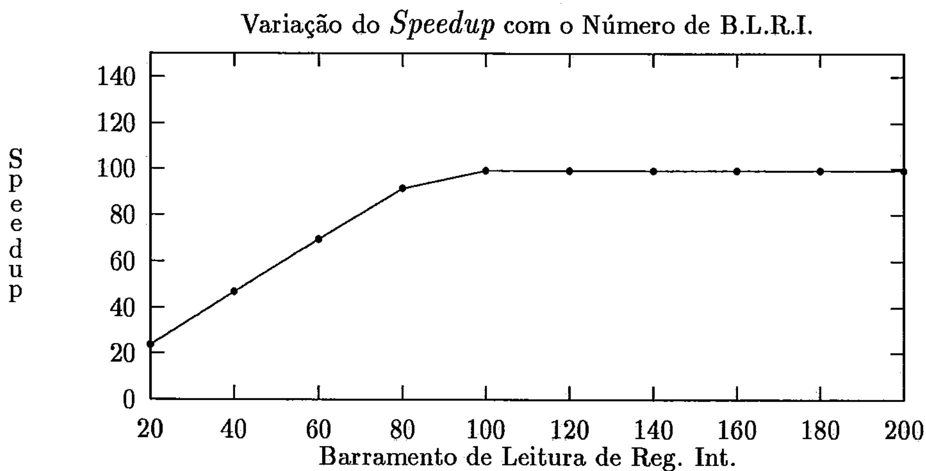


Figura V.20: *Speedup* versus BLRI: BOLHA.C

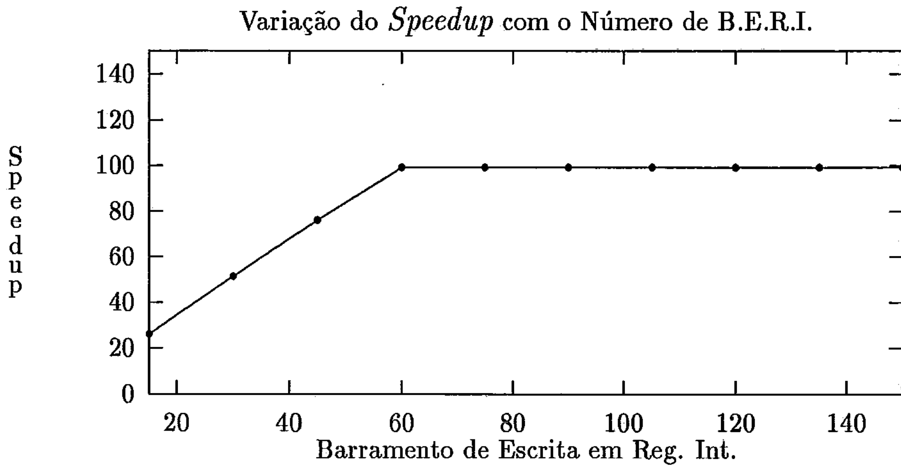


Figura V.21: *Speedup* versus BERI: BOLHA.C

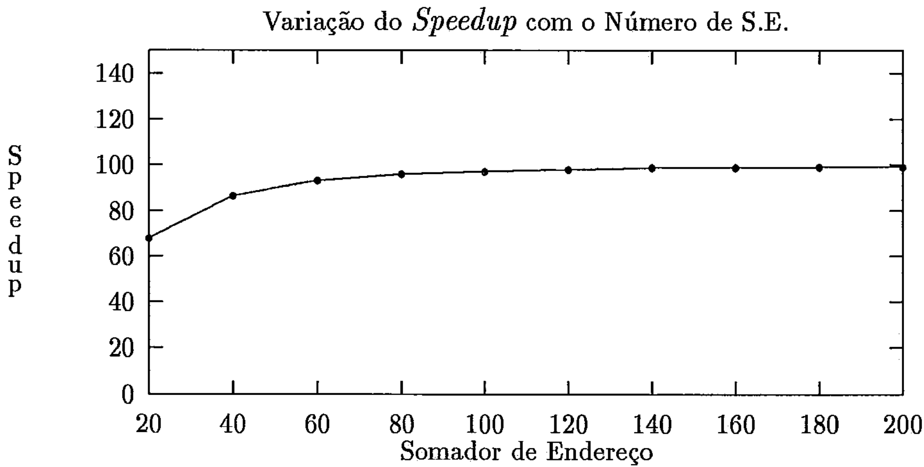


Figura V.22: *Speedup* versus SE: BOLHA.C

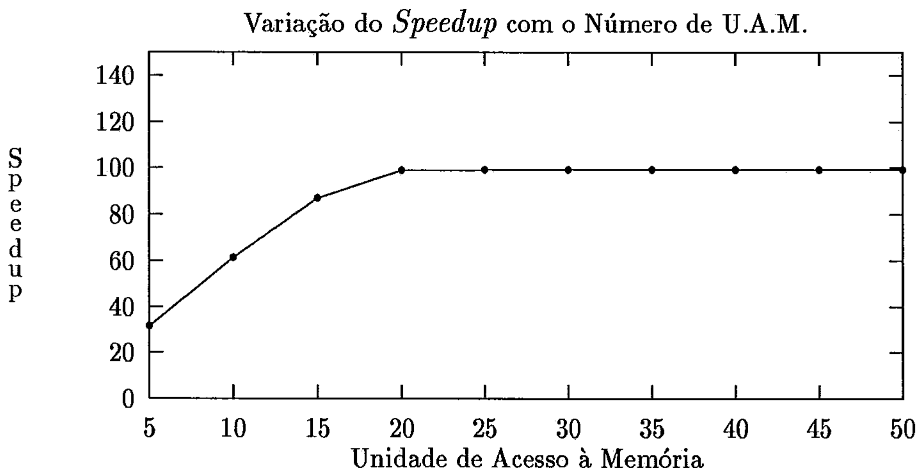


Figura V.23: *Speedup* versus UAM: BOLHA.C

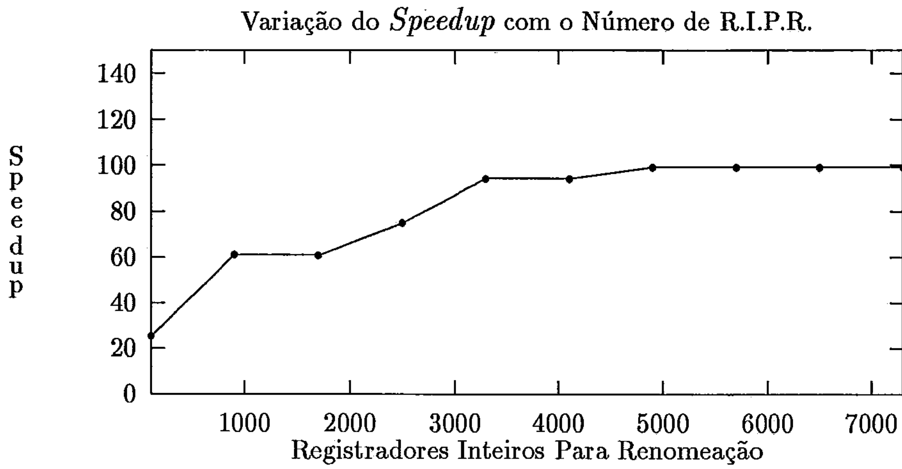


Figura V.24: *Speedup* versus RIPR: BOLHA.C

Parâmetro	Quantidade
Unidade Lógica e Aritmética Inteira	50
Barramento de Leitura em Reg. I.	100
Barramento de Escrita em Reg. I.	60
Somador de Endereços	140
Unidade de Acesso à Memória	20
Registradores Inteiros Para Renomeação	4800

Tabela V.8: Parâmetros Ideais para o Programa BOLHA.C

V.6 Experimento 5: Decomposição LU

O algoritmo de Decomposição LU é uma aplicação tipicamente numérica. Nele estão presentes operações de soma, subtração, comparação, multiplicação e divisão de números em ponto flutuante. As operações contidas nele se restringem basicamente a estas e a operações com índices. No Apêndice A pode ser vista, com o nome LU.C, a listagem da implementação deste algoritmo. Uma matriz com 12x12 componentes foi utilizada nessa implementação.

Os resultados obtidos a partir da execução de Compact numa configuração com recursos ilimitados, podem ser vistos na Tabela V.9 . Os gráficos indicando a variação do *speedup* com cada parâmetro, estão nas Figuras V.25, V.26, V.27, V.28, V.29, V.30, V.31, V.32, V.33, V.34, V.35 e V.36.

Resultado do Programa: LU.C					
Speedup	Instruções	Ciclos	I/C	Ciclos C.	I/C C.
47.387	18110	26063	0.695	550	32.927

Porcentagem de Cada Tipo de Instrução			
Inteiras	Ponto F.	Desvio	Load/Store
0.677	0.112	0.059	0.149

Utilização de Recurso Por Ciclo				
U.L.A.I.	B.L.R.I.	B.E.R.I.	Som. End.	U.A.M.
22.307	36.918	22.527	6.842	4.902
Som. P.F.	Mul. P.F.	B.L.R.P.F.	B.E.R.P.F.	Bar. T.
1.798	1.880	8.396	6.780	0.120

Tabela V.9: Compactação com Recursos Ilimitados: LU.C

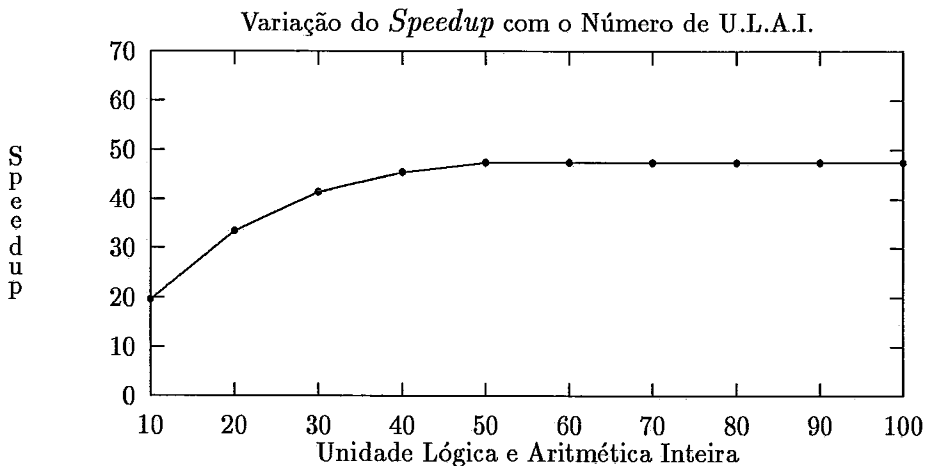


Figura V.25: *Speedup* versus ULAI: LU.C

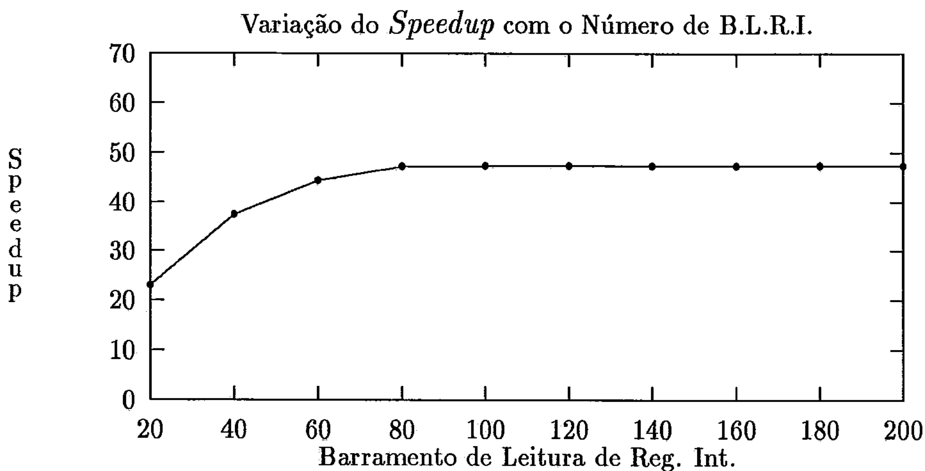


Figura V.26: *Speedup* versus BLRI: LU.C

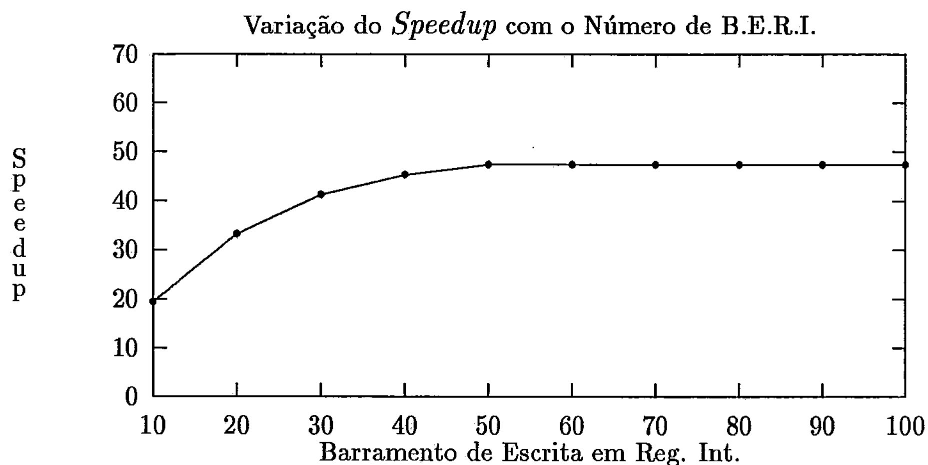


Figura V.27: *Speedup* versus BERI: LU.C

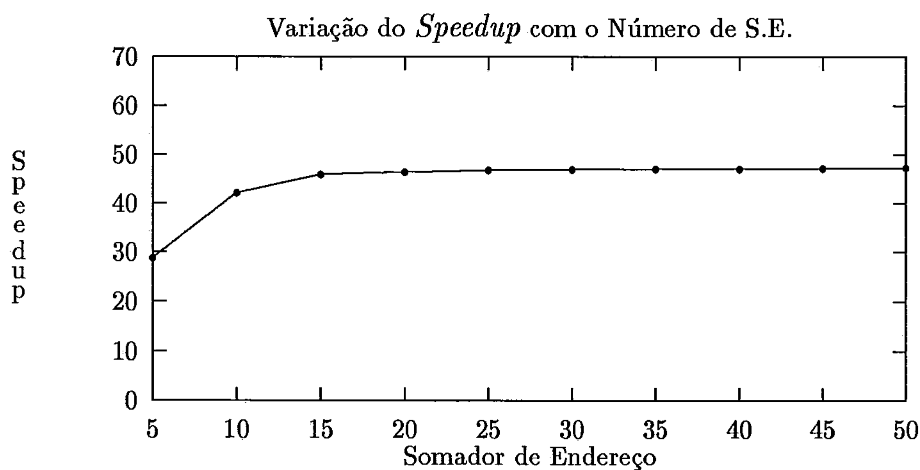


Figura V.28: *Speedup* versus SE: LU.C

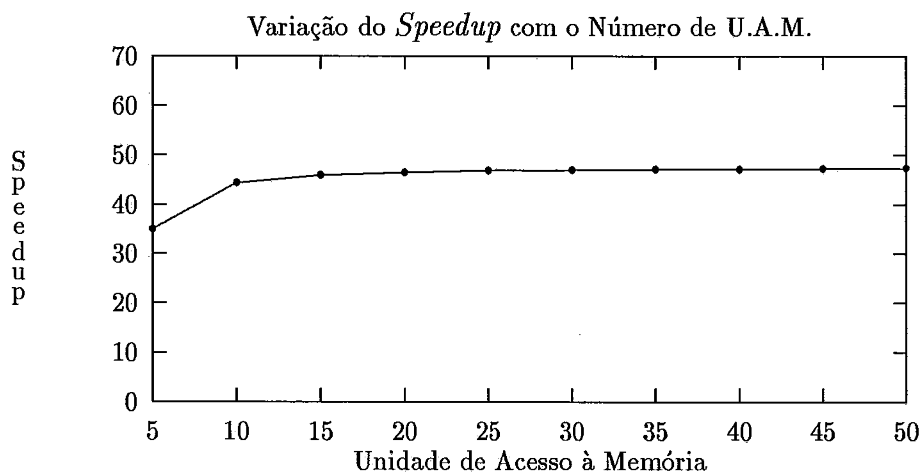


Figura V.29: *Speedup* versus UAM: LU.C

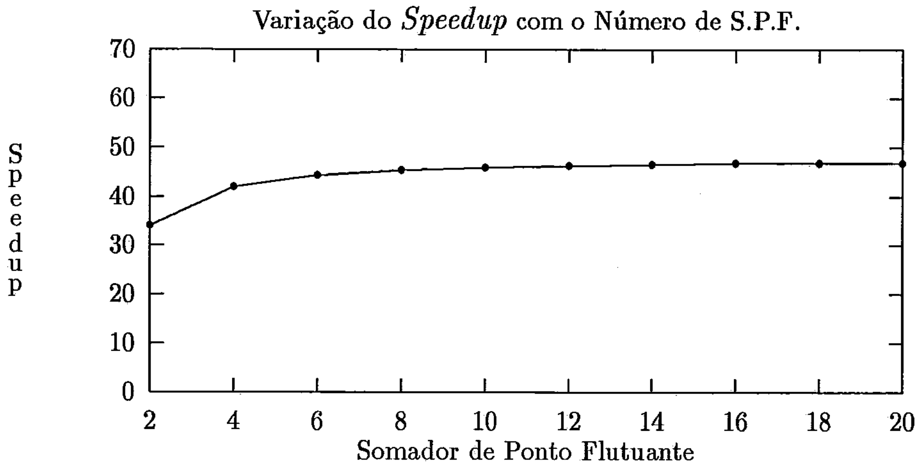


Figura V.30: *Speedup* versus SPF: LU.C

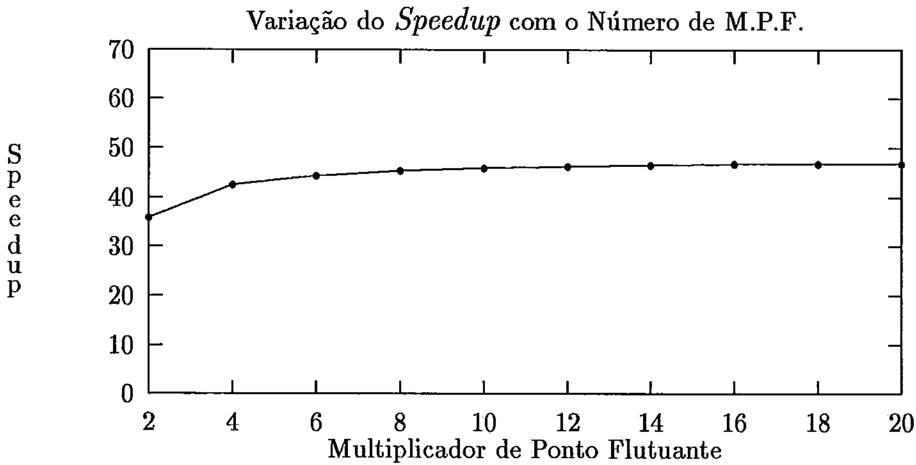


Figura V.31: *Speedup* versus MPF: LU.C

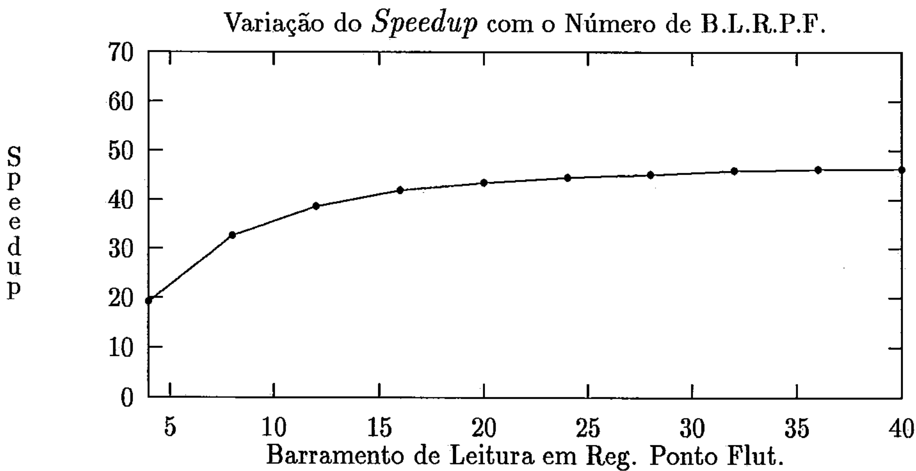


Figura V.32: *Speedup* versus BLRPF: LU.C

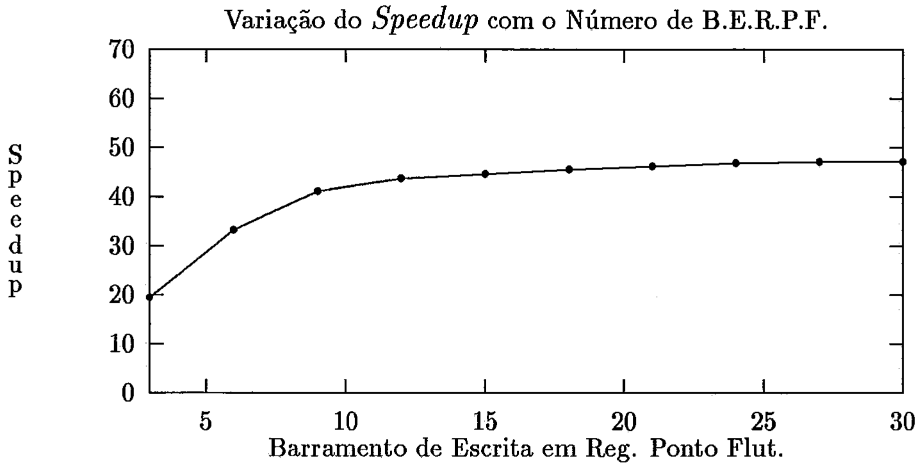


Figura V.33: *Speedup* versus BERPF: LU.C

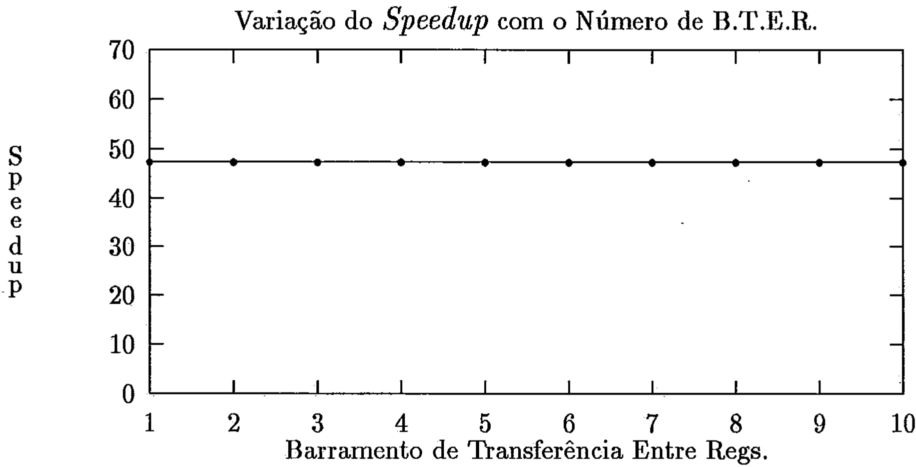


Figura V.34: *Speedup* versus BTER: LU.C

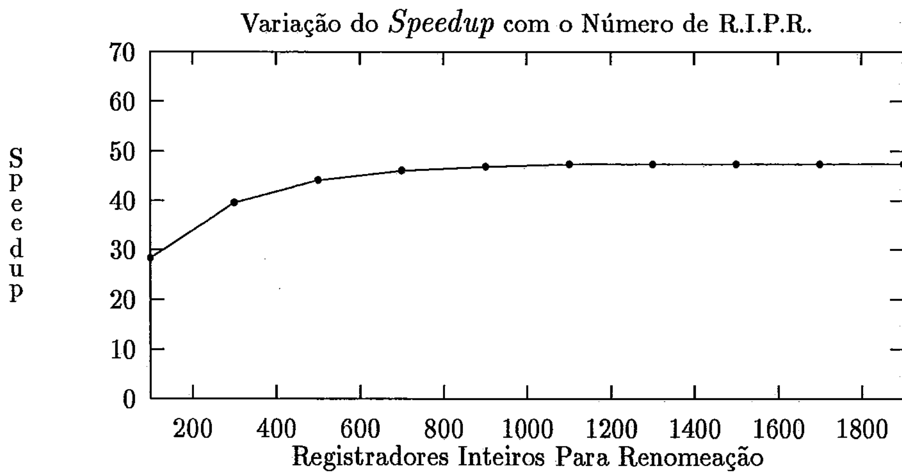


Figura V.35: *Speedup* versus RIPR: LU.C

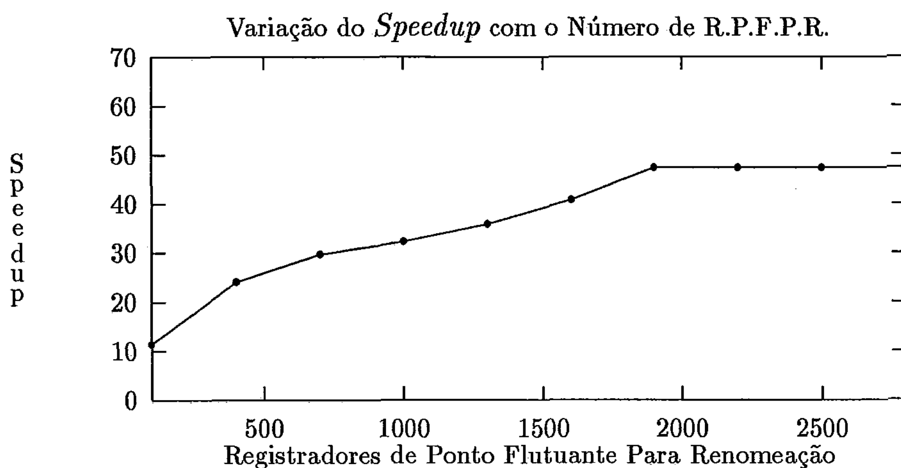


Figura V.36: *Speedup* versus RPFPR: LU.C

Parâmetro	Quantidade
Unidade Lógica e Aritmética Inteira	50
Barramento de Leitura em Reg. I.	100
Barramento de Escrita em Reg. I.	50
Somador de Endereços	30
Unidade de Acesso à Memória	35
Somador de Ponto Flutuante	16
Multiplicador de Ponto Flutuante	16
Barramento de Leitura em Reg. P.F.	35
Barramento de Escrita em Reg. P.F.	28
Barramento de Transferência Entre Regs.	1
Registradores Inteiros Para Renomeação	1100
Registradores de Ponto. Flut. Para Renomeação	1800

Tabela V.10: Parâmetros Ideais para o Programa LU.C

V.7 Experimento 6: Integracão Numérica

A listagem da implementação do algoritmo de Integração Numérica (método dos trapézios) está no Apêndice A, com o nome INTEGRAL.C. Nesta implementação é realizada a integração da função x^2 no intervalo [0,1]. A altura escolhida para os trapézios foi 0.005. Neste programa aparentemente não existem operações com inteiros, já que nenhuma variável inteira é declarada. No entanto, 21.9% das operações executadas são inteiras (vide Tabela V.11). Através da análise do código compactado, pode-se constatar que isso se deve à manipulação de constantes em ponto flutuante. O conjunto de instruções do i860 não possui instruções para transferir constantes imediatas para os registradores de ponto flutuante. Estas constantes devem ser montadas nos registradores inteiros e, só então, transferidas para os registradores de ponto flutuante. As operações envolvendo constantes de ponto flutuante são responsáveis pela quantidade de instruções inteiras indicada no experimento.

A Tabela V.11 contém os dados extraídos da execução de Compact numa máquina com recursos ilimitados. Os gráficos indicando a variação do *speedup* com cada parâmetro podem ser vistos nas Figuras V.37, V.38, V.39, V.40, V.41, V.42, V.43, V.44, V.45, V.46, V.47 e V.48.

Resultado do Programa: INTEGRAL.C					
Speedup	Instruções	Ciclos	I/C	Ciclos C.	I/C C.
19.588	6384	12164	0.525	621	10.280

Porcentagem de Cada Tipo de Instrução			
Inteiras	Ponto F.	Desvio	Load/Store
0.219	0.375	0.156	0.000

Utilização de Recurso Por Ciclo				
U.L.A.I.	B.L.R.I.	B.E.R.I.	Som. End.	U.A.M.
2.256	5.156	2.899	1.607	0.003
Som. P.F.	Mul. P.F.	B.L.R.P.F.	B.E.R.P.F.	Bar. T.
2.565	1.282	8.333	5.456	2.570

Tabela V.11: Compactação com Recursos Ilimitados: INTEGRAL.C

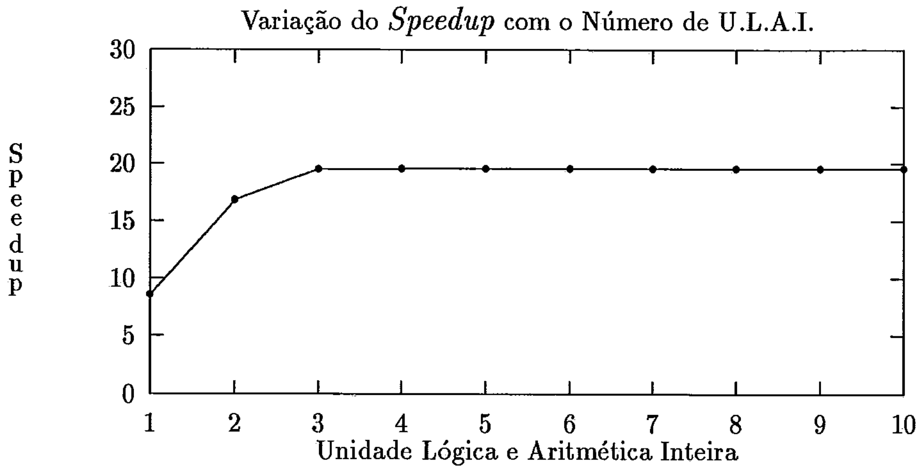


Figura V.37: *Speedup* versus ULAI: INTEGRAL.C

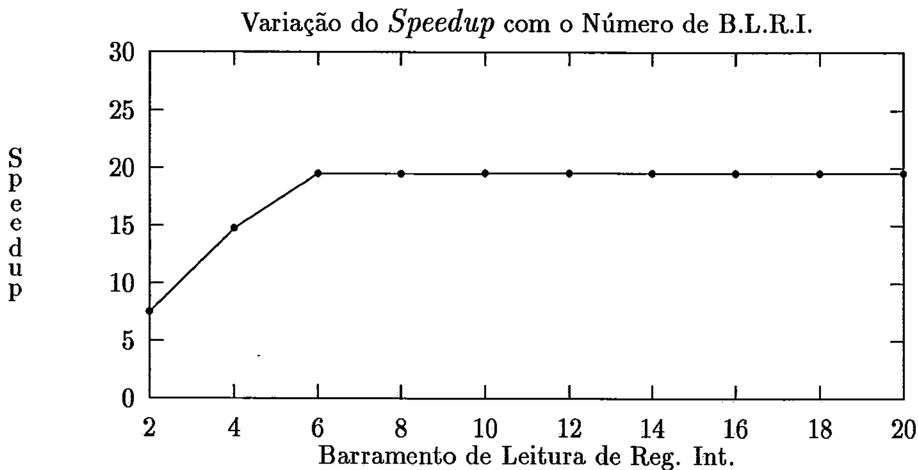


Figura V.38: *Speedup* versus BLRI: INTEGRAL.C

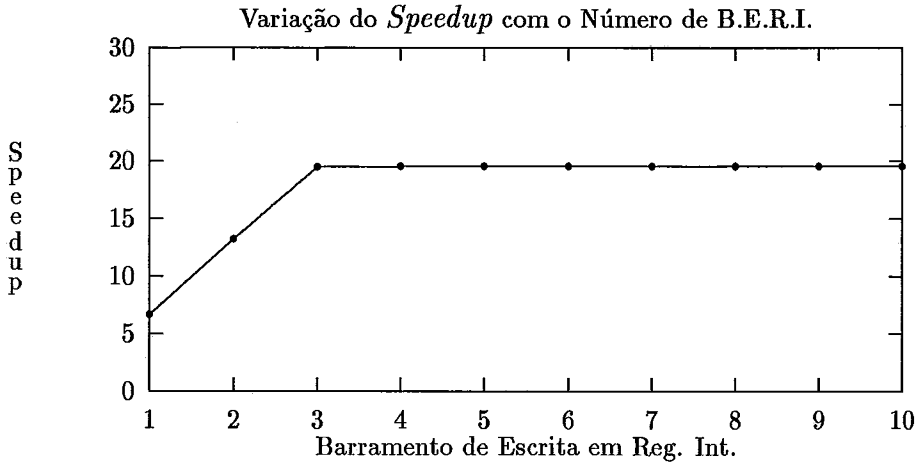


Figura V.39: *Speedup* versus BERI: INTEGRAL.C

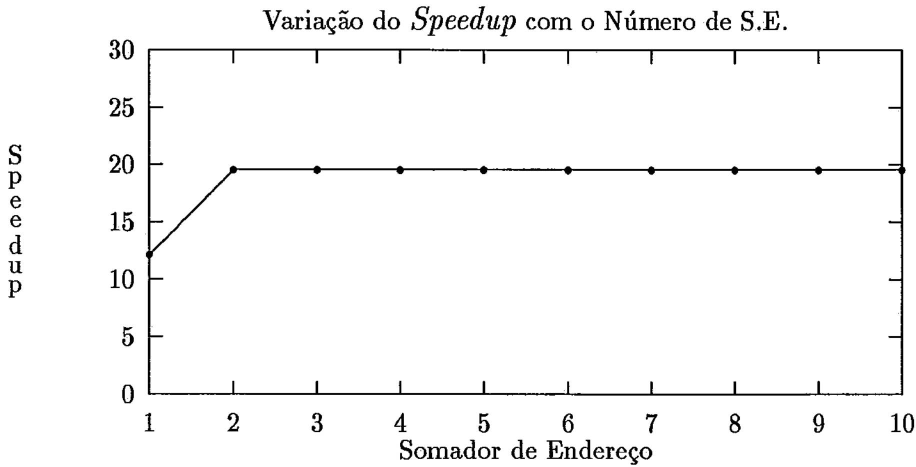


Figura V.40: *Speedup* versus SE: INTEGRAL.C

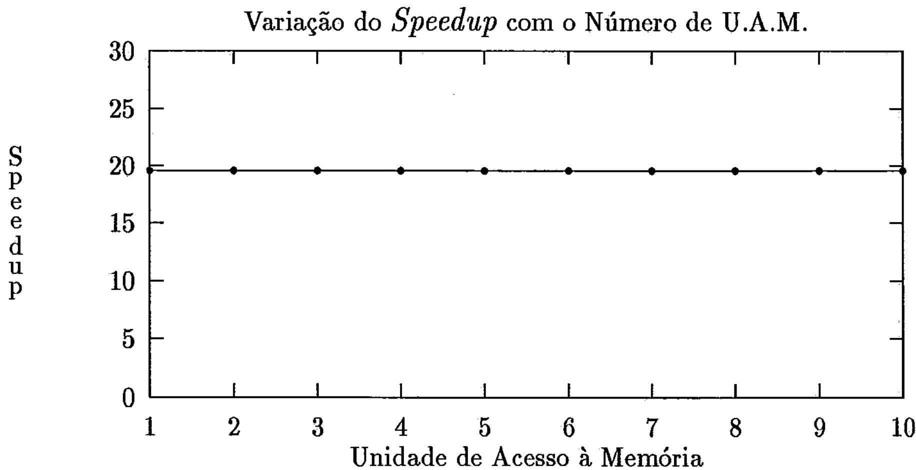


Figura V.41: *Speedup* versus UAM: INTEGRAL.C

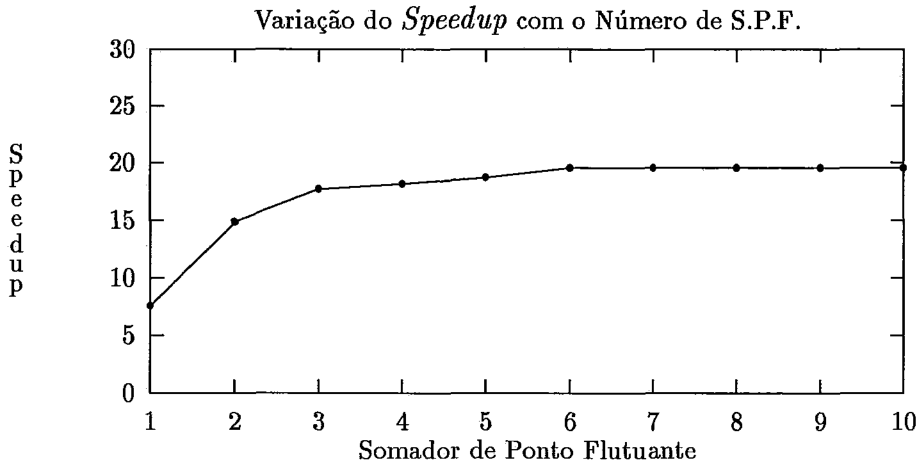


Figura V.42: *Speedup* versus SPF: INTEGRAL.C

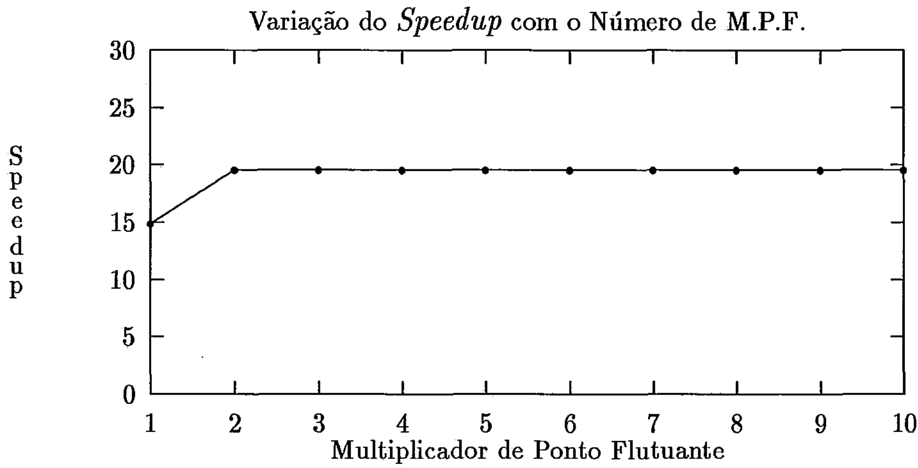


Figura V.43: *Speedup* versus MPF: INTEGRAL.C

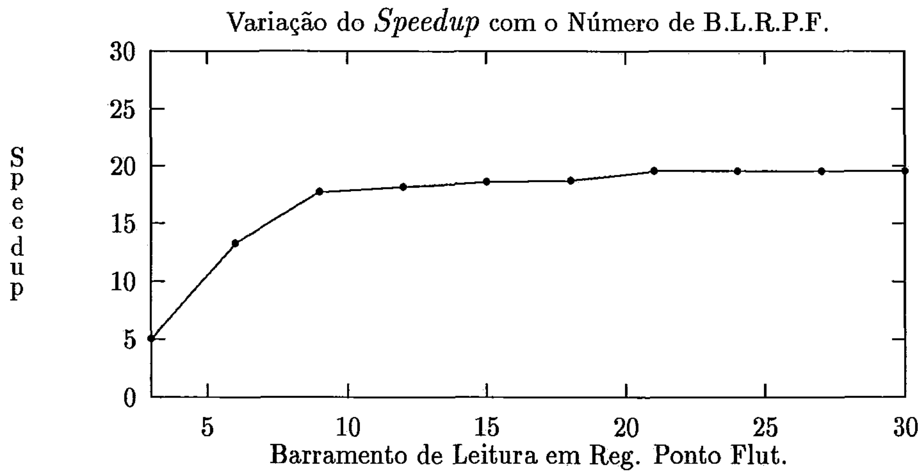


Figura V.44: *Speedup* versus BLRPF: INTEGRAL.C

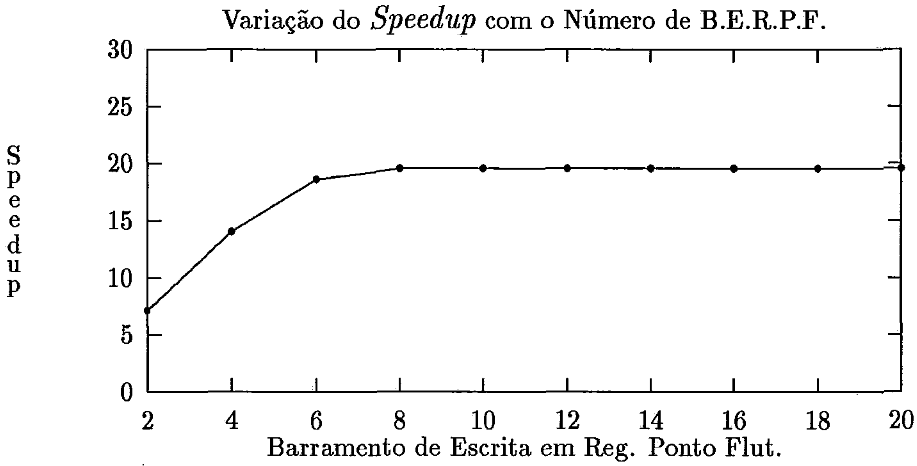


Figura V.45: *Speedup* versus BERPF: INTEGRAL.C

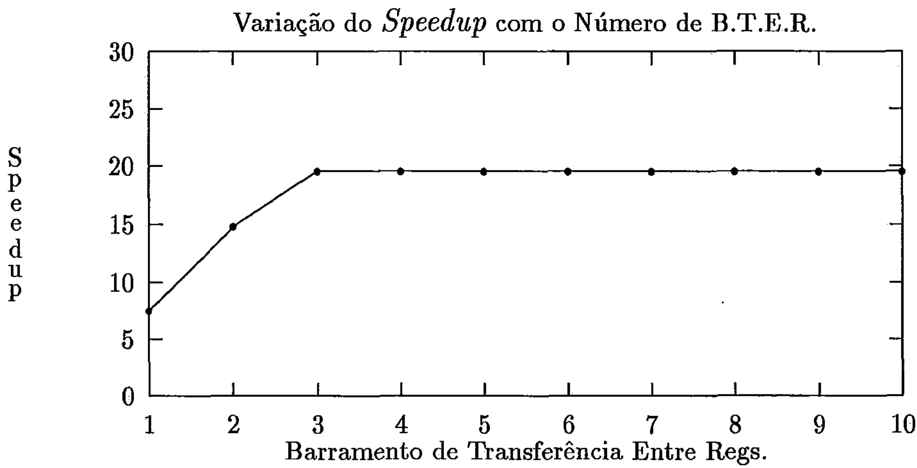


Figura V.46: *Speedup* versus BTER: INTEGRAL.C

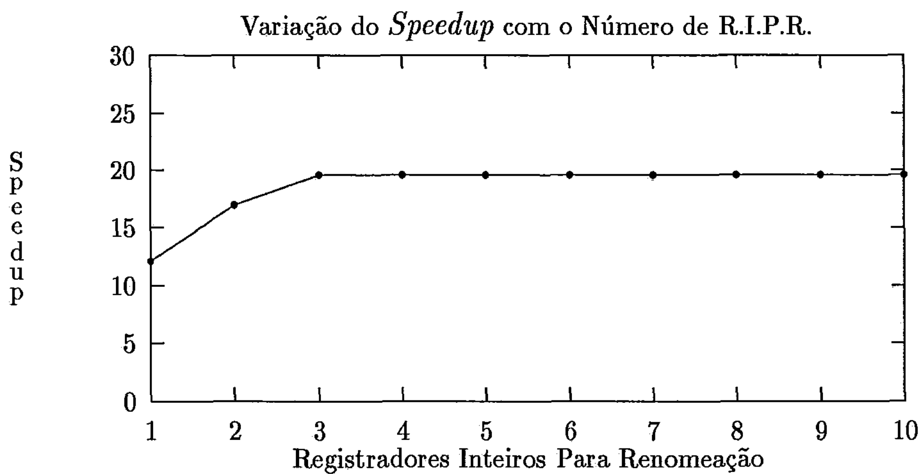


Figura V.47: *Speedup* versus RIPR: INTEGRAL.C

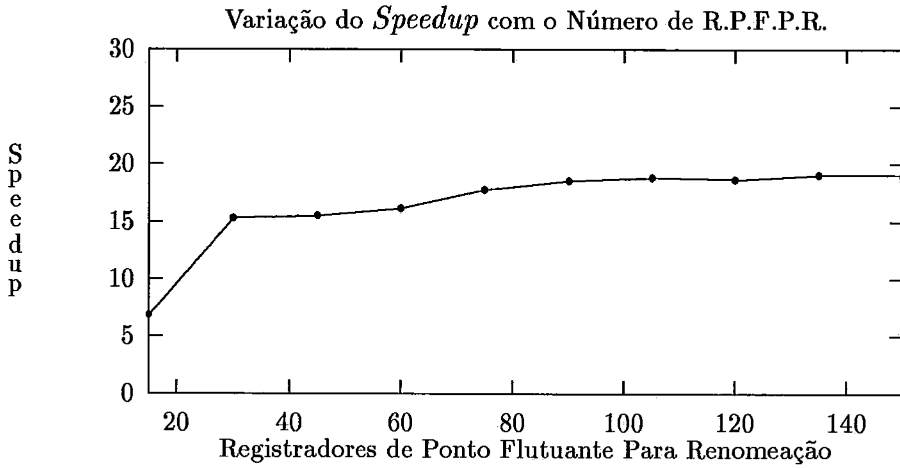


Figura V.48: *Speedup* versus RPFPR: INTEGRAL.C

Parâmetro	Quantidade
Unidade Lógica e Aritmética Inteira	3
Barramento de Leitura em Reg. I.	6
Barramento de Escrita em Reg. I.	3
Somador de Endereços	2
Unidade de Acesso à Memória	1
Somador de Ponto Flutuante	6
Multiplicador de Ponto Flutuante	2
Barramento de Leitura em Reg. P.F.	23
Barramento de Escrita em Reg. P.F.	8
Barramento de Transferência Entre Regs.	3
Registradores Inteiros Para Renomeação	3
Registradores de Ponto. Flut. Para Renomeação	135

Tabela V.12: Parâmetros Ideais para o Programa INTEGRAL.C

V.8 Análise dos Resultados

Como se pode observar nas tabelas apresentadas nas seções anteriores deste capítulo, não existe uma máquina VLIW que possa ser utilizada de forma ótima por todos os programas de teste. Normalizando os parâmetros com relação ao número de ULAIs isto fica mais claro, como pode ser visto na Tabela V.13 . Mas, esta tabela também indica que existem relações entre os diversos parâmetros.

Existe uma relação entre o número de ULAIs, de BLRIs e de BERIs. Para cada ULAI de uma configuração, devem ser introduzidos 2 BLRIs, ou um número um pouco menor que 2 caso existam muitas ULAIs (mais que cinco ULAIs), e 1 BERI, ou um número um pouco maior, no mesmo caso. Estas relações existem por um motivo aparentemente óbvio: as ULAIs operam normalmente com dois operandos de entrada e um de saída. Como para algumas operações é necessário apenas um operando de entrada, o número de BLRIs tende a ser menor que 2 para cada ULAI. Por outro lado, outras unidades funcionais podem escrever nos registradores inteiros (os Somadores de Endereço, por exemplo), e por essa razão, o número de BERIs tende a ser maior que 1 para cada ULAI. Embora essas relações sejam aparentemente evidentes, elas poderiam ser diferentes. Outras unidades funcionais também fazem leituras nos registradores inteiros e isso poderia levar, por exemplo, a relação entre ULAIs e BLRIs para um número superior a 2. Contudo não é isso que mostraram os experimentos. Do mesmo modo, as ULAIs nem sempre escrevem nos registradores (no caso de operações de teste, por exemplo), e isso poderia levar a relação entre ULAIs e BERIs para um número inferior a 1. No entanto, os experimentos indicaram que este número é igual ou superior a 1.

Essas relações são válidas também para as unidades de ponto flutuante, com a diferença de que existem dois tipos de unidades funcionais, específicas para ponto flutuante, que fazem leituras e escritas nos registradores de ponto flutuante. Conforme constatado pelos experimentos, o número de acessos (ao banco de registradores de ponto flutuante) é praticamente igual para os dois tipos de dispositivos funcionais (com uma ligeira predominância para os SPFs). Neste caso, normalizando pelo número de SPFs, para cada SPF devem ser incluídos oito BLRPF,

-	LIVERMOR	QUICK	BINARIA	BOLHA	LU	INTEGRAL
ULAI	1	1	1	1	1	1
BLRI	1.60	1	1.69	2	2	2
BERI	1.20	1	1	1.20	1	1
SE	0.40	0.67	1.13	2.80	0.60	0.67
UAM	0.20	0.13	0.23	0.40	0.70	0.33
SPF					0.32	2.00
MPF					0.32	0.67
BLRPF					0.70	7.67
BERPF					0.56	2.67
BTER					0.02	1
RIPR	2.40	29.17	7.13	96.00	22.00	1
RPFPR					36.00	45.00
SPEEDUP	17.40	93.06	47.76	99.33	47.39	19.59

Tabela V.13: Relação Entre o Valor de Cada Parâmetro e o Número de ULAIs

ou um número um pouco menor, e dois BERPF ou um número também um pouco menor. É difícil determinar qual é a relação entre SPFs e MPFs com apenas dois programas de teste, mas, conforme mostrado pelos experimentos, deve-se ter um número entre 1 e 2 SPFs para cada MPF.

A relação entre ULAIs e SEs é uma das que mais varia de caso para caso. No entanto, observando os gráficos, vemos que o parâmetro SE evolui de forma muito suave nos casos onde o número de SEs comparado com o número de ULAIs é muito grande. Através da observação dos programas compactados, verificou-se que isso ocorre porque, em certos trechos dos programas podem ocorrer fortes concentrações de instruções que usam o mesmo tipo de unidade funcional, como é o caso das SEs. Assim, a redução do número de unidades funcionais desse tipo afeta apenas ligeiramente o *speedup*, afastando-o do ponto ótimo. Aceitando perdas de 5% a 10% no *speedup*, o número de SEs com relação ao de ULAIs poderia ser de 0.6 SEs para cada ULAI.

A relação entre o número de UAMs e o de ULAIs também varia bastante conforme o programa de teste e pelo mesmo motivo que a relação entre o número de SEs e o de ULAIs (embora não varie tanto quanto esta). Uma boa relação entre o número destas duas unidades funcionais seria de 0.3 UAM para cada ULAI, como indicaram os experimentos.

Prog.	LIVERMOR	QUICK	BINARIA	BOLHA	LU	INTEGRAL
Speedup	12.68	13.30	9.59	13.89	15.40	18.07

Tabela V.14: *Speedup* da Máquina TRACE com cada programa de teste

A relação entre o número de BTERs e o número ULAIs também varia muito nos dois experimentos em que os BTERs são utilizados, mas por um motivo completamente diferente. O conjunto de instruções do i860 não contém instruções para transferir constantes imediatas para os registradores de ponto flutuante. No i860 estas constantes devem ser montadas nos registradores inteiros e posteriormente transferidas para os registradores de ponto flutuante. No programa INTEGRAL, onde o número necessário de BTERs é grande (três) se comparado ao de ULAIs (também três), existem três constantes que devem ser transferidas para registradores de ponto flutuante: INCR, LIM e 2. O compilador faz estas transferências a cada passo do *loop* de INTEGRAL, o que gera um grande número de transferências entre registradores, conforme indicado pelos experimentos. No caso do programa LU, o número de transferências entre registradores é muito menor e, por conseguinte, a relação entre número de BTERs e o de ULAIs também.

O número absoluto de registradores para renomeação, de inteiros e de ponto flutuante, a princípio parece muito grande, mas, o seu número normalizado pelo número de ULAIs mostra que os totais obtidos nos experimentos estão dentro do que seria previsto. Este número normalizado nada mais é do que o número de registradores que devem ser incluídos para cada nova ULAI de uma VLIW. O número médio de registradores que devem ser introduzidos por ULAI está bem próximo do tamanho do banco de registradores dos microprocessadores de 32 bits (usualmente 32 registradores).

Com o objetivo de avaliar o desempenho de uma mesma configuração de VLIW na execução de todos os programas de teste, foram produzidos os dados apresentados na Tabela V.14 . Os parâmetros da VLIW utilizada são uma aproximação dos que definem a TRACE [23] - uma máquina VLIW comercial produzida pela Multiflow Computer Inc. - e estão indicados na Tabela V.15 .

O que salta aos olhos nos resultados dos experimentos é a quantidade

Quantidade	Parâmetro
8	Unidades Lógicas e Aritméticas Inteiras
4	Somadores de Ponto Flutuante
4	Multiplicadores de Ponto Flutuante
16	Barramentos de leitura em Registrador I.
8	Barramentos de escrita em Registrador I.
16	Barramentos de leitura em Registrador P. F.
8	Barramentos de escrita em Registrador P. F.
4	Somadores de Endereço
8	Canais de Acesso à Memória
4	Barr. de Transf. entre Reg. Int. e P.F.
288	Registradores Inteiros para Rename
288	Registradores P. F. para Rename

Tabela V.15: Parâmetros da Máquina TRACE

de paralelismo disponível nos programas. Um *speedup* de 99 é, sem dúvida, um *speedup* expressivo e pode ser conseguido por máquinas com arquitetura VLIW.

Capítulo VI

CONCLUSÃO

Esta tese apresentou uma avaliação do impacto de importantes parâmetros arquiteturais no desempenho de Arquiteturas do tipo VLIW.

Interpretando o código objeto derivado de um conjunto de programas de teste, foram produzidos arquivos contendo *traces* da execução de cada um dos componentes da bateria de testes e, utilizando tradicionais técnicas de compactação de código, os arquivos de *trace* foram "paralelizados". Esse processo de compactação leva em conta as dependências entre as instruções do programa de teste e os recursos da máquina VLIW.

O modelo de máquina básica empregado nos experimentos representa uma família de arquiteturas VLIW, que são especificadas através de parâmetros. Esses parâmetros indicam o número e mistura de unidades funcionais constituindo o processador VLIW. Variando cada um desses parâmetros, define-se um novo membro da família.

Para cada programa de teste e para cada configuração da máquina VLIW, foi gerado um código compactado específico.

Com o objetivo de avaliar a taxa de aceleração (isto é, o *speedup*) obtida pelas diversas configurações da máquina básica, empregou-se como referência uma arquitetura constituída por uma unidade de cada tipo. As instruções nessa máquina de referência são executadas seqüencialmente, e o tempo de latência de suas unidades funcionais é o mesmo como na arquitetura VLIW.

Para cada componente da bateria de testes, avaliou-se o impacto no desempenho provocado pela variação de cada um dos parâmetros individualmente (número de unidades funcionais, de barramentos, de registradores utilizados durante o processo de renomeação, etc). Através desses resultados, obteve-se a mistura ideal de componentes que deve constituir a arquitetura VLIW de modo a minimizar o tempo de execução de cada programa de teste.

Examinando os resultados dos experimentos, verifica-se que taxas de aceleração com valores próximos de 100 foram obtidas por algumas configurações do modelo VLIW.

É importante observar que, apesar de extraordinário, esse desempenho das máquinas VLIW representa um limite superior da taxa de aceleração que pode ser atingida durante a execução de cada programa de teste: a utilização do arquivo de *trace* permitiu a remoção das incertezas provocadas pelos comandos de desvio condicional (o endereço da instrução sucessora de um comando de ramificação condicional sempre é conhecido durante o processo de compactação). Por esse motivo, foi possível transformar cada programa de teste num único bloco básico, viabilizando desse modo a extração de todo paralelismo existente.

A partir dos resultados obtidos nos experimentos, pode-se inferir que o processador cuja arquitetura originou nosso modelo de máquina VLIW (isto é, o processador i860 da Intel) ficaria mais balanceado se os seus projetistas tivessem optado pela inclusão de uma outra ULAI (Unidade Lógica e Aritmética Inteira) no *CORE* (dispositivo funcional que executa as leituras e escritas na memória, as operações inteiras e de controle do seqüenciamento). No modo dual, esse processador é capaz de despachar duas instruções simultaneamente: uma para a unidade *CORE* e outra para a unidade de ponto flutuante. Esse modo dual de operação poderia ser utilizado mais eficientemente se duas instruções com inteiros pudessem também ser executadas em paralelo pelo *CORE* contendo uma segunda ULAI. Conforme indicado pelos experimentos, essa unidade adicional permaneceria em operação praticamente ao longo de todo o tempo de execução dos programas de nossa bateria de testes.

Apêndice A

Programas Exemplo

```
/*
***************************************************************************
Programa: LIVERMOR.C
Descricao: Livermore Loop 24
***************************************************************************
*/

#define N      1000
int x[N] =

{346, 130,1982,1090,2656,1117,2595, 415,1948,1126, 4,2558, 571,1879, 492,
1360,2412,2721,1463,1047, 119,1441,1190,1985,1214, 509, 252,2571,2779,1816,
681,1651,2995,2593, 734,1310, 979, 995, 561,1092, 489,2288,1466,2664,2892,
1863,1766,2364,2639, 151,2427, 100,1795,2812, 108, 666, 347,1042,1774, 169,
2589,2383, 666,1941,1390,1878,1565,1779,1190,2233, 53,1429,2285,2422,2333,
1937,2636,1268, 460, 458, 936,2160, 842,2142,2667, 115, 116,2418,1156,1279,
8, 859,1561,2297, 755,1981, 275,2040,1690,1401, 137,1735,2343,1267,2312,
1111,1733,1993, 554,1353,2126,1018,1086, 970,2484,2614,2431,2999, 86, 730,
2504,1891,1492, 15,2143,1246,2484,2180, 168,1704, 679,2528, 365,2966,1135,
2740, 323, 580,1378, 736, 327,1164,1748,2020, 113, 445, 249,2243, 480, 672,
2625,2691,1799, 422, 344,2231, 480, 870,2821,1776,2903,1205,2522,2192,1113,
1878,2172, 121, 381,2461,2332,2982,2562, 774, 118, 505,1889,1323,1152,2436,
2365,1365,2079,2683, 762,2826,1109, 313,1179, 367,1310,1146,2623, 752,2028,
177,2013,1446,2935,1747,2094,2025,1778,1763,1563,1974,2459,1111,2831, 281,
2099,2051,1103,2798,2294,1764,2656,2693,2147,2287, 472,2732,1926,2962, 785,
193,2125,2948,1930,2207,2104, 16,2918, 184,1326,2096, 794, 421,2269,1543,
666,2642, 40,2695,2100,2788,1855,1212, 57,1791, 706,1624, 288, 211,2918,
2961, 851,1783,1396,1951, 582, 649, 935, 698,2550,2795,2862, 675, 416,2722,
2546, 874, 435,2059,1656, 656,1463,1103,2849,2749,1622,1529,1921, 780,1785,
1309, 270, 781, 671,1186,1243, 851,2460,2123,2580,2624,1070, 818, 467, 181,
516,1046,1259, 6, 185,2439,2371,1205,1221,1302,2566,1854, 13, 418,1745,
2659,1214, 321,2702,1592, 565,2708,1720, 993,1137,1701, 767, 226,1028,2389,
36, 743,1574, 816,2879,1909,1812,2020,1336,1760, 104,1738,1101,1374, 744,
1683, 554,1613,2820,1567,1670, 392,1468,1290,2004, 471, 959,1712,2655,1767,
2826, 978, 123,2471, 303,2481, 616,2149,2524,1682, 985,1560, 463,1787,2749,
2991,2335,2652,1308,1210,2245,1625,1133,2087, 324, 287,1087,1527, 893,2044,
909,2345,2175, 308,1862, 391,2574,1513, 180,1128,1538,1250,2404,1454, 418,
2312, 775, 984, 921,2596, 319,2002, 360, 629,2324,2339, 693,2456,2152,2688,
2836, 608, 233,1887,2331,1433,2510,1278,2187, 175,2252, 845,1917, 652,2570,
2040,2177,1505, 19, 465, 646,1124,1322,2221,1894, 435,2139,1182,2608,1432,
2665, 161,2146,2573, 995,1921,2327,2049, 979, 123,1871,1988,1784,2595,2269,
1991,1043, 903, 717,2511,1216, 74,1198,2787,1871,2851,1805,2996, 282, 247,
484,2253,1048,2836, 345,2130, 370,2807,1810,2029,1997,1600,1281,2495, 696,
2812,2747, 780,2503,2307,1102,2338,1912,1891,2882, 10,1636, 21,1215, 448,
193, 230,2977, 135,1774,1120,2113, 496,2504, 737,2437, 653,2947,1364, 1,
1265,1180, 223,1814,1634,1749,1965,1394, 177, 617,1988,2241,2161,2178,1377,
1172, 22, 512, 891,1360,2399,1971,1333,2593,2911, 275,2736,1174, 304, 567,
1287, 311,1889, 627, 924,2246,2114,1239,1339,2587,1549,2348, 342,2899, 711,
```

```
1177,2784,2418,1206,2686,2662,1143, 309,1351, 779,2857, 406,2300,2426,2303,
2311,2943,1153,1652,2984,2470,1075,2363,2652, 707,2342, 528,1229,1321,2455,
561,1772,1664,1251,2627,1745, 828, 199,2457, 232,1513, 210,1137, 38,2467,
1728,2647,1874,2052, 266, 159,1190,2573,1436,2030, 87, 850,1567,1834,1018,
560, 200,1938,1402,2561,2304,2631, 407, 858,1062,1203,1713, 179, 362,1593,
1582,1631, 467, 111,2600, 758,1378, 897,2938,2608,1599,2422,2899,1947,1357,
1511, 507,1288,2050,2590,1526,2602,2689, 188,2752, 664,2411,1063, 704, 187,
2651,2082,2142,1386, 993, 374,1505,1972,1921, 321,1380,2215, 221,2564, 745,
1096,1735, 730, 778,2467, 539, 86, 715,1948, 61,1524, 595,1637,1166, 332,
2259, 730,2477, 620, 80, 594,1027,1196,1710, 800,2079,2887,1765,2233, 324,
951,1385,2583,2711,2888, 468, 327,2037,1841,2942, 343,1060,1023,1312, 603,
920,2315,1149,2888,1688, 685, 778,2088, 586,2114,2047, 928,2717,1675,2293,
312, 997,2463,2522,1282,1972, 11,2707, 837,2510,2047,2594,2186, 77,2459,
2232,2728,1559,2626,1708,1556,2117, 151,1644, 204, 589, 743,1451,2397,1913,
1462,1260,1530,2929,1756,1965, 183,2726,1818,2183,1933,2760,1976,2461,2695,
1650,2378,1011, 77,2037, 964, 521,1824, 274,2585, 879,2014,1715,2475,1035,
1135, 732, 951,1912,1776,2406,1495,2697,1398, 325,2493,1071,1822,2650,1445,
2671,1972,1066,1411,2096,2290,2440,1798, 971,2594,1888,2252,1403, 325,2550,
1550,2014,1190,1151,1542,2060,1137,1246,1727,1111, 393,1872,2613,2340,2682,
180,2045,1833,2010, 61,2655,2888,2670,2386,1914,2386,1891, 672,2807, 463,
451,1205, 217,1317,2961,1432,2028, 534, 671,2167,1741,2471, 287,2922,2449,
749,1191,1877, 217,1659,1385, 587,2436, 882, 563,2393,2175, 272,2627,2784,
1370,2563,2670, 160,1803,1975,2737, 985, 664, 190,1240,1187,1671,2216,1247,
2472,2665,1779, 766, 407,2944, 500, 561,2044,1173,1295,2047,2511,2685,2235,
834,2834,1221,1233,1071, 727,1717, 429,2391,2432,1634,1910,2142,1920, 199,
1891,1634,2275,2694,2006,2327,2729, 986,1885,2121, 816,1511,1134,1939,1387,
701, 59,1127,1645,2337, 254, 638,2818, 605,1527,2448,2151, 51,1827,2711,
1048, 549,1751,2392,1995, 287,2957,1530,2171, 951, 213,2003,2736, 28, 968,
1559,2268,2182, 633, 779, 24,1853,1205,2930,2873};

int
main (void)
{
int k, m;

        m = 0;
        for (k = 0; k < N; k++)
                if (x[k] < x[m])
                        m = k;
        return (m);
}



/*
******************************************************************************
Programa: QUICK.C
Descricao: Ordena um vetor de tamanho 200. Algoritmo Quick-Sort
******************************************************************************
*/

#define N       200

void quick (int, int);

int     x[N] = { 46, 130, 182, 190, 256, 217, 195, 115, 148, 226,
                4, 158, 271, 79, 192, 160, 12, 21, 263, 147,
                119, 241, 290, 185, 14, 209, 252, 171, 79, 16,
                81, 151, 295, 193, 134, 110, 79, 95, 261, 192,
                189, 188, 266, 264, 192, 63, 266, 264, 239, 151,
                27, 100, 295, 112, 108, 66, 47, 142, 274, 169,
                189, 283, 66, 141, 190, 78, 65, 279, 290, 133,
                53, 229, 185, 22, 233, 137, 236, 68, 160, 158,
                36, 60, 242, 42, 267, 115, 116, 18, 256, 79,
                8, 259, 61, 197, 155, 181, 275, 240, 190, 201,
                137, 235, 243, 67, 212, 211, 233, 193, 254, 153,
                26, 118, 186, 70, 84, 214, 31, 299, 86, 130,
                104, 91, 292, 15, 43, 46, 84, 80, 168, 204,
                79, 128, 65, 266, 235, 40, 23, 280, 178, 136,
                27, 264, 248, 220, 113, 145, 249, 143, 180, 72,
                225, 291, 299, 122, 44, 131, 180, 270, 121, 276,
```

```
                203, 5, 122, 92, 213, 78, 72, 121, 81, 61,
                232, 282, 162, 174, 118, 205, 89, 123, 252, 36,
                265, 165, 279, 283, 162, 126, 209, 13, 279, 67,
                110, 246, 223, 152, 228, 177, 213, 246, 235, 247 };


void quick (inicio, fim)
int inicio, fim;
{
int antes, depois, meio, aux;

        antes = inicio;
        depois = fim;
        meio = x[(inicio + fim) / 2];
        do
        {
                while(x[antes] < meio)
                        antes++;
                while(meio < x[depois])
                        depois- -;
                if (antes <= depois)
                {
                        aux = x[antes];
                        x[antes++] = x[depois];
                        x[depois- -] = aux;
                }

        } while(depois >= antes);

        if (inicio < depois)
                quick (inicio, depois);
        if (antes < fim)
                quick (antes, fim);
}

int
main (void)
{
        quick (0, N - 1);
        return (x[N - 1]);
}



/*
***************************************************************************
Programa: BINARIA.C
Descricao: Realiza buscas binarias em um vetor ordenado de tamanho 1000
***************************************************************************
*/

#define N       1000

int Binaria (int, int, int);

int x[N] =

{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75,
76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90,
91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105,
106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120,
121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135,
136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150,
151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165,
166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180,
181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195,
196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210,
```

```
211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225,
226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240,
241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255,
256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270,
271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285,
286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300,
301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315,
316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330,
331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345,
346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 360,
361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375,
376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 390,
391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405,
406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420,
421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435,
436, 437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450,
451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465,
466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478, 479, 480,
481, 482, 483, 484, 485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495,
496, 497, 498, 499, 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510,
511, 512, 513, 514, 515, 516, 517, 518, 519, 520, 521, 522, 523, 524, 525,
526, 527, 528, 529, 530, 531, 532, 533, 534, 535, 536, 537, 538, 539, 540,
541, 542, 543, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553, 554, 555,
556, 557, 558, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568, 569, 570,
571, 572, 573, 574, 575, 576, 577, 578, 579, 580, 581, 582, 583, 584, 585,
586, 587, 588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 598, 599, 600,
601, 602, 603, 604, 605, 606, 607, 608, 609, 610, 611, 612, 613, 614, 615,
616, 617, 618, 619, 620, 621, 622, 623, 624, 625, 626, 627, 628, 629, 630,
631, 632, 633, 634, 635, 636, 637, 638, 639, 640, 641, 642, 643, 644, 645,
646, 647, 648, 649, 650, 651, 652, 653, 654, 655, 656, 657, 658, 659, 660,
661, 662, 663, 664, 665, 666, 667, 668, 669, 670, 671, 672, 673, 674, 675,
676, 677, 678, 679, 680, 681, 682, 683, 684, 685, 686, 687, 688, 689, 690,
691, 692, 693, 694, 695, 696, 697, 698, 699, 700, 701, 702, 703, 704, 705,
706, 707, 708, 709, 710, 711, 712, 713, 714, 715, 716, 717, 718, 719, 720,
721, 722, 723, 724, 725, 726, 727, 728, 729, 730, 731, 732, 733, 734, 735,
736, 737, 738, 739, 740, 741, 742, 743, 744, 745, 746, 747, 748, 749, 750,
751, 752, 753, 754, 755, 756, 757, 758, 759, 760, 761, 762, 763, 764, 765,
766, 767, 768, 769, 770, 771, 772, 773, 774, 775, 776, 777, 778, 779, 780,
781, 782, 783, 784, 785, 786, 787, 788, 789, 790, 791, 792, 793, 794, 795,
796, 797, 798, 799, 800, 801, 802, 803, 804, 805, 806, 807, 808, 809, 810,
811, 812, 813, 814, 815, 816, 817, 818, 819, 820, 821, 822, 823, 824, 825,
826, 827, 828, 829, 830, 831, 832, 833, 834, 835, 836, 837, 838, 839, 840,
841, 842, 843, 844, 845, 846, 847, 848, 849, 850, 851, 852, 853, 854, 855,
856, 857, 858, 859, 860, 861, 862, 863, 864, 865, 866, 867, 868, 869, 870,
871, 872, 873, 874, 875, 876, 877, 878, 879, 880, 881, 882, 883, 884, 885,
886, 887, 888, 889, 890, 891, 892, 893, 894, 895, 896, 897, 898, 899, 900,
901, 902, 903, 904, 905, 906, 907, 908, 909, 910, 911, 912, 913, 914, 915,
916, 917, 918, 919, 920, 921, 922, 923, 924, 925, 926, 927, 928, 929, 930,
931, 932, 933, 934, 935, 936, 937, 938, 939, 940, 941, 942, 943, 944, 945,
946, 947, 948, 949, 950, 951, 952, 953, 954, 955, 956, 957, 958, 959, 960,
961, 962, 963, 964, 965, 966, 967, 968, 969, 970, 971, 972, 973, 974, 975,
976, 977, 978, 979, 980, 981, 982, 983, 984, 985, 986, 987, 988, 989, 990,
991, 992, 993, 994, 995, 996, 997, 998, 999,1000};


int
Binaria (inicio, fim, dado)
int inicio;
int fim;
int dado;
{
int meio;

        meio = (inicio + fim) / 2;

        while ((dado != x[meio]) && (inicio != fim))
        {
                if (dado > x[meio])
                        inicio = meio + 1;
                else
                        fim = meio - 1;
```

```
                meio = (inicio + fim) / 2;
        }
        if (dado == x[meio])
                return (meio);
        else
                return (-1);
}


int
main (void)
{
int i;
int dados[10] = {1, 33, 888, 123, 989, 300, 277, 156, 745, 48};
int posicoes[10];

        for (i = 0; i < 10; i++)
                posicoes[i] = Binaria (1, N, dados[i]);

        return (posicoes[i - 1]);
}



/*
**************************************************************************
Programa: BOLHA.C
Descricao: Ordena um vetor de tamanho 60. Algoritmo Bubble-Sort
**************************************************************************
*/

#define N       60
#define true    1
#define false   0

int     x[N] = { 46, 130, 182, 190, 256, 217, 195, 115, 148, 226,
                4, 158, 271, 79, 192, 160, 12, 21, 263, 147,
                119, 241, 290, 185, 14, 209, 252, 171, 79, 16,
                81, 151, 295, 193, 134, 110, 79, 95, 261, 192,
                189, 188, 266, 264, 192, 63, 266, 264, 239, 151,
                110, 246, 223, 152, 228, 177, 213, 246, 235, 247 };


int
main (void)
{
int fim, ind, troquei, aux;

        fim = N - 1;
        do
        {
                troquei = false;
                for (ind = 0; ind <= (fim - 1); ind++)
                        if (x[ind] > x[ind + 1])
                        {
                                aux = x[ind];
                                x[ind] = x[ind + 1];
                                x[ind + 1] = aux;
                                troquei = true;
                        }
                fim = fim - 1;
        }
        while (troquei);
        return (x[N-1]);
}
```

```
/*
***********************************************************************
Programa: LU.C
Descricao: Decompoe um matriz 12x12. Metodo de decomposicao LU
***********************************************************************
*/

#define N     12

double A[N][N] =        {{4.0,1.0,3.0,3.0,2.0,6.0,9.0,3.0,5.0,8.0,5.0,3.0},
                        {8.0,2.0,8.0,9.0,7.0,5.0,5.0,6.0,2.0,5.0,3.3,9.0},
                        {4.0,3.0,1.0,7.0,5.0,1.5,2.0,1.0,9.0,7.0,2.8,8.0},
                        {5.0,2.0,3.0,9.0,4.0,3.5,2.0,2.0,5.0,9.0,3.8,8.0},
                        {3.0,4.0,7.0,4.0,4.0,2.0,7.0,4.0,7.0,3.0,1.0,3.0},
                        {9.0,7.0,4.0,6.0,7.0,9.0,4.0,4.0,4.5,5.5,2.0,5.0},
                        {4.0,5.0,4.0,3.0,2.0,1.5,2.0,9.0,2.3,6.7,4.0,6.0},
                        {5.0,3.0,2.0,6.0,9.0,8.0,4.0,5.6,8.2,2.0,5.0,2.0},
                        {4.0,5.0,9.0,3.0,2.0,1.5,2.0,9.0,2.3,6.0,6.0,1.0},
                        {6.0,4.0,2.0,5.0,7.0,5.6,4.0,8.6,5.2,2.0,7.0,6.0},
                        {7.0,9.0,8.0,3.0,2.0,6.6,9.0,5.6,5.2,2.0,7.0,6.0},
                        {5.0,3.0,2.0,6.0,7.0,8.0,4.0,5.6,8.2,7.7,9.0,5.0}};


double
main (void)
{
double aux, pivot;
int i, j, k, imax;

        for (i = 0; i <= (N-2); i++)
        {
                pivot = 0;
                imax = i;

                for (j = i; j <= (N-1); j++)
                {
                        aux = (A[j][i] < 0) ? -A[j][i] : A[j][i];
                        if (aux > pivot)
                        {
                                pivot = aux;
                                imax = j;
                        }
                }
                if (imax != i)
                        for (j = i; j <= (N-1); j++)
                        {
                                aux = A[i][j];
                                A[i][j] = A[imax][j];
                                A[imax][j] = aux;
                        }

                for (j = i+1; j <= (N-1); j++)
                        A[j][i] /= A[i][i];

                for (k = i+1; k <= (N-1); k++)
                        for (j = i+1; j <= (N-1); j++)
                                A[k][j] -= (A[k][i]*A[i][j]);
        }
        return (A[N-1][N-1]);
}
```

```
/*
***********************************************************************
Programa: INTEGRAL.C
Descricao: Calcula a area da funcao x^2. Metodo do trapezio
***********************************************************************
*/

#define LIM   1
```

```
#define INCR 0.005

double Y (double);

double
Y(t)
double t;
{
        return(t*t);
}


float
main (void)
{
double t, tant, Area, BMaior, BMenor;

        tant = 0;
        Area = 0.0;
        for(t = INCR; t <= LIM; t += INCR)
        {
                BMaior = Y(t);
                BMenor = Y(tant);
                Area += (BMaior + BMenor) * INCR / 2;
                tant = t;
        }
        return (Area);
}
```

@

Apêndice B

Ferramentas de Avaliação

B.1 O Sim860

O Sim860 é um simulador do i860 que reproduz as características deste processador e é capaz de simular todas as suas instruções, com exceção de algumas instruções gráficas e de controle do barramento externo. O Sim860 é orientado para a tela e, através de janelas, pode-se acompanhar a execução de cada instrução sendo simulada.

O Sim860 foi projetado para gerar um arquivo contendo o *trace* da execução do programa sendo simulado. Assim, além de ser útil na depuração de programas em *Assembly* do i860, ele pode ser utilizado como uma poderosa ferramenta de pesquisa.

As instruções não implementadas no Sim860 estão listadas em B.1.3.

O Sim860 foi inteiramente escrito em C, e foi compilado no Turbo C da Borland.

B.1.1 Os Parâmetros de Entrada de Sim860

O comando para a execução de Sim860 tem a seguinte sintaxe:

- sim860 <*programa*> <*arquivo-trace*> <*memória*>

O parâmetro <*programa*> é um arquivo de entrada. Trata-se de um executável em *Assembly* do i860 gerado por um compilador qualquer. O formato binário que este arquivo deve ter está descrito em B.4 e é igual ao formato do executável gerado pelo compilador HighC da Metaware [36].

O parâmetro <*arquivo-trace*> especifica o identificador do arquivo de saída gerado por Sim860 e contém, após a execução do Sim860, o *trace* do programa simulado. O formato binário deste arquivo também pode ser encontrado em B.4.

O arquivo <*memória*> é, também, um arquivo de saída gerado pelo Sim860. Ele contém, após a simulação, o estado da memória deixado pelo programa simulado. Este arquivo é útil quando se deseja verificar o estado final de uma matriz manipulada pelo programa simulado, por exemplo. O formato binário deste arquivo pode ser encontrado em B.4.

Os três parâmetros são obrigatórios e a ausência de um deles é detectada pelo Sim860 e indicada através de uma mensagem de erro.

B.1.2 Usando o Sim860

Após o comando de execução, o Sim860 apresenta a identificação do pacote e do autor e, a seguir, a tela de operação. Esta tela está dividida em dez janelas, como na Figura B.1 . Nestas janelas é indicado o *status* do processador após a execução de uma instrução.

J1 - Nesta janela estão indicados os conteúdos dos registradores inteiros do i860. O registradores pares ficam à esquerda;

J2 - Indica o conteúdo dos registradores de ponto flutuante. Os registradores pares ficam à esquerda;

J3 - Indica o estado do *flag* CC;

J4 - Indica o estado do *flag* OF;

J5 - Indica o código da instrução corrente em hexadecimal;

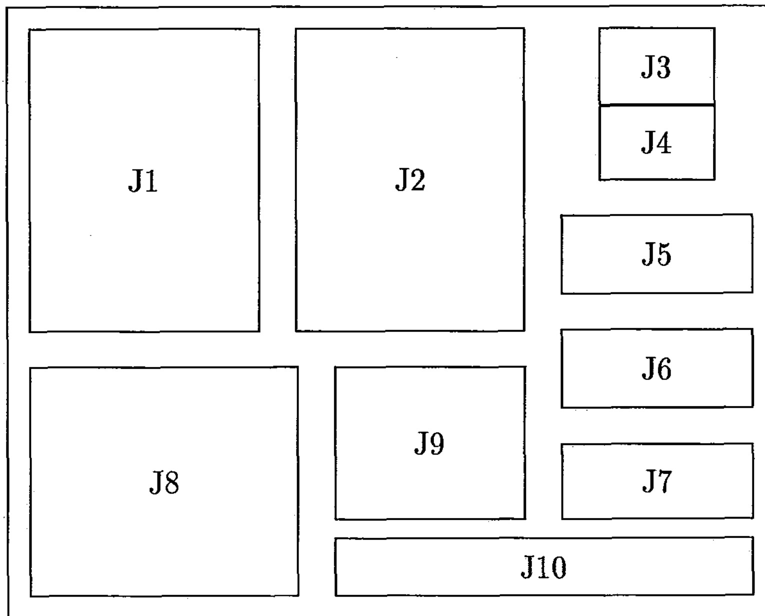


Figura B.1: Tela do Programa Sim860

J6 - Indica o endereço do último acesso de leitura ou escrita na memória, em hexadecimal;

J7 - Indica o valor do último dado lido ou escrito na memória, em hexadecimal. Indica também se o dado foi lido ou escrito;

J8 - Nesta janela estão indicadas até 8 instruções, pertencentes ao grupo das 8 últimas instruções executadas. A última delas que foi executada é apresentada em vídeo inverso. Quando ocorre uma instrução de desvio para uma instrução fora da janela, esta é limpa, e é apresentada apenas a instrução alvo na mesma;

J9 - Indica o conteúdo dos registradores EPSR, PSR, DIRBASE, FIR e PC, do i860. O campo NPC desta janela, indica o endereço de desvio calculado pela última instrução de desvio;

J10 - Apresenta o número da versão do Sim860. Quando o Sim860 termina uma simulação normalmente, é apresentada a mensagem "Programa terminado normalmente" nesta janela.

O Sim860 possui diferentes modos de execução. São eles:

Modo Step - É neste modo que o Sim860 inicia sua execução. Neste modo é executada uma instrução por vez, a cada acionamento da tecla de espaço. Para entrar neste modo, vindo de um outro, basta que se acione a tecla ESPAÇO;

Modo Tela - Neste modo as instruções são executadas sem interrupção e exibidas na tela. Para entrar neste modo é necessário que se acione a tecla ENTER (ou RETURN);

Modo Direto - Neste modo as instruções são executadas sem interrupção, mas as janelas da tela não são atualizadas. Por esse motivo, neste modo o programa é executado centenas de vezes mais rápido que no Modo Tela. Para entrar neste modo usa-se a tecla R;

A forma como os dados são exibidos nas janelas também pode ser alterada com o acionamento de teclas específicas:

F2 - Passa a exibir os registradores inteiros em decimal;

F3 - Passa a exibir os registradores inteiros em hexadecimal;

F4 - Passa a exibir os registradores de ponto flutuante no formato de 64 bits (*double*). Neste formato não existem registradores ímpares. Cada par de registradores (F0 e F1, F2 e F3, etc) é tratado como se fosse um único registrador (quando o i860 opera com números de ponto flutuante de 64 bits ele trata seus registradores desta maneira);

F5 - Passa a exibir os registradores de ponto flutuante em hexadecimal;

F6 - Passa a exibir os registradores de ponto flutuante em decimal;

F1 - Quando esta tecla é acionada, os registradores inteiros passam a ser exibidos em hexadecimal e os de ponto flutuante no formato de 32 bits. Este é o formato default do Sim860.

O Sim860 permite também a inclusão de um *break-point*, através da tecla F7. Quando esta tecla é acionada, a janela J5 se modifica para receber um número em hexadecimal. Este número passa a ser o endereço de *break-point*.

Para interromper a execução do programa antes de terminar a simulação, basta que se acione a tecla F.

B.1.3 Instruções Não Implementadas

As instruções não implementadas no Sim860 são: faddp, faddz, form, fzchkl, fzchks, lock, pfaddp, pfaddz, pform, pfzchkl, pfzchks, pst.d e unlock.

B.2 Scode

O programa Scode gera, a partir de um arquivo contendo um executável para o processador i860, uma descrição detalhada de cada instrução presente neste executável. Com o arquivo de *trace* gerado pelo Sim860 e a saída de Scode, pode-se analisar detalhadamente cada instrução simulada pelo i860.

O Scode foi inteiramente escrito em C, e foi compilado no Turbo C da Borland.

B.2.1 Os Parâmetros de Entrada de Scode

O comando para a execução de Scode tem a seguinte sintaxe:

- scode <*programa*> <*arquivo-scode*> <*memória*>

O parâmetro <*programa*> é um arquivo de entrada e tem as mesmas características do arquivo de entrada, de mesmo nome, do Sim860 (ver B.1.1).

O parâmetro <*arquivo-scode*> é um arquivo de saída gerado por Scode e contém, após a execução de Scode, registros com a descrição de cada instrução presente no arquivo <*programa*>, na seqüência em que elas aparecem em <*programa*>. O formato binário dos registros pode ser visto em B.4.

O arquivo <*memória*> é, também, um arquivo de saída gerado por Scode. Ele conterá, após a execução de Scode, o estado da memória antes da

execução do programa <*programa*>. O seu formato binário é apresentado em B.4.

Os três parâmetros são obrigatórios e a ausência de um deles é detectada por Scode e indicada através de um erro.

B.3 Compact

O programa Compact é um compactador, que gera um código próprio para uma máquina VLIW a partir dos arquivos de saída dos programas Sim860 e Scode. As características desta máquina são passadas através de um terceiro arquivo de entrada. Como saída ele produz estatísticas a respeito da compactação e dos dados de entrada.

O programa Compact foi inteiramente escrito em C, e foi compilado no Turbo C da Borland. Esta mesma versão foi também compilada pelo BSD C, do sistema operacional BSD 386 de Berkeley.

B.3.1 Os Parâmetros de Entrada de Compact

O comando para a execução de Compact tem a seguinte sintaxe:

- compact <*arquivo-trace*> <*arquivo-scode*> <*parâmetros*> [<*arquivo-saída*>]

Os parâmetros <*arquivo-trace*> e <*arquivo-scode*> são os arquivos de saída de Sim860 e Scode, respectivamente. O parâmetro <*parâmetros*> é um arquivo de entrada contendo as características da máquina VLIW para qual o programa está sendo compactado e alguns comandos. O formato do arquivo <*parâmetros*> pode ser visto em B.4.

O parâmetro <*arquivo-saída*> é um arquivo texto que contém as estatísticas colhidas por Compact durante a compactação e, de acordo com o número na última linha de <*parâmetros*>, a listagem dos ciclos de máquina da VLIW e as instruções alocadas neles. As estatísticas contidas em <*arquivo-saída*> estão indicadas na Figura IV.2.

O parâmetro <*arquivo-saída*> é opcional. Caso ele não seja indicado, as informações que iriam para ele são enviadas para *stdout* (*standart output*, normalmente a tela).

B.4 O Formato dos Arquivos

O arquivo <*programa*>, que deve ser passado como parâmetro para o Sim860 e para o Scode, é um arquivo executável que segue o *Common Object File Format* (*COFF*), definido no *Unix System V/386 Programmers' Guide.* Apenas algumas diferenças são necessárias para o microprocessador i860 e estão descritas em [46].

O arquivo <*arquivo-trace*> é gerado pelo Sim860. Trata-se de um arquivo binário composto unicamente de registros com o seguinte formato:

```
struct trace {
        unsigned long endereco;
        unsigned char tipo;
        char lixo[3];
}
```

Estes registros aparecem no arquivo na seqüência em que são gerados pelo Sim860, que é igual a seqüência dos acessos à memória feitos pelo programa sendo simulado. O ítem *endereco* da *struct trace* é, como o nome diz, o endereço acessado pelo programa simulado. O vetor *lixo* foi incluído para permitir compatibilidade deste arquivo com sistemas operacionais onde a função *fread*, do C, só consegue ler ítens com granularidade de quatro Bytes (no Unix BSD386 de Berkeley, por exemplo). O ítem *tipo* especifica o tipo do acesso[1]:

0x0 - Acesso de leitura ou escrita de um dado na memória;

0x80 - Acesso de leitura ou escrita de um dado na memória com *data cache miss*;

[1] O prefixo 0x especifica um número em hexadecimal

0x1 - Acesso do tipo *fetch*. É um acesso de leitura a uma instrução;

0x81 - Acesso do tipo *fetch* com *instruction cache miss*;

0x2 - Não caracteriza um acesso à memória. Serve para indicar o número total de *tlb miss* gerados pela execução do programa simulado. Este número vem indicado no campo *endereco*. Um "acesso" deste tipo, aparece no fim do *trace* e serve, também, para indicar o seu final;

0x3 - Não caracteriza um acesso à memória. Indica no campo *endereco* o número total de *instruction cache miss* gerados pela execução do programa;

0x4 - Também não caracteriza um acesso à memória. Indica no campo *endereco* o número total de *data cache miss* gerados pela execução do programa.

Os "acessos" dos tipos 0x2, 0x3 e 0x4 aparecem no fim do *trace* nesta ordem e indicam o seu final.

O arquivo *memoria* é gerado por Sim860 e por Scode e contém o estado da memória antes ou após a execução do programa, quando gerado por Scode ou Sim860, respectivamente. Este arquivo é um retrato fiel do conteúdo binário de uma memória normal que tivesse sido alocada para um programa por um sistema operacional Unix. Neste arquivo a memória está dividida em páginas de 4096 Bytes (4K Bytes); um grupo de páginas para cada segmento do programa (ver *COFF - Unix System V/386 Programmers' Guide*). Os segmentos estão gravados no arquivo na ordem: *text*, *data*, *bss*, *stack*. O tamanho escolhido para o *stack* foi de duas páginas e seu endereço inicial 0x80000000. Para alterar estes valores é necessário modificar os fontes. Os demais segmentos têm seu tamanho e endereço escolhidos pelo compilador.

O arquivo <*arquivo-scode*> é gerado pelo programa Scode e é inteiramente composto de registros que descrevem as características de cada instrução presente em <*programa*>. O formato dos registros pode ser visto na Figura B.2 .

O arquivo <*parametros*> é um dos arquivos de entrada de Compact. Trata-se de um arquivo texto como o da Figura B.3 . Os números devem estar na

```
struct instout
    {
        unsigned long endereco;
        unsigned long codigo;
        char nome[12];
        TIPOINST tipo;
        int lixo1; /* compatibilidade com BSD 386 */
        unsigned long op1;
        unsigned long op2;
        unsigned long op3;
        char read_set[8];
        /* [0], [1], [2] - > 'r': reg. inteiro, 'f': reg. ponto flut. */
        /* 'i': contante (imediato) */
        /* a posicao 0, 1 ou 2 indica 1o 2o ou 3o operando (op1, op2 ou op3) */
        /* [3] - > 'c': flag CC */
        /* [4] - > 'o': flag OF */
        /* [5] - > 'p': PSR, 'e': PC, 'k': PC e PSR */
        /* um '0' em qualquer posicao indica que o recurso nao e' usado */

        char pipe_read[8];

        char write_set[8];
        /* [0], [1], [2] - > 'r': reg. inteiro, 'f': reg. ponto flut. */
        /* a posicao 0, 1 ou 2 indicam 1o 2o ou 3o operando (op1, op2 ou op3) */
        /* [3] - > 'c': flag CC */
        /* [4] - > 'o': flag OF */
        /* [5] - > 'p': PSR, 'e': PC, 'k': PC e PSR */
        /* um '0' em qualquer posicao indica que o recurso nao e' usado */

        char pipe_write[8];

        char unit_set[12];
        /* [0] - > 'i': alu inteira; 'a': adder P.F.; 'm': multiplier P.F. */
        /* 'd': adder e multiplier P.F. */
        /* [1] - > 'r': barramento de leitura de registradores Int. */
        /* [2] - > 'w': barramento de escrita em registradores Int. */
        /* [3] - > 'm': barramento de acesso 'a memoria */
        /* [4] - > 't': barramento entre bancos de registradores */
        /* [5] - > 'a': adder para calculo de endereco */
        /* [6] - > 'r': barramento de leitura de registradores P.F. */
        /* [7] - > 'w': barramento de escrita em registradores P.F. */
        /* um '0' em qualquer posicao indica que o recurso nao e' usado */

        char pipe_unit[12];

        char pipe_set[4];
        /* [0] - > 'm': pipe de memoria; 'b': pipe de branch; */
        /* 'f': pipe P.F.; 'i': pipe inteira; 't': pipe transf. */
        /* [1] - > um valor n que indica o numero de ciclos para a execucao */
        /* um '0' em qualquer posicao indica que o recurso nao e' usado */

    }
```

Figura B.2: Formato do Registro do <*arquivo-scode*>

10 Alu inteira
5 Adder de Ponto Flutuante
5 Multiplier de Ponto Flutuante
15 Barramento de leitura de Registrador I.
10 Barramento de escrita em Registrador I.
10 Barramento de leitura de Registrador P. F.
8 Barramento de escrita em Registrador P. F.
10 Adder de Endereco
8 Canais de Acesso a Memoria
2 Barr. de Transf. entre Reg. Int. e P.F.
1 Faz ou Nao Faz Rename (1/0)
50 Numero de Reg. Inteiros para Rename
30 Numero de Reg. de P. F. para Rename
0 Imprime ou Nao Imprime as Inst. (1/0)

Figura B.3: Arquivo de Parâmetros de Compact

primeira coluna e a linha em que aparecem determina sua função. O texto é apenas comentário explicando a função de cada número e não é utilizado por Compact.

Os fontes dos programas Sim860, Scode, e Compact podem ser conseguidos na biblioteca do Programa de Engenharia de Sistemas e Computação da COPPE - Universidade Federal do Rio de Janeiro.

Referências Bibliográficas

[1] D. A. Patterson D. J. Kuck and D. H. Lawrie, "*A VLSI RISC*", Computer, Vol. 15, No. 9, September 1982, pp. 8-21.

[2] D. D. Gajski and J. K. Peir, "*Essential Issues in Multiprocessor Systems*", Computer, Vol. 18, No. 6, June 1985, pp. 10-16.

[3] R. Duncan, "*A Survey of Parallel Computer Architectures*", Computer, February 1990, pp. 5-16.

[4] R. W. Rockney, "*MIMD Computing in the USA - 1984*", Parallel Computing, Vol. 2, 1985, pp. 119-136.

[5] M. J. Flynn, "*Very High-Speed Computing Systems*", Proceedings of the IEEE 54, 1966, pp. 1901-1909.

[6] W. Händler, "*The Impact of Classification Schemes on Computer Architecture*", Proceedings of 1977 International Conference on Parallel Processing, 1977, pp. 7-15.

[7] R. M. Russel, "*The Cray-1 Computer System*", Communications of ACM, Vol. 21, No. 1, January 1978, pp. 63-72.

[8] K. Miura and K. Ushida, "*FACOM Vector Processor System: VP-100/VP-200*", Supercomputers: Design and Applications, IEEE Computer Society, 1984, pp. 59-73.

[9] T. Watanabe, "*Architecture and Performance of NEC Supercomputer SX System*", Parallel Computing, Vol. 5, 1987, pp. 247-255.

[10] H. S. Stone, "*High-Performance Computer Architecture*", Addison-Wesley, 1987.

[11] W. D. Hillis, "*The Connection Machine*", Cambridge, Mass., MIT Press, 1986.

[12] C. L. Seitz, "*The Cosmic Cube*", Communications of the ACM, Vol. 28, No. 1, January 1985, pp. 22-33.

[13] BBN Laboratories Inc, "*Butterfly Parallel Processors Overview*", BBN Report no. 6148, Version 1, March 1986.

[14] D. Gajsky, D. Kuck, D. Lawrie and A. Sameh, "*Cedar - A Large Scale Multiprocessor*", IEEE Proceedings 1983 International Conference on Parallel Processing, 1983, pp. 524-529.

[15] D. M. Pase and A. R. Larrabee, "*Intel iPSC Concurrente Computer*", Programming Parallel Processors, Addison-Wesley, 1988, pp. 105-124.

[16] A. F. Souza, "*Uma Máquina Paralela Híbrida*", Anais do VIII Congresso da SBC, UFRJ/NCE, Julho de 1988, pp. 315-332.

[17] A. F. Souza, "*O Hardware de uma Máquina Paralela Híbrida*", Projeto Final de Graduação, Departamento de Engenharia Eletrônica - EE/UFRJ, Agosto de 1988.

[18] E. S. T. Fernandes, C. L. Amorim, V. C. Barbosa, F. M. G. França and A. F. Souza, "*MPH - A Hybrid Parallel Machine*", Microprocessing and Microprogramming, North - Holland, Vol. 25, 1989, pp. 229-232.

[19] C. L. Amorim, R. Citro, A. F. Souza and E. M. Chaves Filho, "*The NCP-1 Parallel Computer System*", Relatório Técnico ES-241, Programa de Engenharia de Sistemas e Computação - COPPE/UFRJ, Abril de 1991.

[20] J. A. Fisher, "*Trace Scheduling: A Technique for Global Microcode Compaction*", IEEE Transactions on Computers, Vol. C30, No. 7, July 1981, pp. 478-490.

[21] J. A. Fisher, "*The VLIW Machine: A Multiprocessor for Compiling Scientific Code*", Computer, July 1984, pp. 174-182.

[22] J. R. Ellis, "*Bulldog: A Compiler for VLIW Architectures*", Cambridge, MA: MIT Press, 1986.

[23] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth and P. K. Rodman, "*A VLIW Architecture for a Trace Scheduling Compiler*", IEEE Transactions on Computers, Vol. 37, No. 8, August 1988, pp. 967-979.

[24] A. Nicolau, "*Percolation Scheduling: A Parallel Compilation Technique*", Technical Report, Department of Computer Science, Cornell University, May 1985, TR 85-678.

[25] D. Landskov, S. Davidson, and B. Shriver, "*Local Microcode Compaction Techniques*", Computer Surveys, Vol. 12, No. 3, September 1980, pp. 261-294.

[26] S. Davidson, D. Landskov, B. D. Shriver and P W. Mallett, "*Some Experiments in Local Microcode Compaction for Horizontal Machines*", IEEE Transactions on Computers, Vol. C30, No. 7, July 1981, pp. 460-477.

[27] A. Nicolau and J. A. Fisher, "*Measuring the Parallelism Avaliable for Very Long Instruction Word Architectures*", IEEE Transactions on Computers, Vol. C33, No. 11, November 1984, pp. 968-976.

[28] Arvind and V. Kathail, "*A Multiple Processor Data Flow Machine that Supports Generalized Procedures*", Proceedings of the 8th Annual Symposium on Computer Architecture, ACM (SIGARCH), vol. 9, No. 3, May 1981, pp. 291-302.

[29] Intel, "*i860 64-Bit Microprocessor Hardware Manual - Preliminary*", Intel, Order Number: 240296-003, October 1989.

[30] Intel, "*i860 64-Bit Microprocessor Programmer's Reference Manual*", Intel, 1989.

[31] F. H. McMahon, "*Fortran Kernels: MFLOPS*", Lawrence Livermore National Laboratory, 1983.

[32] D. Knuth, "*Art of Computer Programming*", Addison-Wesley Publishing Co., USA, 1973.

[33] E. Horowitz and S. Sahni, "*Fundamentals of Computer Algorithms*", Computer Science Press Inc., Maryland, USA, 1978.

[34] G. Forsythe e C. B. Moler, "*Computer Solution of Linear Algebraic Systems*", Prentice Hall, Englewood Cliffs, New Jersey, 1967.

[35] S. D. Conte, "*Elementary Numerical Analysis*", McGraw-Hill Book Co., New York, NY, USA, 1965.

[36] Metaware, "*HighC Programmer's Guide*", Metaware Incorporated, Santa Cruz CA, 1990.

[37] E. M. Riseman and C. C. Foster, "*The Inibition of Potential Parallelism by Conditional Jumps*", IEEE Transactions on Computers, vol. C-21, December 1972, pp. 1405-1411.

[38] D. J. DeWitt, "*A Machine Independent Approach to the Production of Optimal Horizontal Microcode*", Ph.D. Dissertation, Universit of Michigan, Ann Arbor, Techinical Report 76 DT4, August 1976.

[39] R. M. Tomasulo, "*An Efficient Algorithm for Exploiting Multiple Arithmetic Units*", IBM Journal, January 1967, pp. 25-33.

[40] G. S. Tjaden and M. J. Flynn, "*Detection and Parallel Execution of Independent Instructions*", IEEE Transactions on Computers, vol. C-19, No. 10, October 1970, pp. 889-895.

[41] M. Tokoro, E. Tamura and T. Takizuka, "*Optimisation of Microprograms*", IEEE Transactions on Computers, vol. C-30, No. 7, July 1981, pp. 491-504.

[42] M. Harris, "*Extending Microcode Compaction for Real Architectures*", ACM Micro, No. 20, 1987, pp. 40-53.

[43] S. U. Rao and A. K. Majumdar, "*Global Microcode Compaction - A Performance Evaluation by Simulation*", Microprocessing and Microprogramming, North - Holland, No. 20, 1988, pp. 159-174.

[44] S. Dasgupta and J. Tartar, "*The Indentification of Maximal Parallelism in Straight Line Microprograms*", IEEE Transaction on Computers, vol. C-25, pp. 986-991, October, 1976.

[45] A. V. Aho, R. Sethi and J. D. Ullman, "*Compilers. Principles, Techniques, and Tools*", Addison-Wesley, 1986.

[46] Intel, "*i860 64-Bit Microprocessor Assembler and Linker Reference Manual*", Intel, Order Number: 240436-003, January 1990.