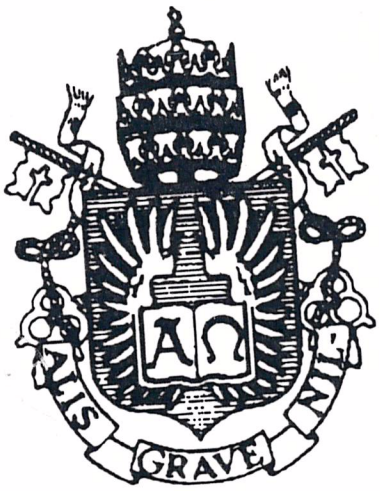
Marisa



PUC

Monografias em Ciência da Computação
nº 9/92

Esquemas de Escalonamento de Instruções para a Arquitetura RISC/6000

Mariza Andrade da Silva Bigonha

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA

Monografias em Ciência da Computação, Nº 9/92
Editor: Carlos J. P. Lucena

Abril, 1992

Esquemas de Escalonamento de Instruções para a Arquitetura RISC/6000*

Mariza Andrade da Silva Bigonha

* Trabalho apresentado ao Prof. José Lucas Rangel.

Trabalho parcialmente financiado pela CAPES/UFMG e Secretaria de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

Para obter cópias:
Rosane T. L. Castilho
Assessoria de Biblioteca, Documentação e Informação
Rua Marquês de São Vicente, 225 - Gávea
22.453 - Rio de Janeiro, RJ.
Brasil

Abstract

Well-designed instruction-scheduling algorithms are essential to the performance of optimizing compilers. They are used to reduce possible run-time delays for the compiled code. The goal of this paper is to present techniques for instruction-scheduling within the scope of and beyond basic blocks,i.e., straight-line sections of code. In order to establish the proper context in which these techniques are applied, this paper presents a description of the architecture and machine design of the IBM RISC System/6000* processor. The text presentes an overview of the most important projected decisions related to this matter as described in the january 1990 edition of the *IBM Journal of Research and Development*. The paper also presents the machine organization, programs, and storage facilities that characterize this new family of superscalar RISC workstations and servers.

Keywords:RISC Architecture, Machine Organization, Instruction Scheduling, RS/6000*.

Sinopse

Algoritmos de escalonamento de instruções bem formulados são essenciais para o bom desempenho de compiladores otimizadores. Eles são utilizados em compiladores com o objetivo de reduzir possíveis atrasos durante a execução do código compilado. O objetivo deste trabalho é apresentar técnicas de escalonamento de instruções dentro e além do escopo de blocos básicos, i.e., seqüência de código com uma única entrada e saída. Para estabelecer o contexto no qual estas técnicas se encaixam, esta monografia apresenta a descrição da arquitetura e do projeto do processador do Sistema RISC/6000* da IBM. O texto captura um resumo das decisões mais importantes descritas no "*IBM Journal of Research and Development*", de janeiro de 1990, relacionadas com a pesquisa, decisões de projeto e desenvolvimento realizado pelos projetistas do sistema RS/6000. São apresentados também a organização de máquina, os programas e as facilidades de armazenamento que caracterizam esta nova família de estações de trabalho e servidoras baseadas em arquiteturas RISC.

Palavras-chave: Arquiteturas RISC, Organização de Máquina, Escalonamento de Instruções, RS/6000*.

SUMÁRIO

1 Introdução

Este texto apresenta um resumo dos trabalhos realizados por um grupo de pesquisadores responsáveis pelo projeto e desenvolvimento do sistema RISC/6000* da IBM [COCKE 90, OEHLER 90, BAKOGLU 90, WARREN 90, GROHOSKI 90].

O trabalho está distribuído da seguinte forma. Na seção dois é apresentada a evolução da tecnologia RISC na IBM e são descritas a configuração do sistema atualmente disponível, seu desempenho e suas capacidades funcionais através da apresentação da evolução da arquitetura "superscalar" POWER (*Performance Optimization With Enhanced RISC*) e das decisões de projeto que deram origem à implementação corrente. A seção três apresenta os problemas relacionados com o projeto de múltiplas unidades de execução e a implementação vigente da organização de máquina do processador RS/6000, dando ênfase especial para as operações de sincronização dos registradores e para o processamento do conjunto de instruções. Algoritmos bem projetados para resolver o problema do escalonamento de instruções são essenciais para o bom desempenho de compiladores otimizadores. A seção 4 descreve técnicas de escalonamento de instruções necessárias para o conjunto de instruções e arquitetura do processador RS/6000. A seção 5 apresenta um algoritmo proposto por Warren [WARREN 90] desenvolvido para atender as necessidades de escalonamento descritas na seção 4. A seção 6 explora a possibilidade de ampliar o escopo do algoritmo de escalonamento de instruções, desenvolvido por Warren, apresentando uma técnica mais sofisticada. A seção 7 apresenta a conclusão.

2 Evolução da Tecnologia RISC na IBM

2.1 Histórico

A primeira máquina RISC, o 801, foi projetada no laboratório de pesquisa da IBM Thomas J. Watson em 1975. A partir desta data, houve várias implementações baseadas na idéia original. Cada uma destas implementações estendeu a idéia original de uma forma diferente. Foram incluídas extensões para coprocessadores, endereçamento virtual, e controle de entrada e saída. Muito embora a maior parte dessas implementações nunca tenham feito parte de produto algum da IBM, algumas o fizeram. O produto mais expressivo baseado em RISC foi o IBM RT System [OEHLER 90].

Em meados de 1975 não havia nenhum software para o 801. Foi então elaborado um montador (*assembler*) e um simulador. O simulador foi projetado para traduzir cada instrução de 24 bits do 801 para instruções de 32 bits do código do sistema/370, responsável pela implementação das instruções do 801. A linguagem de alto nível desenvolvida foi PL/8, um subconjunto do PL/1 [AUSLANDER 82]. Inicialmente, PL/8 era um autêntico subconjunto de PL/1, permitindo assim, que o compilador fosse desenvolvido no sistema/370

e sua saída, código do 801, testado no simulador.

Em torno de 1977, muito embora existissem somente resultados do simulador como base, o 801 havia conseguido um excelente desempenho. A qualidade do código gerado era expressiva quando comparada com o código produzido pelo PL/1. Mesmo compilado para o 801, e simulado no sistema/370, modelo 168, ele executava mais rápido em tempo real do que o mesmo programa, se executado diretamente no modelo 168 quando compilado pelo compilador PL/1.

O projeto do 801 foi concluído em 1978, tendo cumprido seu objetivo de executar uma instrução por ciclo. Entretanto, alguns dos problemas detectados nesta primeira arquitetura RISC permaneceram. Eram eles:

- o grande número de manipulação de *strings* ocorrendo no compilador;
- a necessidade de mais *hardware* para propagar vai-um entre os subcampos de 4-bits na aritmética decimal;
- a alocação de registradores, embora parecesse eficiente poderia ser melhorada: seus 16 registradores não eram suficientes para usufruir das vantagens da arquitetura e seriam necessário 32 registradores.

Além destes problemas, o 801 permitia no máximo 16 megabytes de memória. Com a redução dos preços de memória naquela época, era evidente que para tal arquitetura competir no mercado era preciso expandir a sua capacidade de endereçamento, e, portanto, memória virtual não poderia ser ignorada.

No segundo projeto do 801, as instruções passaram a ser de 32 bits ao invés de 24 bits e o número de registradores passou para 32. O conjunto de instruções foi acrescido de algumas instruções mais complexas. Mais notável foi a integração das instruções de ponto flutuante na arquitetura. Elas foram incluídas no conjunto de instruções originais do 801 porque tais operações ocorrem freqüentemente na maior parte das aplicações científicas.

A partir de 1985, alguns pesquisadores envolvidos com o projeto inicial, o 801, consideraram novamente o problema das arquiteturas de máquina. Examinando não somente o Sistema /370, mas tendo como base a experiência adquirida com a elaboração do projeto do 801 e seus sucessores, eles efetuaram estudos sobre a organização e eficiência da unidade de ponto flutuante, sobre a eficácia da arquitetura tendo como objetivo o compilador, e mais importante, reexaminaram os efeitos que a organização de máquina e arquitetura teriam nas execuções paralelas e *pipelining*. O objetivo deste novo projeto era definir uma arquitetura cuja implementação pudesse facilmente executar mais de uma instrução por ciclo.

Para avaliar o processador existente, os pesquisadores consideraram três componentes: número de instruções, ciclos por instruções e o tempo do ciclo. Como já existia um modelo

de fluxo de dados da arquitetura original que se concentrou em reduzir o número de níveis lógicos necessário para efetuar um ciclo sentiu-se que, desde que este modelo não sofresse mudanças substanciais, uma nova tecnologia, com um nível mais elevado de integração poderia trazer beneficios e reduzir o tempo de ciclo. Como conseqüência desta observação não se concentrou nenhum esforço a nível de arquitetura para reduzir este tempo. Toda a ênfase foi direcionada para os outros dois componentes, redução do número de instruções e redução do número de ciclos por instrução. Para reduzir o número de instruções necessárias à execução de uma tarefa, instruções do tipo funções compostas (*compound-function instructions*) foram exploradas. Estas instruções, tais como multiplica-e-soma, carrega-e-atualiza, armazena-e-atualiza podiam substituir duas ou mais das instruções originais do 801. O uso de VLSI propiciou o uso de *pipelining* adicionais para diminuir os atrasos causados pelos acessos à memória e desvios condicionais. Como resultado desta atividade este novo projeto evoluiu, originando a segunda geração da arquitetura RISC, denominada arquitetura AMERICA. Nela havia três processadores semi-autônomos: o processador do fluxo de instruções, o processador de ponto fixo e processador de ponto flutuante. A nova máquina possuía uma unidade para multiplicação-e-soma de ponto flutuante e era capaz de executar concorrentemente uma instrução de ponto fixo, uma de ponto flutuante, uma de desvio e uma LCR (*condition register logic*).

A organização do processador RS/6000 é idêntica à organização da arquitetura AMERICA [GROHOSKI 90]. A unidade de cache de instruções (instruction cache unit (ICU)) carrega as instruções, decodifica e executa as instruções de desvios, etc. Ela dispara duas instruções por ciclo para as unidades de ponto fixo e ponto flutuante e recebe informações sobre o código de condição de cada unidade em barramentos dedicados. A unidade de ponto fixo (FXU) decodifica e executa instruções de ponto fixo, efetua cálculos de endereços para carga e armazenamento de ponto flutuante, além de conter o endereço, diretórios e controle para o cache de dados. A unidade de ponto flutuante (FPU) é um *chip* de alta velocidade capaz de executar carregamentos (*loads*) de ponto flutuante em paralelo com as instruções aritméticas, além de decodificar suas próprias instruções. A unidade de controle do sistem (SCU) contém o controle e a interface de entrada e saída e de memória. A unidade de cache de dados (DCU) contém arranjos de 64 kbytes cache de dados (*data-cache arrays*) e *data-cache buffers*. Todas estas unidades serão examinadas com mais detalhes ao longo deste texto.

Do ponto de vista de software, o conjunto de instruções continuava simples. A nova máquina comportava-se como se executasse seqüencialmente. Entretanto, isto na realidade não ocorria pois os três processadores podiam operar concorrentemente.

Esta nova arquitetura, posteriormente, evoluiu para o processador RISC/6000 e novos produtos foram desenvolvidos. Atualmente, a idéia do projeto original está sendo implementado no IBM RISC SYSTEM/6000* (RS/6000) com a versão do processador RS/6000 da arquitetura "POWER" (*Performance Optimization With Enhanced RISC*) [BAKOGLU 90].

2.2 Arquitetura do Sistema RISC/6000

Uma das principais características da arquitetura RS/6000 é a separação dos componentes do processador em unidades funcionais. Há três unidades principais: ponto fixo, ponto flutuante e desvio. Cada uma destas unidades pode processar instruções em paralelo, sendo a unidade de desvio responsável pelo controle geral e pela integridade da execução do programa.

A Figura 1 apresenta uma visão geral da arquitetura do sistema/RS6000. Nesta figura é mostrado como os registradores são distribuídos entre as várias unidades. Por exemplo, na unidade de desvios os principais registradores são: CR - registrador de condição, LR - registrador de ligação, CTR - registrador contador, MSR - registrador do estado da máquina.
Na unidade de ponto fixo os principais registradores são: GPRs - 32 registradores de propósito geral de 32 bits cada um, TID - identificador das transações, XER - registrador de exceções de ponto fixo, MQ - registrador utilizado para operações de multiplicação e divisão.
Na unidade de ponto flutuante os principais registradores são: FPSCR - registrador de controle de estado, FPRs - 32 registradores de ponto flutuante de 64 bits cada um.

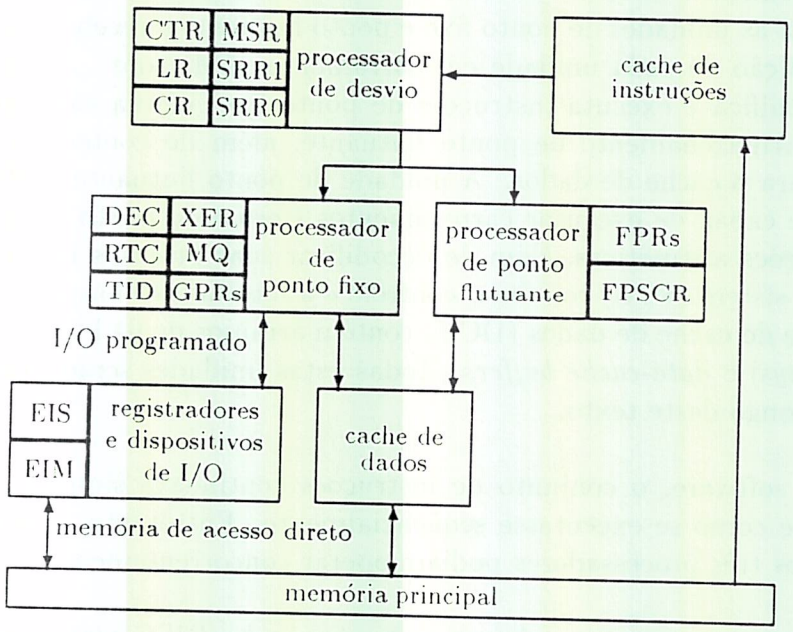


Figura 1: Visão lógica da arquitetura do sistema/ RS6000

Em geral, todos os projetos elaborados para a unidade central de processamento das arquiteturas existentes incluem nesta unidade mecanismos para efetuarem instruções de ponto fixo e desvio, e mesmo para ponto flutuante. Portanto, o que torna a arquitetura

do RS/6000 diferente das demais é a forma em que ela aplica a filosofia do 801 original. Ou seja, ela aumenta o papel do compilador e do sistema operacional na administração do *hardware*. Enquanto a maioria dos projetos de unidade central de processamento dedica um número expressivo de componentes de *hardware*, tentando prover o máximo de paralelismo entre as unidades funcionais através de elaboradas verificações de dependências de registradores, de predição de desvios, etc., a arquitetura do RS/6000 transfere os registradores para as unidades funcionais, separando a atribuição dos códigos de condições das atividades normais das instruções e exigindo a administração do programa em algumas interações *store/load*. Como resultado desta designação de registradores, ela evita grande parte do *hardware* necessário em outras arquiteturas e expõe ao compilador e sistema operacional o paralelismo latente do processador, tornando necessário a administração explícita das várias unidades e suas interações com o objetivo de explorar totalmente o *hardware*.

A segunda característica que distingue o RS/6000 das demais arquiteturas é a ênfase atribuída ao desempenho do ponto flutuante. Bem cedo no projeto foi concebida uma avançada instrução de multiplicação-e-soma de ponto flutuante ((A x B) + C) que permitia uma operação de multiplicação-e-soma de 64 bits ser efetuada em um único ciclo e com o mesmo atraso que uma instrução de simples multiplicação ou adição. Também foi observado que o desempenho do ponto flutuante na maioria das vezes era condicionado pelo desempenho do ponto fixo. Por exemplo, a maioria das operações com matrizes necessita de um número expressivo de cálculos de endereços para carga e armazenamento. Além disso, existem muitas instruções repetidas que devem ser carregadas e executadas. Portanto, para explorar plenamente esta operação, a arquitetura do RS/6000 foi projetada para efetuar todas as operações de armazenamento, incluindo os cálculos de endereços na unidade de ponto fixo e para efetuar a carga do programa e a execução dos desvios na unidade de desvio, com todas as três unidades sobrepostas. A partir destas duas características da arquitetura RS/6000, o princípio básico do projeto foi idealizado: pesquisar uma organização de sistema que ofereça o máximo de sobreposição das unidades funcionais.

2.3 Organização do Sistema RISC/6000

A arquitetura do sistema RS/6000 define caches separados para dados e instruções como ilustrou a Figura 1. Cache é uma memória rápida utilizada como *buffer* entre a memória principal e a unidade central de processamento. Os benefícios desta organização são discutidos em [RADIN 87] e em *The 801 Minicomputer* publicado no periódico SIGARCH Computer Architecture News, 10, 39-47, March 1982.

O cache de instruções é associado com a unidade de desvio, e o cache de dados é compartilhado entre as unidades de ponto fixo e de ponto flutuante para acesso aos dados. A unidade de desvio administra o cache de instruções, e a unidade de ponto fixo administra o cache de dados.

2.3.1 Unidade de Desvio

A unidade de desvio é responsável pela carga das instruções e pela tradução dos endereços das instruções. Toda a administração de interrupções é também efetuada por esta unidade.

Os principais registradores da unidade de desvio são: o registrador de ligação, o registrador contador, o registrador de condição e o registrador do estado da máquina. O registrador de ligação é utilizado para guardar o endereço de retorno da instrução para ligação de subrotina. O registrador contador é utilizado para controlar a iteração de *loops*. Existe uma instrução de desvio que decrementa o registrador contador de um e desvia de acordo com o valor do resultado. O registrador do estado da máquina controla os estados do sistema.

A partir destes quatro registradores, é possível para a unidade de desvio carregar a próxima instrução, decodificá-la e executá-la se a mesma for uma instrução da unidade de desvio. Caso contrário, a unidade de desvio dispara a instrução para as unidades de ponto fixo ou ponto flutuante que se encarregam de decodificá-la em sua apropriada unidade. A unidade de desvio continua sua tarefa até que a fila de instruções das outras unidades fique cheia ou até que haja uma dependência de dados em um dos registradores locais da unidade de desvio requisitando dados da unidade de ponto flutuante ou ponto fixo, e os mesmos ainda não estão disponíveis.

A arquitetura do RS/6000 assume que a unidade de desvio seja capaz de carregar pelo menos três instruções em um ciclo, uma para si própria, e uma para cada uma das outras unidades. Além disso, ela é capaz de disparar uma instrução de ponto flutuante e uma instrução de ponto fixo a cada ciclo.

As instruções de desvio possuem três tipos de cálculo de endereço: instruções de endereço relativo, endereço absoluto e registrador.

Os campos necessários para as instruções de endereço relativo e absoluto fazem parte das instruções e podem ser executados imediatamente. Quando não há condição a ser testada, o tamanho deste campo de instrução é de 26 bits.

O último tipo de endereçamento, registrador, é utilizado quando o endereço alvo é desconhecido em tempo de compilação ou excede 26 bits. O registrador, neste caso, é sempre um registrador da unidade de desvio, tipicamente o registrador de ligação. Para obter maior eficiência é imprescindível que o compilador mova o endereço para o registrador de ligação o mais cedo possível, de tal forma que a instrução alvo seja carregada pela unidade de desvio enquanto as instruções entre a carga do registrador de ligação e o registrador de desvio ainda estejam na fila de instruções.

Para desvios que dependam do registrador de condição, a situação difere um pouco. O registrador de condição é composto de oito campos de condição, dois dos quais são atribuídos

às unidades de ponto fixo e ponto flutuante, um para cada uma, para conter os resultados das operações aritméticas. As instruções aritméticas de ponto fixo e ponto flutuante possuem um bit, o Rc *record bit*, que indica quando o campo de condição correspondente deve ser ligado ou não. Os oito campos podem ser explicitamente ligados pelas instruções aritméticas de comparação, de ponto flutuante ou ponto fixo. Adicionalmente, há uma instrução especial da unidade de desvio que efetua operações lógicas nestes campos.

Existe uma série de motivos que induziram esta distribuição. Em primeiro lugar, a motivação para os dois campos de condição padrão é que as computações de ponto fixo e ponto flutuante envolvendo a atribuição de condição são completamente independentes de cada uma, e podem ser efetuadas em paralelo. Em segundo lugar, a motivação para o controle explícito da atribuição dos campos padrões pelo bit Rc são duas: somente as condições que são testadas necessitam ser salvas, e identificando explicitamente uma instrução que liga a condição facilita para o compilador escaloná-la para o mais longe possível do teste de condição. Em terceiro lugar, a motivação para os campos adicionais de condição é o reconhecimento de que as comparações são operações independentes entre si e não devem interferir umas com as outras ligando um código de condição comum. Finalmente, permitindo que comparações possuam seu próprio campo, as comparações podem ser escalonadas mais cedo, comparações repetidas podem ser eliminadas e comparações invariantes podem ser deslocadas para fora dos *loops* internos. Tendo os códigos de condição do compilador alvo e as comparações em campos diferentes no registrador de condição, o *hardware* pode explorar algoritmos de *scoreboarding* [THORNTON 70] para detectar dependências, sem afetar significantemente o seu desempenho.

A atribuição de um campo de condição por uma operação aritmética ou de comparação, e seu subseqüente teste representa um dos mais importantes requisitos para a coordenação entre as unidades de ponto fixo e ponto flutuante e a unidade de desvio. Sabe-se que é extremamente difícil ter uma operação de atribuição imediatamente seguida pelo teste de desvio sem perda de ciclos. Uma implementação possível para evitar esta perda, e que probabilisticamente reduz esta dependência, mas aumenta a complexidade do *hardware*, é fazer com que a unidade de desvio, condicionalmente, dispare instruções de ponto fixo e ponto flutuante. Esta então é uma área em que a implementação depara com um dos mais tradicionais compromissos, decidir entre uma significativa complexidade de *hardware* e um melhor desempenho. Mesmo quando se prediz o momento em que um desvio vai ocorrer, o compilador pode auxiliar. A seção 6 apresenta estratégias de soluções para dois casos de desvios condicionais que são analisados neste texto, o *if-then-else lógico* e o *loop-closure*. Em geral, sem o auxílio adicional de linguagem não é obvio qual o caminho do *if-then-else lógico* que deve ser tomado. Freqüentemente, a melhor opção é seguir em frente. Por outro lado, o "desvio para trás" para o *loop-closure* é sempre efetuado. Embora um processador RS/6000 não disponha da mesma informação que um compilador, entretanto, pela análise dos casos ele tem o deslocamento. (Em geral, os endereços de desvios são obtidos através do campo imediato na instrução de desvio). Se o deslocamento é negativo, isto é um bom indicativo de que a instrução de desvio é parte de um fechamento de *loop* e será efetuado.

2.3.2 Unidade de Ponto Fixo

A unidade de ponto fixo é responsável pelas instruções aritméticas normais de ponto fixo e por todos os cálculos de endereços dos dados para si própria e para a unidade de ponto flutuante. Neste papel, a unidade de ponto fixo escalona a movimentação de dados entre a unidade de ponto flutuante e o cache de dados.

A unidade de ponto fixo possui 32 registradores de 32 bits, operações com três operandos (RT = RA + RB), uma poderosa instrução de rotação e facilidades de máscara. Todas as instruções são de 32 bits com um campo imediato de 16 bits quando apropriado. O campo imediato pode ser utilizado como deslocamento e é relativo ao registrador. Instruções de carga e armazenamento também possuem características para endereço auto-incremento.

Cada operação aritmética calcula as três condições padrões, *menor que, igualdade e maior que*. *Overflow* é a quarta condição sob a supervisão de um bit de controle de *overflow*. O controle de *overflow* é necessário quando operações de precisão estendida estão sendo efetuadas.

Muito embora as condições sejam geradas em cada instrução aritmética, o primeiro campo do código de condição que indica o retorno para a unidade de desvio (*back in the branch unit*) não é ligado, a não ser que o bit Rc da instrução aritmética seja ligado.

2.3.3 Unidade de Ponto Flutuante

A unidade de ponto flutuante possui 32 registradores de ponto flutuante de 64 bits. Há também um registrador de estado de ponto flutuante e um registrador de controle que contém indicadores de exceção, máscaras de exceção *default* e condições de ponto flutuante.

O processamento do ponto flutuante do sistema RS/6000 é organizado para computações de precisão dupla. Isto significa que dados residentes nos registradores de ponto flutuante são sempre representados no formato de precisão dupla. Assim, quando um dado de precisão simples é carregado, ele é expandido para o formato de precisão dupla.

2.3.4 Modelo de Memória

O modelo de armazenamento da arquitetura RS/6000 é uma extensão do 801. Todos os endereços computados, denominados endereços efetivos, são de 32 bits. Excluindo o caso especial de entrada e saída programado, o modelo possui dois modos de endereçamento, real e virtual. Caso a máquina esteja operando no modo real, o endereço efetivo é igual ao endereço real e os 32 bits são utilizados para acessar a memória real. Caso a máquina esteja operando no modo virtual um nível de tradução deve ser efetuado. O identificador

de segmento (SID) no RS/6000 é de 24 bits resultando em um endereço virtual de 52 bits. O tamanho real de uma página é 4096 bytes e suporta um endereço real de 32 bits.

3 Problemas Associados ao Projeto de Múltiplas Unidades de Execução

Um projeto de arquitetura RISC que utiliza múltiplas unidades funcionais simultaneamente executa várias instruções por ciclo. Portanto, mais de uma instrução deve ser carregada a cada ciclo. O efeito de instruções de desvios no *pipeline* deve ser reduzido porque ele é relativamente maior do que em arquiteturas de máquinas que executam somente uma instrução por ciclo. As unidades de execução devem ser sincronizadas quando ocorrem interrupções para manterem a consistência de programas seqüenciais e para garantirem que operações aritméticas sejam efetuadas utilizando-se de dados corretos e na ordem correta.

Suavizando-se o efeito das instruções de desvio e suprindo as unidades de ponto flutuante e ponto fixo com instruções e dados em alta velocidade é possível obter um grande aumento no desempenho do processador. Mas para que isto aconteça alguns requisitos básicos devem ser observados [GROHOSKI 90]:

1. Projetar um mecanismo para carregar instruções de tal forma que seja viável, em um tempo mínimo, carregar um grande número de instruções.
2. Sobrepor a execução da instrução de desvio com instruções de ponto fixo e ponto flutuante.
3. Sobrepor as unidades de ponto fixo e ponto flutuante com o objetivo de manter a unidade de ponto flutuante abastecida de dados.
4. Manter os efeitos da execução seqüencial de um programa enquanto executando várias instruções em paralelo.
5. Projetar uma unidade de execução de ponto flutuante de alto desempenho.

A solução para os quatro primeiros requisitos que foi desenvolvida durante o projeto da arquitetura `AMERICA` é apresentada nas quatro subseções seguintes.

3.1 Carga de Instruções

O mecanismo de carga de instruções deve possuir uma baixa latência. Esta característica permite que as unidades de execução permaneçam ocupadas enquanto o objeto do desvio

tomado está sendo carregado, mas implica em um projeto de cache com tempo de acesso de um ciclo de máquina.

Embora o processador seja capaz de executar quatro instruções a cada ciclo de máquina, na presença de um denso código de ponto flutuante, ele normalmente executa apenas três instruções (uma instrução de desvio, uma instrução de ponto flutuante e uma instrução de ponto fixo) por ciclo de máquina. O *cache* deve, portanto, pelo menos satisfazer esta demanda. Para sobrepor a execução de uma instrução de desvio com instruções de ponto fixo e ponto flutuante, o verificador (`scanning`) de desvios lógicos, ao examinar o *buffer* de instruções, deve detectar de alguma forma um desvio antes de sua execução. Isto significa que a taxa de transferência (`bandwith`) do cache de instruções deve ser maior que a taxa necessária para a execução dos pipelines.

Levando em conta estas considerações, uma nova organização de cache foi desenvolvida, possibilitando a carga de múltiplas instruções em um ciclo. A nova organização, em que todas as instruções possuem 4 bytes (`1 word`) de comprimento funciona da seguinte forma. Considere um arranjo de cache de instruções composto de quatro arranjos, pequenos e independentes, cada um deles carregando uma instrução por ciclo. Controlando o endereço apresentado a cada arranjo, distribuindo as instruções entre os arranjos caches e providenciando multiplexadores e lógica para seleção de linha, além da incrementação necessária das linhas do arranjo cache, sempre quatro instruções podem ser carregadas desde que as mesmas estejam em uma mesma linha do cache. A Figura 2 apresenta parte da organização geral do arranjo de cache de instruções para caches de conjuntos associativos de caminhos duplos com tamanho de linha para 16 instruções. Cada instrução subseqüente é colocada em um arranjo cache distinto. Se o tamanho da palavra de cada arranjo for de duas instruções, a instrução 0 dos conjuntos associativos `A` e `B` ocupam a linha i do arranjo cache 0. A linha `i` do arranjo cache 1 contém a instrução 1 de uma dada linha do cache, e assim por diante. Desta forma, uma linha do cache é dividida em quatro linhas de cada arranjo cache.

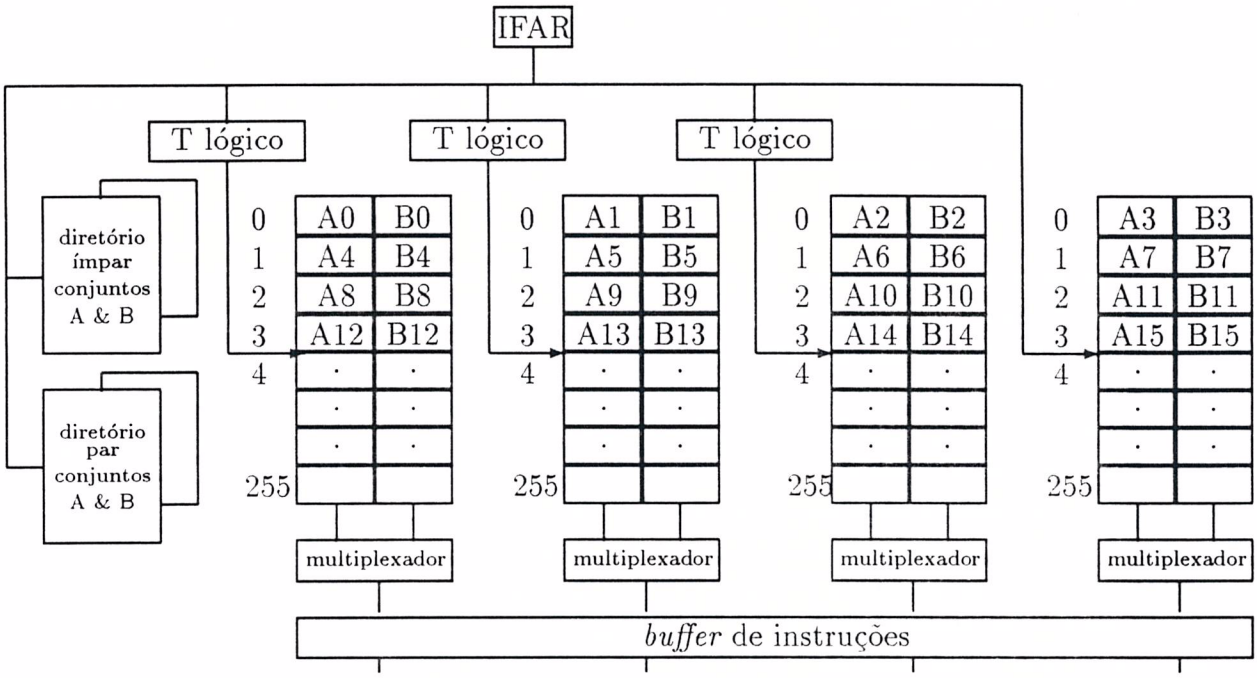


Figura 2: Parte da organização geral do cache de instruções

No esquema apresentado na Figura 2 para carregar as instruções 0, 1, 2 e 3 de uma dada linha do cache, o mesmo endereço de linha pode ser apresentado a todos os arranjos caches. Para carregar as instruções 1, 2, 3 e 4 de uma linha do cache, o endereço da linha para o arranjo cache 0 deve ser incrementado uma vez que a instrução 4 reside na próxima linha do mesmo. Isto é determinado pelo bit de endereço 28 (bit 0 é o bit de mais alta ordem de um endereço de 32 bits).

Considerando todas as 16 possibilidades do endereço inicial da palavra em uma linha de cache, sabe-se que os arranjos caches 0, 1 e 2 necessitam ter seus endereços incrementados enquanto o arranjo 3 não precisa ter.

3.2 Atrasos Causados por Desvios

Dentre os clássicos desafios existentes em um projeto de computador encontra-se a redução dos atrasos causados por desvios em máquinas tipo *pipeline*. Várias abordagens têm sido pesquisadas com o objetivo de efetuar desvios com zero ciclos de máquina.

A abordagem adotada por `AMERICA` é simples e consiste em duas partes. A primeira parte provê uma unidade separada para execução de desvios. Isto permite a efetivação de desvios com zero ciclos. A segunda parte provê lógica para pesquisar por desvios no *buffer* de instruções, com o intuito de gerar o endereço destino do desvio, e determinar,

se possível, o resultado do mesmo. Caso o seu resultado seja indefinido, instruções são disparadas condicionalmente a partir das instruções seguindo a instrução de desvio para as unidades de execução de ponto-flutuante e ponto-fixo. Ao determinar o resultado do desvio duas situações podem ocorrer. As instruções disparadas condicionalmente podem, tanto ser executadas e as instruções de destino do desvio descartadas, ou então elas podem ser canceladas, e as instruções de destino do desvio transmitidas para as unidades de execução.

A justificativa para esta estratégia fundamenta-se em dados estatísticos obtidos através do estudo de instruções de desvios presentes na arquitetura do `801`. Estes dados indicam que desvios compreendem aproximadamente 20% de todas instruções em código de ponto-fixo. Aproximadamente um terço dos desvios é incondicional, outro terço é utilizado para finalizar *do-loops*; e a terça parte final é atribuída a desvios condicionais. Portanto, se for utilizada uma unidade separada para execução de desvios com um mecanismo apropriado de procura à frente, com o objetivo de sobrepor a execução de desvios com a execução das instruções de ponto-fixo, então desvios incondicionais não causam atrasos no *pipeline*, dado que a meta do desvio se encontra no cache. Utilizando a instrução de desvio *loop-closing*, que é basicamente um desvio incondicional para as primeiras `n-1` interações, também não ocasiona um atraso. Dos desvios condicionais restantes, aproximadamente a metade é efetuada, e a outra metade não é executada. Desvios não executados não causam atrasos, uma vez que eles são prognosticados para não serem efetuados. Os desvios efetuados causam algum atraso, um valor estimado é de dois ciclos do *pipeline*. Portanto, desvios ao invés de necessitarem, cada um deles, um ciclo para execução, requerem aproximadamente 0,33 ciclos em média.

3.3 Sobreposição das Unidades de Ponto Fixo e Flutuante

Para obter a sobreposição das unidades de ponto fixo e ponto flutuante vários problemas devem ser solucionados. Um deles, é como sincronizar as unidades de ponto-flutuante e ponto-fixo para manter as interrupções precisas e continuar tendo um grau elevado de processamento de instruções. Uma interrupção é precisa se a partir do momento em que ela estiver sendo processada nenhuma instrução subseqüente inicia sua execução e todas as instruções anteriores já foram completadas. O segundo deles, é como permitir que cargas de ponto-flutuante prossigam quando o estado do registrador de ponto-flutuante é desconhecido para a unidade de ponto fixo. O terceiro problema, é como assegurar que o dado correto foi carregado no registrador de ponto-flutuante ou armazenado do registrador de ponto-flutuante, tendo a unidade de ponto-fixo efetuando os cálculos de endereços para carga e armazenamento de ponto-flutuante. Detalhes sobre como estas questões foram solucionadas são encontradas em [GROHOSKI 90]; elas não são apresentadas neste texto devido a extensão e complexidade das soluções propostas.

3.4 Manutenção da Consistência do Conjunto de Instruções

Em uma arquitetura de máquina com um grau elevado de sobreposição, interrupções podem ser precisas ou imprecisas. Interrupções precisas, conforme definido na seção 3.3, forçam a máquina a preservar a visão de uma arquitetura que executa uma instrução de cada vez, finalizando-a antes de iniciar a execução da próxima. Interrupções imprecisas permitem ao processador deixar um conjunto de instruções em um estágio fragmentado, porém recuperável na vizinhança da interrupção. Por exemplo, instruções anteriores podem ter sido executadas e algumas instruções subseqüentes podem ter iniciado a execução e atualizado alguns registradores. Interrupções imprecisas necessitam que a arquitetura providencie uma forma para reconstruir o conjunto de instruções em torno do ponto da interrupção, de tal forma que o processamento de *software* pós-interrupção possa recriar o estado seqüencial da máquina, possibilitando-a reexecutar as instruções. Devido a complexidade de *pipeline* da arquitetura `AMERICA` seria difícil projetar um esquema para manusear interrupções no modelo impreciso, pois esta deveria levar em consideração as várias possibilidades para a execução de instruções após o ponto de interrupção. Portanto, interrupções precisas foram especificadas para todas as interrupções geradas pelo programa. Para garantir interrupções precisas, foi analizado cada tipo de interrupção e a partir da análise foi desenvolvido um método de solução para cada uma delas. Por exemplo, se uma instrução no *pipeline* ocasiona uma interrupção, esta interrupção é resolvida. Interrupções externas e assíncronas foram solucionadas parando o despacho de instruções e esperando o escoamento do *pipeline*.

3.5 Adaptações Elaboradas no Projeto do Processador do Sistema RISC/6000

Durante a implementação do processador do RS/6000 foram efetuadas várias modificações importantes em sua especificação. Estas mudanças estão relacionadas com a sincronização das unidades de ponto fixo e ponto flutuante, com o registrador de condição apresentado na seção 2.3.1 e com o mecanismo de carga no cache de instruções.

3.5.1 Cache de Instruções

Como especificado originalmente na seção 3.1, o mecanismo de carga de instruções poderia carregar quatro instruções por ciclo desde que elas estivessem dentro das treze primeiras instruções de uma linha do cache de um total de 16. Ao implementar o acesso lógico ao arranjo, notou-se que o mesmo princípio de distribuição (interleaving principle) poderia ser aplicado para o diretório de caches. Dividindo o diretório de cache em componentes pares e ímpares e fornecendo um mecanismo para implementação das linhas ao diretório par, quatro instruções poderiam ser carregadas mesmo se duas linhas de cache estivessem cruzadas. Se a primeira instrução do grupo estivesse em uma linha par do cache, o diretório

ímpar poderia ser acessado com o mesmo endereço da linha para pesquisar pelas instruções restantes na linha sucessora do cache. Da mesma forma, se a primeira instrução estivesse em uma linha ímpar do cache, a próxima linha do cache estaria contida na próxima classe de congruência no diretório par, requerendo que o endereço da linha fosse incrementado. Esta característica complicou a lógica da arquitetura e duplicou algumas comparações, mas foi introduzida sem sacrificar o tempo do ciclo. Como resultado, assumindo que ambas as linhas estão presentes no cache, quatro instruções semtre podem ser carregadas desde que as linhas estejam na mesma página virtual de 4-kbytes.

3.5.2 Registrador de Condição

O registrador de condição foi o primeiro registrador a ser modificado na arquitetura do 801 para refletir a execução concorrente de várias instruções.

O registrador de condição (CR) contém oito campos, cada um deles encarregado de conter os resultados de uma operação do tipo *compare*. Como o registrador de condição possui um bit de bloqueio por campo é possível manter oito operações pendentes. Quando uma instrução é disparada para as unidades de ponto fixo ou ponto flutuante, que liga o campo 0 do registrador de condição, o bit de bloqueio correspondente é acionado. Instruções subseqüentes que tentam ler ou ligar este campo permanecem no *buffer* de instruções. Eventualmente a unidade de ponto flutuante ou ponto fixo executa a instrução e a ICU (*Instruction Cache Unit*) é informada via o barramento do registrador de condição daquela unidade. Então o bit de bloqueio é desligado. Como instruções que ligam o registrador de condição podem ser disparadas condicionalmente, e canceladas, qualquer bit de bloqueio do registrador de condição que foi ligado deve ser desligado.

Durante a implementação, a quantidade de espaço dedicado para a solução dos bloqueios no registrador de condição tornou-se muito grande. Portanto, foram introduzidos somente quatro bits de bloqueio, de tal forma que, quatro operações pendentes poderiam ser mantidas no registrador de condição. A seção 4.4 apresenta um exemplo que ilustra o uso dos bits de bloqueio.

4 Escalonamento de Instruções

Algoritmos de escalonamento de instruções são utilizados em compiladores para reduzir os atrasos atribuídos ao tempo de execução no código compilado. Esta redução pode ser particularmente vantajosa em arquiteturas de máquinas *pipeline*, visto que tais arquiteturas permitem um grau mais elevado de sobreposição durante a execução de instruções. Por exemplo, se há um atraso de um ciclo entre a carga e o uso de um valor `V`, é desejável "cobrir" este atraso com uma instrução que seja independente de `V` e que esteja pronta para ser executada. Portanto, uma simples reorganização ou transformação dos comandos de um programa, normalmente em linguagem intermediária ou código de montagem pode ser considerada como uma forma de explorar o paralelismo potencial inerente no código.

O processador RS/6000 é capaz de executar quatro instruções a cada ciclo de máquina. Se a instrução multiplica-soma de ponto flutuante for considerada como duas instruções, então o processador RS/6000 é capaz de executar cinco instruções em um ciclo. Entretanto sempre ocorrem atrasos ou espera entre instruções devido a comunicação entre *chips* e devido a natureza do *pipeline* da unidade de ponto flutuante. Por exemplo, na execução de uma instrução de carga existe uma espera de um ciclo porque o dado deve ser carregado do *cache* de dados e o mesmo não está localizado no *chip* da unidade de ponto fixo. O atraso no *pipeline* da unidade de ponto flutuante ocorre quando o resultado de uma instrução de ponto flutuante é necessário logo a seguir para a próxima instrução. Esta unidade necessita de dois ciclos para completar a execução de uma única instrução, mas se as instruções são independentes, ou seja, se o resultado de uma não for necessário à próxima instrução, a unidade de ponto flutuante pode executar um conjunto de instruções a uma velocidade de uma por ciclo.

As seções posteriores discutem os mecanismos de escalonamento de instruções necessários para o processador RS/6000.

4.1 Alternância de Ponto Fixo e Ponto Flutuante

As unidades de ponto flutuante e ponto fixo do sistema RISC/6000 operam em paralelo, e cada uma delas é capaz de executar uma instrução por ciclo. A unidade de desvio é capaz de disparar duas instruções por ciclo. Ambas as instruções são enviadas para as duas unidades de execução, e cada unidade descarta a instrução que não é apropriada para ela. Isto significa que para obter um bom desempenho do sistema, a unidade de desvio deve enviar uma instrução de ponto fixo e uma instrução de ponto flutuante a cada ciclo. Naturalmente, como existem instruções que utilizam-se de multiciclos tais como a instrução de dividir de ponto flutuante (9 ciclos) e a instrução de multiplicação de ponto fixo (3 a 5 ciclos), é difícil descrever uma boa ordem para uma longa seqüência de instruções. Entretanto, considerando o armazenamento (*buffering*) nas unidades aritméticas, uma solução adequada seria, simplesmente, alterná-las. Por exemplo, se `F` representa uma instrução

de ponto flutuante, e X denota uma instrução de ponto fixo, a seqüência de instruções FXXFFXX seria uma ordem desejável.

4.2 Atraso de Carga

Existe um atraso de um ciclo entre a carga de um dado de ponto fixo e o seu primeiro uso na unidade de ponto fixo. Para evitar este atraso uma solução seria colocar, se possível, uma instrução de ponto fixo independente entre as duas. Uma instrução independente neste caso seria aquela que não se utiliza de um dado carregado. Entretanto, não havendo uma instrução deste tipo, não há necessidade de inserir uma instrução "no-op" porque o próprio *hardware* possui mecanismos de bloqueios nos registradores e irá esperar quando necessário. Um atraso também ocorre entre uma carga e a primeira instrução utilizada para atribuir o mesmo registrador como objetivo (target) da carga. Mas esta é uma seqüência de código pouco comum porque o item carregado é descartado.

Não há atraso para uma carga de ponto flutuante se o uso for de uma instrução de armazenamento de ponto flutuante. Para outros usos, normalmente, também não há atrasos porque as cargas para ponto flutuante são processadas pela unidade de ponto fixo, e esta unidade, geralmente, executa à frente da unidade de ponto flutuante.

Devido à necessidade de alternância entre as instruções de ponto fixo e ponto flutuante poderia haver também duas instruções de ponto flutuante entre uma carga de ponto fixo e seu primeiro uso. Por exemplo, se L representa uma carga, X denota uma instrução de ponto fixo independente, Y representa a instrução de ponto fixo que L supre e F denota uma instrução de ponto flutuante, uma ordem da forma LFXFY seria ótima. As mesmas considerações apresentadas nesta seção aplicam-se aos atrasos de ponto flutuante.

4.3 Atrasos entre Definição e Uso de Instrução de Ponto Flutuante

Há um atraso de um ou dois ciclos entre qualquer instrução de ponto flutuante e o primeiro uso do valor computado na unidade de ponto flutuante. O atraso é de dois ciclos se a primeira instrução alimenta a posição FRA (bits 11-15) da segunda, senão é de 1 ciclo. Para evitar este atraso deve ser inserido, se possível, uma ou duas instruções independentes de ponto flutuante entre a definição e seu primeiro uso. Várias instruções de ponto flutuante possuem outros campos que podem ser permutados com FRA sem que se mude o efeito da instrução. Portanto o atraso extra pode ser evitado trocando o campo apropriado. Os outros campos são: FRB para *floating-add* e FRC para a família de instruções *floating-multiply* e *floating-multiply-add.*

4.4 Atrasos Causados pela Instrução Composta `compare-desvie`

Esta seção discute os principais efeitos dos atrasos associados com os desvios condicionais. Existe um atraso de três ciclos entre uma instrução de comparação de ponto fixo e um desvio condicional se o desvio é efetuado, e, não há nenhum atraso caso contrário. Para a instrução *compare* de ponto flutuante, o atraso passa a ser de seis ciclos se o desvio acontece e de três, caso contrário. Estes atrasos, portanto, ocorrem em ambas as unidades de ponto fixo e ponto flutuante.

Para entender melhor o período de tempo do desvio condicional é interessante entender como a unidade de desvio opera. Ao disparar instruções, esta unidade as decodificam o suficiente para saber se as mesmas utilizam ou modificam o registrador de condição e em caso positivo, ela indica qual campo do registrador de condição `CR` é afetado. Por exemplo, quando a unidade de desvio dispara uma instrução `compare`, ela bloqueia o campo alvo de `CR`. Quando ela envia uma instrução com o *record bit* ligado, ela bloqueia o `CR0` ou o `CR1`, dependendo se a instrução é de ponto fixo ou ponto flutuante. Mais tarde, quando a instrução é executada pela sua unidade de execução, a unidade de desvio recebe o valor de `CR` e desbloqueia o seu campo.

O desempenho do sistema pode degradar substancialmente pelo uso indiscriminado do *record bit* devido ao bloqueio do `CR`. Portanto, o *record bit* deve ser modificado somente quando o resultado contido em `CR0/CR1` for realmente necessário.

Quando a unidade de desvio executa uma instrução de desvio condicional ela espera, se necessário, pelo campo testado de `CR` ser desbloqueado. Contudo, se o campo de `CR` está bloqueado, esta unidade "condicionalmente dispara" duas instruções. Isto é suficiente para manter a unidade de ponto fixo ocupada se o desvio ocorre, e se as duas instruções disparadas condicionalmente e as duas instruções seguintes a elas são todas instruções de ponto fixo.

Caso apareça uma situação em que, quando um campo de `CR` é bloqueado seja encontrada uma outra instrução que liga este mesmo campo, a unidade de desvio pára de enviar instruções porque ela não pode administrar duas ou mais instruções pendentes que acionam o mesmo campo de `CR`. O programador deve, portanto, evitar tal situação. Por exemplo, considere a seqüência de instruções:

```
andil     ...  (liga CR0 implicitamente)
cmp cr0   ...  (também liga CR0)
bc cr0    ...  (desvio usa CR0)
```

Ambas as instruções `andil` e `cmp` ligam `CR0`, portanto haverá um atraso de três ciclos entre elas. Existe um atraso adicional de três ciclos entre as instruções `cmp` e `bc` se o desvio é efetuado. O programador pode evitar o desvio desnecessário utilizando um campo diferente de `CR` para as instruções `cmp-bc`.

Os campos do registrador CR são "emparelhados" devido ao bloqueio. Os pares são CR0-CR4, CR1-CR5, CR2-CR6, CR3-CR7. Quando uma instrução bloqueia CR0, por exemplo, a unidade de desvio bloqueia também CR4. Assim, para a seqüência mostrada no exemplo acima, o programador deve utilizar um outro campo de CR que não seja 0 ou 4 para a seqüências de instruções cmp-bc.

4.5 Armazenamentos Excessivos de Ponto Flutuante

Uma instrução de armazenamento de ponto flutuante, normalmente, gasta um ciclo de máquina em ambas as unidades, de ponto fixo e ponto flutuante. Porém, quando há mais de quatro instruções *excessive floating-point stores* (stfs) consecutivas pode haver um atraso de um ciclo para cada stf na unidade de ponto fixo após a quarta instrução. Este atraso é atribuído ao fato de que a unidade de ponto fixo é capaz de computar endereços mais rápido do que a unidade de ponto flutuante pode calcular os valores a serem armazenados. Existe uma fila "de armazenamentos pendentes" de comprimento quatro que detém os endereços virtuais dos dados de ponto flutuante a serem armazenados. Se a fila está cheia, futuros stfs aguardam até que haja espaço disponível na fila.

4.6 Atrasos Devido ao Uso da Seqüência store-load-use

O atraso *load-use* é maior devido a presença de um armazenamento precedendo a carga com o endereço efetivo referindo-se aos mesmos 16-bytes do bloco de memória. O comprimento do atraso está vinculado ao tipo de instrução: ponto fixo ou de ponto flutuante. Na tabela apresentada abaixo, ST e L representam, respectivamente, instruções de armazenamento e carga de ponto fixo. STF e LF denotam, respectivamente, instruções de armazenamento e carga de ponto flutuante.

ST-L-use:3	ST-LF-use: 2
STF-L-use: 5	STF-LF-use: 5

Para L, o atraso ocorre na unidade de ponto fixo e para LF o atraso ocorre na unidade de ponto flutuante. Isto acontece entre a carga e o uso do dado carregado, mas também pode ser melhorado incluindo instruções entre o armazenamento e a carga ou entre a carga e seu uso.

As seqüências de instruções STF-L e ST-LF são importantes porque elas são utilizadas para conversões entre instruções de ponto fixo e ponto flutuante.

4.7 Outros Atrasos

4.7.1 Atrasos Devido ao Uso das Instruções `MFSPR` e `MFCR`

Existe um atraso de um ciclo entre a instrução *move from special-purpose register* (MFSPR) ou *move from condition register* (MFCR) e o primeiro uso do dado movido para um registrador de propósito geral na unidade de ponto fixo. O comprimento do atraso é o mesmo, independente de qual seja o registrador de propósito especial.

4.7.2 Atrasos `MTLR-BR`

A seqüência de instrução *move to link register* seguida por *branch (un)conditional register* possui um atraso de quatro ciclos. Esta seqüência é freqüentemente utilizada para retorno de subrotina e para desvios para metas que são variáveis.

4.7.3 Atrasos Devido ao Uso da Sequência `MTCTR-BCT`

Existe um atraso de quatro ciclos entre a instrução *move to count register* e um `BC` que utiliza o registrador contador.

4.7.4 Atrasos Relacionados com Instruções que Utilizam `CR-logic`

Faz parte do conjunto de instruções do RS/6000 o conjunto completo de instruções lógicas para manipulação dos bits de CR. Estas instruções são executadas pela unidade de desvio. A presença de várias instruções consecutivas deste tipo pode ocasionar atrasos nas unidades de execução porque a unidade de desvio não tem condições de disparar instruções suficientemente rápido.

Uma seqüência alternando instruções de ponto fixo e ponto flutuante e tendo mais de duas instruções `CR-logic` consecutivas causa um atraso em cada unidade aritmética. Já uma seqüência de instruções de ponto fixo possuindo mais que cinco instruções *CR-logic* consecutivas causa um atraso na unidade de ponto fixo. Não há atraso para uma instrução *CR-logic* seguida imediatamente por um desvio condicional sobre o conjunto de bits.

4.8 Outras Considerações sobre Escalonamento de Instruções

4.8.1 Minimizando *liveness*

O termo *liveness* denota o tempo de vida, isto é, o tempo em que um registrador possui determinado valor que ainda é necessário.

O escalonamento de instruções tende aumentar a demanda por registradores, portanto é importante não exagerar (*overscheduling*). Ou seja, instruções de carga, geralmente, não devem ser postas mais distante do que o necessário. Por outro lado, a possibilidade de perda de cache sugere exatamente o contrário, ou seja, que instruções de carga sejam colocadas o mais distante possível. Esta é uma decisão difícil de ser tomada, portanto deve ser encontrado um ponto de equilíbrio. Por exemplo, suponha um bloco básico consistindo em três pares de instruções `carrega-armazena` antes da alocação de registradores. Três ordens possíveis são consideradas como ilustrado abaixo. Neste exemplo, `r101` e os demais registradores utilizados representam registradores simbólicos.

L	r101,...	L	r101 ...	L	r101, ...
ST	r101,...	L	r102, ...	L	r102, ...
L	r102,...	L	r103, ...	ST	r101, ...
ST	r102, ...	ST	r101, ...	L	r103, ...
L	r103, ...	ST	r102, ...	ST	r102, ...
ST	r103, ...	ST	r103, ...	ST	r103, ...
	(a)		(b)		(c)

A seqüência (a) mostra como o código, provavelmente, seria arrumado antes do escalonamento de instruções, assumindo que ele foi produzido por três comandos de atribuição consecutivos no programa fonte. Esta ordem gasta nove ciclos para ser executada devido ao atraso de um ciclo entre cada instrução de carga e de armazenamento. A seqüência (b) executa esta seqüência em seis ciclos, mas ela sofre de *overscheduling* porque o alocador de registradores terá que atribuir três registradores físicos distintos para `r101`, `r102` e `r103`. A seqüência (c) também executa a seqüência em seis ciclos, mas requer somente dois registradores físicos, porque `r101` e `r103` podem ser atribuídos para o mesmo registrador físico. Portanto, (c) é a melhor ordem para a seqüência apresentada. Este exemplo também mostra a importância do escalonamento antes da alocação de registradores. Se o escalonamento não tivesse sido feito, seqüência (a) seria a entrada para o alocador de registradores, e ele provavelmente alocaria `r101`, `r102` e `r103` a um mesmo registrador físico, prevenindo qualquer rearranjo.

4.8.2 Evitando Mudanças Semânticas

Para manter o código semanticamente válido, certas movimentações devem ser proibidas. Por exemplo, instruções de ponto flutuante não devem ser postas além das chamadas de subrotinas porque elas ligam os bits de estado do *floating-point status and control register fields* (FPSCR) e a chamada de subrotina pode testá-lo. Instruções de ponto fixo que não sejam de carga e armazenamento podem ser mudadas livremente desde que o bit de OE esteja desligado. Caso ele esteja ligado, a movimentação é proibida devido ao bit de *overflow* sumário na XER.

5 O Algoritmo de Escalonamento

5.1 O Algoritmo Básico

O algoritmo de escalonamento proposto por Warren [WARREN 90] e utilizado nos compiladores `XL` e `PL.8` é baseado em grafos de dependência que são construídos para cada bloco básico. O grafo de dependência possui um nodo para cada instrução e um arco direcionado entre dois nodos se uma das instruções deve preceder a outra por alguma razão. Caso a primeira instrução compute algum valor utilizado pela segunda, aquela é denominada "dependência à frente". Caso a segunda instrução altere um registrador ou uma posição de memória utilizada pela primeira instrução, aquela é denominada "dependência reversa". A maior parte dos atrasos ocorrem devido a "dependência à frente". Os arcos no grafo de dependência são rotulados com o valor do atraso ou a distância entre instruções.

A Figura 3 apresenta um bloco básico contendo o código fonte para aritmética de ponto fixo e o código em linguagem intermediária correspondente, antes da alocação de registradores.

```
A = B + C - D;
IF E > 0 THEN  ...


L     r100,  B(r200)
L     r101,  C(r200)
ADD   r102,  r100, r101
L     r103,  D(r200)
SUB   r104,  r102, r103
ST    r104,  A(r200)
L     r105,  E(r200)
CMP   r106,  r105,0
BC    r106,  ...
```

Figura 3: Exemplo de um bloco básico.

O grafo de dependência correspondente à Figura 3 é apresentado na Figura 4. Os literais `L.B`, `L.C` etc. representam a carga de `B`, `C`, etc. Muito embora a instrução `ST` não tenha dependência à frente ou para trás com a instrução de desvio condicional `BC`, um arco é colocado entre elas para refletir o fato de que a instrução `ST`, aliás, todas as instruções no bloco básico devem ser executadas antes da instrução `BC`, porque `BC` delimita o bloco básico.

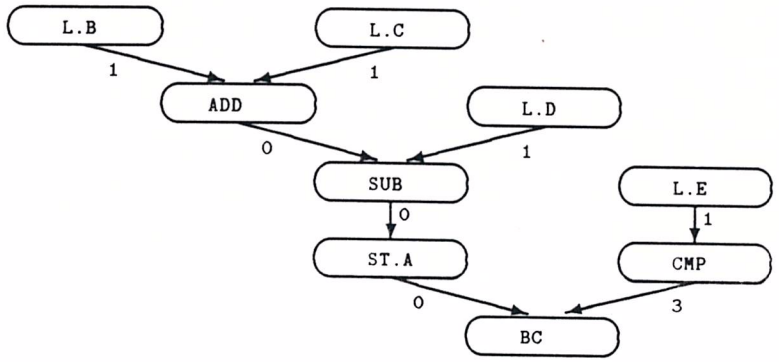


Figura 4: Grafo de dependência referente à Figura 3

O escalonador constrói o grafo de dependência examinando todos os $(n^2 - n)/2$ pares nas n-instruções do bloco básico. Para cada par de instruções, examinam-se os registradores para verificar quando uma instrução modifica ou não um registrador que outra instrução também modifica ou usa. Neste processo é utilizado uma técnica de *hash* para acelerar o casamento de registradores, contudo a construção do grafo de dependência continua sendo a parte do escalonador que consome mais tempo.

Classes de memória e outras informações do dicionário são utilizadas para verificar quando instruções não podem referenciar a mesma posição de memória. Por exemplo, classes estáticas e automáticas não podem sobrepor-se. Caso elas estejam na mesma classe ou em classes sobrepostas, o registrador base e o deslocamento são examinados para ver se elas referem-se a localizações distintas de memória.

Após a construção do grafo de dependência, ele é percorrido de baixo para cima e cada nodo é rotulado com o atraso máximo a partir do nodo em questão até o final do bloco básico. Este cálculo é efetuado somando-se o tempo de atraso a partir do final do bloco básico até cada nodo. Se dois ou mais caminhos convergirem para o mesmo nodo utiliza-se o valor máximo calculado ao longo do caminho. Em seguida instruções são selecionadas, basicamente, decrementando o tempo de atraso a partir de cada nodo até o final. De acordo com a Figura 4, as quatro instruções de "carga" são elegíveis porque elas não possuem predecessores no grafo de dependência. A instrução com o maior atraso em relação ao final do bloco básico é escolhida primeiro. Observando a Figura 4 novamente, nota-se que a carga de E possui um atraso de quatro ciclos até o final do seu caminho enquanto as outras três cargas acumulam somente um ciclo de atraso. Portanto **E** é escolhido.

À medida que instruções são selecionadas, o algoritmo atualiza o valor do "tempo corrente", incrementando-o com o tempo de execução da instrução que acabou de ser selecionada. Inicialmente o "tempo corrente" é zero. O tempo de execução, normalmente, é de um ciclo para a maior parte das instruções. A instrução selecionada é removida

do grafo de dependência e suas sucessoras imediatas no grafo são marcadas com o valor do tempo igual ao "tempo corrente" atualizado mais o atraso da instrução que acabou de ser selecionada para sucedê-la. Este valor do *earliest time* é utilizado para afastar as instruções, se possível, até depois que o requisitado atraso tenha decorrido. O algoritmo continua, e a ordem final para o exemplo mostrado na Figura 3 é apresentado abaixo. O algoritmo encontrou uma ordem ótima. Não há atrasos descobertos, e o código abaixo é executado em oito ciclos no processador RS/6000.

```
L     r105,  E(r200)
L     r100,  B(r200)
CMP   r106,  r105,0
L     r101,  C(r200)
L     r103,  D(r200)
ADD   r102,  r100, r101
SUB   r104,  r102, r103
ST    r104,  A(r200)
BC    r106,  ...
```

A estratégia apresentada para escolher a primeira instrução que está mais distante do final, em termos do atraso acumulado ao longo de seu caminho, é *puramente heurístico* e nem sempre obtem a melhor ordem. Por exemplo, esta estratégia falha se considerarmos o exemplo mostrado na Figura 3 omitindo a carga de `E` e `D`. Neste caso, o algoritmo iria escolher primeiro a instrução `CMP`. Logo após, ele seria forçado a escolher as duas instruções de carga seguida da instrução `ADD`. Isto acarretaria um atraso descoberto entre a segunda instrução de carga e a instrução `ADD`. Entretanto a melhor ordem colocaria as duas instruções de carga primeiro, e então escolheria a instrução `CMP` como cobertura para a segunda instrução de carga. Depois escolheria as instruções `ADD`, `SUB`, `ST` e `BC`. Neste ponto, já existe um número suficiente de instruções para cobrir o atraso de 3 ciclos entre as instruções `CMP-BC`, portanto não há atraso com esta ordem.

```
L     r100,  B(r200)
L     r101,  C(r200)
CMP   r106,  r105,0
ADD   r102,  r100, r101
SUB   r104,  r102, r103
ST    r104,  A(r200)
BC    r106,  ...
```

5.2 Refinamentos Efetuados no Algoritmo Básico

O esquema completo de seleção proposto por Warren é descrito nesta seção. Para facilitar a exposição é apresentado o processamento por subconjunto de instruções, muito

embora seja implementado buscando o restante das instruções e escolhendo a melhor para selecionar. A busca é efetuada uma vez para cada instrução selecionada.

1. Inicialize o conjunto de todas as instruções que ainda não foram selecionadas e que não possuam predecessores no grafo de dependência. Estas são as instruções elegíveis.
2. Identifique o subconjunto daquelas instruções cujo *earliest time* tenha sido alcançado, ou, se nenhuma, aquelas com o menor *earliest time*.
3. Se uma ou mais instruções tiverem sido selecionadas, e se o subconjunto corrente contiver uma ou mais instruções de tipos opostos, ou seja, ponto fixo ou ponto flutuante, a partir da última instrução selecionada, então identifique o subconjunto corrente para aquelas do tipo oposto.
4. Identifique o subconjunto daquelas instruções contendo o total máximo de atraso ao longo do caminho de cada instrução até o final do bloco básico.
5. Identifique o subconjunto daquelas instruções possuindo mínimo *liveness weight*.
6. Identifique o subconjunto para a única instrução que veio primeiro na ordem original.
7. Selecione uma única instrução que está no subconjunto neste ponto, e então repita este processo até que todas as instruções no bloco básico tenham sido selecionadas.

O critério *liveness weight* é utilizado como um esquema simples para reduzir *register spills*. Nele cada instrução é assinalada com um "peso" da seguinte forma:

0 - Movimentação de registradores (ponto-fixo ou ponto flutuante)

1 - Instruções sem registradores-meta (*stores e traps*)

2 - Maior parte das instruções

3 - Cargas de memória

4 - Instruções sem registradores-fonte, por exemplo *load immediate*

As instruções com pesos menores são selecionadas primeiro. A justificativa para esta classificação deve-se a alguns fatos: a) instruções de movimentação de registradores possuem menor peso porque o alocador de registradores efetua um melhor trabalho de união (*coalescing*) se estes são mantidos perto das instruções que definem o fonte do *move-register*; b) as instruções de *stores* e *traps* são selecionadas à frente porque elas possuem "usos" e não "definições". Portanto, elas podem liberar registradores, mas elas nunca aumentam o número de registradores que estão atualmente ativos; c) similarmente, uma instrução

do tipo *load immediate* é selecionada mais tarde porque ela aumenta o número de registradores que estão ativos e não libera nenhum. Os pesos são distribuídos em relação às chamadas de subrotinas. Ou seja, instruções entre o início de um bloco básico e o primeiro *call* são atribuídos pesos entre `0` e `4`. Aquelas instruções entre este *call* e o próximo têm seus pesos acrescidos de `10`. Isto tem o efeito de reduzir a movimentação de instruções entre *calls*.

6 Escalonamento Além de Blocos Básicos

Pesquisas recentes sobre escalonamento de instruções para máquinas *pipeline* têm focalizado primeiramente em otimizações dentro de blocos básicos. Várias considerações foram apresentadas neste texto para o escalonamento de código dentro dos mesmos. Tais escalonamentos são bem eficazes para programas possuindo longos blocos, característica comum nas aplicações científicas. Instruções de desvio, entretanto, restringem a eficácia das arquiteturas *pipeline* de formas que não podem ser solucionadas aplicando transformações apenas dentro dos blocos básicos. O objetivo desta seção é ampliar este leque apresentando técnicas que permitam o escalonamento além dos blocos básicos possibilitando uma redução adicional aos atrasos de tempo de execução. Nas técnicas apresentadas a seguir assume-se que está disponível um algoritmo de escalonamento de instruções para blocos básicos, como aquele apresentado na seção 5.1 proposto por Warren [WARREN 90], e transformações adicionais são desenvolvidas para auxiliar na remoção dos atrasos restantes oriundos dos desvios e dos *loops*. O efeito do atraso entre uma instrução de comparação e um subseqüente desvio condicional, que ocorre somente se o desvio é efetuado sugere uma série de estilos diferentes de apresentação de *loops* e duplicação de código.

6.1 Transformações em `loops` para Redução de Atrasos

Para eliminar o atraso, parcialmente ou completamente, oriundo da efetivação do desvio do par de instruções `CMP-BC`, a estratégia utilizada pelo escalonador consiste na colocação de uma ou mais instruções entre a instrução *compare* (CMP) e a instrução de desvio condicional (BC) para cobrir o atraso existente. Se o tempo total `t` gasto na execução destas instruções for de três ciclos ou mais, nenhum atraso ocorre ao longo de qualquer caminho tomado. Entretanto, se `t` for menor que três ciclos, um atraso de (3 - t) ciclos ocorre se o desvio for efetuado. De maneira similar, a instrução tripla, `CMP-BC-B` produz um atraso de quatro ciclos se o desvio não é efetuado.

Na maioria dos compiladores, incluindo o compilador `PL-8`, a condição de saída de um *loop* é verificada ao final do mesmo. Se o par `CMP-BC` somente é parcialmente coberto pelo escalonador de bloco básico, o atraso ocorrerá para cada iteração do *loop*. Uma maneira simples para eliminar este atraso pode ser gerar *loops* de acordo com o esquema apresentado na Figura 5 (b).

Na abordagem da Figura 5 (b) verifica-se a negação da condição de saída do `loop` no início do mesmo e adiciona um desvio incondicional (B) ao final do corpo do *loop*. Neste caso, o atraso pode ocorrer somente após sair do *loop*, ao custo de duplicar o seu corpo. Com a presença da tripla `CMP-BC-B`, pode ser necessário cobrir o par `CMP-BC` com instruções movidas acima retiradas do corpo do *loop*. O escalonador de bloco básico cujo escopo é limitado a um único bloco básico não será capaz de cobrir, mesmo que parcialmente o par `CMP-BC` porque ele é um outro bloco básico composto de dois comandos. Da mesma forma, a geração de *loops* como ilustrado na Figura 5 (c), para favorecer o caminho seqüencial do par `CMP-BC` com um atraso de tamanho "zero" é necessário cobrir a tripla `CMP-BC-B` por causa do desvio condicional seguido pelo desvio incondicional.

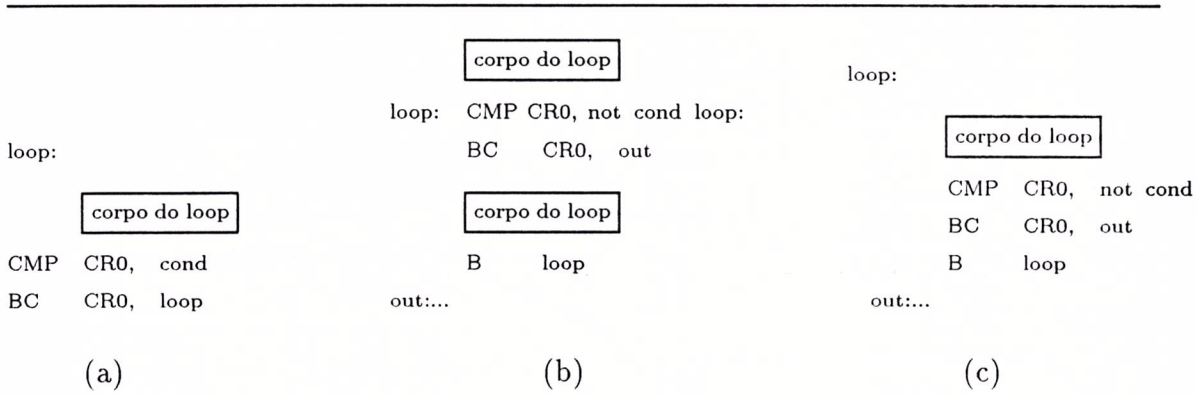


Figura 5: Três esquemas para cobrir "`CMP-BC`" usando escalonador de bloco básico

A abordagem sugerida por Golumbic e Rainish [GOLUMBIC 90] assume um escalonador de bloco básico sofisticado que aceita *loops* gerados de acordo com o exemplo apresentado na Figura 5 (a) e prossegue cobrindo pares `CMP-BC` o máximo possível dentro do escopo do bloco básico. A figura 6 (a) apresenta o *loop* após o escalonador de bloco básico ter tentado cobrir o atraso. Se o par `CMP-BC` for somente parcialmente coberto (t<3), então efetue as seguintes transformações *loop-code replication*:

1. Crie novo rótulo (out) antes do primeiro comando seguindo o *loop* (se não existe nenhum).
2. Copie as primeiras `4-t` instruções a partir do início do `loop` para depois de BC (se o número `n` de instruções do início do *loop* é menor que `4-t`, então copie somente `n` instruções).
3. Crie novo rótulo, *loop1*, após a instrução número (4 - t).
4. Negue a condição de `CMP`, troque o rótulo de `BC` e adicione um desvio incondicional `B` seguindo as instruções copiadas como ilustradas no esquema da Figura 6 (b).

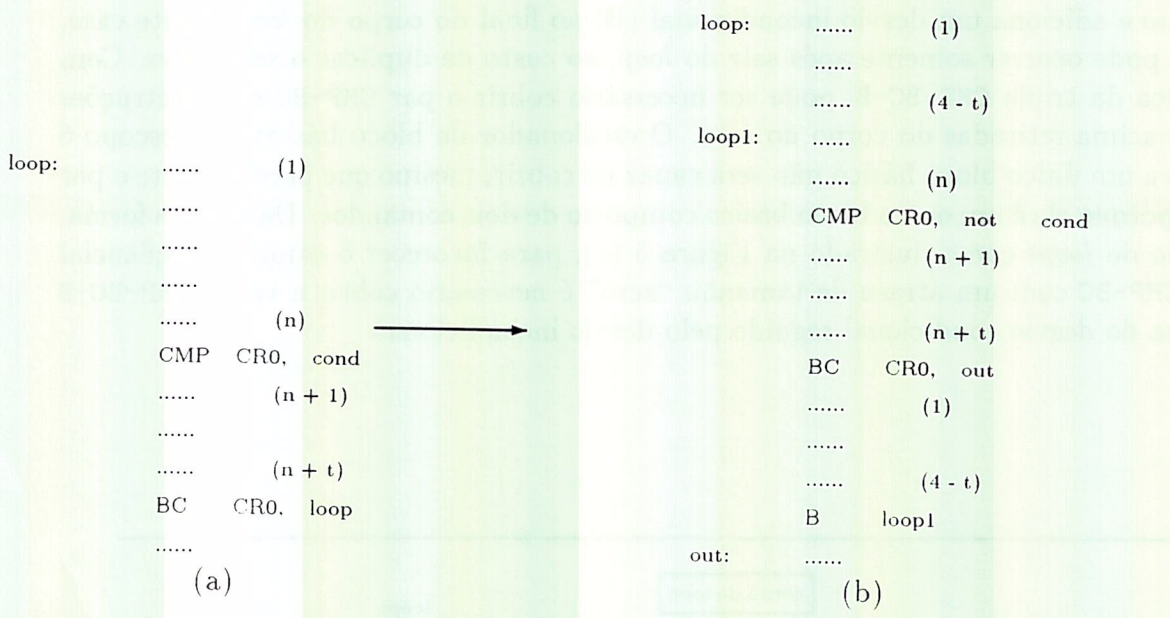


Figura 6: Transformação usando duplicação de código para cobrir o par `CMP-BC`

Do ponto de vista de alocação global de registradores tais transformações poderiam ser efetuadas tanto antes como depois da alocação de registradores. Entretanto, devido a outras otimizações [BERNSTEIN 89] é preferível efetuá-las após a alocação.

Após as transformações mostradas na Figura 6 (b) serem efetuadas, o rótulo `loop` pode se tornar um rótulo inútil, neste caso, ele é eliminado. "Arranjos" adicionais podem fundir alguns dos blocos básicos. Neste ponto, a propagação de constantes e o escalonamento no bloco básico pode ser executado novamente nestes novos blocos para obter outros melhoramentos.

6.2 Comandos `IF-THEN-ELSE`

A instrução tripla `CMP-BC-B`, que produz um atraso de 4 ciclos se o desvio não for efetuado aparece freqüentemente com as instruções geradas para os comandos `if-then-else` tendo parte do `then` pequena. Na seqüência de instruções apresentadas na Figura 7 um atraso de três ciclos ocorre se o desvio condicional não é efetuado, uma vez que a tripla `CMP-BC-B` é coberta por apenas uma instrução `ADD a,a,1`. Um atraso de três ciclos também ocorre se o desvio condicional é efetuado, assumindo que o bloco do rótulo `out` é longo o suficiente, ou seja, três ou quatro instruções . É possível reconhecer tal sequência na linguagem intermediária e transformá-la revertendo os papéis dos comandos `then` e `else` como ilustrado na Figura 8. Esta transformação pode facilmente ser efetuada após o assinalamento dos registradores, o que torna a seqüência de código mais realista para ser analisada.

código fonte			*instruções geradas*	
if(condition)	then		CMP	CRO,cond
do;			BC	CR0,else
	a=a+1;		ADD	a,a,1
end;			B	out
else		else:	ADD	a,a,b
do;			ADD	a,a,c
	a=a+b+c+d+e;		ADD	a,a,d
end;			ADD	a,a,e
out:.....		out:	.....	
.....			.....	

Figura 7: Seqüência `if-then-else` com `then` curto

antes da transformação				*após a transformação*			
	CMP	CR0,	cond		CMP	CR0,	not cond
	BC	CR0,	else		BC	CR0,	then
then:	.....			else:	.....		
	B	out			.....		
else:	.....				.....		
	.....				.....		
	.....				B	out	
	.....			then:	.....		
out:	.....			out:	.....		
	.....				.....		

Figura 8: Seqüência mostrada na Figura 7 após transformação

6.3 Colagem de Blocos para Acelerar Comandos `if-then-else`

Para comandos possuindo ambas as partes `short then` e `short else` a transformação anterior não se aplica. Por exemplo,

if[a(i)>0] then count1 = count1 + 1;
else count2 = count2 + 1;

produz um atraso de 3 ciclos se `a(i)>0` devido a tripla `CMP-BC-B`, e um atraso de 3 ciclos por causa do par `CMP-BC`. Porém, em certas situações, há possibilidades de diminuir este atraso efetuando outro tipo de transformação, denominada colagem *gluing*. Nesta

transformação, copia-se o bloco básico seguindo o comando `if` e criam-se dois novos blocos, `new-then` e `new-else`. Após tal transformação, o atraso ao longo do caminho meta permanece como antes, mas o atraso ao longo do caminho seqüencial é reduzido, e provavelmente desaparece. A Figura 9 mostra um exemplo desta transformação.

antes da colagem		*após a colagem*	
	CMP CR0, cond		CMP CR0, cond
	BC else		BC new-else
then:	instruction-then	new-then:	instruction-then
	B out		instruction-out-1
else:	instruction-else		.....
out:	instruction-out-1		.....
	.....		.....
	.....		instruction-out-n
	.....		B out1
	.....	new-else:	instruction-else
end-out:	instruction-out-n		instruction-out-1
out1:	.....		.....
			.....
			.....
			instruction-out-n
			B out1
		out:	instruction-out-1
			.....
			.....
			instruction-out-n
		out1:	.....

Figura 9: Exemplo de transformação baseada em colagem

Se o bloco `out` finaliza com um desvio incondicional, ele pode ser copiado sem nenhuma alteração, mas se ele finaliza com uma instrução que não é de desvio, deve ser adicionado um desvio incondicional ao final de ambos os blocos `new-then` e `new-else`.

Nestes dois casos podemos criar novos blocos antes da primeira chamada do escalonador, porque *gluing* pode auxiliar a cobrir os atrasos oriundos da ocupação dos registradores no próprio bloco `out`. Se o bloco `out` finaliza com as instruções `CMP-BC`, podemos copiar somente as instruções anteriores ao BC, criar um novo rótulo antes de BC e colocar um desvio incondicional neste rótulo no final dos blocos `new-then` e `new-else`.

Em geral, colagem pode ser aplicada de maneira similar sempre que encontrarmos uma

tripla CMP-BC-B que é coberta por poucas instruções, ou seja, menos que quatro. A transformação aumenta o tamanho do bloco básico e cria novas oportunidades para propagação de constantes.

7 Conclusão

Uma das diferenças reais do sistema RISC/6000 sobre os RISCs tradicionais reside no fato de que esta arquitetura distribui os registradores pelas funções, permitindo execução simultânea com uma coordenação limitada. Pela exposição das áreas de coordenação exigidas, esta arquitetura permite ao compilador gerar um código altamente otimizado.

O estado da arte em relação as arquiteturas de ponto flutuante para máquinas RISC apresentou melhorias com a implementação do ponto flutuante para o sistema RS/6000 sob duas formas distintas. Primeiro, ela tornou a unidade de ponto flutuante um processador *tightly coupled* com um conjunto de registradores paralelos com cache de acesso direto, utilizando *hardware* adicional para evitar conflitos. Segundo, e mais significativo, a arquitetura foi otimizada em torno de uma única unidade que combina duas operações, de multiplicação e de adição, aumentando o desempenho da unidade de ponto flutuante enquanto reduz o número de portas, unidades funcionais distintas, e erros de arredondamento.

Algoritmos de escalonamento de instruções bem projetados são essenciais para o bom desempenho de compiladores otimizadores. Foi descrito neste trabalho um algoritmo proposto por Warren [WARREN 90] que foi desenvolvido para satisfazer todas as condições de escalonamento do conjunto de instruções e arquitetura do sistema RS/6000. Como Warren sugere, entretanto, ainda há um campo aberto para técnicas de escalonamento bem mais sofisticadas para os futuros sistemas. Golumbic e Rainish [GOLUMBIC 90] exploram uma destas possibilidades em seu trabalho sobre escalonamento além dos blocos básicos, também apresentado neste texto.

Para usufruir das vantagens das arquiteturas *pipeline* foi apresentado nesta monografia algumas transformações estruturais para *loops* e desvios, estendendo a visão de um escalonamento de código além dos blocos básicos. Estas técnicas têm sido implementadas no contexto das futuras versões do compilador PL/8. Experiências mostram que a média de tempo de execução acelerou de 4 a 5% além da técnica apresentada por [WARREN 90] para escalonamento em blocos básicos no código compilado para a unidade central de processamento do RS/6000 [GOLUMBIC 90].

Os esquemas de escalonamento de instruções descritos muito embora sejam apropriados para o processador do sistema RS/6000 da IBM, podem ser utilizados em outras arquiteturas ajustando-se alguns parâmetros, tais como os atrasos e tempo de execução.

A tendência atual da pesquisa está direcionada para arquiteturas que requisitem métodos de escalonamentos de instruções mais sofisticados, ou seja, que possuam mais situações que necessitem de escalonamento para obter um melhor desempenho. Os compiladores de alta qualidade no futuro empregarão técnicas de escalonamento mais sofisticadas do que as apresentadas neste texto. Trabalhos significativos têm sido elaborados nesta área, particularmente para arquiteturas com VLIW (Very Long Instruction Word) [COLWELL 87].

A família dos compiladores XL provavelmente será estendida de tal forma que o escalonador irá movimentar código além dos blocos básicos quando esta movimentação prover algum melhoramento. Um enfoque sobre isto foi apresentado em [GOLUMBIC 90].

Bibliografia

[AUSLANDER 82] AUSLANDER Marc & Martin HOPKINS, "An overview of the PL.8 Compiler", SIGPLAN Notices, *Proceedings of the SIGPLAN 82 Symposium on Compiler Construction.*, Vol. 17, No. 4, Abril 1982.

[BERNSTEIN 89] BERNSTEIN, D., GOLDIN, D. Q., GOLUMBIC, M. C., KRAWCZYK, H., MANSOUR, Y., NAHSHON, I., & PINTER, R. Y., "Spill Code Minimization Techniques for Optimizing Compilers", *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, ACM Press, New York, June 1989, pp. 258-263.

[BAKOGLU 90] BAKOGLU H. B., GROHOSKI G. F. & MONTOYE R. K., "The IBM RISC System/6000 processor: Hardware Overview", *IBM Journal of Research and Development*, Vol. 34 No. 1 January 1990.

[COCKE 90] COCKE, John & MARKSTEIN V., "The evolution of RISC technology at IBM", *IBM Journal of Research and Development*, Vol. 34 No. 1 January 1990.

[COLWELL 87] COLWELL Robert P., NIX Robert P., O'DONNELL John J., PAPWORTH David B & RODMAN Paul K., "A VLIW Architecture for a Trace Scheduling Compiler", *ASPLOS-II Proceedings - Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California October 5-8, 1987.

[GOLUMBIC 90] GOLUMBIC, M. C. & RAINISH V., "Instruction scheduling beyond basic blocks", *IBM Journal of Research and Development*, Vol. 34 No. 1 January 1990.

[GROHOSKI 90] GROHOSKI, G. F.,"Machine organization of the IBM RISC System/6000 processor", *IBM Journal of Research and Development*, Vol. 34 No. 1 January 1990.

[OEHLER 90] OEHLER, R. R. and GROVES, R. D., "IBM RISC System/6000 processor architecture", *IBM Journal of Research and Development*, Vol. 34 No. 1 January 1990.

[RADIN 87] RADIN, G.e M.E. HOPKINS, "A perspective on the 801/Reduced Instruction Set Computer", *IBM System Journal*, 26, 107-121 (1987).

[THORNTON 70] THORNTON, J. E., Design of a Computer: The Control Data 6600. *Glenview, Ill: Scott, Foresman*, 1970.

[WARREN 90] WARREN, Jr. H. S.,"Instruction Scheduling for the IBM RISC System/6000 Processor", *IBM Journal of Research and Development*, Vol. 34 No. 1 January 1990.