Otimização de Código para Arquiteturas Superescalares

Juliana Serrano do Carmo Ferraz

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. Claudio Luis de Amorim, Ph. D
(Presidente)

Prof. Edil Severiano T. Fernandes, Ph. D.

Prof. Felipe Maia Galvão França, Ph. D.

Prof. Raul Queiroz Feitosa, Ph. D.

RIO DE JANEIRO, RJ - BRASIL
ABRIL DE 1995

FERRAZ, JULIANA SERRANO DO CARMO

Otimização de Código para Arquiteturas Superescalares

[Rio de Janeiro] 1995

IX, 137 p., 29,7 cm (COPPE/UFRJ, M. Sc., Engenharia de Sistemas e Computação, 1995)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Processadores Superescalares RISC

2. Arquiteturas Paralelas

3. Otimização de código

4. Simulação

I. COPPE/UFRJ II. Título (Série).

Ao meu marido Marcelo

Agradecimentos

Ao professor Claudio Amorim, pelo seu trabalho de orientação.

Aos amigos do laboratório de computação paralela da COPPE/SISTEMAS, por toda a ajuda que deram.

À minha família, sempre presente.

Um agradecimento especial a meus pais por tudo que me proporcionaram ao longo da vida.

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

À CAPES, pelo apoio financeiro.

Um agradecimento muito especial ao meu marido, Marcelo, por toda a sua ajuda, incentivo e presença, necessários para a conclusão deste trabalho.

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

Otimização de Código para Arquiteturas Superescalares

Juliana Serrano do Carmo Ferraz

Abril de 1995

Orientador: Claudio Luis de Amorim

Programa: Engenharia de Sistemas e Computação

Os processadores superescalares RISC oferecem um *hardware* bastante eficiente, utilizando várias técnicas de paralelismo, mas implementando um conjunto de instruções reduzido. Na realidade, criam um compromisso entre *hardware* e *software* pois o desempenho máximo do processador, no caso de linguagens de alto nível, só pode ser alcançado dispondo-se de um compilador que gere um código eficiente.

Este trabalho descreve o desenvolvimento de uma ferramenta para o processador i860 capaz de avaliar a utilização de recursos deste processador e prover ao programador um meio para identificar trechos de programa onde os recursos possam ser melhor utilizados. Através de um conjunto de programas é possível verificar a utilização de recursos e mostrar como esta ferramenta pode auxiliar ao programador.

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

Code Optimization for Superscalar Architectures

Juliana Serrano do Carmo Ferraz

April, 1995

Thesis Supervisor: Claudio Luis de Amorim

Department: Programa de Engenharia de Sistemas e Computação

The superscalar RISC processor offers many hardware resources, by using parallel technics and a reduced intruction set. The maximum efficiency of a hardware can only be achieved, when programing in high level languages, if an efficient compiler is available.

This work describes the development of a tool that not only evaluates the utilization of the i860 processor but also helps the programer to identify the parts of the program where the resourses could be more efficiently used. Thus, the set of programs here presented provide a means to verify how the resourses of the processor are utilized and to show how this tool can help the programer.

Índice

Lista de Figuras

Pág.

Capítulo 1

Introdução

A busca de melhorias no desempenho tem sido uma constante no desenvolvimento de microprocessadores. Para atingir este objetivo, várias técnicas de paralelismo têm sido implementadas, permitindo um aumento no número de instruções executadas por unidade de tempo.

Toda melhoria de desempenho está relacionada aos passos envolvidos na execução de uma instrução: busca da instrução (B), decodificação da instrução e obtenção dos operandos (D), execução das operações especificadas pela instrução (E) e escrita dos resultados (W).

A técnica de *pipelining* tem sido utilizada para melhorar o desempenho. Nesta os passos de execução de diferentes instruções são efetuados por unidades independentes, denominadas "estágios do *pipeline*". Enquanto a execução de determinada instrução se encontra em um estágio do *pipeline*, a execução de outra se encontra em um outro estágio, como mostra a figura 1.1. O resultado de cada estágio do *pipeline* é transferido para o próximo e, dessa forma, o tempo médio de execução por instrução é reduzido com a execução de mais de uma instrução em cada instante. Todavia, o tempo total de execução de uma instrução não é alterado.

Cada estágio do *pipeline* não requer necessariamente o mesmo tempo para ser executado, mas se os tempos de execução de cada estágio forem iguais, a técnica de *pipelining* pode ser melhor aproveitada, pois neste caso a sobreposição dos estágios não provoca qualquer espera por parte de algum estágio.

I0	B	D	E	W			
I1		B	D	E	W		
I2			B	D	E	W	
I3				B	D	E	W

Figura 1.1. - Utilização da técnica de *pipelining* para a execução de uma seqüência de instruções

Durante o desenvolvimento dos primeiros microprocessadores o tempo de busca da instrução era muito longo comparado aos tempos de decodificação e execução. Isto motivou o surgimento de processadores CISC (*Complex-instruction-set computer*), que possuem instruções mais complexas que aumentam o tempo de decodificação e execução para um tempo próximo ao de busca da instrução, acarretando melhor aproveitamento da técnica de *pipelining*.

No momento em que a tecnologia de memória e encapsulamento mudam as limitações do estágio de busca, reduzindo bastante seu tempo de execução, o desempenho em processadores CISC passa a ser limitado pelos estágios de decodificação e execução. Esta situação motivou o surgimento de arquiteturas RISC (*Reduced-instruction-set computer*).

As arquiteturas RISC têm como principal objetivo reduzir o número de ciclos de relógio necessários para executar uma instrução, proporcionando um aumento de desempenho no número total de instruções. Sua filosofia de implementação é "simplicidade e eficiência". Os processadores RISC provêem o uso eficiente do *hardware* através de simplificações na semântica da codificação do conjunto de instruções do processador. Pode-se, com isso, reduzir o número de ciclos de relógio necessários para a execução de uma instrução, contudo, em detrimento do número de instruções para executar um programa, que aumenta. Para poder obter um ganho frente às arquiteturas CISC, o programa a ser executado por uma arquitetura RISC deve aproveitar ao máximo os recursos do processador, permitindo que o número total de instruções necessárias para a

execução de um programa seja reduzido. Caso contrário, o desempenho oferecido por estas arquiteturas estaria aquém do desejado.

Até agora descrevemos as arquiteturas RISC focalizando apenas o uso de uma técnica de paralelismo: *pipelining*. Os processadores RISC, entretanto, são capazes de executar uma instrução por ciclo de relógio, sendo denominados processadores "escalares" RISC. Os métodos de paralelismo para aumentar o desempenho de um processador, contudo, não se esgotam com a técnica de *pipelining*. É possível melhorar o desempenho executando múltiplas instruções por ciclo de relógio, aumentando-se o número médio de instruções executadas por ciclo de relógio. São os processadores "superescalares" RISC. A figura 1.2 mostra a temporização da execução de uma instrução num processador superescalar, considerando a existência de duas unidades funcionais.

I0	B	D	E	W			
I1	B	D	E	W			
I2		B	D	E	W		
I3		B	D	E	W		
I4			B	D	E	W	
I5			B	D	E	W	
I6				B	D	E	W
I7				B	D	E	W

Figura 1.2. - Comportamento do *pipeline* durante a execução de uma seqüência de instruções numa arquitetura superescalar com duas unidades funcionais.

Um processador superescalar RISC utiliza múltiplas unidades funcionais para permitir que instruções que utilizam unidades funcionais diferentes possam ser executadas simultaneamente. É claro que a introdução de múltiplas unidades funcionais envolve fatores como disponibilidade de área no *chip* e custo, que limitam o número de unidades funcionais em um processador superescalar RISC.

As arquiteturas RISC foram desenvolvidas com o intuito de manter o *hardware* o mais simples possível e alcançar um desempenho relativamente elevado. Esta simplificação do *hardware*, no entanto, estabelece um compromisso com o desenvolvimento de *software*, principalmente de compiladores. Isto porque, além de outros fatores como a presença de *delayed branches*, o próprio conjunto reduzido de instruções já constitui um problema. Muitos comandos em linguagem de alto nível podem ser sintetizados de várias maneiras, dependendo dos operandos. Como conseqüência, os compiladores nem sempre conseguem gerar um código "ótimo" tamanha é a quantidade de regras que este precisa gerir.

Baseado nas dificuldades de desenvolver compiladores altamente eficientes para arquiteturas superescalares, bem como na dificuldade de programar em *assembly* num processador superescalar RISC, esse trabalho tem como objetivo desenvolver uma ferramenta que permita ao programador identificar trechos de um programa que não se encontram otimizados. A ferramenta consiste de um simulador para o processador superescalar RISC Intel i860, utilizado na máquina paralela NCP-1 [14], desenvolvida pelo grupo de computação paralela do Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ.

Diferente dos produtos oferecidos pela Intel para o i860, o simulador SIM860 descreve o estado dos recursos do processador a cada ciclo de relógio da execução de um programa. Acompanha o simulador um programa para fornecer informações gráficas a respeito da utilização dos recursos do processador durante toda ou parte do tempo de execução, denominado SHOWREPO. Estes dois programas juntos fornecem elementos para otimizar um código gerado pelo compilador ou um código desenvolvido diretamente em *assembly*, em tempo de execução.

Este trabalho está dividido em seis capítulos. O primeiro é introdutório. No capítulo 2 descrevemos a arquitetura do processador i860, de modo a auxiliar no entendimento dos passos envolvidos no desenvolvimento do simulador.

No capítulo 3 apresentamos as características do simulador, analisando sua entrada e saída, assim como suas limitações. Neste capítulo também são descritas as características do programa SHOWREPO.

No capítulo 4 é descrita a implementação do simulador baseado nas características do processador.

No capítulo 5 são descritos os experimentos efetuados e seus resultados.

Finalmente, no capítulo 6 são apresentadas as conclusões deste trabalho.

O apêndice A apresenta a listagem completa dos programas utilizados nos experimentos.

O apêndice B apresenta os códigos de erro que podem apresentados pelo simulador.

Capítulo 2

A Arquitetura do Processador Intel i860

2.1. Introdução

O microprocessador Intel i860 apresenta um desempenho de pico (100 Mflops) equivalente ao de um "supercomputador" em um simples componente VLSI. A arquitetura do processador explora vários conceitos de arquiteturas avançadas objetivando prover desempenho equilibrado de operações com inteiros, ponto-flutuante e operações gráficas. Sua arquitetura de 64 bits segue técnicas de um projeto RISC (*Reduced instruction set computer*), com unidades de processamento *pipelined* e paralelo, grandes barramentos de dados, e duas memórias *cache* internas, uma de dados e outra de instruções. Incorporando unidades funcionais de inteiros, ponto-flutuante e gráfica em um único *chip*, reduz o *overhead* de comunicação entre *chips*.

O Intel i860 XP constitui a nova geração da UCP i860. Em relação à primeira, denominada i860 XR, esta nova geração possui registradores adicionais, *caches* maiores, *burst bus access* e freqüência de operação de 50 MHz, entre outras características.

A figura 2.1 ilustra a arquitetura interna do i860, que consiste das seguintes unidades:

- Unidade central;
- Unidade de ponto-flutuante;
- Unidade de gerenciamento de memória;
- Memórias *cache* de dados e instruções;
- Unidade de controle de barramento e *cache*;
- Unidade de controle de concorrência independente (DCCU - *Detached concurrent control unit*).

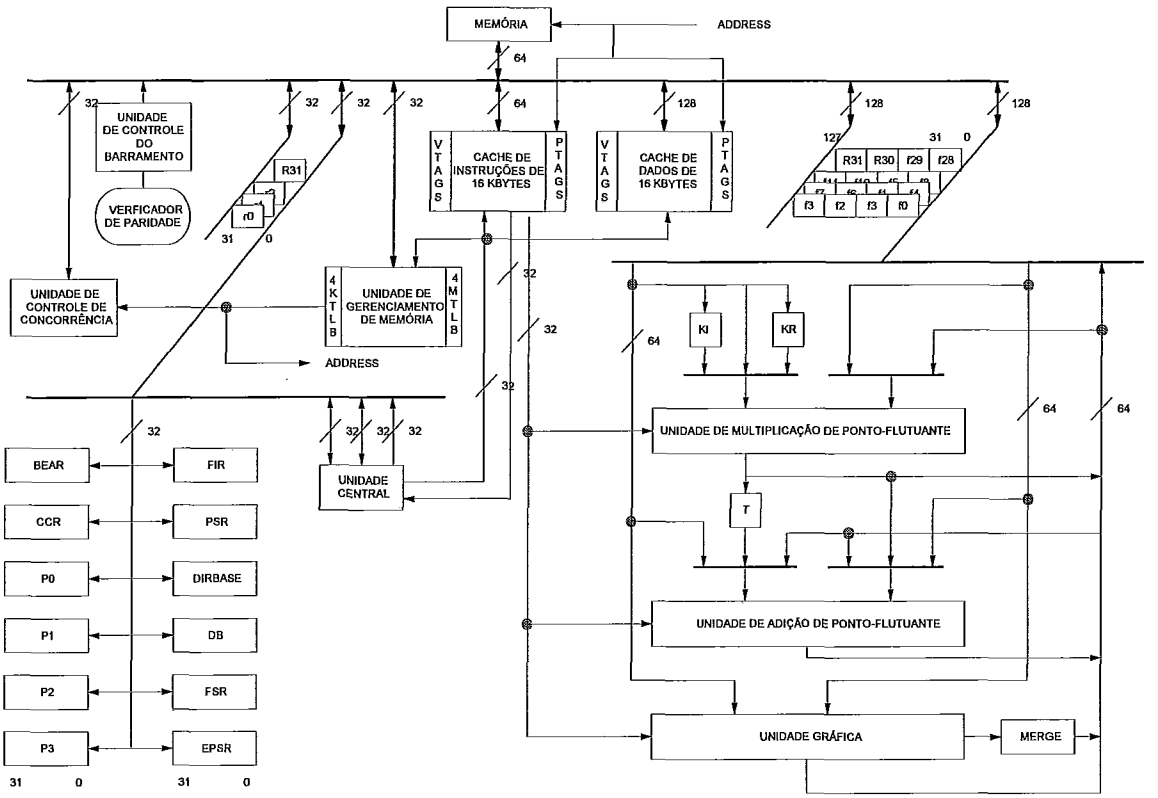


Figura 2.1. - Arquitetura interna do processador Intel i860.

Objetivando executar cada instrução de forma eficiente, o processamento da instrução é dividido em quatro estágios: busca da instrução, decodificação da instrução, execução da instrução e escrita dos resultados nos registradores. Cada estágio desempenha sua atividade em um ciclo de relógio e passa suas informações para o próximo estágio. A técnica de *pipelining* é utilizada para a execução de uma instrução e estes quatro estágios compõem o que denominamos "*pipeline* de instrução", cujo funcionamento está ilustrado na figura 2.2.

O estágio de busca obtém uma instrução da *cache* de instruções e a armazena num *buffer*. Quando a instrução não é encontrada na *cache*, a busca é feita na memória externa. Isto impede que uma instrução possa ser trazida em um ciclo de relógio.

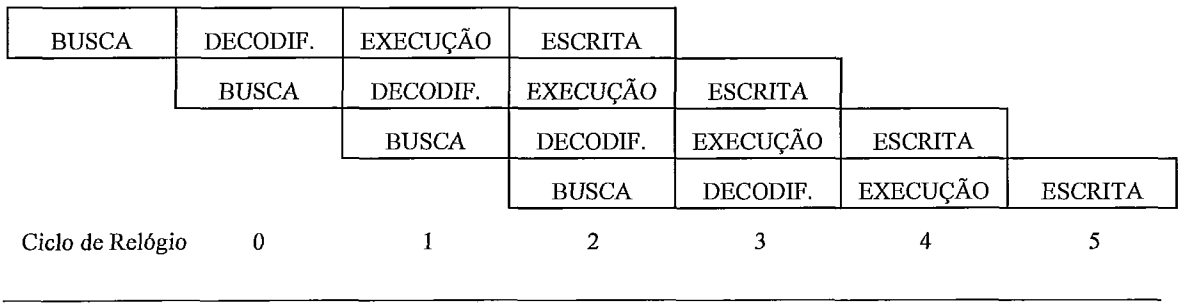


Figura 2.2 - *Pipeline* de instrução

O acesso aos registradores fonte é feito no estágio de decodificação, ao mesmo tempo em que a instrução é decodificada.

Ao estágio de execução cabe, além de efetuar a operação desejada, fazer o cálculo de endereço para operações que envolvam acessos à memória.

Os resultados das operações são escritos nos registradores internos durante o estágio de escrita.

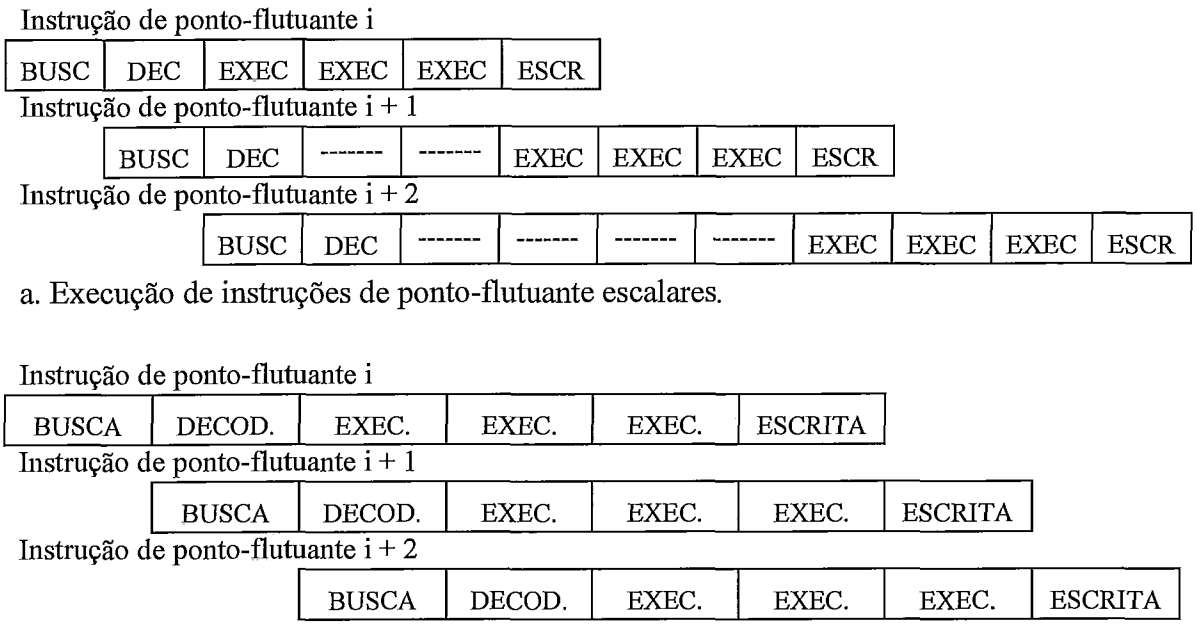
Alguns cuidados são tomados para garantir a ocupação de cada estágio do *pipeline* apenas durante um ciclo de relógio. Todas as instruções possuem tamanho fixo, 32 bits, e o código de operação bem como os conjuntos de bits que identificam os registradores fonte e destino estão localizados sempre na mesma posição dentro da instrução. São estes fatores que permitem que os registradores possam ser acessados durante o estágio de decodificação da instrução.

Quanto ao estágio de execução, nem sempre é possível completá-lo em apenas um período de relógio. Para operações com inteiros, este desempenho pode quase sempre ser alcançado. Operações em ponto-flutuante, ao contrário, não conseguem completar o estágio de execução em apenas um período de relógio, consumindo múltiplos períodos de relógio para completarem seus respectivos estágios de execução. Estes estágios podem ser organizados de duas formas:

- escalar: a instrução é iniciada e passa por todos os múltiplos estágios de execução;

- *pipelined*: os múltiplos estágios de execução formam um *pipeline* de ponto-flutuante, permitindo que uma nova instrução possa ser iniciada sem que a anterior tenha sido completada.

A figura 2.3 esquematiza os dois tipos de execução de operações de ponto-flutuante.



a. Execução de instruções de ponto-flutuante escalares.

b. Execução de instruções de ponto-flutuante *pipelined.*

Figura 2.3. - Tipos de execuções de instruções de ponto-flutuante.

A unidade de ponto-flutuante é composta por uma unidade de adição e uma de multiplicação. Cada uma destas unidades possui um *pipeline* isolado.

A unidade gráfica é uma unidade separada, e possui um outro *pipeline*.

As instruções de ponto-flutuante e gráficas *pipelined* introduzem uma forma de paralelismo que não é transparente para o programador. Os resultados só fluem pelos estágios do *pipeline* quando uma nova operação é introduzida no primeiro estágio do *pipeline.*

O modo de operação inicial do i860 é denominado "modo simples". Neste modo, a cada período de relógio o processador é capaz iniciar apenas uma instrução, mas é capaz de executar uma operação de inteiros, e uma operação de ponto-flutuante ou gráfica, como pode ser visto na figura 2.4. Isto porque operações de inteiros e de ponto flutuante

utilizam unidades funcionais distintas. Se houver dependência de dados entre as duas instruções, não é possível existir sobreposição dos estágios de execução.

Ifp	BUSCA	DECODIF.	EXECUÇÃO	EXECUÇÃO	EXECUÇÃO	ESCRITA
Icore		BUSCA	DECODIF.	EXECUÇÃO	ESCRITA	
Icore			BUSCA	DECODIF.	EXECUÇÃO	ESCRITA
Ciclo de Relógio	0	1	2	3	4	5

Figura 2.4. - Execução de operações de inteiro e ponto-flutuante escalar simultaneamente em modo simples.

No modo simples duas operações podem ser executadas ao mesmo tempo, mas duas instruções não podem ser iniciadas simultaneamente. Com isso, o processador pode atingir, no máximo, desempenho igual a uma instrução por ciclo de relógio.

Objetivando melhorar esse desempenho, o i860 é capaz de executar duas instruções ao mesmo tempo, o que assegura um aumento de desempenho. Este modo de operação é denominado "modo dual" e é controlado explicitamente pelo programador. Para atingir este desempenho, duas instruções são buscadas simultaneamente, uma de inteiros e outra de ponto-flutuante ou gráfica. Como o barramento de acesso à *cache* de instruções possui 64 bits e cada instrução possui um tamanho fixo de 32 bits, é possível efetuar a busca sem qualquer atraso, desde que as duas instruções estejam localizadas em um endereço múltiplo de 8 (8 bytes). A figura 2.5 ilustra o conteúdo do *pipeline* de instrução quando o processador opera em modo dual.

Ifp	BUSCA	DECODIF.	EXECUÇÃO	ESCRITA	
Icore	BUSCA	DECODIF.	EXECUÇÃO	ESCRITA	
Ifp		BUSCA	DECODIF.	EXECUÇÃO	ESCRITA
Icore		BUSCA	DECODIF.	EXECUÇÃO	ESCRITA
Ciclo de Relógio	0	1	2	3	4

Figura 2.5. - Execução de instruções de inteiro e ponto-flutuante escalar simultaneamente em modo dual.

Além de permitir a execução de uma instrução de inteiros e uma de ponto-flutuante simultaneamente, o i860 possui instruções de ponto-flutuante que utilizam ambas as unidades de adição e multiplicação de ponto-flutuante, permitindo que sejam gerados dois resultados de ponto-flutuante a partir de uma única instrução. Estas são denominadas "instruções de operações duais". Dessa forma, é possível obter-se três resultados num único período de relógio quando o i860 estiver operando em modo dual e a instrução de ponto-flutuante for uma instrução de operação dual.

O i860 possui conjuntos de registradores separados para as unidades de inteiros e de ponto-flutuante, além de quatorze registradores de controle e quatro registradores específicos para uso em instruções de operações duais e em instruções gráficas. A figura 2.6 ilustra o conjunto de registradores do i860.

2.2. Unidade Central

A unidade central pode ser considerada o centro de inteligência do i860. Controla todas as suas operações: busca de instruções, decodificação de instruções e execução de instruções aritméticas de inteiros, lógicas, manipulação de bits, *load* e *store* (inclusive de ponto-flutuante), desvio do fluxo de controle, transferência entre registradores de inteiros e de ponto-flutuante, *cache flush* (invalida uma linha da *cache* de dados) e tratamento de exceções.

Essa unidade inclui um conjunto de 32 registradores de inteiros de 32 bits cada, uma unidade lógica e aritmética de 32 bits, e registradores de controle de 32 bits: psr (*processor status register*) e epsr (*extended processor status register*) que armazenam estados do processador; db (*data breakpoint register*) que permite que um *trap* seja gerado quando o processador acessa um operando cujo endereço é igual ao valor armazenado neste registrador; fir (*fault instruction register*) que armazena o endereço de uma instrução quando da ocorrência de um *trap* de instrução; dirbase (***directory base***) que armazena informações de controle para as memórias *cache*, tradução de endereço e opções do barramento de dados; fsr (*floating-point status register*) que contém informações referentes aos possíveis modos de arredondamento e informações do estado do processador quando ocorre um trap causado por um erro em uma operação de ponto-flutuante. Na versão i860 XP encontramos ainda outros registradores de controle: os registradores *privileged*, p0, p1, p2 e p3, para serem utilizados pelo sistema operacional; bear (*bus error address register*) que auxilia o gerenciador de *traps* a determinar erros de acesso à memória; ccr (*concurrency control register*), que controla a operação da unidade

de controle de concorrência; e os registradores NEWCURR e STAT, que são integrados à unidade de controle de concorrência.

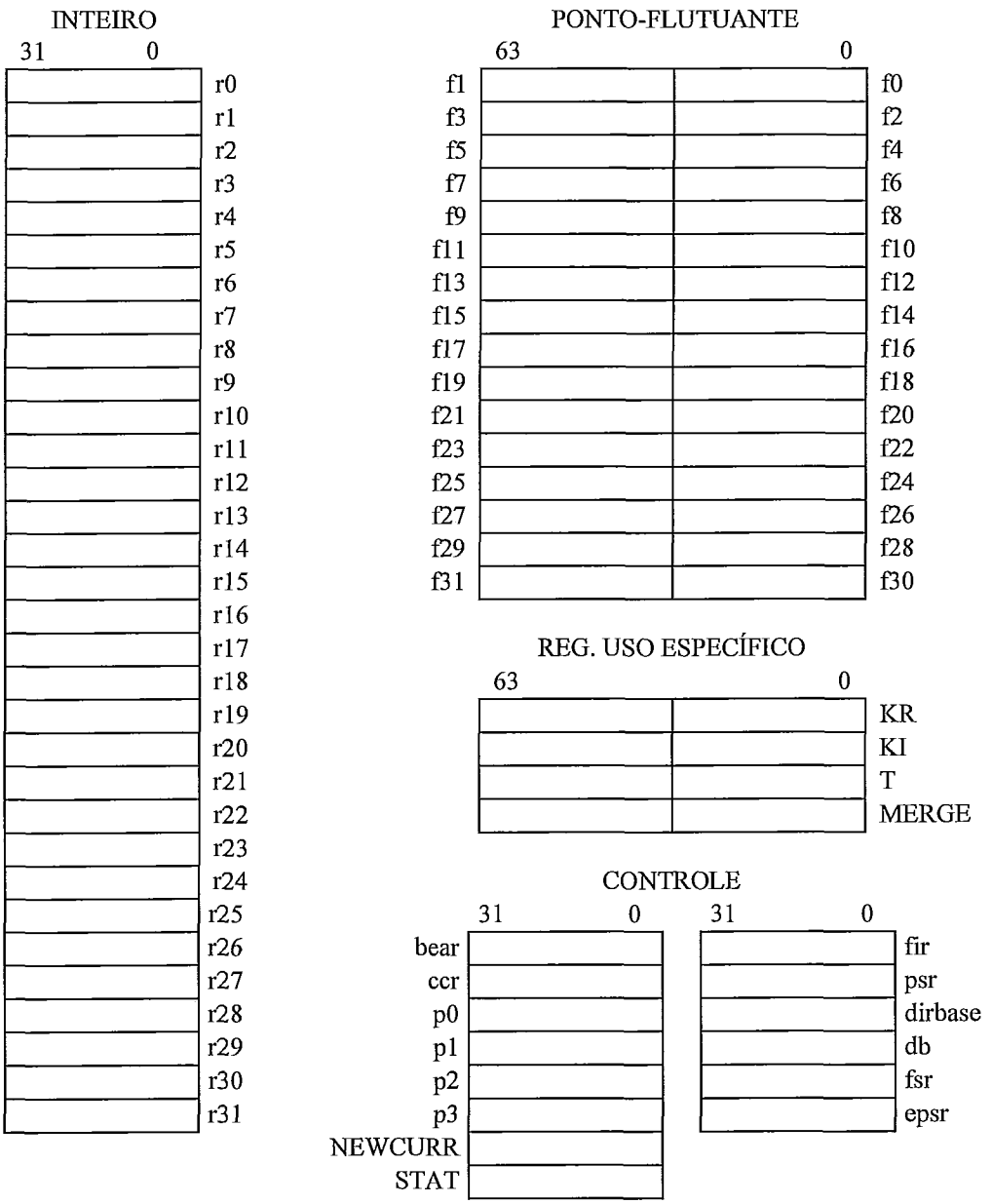


Figura 2.6. - Conjunto de registradores do i860.

Os registradores de inteiros são denominados r0 a r31. Todos os registradores podem ser lidos ou escritos. Contudo, no registrador r0 a escrita é ignorada e a leitura retorna sempre o valor zero.

Os registradores de controle (fir, psr, dirbase, db, fsr, epsr, bear, ccr, p0, p1, p2, p3, NEWCURR e STAT) são lidos e escritos somente por operações de *load* e *store* próprias para acesso a estes registradores.

A busca de instruções é sempre feita diretamente na *cache* de instruções. Quando ocorre uma falta de bloco na *cache*, a instrução é buscada diretamente da memória, ao mesmo tempo em que é carregado todo o bloco na *cache* de instruções. Esta falta provoca um atraso na busca da instrução, que já não pode mais ser concluída em um período de relógio, quebrando o fluxo do *pipeline* de instrução.

Operações de desvio e chamada de subrotinas podem alterar a seqüência de execução das instruções. Para operações de desvio, se o fluxo do *pipeline* for quebrado, o *pipeline* irá requer um período de relógio adicional para reiniciar a busca de instrução na seqüência normal do programa. Se houver uma falta na *cache* na instrução é necessário buscar a instrução na memória e o *pipeline* de instrução suspende a execução. Todo o cálculo de endereço alvo para operações que alteram a seqüência de execução das instruções é feito por uma unidade de soma dedicada.

As únicas operações de memória permitidas no i860 são *load* e *store*. Todas as outras operações utilizam como operandos valores contidos nos registradores. Operações de *load* requerem um mínimo de dois períodos de relógio para serem executadas. Se o registrador destino for utilizado na instrução seguinte, o fluxo do *pipeline* de instrução é interrompido, como mostra a figura 2.7.a. A figura 2.7.b. ilustra um caso em que o registrador destino de uma operação de *load* não é utilizado pela instrução seguinte, e o *pipeline* de instrução flui normalmente.

Quando é feita uma operação de *load* ou *store*, o dado é inicialmente procurado na *cache* de dados. Caso não seja encontrado é constatada uma ocorrência de falta na *cache* de dados. Neste caso, o dado é buscado da memória externa junto com o restante da linha da *cache*. A Operação de *load* é encerrada quando o dado é lido, mas a *cache* continua ocupada até que o preenchimento da linha tenha sido concluído. Somente operações de *load* são capazes de preencher uma linha da *cache*. As operações de *store* que faltam na *cache* escrevem os dados diretamente em *buffers* de escrita para posterior escrita na memória externa, liberando a unidade central e a *cache* para outras operações. O i860 dispõe de três *buffers* de escrita, de 128 bits cada um.

Operações de *store* nas quais os dados são encontrados na *cache* utilizam-na por dois períodos de relógio. Quando existem operações de *store* consecutivas e os dados são encontrados na *cache*, é possível fazer um *store* a cada período de relógio.

ld.l address,r5	BUSCA	DECODIF.	EXECUÇÃO	EXECUÇÃO	ESCRITA	
addu r5,r5,r5		BUSCA	DECODIF.	----------	EXECUÇÃO	ESCRITA
addu r6,r7,r8			BUSCA	----------	DECODIF.	EXECUÇÃO
Ciclo de Relógio	0	1	2	3	4	5

a. a instrução que segue a instrução de *load* utiliza o registrador destino do *load*.

ld.l address,r5	BUSCA	DECODIF.	EXECUÇÃO	EXECUÇÃO	ESCRITA	
addu r6,r7,r8		BUSCA	DECODIF.	EXECUÇÃO	ESCRITA	
addu r5,r5,r5			BUSCA	DECODIF.	EXECUÇÃO	ESCRITA
Ciclo de Relógio	0	1	2	3	4	5

b. A instrução que segue a instrução de *load* não utiliza o registrador destino do *load*.

Figura 2.7. - *Pipeline* de instrução com operação de *load*.

A unidade central também é responsável pela execução das instruções de *load* e *store* de ponto-flutuante: fld (*floating-point load*), fst (*floating-point store*) e pfld (*pipelined floating-point load*). A diferença entre as duas instruções de *load* de ponto-flutuante reside no fato de que, quando ocorre uma falta na cache de dados, uma instrução fld preenche uma linha na cache de dados ao mesmo tempo em que transfere o dado da memória para um registrador de ponto-flutuante, enquanto uma instrução pfld apenas transfere o dado da memória para um registrador de ponto-flutuante.

A instrução pfld utiliza um *pipeline* de três estágios para acessos ao barramento externo, garantindo até três acessos simultâneos no barramento. Por não armazenar os dados lidos na *cache* de dados, a utilização de pflds é interessante quando se deseja operar sobre um volume muito grande de dados que não cabem na *cache*. Acessos de fld podem acarretar a substituição de linhas, congestionando o barramento com acessos de escrita da linha de volta na memória.

2.3. Unidade de Ponto Flutuante

A unidade de ponto-flutuante consiste de uma unidade de controle de ponto-flutuante, um banco de registradores de ponto-flutuante, uma unidade de soma de ponto-flutuante, uma unidade de multiplicação de ponto-flutuante e três registradores para armazenar valores intermediários em operações duais de ponto-flutuante.

Para aumentar o desempenho de suas operações, que não podem ser executadas em apenas um período de relógio, a unidade de ponto-flutuante utiliza paralelismo. Um tipo de paralelismo usado é o *pipeline*, onde cada operação é tratada como uma série de operações mais primitivas, cada uma correspondendo a um estágio. Cada estágio do *pipeline* armazena resultados intermediários e informação de estados relacionados a estes resultados. O número de estágios de *pipeline* de execução da operação varia de 2 a 3.

Existem dois *pipelines* de ponto-flutuante: um para multiplicação e um para adição. O *pipeline* de adição possui três estágios de um período de relógio cada. O número de estágios no *pipeline* de multiplicação depende da precisão dos operandos fonte: três estágios de um período de relógio para operandos de precisão simples e dois estágios de dois períodos de relógio para operandos de precisão dupla. O resultado do último estágio de cada *pipeline* é armazenado no registrador destino da instrução que acaba de entrar no *pipeline*. Para cada nova instrução *pipelined*, o *pipeline* avança um estágio, produzindo um resultado a cada ciclo.

Além de execução em *pipeline*, as operações em ponto-flutuante também podem ser executadas em modo escalar. Neste modo, a execução de operações em ponto-flutuante não pode se sobrepor dentro do *pipeline*. A execução de uma operação escalar passa por todos os estágios do *pipeline* antes que uma nova operação seja introduzida. Desta forma, qualquer resultado que esteja sendo gerado pelo *pipeline* e ainda não tenha sido armazenado é perdido. Para que isto seja evitado, as últimas operações dentro do *pipeline* antes que se execute uma operação escalar devem ser operações *pipelined* que permitam retirar estes resultados de dentro do *pipeline*. Após a execução de um operação escalar, todos os estágios do *pipeline* contêm valores indefinidos. Normalmente o modo escalar é utilizado quando a instrução seguinte depende do resultado da instrução anterior ou quando se deseja evitar a "complexidade" do *pipeline*.

A maioria das instruções de ponto-flutuante possuem um versão escalar e outra *pipelined*.

A figura 2.8 ilustra os três estágios do *pipeline* de adição. Cada estágio armazena resultados intermediários e informações do estado do processador referente a estes

resultados intermediários. Quando a primeira instrução é executada na figura 2.8, o *pipeline* de adição está vazio e os três primeiros resultados são descartados (escritas no registrador f0 são descartadas). A cada instrução, o *pipeline* avança um estágio. Na quarta instrução, o registrador destino recebe o resultado da primeira instrução e, a partir de então, é obtido um resultado a cada ciclo. O tempo destinado a encher e esvaziar o *pipeline* torna-se desprezível quando é realizado um número muito grande de instruções.

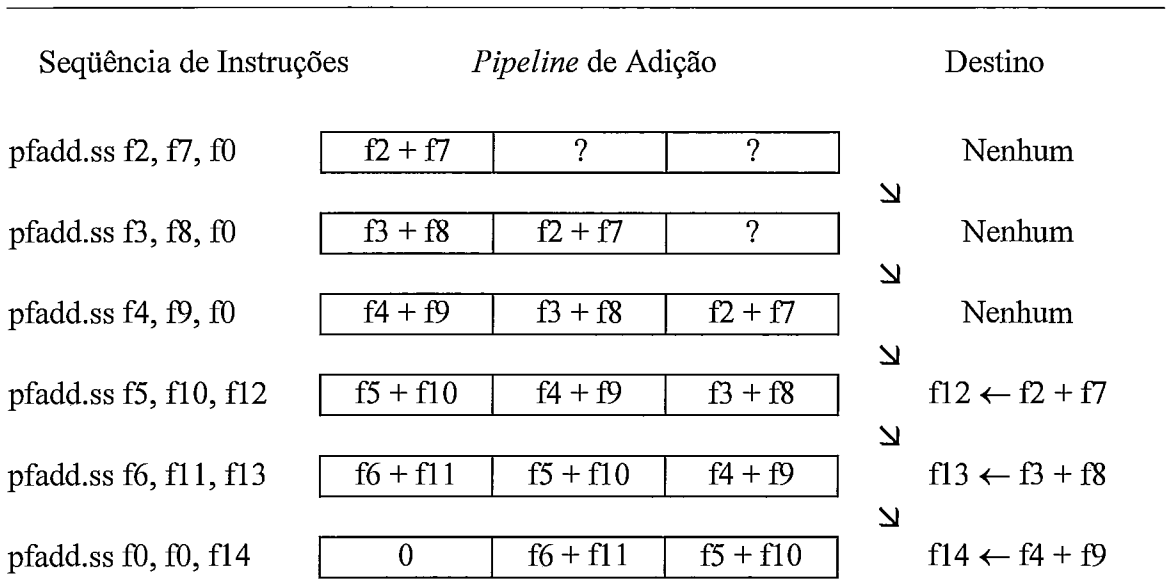


Figura 2.8. - Exemplo do *pipeline* de adição.

O *pipeline* de multiplicação opera de modo semelhante ao de adição quando todos os operandos são de precisão simples. Quando ocorre uma variação de precisão, de simples para dupla ou de dupla para simples, alguns resultados podem ser perdidos, uma vez que o *pipeline* de multiplicação possui três estágios para precisão simples e dois estágios para precisão dupla.

A execução de uma seqüência de instruções de multiplicação *pipelined* com operandos de precisão diferentes pode ser observada na figura 2.9. Inicialmente o *pipeline* está vazio. As duas primeiras instruções, de precisão simples, operam como no *pipeline* de adição. O comportamento é modificado quando a terceira instrução possui operandos de precisão dupla, e o *pipeline* de multiplicação passa a ter somente dois estágios. Podemos continuar enxergando o *pipeline* como tendo três estágios, só que uma instrução permanece por dois períodos de relógio no primeiro estágio, e outra instrução permanece nos segundo e terceiro estágios, simultaneamente, também por dois períodos de relógio. A

operação que acaba de entrar no *pipeline* fica no primeiro estágio e uma nova operação só pode ser executada após dois ciclos de relógio. A operação que estava no primeiro estágio passa a ocupar o segundo e terceiro estágios. A operação que estava no segundo estágio é descartada. Este comportamento é observado quando encontramos a instrução que faz o produto de f3 por f8 no segundo e terceiro estágios, simultaneamente. Quando passamos a ter novamente instruções com operandos de precisão simples, voltamos a ter um *pipeline* de multiplicação com três estágios. Neste caso, o primeiro estágio recebe a instrução que está entrando, o segundo estágio recebe a operação que estava no primeiro estágio e o terceiro estágio recebe o valor zero.

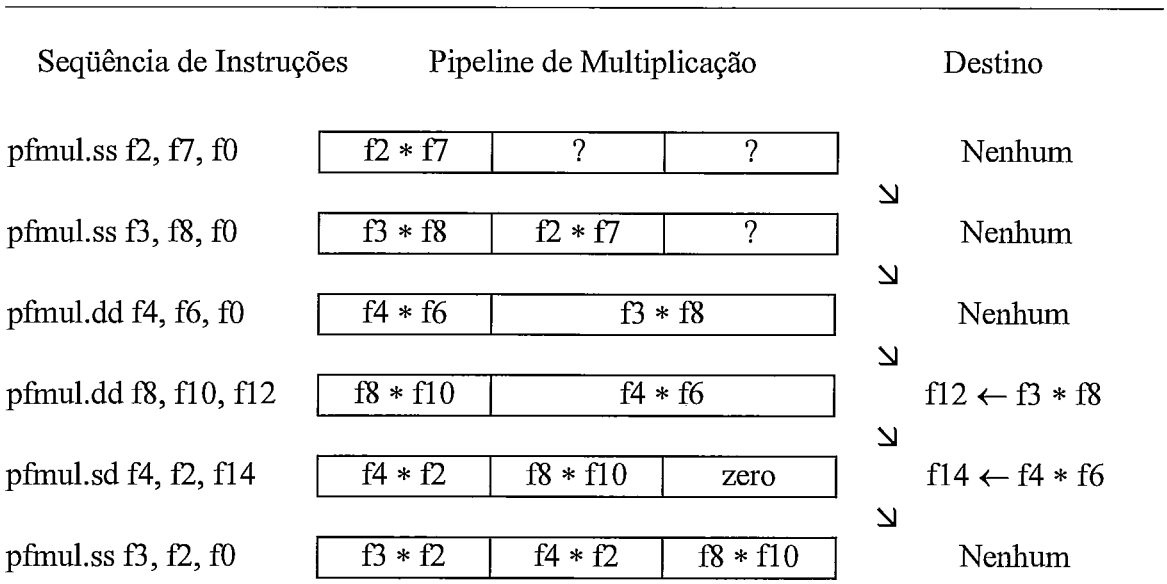


Figura 2.9. - Exemplo do *pipeline* de multiplicação com operandos de precisão variadas.

Tendo em vista que as transições envolvidas em operações com precisões diferentes tornam difícil o controle do *pipeline*, misturas de precisão de operandos para operações de multiplicação devem se evitadas. Como isto nem sempre é possível, muita atenção é necessária para que resultados importantes não sejam perdidos.

No caso de operações escalares, a conseqüência da mudança de precisão dos operandos reside apenas no fato de que a instrução requer um pouco mais de tempo para ser executada.

As instruções observadas nas figuras 2.8 e 2.9 permitem que uma operação possa ser executada sem que uma operação anterior termine. Contudo, a operação relacionada à

instrução pfadd, por exemplo, não pode avançar no *pipeline* de adição ao mesmo tempo que a operação relacionada à instrução pfmul avança no *pipeline* de multiplicação. Para que isto aconteça, o i860 provê instruções de operação dual, uma outra forma de paralelismo. Numa única instrução as unidades de soma e de multiplicação operam em paralelo, duplicando o número de operações de ponto flutuante efetuadas. A figura 2.10 ilustra as unidades de soma e adição e os caminhos necessários para operações duais.

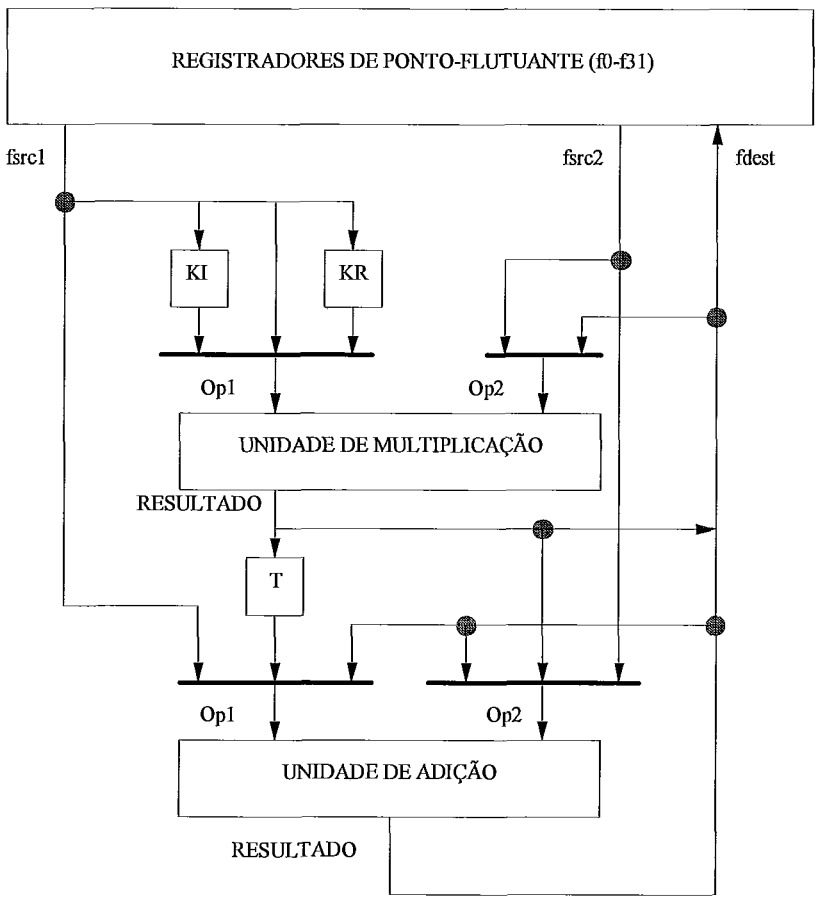


Figura 2.10. - Estrutura interna do i860 para a execução de instruções de operações duais.

As operações duais possibilitam a realização de operações de soma-e-multiplicação e subtração-e-multiplicação. Para estes casos específicos de instruções, três registradores especiais são utilizados: KR, KI e T. Os registradores KR e KI podem ser usados para armazenar constantes ou valores temporariamente, alimentando o *pipeline* de multiplicação. O registrador T armazena o valor do resultado da multiplicação, que pode ser passado como operando para numa operação futura de soma ou subtração.

Podemos encontrar ainda outra forma de paralelismo, na qual é possível iniciar uma instrução de inteiro e outra de ponto-flutuante simultaneamente: o "modo dual". A ativação e desativação do modo dual é controlada por *software*. As transições do modo dual podem ser vistas na figura 2.11.

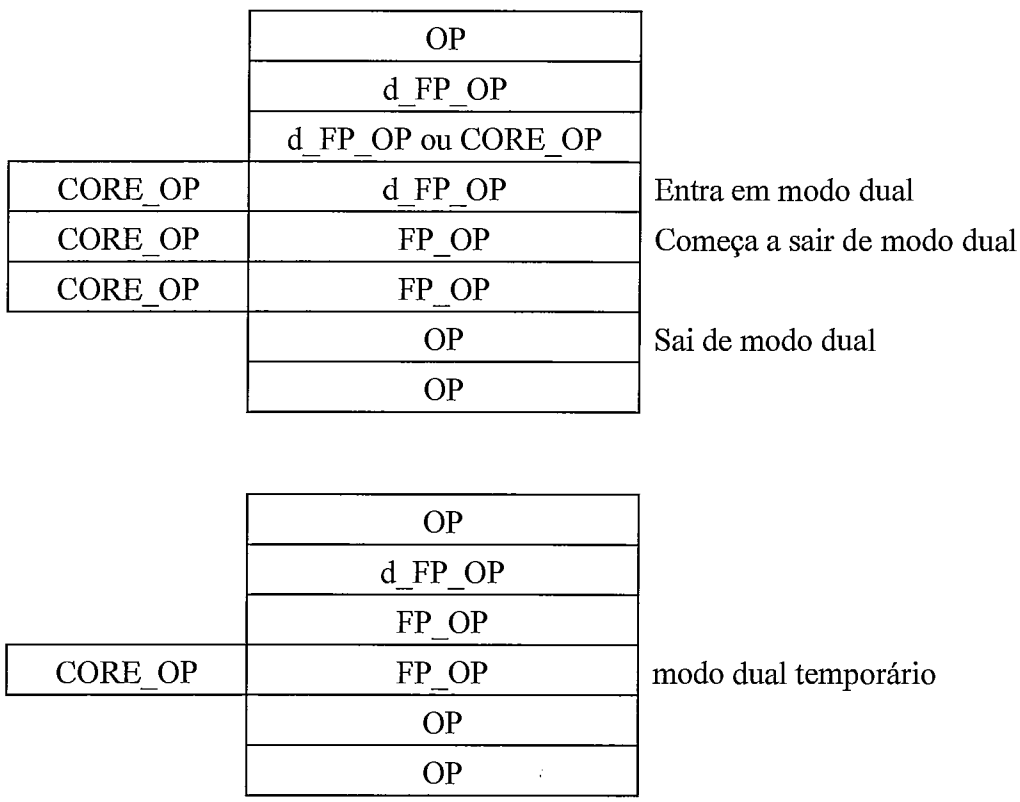


Figura 2.11. - Transições de modo dual.

Quando o i860 observa uma instrução de ponto-flutuante dual, imediatamente inicia a transição de modo simples para modo dual. Se a instrução seguinte for uma instrução de ponto-flutuante dual ou uma instrução executada pela unidade central, o i860 entra então em modo dual e executa as operações seguintes como pares de operações. Quando um par de operações contiver uma instrução de ponto-flutuante sem ser dual, o i860 executa mais um par de operações e sai de modo dual. Esta situação é mostrada na

figura 2.11.a. A figura 2.11.b. mostra o caso de uma seqüência de apenas um par de operações em modo dual.

O modo dual permite que a busca e a escrita de operandos, atualização de índices de vetor e controle de *loops* possam ser feitas em paralelo com a execução de instruções de ponto-flutuante.

A unidade de controle de ponto-flutuante controla as unidade de soma e multiplicação de ponto-flutuante, enviando instruções, verificando a ocorrência de exceções de fonte ou resultados e atualizando os bits de estado no fpsr (*floating-point status register*).

O banco de registradores contem 32 registradores de ponto-flutuante de 32 bits cada, denominados f0 a f31. Os registradores também podem ser referenciados aos pares, para trabalhar com valores em precisão dupla. Neste caso, podem ser usados somente registradores pares (f0, f2, f4, f6, f8,...). Instruções de *load* e *store* (fld, fst, pfld) também suportam transferência de dados de 128 bits, utilizando grupos de quatro registradores (f0, f4, f8, f12, f16, ...). Todos os registradores podem ser lidos ou escritos, contudo no caso dos registradores o f0 e f1 a escrita é ignorada e a leitura retorna sempre o valor zero.

A unidade de soma de ponto-flutuante efetua soma, subtração, comparação e conversão sobre valores de 32 bits e 64 bits.

A unidade de multiplicação de ponto-flutuante efetua operações de multiplicação de ponto-flutuante e de inteiros, bem como operações para calcular o inverso de um valor e o inverso da raiz quadrada de um valor, sobre operandos de precisão simples ou dupla.

2.4. Unidade Gráfica

A unidade gráfica possui uma lógica de inteiros de 64 bits que suporta algoritmos gráficos em três dimensões. Esta unidade pode operar em paralelo com a unidade central. O i860 é capaz de reconhecer *pixels (picture elements)* como um dado de 8, 16 ou 32 bits. É possível computar valores de intensidade das cores vermelha, azul ou verde num *pixel*.

Um registador de propósito especial denominado MERGE, de 64 bits, auxilia na paralelização de algoritmos gráficos.

2.5. Unidade de Gerenciamento de Memória

A memória do i860 XP é acessada em bytes com um espaço de endereçamento virtual de 2^{32} bytes. Qualquer dado pode estar localizado em qualquer lugar neste espaço de endereçamento. A aritmética de endereço utiliza entradas de 32 bits e produz saídas de 32 bits. Em caso de *overflow* são considerados os 32 bits menos significativos.

A unidade de gerenciamento de memória implementa as características básicas de memória virtual paginada incluindo a proteção de memória no nível de páginas.

O i860 suporta dois tamanhos de páginas: 4 K bytes e 4 M bytes. A primeira, compatível com o i860 XR, possui uma estrutura de diretório de páginas em dois níveis e tabelas de páginas com entradas de 1 K cada. Páginas de 4 M não utilizam o segundo nível de tabela de páginas. Estes dois tamanhos de páginas podem ser usados juntos em qualquer combinação.

A unidade de gerenciamento de memória efetua a tradução de endereço do espaço de endereçamento lógico para o espaço de endereçamento físico, para acesso tanto a dados como a instruções. A tradução de endereço é opcional e, quando habilitada, utiliza tabelas de páginas. Inicialmente o i860 encontra-se com a tradução desabilitada.

Para diminuir o *overhead* causado pelos acessos às tabelas de páginas, o processador guarda as informações de tradução referentes às 64 páginas de 4 K bytes e 16 páginas de 4 M bytes mais recentemente utilizadas em duas memórias associativas (*four-way set-associative*), chamadas TLB (*Translation Lookaside Buffer*). Estas memórias atuam como uma cache de tradução de endereços. Quando o processador não encontra a informação de tradução para uma determinada página na TLB, ele procura na tabela de páginas, localizada na memória externa, e atualiza a TLB.

2.6. *Caches* Internas

Além das TLBs, o i860 possui duas memórias *cache* internas, uma de instrução e outra de dados, que podem operar em paralelo. Ambas as *caches* são endereçadas virtualmente, o que provê uma rápida operação uma vez que o acesso às TLBs é feito em paralelo ao acesso à *cache*.

O preenchimento de uma linha na *cache* só é feito quando ocorre uma falta de bloco da *cache*. A técnica utilizada para carregar um linha da *cache* é conhecida como *wrap-around*. Inicialmente o processador lê os oito bytes que contêm o dado solicitado,

uma instrução ou par de instruções necessários para o processador. Dispondo disto, o i860 continua seu processamento e, paralelamente a informação lida é armazenada na *cache*. Em seguida, efetua a leitura dos outros 24 bytes necessários para completar a linha da *cache* que está sendo carregada.

Uma linha da *cache* pode ser totalmente preenchida com quatro transferências em *burst bus*. Neste caso, um novo dado (com largura de 64 bits ou 8 bytes) pode ser lido pelo i860 XP a cada período de relógio.

A *cache* de instruções (*four-way set-associative*) tem 16 Kbytes de tamanho e blocos de 32 bytes, podendo transferir até 64 bits por período de relógio. A *cache* de dados (*four-way set-associative*), por outro lado, tem 16 Kbytes de tamanho e bloco de 32 bytes, podendo transferir até 128 bits por período de relógio.

Escrita em posições de memória que não estejam presentes na *cache* atualizam diretamente a memória. Quando presente na *cache*, a política de atualização da memória normalmente usada é o *write back*, onde a memória somente é atualizada quando a linha na *cache* é substituída por outros dados, reduzindo o tráfego no barramento externo. Contudo, pode-se adotar a política de *write through*, que sempre atualiza a *cache* e a memória, ou utilizar a política *write once*, uma mistura das políticas *write back* e *write through*, onde a política *write through* é adotada na primeira escrita na memória e, a partir de então, é adotada a política *write back*. A política *write once* é bastante interessante pois garante consistência na *cache* com acesso mínimo à memória, em caso de multiprocessamento. Assim, a primeira escrita difunde para os outros processadores a informação de que a posição de memória em questão foi modificada.

Quando um conjunto da cache está com todas as linhas válidas e uma nova linha precisa ser carregada, a escolha da linha que será descartada é aleatória.

Qualquer área de memória pode ser colocada na *cache*. Contudo, tanto o *hardware* quanto o *software* são capazes de impedir que determinadas áreas sejam armazenadas na *cache*.

A *cache* de dados pode ser totalmente invalidada através de instruções de *flush*, que invalidam um bloco da *cache* a cada instante.

2.7. Unidade de Controle de *Cache* e do Barramento

A unidade de controle de *cache* e do barramento efetua acessos de dados e instruções para a unidade central. Ela recebe os pedidos e as especificações da unidade central, realiza o processamento de falta na *cache* de dados e de instrução, controla a TLB e faz a interface com o barramento externo. Sua estrutura em *pipeline* suporta ate três ciclos de barramento pendentes.

Esta unidade também controla a utilização dos três *buffers* de escrita que permitem que uma operação de escrita possa ser atrasada caso o barramento esteja ocupado com outro acesso. O dado a ser escrito é armazenado em um dos *buffers* e a unidade central é liberada para executar outra instrução. Assim que possível o dado é copiado do *buffer* de escrita para a memória externa.

2.8. Unidade de Controle de Concorrência

O i860 suporta processamento paralelo, onde múltiplos processadores trabalham simultaneamente em diferentes partes de um mesmo problema.

A unidade de controle de concorrência (DCCU - "*Detached concurrency control unit*") é um subconjunto compatível com uma unidade de controle de concorrência externa que controla o paralelismo e o sincronismo a nível de *loop* sistemas de multiprocessadores.

Capítulo 3

Características do Simulador

3.1. Introdução

O simulador SIM860 foi desenvolvido para reproduzir as características do processador i860, da Intel, com o ojetivo de proporcionar uma ferramenta para análise e otimização de código para este processador. Oferecendo ao usuário condições para efetuar uma análise temporal de execução, o SIM860 permite que sejam identificadas situações de conflitos de recursos no programa, que impedem que o i860 opere com desempenho máximo.

A análise temporal do programa a ser executado pelo i860 é fornecida através de um relatório de saída (gerado pelo SIM860) e de gráficos (gerados pelo programa SHOWREPO) que informam acerca da utilização dos recursos do processador durante toda a execução de um programa. Embora existam vários produtos relacionados ao I860 fornecidos pela Intel, inclusive um simulador, nenhum deles permite que se acompanhe o comportamento interno dos diversos recursos do processador a cada ciclo de relógio.

O processador i860 atualmente dispõe de compiladores para linguagens de alto nível, produzidos pela Intel e pela Metaware. Estes compiladores são de grande utilidade frente à dificuldade em se programar um processador superescalar RISC. Embora utilizem técnicas para otimização de código, estes compiladores geram códigos que ainda poderiam sofrer otimizações. As informações fornecidas pelo simulador auxiliam o desenvolvimento de compiladores, permitindo que o desempenho do código gerado possa ser avaliado.

Este capítulo tem como objetivo descrever as características do simulador, bem como do programa "SHOWREPO", que permite a construção de gráficos a partir de uma das saídas do simulador.

O simulador oferece ao usuário um relatório contendo informações a respeito da utilização de recursos durante cada ciclo de relógio da execução do programa. De posse disto, é possível identificar trechos de programa que impedem que o i860 opere devido a indisponibilidade de recursos. Identificando-se a localização dos pontos de conflito, é possível fazer uma análise e verificar se uma reorganização das instruções do programa pode oferecer algum ganho.

3.2. Parâmetros de Entrada

A entrada do simulador é composta por um arquivo executável, em formato COFF (*Common Object File Format*), definido no *Unix System V/386 Programer's Guide*. O formato COFF é encontrado nas saídas dos montadores e linkers, com pequenas modificações necessárias para o I860[3]. Um conjunto de parâmetros opcionais de entrada permitem ao usuário escolher quais saídas deseja que sejam geradas pelo simulador.

O arquivo de entrada executável elimina a utilização de um outro programa para gerar uma sequência de instruções, facilitando o trabalho do usuário.

Os parâmetros de entrada permitidos pelo simulador são:

- -r1= arquivo de relatório completo
- -r2= arquivo de relatório simplificado
- -m1= tempo do primeiro acesso à memória
- -m2= tempo dos demais acessos à memória
- -d= arquivo de *dump* de memória

O usuário pode optar por um relatório completo (-r1) ou simplificado (-r2). O relatório simplificado armazena num arquivo somente um conjunto de informações globais. O relatório completo armazena o estado de todos os recursos do simulador a cada período de relógio da execução do programa, além de todas as informações fornecidas pelo relatório simplificado. Ambos os relatórios são armazenados em um arquivo e não podem ser utilizados conjuntamente.

As opções -m1 e -m2 permitem ao usuário alterar o tempo de acesso à memória externa, em número de ciclos de relógio. Como o simulador implementa o i860 XP, os tempos de acesso do simulador inicialmente escolhidos são os tempos mínimos permitidos pelo i860 XP. Com estas opções, é possível utilizar a temporização da versão XR ou fazer uma análise de como os acessos à memória podem estar interferindo no

desempenho de um programa. Estas duas opções decorrem da propriedade do barramento do i860 suportar *pipelining* e, com isso, um acesso pode ser iniciado antes que outro termine. O parâmetro m1 retrata o tempo, em ciclos de relógio, que o primeiro acesso leva para ser concluído; m2 retrata o tempo, em ciclos de relógio, que o próximo acesso leva para ser concluído a partir do término do acesso anterior, em *pipelining*. Para o i860 XP estes tempos mínimos são, respectivamente, 2 ciclos e 1 ciclo de relógio. Estes parâmetros influenciam principalmente em acessos de busca de linha para as memórias *cache*, que são sempre feitos em *pipelining*.

O parâmetro -d permite ao usuário gerar um *dump* de memória e armazená-lo num arquivo. Neste arquivo é apresentado o conteúdo da memória utilizada pelo i860 antes e depois da execução de programa. Estas informações permitem ao usuário verificar se as possíveis alterações feitas na memória estão de acordo com o esperado pelo programa.

O nome de todos os arquivos de saída podem ser escolhidos pelo usuário. Apenas como sugestão, as extensões .r1, .r2 e .dmp são interessantes para serem usadas nas opções de geração de relatório completo (-r1), simplificado (-r2) e dump de memória (-d), respectivamente.

Quando a opção -r1 é escolhida, uma alteração no programa fonte deve ser feita. Como o relatório completo apresenta informações a cada ciclo de relógio, este normalmente tende a ficar muito extenso, tirando de foco o trecho de programa que merece atenção. Para resolver este problema, uma instrução "trap r5,r5,r0" deve ser colocada no início do trecho de programa que se deseja enfocar, e uma instrução "trap r6,r6,r0" no final do trecho. Desta forma, a análise minuciosa contida no relatório completo pode ficar restrita a um trecho de programa. Caso se deseje gerar um relatório de todo o programa, estas duas instruções devem estar localizadas no ponto de início e no ponto de término de execução do programa, respectivamente. Vale ressaltar que uma instrução "trap r6,r6,r0" não exerce qualquer efeito caso não exista anteriormente uma instrução "trap r5,r5,r0" para habilitar o início da geração de relatório. Um programa pode ter vários trechos diferentes delimitados por estas instruções.

3.3. Saídas do simulador

Na seção anterior foram descritos os parâmetros de entrada do simulador, possibilitando a identificação de algumas das saídas do simulador. Nesta seção serão apresentados os formatos destas saídas, indicando quais informações são encontradas em cada uma das saídas.

3.3.1. Informações na tela

Durante a simulação, são testadas diversas situações de erro que interrompem a execução do programa caso sejam encontradas algumas destas situações. O simulador não premite que a simulação prossiga, facilitando a identificação do erro. Antes de encerrar a simulação é impressa na tela uma mensagem de término de programa, mesmo que a simulação tenha sido bem sucedida. Esta mensagem fornece um código de término de programa, que possui valor zero quando a simulação é feita com sucesso. O conjunto de códigos de término de programa é encontrado no apêndice B.

3.3.2. Arquivo temporário

Durante a geração de um relatório, o simulador cria um arquivo de registros temporário (RELATORI.LOG). Cada registro armazena informações sobre os recursos utilizados em um ciclo de relógio, além de informações sobre as instruções que se encontram em cada estágio de execução do pipeline de instrução. Este arquivo não é destruído ao término da simulação e é usado pelo programa "SHOWREPO" para a construção de gráficos. No entanto este arquivo é sobre-escrito cada vez que o programa SIM860 é executado com alguma opção de relatório selecionada. A figura 3.1 mostra a estrutura dos registros encontrados no arquivo temporário.

Quando um relatório simplificado é gerado, o arquivo temporário contém somente as informações globais, não podendo ser utilizado como entrada para o programa SHOWREPO.

3.3.3. Relatórios

Caso o usuário tenha entrado com o parâmetro -r2, é gerado um relatório simplificado contendo apenas informações globais da simulação. A figura 3.2 ilustra um relatório de saída simplificado.

Quando o parâmetro de entrada é -r1, é gerado um relatório completo do simulador, somente para a execução das instruções localizadas entre a execução de uma instrução de trap r5,r5,r0 e trap r6,r6,r0. Uma ilustração da estrutura de um relatório detalhado pode ser vista na figura 3.3. Um relatório completo sempre apresenta, no final do arquivo, as informações contidas num relatório simplificado.

```
typedef struct {
    unsigned long    dFetchInstr ,      /* instrução no estágio de fetch                */
                     dDecodeInstr ,     /* instrução no estágio de decodificação        */
                     dExecuteInstr,     /* instrução no estágio de execução             */
                     dWriteInstr ;      /* instrução no estágio de escrita              */
    unsigned long    dClk       ;       /* período de relógio                           */
    unsigned char    bCoreActivity,     /* utilização da unidade central                */
                     bScalarFpActivity, /* utilização da unidade de ponto-flut.         */
                     abFpPAddActivity[3], /* utilização do pipeline de adição           */
                     abFpPMplyActivity[3], /* utilização do pipeline de multiplicação   */
                     abFpGrphActivity,  /* utilização da unidade gráfica                */
                     bInstrCacheHit,    /* hit na cache de instrução                    */
                     bInstrCacheMiss,   /* falta na cache de instrução                  */
                     bDataCacheHit,     /* hit na cache de dados                        */
                     bDataCacheMiss,    /* falta na cache de dados                      */
                     bCacheNotAvailable, /* cache de dados ocupada com acesso           */
                     bMemAccess,        /* barramento externo com acesso                */
                     bOperationMode;    /* modo de operação                             */
} tgDetailedReportLineInfo ;
```

Figura 3.1. - Estrutura dos registros armazenados no arquivo temporário relatori.log.

RELATÓRIO RESUMIDO DO SIMULADOR

Número total de clocks		7437
Número total de instrucões	7079	
Número de clocks que a core unit ficou ocupada	5254	
Número de clocks que a core unit ficou ocupada com load		2354
Número de clocks que a core unit ficou ocupada com store		74
Número de clocks que a FP unit ficou ocupada		5040
Número de clocks que a unidade de adição de PF ficou ocupada		4536
Número de clocks que a unidade de multiplicação de PF ficou ocupada		1440
Número de clocks que a unidade gráfica de PF ficou ocupada		144
Número de Instruction Cache Miss		8
Número de Instruction Cache Hit		7071
Número de Data Cache Miss		96
Número de Data Cache Hit		1132
Número de avanços no pipeline de adição		4536
Número de avanços no pipeline de multiplicação	1440	
Número de avanços no pipeline da unidade gráfica	0	

Figura 3.2. - Exemplo de relatório simplificado do simulador.

RELATÓRIO DE SAÍDA DO SIMULADOR

INSTRUCTION PIPELINE				CLK	SRC1	SRC2	DEST	CORE	FP	FPADD			FPMPLY			GRPH	ICH	ICM	DCH	DCM	CNA	MA	OPM
FTCH	DEC	EXEC	WR	-	-	-	-	-	-	S1	S2	S3	S1	S2	S3	-	-	-	-	-	-	-	-

Número total de clocks
Número total de instruções
Número de clocks que a core unit ficou ocupada
Número de clocks que a core unit ficou ocupada com load
Número de clocks que a core unit ficou ocupada com store
Número de clocks que a FP unit ficou ocupada
Número de clocks que a unidade de adição de PF ficou ocupada
Número de clocks que a unidade de multiplicação de PF ficou ocupada
Número de clocks que a unidade gráfica de PF ficou ocupada
Número de Instruction Cache Miss
Número de Instruction Cache Hit
Número de Data Cache Miss
Número de Data Cache Hit
Número de avancos no pipeline de adição
Número de avancos no pipeline de multiplicação
Número de avancos no pipeline da unidade gráfica

Figura 3.3.- Formato do relatório de saída do simulador.

Cada ítem constante do cabeçalho do relatório simplificado é identificado da seguinte forma:

- Instruction Pipeline:
  - FTCH - Estágio de busca da instrução: indica qual instrução está sendo buscada no período de relógio indicado por CLK;
  - DEC - Estágio de decodificação da instrução: indica qual instrução está sendo decodificada no período de relógio indicado por CLK;
  - EXEC - Estágio de execução da instrução: indica qual instrução está sendo executada no período de relógio indicado por CLK;
  - WR - Estágio de escrita: indica qual instrução está escrevendo nos registradores destino.
- CLK - Indica o período de relógio corrente.
- SRC1 - Indica o registrador fonte 1 utilizado na instrução que está sendo executada.
- SRC2 - Indica o registrador fonte 2 utilizado na instrução que está sendo executada.
- DEST - Indica o registrador destino utilizado pela instrução que está sendo executada.
- CORE - Indica se a unidade central está sendo usada para executar uma instrução.
- FP - Indica que a unidade de ponto-flutuante está sendo usada para executar uma instrução.
- FPADD - Indica se a unidade de ponto-flutuante de adição está sendo usada.
  - S1 - Primeiro estágio do *pipeline* de adição.
  - S2 - Segundo estágio do *pipeline* de adição.
  - S3 - Terceiro estágio do *pipeline* de adição.
- FPMPLY - Indica se a unidade de ponto-flutuante de multiplicação está sendo usada.
  - S1 - Primeiro estágio do *pipeline* de multiplicação.
  - S2 - Segundo estágio do *pipeline* de multiplicação.
  - S3 - Terceiro estágio do *pipeline* de multiplicação.
- GRPH - Indica se a unidade gráfica está sendo usada.
- ICH - Indica se houve *hit* na *cache* de instrução.
- ICM - Indica se houve *miss* na *cache* de instrução.
- DCH - Indica se houve *hit* na *cache* de dados.
- DCM - Indica se houve *miss* na *cache* de dados.
- MA - Indica se está havendo acesso à memória externa.
- CNA - Indica que a cache de dados não está disponível para um acesso.
- OPM - Indica o modo de operação: S indica modo de instrução simples e D indica modo de instrução dual.

3.3.4. *Dump* de memória

O parâmetro de entrada -d permite que o simulador armazene num arquivo o conteúdo de toda a memória utilizada pelo programa, antes e após sua execução. A figura 3.4 apresenta um exemplo de como é mostrado o conteúdo de memória.

Se o usuário entrar com a opção -d mas não determinar o nome do relatório de saída, o simulador não gera o arquivo de *dump*.

```
Allocating Text Section:
Memory Initial Address: 0
EC020000 E4420440 8442FFF0 1C400801 1C402805 6C000008 A0000000 14450005

Allocating Data Section:
Memory Initial Address: 120
147AE148 3FD147AE 33333333 3FD33333 51EB851F 3FD51EB8 F5C28F5C 3FDF5C28

Allocating Bss Section:
Memory Initial Address: 9A0
  3014D    6C9E     AC2      19       0 7865742E 60AB0074       0

Releasing Text Section:
Memory Initial Address: 0
EC020000 E4420440 8442FFF0 1C400801 1C402805 6C000008 A0000000 14450005

Releasing Data Section:
Memory Initial Address: 120
66666666  3FE66666  33333333 3FD33333 9999999A 3FB99999       0       0

Releasing Bss Section:
Memory Initial Address: 9A0
  3014D    6C9E       0       0       0 7865742E 60AB0074       0

Releasing Abs Section:
```

Figura 3.4. - Exemplo de um arquivo de *dump* de memória.

3.4. Limitações do simulador

Não foram implementadas interrupções, gerenciador de traps, entrada e saída e tradução de endereços, não existindo, portando, TLBs para o simulador. Não é possível simular funções que efetuem qualquer tipo de chamada ao sistema operacional.

Algumas instruções também não foram implementadas: pst.d, intovr, ldio.x, stio.x,ldint.x, scyc.b, fzchks, pfzchks, fzchkl, pfzchkl, faddp, pfaddp, faddz, pfaddz, form

e pform. Na realidade, muitas destas instruções estão relacionadas às limitações descritas no parágrafo anterior.

3.5. O programa SHOWREPO

Um programa para construção de gráfico a partir das informações geradas pelo simulador também é usado para auxiliar o processo de otimização.

O programa SHOWREPO utiliza as informações armazenadas no arquivo temporário para obter conhecimento de como os recursos estão sendo utilizados ao longo do programa.

Todos os gráficos gerados pelo SHOWREPO apresentam informações de percentagem de um recurso utilizado x intervalo de amostragem do ciclos de relógio. Em cada intervalo é verificado o número de ciclos de relógio que um determinado recurso ficou ativo (ou ocupado). Este número sobre o número de ciclos de um intervalo determina a percentagam de utilização de determinado recurso num intervalo.

Podem ser analisadas as percentagens de até três recursos em um único gráfico. A determinação de quais recursos devem se apresentados é obtida através da linha de comando. É possível inclusive, fornecer o intervalo de tempo que o gráfico deverá analisar, bem como o número de ciclos de relógio exitentes em um intervalo de tempo.

As opções que podem ser fornecidas através da linha de comando são:

- -f - informa o nome do arquivo temporário a ser usado

- -p1, -p2, -p3 - informam quais são os recursos que serão apresentados. As opções são:

  - CORE_ACTIVITY - fornece informações acerca da utilização da unidade central;
  - SCALAR_FP_UNIT - fornece informações acerca da utilização da unidade de ponto-flutuante, seja com instruções escalares ou *pipelined*;
  - GRAPH_ACTIVITY - fornece informações acerca da utilização da unidade gráfica;
  - INSTR_CACHE_MISS - fornece informações sobre faltas na cache de instruções;
  - INSTR_CACHE_HIT - fornece informações sobre hits na cache de instruções;

- DATA_CACHE_MISS - fornece informações sobre faltas na cache de dados;
- DATA_CACHE_HIT - fornece infromações sobre hits na cache de instruções;
- CACHE_NOT_AVAILABLE - informa sobre instantes nos quais a cache de dados ficou ocupada;
- MEM_ACCESS - informa sobre a utilização do barramento externo;
- SINGLE_OP_MODE - informa quais os instantes que o processador operou em modo singular;
- DUAL_OP_MODE - informa quais os instantes que o processador operou em modo dual.

• -x1 - ciclo de relógio a partir do qual será mostrado o gráfico (caso não seja fornecido, supõe-se zero)

• -x2 - ciclo de relógio até onde será mostrado o gráfico (caso não seja fornecido supõe-se o número de ciclos necessário para simular o programa)

• -s - determina quantos ciclos de relógio existirão em cada intervalo de tempo (caso não seja fornecido supõe-se o valor 10)

Capítulo 4

A Implementação do Simulador

4.1. Introdução

No capítulo 3 foram apresentadas as características do similador SIM860, bem como do programa SHOWREPO, que acompanha o SIM860.

Neste capítulo temos o objetivo de proporcionar o entendimento de como os recursos do i860 foram controlados para reproduzir as características do processador, bem como apresentar a validação dos resultados obtidos pelo simulador.

O simulador foi desenvolvido com orientação para os recursos do processador i860. As instruções fluem pelo estágio do *pipeline* de instrução à medida que os recursos necessários para sua execução estejam disponíveis. Todas as dependências de dados são tratadas como dependências de recursos. Um registrador, por exemplo, é visualizado como um recurso que pode estar disponível (conteúdo liberado para ser usado) ou não.

Cada estágio do *pipeline* pode exercer interferência sobre outro estágio, interrompendo o fluxo de instruções no *pipeline*. Por exemplo, uma instrução que demore mais de um ciclo de relógio para ser executada, impede que uma nova instrução seja buscada ou decodificada. A transferência de informação entre os estágios é feita através de informações contidas em uma estrutura denominada "estrutura de acoplamento".

A apresentação da implementação do simulador está dividida em quatro seções:

1. Estrutura de dados - nesta fase são apresentadas todas as estruturas de dados envolvidas na implementação de cada recurso do i860.

2. Iniciação do simulador - nesta fase mostramos como é feita a carga do programa de entrada e como se encontram os recursos antes que seja iniciada uma simulação.

3. Execução - nesta fase apresenta-se o curso da simulação, cada estágio do *pipeline* e suas atribuições, a ordem de execução, a interface entre cada estágio, o controle de recursos, e o controle dos modos de execução do i860.

4. Término do programa - nesta fase são descritos os procedimentos necessários para encerrar uma simulação.

A linguagem de programação escolhida para implementar o simulador foi a linguagem C, que oferece facilidades para algumas estruturas aqui empregadas.

4.2. Descrição do *software*

4.2.1 Estruturas de Dados

1. Bancos de Memória

Os bancos de memória simulam a memória externa do i860. Não é reservada uma área fixa da memória do PC para armazenar o código e os dados do programa que será simulado. O espaço de memória necessário é determinado em tempo de execução.

O arquivo de entrada do simulador se encontra em formato COFF (*Common Object File Format*) [15]. Um arquivo neste formato contém os seguintes elementos:

- cabeçalho de arquivo;
- cabeçalho de informação opcional;
- tabela de cabeçalho de seções;
- dados referentes aos cabeçalhos das seções;
- informação de realocação;
- números de linhas;
- tabela de símbolos;
- tabela de cadeia de caracteres.

Dos elementos acima, somente alguns são suficientes para a obtenção de toda a informação necessária para a execução de um programa no simulador: cabeçalho de arquivo, cabeçalho de informação opcional, tabela de seções e os dados referentes à tabela de seções.

O cabeçalho de arquivo fornece informações a respeito do número de seções existentes e número de *bytes* contidos no cabeçalho de informação opcional.

Como o próprio nome diz, o cabeçalho de informação opcional nem sempre está presente num arquivo em formato COFF. Sua existência é verificada através de um campo no cabeçalho de arquivo que informa o número de *bytes* existentes no cabeçalho de informação adicional. Quando presente, informa o ponto de início de execução. Quando ausente, o ponto de início de execução corresponde ao início da única seção de código que existirá no arquivo.

A tabela de seções especifica o *layout* de dados dentro do arquivo objeto. São armazenadas nesta tabela informações sobre cada uma das seções: endereço físico, endereço virtual, número de bytes da seção, endereço onde os bytes da seção estão localizados no arquivo e o tipo do dado armazenado na seção. São quatro os tipos de dados encontrados num arquivo COFF gerado pelos montadores e *linkers* para o i860 [3]: código (.TEXT), dados iniciados (.DATA), dados não iniciados (.BSS) ou, código ou dados que estejam num endereço absoluto de memória (.ABS). Uma seção contém um único tipo de dado. Cada tipo pode ter seus dados distribuídos em mais de uma seção.

Para representar as seções do PC usou-se bancos de memória. Um banco de memória corresponde a uma área da memória do PC que armazena os dados das seções do arquivo COFF. Para cada tipo de dado há um banco de memória correspondente. Como podemos encontrar mais de uma seção com uma mesmo tipo de dado, um banco de memória é representado por uma lista encadeada, onde cada elemento da lista contém informações referentes a uma seção. A figura 4.1 ilustra os quatro bancos de memória.

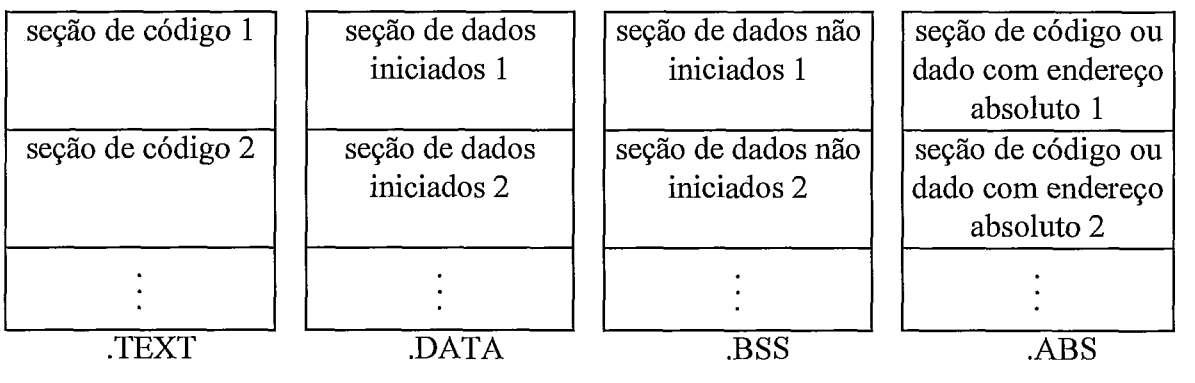


Figura 4.1. - Bancos de memória.

Cada elemento de qualquer uma destas listas possui o endereço físico e virtual inicial da seção do ponto de vista do i860; endereço final da seção também visto pelo i860; e um ponteiro que indica o início de uma área de memória do PC onde os dados da seção estão armazenados. No início de uma simulação os dados contidos no arquivo em formato COFF são copiados para cada banco de memória, e, neste momento, é determinado o valor do ponteiro existente em um elemento do banco de memória.

Quando o i860 deseja fazer um acesso seja a uma área de código ou dados, o simulador verifica se o endereço de acesso do i860 se encontra em algum elemento da lista correspondente ao tipo de acesso. Instruções são buscadas nos bancos de memória .TEXT e .ABS, enquanto o acesso a dados é feito nos bancos .DATA, .BSS E .ABS. Quando o endereço é encontrado, o acesso é feito diretamente na memória do PC, a partir do ponteiro no elemento da lista onde o endereço do i860 foi encontrado. Uma vez localizado o endereço em um elemento de um banco de memória, a procura é interrompida.

2. Memórias *Cache*

O simulador implementa somente as *caches* de instruções e dados.

Para efeito de simulação, a memória *cache* de instruções não possui uma cópia dos dados existentes na memória externa. No lugar da cópia, um ponteiro contém o endereço na memória do PC onde está localizado o primeiro dado da linha da *cache*. Isto só é possível pois a *cache* de instruções não permite que sejam feitas escritas e uma substituição de uma das linhas num conjunto da *cache* não acarreta uma escrita da linha que está sendo substituída na memória externa. Apenas o ponteiro que indica o início dos dados da linha na memória recebe o endereço inicial do novo conjunto de dados da linha. Na figura 4.2 vê-se a representação da memória *cache* de instruções vista pelo simulador.

A memória *cache* de dados, contudo, não pôde seguir o mesmo caminho de implementação da *cache* de instruções. Como a *cache* de dados permite acessos de escrita, uma linha que esteja na *cache* é atualizada na memória externa somente quando for substituída por outra. Para estar de acordo com este comportamento, quando uma linha é carregada para a *cache*, sempre é feita uma cópia dos dados da memória externa para a linha da *cache* de dados. A escrita de uma linha de volta na memória só se processa quando esta tiver sido modificada. Esta informação esta contida nos bits MESI da *cache* de dados do i860 [1]. A implementação da memória *cache* de dados está ilustrada na figura 4.3

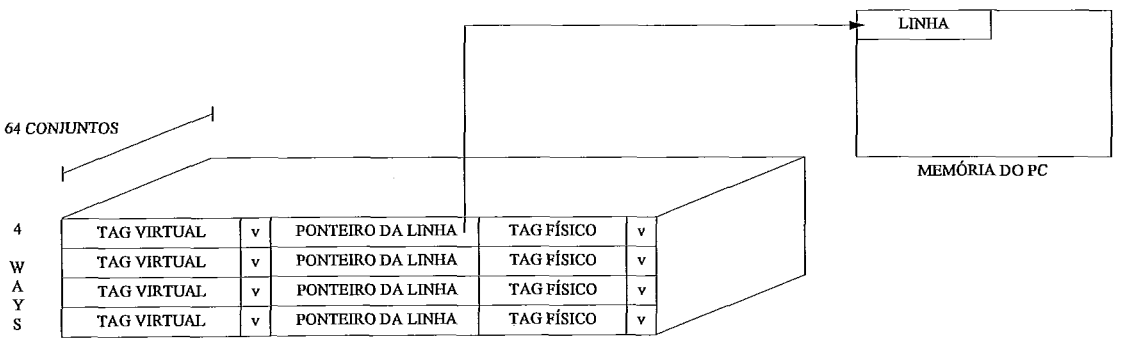


Figura 4.2. - Representação da memória *cache* de instruções.

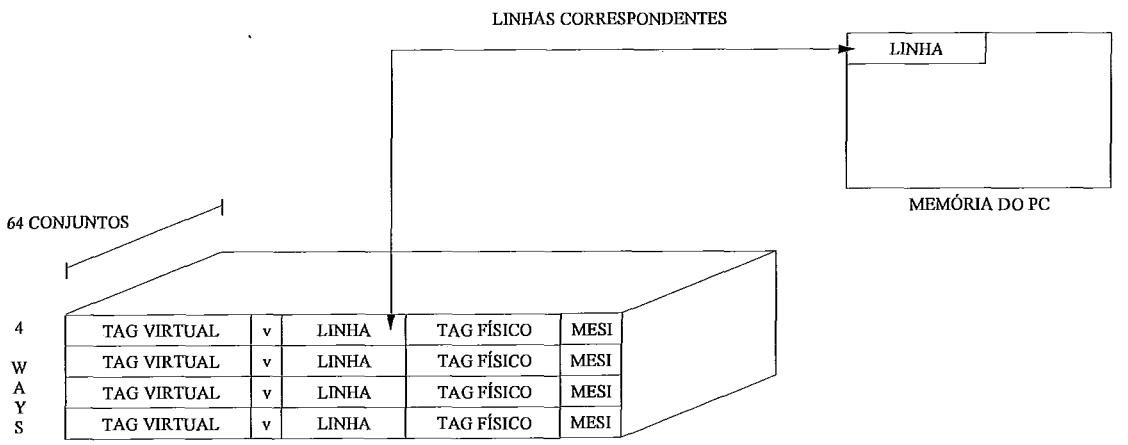
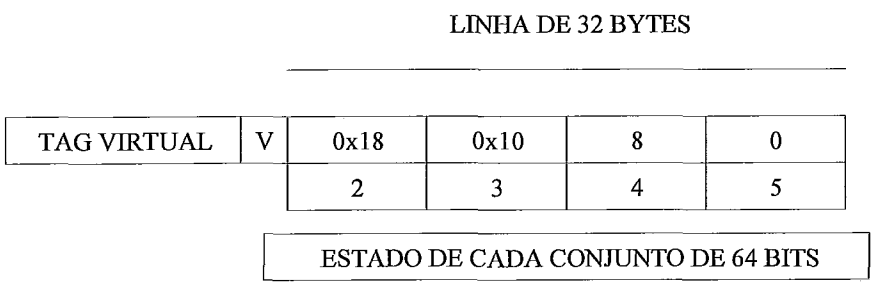


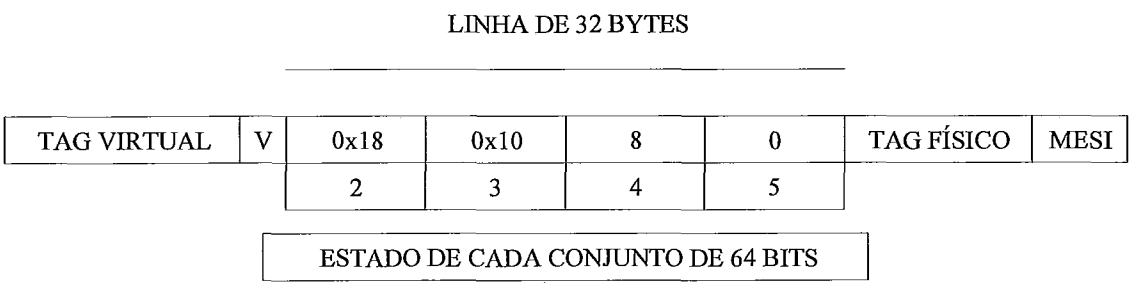
Figura 4.3. - Representação da memória *cache* de dados.

Quando ocorre uma falta em qualquer uma das *caches*, uma linha é buscada na memória. O preenchimento de uma linha da *cache* requer quatro acessos à memória externa, onde cada acesso é capaz de preencher um conjunto de 64 bits da linha. Portanto, cada conjunto de 64 bits da linha demorará um tempo diferente para ser trazido para a *cache*. No simulador, a busca de uma linha da *cache* é feita de uma só vez. Para reproduzir a temporização do preenchimento da linha da *cache*, cada elemento de 64 bits

de uma linha da *cache* possui um estado correspondente, que informa o número de períodos de relógio que aquele elemento levará para se tornar disponível na linha da *cache* (figura 4.4). Dependendo da posição que o elemento que está sendo acessado ocupar em uma linha da *cache*, a ordem com que os elementos da linha serão buscados na memória é diferente [2], e os estados de cada elemento são atualizados, repeitando esta ordem.



a. *Cache* de instrução



b. *Cache* de dados

Figura 4.4. - Linha da *cache* de dados e de instrução acompanhada dos estados de cada elemento de 64 bits da linha.

Na *cache* de dados, a ocorrência de uma falta pode acarretar um tipo de atraso não existente na *cache* de instruções. Não havendo linha inválida na *cache* de dados, uma das linhas existentes será descartada. Se esta tiver sido modificada, deverá, necessariamente, ser escrita na memória externa, antes que a outra linha seja trazida. Contudo, para evitar atraso na busca da nova linha, o i860 dispões de três *buffers* de escrita que são utilizados em qualquer acesso de escrita na memória externa. Quando uma linha é substituída, a

antiga linha é escrita em dois destes *buffers* e, depois que a nova linha é trazida, a escrita da antiga, na memória, é efetuada. Isto permite que os recursos envolvidos na reposição da linha sejam liberados no menor tempo possível para que outras instruções possam ser executadas. Contudo quando estes *buffers* de escrita não estão disponíveis no momento da reposição da linha, a busca da linha na memória externa só poderá ser iniciada quando pelo menos dois *buffers* estiverem desocupados. Este fato inevitavelmente provoca um atraso na execução das próximas instruções e, consequentemente, uma quebra no fluxo do *pipeline* de execução.

Caso uma linha de um conjunto da *cache* precise ser preenchida mas não haja um linha inválida no conjunto, um algoritmo pseudo-aleatório determina qual linha do conjunto será descartada, para substituição pela nova linha. Para reproduzir esta situação o simulador utiliza a função para gera números pseudo-aleatórios *rand*, da biblioteca padrão da linguagem C.

3. Acesso ao Barramento Externo

O controle de utilização do barramento externo é feito por uma "fila de acesso ao barramento externo". Cada vez que é necessário efetuar um acesso de leitura ou escrita à memória externa, a ordem de chegada deve ser obedecida. Se a fila estiver vazia, o acesso é iniciado imediatamente e, caso contrário, se alguma instrução ou busca de instrução depender da utilização do barramento externo, deverá esperar esta fila esvaziar para poder iniciar do seu acesso. Esta estrutura de fila garante que os acessos serão feitos na ordem em que chegarem.

Cada elemento da fila contém o número de períodos de relógio que o barramento fica ocupado com um acesso à memória. O primeiro elemento é sempre decrementado ao início de um novo período de relógio.

Um acesso pode ser enfilerado quando ocorre uma falta na *cache* de intruções ou quando houver falta na *cache* de dados para operações de *load* e *store*. O preenchimento de uma linha também requer o uso do barramento externo.

4. Acesso aos *Buffers* de escrita

A utilização dos *buffers* de escrita também é controlada por uma fila. Os três primeiros elementos desta fila correspondem aos três *buffers* existentes. A inexistência de um *buffer*

vazio é caracterizada pelo fato de existirem pelo menos três elementos na fila. Neste caso, a utilização dos *buffers* para instruções de *store* só poderá ser feita após a liberação do primeiro elemento da fila (um *buffer* de 128 é suficiente para uma operação de *store*), e, no caso de reposição de uma linha da *cache*, após a liberação de dois elementos da fila (cada *buffer* tem capacidade de armazenar informação de metade de uma linha da *cache*).

5. Banco de Registradores

Os registradores de inteiros foram representados por um vetor de elementos inteiros de 32 bits sem representação de sinal.

Como os registradores de ponto flutuante podem ser utilizados para armazenar números reais em precisão simples (utilizando apenas um registrador de 32 bits) ou números em precisão dupla (utilizando um par de registradores combinando 64 bit), a melhor estrutura para implementa-los é uma *union*. A *union* permite que uma mesma área de memória seja observada de diferentes modos. Ora podemos vê-la como um conjunto de 32 registradores de números em precisão simples, ora podemos observá-la com um conjunto de 16 registradores de números em precisão dupla.

Os registradores de controle foram representados como variáveis que armazenam valores inteiros de 32 bits sem representação de sinal. Isto facilita operações de *set* e *clear* de determinados bits de controle.

Os registradores de uso específico também podem armazenar mais de um tipo de dados, mas neste caso não houve a necessidade de representá-los como uma *union* pois estes registradores possuem um tamanho fixo de 64 bits. Para implementar os diferentes tipos de dados, dependendo da precisão adotada, utiliza-se *type casting*.

Cada registrador possui um registrador auxiliar associado. Estes registradores são usados para armazenar o resultado de uma execução temporariamente. Isto é necessário pois cada registrador só pode ter seu conteúdo atualizado no estágio de escrita. Isto garante a atualização no momento correto de todos os registradores envolvidos na execução de uma instrução.

6. *Pipelines* de adição e multiplicação em ponto-flutuante

Os *pipelines* de adição e multiplicação foram ambos implementados como um vetor de três elementos, como ilustra a figura 4.5.

Resultado	Resultado	Resultado
Precisão do resultado	Precisão do resultado	Precisão do resultado
Validade do resultado	Validade do resultado	Validade do resultado
1° Estágio	2° Estágio	3° Estágio

a. Pipeline de soma e de multiplicação (operandos em precisão simples).

Resultado	Resultado
Precisão do resultado	Precisão do resultado
Validade do resultado	Validade do resultado
1° Estágio	2° Estágio

b. Pipeline de multiplicação para operandos com precisão dupla.

Figura 4.5. - Representação dos *pipelines* de soma e multiplicação. O *pipeline* de multiplicação possui números de estágios diferentes quando os operandos são de precisão simples e precisão dupla.

Cada elemento do vetor armazena informação sobre o resultado de uma operação, sua precisão e sua validade. Para efeito de simulação, uma operação é sempre concluída no primeiro estágio do *pipeline*, mas seu resultado só se torna disponível quando este resultado atinge o último estágio.

Uma instrução de ponto-flutuante escalar passa por todos os estágio do *pipeline* invalidando os resultados exixtentes em cada estágio. Esta informação fica armazenada na "validade do resultado". Na realidade, esta informação tem mais significado para a geração do relatório, pois determina se o estágio do *pipeline* está sendo usado com uma operação *pipelined* ou não.

4.2.2. Iniciação do Simulador

A iniciação do simulador consiste em preparar o programa para iniciar uma simulação.

Em primeiro lugar, é feita a leitura do arquivo de entrada, em formato COFF. É alocado espaço de memória do PC para o armazenamento de todos os dados das seções nos bancos de memória, e todos os dados das seções são copiados para a memória do PC.

Em seguida é feita a iniciação de todos os registradores internos do processador, mantendo-os na configuração encontrada logo após a ocorrência de um *reset* no processador [1].

As *caches* de instruções e de dados são invalidadas e todos os recursos existentes no processador se encontram disponíveis para qualquer execução de instrução: unidade de *core* e de ponto flutuante livres (incluindo unidade de adição, multiplicação e gráfica), barramento externo, *buffers*, e todos os registradores.

4.2.3. Execução

A execução do programa é dividida em quatro estágios, cada um correspondendo a cada estágio do *pipeline* de instrução: busca da instrução, decodificação da instrução, execução das operações definidas pela instrução e escrita dos resultados. Para simplificar, a partir de agora vamos no referenciar a estes estágios como apenas, busca, decodificação, execução e escrita.

Cada estágio é responsável por transferir a informação para o próximo estágio. A figura 4.6 apresenta a estrutura do *pipeline* de instrução.

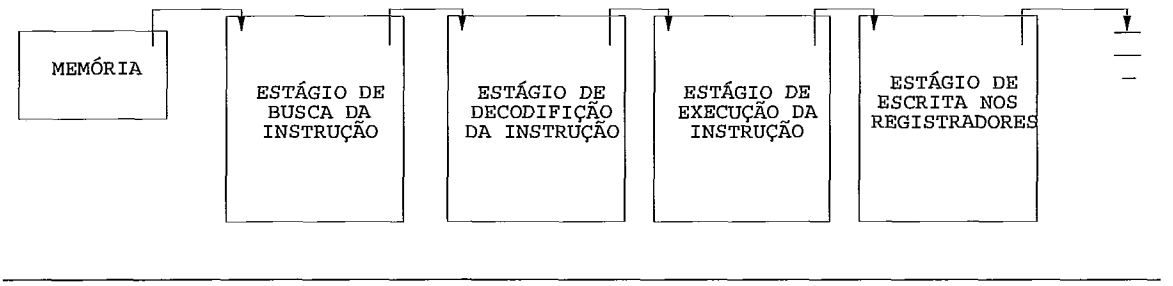


Figura 4.6. - Estrutura do *pipeline* de instrução.

1. Atribuições de cada estágio do *pipeline* de instrução

• Busca da Instrução

Neste estágio é feita a busca da instrução cujo endereço está definido pelo contador de programa localizado dentro da unidade central. Inicialmente é feito uma acesso à *cache*

de instruções. Se a instrução não se encontrar na *cache* de instruções o dado é então buscado diretamente na memória.

Buscar um dado diretamente na memória significa localizar na lista correspondente a área de código em que posição na memória do PC está a instrução. É feita a busca de uma linha da *cache* de instruções e, como a instrução não estará disponível para ser decodificada no próximo período de relógio (tempo de duração do estágio de busca), o estágio de busca alimenta o estágio de decodificação com uma instrução especial, denominada "instrução *dummy*". Esta não impede que qualquer instrução válida no restante do *pipeline* tenha seu fluxo interrompido, mas permite aos próximos estágios que prossigam com suas funções.

Também é responsabilidade deste estágio, quando for necessário buscar uma linha da *cache* decorrente de uma falta, determinar o estado de cada elemento de 64 bits de uma linha da *cache*. Isto porque quando uma linha é buscada, cada um destes elementos estará disponível em um instante diferente (seção 4.2.1., item 2).

- **Decodificação da Instrução**

O estágio de decodificação recebe do estágio de busca a instrução buscada no ciclo de relógio imediatamente anterior.

Antes de verificar se uma instrução proveniente do estágio de busca é válida, ou seja, diferente de um instrução *dummy*, o estágio de decodificação verifica se o estágio de execução poderá receber uma nova instrução no próximo período de relógio. A única condição que impede que o estágio de execução receba um nova instrução no próximo período de relógio decorre de sua impossibilidade de executar a última operação recebida, em razão da indisponibilidade dos recursos necessários para sua execução. Se isto ocorrer, o estágio de decodificação adia a decodificação da instrução oriunda do estágio de busca para o próximo período de relógio. Ao mesmo tempo, não é possível buscar qualquer nova instrução.

Tratando-se de uma instrução válida, este estágio se encarrega de determinar qual é a instrução atual, determinando a função que irá executá-la, e verificar quando os recursos do processador necessários para sua execução estarão disponíveis. Estes recursos são: unidade central livre, unidade de ponto flutuante livre, registradores fonte livres e acesso à *cache* de dados livre para instruções *load* e *store*. Por exemplo, o fato da unidade central não estar disponível no próximo período de relógio impede a execução de uma

instrução que utilize esta mesma unidade funcional, mas não impede que uma instrução que utilize a unidade de ponto flutuante deixe de ser executada.

A informação de quando os recursos necessários para a execução da instrução estarão válidos é passada do estágio de decodificação para estágio de execução, que só executa a operação quando perceber que os recursos estão liberados. É portando de responsabilidade do estágio de decodificação determinar toda a disponibilidade de recursos necessários para a execução de uma instrução.

- **Execução da operação**

O estágio de execução executa uma operação somente quando todos os recursos necessários para a execução da instrução estão disponíveis.

Toda execução de uma operação, para efeito de simulação, é feita no primeiro período de relógio dentre o número total de períodos de relógio necessários para sua execução. No caso de uma operação demorar mais de um período de relógio para ser executada, seus resultados só se encontram disponíveis no último período de relógio, e os recursos que utiliza são sinalizados de modo a ficarem indisponíveis até que sua "execução" seja concluída.

Como o i860 é capaz de executar duas operações simultaneamente, torna-se imperativo que o simulador SIM860 respeite esta característica. É o caso observado, por exemplo, quando uma operação de ponto-flutuante está sendo executada junto a uma operação de inteiros.

Cada vez que uma operação é concluída, indica ao estágio de escrita que os registradores destino, e os registradores de controle, quando necessário, devem ser atualizados. A execução armazena os resultados em registradores auxiliares e indica, através de uma variável de estado vinculada a estes registradores auxiliares, o momento no qual a informação neles contida deve ser passada aos registradores do processador. Esta variável de estado contém a informação do número de períodos de relógio que o resultado de uma operação leva para se tornar disponível.

Instruções de chamada de outras rotinas ou instruções de desvio podem alterar a seqüência de execução do programa. É neste estágio que é determinada uma possível alteração na seqüência de execução do programa.

- **Escrita dos resultados**

O estágio de escrita recebe informações da execução a respeito do momento em que uma escrita deve ser feita. Esta informação é obtida através de uma variável de estado, relacionada a cada registrador, como descrito no ítem 4. desta seção.

2. Ordem de Execução

A cada período de relógio todos os estágios do *pipeline* de instrução têm que cumprir com as suas atribuições. Contudo, como trata-se de um programa sequencial, tem-se que avaliar qual estágio entrará em ação primeiro. Para tal, vamos analisar alguns aspectos do processador.

O i860 é capaz de verificar se um resultado que deveria estar sendo escrito no período de relógio corrente seria necessário para a execução da instrução corrente. Caso necessário, para evitar a espera de uma escrita no registrador e uma posterior leitura, é possível alimentar o estágio de execução diretamente com o resultado da execução anterior. Se efetuarmos o estágio de escrita antes do estágio de execução, o registrador necessário para a execução já se encontraria atualizado, permitindo que a alimentação do dado pudesse ser feita de forma direta e sem acarretar qualquer problema.

Durante a execução da operação corrente, desvios e chamadas de subrotinas podem alterar o conteúdo do contador de programa. O novo conteúdo já teria que estar válido quando a busca da instrução no período de relógio corrente fosse feita. Se o estágio de busca já tivesse sido concluído no momento da execução e a execução alterasse o contador de programa, seria necessário fazer o estágio de busca voltar à condição inicial (*roll back*) e refazer a busca considerando o novo conteúdo do contador de programa .

O estágio de decodificação avalia os recursos disponíveis para a execução de uma operação no próximo período de relógio. Estes recursos podem ser alterados pela operação que está sendo executada no período de relógio corrente. Para avaliar os recursos corretamente, estas alterações já têm que ter sido consideradas pela decodificação.

Percebe-se, portanto, que é bastante interessante efetuar o estágio de execução depois da escrita, e a busca e decodificação depois da execução. Uma boa solução é executar os estágios de trás para frente, ou seja, começando pela escrita, execução, decodificação e busca. Isto não afetaria a passagem de informação de um estágio para outro.

3. Interface entre os Estágios.

A informação é transferida de um estágio para outro através de um "estrutura de acoplamento". Todo estágio possui uma estrutura própria, mesmo que as informações não lhe sejam todas relevantes. Nesta estrutura identificamos as seguintes informações:

- Instrução do estágio - instrução que está sendo avaliada por um determinado estágio durante um período de relógio.
- Número de períodos de relógio que a instrução demorará para ter sua execução concluída (só é relevante para o estágio de execução).
- Segunda instrução do estágio - como é possível duas instruções estarem sendo executadas simultaneamente, esta indica a instrução que está sendo executada concorrentemente.
-Número de períodos de relógio necessários para que seja concluída a execução da segunda instrução.
- Endereço da instrução do ponto de vista do i860.
- Endereço da função encarregada de efetuar a operação no estágio de execução.
- Próxima instrução a ser executada pelo estágio de execução.
- Número de períodos de relógio que o estágio de execução deve aguardar para que todos os recursos necessários para a execução da próxima operação estejam disponíveis.

Sempre que um período de relógio é iniciado as informações entre um estágio e outro são atualizadas. Estas atualizações refletem-se nas seguintes interfaces entre estágios:

- Busca → decodificação

A "instrução do estágio" relacionada à busca é passada para a decodificação. Esta instrução foi trazida pelo estágio de busca no período de relógio anterior é decodificada no período de relógio corrente.

A figura 4.7 ilustra a interface existente entre os estágios de busca e decodificação, apresentando as informações que são transferidas da busca para a decodificação.

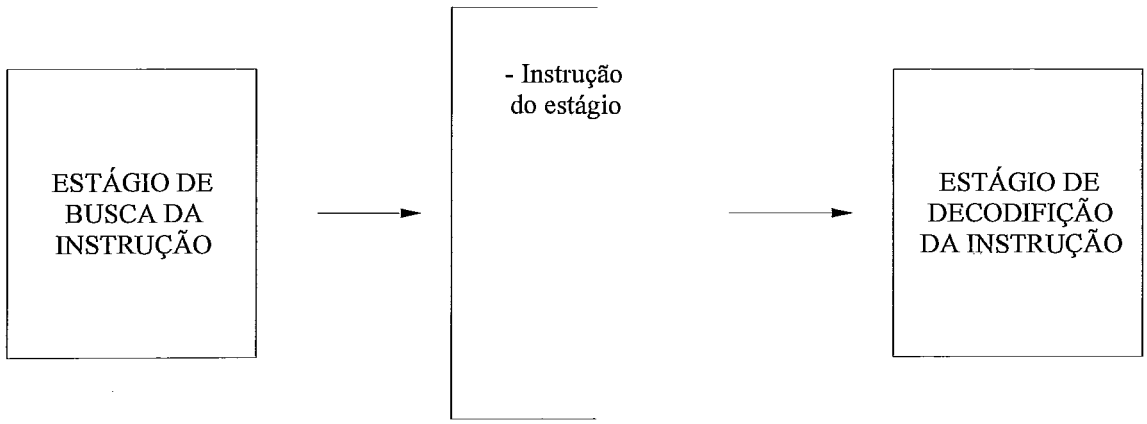


Figura 4.7. - Informações existentes na interface entre os estágios de busca e decodificação.

- Decodificação → execução

O estágio de decodificação identifica no período de relógio anterior qual é a operação que deve ser executada pelo estágio de execução no período de relógio corrente. Também durante a decodificação, são analisadas as disponibilidades de recursos para que a execução possa ser efetuada. Estas duas informações são passadas para o estágio de execução através da "informação da próxima operação a ser efetuada" e do "número de períodos de relógio que os recursos necessários para a execução estarão ocupados com outras atividades". A decodificação também informa à execução o endereço da instrução que está sendo passada e o endereço da função que irá efetuar a operação.

A figura 4.8 esquematiza a interface existente entre o estágio de decodificação e o estágio de execução, apresentando as informações que são transferidas da decodificação para a execução.

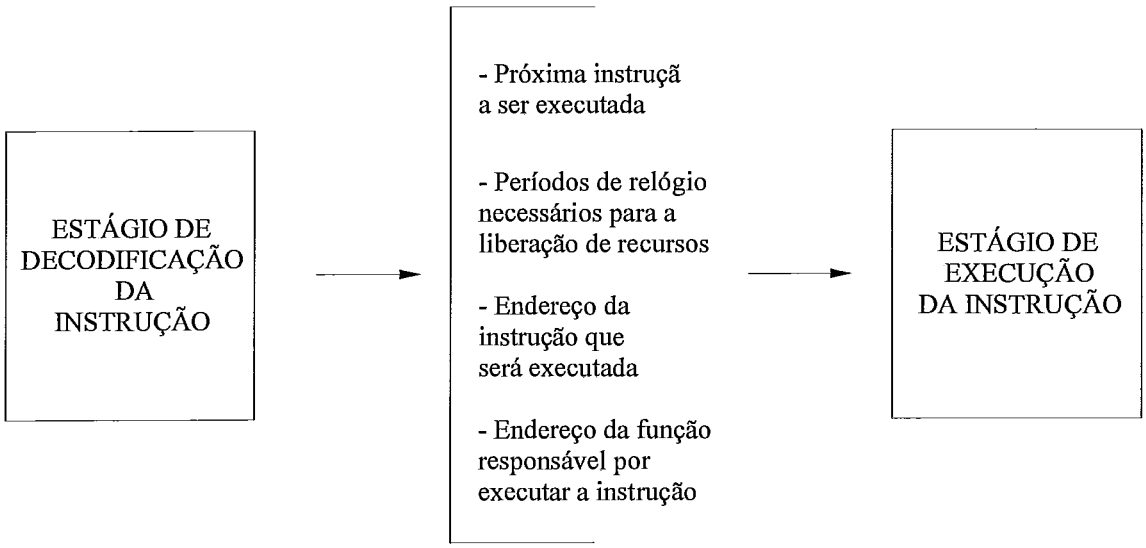


Figura 4.8. - Informações existentes na interface entre os estágios de decodificação e execução

- Execução → escrita

Ao término da execução de uma instrução, o estágio de escrita recebe a "instrucão de estágio" que concluiu sua execução. O mesmo acontece para a segunda "instrução de estágio" que concluiu sua execução (neste momento o estado dos registradores auxiliares informam que a escrita nos registradores destino e de controle já podem ser efetuadas). Além destas instruções o estágio de escrita recebe da "execução" os estados dos registradores auxiliares (para maiores detalhes consultar o item 4 desta seção).

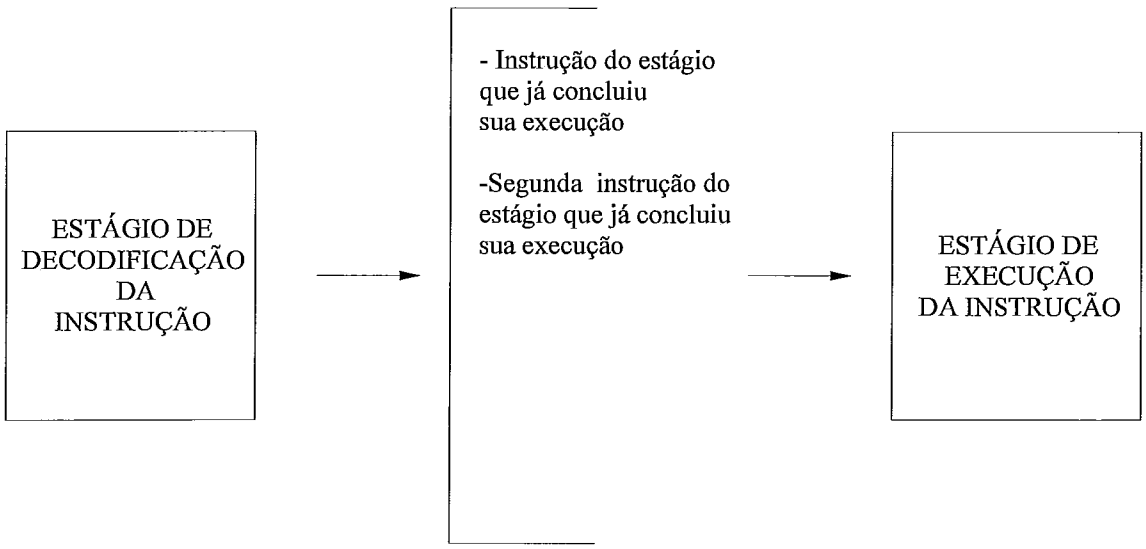


Figura 4.9. - Informações existentes na interface entre os estágios de execução e escrita.

Além destas informações, que seguem o fluxo do *pipeline* de instrução, o estágio de execução pode exercer influência direta sobre os estágios de busca e decodificação. Isto porque se uma instrução não pode ser executada porque os recursos que necessita não estão disponíveis, o fluxo do *pipeline* tem que ser interrompido, e não é possível continuar fazendo busca e decodificação pois o *pipeline* não poderia continuar caminhando.

Toda vez que uma operação pode ser executada, o estágio de execução inválida as informações provenientes da próxima instrução a ser executada e sinaliza o tempo que os recursos estarão bloqueados, sempre dentro da estrutura relacionada ao estágio de execução. Quando não é possível efetuar a operação, o estágio de execução não inválida estas informações. Desta forma, os estágios de busca e decodificação conseguem verificar se podem dar continuidade às suas funções. Isto estabelece uma relação entre os estágios de execução e busca/decodificação, no sentido inverso ao fluxo do *pipeline* de instrução.

A figura 4.10 ilustra as informações existentes na interface entre os estágios de execução e busca e execução e decodificação.

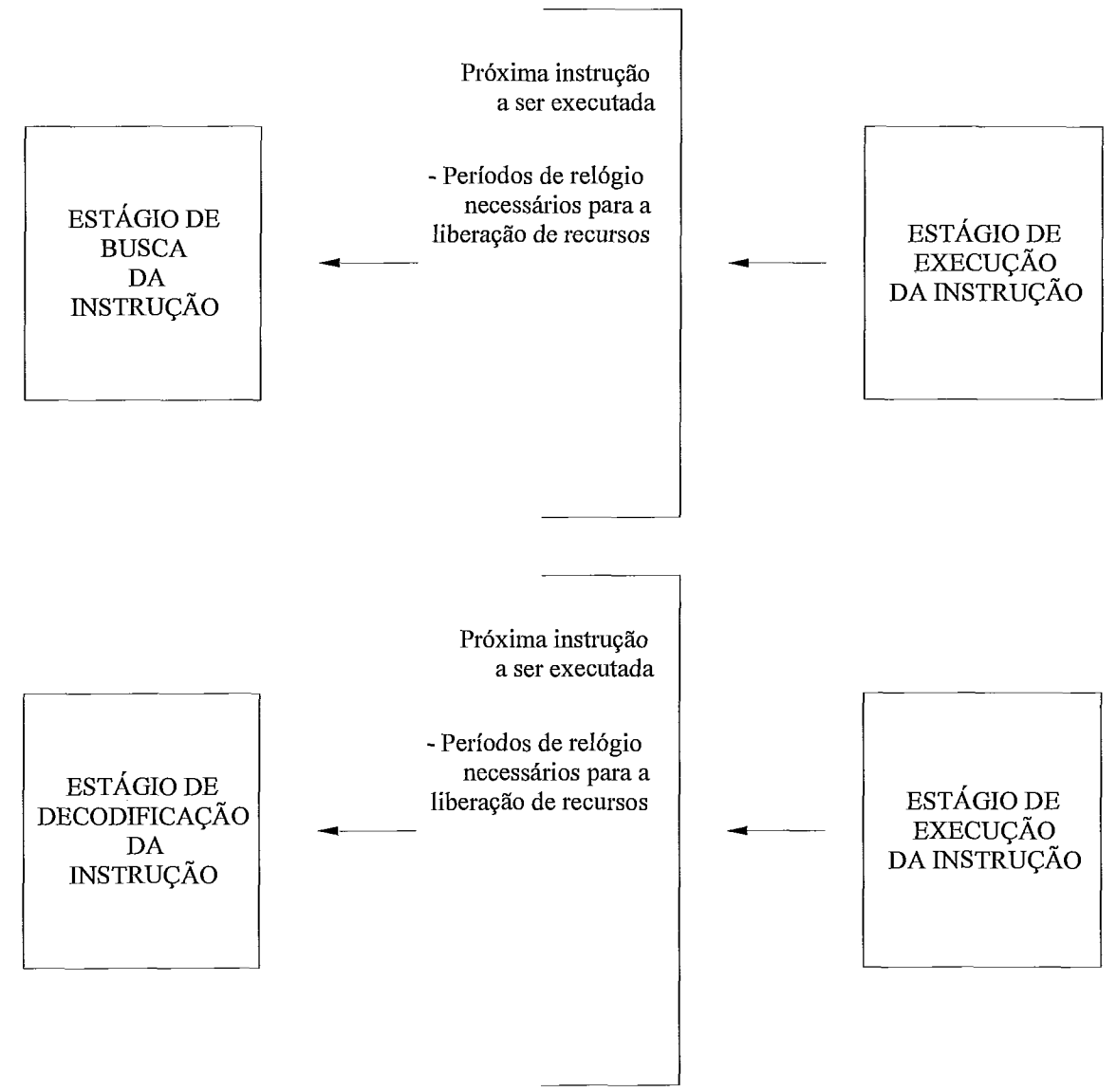


Figura 4.10. Informações existentes na interface entre os estágios de execução e busca e de execução e decodificação.

Os estágios de busca e execução ainda podem interferir-se mutuamente através de um recurso comum: o barramento externo. Faltas existentes nas *caches* de instrução e dados ocupam o barramento externo podendo acarretar atrasos tanto na busca de uma instrução quanto na leitura ou escrita de um dado na memória externa.

4. Controle de Recursos

Até o momento já foi mencionada a questão da disponibilidade de recursos para a execução de uma instrução, mas ainda não foram listados os recursos disponíveis no processador, que passaremos agora a descrever.

• Unidades Funcionais

O i860 dispõe de três unidades funcionais: a unidade central, a unidade de ponto flutuante e a unidade gráfica, representadas pelas variáveis bgCoreUnit, bgFPUnit e bgGrphUnit, respectivamente. Todas indicam o número de períodos de relógio que as unidades funcionais ficam ocupadas durante uma determinada operação. São atualizadas dentro do estágio de execução, e utilizadas pelo estágio de decodificação para verificar a disponibilidade de recursos. Como armazenam informação sobre o número de períodos de relógio durante os quais o recurso ficará ocupado, têm seus conteúdos decrementados de uma unidade a cada início do período de relógio.

• Registradores

Todos os registradores existentes no i860 possuem no simulador um registrador auxiliar e um registrador de estado associados a ele.

Quando a execução de uma operação altera o conteúdo de um registrador, na realidade, somente para o efeito de simulação, altera o conteúdo de um registrador auxiliar e seu registrador de estado correspondente. O registrador de estado informa o número de períodos de relógio que o registrador do i860 associado ao registrador auxiliar fica impossibilitado de ser usado.

Valor zero no registrador de estado indica ao estágio de escrita que o conteúdo do registrador auxiliar deve ser transferido para o conteúdo do registrador do i860. O estado então assume o valor -1 e passa a indicar que o registrador está livre para ser utilizado. A informação de disponibilidade de um registrador, portanto, é definida por seu estado. A figura 4.11 mostra o controle do recurso registrador.

```
reg j -> registrador j seja de inteiro ou ponto flutuante
regaux j -> registrador auxiliar associado a reg j
regest j -> registrador de estado associado a reg j

   caso regest j > 0
      reg j nao pode ser utilizado ;
      regest j é decrementado de uma unidade
   caso regest j == 0
      reg j recebe regaux j
      regest j é decrementado de uma unidade e atinge o valor -1, indicando
               que reg j pode ser utilizado
```

Figura 4.11. - Comportamento dos registradores sob a ótica de recursos gerenciados pelo simulador.

• Barramento Externo

O acesso ao barramento externo é representado por uma fila onde cada elemento corresponde a um acesso pendente no barramento, informando o número de períodos de relógio que um acesso requer para ser concluído. Sempre que uma operação ou busca de instrução requer o uso do barramento externo, o tempo total de ocupação do barramento é calculado para determinar o atraso na operação desejada.

Este recurso não pode ser avaliado no estágio de decodificação pois sua necessidade só é determinada quando da ocorrência de falta seja na *cache* de instruções seja na *cache* de dados (em tempo de execução).

• Buffers de escrita

Os *buffers* de escrita também estão representados por uma fila onde cada elemento indica o número de períodos de relógio que um *buffer* fica ocupado, aguardando a liberação do barramento para que uma escrita possa ser efetuada.

Este recurso também não pode ser avaliado no estágio de decodificação pois sua utilização só é determinada quando da ocorrência de falta na *cache* de dados, em tempo de execução.

5. Controle dos *pipelines* de ponto flutuante

A unidade de ponto flutuante possui dois *pipelines*: *pipeline* para execução de operações de adição e *pipeline* para execução de operações de multiplicação.

O controle dos *pipelines* é feito através de dois vetores, cada um com um número de elementos correspondente ao número de estágios. Nestes vetores são armazenados os resultados das operações em cada estágio do *pipeline*.

Quando uma nova operação é inserida em algum dos *pipelines*, o resultado do último estágio é armazenado no registrador auxiliar do destino e seu estado indica para o estágio de escrita que no próximo período de relógio o conteúdo do registrador auxiliar correspondente ao registrador destino deverá ser copiado no registrador do i860. Os *pipelines* não transferem informações de um estágio para outro a menos que uma nova instrução seja inserida no *pipeline*.

6. Controle da mudança de precisão

Quando os operandos de uma operação de multiplicação são de precisão dupla, o *pipeline* de multiplicação passa a ter apenas dois estágios de dois períodos de relógio cada, ao invés de três estágios de um período cada. Desta forma, a transição de operações de multiplicação com dados de entrada de diferentes precisões requer atenção.

A princípio, pensamos em implementar um *pipeline* de multiplicação com um número variado de estágios. Contudo, a implementação torna-se bem mais simples se considerarmos que, quando os dados são de precisão dupla, a operação que entre no primeiro estágio do *pipeline* permanece neste estágio por dois períodos de relógio, seguindo depois para os segundo e terceiro estágios de uma só vez, mas gerando um resultado somente a cada dois períodos de relógio. Esta solução satisfaz plenamente as condições de funcionamento do *pipeline* e garante uma implementação bem mais simples.

Quando estamos entrando numa operação de precisão simples partindo de uma de precisão dupla, o processo também é simples. A operação no primeiro estágio passa para o segundo estágio e o último estágio do *pipeline* passa a ter como resultado o valor zero.

No capítulo 2 foi apresentado um trecho de programa que ilustra como é controlada a mudança de precisão entre os *pipelines* (seção 2.3)

7. Controle dos modos de operação

O i860 pode operar em dois modos: modo simples e modo dual. As transições entre estes modos foi implementada pelo simulador através de uma máquina de estados, como mostra a figura 4.12.

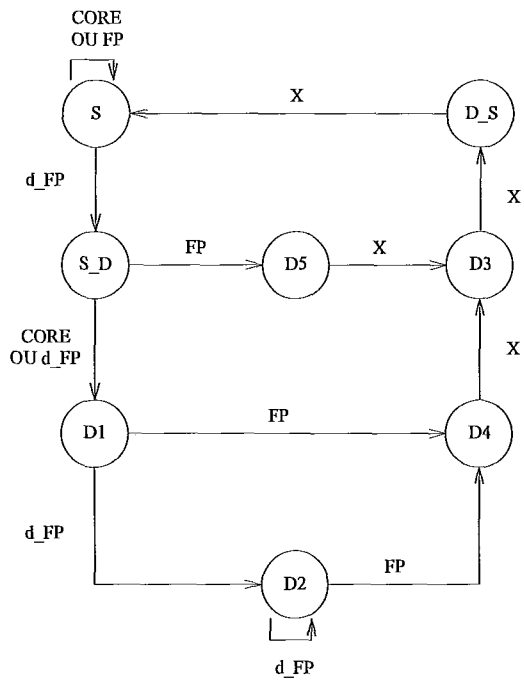


Figura 4.12. - Máquina de estados que representa as condições de entrada e saída de modo dual.

Na figura 4.12 a variável CORE indica a existência de uma instrução que seja executada pela unidade central, enquanto a variável FP indica uma instrução executada pela unidade de ponto-flutuante ou gráfica com modo dual inativo e a variável d_FP, com modo dual ativo.

Em modo dual, os quatro estágios do *pipeline* de instrução devem operar sobre um par de instruções. Contudo, durante o período de transição, nem todos os estágios operam sobre um par de instruções. Para cada estado, os estágios do *pipeline* de instrução operam da seguinte forma:

S - todos os estágios operam apenas sobre uma instrução;

S_D - O estágio de busca passa a buscar duas instruções;

D1 - É buscado e decodificado um par de instruções. Os demais estágios continuam operando sobre uma única instrução.

D2 - Todos os estágios operam sobre um par de instruções;

D3 - Os estágios de busca e decodificação operam sobre uma única instrução enquando os demais ainda operam sobre um par de instruções.

D4 - A máquina de estados se prepara para sair de modo dual e o estágio de busca passa a buscar apenas uma instrução;

D5 - A permanência em modo dual só dura um par de instruções. É feita somente a decodificação do par de instruções que foi buscado do estado anterior.

D_S - É feita a escrita do último par de instruções. Os demais estágios já operam sobre apenas uma instrução.

4.2.4. Término do programa

O encerramento da simulação é detectado quando o estágio de busca encontra um instrução trap r0, r0, r0. Neste instante são iniciados os procedimentos para a saída do programa de simulação.

Dependendo das opções de simulação selecionadas alguns arquivos são criados neste momento. Caso tenha sido especificada a geração de algum relatório, seja simplificado ou completo, durante a simulação as informações necessárias para os relatórios foram armazenadas em um arquivo temporário chamado RELATORI.LOG. Nesse momento de término do programa as informações do arquivo temporário são lidas e escritas (após a devida formatação, apresentada na seção 3.3.3) no arquivo do relatório de saída.

Caso tenha sido escolhida a opção de geração de *dump* de memória, neste momento é fechado o arquivo com o conteúdo de memória existente na área de memória do PC reservada para operar com memória do i860.

Feito isto, toda a área de memória do PC destinada ao i860 e alocada em tempo de execução do simulador é liberada.

Ao encerrar a simulação é escrito o código de término de programa.

Independente da simulação ser bem sucedida, o término de programa segue sempre estes passos, garantindo que a condição na qual o computador se encontrava, antes da simulação ter início, seja restaurada.

4.2.5. Validação do Simulador

Para validar o simulador dois parâmetros foram avaliados: os resultados dos programas simulados e seus tempos totais de execução.

Os programas de testes foram todos escritos originalmente em C. Seus resultados após a simulação foram comparados com os obtidos pelos respectivos programas C originais executados em ambiente PC. Para realizar esta comparação de resultados armazenou-se em arquivo o conteúdo da área de memória aonde os resultados obtidos na execução em ambiente PC estavam localizados. Em seguida usou-se a opção -d para a obtenção do *dump* da memória do I860 antes e depois da simulação. De posse destes dois conteúdos de memória pôde-se validar os resultados obtidos pela simulação.

A temporização, ponto principal do simulador, pôde ser validada somente através da temporização total da execução de um programa para o I860. Isto porque a única forma de validar a temporização foi através do simulador da Intel, que fornece apenas o número total de períodos de relógio necessários para a execução de um programa, e não a análise temporal durante a execução de um programa. Como o simulador da Intel fornece uma temporização aproximnada [4], era de se esperar uma pequena diferença entre as temporizações resultantes do simulador da Intel e do SIM860. De fato esta diferença existiu, mas ficou limitada a menos de 5% nos programas utilizados para testes. Portanto, a temporização do SIM860 mostrou-se coerente.

Capítulo 5

Experimentos e Resultados

5.1 Introdução

O presente capítulo apresenta um conjunto de experimentos feitos para ilustrar como o simulador, associado ao programa SHOWREPO, pode auxiliar na obtenção de um código otimizado.

A otimização do código é alcançada seguindo uma metodologia. Contudo, em alguns casos, pode não ser possível obter qualquer melhoria no código.

Para tentar otimizar um código, um conjunto de passos devem ser seguidos:

1. Um programa escrito em linguagem de alto nível é compilado para gerar um programa em linguagem *assembly*. A compilação é feita para os diversos níveis de otimização fornecidos pelo compilador, e deve-se trabalhar com o código que seja executado de forma mais eficiente. Para determinar qual o código mais eficiente, cada programa é deve ser executado pelo SIM860 com opção de geração de relatório simplificado. Aquele que consumir menor número de ciclos de relógio para ser executado é o mais eficiente.

2. O programa "standard.s", ilustrado na figura 5.1, deve ser introduzido no início de cada programa em *assembly* de modo a incluir uma área no programa destinada à pilha e permitir que o mesmo sempre seja encerrado por uma instrução do tipo *trap* r0, r0, r0, que indica ao simulador término de execução.

3. Inicialmente, não é possível identificar quais são os trechos de programa passíveis de otimização. Desta forma, torna-se necessário analisar a utilização dos recursos durante toda a execução de uma forma bastante abrangente, sem focalizar qualquer ponto específico. As instruções *trap* r5, r5, r0 (ativadora de início de

relatório) e *trap* r6,r6,r0 (desativadora de geração de relatório) delimitam um trecho do programa para o qual o relatório completo será gerado. Assim, no caso de se analisar todo o programa, estas instruções devem estar localizadas no início e fim do programa. No caso de análises mais focalizadas, estas instruções delimitarão apenas o trecho do programa sendo analisado. A geração de relatório simplificado independe da existência de um trecho delimitado no programa.

4. O programa em *assembly* é montado e ligado.

```
    .file  ""
    .data
    .align  16
stackarea   :  .long [400]10      // data area to be used as a stack
stackend    :  .long 0
framearea   :  .long [400]20      // data area to be used as a frame
register_dump:  .long [20]10

    .text
    .align  8

_L00TEXT:
    .text; .align  4

_start::
    // stack initialization
    mov    stackend, sp

    // Call the user program
    call   _main
    nop

    // Signal that the simulator MUST halt
    trap   0,r0,r0

_startup::
```

Figura 5.1. - Trecho de programa presente em todos os programas em *assembly* que serão executados pelo simulador.

5. O programa executável é simulado pelo SIM860. Se a entrada especifica um relatório completo, é criado o arquivo temporário RELATORI.LOG, necessário para a geração dos gráficos pelo SHOWREPO.

6. É gerado um gráfico que apresenta a percentagem de utilização da unidade central e de ponto-flutuante durante o trecho delimitado.

7. São avaliadas as percentagens de utilização da unidades central e de ponto-flutuante de modo a identificar pontos em que suas percentagens de utilização são mínimas.

8. São gerados novos gráficos com um intervalo de amostragem menor em trechos que permitam que seja feita melhor verificação da percentagem de utilização das unidades central e de ponto-flutuante. Normalmente um intervalo de amostragem de cinco períodos de relógio fornece uma boa precisão na avaliação.

9. Identificados os pontos onde realmente os recursos estão sendo pouco utilizados, é verificado no relatório completo os motivos que provocam tal comportamento. Se houver condições de rearrumar as instruções de modo a eliminar o conflito porventura existente, as modificações devem ser feitas diretamente no código em *assembly*.

10. O processo se repete a partir do passo 3 caso haja um outro trecho a ser analisado.

A seção 3.5 apresentou os parâmetros que podem ser avaliados pelos gráficos gerados pelo programa SHOWREPO. Dentre os diversos gráficos, o ponto de partida para a otimização de código é aquele que fornece informações sobre a percentagem de utilização das unidades central e de ponto-flutuante.

Em alguns casos, um gráfico que informe como está a utilização do barramento externo, junto com a utilização das unidades central e de ponto-flutuante, pode auxiliar bastante . São casos nos quais a ocorrência de faltas na *cache* de dados pode congestionar o barramento e propiciar um queda no desempenho.

A questão do barramento externo é explorada visto que o recurso "barramento externo" não parece ser analisado pelos compiladores quando é gerado um código para o i860. Quando possível, é interessante manter duas instruções de *load* afastadas, não para evitar a disputa pela unidade central, mas sim para evitar a disputa pelo barramento externo, e pelo acesso à *cache* de dados.

Os programas utilizados nos experimentos procuram retratar situações que podem estar presentes em outros casos, embora não haja um regra para códigos gerados em cada

caso. O objetivo é apresentar a metodologia empregada, e que pode, além do mais, ser utilizada em outros programas.

Normalmente a avaliação de código é feita em um bloco básico, mas isto não é obrigatório. A avaliação dos recursos é facilitada quando se restringe a uma parte do programa.

Antes de entrarmos no detalhe de cada experimento, é importante ressaltar que nem sempre os códigos gerados pelo compilador são os mais eficientes. Por exemplo, normalmente o código gerado utilizando-se o nível máximo de otimização procura operar em modo dual. Contudo, para que isto seja possível são introduzidos tantos nops (*no operation instruction*) e fnops (*floating-point no operation instruction*) que o processador passa um boa parte do tempo sem qualquer rendimento. Nestes casos, é interessante construir um programa que opere em modo simples mas mantenha todas as instruções *pipelined* do programa com modo dual. Este programa, em alguns casos, pode se mostrar mais eficiente (situação encontrada no primeiro experimento). Portanto, sempre que possível deve ser gerada uma versão do código sem modo dual para ter-se a garantia de que esta não é mais eficiente. Não foi observado qualquer caso no qual o compilador gere um código com instruções *pipelined* sem operar em modo dual.

5.2 Experimentos

5.2.1. Filtro FIR (*Finite Impulse Response*)

Este experimento implementa em linguagem C um filtro FIR. Duas características da implementação do Filtro FIR motivaram a utilização do simulador aqui desenvolvido para análise deste programa:

1- sendo normalmente empregado para processamento de sinais em tempo real, sua otimização torna-se de grande importância.

2- permite a exploração de instruções de operações duais posto que a fórmula utilizada para cálculo de cada elemento do sinal de saída envolve uma seqüência de multiplicações e somas.

$$y(n) = \sum_{i=1}^{M-1} ai \,.\, x(n-i)$$

onde M é a ordem do filtro.

O programa em C, construído para implementar o filtro FIR, é apresentado no apêndice A. Ele foi compilado em todos os níveis de otimização e, em seguida, gerado um relatório simplificado seguindo todos os passos preparatórios de um programa *assembly* para execução no simulador (programa "standard. s").

Neste caso, o nível de otimização 4 do compilador mostrou-se mais eficiente. No entanto, pode-se observar que a eliminação de todas as instruções de modo dual (nops e fnops introduzidos pelo compilador para funcionamento em modo dual) resultou em uma versão ainda mais eficiente. Desta forma, todo o nosso esforço para otimizar o código foi empreendido descartando-se o modo dual, porém, empregando-se instruções de ponto-flutuante *pipelined.* O código usado para otimização pode ser visto na figura 5.2.

```
      .file  "firnewop.c"
      .data
      .align  16
stackarea   :  .long [400]10       // data area to be used as a stack
stackend    :  .long 0
framearea   :  .long [400]20       // data area to be used as a frame
register_dump:  .long [20]10

      .text
      .align  8

_L00TEXT:
      .text; .align  4

//Stack allocation: Autos: Regsave:

_start::
      // stack initialization
      mov    stackend, sp

      // Call the user program
      call   _main
      nop

      // Signal that the simulator MUST halt
      trap   0,r0,r0

// PGC Rel 2.1 -opt 4
      .text
      .globl _main
      .align  8
_main:
.a1 = 0
.f1 = 32
```

Figura 5.2. Código utilizado para otimização do programa que implementa um filtro FIR.

```
    orh h%.STACK+.f1-16, r0, r28
     or l%.STACK+.f1-16, r28, r28
     st.l r4, -8(r28)
     st.l r5, -4(r28)
     orh h%_axgFilterParms, r0, r31
     or l%_axgFilterParms, r31, r5
     adds -1, r0, r16
     orh h%_axgXSignal+64, r0, r31
     or l%_axgXSignal+64, r31, r21
     orh h%_axgYSignal, r0, r31
     or l%_axgYSignal, r31, r4
     adds 71, r0, r20
.B125: //.M0001
     fiadd.dd f0, f0, f8
.DB.B125125:
     trap r5, r5, r0  // turn on the report generation
     mov r21, r18              --
     fiadd.dd f0, f0, f8       |
     mov r21, r18              |
     mov r5, r19               |
     mov r5, r19               |
     adds 7, r0, r17           |
     bla r16, r17,.B124        |
     pfmul.dd f0, f0, f0       |
.B124: //.M0000               |
.align 8                       |        --
     pfadd.dd  f0,  f0,  f0    |        |
.PL1001:                       |        |
     fld.d r0(r18), f16        |        |
.DB.B124124:                   |        |
     fld.d r0(r19), f18        |   I    |   I.1
     addu 8, r18, r18          |        |
     addu 8, r19, r19          |        |
     pfmul.dd f18, f16, f0     |        |
     bla r16, r17, .PL1002     |        |
     nop                       --       --
.PL1002:                                          --
     pfmul.dd f0, f0, f0                          |
     fld.d r0(r18), f16                           |
     fld.d r0(r19), f18                           |
     mi2p1.dd f8, f0, f30                         |
     addu 8, r18, r18                             |    II
     pfadd.dd f0, f0, f0                          |
     addu 8, r19, r19                             |
     mm12msm.dd f18, f16, f0                      |
     trap r6, r6, r0  // turn off the report generation
     bla r16, r17, .PL1002                        |
     pfadd.dd f0, f0, f8                          --
```

Figura 5.2. Código utilizado para otimização do programa que implementa um filtro FIR (continuação).

```
.PL1003:                        --
      pfmul.dd f0, f0, f0       |
      mi2p1.dd f8, f0, f30      |
      pfadd.dd f0, f0, f0       |     III
      pfadd.dd f0, f0, f0       |
      bla r16, r17, .PL1003     |
      pfadd.dd f0, f0, f8       |
// lineno: 66                   --
      fst.d f8, r0(r4)
      addu 8, r21, r21
      addu 8, r4, r4
      mov r20, r29
      adds -1, r20, r20
      xor 0x0000, r29, r0
      bnc.t .DB.B125125
      fiadd.dd f0, f0, f8
// lineno: 0
// lineno: 72
      ld.l -8(r28), r4
      ld.l -4(r28), r5
      bri r1
      nop
```

Figura 5.2. Código utilizado para otimização do programa que implementa um filtro FIR (continuação).

Uma vez obtido o código em *assembly*, um relatório completo do programa foi gerado, fornecendo informações sobre os trechos I e II indicados na figura 5.2. O trecho III, também indicado na figura 5.2, não é passível de otimização porque utiliza somente instruções de ponto-flutuante.

O gráfico apresentado na figura 5.3 mostra que as unidades central e de ponto-flutuante, durante a execução dos trechos I e II, não estão operando em paralelo, pois, quando uma cresce em porcentagem de utilização, a outra decai , e vice-versa.

Na figura 5.4 vê-se o relatório da segunda passagem da execução do programa pelo trecho I (a segunda passagem elimina a interferência devida a faltas na *cache* de instrução). Verificamos que poderia ser feito um rearranjo das instruções entre os períodos de relógio 194 e 203. A primeira operação de fld poderia ser executada antes de pfadd, pois não há qualquer dependência de recursos comuns às duas (deve-se lembrar que nesta análise os registradores também são considerados recursos) e não há qualquer instrução no programa que altere a seqüência de execução para PL1001. Desta forma, é possível transpor o

primeiro fld para antes do pfadd. Outra observação é o bla ser seguido de uma instrução nop quando a instrução addu 8, r19, r19 poderia ser colocada no lugar do nop.

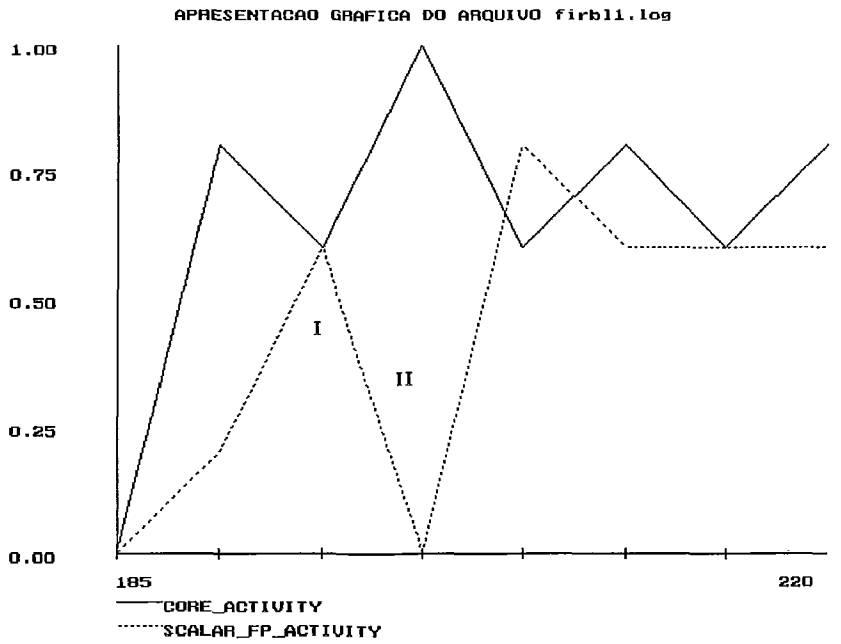


Figura 5.3. - Percentagem de utilização das unidades central e de ponto-flutuante durante execução de um programa.

A figura 5.4 mostra ainda uma passagem pelo trecho II. Esta seguramente não é a primeira passagem, pela mesma razão apresentada quando nos referimos ao trecho I. Observamos um conflito de unidades funcionais quando existem dois flds sucessivos entre duas instruções de ponto-flutuante (trecho compreendido entre os períodos de relógio 206 e 212 na figura 5.4). Se tivéssemos instruções de ponto-flutuante e de inteiros intercaladas, as unidades central e de ponto-flutuante poderiam trabalhar em paralelo.

A figura 5.5 mostra as alterações feitas no programa nos trechos I e II. A figura 5.6 apresenta o relatório composto dos trechos I e II após as modificações introduzidas no programa. Vê-se que o tempo destinado ao trecho I (figura 5.2) foi reduzido de 203 -194

(= 10) para 189 - 183 (= 7) ciclos de relógio, economizando-se, assim, 3 ciclos somente neste trecho. No trecho II, as quatro primeiras instruções que antes duravam 212 - 204 (= 9) ciclos para sua execução, duram agora 197 - 190 (= 8) ciclos, economizando-se, dessa forma, 1 ciclo.

RELATORIO DE SAIDA DO SIMULADOR pagina 6

INSTRUCTION PIPELINE				CLK	SRC1	SRC2	DEST	CORE	FP	FPADD			FPMPLY			GRPH	ICH	ICM	DCH	DCM	CNA	MA	OPM
FETCH	DECODE	EXEC.	WRITE	-	-	-	-	-		S1	S2	S3	S1	S2	S3								
fld.y	---	---	---	194	df0	df0	df0	-	X	X	X	X	X	X	X	-	X	-	-	-	-	-	S
				194	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
addu	fld.y	fld.y	pfadd.p	195	r0	r18	f16	X	-	X	X	X	X	X	X	-	X	-	X	-	X	-	S
				195	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
---	---	fld.y	---	196	r0	r18	f16	X	-	X	X	X	X	X	X	-	-	-	X	-	X	-	S
				196	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
addu	addu	fld.y	fld.y	197	r0	r19	f18	X	-	X	X	X	X	X	X	-	X	-	X	-	X	-	S
				197	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
---	---	fld.y	---	198	r0	r19	f18	X	-	X	X	X	X	X	X	-	-	-	X	-	X	-	S
				198	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
pfmul.p	addu	addu	fld.y	199	--	r18	r18	X	-	X	X	X	X	X	X	-	X	-	-	-	-	-	S
				199	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
bla	pfmul.p	addu	addu	200	--	r19	r19	X	-	X	X	X	X	X	X	-	X	-	-	-	-	-	S
				200	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
shl	bla	pfmul.p	addu	201	df18	df16	df0	-	X	X	X	X	X	X	X	-	X	-	-	-	-	-	S
				201	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
pfmul.p	shl	bla	---	202	r16	r17	--	X	X	X	X	X	X	X	X	-	X	-	-	-	-	-	S
		pfmul.p		202	df18	df16	df0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fld.y	pfmul.p	shl	bla	203	r0	r0	r0	X	-	X	X	X	X	X	X	-	X	-	-	-	-	-	S
			pfmul.p	203	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fld.y	fld.y	pfmul.p	shl	204	df0	df0	df0	-	X	X	X	X	X	X	X	-	X	-	-	-	-	-	S
				204	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
Mi2pl	fld.y	fld.y	---	205	r0	r18	f16	X	X	X	X	X	X	X	X	-	X	-	X	-	X	-	S
		pfmul.p		205	df0	df0	df0	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
---	---	fld.y	---	206	r0	r18	f16	X	-	X	X	X	X	X	X	-	-	-	X	-	X	-	S
		pfmul.p		206	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
addu	Mi2pl	fld.y	fld.y	207	r0	r19	f18	X	-	X	X	X	X	X	X	-	X	-	X	-	X	-	S
				207	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S

Figura 5.4. - Trecho do relatório completo que inclui as passagens pelos trechos I e II do programa.

RELATORIO DE SAIDA DO SIMULADOR pagina 7

INSTRUCTION PIPELINE				CLK	SRC1	SRC2	DEST	CORE	FP	FPADD			FPMPLY			GRPH	ICH	ICM	DCH	DCM	CWA	MA	OPM
FETCH	DECODE	EXEC.	WRITE	-	-	-	-	-		S1	S2	S3	S1	S2	S3								
fld.y	fld.y	pfmul.p	shl	204	df0	df0	df0	-	X	X	X	X	X	X	X	-	X	-	-	-	-	-	S
---	---	---	---	204	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
Mi2pl	fld.y	fld.y	---	205	r0	r18	f16	X	X	X	X	X	X	X	X	-	X	-	X	-	X	-	S
---	---	pfmul.p	---	205	df0	df0	df0	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
---	---	fld.y	---	206	r0	r18	f16	X	-	X	X	X	X	X	X	-	-	-	X	-	X	-	S
---	---	---	pfmul.p	206	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
addu	Mi2pl	fld.y	fld.y	207	r0	r19	f18	X	-	X	X	X	X	X	X	-	X	-	X	-	X	-	S
- ---	---	---	---	207	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
pfadd.p	addu	Mi2pl	---	208	df8	df0	df30	X	X	X	X	X	X	X	X	-	X	-	X	-	X	-	S
---	---	fld.y	---	208	r0	r19	f18	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
addu	pfadd.p	addu	---	209	--	r18	r18	X	X	X	X	X	X	X	X	-	X	-	-	-	-	-	S
---	---	Mi2pl	fld.y	209	df8	df0	df30	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
Mm12asm	addu	pfadd.p	addu	210	df0	df0	df0	-	X	X	X	X	X	X	X	-	X	-	-	-	-	-	S
---	---	---	Mi2pl	210	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
pfsm.p	Mm12asm	addu	pfadd.p	211	--	r19	r19	X	-	X	X	X	X	X	X	-	X	-	-	-	-	-	S
---	---	---	---	211	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
bla	pfsm.p	Mm12asm	addu	212	df18	df16	df0	-	X	X	X	X	X	X	X	-	X	-	-	-	-	-	S
---	---	---	---	212	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S

Figura 5.4. - Trecho do relatório completo que inclui as passagens pelos trechos I e II do programa (continuação).

Vistos isoladamente, estes valores podem parecer desprezíveis, mas levando-se em conta o tempo total o ganho pode ser considerável. O *speedup* neste caso é:

$$speedup = 8234/7437 = 1{,}11$$

Podemos observar, no entanto, que duas instruções de mov foram comentadas na figura 5.5. Estas instruções são redundantes e, por isto, podem ser retiradas. Se não fossem retiradas, ainda assim encontraríamos um *speedup* satisfatório:

$$speedup = 8234/7583 = 1{,}09$$

A figura 5.7 ilustra a utilização das unidades central e de ponto-flutuante do código já otimizado.

```
trap r5, r5, r0   // turn on the report generation
      mov r21, r18
      fiadd.dd f0, f0, f8
//      mov r21, r18
//      mov r5, r19
      mov r5, r19
      adds 7, r0, r17
      bla r16, r17,.B124
      pfmul.dd f0, f0, f0
// lineno: 62
.B124:  //.M0000
.align 8
      fld.d r0(r18), f16
      pfadd.dd  f0,  f0,  f0
.DB.B124124:
      fld.d r0(r19), f18
      addu 8, r18, r18
      pfmul.dd f18, f16, f0
      bla r16, r17, .PL1002
      addu 8, r19, r19
.PL1002:
      fld.d r0(r18), f16
      pfmul.dd f0, f0, f0
      fld.d r0(r19), f18
      mi2pl.dd f8, f0, f30
      addu 8, r18, r18
      pfadd.dd f0, f0, f0
      addu 8, r19, r19
      mm12msm.dd f18, f16, f0
      trap r6, r6, r0   // turn off the report generation
      bla r16, r17, .PL1002
      pfadd.dd f0, f0, f8
```

Figura 5.5. - Trechos de programa I e II após as otimizações.

INSTRUCTION PIPELINE				CLK	SRC1	SRC2	DEST	CORE	FP	FPADD			FPMPLY			GRPH	ICH	ICM	DCH	DCM	CNA	MA	OPM
FETCH	DECODE	EXEC.	WRITE	-	-	-	-	-		S1	S2	S3	S1	S2	S3								
---	---	---	---	179	--	--	-	-		-	-	-	-	-	-	-	-	-	-	-	-	-	S
fld.y	pfmul.p	bla	adds	180	r16	r17	--	X		X	X	X	X	X	X	-	X	-	-	-	-	-	S
---	---	---	---	180	--	--	--	-		-	-	-	-	-	-	-	-	-	-	-	-	-	S
pfadd.p	fld.y	pfmul.p	---	181	df0	df0	df0	X	X	X	X	X	X	X	X	-	X	-	-	-	-	-	S
---	---	bla	---	181	r16	r17	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fld.y	pfadd.p	fld.y	---	182	r0	r18	f16	X	X	X	X	X	X	X	X	-	X	-	X	-	X	-	S
---	---	pfmul.p	bla	182	df0	df0	df0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
addu	fld.y	pfadd.p	---	183	df0	df0	df0	X	X	X	X	X	X	X	X	-	X	-	X	-	X	-	S
---	---	fld.y	pfmul.p	183	r0	r18	f16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
pfmul.p	addu	fld.y	pfadd.p	184	r0	r19	f18	X	-	X	X	X	X	X	X	-	X	-	X	-	X	-	S
---	---	---	fld.y	184	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	fld.y	---	185	r0	r19	f18	X	-	X	X	X	X	X	X	-	-	-	X	-	X	-	S
---	---	---	---	185	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
bla	pfmul.p	addu	fld.y	186	--	r18	r18	X	-	X	X	X	X	X	X	-	X	-	-	-	-	-	S
---	---	---	---	186	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
addu	bla	pfmul.p	addu	187	df18	df16	df0	-	X	X	X	X	X	X	X	-	X	-	-	-	-	-	S
---	---	---	---	187	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fld.y	addu	bla	---	188	r16	r17	--	X	X	X	X	X	X	X	X	-	X	-	-	-	-	-	S
---	---	pfmul.p	---	188	df18	df16	df0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
pfmul.p	fld.y	addu	bla	189	--	r19	r19	X	-	X	X	X	X	X	X	-	X	-	-	-	-	-	S
---	---	---	pfmul.p	189	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fld.y	pfmul.p	fld.y	addu	190	r0	r18	f16	X	-	X	X	X	X	X	X	-	X	-	X	-	X	-	S
---	---	---	---	190	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
Mi2p1	fld.y	pfmul.p	---	191	df0	df0	df0	X	X	X	X	X	X	X	X	-	X	-	X	-	X	-	S
---	---	fld.y	---	191	r0	r18	f16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
addu	Mi2p1	fld.y	---	192	r0	r19	f18	X	X	X	X	X	X	X	X	-	X	-	X	-	X	-	S
---	---	pfmul.p	fld.y	192	df0	df0	df0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
pfadd.p	addu	Mi2p1	---	193	df8	df0	df30	X	X	X	X	X	X	X	X	-	X	-	X	-	X	-	S
---	---	fld.y	pfmul.p	193	r0	r19	f18	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S

Figura 5.6. - Trecho do relatório completo que inclui as passagens pelos trechos I e II do programa, após a otimização.

RELATORIO DE SAIDA DO SIMULADOR pagina 6

INSTRUCTION PIPELINE				CLK	SRC1	SRC2	DEST	CORE	FP	FPADD			FPMPLY			GRPH	ICH	ICM	DCH	DCM	CNA	MA	OPM
FETCH	DECODE	EXEC.	WRITE	-	-	-	-	-		S1	S2	S3	S1	S2	S3								
pfmul.p	fld.y	addu	bla	189	--	r19	r19	X	-	X	X	X	X	X	X	-	X	-	-	-	-	-	S
---	---	---	pfmul.p	189	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fld.y	pfmul.p	fld.y	addu	190	r0	r18	f16	X	-	X	X	X	X	X	X	-	X	-	X	-	X	-	S
---	---	---	---	190	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
Mi2p1	fld.y	pfmul.p	---	191	df0	df0	df0	X	X	X	X	X	X	X	X	-	X	-	X	-	X	-	S
---	---	fld.y	---	191	r0	r18	f16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
addu	Mi2p1	fld.y	---	192	r0	r19	f18	X	X	X	X	X	X	X	X	-	X	-	X	-	X	-	S
---	---	pfmul.p	fld.y	192	df0	df0	df0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
pfadd.p	addu	Mi2p1	---	193	df8	df0	df30	X	X	X	X	X	X	X	X	-	X	-	X	-	X	-	S
---	---	fld.y	pfmul.p	193	r0	r19	f18	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
addu	pfadd.p	addu	---	194	--	r18	r18	X	X	X	X	X	X	X	X	-	X	-	-	-	-	-	S
---	---	Mi2p1	fld.y	194	df8	df0	df30	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
Mm12asm	addu	pfadd.p	addu	195	df0	df0	df0	-	X	X	X	X	X	X	X	-	X	-	-	-	-	-	S
---	---	---	Mi2p1	195	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
pfsm.p	Mm12asm	addu	pfadd.p	196	--	r19	r19	X	-	X	X	X	X	X	X	-	X	-	-	-	-	-	S
---	---	---	---	196	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
bla	pfsm.p	Mm12asm	addu	197	df18	df16	df0	-	X	X	X	X	X	X	X	-	X	-	-	-	-	-	S
---	---	---	---	197	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S

Figura 5.6. - Trecho do relatório completo que inclui as passagens pelos trechos I e II do programa, após a otimização (continuação).

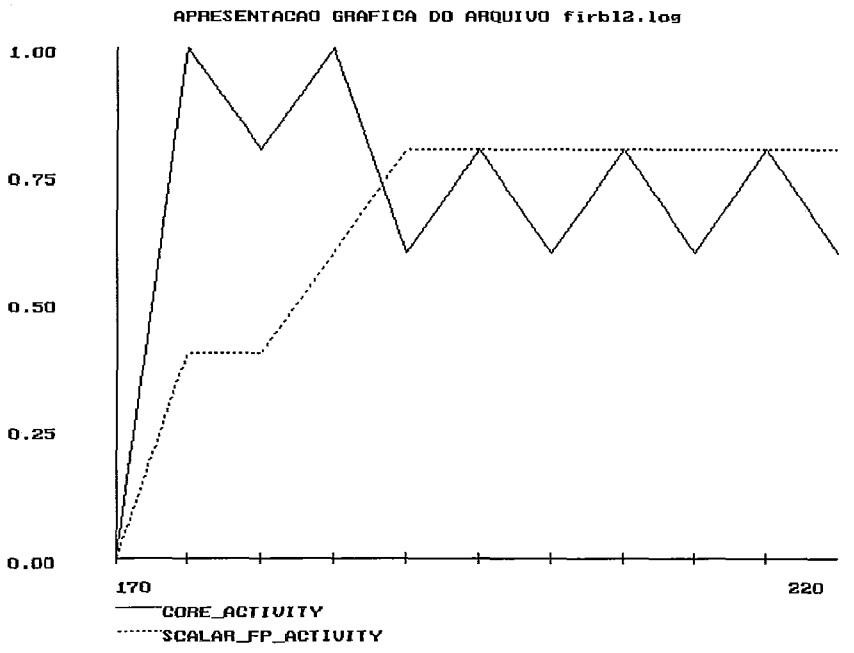


Figura 5.7. - Percentagem de utilização das unidades central e de ponto-flutuante durante execução do programa de implementação do filtro FIR, após a otimização.

5.2.2. Cálculo de um elemento da série Fibonacci.

Este exemplo busca, na realidade, analisar um programa recursivo que trabalha com inteiros. O programa calcula o quinto elemento da série de Fibonacci. A listagem do programa em C se encontra no apêndice A.

O primeiro fato observado é que todos os níveis de otimização do compilador forneceram o mesmo código (figura 5.8). Entretanto, isto não significa impossibilidade de conseguir-se algum grau de otimização.

```
        .file   "fibo5.c"
// PGC Rel 2.1 -opt 4

        .data
        .align  16
stackarea   :  .long [400]10        // data area to be used as a stack
stackend    :  .long 0
framearea   :  .long [400]20        // data area to be used as a frame
register_dump:  .long [20]10

        .text
        .align  8

_L00TEXT:
        .text; .align  4

//Stack allocation: Autos: Regsave:

_start::
        // stack initialization
        mov     stackend, sp
        trap r5, r5, r0    // turn on the report generation

        // Call the user program
        call    _main
        nop

        // Signal that the simulator MUST halt

        trap r6, r6, r0    // turn off the report generation
        trap    0,r0,r0

        .text
        .globl  _main
        .align  8

_main:
.a1 = 0
.f1 = 32
        addu -(.a1+.f1), sp, sp
        st.l fp,(.f1-16)(sp)
        addu (.f1-16), sp, fp
        st.l r1, 4(fp)
// lineno: 25
        adds 5, r0, r16
        call _fib
        nop
        st.l r16, -4(fp)
```

Figura 5.8. - Código obtido pelo compilador para obtenção do quinto termo da série de Fibonacci.

```
// lineno: 30
        adds .a1+16, fp, r31
        ld.l 4(fp), r1
        ld.l 0(fp), fp
        bri r1
        mov r31, sp
        .globl  _fib
        .align  8
_fib:
.a2 = 80
.f2 = 48
        addu -(.a2+.f2), sp, sp
        st.l fp,(.f2-16)(sp)
        addu (.f2-16), sp, fp
        st.l r1, 4(fp)
        st.l r4, -16(fp)
        mov r16, r4
// lineno: 0
// lineno: 35
        subs 2, r16, r0
        bnc .B58
        call _fib
        adds -2, r4, r16
        mov r16, r28
        adds -1, r4, r16
        call _fib
        st.l r28, -4(fp)
        ld.l -4(fp), r28
        br .B69
        adds r16, r28, r16
// lineno: 38
.B58:   //.B0000
        adds 1, r0, r16
.DB.B5858:
// lineno: 0
.B69:   //.R0000
// lineno: 39
        ld.l -16(fp), r4
        adds .a2+16, fp, r31
        ld.l 4(fp), r1
        ld.l 0(fp), fp
        bri r1
        mov r31, sp
```

Figura 5.8. - Código obtido pelo compilador para obtenção do quinto termo da série de Fibonacci (continuação).

Neste caso, como a unidade de ponto-flutuante não é utilizada, qualquer decréscimo de utilização porventura ocorrente na unidade central só pode ser devida a faltas na *cache* de dados ou a dependência de registradores entre instruções próximas.

A figura 5.9 apresenta o gráfico da percentagem da ocorrência de faltas nas *caches* de dados e de instrução, e a informação da percentagem de utilização da unidade central.

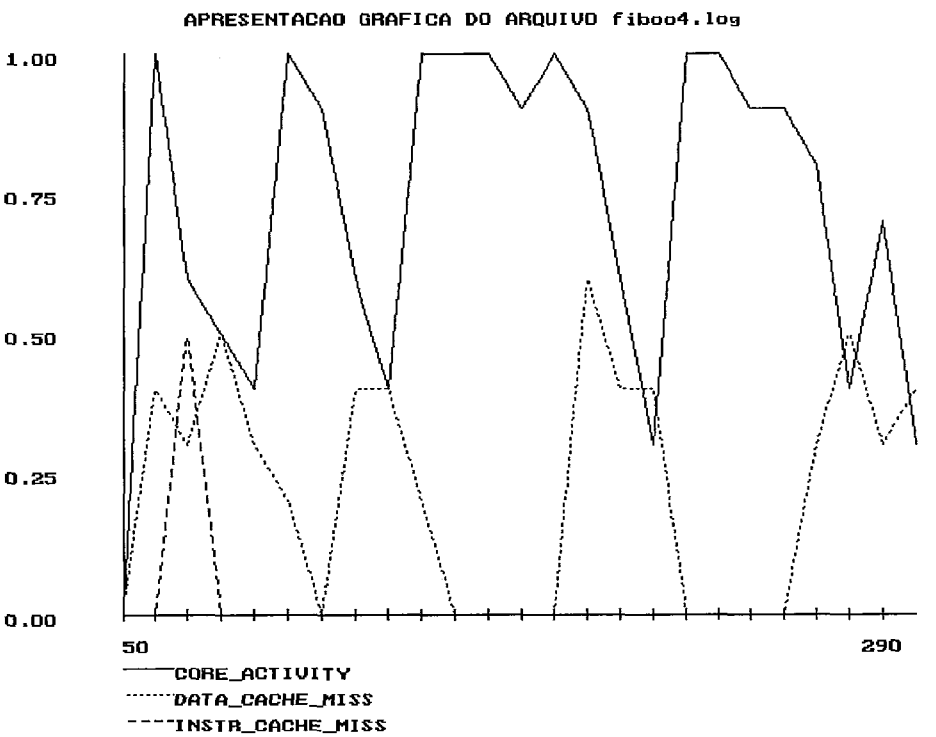


Figura 5.9. - Percentagem de utilização das unidades central e da ocorrência de faltas nas *caches* de dados e de instruções.

Como a baixa utilização da unidade central é devida a faltas na *cache* de dados (ciclos de relógio 90, 130 e 210), imaginamos que talvez com um rearranjo das instruções pudéssemos melhorar sua utilização. Todavia, verificamos através da avaliação do relatório completo do programa, que seria impossível deslocar qualquer instrução que fosse potencialmente capaz de gerar conflitos.

5.2.3 Solução de sistemas lineares através do método de Eliminação Gaussiana.

A solução de sistemas lineares pelo método de eliminação gaussiana é muito empregado em computação. Este fato, por si só, torna seu estudo interessante, embora sua otimização não seja imperativa como no caso da implementação do filtro FIR.

O programa em linguagem C que implementa a solução de sistemas lineares através do método de eliminação gaussiana é apresentado no apêndice A.

O programa em *assembly* gerado pelo compilador com nível de otimização 4 (mais eficiente) é muito extenso. Por isso mesmo, optamos pela apresentação dos trechos relevantes do programa (apresentado *in extenso* no apêndice A) à medida em que for sendo descrito o desenvolvimento da metodologia para este problema. Em primeiro lugar, localizamos no programa *assembly* os *loops* do programa em C. Em seguida, analisamos cada *loop* separadamente, pois torna-se mais fácil otimizar cada trecho *per se*.

O primeiro *loop* analisado foi o que fornece a solução final do sistema após a pivotação. O trecho de programa a ser analisado é mostrado na figura 5.10 .

Como pode ser visto, as instruções *trap* r5, r5, r0 e *trap* r6, r6, r0 estão delimitando o *loop*. Desta forma, restringe-se a análise a este trecho do programa. Novamente, para eliminar a interferência de faltas na *cache* de instruções, é avaliada a segunda passagem da execução por este trecho.

O gráfico que informa sobre a utilização da unidade central e de ponto-flutuante no intervalo de tempo necessário para a execução desse trecho é apresentado na figura 5.11 . Os números na figura 5.11 indicam pontos que devem ser analisados. A análise do relatório detalhado mostra que não é possível a otimização nos 4 primeiros pontos, mas que é possível otimizar as situações V, VI e VII.

```
        trap r5,r5,r0    // turn-on the detailed report generation
.DB.B295295:
        adds -16, fp, r30
        shl 2, r28, r29
        fld.l r30(r29), f16
        adds 40, r0, r16
        ixfr r16, f18
        ld.l r0(r21), r24
        fmlow.dd f18, f16, f20
        fxfr f20, r22
        adds 40, r22, r23
        adds r23, r4, r14
        fld.d r0(r14), f22
        fst.d f22, r0(r20)
        bte 0x0000, r24, .B241
        mov r6, r19
        mov r14, r18
        adds 1, r0, r25
        subs r24, r25, r0
        st.l r25, 0(r17)
        bc .B244
        mov r14, r18
        mov r6, r19
// lineno: 103
.B283: //.M0000
        fld.d -8(r18)++, f16
.DB.B283283:
        fld.d -8(r19)++, f18
        fld.d r0(r20), f22
        ld.l r0(r17), r28
        fmul.dd f18, f16, f20
        adds 1, r28, r29
        st.l r29, 0(r17)
        fsub.dd f22, f20, f24
        ld.l r0(r21), r30
        subs r30, r29, r0
        fst.d f24, r0(r20)
        bnc.t .DB.B283283
        fld.d -8(r18)++, f16
// lineno: 106
.B244: //.B0015
// lineno: 108
.B241: //.B0013
        ld.l r0(r13), r28
```

Figura 5.10 - Trecho do programa que determina a solução final do sistema linear.

```
.DB.B241241:
    ld.l r0(r21), r22
    adds -4, r28, r29
    shl 3, r29, r30
    fld.d r14(r30), f16
    shl 3, r28, r16
    adds 1, r22, r23
    frcp.dd f16, f26          // f16=A, f26=1/A

    st.l r23, 0(r21)
    adds -1, r28, r24
    fld.d r0(r20), f18        // f18=Y
    fmul.dd f18, f26, f28     // f28=Y/A
    st.l r24, 0(r13)
    adds -4, r23, r0
    fst.d f28, r16(r5)
    bc.t .DB.B295295
    ld.l r0(r13), r28

// lineno: 0
    trap r6,r6,r0    // turn-off the detailed report generation
```

Figura 5.10 - Trecho do programa que determina a solução final do sistema linear (continuação).

A figura 5.12 refere-se à mesma experiência ilustrada na figura 5.11, porém o gráfico, agora apresentado, focaliza o intervalo entre os ciclos de relógio 1400 e 1500 e apresenta um intervalo de amostragem de 5 ciclos de relógio.

Torna-se mais fácil, agora, analisar os três últimos casos. Os trechos do relatório que mostram situações de conflito de utilização de recursos relativos a estes casos são apresentados na figura 5.13 .

Em V, as instruções fmlow e fxfr estão requisitando a mesma unidade funcional, o que não é desejável. Para agravar a situação, a instrução fxfr utiliza o registrador destino da fmlow, registrador este que só se torna disponível dois ciclos de relógio depois da execução do fmlow. A única movimentação possível é colocar o ld.l entre o fmlow e o fxfr. Contudo, observando-se o relatório após a otimização, figura 5.14, verifica-se que o intervalo de tempo delimitado pelo início da operação do ixfr e o início da operação fxfr é igual nos dois casos, impossibilitando a obtenção de ganho. Podemos observar, contudo, que a instrução adds após a operação fxfr fica aguardando a disponibilidade do registrador r22. Esta espera não seria necessária se o compilador constatasse que o par de instruções

```
adds    40, r22, r23
adds     r23, r4, r14
```

poderia ser substituído pelo par

```
adds    40, r4,  r14
adds    r14, r22, r14
```

Esta alteração elimina a dependência entre registradores entre o fxfr e o primeiro adds, eliminando 1 ciclo de relógio de espera.

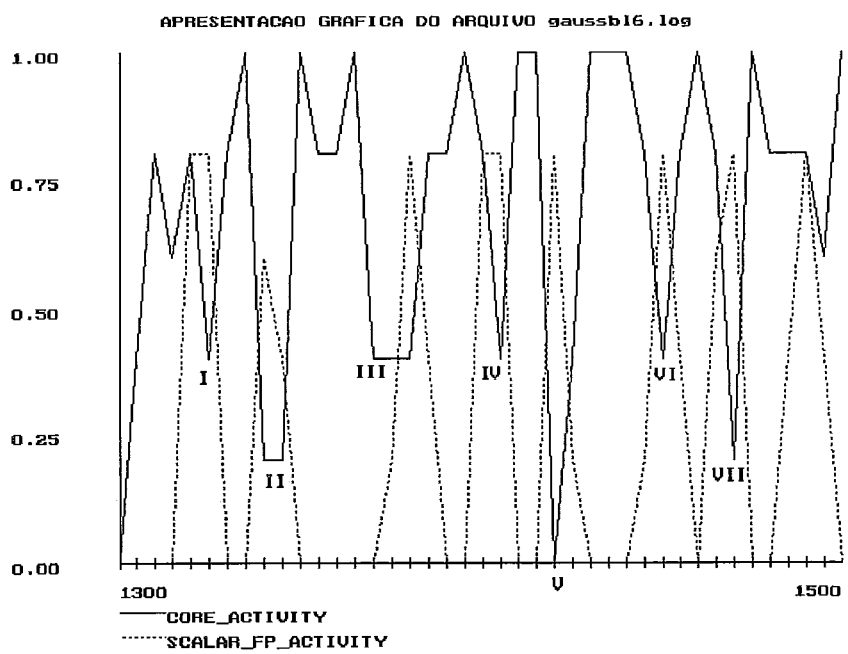


Figura 5.11. - Percentagem de utilização das unidades central e de ponto-flutuante durante execução de um trecho do programa para solução de sistemas lineares pelo método de eliminação gaussiana.

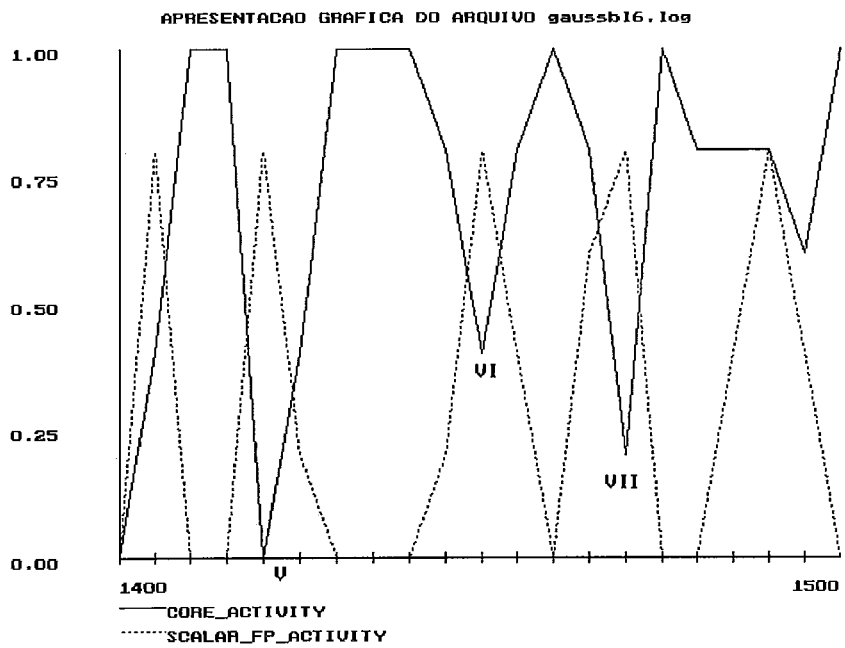


Figura 5.12. - Percentagem de utilização das unidades central e de ponto-flutuante durante execução de um programa de solução de sistemas lineares, durante outro intervalo de tempo.

INSTRUCTION PIPELINE				CLK	SRC1	SRC2	DEST	CORE	FP	FPADD			FPMPLY		GRPH	ICH	ICM	DCH	DCM	CNA	MA	OPM
FETCH	DECODE	EXEC.	WRITE	-	-	-	-	-		S1	S2	S3	S1	S2	S3								
ld.x	ixfr	adds	fld.y	1413	--	r0	r16	X	-	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1413	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fmlow.d	ld.x	ixfr	adds	1414	r16	--	f18	X	-	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1414	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fxfr	fmlow.d	ld.x	ixfr	1415	r0	r21	r24	X	-	-	-	-	-	-	-	-	X	-	X	-	X	-	S
---	---	---	---	1415	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	ld.x	---	1416	r0	r21	r24	-	-	-	-	-	-	-	-	-	-	X	-	X	-	-	S
---	---	---	---	1416	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
adds	fxfr	fmlow.d	ld.x	1417	df18	df16	df20	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1417	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	fmlow.d	---	1418	df18	df16	df20	-	X	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	---	1418	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	fmlow.d	---	1419	df18	df16	df20	-	X	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	---	1419	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	fmlow.d	---	1420	df18	df16	df20	-	X	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	---	1420	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	fmlow.d	1421	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	---	1421	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
adds	adds	fxfr	---	1422	f20	f0	r22	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1422	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	fxfr	1423	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	---	1423	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fld.y	adds	adds	---	1424	--	r22	r23	X	-	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1424	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fst.y	fld.y	adds	adds	1425	r23	r4	r14	X	-	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1425	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
bte	fst.y	fld.y	adds	1426	r0	r14	f22	X	-	-	-	-	-	-	-	-	X	-	X	-	X	-	S
---	---	---	---	1426	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S

Figura 5.13. - Trecho do relatório completo que inclui os conflitos existentes para o problema da solução de sistemas lineares.

INSTRUCTION PIPELINE				CLK	SRC1	SRC2	DEST	CORE	FP	FPADD	FPMPLY	GRPH	ICH	ICM	DCH	DCM	CNA	MA	OPM
FETCH	DECODE	EXEC.	WRITE	-	-	-	-	-		S1 S2 S3	S1 S2 S3								
fld.y	fld.y	fld.y	shl	1438	--	r18	f16	X	-	- - -	- - -	-	X	-	X	-	X	-	S
---	---	---	---	1438	--	--	--	-	-	- - -	- - -	-	-	-	-	-	-	-	S
---	---	fld.y	---	1439	--	r18	f16	X	-	- - -	- - -	-	-	-	X	-	X	-	S
---	---	---	---	1439	--	--	--	-	-	- - -	- - -	-	-	-	-	-	-	-	S
ld.x	fld.y	fld.y	fld.y	1440	--	r19	f18	X	-	- - -	- - -	-	X	-	X	-	X	-	S
---	---	---	---	1440	--	--	--	-	-	- - -	- - -	-	-	-	-	-	-	-	S
---	---	fld.y	---	1441	--	r19	f18	X	-	- - -	- - -	-	-	-	X	-	X	-	S
---	---	---	---	1441	--	--	--	-	-	- - -	- - -	-	-	-	-	-	-	-	S
fmul.p	ld.x	fld.y	fld.y	1442	r0	r20	f22	X	-	- - -	- - -	-	X	-	X	-	X	-	S
---	---	---	---	1442	--	--	--	-	-	- - -	- - -	-	-	-	-	-	-	-	S
---	---	fld.y	---	1443	r0	r20	f22	X	-	- - -	- - -	-	-	-	X	-	X	-	S
---	---	---	---	1443	--	--	--	-	-	- - -	- - -	-	-	-	-	-	-	-	S
adds	fmul.p	ld.x	fld.y	1444	r0	r17	r28	X	-	- - -	- - -	-	X	-	X	-	X	-	S
---	---	---	---	1444	--	--	--	-	-	- - -	- - -	-	-	-	-	-	-	-	S
st.x	adds	fmul.p	---	1445	df18	df16	df20	-	X	- - -	- - -	-	X	-	X	-	X	-	S
---	---	ld.x	---	1445	r0	r17	r28	-	-	- - -	- - -	-	-	-	-	-	-	-	S
fsub.p	st.x	adds	---	1446	--	r28	r29	X	X	- - -	- - -	-	X	-	-	-	-	-	S
---	---	fmul.p	ld.x	1446	df18	df16	df20	-	-	- - -	- - -	-	-	-	-	-	-	-	S
ld.x	fsub.p	st.x	adds	1447	r29	r17	--	X	X	- - -	- - -	-	X	-	X	-	X	-	S
---	---	fmul.p	---	1447	df18	df16	df20	-	-	- - -	- - -	-	-	-	-	-	-	-	S
---	---	st.x	---	1448	r29	r17	--	-	X	- - -	- - -	-	-	-	X	-	X	-	S
---	---	fmul.p	---	1448	df18	df16	df20	-	-	- - -	- - -	-	-	-	-	-	-	-	S
---	---	---	st.x	1449	--	--	--	-	-	- - -	- - -	-	-	-	-	-	-	-	S
---	---	---	fmul.p	1449	--	--	--	-	-	- - -	- - -	-	-	-	-	-	-	-	S
subs	ld.x	fsub.p	---	1450	df22	df20	df24	-	X	- - -	- - -	-	X	-	-	-	-	-	S
---	---	---	---	1450	--	--	--	-	-	- - -	- - -	-	-	-	-	-	-	-	S

Figura 5.13. - Trecho do relatório completo que inclui os conflitos existentes para o problema da solução de sistemas lineares (continuação).

INSTRUCTION PIPELINE				CLK	SRC1	SRC2	DEST	CORE	FP	FPADD		FPMPLY		GRPH	ICH	ICM	DCH	DCM	CNA	MA	OPM
FETCH	DECODE	EXEC.	WRITE	-	-	-	-	-		S1	S2	S3	S1	S2	S3								
ld.x	fmlow.d	ixfr	adds	1412	r16	--	f18	X	-	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1412	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	ixfr	1413	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	---	1413	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	---	1414	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	---	1414	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fxfr	ld.x	fmlow.d	---	1415	df18	df16	df20	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1415	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
adds	fxfr	ld.x	---	1416	r0	r21	r24	X	X	-	-	-	-	-	-	-	X	-	-	X	-	X	S
---	---	fmlow.d	---	1416	df18	df16	df20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	S
---	---	ld.x	---	1417	r0	r21	r24	-	X	-	-	-	-	-	-	-	-	-	-	X	-	X	S
---	---	fmlow.d	---	1417	df18	df16	df20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	S
---	---	---	ld.x	1418	--	--	--	-	X	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	fmlow.d	---	1418	df18	df16	df20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	---	1419	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	fmlow.d	1419	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
adds	adds	fxfr	---	1420	f20	f0	r22	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1420	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fld.y	adds	adds	fxfr	1421	--	r4	r14	X	-	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1421	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fst.y	fld.y	adds	adds	1422	r14	r22	r14	X	-	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1422	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
bte	fst.y	fld.y	adds	1423	r0	r14	f22	X	-	-	-	-	-	-	-	-	X	-	-	X	-	X	S
---	---	---	---	1423	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	S
---	---	fld.y	---	1424	r0	r14	f22	X	-	-	-	-	-	-	-	-	-	-	-	X	-	X	S
---	---	---	---	1424	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	S
shl	bte	fst.y	fld.y	1425	r0	r20	f22	X	-	-	-	-	-	-	-	-	X	-	-	X	-	X	S
---	---	---	---	1425	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	S
shl	shl	bte	fst.y	1426	--	r24	--	X	-	-	-	-	-	-	-	-	X	-	-	X	-	X	S
---	---	---	---	1426	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	S

Figura 5.14. - Trecho do relatório completo após uma modificação para solucionar os conflitos de recursos existentes para o problema da solução de sistemas lineares.

INSTRUCTION PIPELINE				CLK	SRC1	SRC2	DEST	CORE	FP	FPADD			FPMPLY			GRPH	ICH	ICM	DCH	DCM	CNA	MA	OPM
FETCH	DECODE	EXEC.	WRITE	-	-	-	-	-		S1	S2	S3	S1	S2	S3								
fld.y	fld.y	shl	shl	1434	r0	r6	r19	X	-	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1434	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
ld.x	fld.y	fld.y	shl	1435	--	r18	f16	X	-	-	-	-	-	-	-	-	X	-	X	-	X	-	S
---	---	---	---	1435	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
---	---	fld.y	---	1436	--	r18	f16	X	-	-	-	-	-	-	-	-	-	-	X	-	X	-	S
---	---	---	---	1436	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
fmul.p	ld.x	fld.y	fld.y	1437	--	r19	f18	X	-	-	-	-	-	-	-	-	X	-	X	-	X	-	S
---	---	---	---	1437	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
---	---	fld.y	---	1438	--	r19	f18	X	-	-	-	-	-	-	-	-	-	-	X	-	X	-	S
---	---	---	---	1438	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
fld.y	fmul.p	ld.x	fld.y	1439	r0	r17	r28	X	-	-	-	-	-	-	-	-	X	-	X	-	X	-	S
---	---	---	---	1439	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
adds	fld.y	fmul.p	---	1440	df18	df16	df20	-	X	-	-	-	-	-	-	-	X	-	X	-	X	-	S
---	---	ld.x	---	1440	r0	r17	r28	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
ld.x	adds	fld.y	---	1441	r0	r20	f22	X	X	-	-	-	-	-	-	-	X	-	X	-	X	-	S
---	---	fmul.p	ld.x	1441	df18	df16	df20	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
---	---	fld.y	---	1442	r0	r20	f22	X	X	-	-	-	-	-	-	-	-	-	X	-	X	-	S
---	---	fmul.p	---	1442	df18	df16	df20	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
fsub.p	ld.x	adds	fld.y	1443	--	r28	r29	X	X	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	fmul.p	---	1443	df18	df16	df20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
st.x	fsub.p	ld.x	adds	1444	r0	r21	r30	X	-	-	-	-	-	-	-	-	X	-	X	-	X	-	S
---	---	---	fmul.p	1444	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
subs	st.x	fsub.p	---	1445	df22	df20	df24	-	X	-	-	-	-	-	-	-	X	-	X	-	X	-	S
---	---	ld.x	---	1445	r0	r21	r30	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
fst.y	subs	st.x	---	1446	r29	r17	--	X	X	-	-	-	-	-	-	-	X	-	X	-	X	-	S
---	---	fsub.p	ld.x	1446	df22	df20	df24	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
bnc.t	fst.y	subs	---	1447	r30	r29	r0	X	X	-	-	-	-	-	-	-	X	-	X	-	X	-	S
---	---	st.x	---	1447	r29	r17	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S

Figura 5.14. - Trecho do relatório completo após uma modificação para solucionar os conflitos de recursos existentes para o problema da solução de sistemas lineares (Cont.).

Em VI, temos duas operações de fld seguidas de uma operação de ld, ou seja, três operações de leitura na memória. Se for possível deslocar algumas destas instruções algum ganho poderá ser obtido. Em VII, na realidade, temos nada mais do que uma segunda passagem pelo trecho de programa analisado em VI. Estas modificações também podem ser vistas na figura 5.14. A figura 5.15 ilustra a percentagem de utilização das unidades central e de ponto flutuante depois de feitas as otimizações.

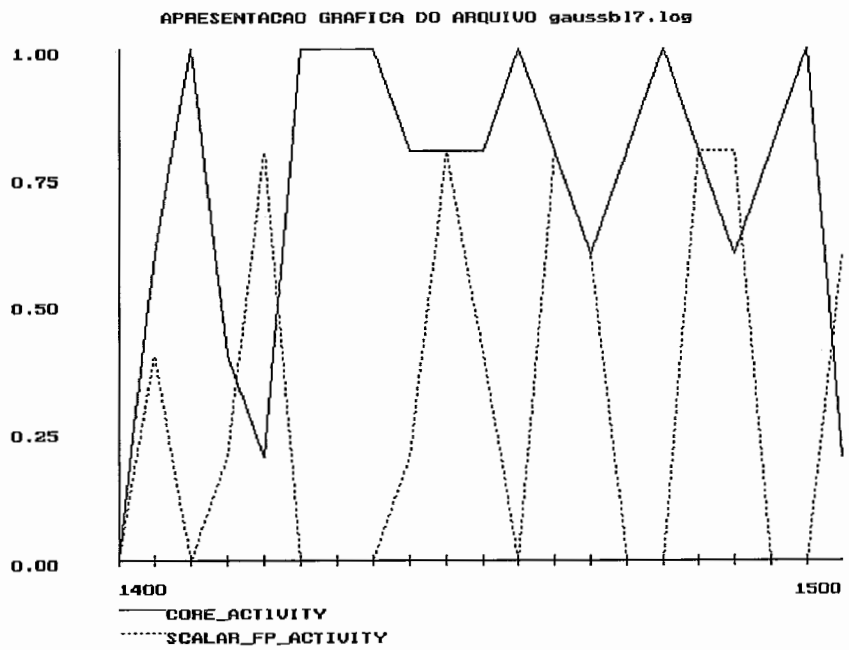


Figura 5.15. - Percentagem de utilização das unidades central e de ponto-flutuante durante execução de um programa de solução de sistemas lineares após algumas otimizações.

O segundo *loop* a ser otimizado corresponde ao que executa a pivotação. O trecho de programa em *assembly* correspondente a este *loop* é apresentado na figura 5.16.

```
trap r5,r5,r0    // turn-on the detailed report generation
.B296: //.M0004
      fld.d r0(r20), f16
.DB.B296296:
      frcp.dd f16, f24
      ld.l r0(r17), r28
      adds -16, fp, r30
      shl 2, r28, r29
      adds 40, r0, r16
      ld.l r0(r21), r23
      fld.l r30(r29), f18
      shl 3, r23, r24
      mov r14, r18
      ixfr r16, f28
      fmlow.dd f18, f28, f30
      ld.l r0(r21), r26
      fxfr f30, r22
      adds r24, r22, r25
      adds r25, r12, r19
      fld.d r0(r19), f26          //f26=Const
      fmul.dd f26, f24, f14
      adds 1, r26, r27
      adds -5, r27, r0
      fst.d f0, r0(r19)
      fst.d f14, r0(r5)
      st.l r27, 0(r4)
      bnc .B236
      mov r14, r18
.B284: //.M0001
      fld.d 8(r18)++, f16
.DB.B284284:
      fld.d r0(r5), f18
      fld.d 8(r19)++, f22
      fmul.dd f16, f18, f20
      ld.l r0(r4), r28
      adds 1, r28, r29
      fsub.dd f22, f20, f24
      adds -5, r29, r0
      st.l r29, 0(r4)
      fst.d f24, r0(r19)
      bc.t .DB.B284284
      fld.d 8(r18)++, f16
.B236: //.B0010
      ld.l r0(r17), r28
.DB.B236236:
      adds 1, r28, r29
      adds -4, r29, r0
      st.l r29, 0(r17)
      bc.t .DB.B296296
      fld.d r0(r20), f16
trap r6,r6,r0    // turn-off the detailed report generation
```

Figura 5.16. - Trecho de programa em *assembly* que efetua a pivotação.

Novamente, as instruções *trap* r5, r5, r0 e *trap* r6, r6, r0 delimitam o trecho que será analisado e estará contido no relatório completo gerado pelo simulador.

A figura 5.17 apresenta o gráfico que informa sobre a utilização das unidades central e de ponto-flutuante. Existem dois pontos de utilização mínima da unidade central, provavelmente em razão da ausência de paralelismos das instruções da unidade central e de ponto-flutuante.

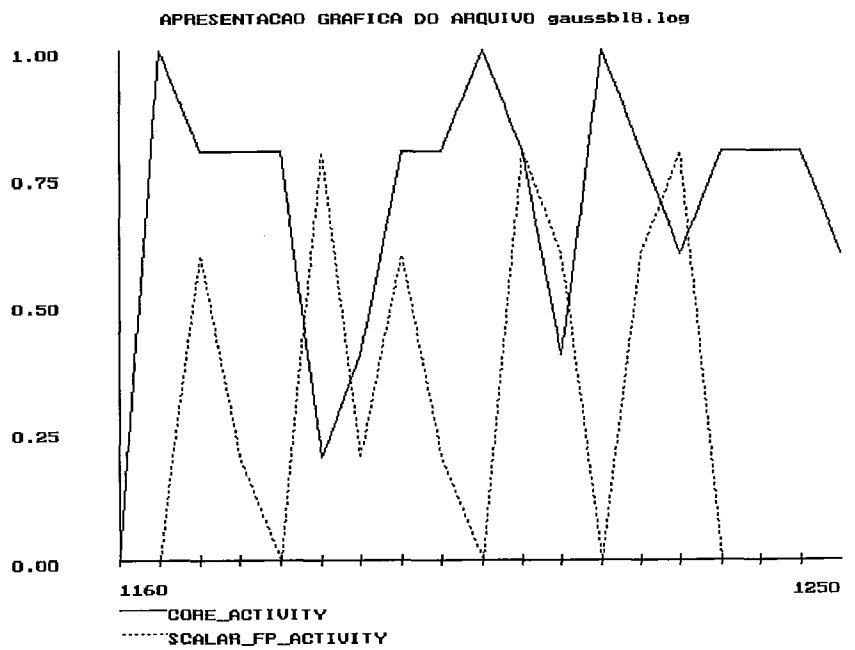


Figura 5.17. - Percentagem de utilização das unidades central e de ponto-flutuante durante a pivotação no programa para solução de sistemas lineares.

No primeiro ponto de mínimo observamos um problema semelhante ao trecho primeiramente analisado, como mostra o relatório ilustrado na figura 5.18. No entanto, neste caso foi possível movimentar instruções de cima para entre o fmlow e o fxfr.

INSTRUCTION PIPELINE				CLK	SRC1	SRC2	DEST	CORE	FP	FPADD			FPMPLY			GRPH	ICH	ICM	DCH	DCM	CNA	MA	OPM
FETCH	DECODE	EXEC.	WRITE	-	-	-	-	-		S1	S2	S3	S1	S2	S3								
ixfr	shl	shl	fld.y	1177	--	r23	r24	X	-	X	X	X	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1177	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fmlow.d	ixfr	shl	shl	1178	r0	r14	r18	X	-	X	X	X	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1178	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
ld.x	fmlow.d	ixfr	shl	1179	r16	--	f28	X	-	X	X	X	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1179	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	ixfr	1180	--	--	--	-	-	X	X	X	-	-	-	-	-	-	-	-	-	-	S
---	---	---	---	1180	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	---	1181	--	--	--	-	-	X	X	X	-	-	-	-	-	-	-	-	-	-	S
---	---	---	---	1181	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fxfr	ld.x	fmlow.d	---	1182	df18	df28	df30	-	X	X	X	X	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1182	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
adds	fxfr	ld.x	---	1183	r0	r21	r26	X	X	X	X	X	-	-	-	-	X	-	-	X	-	X	S
---	---	fmlow.d	---	1183	df18	df28	df30	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	S
---	---	ld.x	---	1184	r0	r21	r26	-	X	X	X	X	-	-	-	-	-	-	-	X	-	X	S
---	---	fmlow.d	---	1184	df18	df28	df30	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	S
---	---	---	ld.x	1185	--	--	--	-	X	X	X	X	-	-	-	-	-	-	-	-	-	-	S
---	---	fmlow.d	---	1185	df18	df28	df30	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	---	1186	--	--	--	-	-	X	X	X	-	-	-	-	-	-	-	-	-	-	S
---	---	---	fmlow.d	1186	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
adds	adds	fxfr	---	1187	f30	f0	r22	-	X	X	X	X	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1187	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	fxfr	1188	--	--	--	-	-	X	X	X	-	-	-	-	-	-	-	-	-	-	S
---	---	---	---	1188	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fld.y	adds	adds	---	1189	r24	r22	r25	X	-	X	X	X	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1189	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fmul.p	fld.y	adds	adds	1190	r25	r12	r19	X	-	X	X	X	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1190	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
adds	fmul.p	fld.y	adds	1191	r0	r19	f26	X	-	X	X	X	-	-	-	-	X	-	-	X	-	X	S
---	---	---	---	1191	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	S

Figura 5.18. - Trecho do relatório completo para analisar os conflitos de recursos existentes no *loop* de pivotação, para o problema da solução de sistemas lineares.

INSTRUCTION PIPELINE				CLK	SRC1	SRC2	DEST	CORE	FP	FPADD		FPMPLY		GRPH	ICH	ICM	DCH	DCM	CNA	MA	OPM
FETCH	DECODE	EXEC.	WRITE	-	-	-	-	-		S1	S2	S3	S1	S2	S3						
fld.y	fld.y	shl	bnc	1201	r0	r14	r18	X	-	X	X	X	-	-	-	-	X	-	-	-	-	-	S
---	---	---	st.x	1201	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fld.y	fld.y	fld.y	shl	1202	--	r18	f16	X	-	X	X	X	-	-	-	-	X	-	-	X	-	X	S
---	---	---	---	1202	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	S
---	---	fld.y	---	1203	--	r18	f16	X	-	X	X	X	-	-	-	-	-	-	-	X	-	X	S
---	---	---	---	1203	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	S
fmul.p	fld.y	fld.y	fld.y	1204	r0	r5	f18	X	-	X	X	X	-	-	-	-	X	-	-	X	-	X	S
---	---	---	---	1204	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	S
---	---	fld.y	---	1205	r0	r5	f18	X	-	X	X	X	-	-	-	-	-	-	-	X	-	X	S
---	---	---	---	1205	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	S
ld.x	fmul.p	fld.y	fld.y	1206	--	r19	f22	X	-	X	X	X	-	-	-	-	X	-	-	X	-	X	S
---	---	---	---	1206	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	S
adds	ld.x	fmul.p	---	1207	df16	df18	df20	X	X	X	X	X	-	-	-	-	X	-	-	X	-	X	S
---	---	fld.y	---	1207	--	r19	f22	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	S
fsub.p	adds	ld.x	---	1208	r0	r4	r28	X	X	X	X	X	-	-	-	-	X	-	-	X	-	X	S
---	---	fmul.p	fld.y	1208	df16	df18	df20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	S
---	---	ld.x	---	1209	r0	r4	r28	-	X	X	X	X	-	-	-	-	-	-	-	X	-	X	S
---	---	fmul.p	---	1209	df16	df18	df20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	S
adds	fsub.p	adds	ld.x	1210	--	r28	r29	X	X	X	X	X	-	-	-	-	X	-	-	-	-	-	S
---	---	fmul.p	---	1210	df16	df18	df20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	adds	1211	--	--	--	-	-	X	X	X	-	-	-	-	-	-	-	-	-	-	S
---	---	---	fmul.p	1211	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
st.x	adds	fsub.p	---	1212	df22	df20	df24	-	-	X	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1212	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fst.y	st.x	adds	---	1213	--	r29	r0	X	X	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	fsub.p	---	1213	df22	df20	df24	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
bc.t	fst.y	st.x	adds	1214	r29	r4	--	-	X	X	-	-	-	-	-	-	X	-	X	-	X	-	S
---	---	fsub.p	---	1214	df22	df20	df24	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	S

Figura 5.18. - Trecho do relatório completo para analisar os conflitos de recursos existentes no *loop* de pivotação, para o problema da solução de sistemas lineares (continuação).

No segundo ponto de mínimo, temos novamente o problema dos flds seguidos, e também é possível efetuar alterações na sequência das instruções. Neste caso, também é posível efetuar a substituição do par

```
adds r24, r22, r25
adds r25, r21, r19
```

pelo par

```
adds r24, r21, r19
adds r19, r22, r19
```

Esta modificação facilita o remanejamento das instruções.

A figura 5.19 apresenta o novo código referente a este trecho do programa enquanto a figura 5.20 ilustra a percentagem de utilização das unidades central e de ponto flutuante ao longo da execução do segundo *loop* otimizado. O relatório do trecho de programa otimizado pode ser visto na figura 5.21.

```
trap r5,r5,r0    // turn-on the detailed report generation
// lineno: 75
.B296:  //.M0004
      fld.d r0(r20), f16
.DB.B296296:
       frcp.dd f16, f24
       ld.l r0(r17), r28
      adds -16, fp, r30
      shl 2, r28, r29
      adds 40, r0, r16
      fld.l r30(r29), f18
      ld.l r0(r21), r23
      ixfr r16, f28
      fmlow.dd f18, f28, f30
      shl 3, r23, r24
      mov r14, r18
      ld.l r0(r21), r26
      adds r24, r12, r19
      fxfr f30, r22
      adds 1, r26, r27
      adds r19, r22, r19
      fld.d r0(r19), f26        //f26=Const
      fmul.dd f26, f24, f14
```

Figura 5.19. - Trecho do programa responsável pela pivotação após a otimização.

```
        adds -5, r27, r0
        fst.d f0, r0(r19)
        st.l r27, 0(r4)
        fst.d f14, r0(r5)
        bnc .B236
        mov r14, r18
// lineno: 81
.B284:  //.M0001
        fld.d 8(r18)++, f16
.DB.B284284:
        fld.d r0(r5), f18
        ld.l r0(r4), r28
        fmul.dd f16, f18, f20
        fld.d 8(r19)++, f22
        adds 1, r28, r29
        fsub.dd f22, f20, f24
        adds -5, r29, r0
        st.l r29, 0(r4)
        fst.d f24, r0(r19)
        bc.t .DB.B284284
        fld.d 8(r18)++, f16
// lineno: 84
.B236:  //.B0010
        ld.l r0(r17), r28
.DB.B236236:
        adds 1, r28, r29
        adds -4, r29, r0
        st.l r29, 0(r17)
        bc.t .DB.B296296
        fld.d r0(r20), f16
// lineno: 0
trap r6,r6,r0     // turn-off the detailed report generation
```

Figura 5.19. - Trecho do programa responsável pela pivotação após a otimização (continuação).

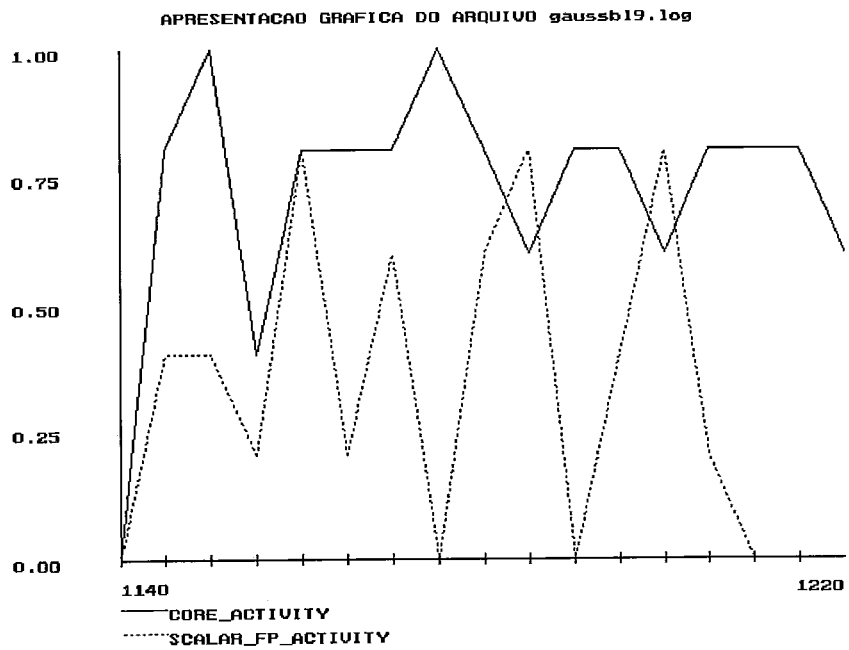
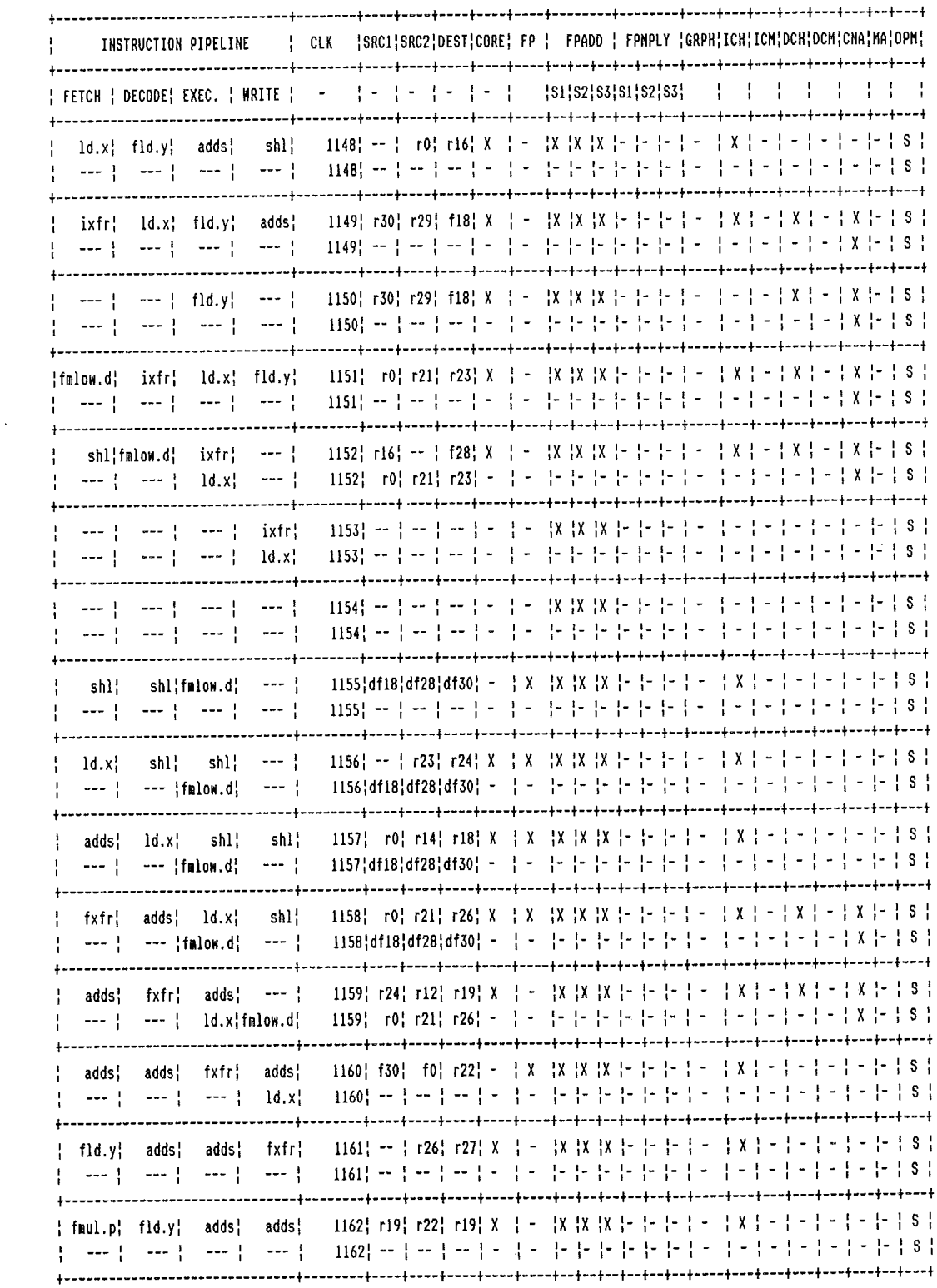


Figura 5.20. - Percentagem de utilização das unidades central e de ponto-flutuante durante a pivotação no programa para solução de sistemas lineares com código otimizado.



INSTRUCTION PIPELINE				CLK	SRC1	SRC2	DEST	CORE	FP	FPADD			FPMPLY			GRPH	ICH	ICM	DCH	DCM	CNA	MA	OPM
FETCH	DECODE	EXEC.	WRITE	-	-	-	-	-		S1	S2	S3	S1	S2	S3								
ld.x	fld.y	adds	shl	1148	--	r0	r16	X	-	X	X	X	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1148	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
ixfr	ld.x	fld.y	adds	1149	r30	r29	f18	X	-	X	X	X	-	-	-	-	X	-	X	-	X	-	S
---	---	---	---	1149	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
---	---	fld.y	---	1150	r30	r29	f18	X	-	X	X	X	-	-	-	-	-	-	X	-	X	-	S
---	---	---	---	1150	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
fmlow.d	ixfr	ld.x	fld.y	1151	r0	r21	r23	X	-	X	X	X	-	-	-	-	X	-	X	-	X	-	S
---	---	---	---	1151	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
shl	fmlow.d	ixfr	---	1152	r16	--	f28	X	-	X	X	X	-	-	-	-	X	-	X	-	X	-	S
---	---	ld.x	---	1152	r0	r21	r23	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
---	---	---	ixfr	1153	--	--	--	-	-	X	X	X	-	-	-	-	-	-	-	-	-	-	S
---	---	---	ld.x	1153	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	---	1154	--	--	--	-	-	X	X	X	-	-	-	-	-	-	-	-	-	-	S
---	---	---	---	1154	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
shl	shl	fmlow.d	---	1155	df18	df28	df30	-	X	X	X	X	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1155	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
ld.x	shl	shl	---	1156	--	r23	r24	X	X	X	X	X	-	-	-	-	X	-	-	-	-	-	S
---	---	fmlow.d	---	1156	df18	df28	df30	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
adds	ld.x	shl	shl	1157	r0	r14	r18	X	X	X	X	X	-	-	-	-	X	-	-	-	-	-	S
---	---	fmlow.d	---	1157	df18	df28	df30	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fxfr	adds	ld.x	shl	1158	r0	r21	r26	X	X	X	X	X	-	-	-	-	X	-	X	-	X	-	S
---	---	fmlow.d	---	1158	df18	df28	df30	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
adds	fxfr	adds	---	1159	r24	r12	r19	X	-	X	X	X	-	-	-	-	X	-	X	-	X	-	S
---	---	ld.x	fmlow.d	1159	r0	r21	r26	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
adds	adds	fxfr	adds	1160	f30	f0	r22	-	X	X	X	X	-	-	-	-	X	-	-	-	-	-	S
---	---	---	ld.x	1160	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fld.y	adds	adds	fxfr	1161	--	r26	r27	X	-	X	X	X	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1161	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fmul.p	fld.y	adds	adds	1162	r19	r22	r19	X	-	X	X	X	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1162	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S

Figura 5.21. - Trecho do relatório completo após uma modificação para solucionar os conflitos de recursos existentes no *loop* de pivotação, para o problema da solução de sistemas lineares.

INSTRUCTION PIPELINE				CLK	SRC1	SRC2	DEST	CORE	FP	FPADD			FPMPLY			GRPH	ICH	ICM	OCH	OCM	CNA	MA	OPM
FETCH	DECODE	EXEC.	WRITE	-	-	-	-	-		S1	S2	S3	S1	S2	S3								
fld.y	bc.t	fst.y	st.x	1187	r0	r19	f24	X	-	-	-	-	-	-	-	-	X	-	X	-	X	-	S
---	---	---	---	1187	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
fld.y	fld.y	bc.t	fst.y	1188	--	--	--	X	-	-	-	-	-	-	-	-	X	-	X	-	X	-	S
---	---	---	---	1188	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
ld.x	fld.y	fld.y	bc.t	1189	--	r18	f16	X	-	-	-	-	-	-	-	-	X	-	X	-	X	-	S
---	---	---	---	1189	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
---	---	fld.y	---	1190	--	r18	f16	X	-	-	-	-	-	-	-	-	-	-	X	-	X	-	S
---	---	---	---	1190	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
fmul.p	ld.x	fld.y	fld.y	1191	r0	r5	f18	X	-	-	-	-	-	-	-	-	X	-	X	-	X	-	S
---	---	---	---	1191	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
---	---	fld.y	---	1192	r0	r5	f18	X	-	-	-	-	-	-	-	-	-	-	X	-	X	-	S
---	---	---	---	1192	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
fld.y	fmul.p	ld.x	fld.y	1193	r0	r4	r28	X	-	-	-	-	-	-	-	-	X	-	X	-	X	-	S
---	---	---	---	1193	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
adds	fld.y	fmul.p	---	1194	df16	df18	df20	-	X	-	-	-	-	-	-	-	X	-	X	-	X	-	S
---	---	ld.x	---	1194	r0	r4	r28	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
fsub.p	adds	fld.y	---	1195	--	r19	f22	X	X	-	-	-	-	-	-	-	X	-	X	-	X	-	S
---	---	fmul.p	ld.x	1195	df16	df18	df20	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
---	---	fld.y	---	1196	--	r19	f22	X	X	-	-	-	-	-	-	-	-	-	X	-	X	-	S
---	---	fmul.p	---	1196	df16	df18	df20	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	S
adds	fsub.p	adds	fld.y	1197	--	r28	r29	X	X	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	fmul.p	---	1197	df16	df18	df20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	adds	1198	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	fmul.p	1198	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
st.x	adds	fsub.p	---	1199	df22	df20	df24	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	1199	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fst.y	st.x	adds	---	1200	--	r29	r0	X	X	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	fsub.p	---	1200	df22	df20	df24	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S

Figura 5.21. - Trecho do relatório completo após uma modificação para solucionar os conflitos de recursos existentes no *loop* de pivotação, para o problema da solução de sistemas lineares (continuação).

A otimização feita no primeiro mínimo fornece, desde a execução da operação de addu 40, r0, r16 até a execução da operação de fld r0(r19), r26, um ganho de quatro ciclos de relógio.

Totalizando as otimizações, o *speedup* resultante é de:

$$speedup = 1669/1610 = 1.04$$

Podemos observar, através do programa apresentado no apêndice A (A.4), uma série de instruções como comentário ao longo do programa. Nos trechos de programa apresentados nesta seção, os comentários foram todos omitidos. Isto porque estas instruções constituem código redundante gerado pelo compilador. Se avaliarmos o *speedup* a partir do código gerado pelo compilador, teremos:

$$speedup = 1954/1610 = 1.21$$

5.2.4. Operações com matrizes

Este experimento é subdividido em dois experimentos. O primeiro, consiste apenas em efetuar a multiplicação de duas matrizes A e B, pequenas o suficiente para serem completamente armazenadas na *cache* de dados, sem que haja qualquer reposição da linha da *cache*.

O segundo caso consiste de algumas operações sobre matrizes de tamanho 200x8. Este experimento objetiva mostrar o ganho obtido quando utiliza-se instruções de *load* de ponto-flutuante *pipelined* (pfld), instução esta não encontrada em códigos gerados pelo compilador.

Os programas em linguagem C para ambos os experimentos são encontrados no apêndice A.

1. Multiplicação de matrizes

Este experimento consiste apenas na multiplicação de duas matrizes A e B e seu armazenamento em outra matriz C. O código em *assembly* correspondente ao programa está ilustrado na figura 5.22.

```
      .file  "matrixpd.c"
// PGC Rel 2.1 -opt 2
      .data
      .align  16
stackarea   :  .long [400]10       // data area to be used as a stack
stackend    :  .long 0
framearea   :  .long [400]20       // data area to be used as a frame
register_dump:  .long [20]10

      .text
      .align  8

_L00TEXT:
      .text; .align  4

//Stack allocation: Autos: Regsave:

_start::
      // stack initialization
      mov    stackend, sp
      trap   r5, r5, r0 // turn on the detailed report generation
      // Call the user program
      call   _main
      nop

      trap   r6, r6, r0 // turn off the detailed report generation
      // Signal that the simulator MUST halt
      trap   0,r0,r0
      .text
      .globl  _main
      .align  8
_main:
.a1 = 0
.f1 = 32
      orh h%.STACK+.f1-16, r0, r28
      or l%.STACK+.f1-16, r28, r28
      st.l r4, -8(r28)
// lineno: 0
      orh h%_xgProdValue, r0, r31
      or l%_xgProdValue, r31, r4
      adds -1, r0, r18
// lineno: 51
      orh h%_axgBMatrix, r0, r31
      or l%_axgBMatrix, r31, r20
      orh h%_axgAMatrix, r0, r31
      or l%_axgAMatrix, r31, r21
      adds 7, r0, r19
      bla r18, r19,.B84
      pfmul.dd f0, f0, f0
```

Figura 5.22. - Código em *assembly* do programa para calcular o produto de matrizes.

```
// lineno: 56
.B84:  //.M0000
       fld.d r0(r21), f16
.DB.B8484:
       fld.d r0(r20), f18
       fld.d r0(r4), f22
       addu 8, r20, r29
       fmul.dd f18, f16, f20
       addu 8, r29, r20
       addu 8, r21, r21
       fadd.dd f20, f22, f24
       bla r18, r19, .B84
       fst.d f24, r0(r4)
// lineno: 0
// lineno: 60
       ld.l -8(r28), r4
       bri r1
       nop
```

Figura 5.22. - Código em *assembly* do programa para calcular o produto de matrizes (continuação).

Da mesma forma que nos outros experimentos, é analisada a percentagem de utilização das unidades central e de ponto-flutuante ao longo da execução do programa. O gráfico apresentado na figura 5.23 ilustra esta utilização.

Podemos observar vários pontos nos quais a utilização da unidade central decai à medida que a utilização da unidade de ponto-flutuante cresce, e vice-versa, caracterizando ausência de paralelismo entre as duas unidades. Localizando estes pontos no relatótio, figura 5.24, algumas observações podem ser feitas:

- Os dois flds seguidos nos períodos de relógio 28 e 35 atrasam o processamento quando ocorre falta na *cache* de dados, pois esta fica ocupada com a busca da linha.

- A instrução fadd (período de relógio 55) fica aguardando liberação de recursos para ser executada.

- A instrução fst (período de relógio 59) fica aguardando a liberação do registrador f24 (da fadd) para ser executada.

Efetuando uma monitoração das instruções é possível ocupar estes períodos de espera com a execução de outras instruções que não precisem dos recursos bloqueados. A

figura 5.25 apresenta o código com algumas otimizações enquanto a figura 5.26 ilustra a melhor utilização das unidades funcionais ao longo da execução do programa modificado. A figura 5.27 contém os trechos do relatório que mostram a execução da instrução durante os ciclos de espera antes existentes.

O *speedup* decorrente desta otimização é:

$$speedup = 212/186 = 1.14$$

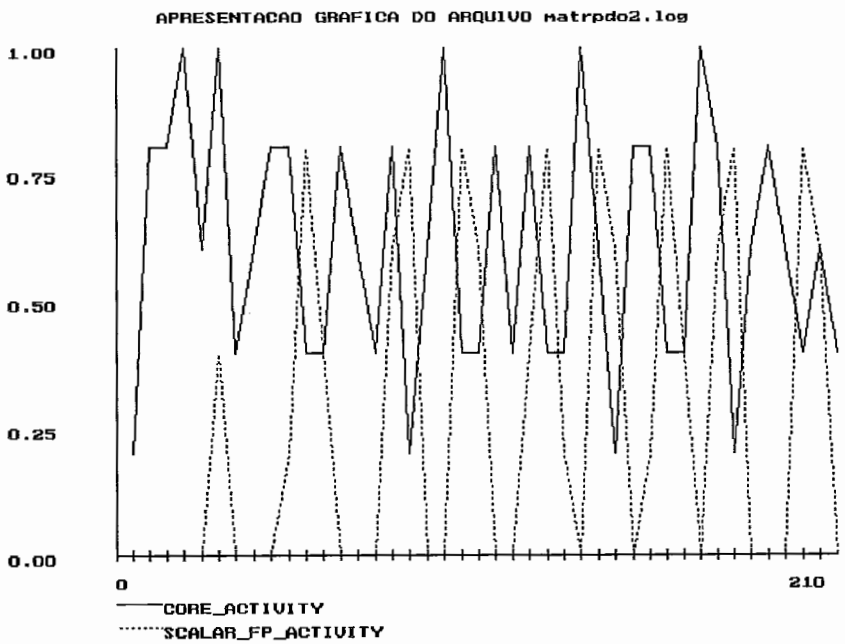


Figura 5.23. - Percentagem de utilização das unidades central e de ponto-flutuante durante a execução do programa de multiplicação de matrizes.

INSTRUCTION PIPELINE				CLK	SRC1	SRC2	DEST	CORE	FP	FPADD			FPMPLY			GRPH	ICH	ICM	DCH	DCM	CNA	MA	OPM
FETCH	DECODE	EXEC.	WRITE	-	-	-	-	-		S1	S2	S3	S1	S2	S3								
pfmul.p	bla	adds	or	25	--	r0	r19	X	-	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	25	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fld.y	pfmul.p	bla	adds	26	r18	r19	--	X	-	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	26	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fld.y	fld.y	pfmul.p	---	27	df0	df0	df0	X	X	-	-	-	X	-	-	-	X	-	-	-	-	-	S
---	---	bla	---	27	r18	r19	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fld.y	fld.y	fld.y	---	28	r0	r21	f16	X	X	-	-	-	X	-	-	-	X	-	-	X	X	X	S
---	---	pfmul.p	bla	28	df0	df0	df0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	fld.y	---	29	r0	r21	f16	X	-	-	-	-	X	-	-	-	-	-	-	X	X	X	S
---	---	---	pfmul.p	29	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	fld.y	---	30	r0	r21	f16	X	-	-	-	-	X	-	-	-	-	-	-	X	X	X	S
---	---	---	---	30	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	fld.y	---	31	r0	r21	f16	X	-	-	-	-	X	-	-	-	-	-	-	X	X	X	S
---	---	---	---	31	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	fld.y	32	--	--	--	-	-	-	-	-	X	-	-	-	-	-	-	-	X	X	S
---	---	---	---	32	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	---	33	--	--	--	-	-	-	-	-	X	-	-	-	-	-	-	-	-	X	S
---	---	---	---	33	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	---	34	--	--	--	-	-	-	-	-	X	-	-	-	-	-	-	-	-	X	S
---	---	---	---	34	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	fld.y	fld.y	---	35	r0	r20	f18	X	-	-	-	-	X	-	-	-	-	X	-	X	X	X	S
---	---	---	---	35	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	fld.y	---	36	r0	r20	f18	X	-	-	-	-	X	-	-	-	-	X	-	X	X	X	S
---	---	---	---	36	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S

Figura 5.24.- Relatório referente à utilização das unidades central e de ponto-flutuante, durante a execução do programa para multiplicação de matrizes.

INSTRUCTION PIPELINE				CLK	SRC1	SRC2	DEST	CORE	FP	FPADD			FPMPLY			GRPH	ICH	ICM	DCH	DCM	CHA	HA	OPM
FETCH	DECODE	EXEC.	WRITE	-	-	-	-	-		S1	S2	S3	S1	S2	S3								
addu	fmul.p	addu	fld.y	49	--	r20	r29	X	-	-	-	X	-	-	-	X	-	-	-	X	X	S
---	---	---	---	49	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
addu	addu	fmul.p	addu	50	df18	df16	df20	-	X	-	-	-	-	-	-	X	-	-	-	X	-	S
---	---	---	---	50	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fadd.p	addu	addu	---	51	--	r29	r20	X	X	-	-	-	-	-	-	X	-	-	-	X	-	S
---	---	fmul.p	---	51	df18	df16	df20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
bla	fadd.p	addu	addu	52	--	r21	r21	X	X	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	fmul.p	---	52	df18	df16	df20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	addu	53	--	--	--	-	X	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	fmul.p	---	53	df18	df16	df20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	---	54	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	fmul.p	54	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fst.y	bla	fadd.p	---	55	df20	df22	df24	-	X	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	55	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fld.y	fst.y	bla	---	56	r18	r19	--	X	X	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	fadd.p	---	56	df20	df22	df24	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	bla	57	--	--	--	-	X	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	fadd.p	---	57	df20	df22	df24	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	---	58	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	fadd.p	58	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fld.y	fld.y	fst.y	---	59	r0	r4	f24	X	-	-	-	-	-	-	-	X	-	X	-	X	-	S
---	---	---	---	59	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	fst.y	60	--	--	--	-	-	-	-	-	-	-	-	-	-	-	X	-	X	S
---	---	---	---	60	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fld.y	fld.y	fld.y	---	61	r0	r21	f16	X	-	-	-	-	-	-	-	X	-	-	X	X	X	S
---	---	---	---	61	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	fld.y	---	62	r0	r21	f16	X	-	-	-	-	-	-	-	-	-	-	X	X	X	S
---	---	---	---	62	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S

Figura 5.24.- Relatório referente à utilização das unidades central e de ponto-flutuante, durante a execução do programa para multiplicação de matrizes (continuação).

```
        .file  "matrixpd.c"
// PGC Rel 2.1 -opt 2
        .data
        .align  16
stackarea   :  .long [400]10        // data area to be used as a stack
stackend    :  .long 0
framearea   :  .long [400]20        // data area to be used as a frame
register_dump:  .long [20]10

        .text
        .align  8

_L00TEXT:
        .text; .align  4

//Stack allocation: Autos: Regsave:

_start::
        // stack initialization
        mov     stackend, sp
        trap    r5, r5, r0 // turn on the detailed report generation
        // Call the user program
        call    _main
        nop

        trap    r6, r6, r0 // turn off the detailed report generation
        // Signal that the simulator MUST halt
        trap    0,r0,r0
        .text
        .globl  _main
        .align  8
_main:
.a1 = 0
.f1 = 32
        orh h%.STACK+.f1-16, r0, r28
        or l%.STACK+.f1-16, r28, r28
        st.l r4, -8(r28)
orh h%_xgProdValue, r0, r31
        or l%_xgProdValue, r31, r4
        adds -1, r0, r18
orh h%_axgBMatrix, r0, r31
        or l%_axgBMatrix, r31, r20
        orh h%_axgAMatrix, r0, r31
        or l%_axgAMatrix, r31, r21
        adds 7, r0, r19
        bla r18, r19,.B84
        pfmul.dd f0, f0, f0
.B84:   //.M0000
        fld.d r0(r21), f16
```

Figura 5.25. - Código do programa para multiplicação de matrizes após algumas otimizações.

```
.DB.B8484:
        addu 8, r20, r29
        fld.d r0(r20), f18
        fmul.dd f18, f16, f20
        fld.d r0(r4), f22
        addu 8, r29, r20
        fadd.dd f20, f22, f24
        addu 8, r21, r21
        bla r18, r19, .B84
        fst.d f24, r0(r4)
// lineno: 0
// lineno: 60
        ld.l -8(r28), r4
        bri r1
        nop
```

Figura 5.25. - Código do programa para multiplicação de matrizes após algumas otimizações (continuação).

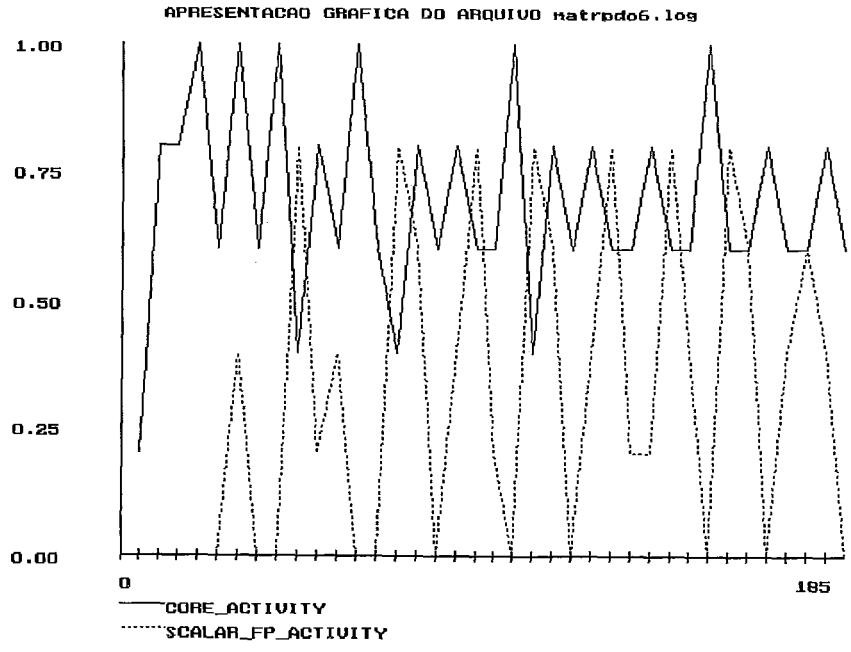


Figura 5.26. - Percentagem de utilização das unidades central e de ponto-flutuante durante a execução do programa de multiplicação de matrizes, após as otimizações.

RELATORIO DE SAIDA DO SIMULADOR pagina 2

INSTRUCTION PIPELINE				CLK	SRC1	SRC2	DEST	CORE	FP	FPADD			FPMPLY			GRPH	ICH	ICM	DCH	DCM	CHA	MA	OPM
FETCH	DECODE	EXEC.	WRITE	-	-	-	-	-		S1	S2	S3	S1	S2	S3								
pfmul.p	bla	adds	or	25	--	r0	r19	X	-	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	25	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fld.y	pfmul.p	bla	adds	26	r18	r19	--	X	-	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	26	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
addu	fld.y	pfmul.p	---	27	df0	df0	df0	X	X	-	-	-	X	-	-	-	X	-	-	-	-	-	S
---	---	bla	---	27	r18	r19	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fld.y	addu	fld.y	---	28	r0	r21	f16	X	X	-	-	-	X	-	-	-	X	-	-	X	X	X	S
---	---	pfmul.p	bla	28	df0	df0	df0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	fld.y	---	29	r0	r21	f16	X	-	-	-	-	X	-	-	-	-	-	-	X	X	X	S
---	---	---	pfmul.p	29	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	fld.y	---	30	r0	r21	f16	X	-	-	-	-	X	-	-	-	-	-	-	X	X	X	S
---	---	---	---	30	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	fld.y	---	31	r0	r21	f16	X	-	-	-	-	X	-	-	-	-	-	-	X	X	X	S
---	---	---	---	31	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	fld.y	addu	fld.y	32	--	r20	r29	X	-	-	-	-	X	-	-	-	-	-	X	-	-	X	S
---	---	---	---	32	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	addu	33	--	--	--	-	-	-	-	-	X	-	-	-	-	-	X	-	-	X	S
---	---	---	---	33	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	---	34	--	--	--	-	-	-	-	-	X	-	-	-	-	-	X	-	-	X	S
---	---	---	---	34	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fmul.p	---	fld.y	---	35	r0	r20	f18	X	-	-	-	-	X	-	-	-	-	-	X	-	-	X	S
---	---	---	---	35	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fld.y	fmul.p	fld.y	---	36	r0	r20	f18	X	-	-	-	-	X	-	-	-	X	-	-	X	X	X	S
---	---	---	---	36	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S

Figura 5.27.- Relatório referente à utilização das unidades central e de ponto-flutuante, durante a execução do programa para multiplicação de matrizes, feitas algumas otimizações.

INSTRUCTION PIPELINE				CLK	SRC1	SRC2	DEST	CORE	FP	FPADD			FPMPLY			GRPH	ICH	ICM	DCH	DCM	CNA	HA	OPM
FETCH	DECODE	EXEC.	WRITE	-	-	-	-	-		S1	S2	S3	S1	S2	S3								
fmul.p	fld.y	addu	fld.y	81	--	r20	r29	X	-	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	81	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fld.y	fmul.p	fld.y	addu	82	r0	r20	f18	X	-	-	-	-	-	-	-	-	X	-	X	-	X	-	S
---	---	---	---	82	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	fld.y	---	83	r0	r20	f18	X	-	-	-	-	-	-	-	-	-	-	X	-	X	-	S
---	---	---	---	83	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
addu	fld.y	fmul.p	fld.y	84	df18	df16	df20	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	84	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fadd.p	addu	fld.y	---	85	r0	r4	f22	X	X	-	-	-	-	-	-	-	X	-	X	-	X	-	S
---	---	fmul.p	---	85	df18	df16	df20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	fld.y	---	86	r0	r4	f22	X	X	-	-	-	-	-	-	-	-	-	X	-	X	-	S
---	---	fmul.p	---	86	df18	df16	df20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
addu	fadd.p	addu	fld.y	87	--	r29	r20	X	X	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	fmul.p	---	87	df18	df16	df20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	addu	88	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	fmul.p	88	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
bla	addu	fadd.p	---	89	df20	df22	df24	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	---	---	89	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fst.y	bla	addu	---	90	--	r21	r21	X	X	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	fadd.p	---	90	df20	df22	df24	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
fld.y	fst.y	bla	addu	91	r18	r19	--	X	X	-	-	-	-	-	-	-	X	-	-	-	-	-	S
---	---	fadd.p	---	91	df20	df22	df24	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	bla	92	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	fadd.p	92	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
addu	fld.y	fst.y	---	93	r0	r4	f24	X	-	-	-	-	-	-	-	-	X	-	X	-	X	-	S
---	---	---	---	93	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S
---	---	---	fst.y	94	--	--	--	-	-	-	-	-	-	-	-	-	-	-	X	-	X	-	S
---	---	---	---	94	--	--	--	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	S

Figura 5.27.- Relatório referente à utilização das unidades central e de ponto-flutuante, durante a execução do programa para multiplicação de matrizes, feitas algumas otimizações (continuação).

2. Operações sobre matrizes 200x8

O objetivo deste experimento é analisar o baixo desempenho existente quando ocorrem substituições de linhas na memória *cache* e não são usadas instruções pflds.

A figura 5.28 apresenta o programa gerado pelo compilador. Quando ocorre uma falta na *cache* de dados e é necessário subtituir uma das linhas de um conjunto, a antiga linha é escrita em dois *buffers* de escrita e a linha é buscada. Como o conteúdo dos buffers será escrito na memória logo após a otenção da linha, o barramento externo fica ocupado até que a antiga linha seja escrita. Se em seguida existir um fld , este terá que esperar a liberação do barramento. Desta forma, sua execução é extendida em vários ciclos de relógio, como mostra o trecho do relatório completo apresentado na figura 5.29.

A figura 5.30 apresenta o código do programa utilizando instruções pfld. Neste caso, o dado é sempre buscado na memória, em acessos pipelined. Esta modificação garante uma boa melhoria no desempenho do programa:

$$speedup = 39868/30577 = 1.30$$

```
.file  "matrixgg.c"
       .data
       .align  16
stackarea   :  .long [400]10       // data area to be used as a stack
stackend    :  .long 0
framearea   :  .long [400]20       // data area to be used as a frame
register_dump:  .long [20]10

       .text
       .align  8

_L00TEXT:
       .text; .align  4

//Stack allocation: Autos: Regsave:

_start::
       // stack initialization
       mov     stackend, sp

       // Call the user program
       call    _main
       nop

       // Signal that the simulator MUST halt
       trap    0,r0,r0
```

Figura 2.28. - Código gerado pelo compilador para o programa de matrizes 200x8.

```
// PGC Rel 2.1 -opt 4
        .text
        .globl  _main
        .align  8
_main:
.a1 = 0
.f1 = 48
        orh h%.STACK+.f1-16, r0, r28
        or l%.STACK+.f1-16, r28, r28
        st.l r4, -24(r28)
        st.l r5, -20(r28)
        st.l r6, -16(r28)
        st.l r7, -12(r28)
        st.l r8, -8(r28)
        st.l r9, -4(r28)
// lineno: 0
        adds -1, r0, r18
// lineno: 621
        mov r0, r6
//      mov r0, r6
        orh h%_axgCMatrix+-8, r0, r31
        or l%_axgCMatrix+-8, r31, r7
        orh h%_axgBMatrix+-8, r0, r31
        or l%_axgBMatrix+-8, r31, r8
        orh h%_axgAMatrix+-8, r0, r31
        or l%_axgAMatrix+-8, r31, r9
// lineno: 622
.B174:  //.M0001
//      mov r7, r4
.DB.B174174:
        adds  -75, r6, r0
        bc  TRAP_OFF
        trap r5, r5, r0      //turn on the report generation
TRAP_OFF:
        mov r7, r4
        adds 7, r0, r19
        mov r8, r20
        mov r9, r21
        mov r7, r5
        bla r18, r19,.B163
        pfmul.dd f0, f0, f0
// lineno: 624
.B163:  //.M0000
.align 8
        d.pfadd.dd  f0,  f0,  f0
            fld.d 8(r20)++, f16
.DB.B163163:
        d.fnop
            fld.d 8(r21)++, f20
        d.pfadd.dd f16, f16, f0
            fld.d 8(r5)++, f26
```

Figura 2.28. - Código gerado pelo compilador para o programa de matrizes 200x8 (continuação).

```
    d.pfadd.dd f20, f20, f0
        nop
    d.pfadd.dd f0, f0, f0
        nop
    d.pfadd.dd f0, f0, f16
        fst.d f16, r0(r20)
    d.pfadd.dd f20, f16, f20
        fst.d f20, r0(r21)
    d.pfadd.dd f0, f0, f0
        bla r18, r19, .PL1002
    d.pfadd.dd f0, f0, f0
        nop

.PL1002:
    d.pfadd.dd f30, f26, f30
        fld.d 8(r20)++, f16
    d.pfadd.dd f0, f0, f0
        fld.d 8(r21)++, f20
    d.pfadd.dd f0, f0, f0
        fld.d 8(r5)++, f26
    d.pfadd.dd f16, f16, f30
        fst.d f30, 8(r4)++
    d.pfadd.dd f20, f20, f0
        nop
    d.pfadd.dd f0, f0, f0
        nop
    d.pfadd.dd f0, f0, f16
        fst.d f16, r0(r20)
    d.pfadd.dd f20, f16, f20
        fst.d f20, r0(r21)
    d.fnop
        trap r6, r6, r0     //turn off the report generation
    d.pfadd.dd f0, f0, f0
        bla r18, r19, .PL1002
    d.pfadd.dd f0, f0, f0
        nop

.PL1003:
    d.pfadd.dd f30, f26, f30
        nop
    d.pfadd.dd f0, f0, f0
        nop
    d.pfadd.dd f0, f0, f0
        nop
    d.pfadd.dd f0, f0, f30
        fst.d f30, 8(r4)++
    d.fnop
        bla r18, r19, .PL1003
    d.fnop
        nop
```

Figura 2.28. - Código gerado pelo compilador para o programa de matrizes 200x8 (continuação).

```
fnop
        nop
    fnop
        nop
// lineno: 630
    adds 1, r6, r6
    addu 64, r8, r8
    addu 64, r9, r9
    addu 64, r7, r7
    adds -200, r6, r0
    bc.t .DB.B174174
    mov r7, r4
// lineno: 0
// lineno: 631
    ld.l -24(r28), r4
    ld.l -20(r28), r5
    ld.l -16(r28), r6
    ld.l -12(r28), r7
    ld.l -8(r28), r8
    ld.l -4(r28), r9
    bri r1
    nop
```

Figura 2.28. - Código gerado pelo compilador para o programa de matrizes 200x8 (continuação).

INSTRUCTION PIPELINE				CLK	SRC1	SRC2	DEST	CORE	FP	FPADD			FPMPLY			GRPH	ICH	ICM	DCH	DCM	CNA	MA	OPM
FETCH	DECODE	EXEC.	WRITE	-	-	-	-	-		S1	S2	S3	S1	S2	S3								
---	---	---	---	13648	--	--	--	-	-	X	X	X	X	X	X	-	-	-	-	-	X	X	D
---	---	---	---	13648	--	--	--	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	---	---	13649	--	--	--	-	-	X	X	X	X	X	X	-	-	-	-	-	X	X	D
---	---	---	---	13649	--	--	--	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
fst.y	fld.y	fld.y	---	13650	--	r21	f20	X	X	X	X	X	X	X	X	-	X	-	-	X	X	X	D
pfadd.p	pfadd.p	pfadd.p	---	13650	df0	df0	df0	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	fld.y	---	13651	--	r21	f20	X	-	X	X	X	X	X	X	-	-	-	-	X	X	X	D
---	---	pfadd.p	---	13651	df0	df0	df0	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	fld.y	---	13652	--	r21	f20	X	-	X	X	X	X	X	X	-	-	-	-	X	X	X	D
---	---	pfadd.p	---	13652	df0	df0	df0	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	fld.y	---	13653	--	r21	f20	X	-	X	X	X	X	X	X	-	-	-	-	X	X	X	D
---	---	pfadd.p	---	13653	df0	df0	df0	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	fld.y	---	13654	--	r21	f20	X	-	X	X	X	X	X	X	-	-	-	-	X	X	X	D
---	---	pfadd.p	---	13654	df0	df0	df0	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	fld.y	---	13655	--	r21	f20	X	-	X	X	X	X	X	X	-	-	-	-	X	X	X	D
---	---	pfadd.p	---	13655	df0	df0	df0	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	fld.y	---	13656	--	r21	f20	X	-	X	X	X	X	X	X	-	-	-	-	X	X	X	D
---	---	pfadd.p	---	13656	df0	df0	df0	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	fld.y	---	13657	--	r21	f20	X	-	X	X	X	X	X	X	-	-	-	-	X	X	X	D
---	---	pfadd.p	---	13657	df0	df0	df0	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	fld.y	---	13658	--	r21	f20	X	-	X	X	X	X	X	X	-	-	-	-	X	X	X	D
---	---	pfadd.p	---	13658	df0	df0	df0	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	fld.y	---	13659	--	r21	f20	X	-	X	X	X	X	X	X	-	-	-	-	X	X	X	D
---	---	pfadd.p	---	13659	df0	df0	df0	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	fld.y	---	13660	--	r21	f20	X	-	X	X	X	X	X	X	-	-	-	-	X	X	X	D
---	---	pfadd.p	---	13660	df0	df0	df0	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	fld.y	---	13661	--	r21	f20	X	-	X	X	X	X	X	X	-	-	-	-	X	X	X	D
---	---	pfadd.p	---	13661	df0	df0	df0	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D

Figura 5.29.- Relatório referente à utilização das unidades central e de ponto-flutuante, durante a execução do programa de operações sobre matrizes 200x8.

INSTRUCTION PIPELINE				CLK	SRC1	SRC2	DEST	CORE	FP	FPADD			FPMPLY			GRPH	ICH	ICM	DCH	DCM	CNA	MA	OPM
FETCH	DECODE	EXEC.	WRITE	-	-	-	-	-		S1	S2	S3	S1	S2	S3								
---	---	---	fld.y	13662	--	--	--	-	-	X	X	X	X	X	X	-	-	-	-	-	X	X	D
---	---	---	pfadd.p	13662	--	--	--	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	---	---	13663	--	--	--	-	-	X	X	X	X	X	X	-	-	-	-	-	X	X	D
---	---	---	---	13663	--	--	--	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	---	---	13664	--	--	--	-	-	X	X	X	X	X	X	-	-	-	-	-	X	X	D
---	---	---	---	13664	--	--	--	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
shl	fst.y	fld.y	---	13665	--	r5	f26	X	X	X	X	X	X	X	X	-	X	-	-	X	X	X	D
pfadd.p	pfadd.p	pfadd.p	---	13665	df0	df0	df0	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	fld.y	---	13666	--	r5	f26	X	-	X	X	X	X	X	X	-	-	-	-	X	X	X	D
---	---	pfadd.p	---	13666	df0	df0	df0	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	fld.y	---	13667	--	r5	f26	X	-	X	X	X	X	X	X	-	-	-	-	X	X	X	D
---	---	pfadd.p	---	13667	df0	df0	df0	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	fld.y	---	13668	--	r5	f26	X	-	X	X	X	X	X	X	-	-	-	-	X	X	X	D
---	---	pfadd.p	---	13668	df0	df0	df0	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	fld.y	---	13669	--	r5	f26	X	-	X	X	X	X	X	X	-	-	-	-	X	X	X	D
---	---	pfadd.p	---	13669	df0	df0	df0	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	fld.y	---	13670	--	r5	f26	X	-	X	X	X	X	X	X	-	-	-	-	X	X	X	D
---	---	pfadd.p	---	13670	df0	df0	df0	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	fld.y	---	13671	--	r5	f26	X	-	X	X	X	X	X	X	-	-	-	-	X	X	X	D
---	---	pfadd.p	---	13671	df0	df0	df0	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	fld.y	---	13672	--	r5	f26	X	-	X	X	X	X	X	X	-	-	-	-	X	X	X	D
---	---	pfadd.p	---	13672	df0	df0	df0	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	fld.y	---	13673	--	r5	f26	X	-	X	X	X	X	X	X	-	-	-	-	X	X	X	D
---	---	pfadd.p	---	13673	df0	df0	df0	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	fld.y	---	13674	--	r5	f26	X	-	X	X	X	X	X	X	-	-	-	-	X	X	X	D
---	---	pfadd.p	---	13674	df0	df0	df0	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D
---	---	fld.y	---	13675	--	r5	f26	X	-	X	X	X	X	X	X	-	-	-	-	X	X	X	D
---	---	pfadd.p	---	13675	df0	df0	df0	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	D

Figura 5.29.- Relatório referente à utilização das unidades central e de ponto-flutuante, durante a execução do programa de operações sobre matrizes 200x8 (continuação).

```
.file  "matrixgg.c"

        .data
        .align  16
stackarea   :  .long [400]10        // data area to be used as a stack
stackend    :  .long 0
framearea   :  .long [400]20        // data area to be used as a frame
register_dump:  .long [20]10

        .text
        .align  8

_L00TEXT:
        .text; .align  4

//Stack allocation: Autos: Regsave:

_start::
        // stack initialization
        mov     stackend, sp

        // Call the user program
        call    _main
        nop

        // Signal that the simulator MUST halt
        trap    0,r0,r0

// PGC Rel 2.1 -opt 4
        .text
        .globl  _main
        .align  8
_main:
.a1 = 0
.f1 = 48
        orh h%.STACK+.f1-16, r0, r28
        or l%.STACK+.f1-16, r28, r28
        st.l r4, -24(r28)
        st.l r5, -20(r28)
        st.l r6, -16(r28)
        st.l r7, -12(r28)
        st.l r8, -8(r28)
        st.l r9, -4(r28)

// lineno: 0
        adds -1, r0, r18
// lineno: 621
        mov r0, r6
//      mov r0, r6
        orh h%_axgCMatrix+-8, r0, r31
        or l%_axgCMatrix+-8, r31, r7
        orh h%_axgBMatrix+-8, r0, r31
```

Figura 2.30. - Código para o programa de matrizes 200x8, utilizando instruções pfld.

```
        or l%_axgBMatrix+-8, r31, r8
        orh h%_axgAMatrix+-8, r0, r31
        or l%_axgAMatrix+-8, r31, r9
.B174: //.M0001
//      mov r7, r4
.DB.B174174:
        adds -74, r6, r0
        bc   TRAP_OFF
        trap r5, r5, r0     //turn on the report generation
TRAP_OFF:
        mov r7, r4
        adds 7, r0, r19
        mov r8, r20
        mov r9, r21
        mov r7, r5
        pfld.d 8(r20)++, f0
        pfld.d 8(r21)++, f0
        pfld.d 8(r5)++, f0
        bla r18, r19,.B163
        pfmul.dd f0, f0, f0
// lineno: 624
.B163: //.M0000
.align 8
        d.pfadd.dd  f0,  f0,  f0
                pfld.d 8(r20)++, f16
.DB.B163163:
        d.fnop
                pfld.d 8(r21)++, f20
        d.pfadd.dd f16, f16, f0
                pfld.d 8(r5)++, f26
        d.pfadd.dd f20, f20, f0
                nop
        d.pfadd.dd f0, f0, f0
                nop
        d.pfadd.dd f0, f0, f16
                fst.d f16, r0(r20)
        d.pfadd.dd f20, f16, f20
                fst.d f20, r0(r21)
        d.pfadd.dd f0, f0, f0
                bla r18, r19, .PL1002
        d.pfadd.dd f0, f0, f0
                nop
.PL1002:
        d.pfadd.dd f30, f26, f30
                pfld.d 8(r20)++, f16
        d.pfadd.dd f0, f0, f0
                pfld.d 8(r21)++, f20
        d.pfadd.dd f0, f0, f0
                pfld.d 8(r5)++, f26
        d.pfadd.dd f16, f16, f30
                fst.d f30, 8(r4)++
```

Figura 2.30. - Código para o programa de matrizes 200x8, utilizando instruções pfld (continuação).

```
d.pfadd.dd f20, f20, f0
        nop
    d.pfadd.dd f0, f0, f0
        nop
    d.pfadd.dd f0, f0, f16
        fst.d f16, r0(r20)
    d.pfadd.dd f20, f16, f20
        fst.d f20, r0(r21)
    d.fnop
        trap r6, r6, r0     //turn off the report generation
    d.pfadd.dd f0, f0, f0
        bla r18, r19, .PL1002
    d.pfadd.dd f0, f0, f0
        nop

.PL1003:
    d.pfadd.dd f30, f26, f30
        nop
    d.pfadd.dd f0, f0, f0
        nop
    d.pfadd.dd f0, f0, f0
        nop
    d.pfadd.dd f0, f0, f30
        fst.d f30, 8(r4)++
    d.fnop
        bla r18, r19, .PL1003
    d.fnop
        nop
    fnop
        nop
    fnop
        nop
// lineno: 630
    adds 1, r6, r6
    addu 64, r8, r8
    addu 64, r9, r9
    addu 64, r7, r7
    adds -200, r6, r0
    bc.t .DB.B174174
    mov r7, r4
// lineno: 0
// lineno: 631
    ld.l -24(r28), r4
    ld.l -20(r28), r5
    ld.l -16(r28), r6
    ld.l -12(r28), r7
    ld.l -8(r28), r8
    ld.l -4(r28), r9
    bri r1
    nop
```

Figura 2.30. - Código para o programa de matrizes 200x8, utilizando instruções pfld (continuação).

5.3 - Análise de resultados.

O primeiro ponto a ser ressaltado está relacionado ao código gerado pelos compiladores para o processador i860.

Atualmente dispomos de um compilador da Intel que gera código bastante otimizado e procura explorar os recursos oferecidos pelo processador. Todavia, programar uma arquitetura como a do i860 não é tarefa fácil e, portanto, o próprio código gerado pelo compilador ainda é passível de otimização.

Isto pôde ser presentemente verificado através dos resultados do *speedup* para cada caso analisado. Embora o aumento da taxa de aceleração não seja aparentemente muito grande, é importante ressaltar que já apresentam alta eficiência. Mais ainda, não introduzimos qualquer modificação na arquitetura envolvida, tratando-se apenas de uma melhor utilização dos recursos que a arquitetura oferece.

A utilização de uma ferramenta como o simulador SIM860, juntamente com o programa SHOWREPO, introduz uma metodologia que pode ser aplicada em qualquer problema. É claro que problemas que envolvam operações de ponto-flutuante têm melhores condições de serem otimizados.

Não há uma regra que diga que uma classe de problemas poderá ter um percentual de otimização x ou y. Cada caso deve ser analisado para que se possa aumentar o desempenho. Outro ponto interessante pode ser observado quando da implementação em linguagem de alto nível. Otimizações em programas escritos em linguagem de alto nível automaticamente são refletidas no código em *assembly*.

Capítulo 6

Conclusões

Este trabalho apresenta a implementação de um simulador para o processador superescalar RISC Intel i860.

A análise temporal fornecida pelo SIM860, principal diferença entre esta ferramenta e as fornecidas pela Intel, tornou bastante complexo o trabalho de projeto e desenvolvimento do simulador. Esta complexidade está vinculada à multiplicidade de recursos do i860 e ao controle da temporização de suas instruções.

A visão completa da utilização dos recursos do processador durante a execução de um programa, fornecida através do relatório gerado pelo simulador, em conjunto com os gráficos fornecidos pelo SHOWREPO, dão subsídios para uma metodologia de análise de desempenho de programas executados pelo i860. Os resultados do emprego desta metodologia são promissores e o escopo das aplicações analisadas deve ser ampliado.

Através desta metodologia pode-se identificar trechos de programas em que os códigos em *Assembly* não exploram adequadamente os recursos do i860 e, portanto, são passíveis de aumento de desempenho (otimização).

Em particular a otimização de códigos gerados por compiladores relacionados a alguns casos mostra que os estes códigos são ainda passíveis de aumento de desempenho, mesmo quando as opções de otimização dos compiladores são utilizadas. A análise estática feita por compiladores limita as possíveis otimizações, podendo levar a falsas otimizações e até mesmo a perda de desempenho.

O trabalho desenvolvido nesta tese contudo, não fica restrito apenas à melhoria de eficiência de códigos de programas.

O simulador pode ser também utilizado para finalidades acadêmicas. Sendo uma ferramenta que permite uma visão completa da execução de um programa numa arquitetura superescalar, pode ser usado para ilustrar o comportamento interno de tais arquiteturas ao longo da execução de um programa.

Embora não seja seu objetivo específico, sua utilização para identificar o que está ocorrendo quando um programa se perde é de grande valia. Isto porque o simulador interrompe a execução do programa quando é detectada uma situação de erro.

Além destas aplicações, o método aqui empregado para implementar o simulador do i860 pode ser usado na implementação de ferramentas para outros processadores superescalares. Como exemplo, citamos o processador Power PC [22], empregado na nova máquina que está sendo desenvolvida pelo Grupo de Computação Paralela do Departamento de Sistemas e Computação da COPPE/UFRJ.

Apêndice A

A.1. - Programa para implementar um filtro FIR (*finit impulse response*)

```
#define   SIGNAL_LENGTH  64
#define   FILTER_ORDER   8

double axgXSignal  = {
      0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,
      0.05, 0.1,  0.15, 0.19, 0.23, 0.27, 0.30, 0.33,
      0.490, 0.494, 0.496, 0.497, 0.498, 0.499, 0.4995, 0.5,
      0.5,  0.4995, 0.499, 0.498, 0.497, 0.496, 0.494, 0.490,
      0.33,  0.30,  0.27,  0.23,  0.19,  0.15,  0.1,  0.05,
      -0.05, -0.1,  -0.15, -0.19, -0.23, -0.27, -0.30, -0.33,
      -0.490, -0.494, -0.496, -0.497, -0.498, -0.499, -0.4995, -0.5,
      -0.5,  -0.4995, -0.499, -0.498, -0.497, -0.496, -0.494, -0.490,
      -0.33, -0.30, -0.27, -0.23, -0.19, -0.15, -0.1,  -0.05,
      0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0} ;


double axgYSignal  = {
      0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,
      0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,
      0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,
      0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,
      0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,
      0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,
      0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,
      0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,
      0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0} ;

double axgFilterParms  = {
      0.10,  0.30,  0.70,  0.90,  0.90,  0.70,  0.30,  0.10} ;

main()
{
  register double *ldlXNewElem      ,
            *ldlYNewElem     ,
            *ldlXScanPtr     ,
            *ldlFilterBaseAddr,
            *ldlFilterScanPtr ,
            xlAccumulator    ;
  register int   ilSignalElemCnt  ,
            ilFilterOrderCnt  ;
```

```
/* Initialize the pointers to the signals */
  ldlXNewElem = &(axgXSignal [FILTER_ORDER]) ;
  ldlYNewElem = axgYSignal ;
  ldlFilterBaseAddr = axgFilterParms ;

  ilSignalElemCnt = SIGNAL_LENGTH + FILTER_ORDER ;

  while (ilSignalElemCnt--) {
    /* Update the pointers to the current X window and to the */
    /* filter parameters                                      */
    ldlXScanPtr     = ldlXNewElem      ;
    ldlFilterScanPtr = ldlFilterBaseAddr ;
    ilFilterOrderCnt = FILTER_ORDER    ;
    xlAccumulator   = 0.0              ;
    while (ilFilterOrderCnt--) {
      xlAccumulator += *ldlXScanPtr * *ldlFilterScanPtr ;
      ldlXScanPtr     ++ ;
      ldlFilterScanPtr ++ ;
    } /* endwhile */
    *ldlYNewElem = xlAccumulator ;

    /* Prepare to evaluate the next element */
    ldlXNewElem ++ ;
    ldlYNewElem ++ ;
  } /* endwhile */
}
```

A.2. - Programa para obter o quinto termo da série de Fibonacci

```
main ()             /* Calcula o valor da serie de Fibonacci */

{
        int i;
        unsigned value, fib();

           value = fib(5);
}

unsigned fib(int x)  /* Funcao recursiva */
{
        if (x > 2)
                return (fib (x-1) + fib (x-2));
        else
                return (1);
}
```

A.3. - Programa para solução de um sistema linear pelo método de Eliminação Gaussiana

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#define X_DIMENSION 4
#define Y_DIMENSION 5
double MatrizA[X_DIMENSION][Y_DIMENSION] =
    {{-1.0, 3.0, 5.0, 2.0, 10.0},
     { 1.0, 9.0, 8.0, 4.0, 15.0},
     { 0.0, 1.0, 0.0, 1.0, 2.0},
     { 2.0, 1.0, 1.0, -1.0, -3.0}};
double MatrizX[Y_DIMENSION-1] = {0.0, 0.0, 0.0, 0.0};
double Y;
double Big;
double Pivot;
double Const;
int   Last;
int   Irev;
int   i, j, k;
int   NextR;
int   L;


void main()
{
  int   ailLineConvTable [X_DIMENSION] = {0,1,2,3},
          *lilTableConvPtr          ,
          ilTempLine                ,
          M = 4                     ,
          N = 5                     ;
  double xlAuxElem                  ,
          *lxlLineIPtr              ,
          *lxlLineJPtr              ,
          *lxlScanLineIPtr          ;


  Last = M - 1;
  for (i=0;i<Last;i++)
  {
        /* acha o maior termo restante na i-esima coluna para ser PIVOT */
        Big = 0;
        /* lilTableConvPtr = &(ailLineConvTable[i]) ; */
        for(j=i;j<M; j++)
        {
          xlAuxElem = MatrizA[ailLineConvTable[j]][i] ;
          xlAuxElem *= xlAuxElem ;
          if (xlAuxElem > Big) {
                Big = xlAuxElem;
                L = j;
          }
          /* lilTableConvPtr ++ ; */
        }
```

```
        if (Big == 0)
        {
          exit(1);
        }
        /* a linha L possui o maior termo */
        if ((i - L) != 0)
        {
          ilTempLine        = ailLineConvTable[i] ;
          ailLineConvTable[i] = ailLineConvTable[L] ;
          ailLineConvTable[L] = ilTempLine        ;
    }

    /* inicia a reducao pivotal */
        lxlLineIPtr = &(MatrizA[ailLineConvTable[i]][i]);
    Pivot = *lxlLineIPtr ;
    NextR = i + 1;

    /* para cada uma das linhas apos a i-esima ... */
    for (j=NextR;j<M;j++)
    {
      lxlLineJPtr = &(MatrizA[ailLineConvTable[j]][i]);
      Const = *lxlLineJPtr/Pivot ;
      *lxlLineJPtr = 0 ;
      lxlScanLineIPtr = lxlLineIPtr ;
      for (k=i+1; k<N; k++)
      {
        lxlLineJPtr ++ ;
        lxlScanLineIPtr ++ ;
        *lxlLineJPtr -= Const * *lxlScanLineIPtr ;
      }
    }
  }
  /* fim da reducao pivotal */
  /* efetue a substituicao */
  Irev = M-1;
  for (i=0 ; i<M ; i++, Irev --)
  {
        /* Irev e o indice de ordem reversa */
        /* Irev = M - 1 - i;              */
        /* prepara Y[Irev] */
        lxlLineIPtr = &(MatrizA[ailLineConvTable[Irev]][N-1]);
        Y = *lxlLineIPtr ;
        if (i != 0)
        {
          lxlLineJPtr = &(MatrizX[N-1]) ;
          lxlScanLineIPtr = lxlLineIPtr ;
          for (j=1; j<=i;j++)
          {
                lxlLineJPtr -- ;
                lxlScanLineIPtr -- ;
                Y -= *lxlLineJPtr * *lxlScanLineIPtr ;
          }
        }
        MatrizX[Irev] = Y/ lxlLineIPtr[Irev-(N-1)];
  }
}
```

A.4. - Programa para determinar a solução de um sistema linear pelo método de Eliminação Gaussiana em linguagem assembly.

Os comentários encontrados neste programa correspondem às otimizações feitas no programa gerado pelo compilador. Todas as instruções comentadas correspondem a código redundante.

```
        .file   "gaussbl.c"
        .data
        .align  16
stackarea   :  .long [400]10        // data area to be used as a stack
stackend    :  .long 0
framearea   :  .long [400]20        // data area to be used as a frame
register_dump:  .long [20]10
        .text
        .align  8
_L00TEXT:
        .text; .align 4
//Stack allocation: Autos: Regsave:

_start::
        // stack initialization
        mov     stackend, sp

        // Call the user program
        call    _main
        nop

        // Signal that the simulator MUST halt
        trap    0,r0,r0

// PGC Rel 2.1 -opt 4
        .text
        .globl  _main
        .align  8
_main:
.a1 = 0
.f1 = 96
        addu -(.a1+.f1), sp, sp
        st.l fp,(.f1-16)(sp)
        addu (.f1-16), sp, fp
        st.l r1, 4(fp)
        st.l r4, -72(fp)
        st.l r5, -68(fp)
        st.l r6, -64(fp)
        st.l r7, -60(fp)
        st.l r8, -56(fp)
        st.l r9, -52(fp)
        st.l r10, -48(fp)
        st.l r11, -44(fp)
        st.l r12, -40(fp)
        st.l r13, -36(fp)
        st.l r14, -32(fp)
        st.l r15, -28(fp)
// lineno: 0
```

```
        orh h%_MatrizA, r0, r31
        or l%_MatrizA, r31, r12
// lineno: 27
        adds 3, r0, r28
        orh ha%_Last, r0, r31
        st.l r28, l%_Last(r31)
        orh h%_i, r0, r31
        or l%_i, r31, r13
        st.l r0, 0(r13)
        adds 1, r0, r29
        st.l r29, -12(fp)
        adds 2, r0, r30
        st.l r30, -8(fp)
        st.l r28, -4(fp)
        orh h%_k, r0, r31
        or l%_k, r31, r4
        orh h%_Const, r0, r31
        or l%_Const, r31, r5
        st.l r0, -16(fp)
        adds -1, r0, r6
// lineno: 42
.B297:  //.M0005
        ld.l r0(r13), r28
.DB.B297297:
        orh ha%_j, r0, r31
        st.l r28, l%_j(r31)
        orh ha%_Big, r0, r31
        fst.d f0, l%_Big(r31)
        adds -4, r28, r0
        bnc .B224
        orh ha%_L, r0, r31
        ld.l l%_L(r31), r10
        subs 3, r28, r7
        fiadd.dd f0, f0, f10
        shl 2, r28, r29
        adds -16, fp, r30
        adds r29, r30, r11
        mov r28, r8
        shl 3, r28, r9
        bla r6, r7,.B294
        pfmul.dd f0, f0, f0
// lineno: 46
.B294:  //.M0002
        fld.l r0(r11), f16
.DB.B294294:
        adds 40, r0, r28
        ixfr r28, f18
        fmlow.dd f16, f18, f20
        fxfr f20, r29
        adds r9, r29, r30
        fld.d r12(r30), f22
        fmul.dd f22, f22, f8
        pfle.dd f8, f10, f0
        bnc .B226
        fiadd.dd f8, f0, f10
        mov r8, r10
// lineno: 53
.B226:  //.B0004
```

```
        adds 1, r8, r8
.DB.B226226:
        bla r6, r7, .B294
        addu 4, r11, r11
// lineno: 0
        orh ha%_j, r0, r31
        st.l r8, l%_j(r31)
        orh ha%_L, r0, r31
        st.l r10, l%_L(r31)
        orh ha%_Big, r0, r31
        fst.d f10, l%_Big(r31)
// lineno: 55
.B224:  //.B0003
        orh ha%_Big, r0, r31
.DB.B224224:
        fld.d l%_Big(r31), f16
        pfeq.dd f0, f16, f0
        bnc .B228
        bri r1
        adds 1, r0, r16
// lineno: 60
.B228:  //.B0005
        orh ha%_L, r0, r31
.DB.B228228:
        ld.l l%_L(r31), r28
        ld.l r0(r13), r29
        bte r28, r29, .B230
        adds -16, fp, r30
        shl 2, r29, r16
        ld.l r30(r16), r15
        shl 2, r28, r22
        ld.l r30(r22), r23
        adds r16, r30, r31
        st.l r23, 0(r31)
        adds r22, r30, r31
        st.l r15, 0(r31)
// lineno: 68
.B230:  //.B0006
        ld.l r0(r13), r28
.DB.B230230:
        adds -16, fp, r30
        shl 2, r28, r29
        fld.l r30(r29), f16
        adds 40, r0, r16
        ixfr r16, f18
        shl 3, r28, r23
        adds 1, r28, r25
        fmlow.dd f18, f16, f20
        orh h%_Pivot, r0, r31
        or l%_Pivot, r31, r26
        fxfr f20, r22
        adds r22, r23, r24
        adds r24, r12, r14
        fld.d r0(r14), f22
        orh ha%_NextR, r0, r31
        st.l r25, l%_NextR(r31)
        fst.d f22, 0(r26)
        orh h%_j, r0, r31
```

```
        or l%_j, r31, r27
        adds -4, r25, r0
        st.l r25, 0(r27)
        bnc .B233
        mov r27, r17
        mov r13, r21
        mov r26, r20
// lineno: 75
.B296:  //.M0004
        fld.d r0(r20), f16
.DB.B296296:
//        orh ha%.C00043, r0, r31
//        fld.d l%.C00043(r31), f22  f22=2
//        frcp.dd f16, f18
        frcp.dd f16, f24
//        fmul.dd f16, f18, f20      f20=1
        ld.l r0(r17), r28
        adds -16, fp, r30
        shl 2, r28, r29
//        fsub.dd f22, f20, f24      f24=1
        adds 40, r0, r16
        ld.l r0(r21), r23
//        fmul.dd f18, f24, f26      f26=1/A
        fld.l r30(r29), f18
        shl 3, r23, r24
        mov r14, r18
//        fmul.dd f16, f26, f28      f28=1
//        fsub.dd f22, f28, f30      f30=1
        ixfr r16, f28
//        fmul.dd f26, f30, f12      f12=1/A
//        fmul.dd f16, f12, f14      f14=1
//        fsub.dd f22, f14, f20      f20=1
//        fmul.dd f20, f12, f24      f24=1/A
        fmlow.dd f18, f28, f30
        ld.l r0(r21), r26
        fxfr f30, r22
        adds r24, r22, r25
        adds r25, r12, r19
        fld.d r0(r19), f26          //f26=Const
        fmul.dd f26, f24, f14
        adds 1, r26, r27
        adds -5, r27, r0
        fst.d f0, r0(r19)
        fst.d f14, r0(r5)
        st.l r27, 0(r4)
        bnc .B236
        mov r14, r18
// lineno: 81
.B284:  //.M0001
        fld.d 8(r18)++, f16
.DB.B284284:
        fld.d r0(r5), f18
        fld.d 8(r19)++, f22
        fmul.dd f16, f18, f20
        ld.l r0(r4), r28
        adds 1, r28, r29
        fsub.dd f22, f20, f24
        adds -5, r29, r0
```

```
        st.l r29, 0(r4)
        fst.d f24, r0(r19)
        bc.t .DB.B284284
        fld.d 8(r18)++, f16
// lineno: 84
.B236:  //.B0010
        ld.l r0(r17), r28
.DB.B236236:
        adds 1, r28, r29
        adds -4, r29, r0
        st.l r29, 0(r17)
        bc.t .DB.B296296
        fld.d r0(r20), f16
// lineno: 0
        st.l r18, -20(fp)
        st.l r19, -24(fp)
// lineno: 85
.B233:  //.B0008
        ld.l r0(r13), r28
.DB.B233233:
        adds 1, r28, r29
        st.l r29, 0(r13)
        orh ha%_Last, r0, r31
        ld.l l%_Last(r31), r30
        subs r29, r30, r0
        bc.t .DB.B297297
        ld.l r0(r13), r28
// lineno: 86
        orh h%_Irev, r0, r31
        or l%_Irev, r31, r13
        adds 3, r0, r28
        st.l r28, 0(r13)
        orh h%_i, r0, r31
        or l%_i, r31, r21
        st.l r0, 0(r21)
        orh h%_j, r0, r31
        or l%_j, r31, r17
        orh h%_Y, r0, r31
        or l%_Y, r31, r20
        orh h%_MatrizA+-8, r0, r31
        or l%_MatrizA+-8, r31, r4
        orh h%_MatrizX, r0, r31
        or l%_MatrizX, r31, r5
        orh h%_MatrizX+32, r0, r31
        or l%_MatrizX+32, r31, r6
// lineno: 95
.B295:  //.M0003
        ld.l r0(r13), r28
//      trap r5,r5,r0    // turn-on the detailed report generation
.DB.B295295:
        adds -16, fp, r30
        shl 2, r28, r29
        fld.l r30(r29), f16
        adds 40, r0, r16
        ixfr r16, f18
        ld.l r0(r21), r24
        fmlow.dd f18, f16, f20
        fxfr f20, r22
```

```
        adds 40, r22, r23
        adds r23, r4, r14
        fld.d r0(r14), f22
        fst.d f22, r0(r20)
        bte 0x0000, r24, .B241
        mov r6, r19
        mov r14, r18
        adds 1, r0, r25
        subs r24, r25, r0
        st.l r25, 0(r17)
        bc .B244
        mov r14, r18
        mov r6, r19
// lineno: 103
.B283:  //.M0000
        fld.d -8(r18)++, f16
.DB.B283283:
        fld.d -8(r19)++, f18
        fld.d r0(r20), f22
        ld.l r0(r17), r28
        fmul.dd f18, f16, f20
        adds 1, r28, r29
        st.l r29, 0(r17)
        fsub.dd f22, f20, f24
        ld.l r0(r21), r30
        subs r30, r29, r0
        fst.d f24, r0(r20)
        bnc.t .DB.B283283
        fld.d -8(r18)++, f16
// lineno: 106
.B244:  //.B0015
// lineno: 108
.B241:  //.B0013
        ld.l r0(r13), r28
.DB.B241241:
        ld.l r0(r21), r22
        adds -4, r28, r29
        shl 3, r29, r30
        fld.d r14(r30), f16
//        orh ha%.C00043, r0, r31
//        fld.d l%.C00043(r31), f22  f22=2
//        frcp.dd f16, f18           f16=A, f18=1/A
//        fmul.dd f16, f18, f20      f20=1
        shl 3, r28, r16
        adds 1, r22, r23
        frcp.dd f16, f26           // f16=A, f26=1/A
        st.l r23, 0(r21)
//        fsub.dd f22, f20, f24      f24=1
        adds -1, r28, r24
//        fmul.dd f18, f24, f26      f26=1/A
        fld.d r0(r20), f18         // f18=Y
//        fmul.dd f16, f26, f28      f28=1
//        fsub.dd f22, f28, f30      f30=1
//        fmul.dd f26, f30, f12      f12=1/A
//        fmul.dd f16, f12, f14      f14=1
//        fsub.dd f22, f14, f20      f20=1
//        fmul.dd f20, f12, f24      f24=1/A
//        fmul.dd f18, f24, f28      f28=Y/A
```

```
        fmul.dd f18, f26, f28       // f28=Y/A
        st.l r24, 0(r13)
        adds -4, r23, r0
        fst.d f28, r16(r5)
        bc.t .DB.B295295
        ld.l r0(r13), r28
// lineno: 0
//        trap r6,r6,r0     // turn-off the detailed report generation
        st.l r18, -20(fp)
        st.l r19, -24(fp)
// lineno: 0
// lineno: 109
        ld.l -72(fp), r4
        ld.l -68(fp), r5
        ld.l -64(fp), r6
        ld.l -60(fp), r7
        ld.l -56(fp), r8
        ld.l -52(fp), r9
        ld.l -48(fp), r10
        ld.l -44(fp), r11
        ld.l -40(fp), r12
        ld.l -36(fp), r13
        ld.l -32(fp), r14
        ld.l -28(fp), r15
        adds .a1+16, fp, r31
        ld.l 4(fp), r1
        ld.l 0(fp), fp
        bri r1
        mov r31, sp
        .data
        .align  8
.C00043:        // (0)
        .long   0x0, 0x40000000 // 2.0000000000000000E+00
        .data
        .globl  _MatrizA
        .align  8
_MatrizA:       //MatrizA
        .long   0x0, 0xbff00000 // -1.0000000000000000E+00
        .long   0x0, 0x40080000 // 3.0000000000000000E+00
        .long   0x0, 0x40140000 // 5.0000000000000000E+00
        .long   0x0, 0x40000000 // 2.0000000000000000E+00
        .long   0x0, 0x40240000 // 1.0000000000000000E+01
        .long   0x0, 0x3ff00000 // 1.0000000000000000E+00
        .long   0x0, 0x40220000 // 9.0000000000000000E+00
        .long   0x0, 0x40200000 // 8.0000000000000000E+00
        .long   0x0, 0x40100000 // 4.0000000000000000E+00
        .long   0x0, 0x402e0000 // 1.5000000000000000E+01
        .long   0x0, 0x0        // 0.0000000000000000E+00
        .long   0x0, 0x3ff00000 // 1.0000000000000000E+00
        .long   0x0, 0x0        // 0.0000000000000000E+00
        .long   0x0, 0x3ff00000 // 1.0000000000000000E+00
        .long   0x0, 0x40000000 // 2.0000000000000000E+00
        .long   0x0, 0x40000000 // 2.0000000000000000E+00
        .long   0x0, 0x3ff00000 // 1.0000000000000000E+00
        .long   0x0, 0x3ff00000 // 1.0000000000000000E+00
        .long   0x0, 0xbff00000 // -1.0000000000000000E+00
        .long   0x0, 0xc0080000 // -3.0000000000000000E+00
        .globl  _MatrizX
```

```
_MatrizX:     //MatrizX
      .long  0x0, 0x0      // 0.0000000000000000E+00
      .long  0x0, 0x0      // 0.0000000000000000E+00
      .long  0x0, 0x0      // 0.0000000000000000E+00
      .long  0x0, 0x0      // 0.0000000000000000E+00
      .data
      .comm  _Y,8
      .comm  _Big,8
      .comm  _Pivot,8
      .comm  _Const,8
      .comm  _Last,4
      .comm  _Irev,4
      .comm  _i,4
      .comm  _j,4
      .comm  _k,4
      .comm  _NextR,4
      .comm  _L,4
//      .extern _exit
```

A.5. - Programa para calcular o produto de matrizes

```
#define X_MATRIX_DIM  8
#define Y_MATRIX_DIM  1

double axgAMatrix [Y_MATRIX_DIM][X_MATRIX_DIM] =
    { 1.00, 1.01, 1.02, 1.03, 1.04, 1.05, 1.06, 1.07 } ;

double axgBMatrix [Y_MATRIX_DIM][X_MATRIX_DIM] =
    { 1.00, 1.01, 1.02, 1.03, 1.04, 1.05, 1.06, 1.07 } ;

double axgCMatrix [Y_MATRIX_DIM][Y_MATRIX_DIM] = { 0.0 } ;

double xgProdValue = 0.0;

main(argc, argv)
  int argc;
  char *argv[ ];
{
  int   ilAuxCnt                ;
  double *lxlAMatrixPtr, *lxlBMatrixPtr ;

  ilAuxCnt    = X_MATRIX_DIM;
  lxlAMatrixPtr = (double *) axgAMatrix ;
  lxlBMatrixPtr = (double *) axgBMatrix ;

  while (ilAuxCnt > 0)
  {
    ilAuxCnt--;
    xgProdValue += *lxlAMatrixPtr * *lxlBMatrixPtr++ ;
    lxlAMatrixPtr++;
    lxlBMatrixPtr++;
  } /* endwhile */
}
```

A.6. - Programa para analisar operações com matrizes 200x8

Toda a parte de iniciação das matrizes neste exemplo foram retiradas do programa pois trata-se de matrizes 200x8.

```
#define X_MATRIX_DIM  200
#define Y_MATRIX_DIM  8

double axgAMatrix [X_MATRIX_DIM][Y_MATRIX_DIM] =
    {{ 1.00, 1.08, 1.16, 1.24, 0.00, 0.08, 0.16, 0.24... }};

double axgBMatrix [X_MATRIX_DIM][Y_MATRIX_DIM] =
    {{ 1.01, 1.09, 1.17, 1.25, 0.01, 0.09, 0.17, 0.25... }};

double axgCMatrix [X_MATRIX_DIM][Y_MATRIX_DIM] =
    {{ 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00... }};

main(argc, argv)
  int argc;
  char *argv[];
{
  int ilMatrixLine, ilMatrixCol, ilAuxCnt ;

  for (ilMatrixLine = 0; ilMatrixLine < X_MATRIX_DIM; ilMatrixLine++) {
    for (ilMatrixCol = 0; ilMatrixCol < Y_MATRIX_DIM; ilMatrixCol++) {
      axgAMatrix [ilMatrixLine][ilMatrixCol] +=
        axgAMatrix [ilMatrixLine][ilMatrixCol] ;
      axgBMatrix [ilMatrixLine][ilMatrixCol] +=
        axgBMatrix [ilMatrixLine][ilMatrixCol] ;
      axgCMatrix [ilMatrixLine][ilMatrixCol] +=
        axgAMatrix [ilMatrixLine][ilMatrixCol] +
        axgBMatrix [ilMatrixLine][ilMatrixCol] ;
    }
  }
}
```

Apêndice B

Códigos de Erro

Erros relativos às operações envolvendo caches:

DATA_WAS_NOT_FOUND_IN_PC	0xBB02	Foi realizado um acesso a uma posição de memória que não pertence aos bancos de memória de dados existentes. AÇÃO: Verificar através das informações de saída na tela durante a execução da simulação e das informações do relatório detalhado em que trecho do programa este erro foi detectado. Em seguida analisar o porque houve um acesso a uma região de memória não existente (sugestão: verifique as atribuições dos ponteiros ou os índices dos vetores utilizados).
MISALIGNED_ADDRESS	0xBB04	Foi realizado um acesso a uma posição de memória não múltipla de 4 (endereço não alinhado). AÇÃO: Verificar através das informações de saída na tela durante a execução da simulação e das informações do relatório detalhado em que trecho do programa este erro foi detectado. Em seguida analisar se o fluxo de execução está sendo perdido (exemplo: chamada indireta de uma função através de ponteiro para função não iniciado) ou se algum acesso incorreto à memória está sendo feito (exemplo: acesso a memória através de ponteiro não iniciado).
INSTR_WAS_NOT_FOUND_IN_PC	0xBB06	Foi realizado um acesso a uma posição de memória que não pertence aos bancos de memória de instruções existentes. AÇÃO: Verificar através das informações de saída na tela durante a execução da simulação e das informações do relatório detalhado em que trecho do programa este erro foi detectado. Em seguida analisar se o fluxo de execução está sendo perdido (exemplo: chamada indireta de uma função através de ponteiro para função não iniciado).

Erros relativos às operações envolvendo a unidade central:

REGISTER_CAN_NOT_BE_WRITEN	0x B102	Foi feito um acesso a um registrador de controle inexistente. AÇÃO: Verificar através das informações de saída na tela durante a execução da simulação e das informações do relatório detalhado em que instrução do programa este erro foi detectado. Caso as instruções do programa tenham sido feitas diretamente em binário, rever a codificação referente ao registrador na instrução aonde a falha foi detectada. Caso o programa tenha sido gerado através de um fonte em ASSEMBLY, reportar o erro ao fornecedor do montador (ASSEMBLER).
DATA_PC_MEMORY_ADDR_NOT_FOUND	0x B104	Erro interno do simulador: endereço inválido escrito na cache de dados. AÇÃO: Contactar o laboratório de Computação Paralela da COPPE Sistemas.
MISALIGNED_ADDR	0x B106	Idêntico a MISALIGNED_ADDRESS
NOT_A_VALID_DEST_REG	0x B108	Foi feito um acesso a um registrador de inteiros ou de ponto flutuante inválido. AÇÃO: Verificar através das informações de saída na tela durante a execução da simulação e das informações do relatório detalhado em que instrução do programa este erro foi detectado. Caso as instruções do programa tenham sido feitas diretamente em binário, rever a codificação referente ao registrador na instrução aonde a falha foi detectada. Caso o programa tenha sido gerado através de um fonte em ASSEMBLY, reportar o erro ao fornecedor do montador (ASSEMBLER).

Erros relativos às operações envolvendo o controle da simulação:

COULD_NOT_DETERMINATE_ENTRY_POINT	0x B002	Arquivo .COFF não possui nenhuma seção de código. AÇÃO: rever o processo de geração do arquivo .COFF.
SRC2_NOT_A_VALID_REG	0x B004	Idêntico a REGISTER_CAN_NOT_BE_WRITEN

DPC_NOT_VALID	0x B006	Instrução inválida recebida. AÇÃO: Verificar através das informações de saída na tela durante a execução da simulação e das informações do relatório detalhado em que instrução do programa este erro foi detectado. Caso as instruções do programa tenham sido feitas diretamente em binário, rever a codificação na instrução aonde a falha foi detectada. Caso o programa tenha sido gerado através de um fonte em ASSEMBLY, reportar o erro ao fornecedor do montador (ASSEMBLER).
BAD_EXECUTE_OVERLAP_CTRL	0x B008	Erro interno do simulador: número excessivo de instruções sendo processadas no estado de "execução". AÇÃO: Contactar o laboratório de Computação Paralela da COPPE Sistemas.

Erros relativos às operações envolvendo a unidade de ponto-flutuante:

INTRUCTION_NOT_VALID	0x B302	Instrução inválida detectada no estágio de decodificação. AÇÃO: Verificar através das informações de saída na tela durante a execução da simulação e das informações do relatório detalhado em que instrução do programa este erro foi detectado. Caso as instruções do programa tenham sido feitas diretamente em binário, rever a codificação referente ao registrador na instrução aonde a falha foi detectada. Caso o programa tenha sido gerado através de um fonte em ASSEMBLY, reportar o erro ao fornecedor do montador (ASSEMBLER).
PRECISION_NOT_VALID	0x B304	Erro interno do simulador: erro no controle da precisão do *pipeline* de multiplicação. AÇÃO: Contactar o laboratório de Computação Paralela da COPPE Sistemas.
NUMBER_OF_STAGES_NOT_VALID	0x B306	Erro interno do simulador: erro no controle do número de estágios do *pipeline* de multiplicação. AÇÃO: Contactar o laboratório de Computação Paralela da COPPE Sistemas.

Erros relativos às operações envolvendo os relatórios:

UNABLE_OPEN_INPUT_FILE	0xDD02	Erro na abertura do arquivo de entrada. AÇÃO: Verifique se o nome do arquivo de relatórios contém algum caracter inválido ou se o caminho para chegar a ele está bem especificado (caminho não contém nenhum diretório inexistente).
UNABLE_OPEN_OUTPUT_FILE	0xDD04	Erro na abertura do arquivo de saída do simulador. AÇÃO: Verifique se o nome do arquivo de relatórios contém algum caracter inválido ou se o caminho para chegar a ele está bem especificado (caminho não contém nenhum diretório inexistente).
WRONG_NBR_INPUT_PARMS	0xDD06	Simulador foi chamado com número inválido de parâmetros de linha de comando. AÇÃO: Reveja a definição dos parâmetros de entrada do SIM860 e o chame novamente.
UNABLE_WRITE_REPORT_DATA	0xDD08	Falha na escrita dos registros do arquivo temporário. RELATORI.LOG durante a execução do SIM860. AÇÃO: Verifique se o disco referente ao diretório aonde o programa foi chamado está cheio ou danificado.
UNABLE_READ_FROM_LOG_FILE	0xDD0A	Falha na leitura dos registros do arquivo temporário RELATORI.LOG durante a formatação do relatório do SIM860. AÇÃO: Verifique se o disco referente ao diretório aonde o programa foi chamado está cheio ou danificado.
INVALID_INSTR	0xDD0C	Instrução inválida detectada durante a formatação do relatório do SIM860. AÇÃO: Verificar através das informações do relatório detalhado em que instrução do programa este erro foi detectado. Caso as instruções do programa tenham sido feitas diretamente em binário, rever a codificação na instrução aonde a falha foi detectada. Caso o programa tenha sido gerado através de um fonte em ASSEMBLY, reportar o erro ao fornecedor do montador (ASSEMBLER).
INVALID_REPORT_NAME	0xDD0E	Nome inválido para o arquivo de relatório. AÇÃO: Escolha um nome de arquivo DOS válido para o relatório de saída do simulador.

INVALID_REPORT_OPTION	0xDD10	Pedido de mais de um tipo de relatório numa única chamada do SIM860. AÇÃO: O SIM860 permite a geração de apenas 1 tipo de relatório em cada execução. Escolha apenas um tipo de relatório (detalhado ou simplificado) e chame novamente o SIM860.
UNABLE_OPEN_REPORT_FILE	0xDD12	Falha na abertura do arquivo de relatório. AÇÃO: Verifique se o disco referente ao diretório aonde o programa foi chamado está cheio ou danificado.

Erros relativos às operações envolvendo o apresentador de gráficos (SHOWREPO):

NO_CHOSEN_PARM	0x1702	Nenhum parâmetro de entrada foi escolhido para o SHOWREPO. AÇÃO: Reveja os parâmetros de entrada do SHOWREPO e o chame novamente.
NO_VALID_PARM	0x1704	Nenhum parâmetro de entrada válido foi escolhido para o SHOWREPO. AÇÃO: Reveja os parâmetros de entrada do SHOWREPO e o chame novamente.
INVALID_LINE_TYPE	0x1706	Registro inválido encontrado pelo SHOWREPO no arquivo de entrada. AÇÃO: Use como arquivo de entrada um arquivo gerado pelo SIM860 (renomeie o arquivo temporário RELATORI.LOG após a execução que você deseja analisar) e chame novamente o SHOWREPO.
IRRECOVERABLE_ERROR	0x1712	Erro interno do simulador AÇÃO: Contactar o laboratório de Computação Paralela da COPPE Sistemas.
NO_INPUT_DATA_FILE	0x1714	Arquivo contendo os registros com as utilizações de recursos durante os ciclos de relógio do I860 não foi especificado. AÇÃO: Chame novamente o SHOWREPO especificando o nome do arquivo de registros após a diretiva -f=
UNABLE_OPEN_DATA_FILE	0x1716	Erro na leitura do arquivo contendo os registros com as utilizações de recursos durante os ciclos de relógio do I860. AÇÃO: Verifique se o nome do arquivo de entrada está certo e se o arquivo não está danificado (verifique seu tamanho e tente copiá-lo para outro disco).

UNABLE_OPEN_GRAPHICS_MODE	0x1718	Modo gráfico do PC não pôde ser iniciado. AÇÃO: Verifique se seu computador suporta a interface de vídeo Hércules ou alguma mais moderna (CGA, VGA, ...). Em caso positivo contacte o laboratório de Computação Paralela da COPPE Sistemas.
UNABLE_POSITION_DATA_FILE	0x1720	Erro no tratamento do arquivo contendo os registros com as utilizações de recursos durante os ciclos de relógio do I860. AÇÃO: Verifique se o arquivo está danificado (verifique seu tamanho e tente copiá-lo para outro disco).

Erros relativos às operações envolvendo leitura de dados do arquivo COFF:

UNABLE_OPEN_FILE	0xAA02	Erro na leitura do arquivo .COFF. AÇÃO: Verifique se o nome do arquivo de entrada está certo e se o arquivo não está danificado (verifique seu tamanho e tente copiá-lo para outro disco).
NO_INPUT_FILE	0xAA04	Nenhum arquivo .COFF especificado. AÇÃO: Reveja os parâmetros de entrada do SIM860 e o chame novamente especificando o arquivo .COFF correspondente ao programa a ser simulado.
UNABLE_READ_FILE_HDR	0xAA06	Erro no tratamento do arquivo .COFF. AÇÃO: Verifique se o arquivo não está danificado (verifique seu tamanho e tente copiá-lo para outro disco) Verifique também se o arquivo é um .COFF válido.
INVALID_OPT_HDR_LEN	0xAA08	Tamanho de cabeçalho opcional inválido. AÇÃO: Verifique se o arquivo não está danificado (verifique seu tamanho e tente copiá-lo para outro disco). Verifique também se o arquivo é realmente um .COFF válido.
UNABLE_ALLOC_MEM	0xAA0A	Impossível alocar memória para armazenar o programa de entrada. AÇÃO: Verifique se a memória de se micro não está sobrecarregada com "driver" ou programas residentes. Verifique também se seu programa não é muito grande para ser executado (maior que 640K de código + dados).
UNABLE_READ_OPT_FILE_HDR	0xAA0C	Idêntico a UNABLE_READ_FILE_HDR
UNABLE_READ_SECTION_HDR	0xAA0E	Idêntico a UNABLE_READ_FILE_HDR
UNABLE_READ_SECTION_DATA	0xAA10	Idêntico a UNABLE_READ_FILE_HDR

Erros relativos às atividades de suporte (Tratamento de erros):

UNABLE_OPEN_TRACE_FILE	0xCC02	Erro na abertura do arquivo de trace (serviço interno do simulador). AÇÃO: verifique se o disco correspondente ao diretório aonde o programa foi chamado está cheio ou danificado.
PARAMETER_TOO_LONG	0xCC04	Erro interno do simulador. AÇÃO: Contactar o laboratório de Computação Paralela da COPPE Sistemas.

Erros relativos às operações envolvendo filas:

EMPTY_QUEUE	0xAE08	Condição de fila vazia em situação inesperada ou inválida. AÇÃO: Contactar o laboratório de Computação Paralela da COPPE Sistemas.
INVALID_INPUT_DATA	0xAE12	Erro interno do simulador: Função da biblioteca de filas chamada com parâmetro inválido. AÇÃO: Contactar o laboratório de Computação Paralela da COPPE Sistemas.
NOT_FOUND	0xAE1E	Condição de elemento não achado em uma fila em situação inesperada ou inválida. AÇÃO: Contactar o laboratório de Computação Paralela da COPPE Sistemas.

Referências Bibliográficas

[1] Intel, "i860 64-Bit Microprocessor Family Programer's Reference Manual", Intel, 1991.

[2] Intel, "i860 XP Microprocessor Data Book", Intel, 1991.

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

[4] Intel, "i860 64-Bit Microprocessor Simulator and Debugger Reference Manual", Intel, Order Number 240437-000, January 1990.

[5] M. Johnson, "Superscalar Microprocessor Design", Prentice Hall, 1991.

[6] A. S. Tanenbaum, "Structured Computer Organization", Prentice Hall, 1990.

[7] N. Margulis, "i860 Microprocessor Architecture", McGraw-Hill, 1990.

[8] D. A. Patterson, and J. L. Hennessy, "Computer Architecture: A Quantitative Approach", Morgan Kaufmann, 1990.

[9] Hwang, "Highly Parallel Computer Architecture", McGraw-Hill, 1989

[10] M. Atkins, "performance and the i860 Microprocessor", IEEE Micro, October 1991, pp. 24-27 e 72-78.

[11] N. Margulis, "i860 Microprocessor Internal Architecture", Microprocessors and Microsystems, Vol. 14, No. 2, March 1990, pp. 89-96.

[12] R. S. Piepho, and William S. Wu, "A Comparison of RISC Architectures", IEEE Micro, August 1989, pp. 51-61.

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

[14] 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.

[15] "The Common Object File Format (COFF)", Unix System V/386 Programe's Guide.

[16] J. E. Smith, "Decoupled Access/Execute Computer Architectures", 9th Annual Symposium on Computer Architecture, April 1982, pp. 112-119.

[17] S. Heath, "Performance Improvement Techniques for the M88000 RISC Architecture, Microprocessors and Microsystems, Vol. 14, No. 6, July/August 1990, pp. 377-384.

[18] R. R. Oehler, and M. W. Blasgen, "IBM RISC sytem/6000: Architecture and Performance, IEEE Micro, June 1991.

[19] S. Mirapuri, Michael Woodacre, and Nader Vasseghi, "The Mips R4000 Processor", IEE Micro, April 1992, pp.10-22.

[20] J. Cocke, and V. Markstein, "The Evolution of RISC Technology at IBM", IBM Journal of Research and Development, Vol. 34, No. 1, pp.4-10.

[21] IBM Microeletronics and Motorola, "PowerPC 604 RISC Microprocessor Tecnical Summary", Advance Information, 1994.

[22] C. May, E, Silha, R. Simpson and H. Waren, "The Power PC Architecture: a specification for a new family of RISC processors", Morgan Kaufmann, Inc., San Francisco, California, 1994.