CONDEX-I, Modelo de Arquitetura VLIW com Capacidade de Execução Condicional e seu Compactador de Código

Anna Dolejsi Santos Marcelo Pires Módica
annads@dcc.uff.br modica@pgcc.uff.br
Departamento de Ciência da Computação
Universidade Federal Fluminense
Niterói - RJ - Brasil

Sumário

Arquiteturas VLIW que suportam a execução condicional, constituem uma alternativa promissora para o exploração do paralelismo a nível de instrução dos programas de aplicação. Especificamos e implementamos o CONDEX-I, um novo simulador para o nosso modelo de arquitetura. Desenvolvemos também um método para geração de código paralelo a partir do código seqüencial. O CONDEX-I utiliza o conjunto de instruções da arquitetura SPARC, viabilizando assim experimentos mais completos desse modelo de processamento. Desejamos investigar o efeito provocado pela inserção de diversos dispositivos funcionais em várias configurações do processador, a aceleração na execução de programas e a qualidade do código paralelo gerado, i.e., a redução no número de instruções e no número de instruções largas de cada programa de teste.

Abstract

VLIW machines supporting the conditional execution are a very promising alternative for exploring the instruction level parallelism of application programs. We specified and simulated the CONDEX-I, a new simulator for our architecture model. In addition we developed a method to generate parallel code for the CONDEX-I machine, from the sequential code. The CONDEX-I uses the instruction set of the SPARC architecture, allowing us to carry out a wider set of experiments. We are interested in the contribution of each processor component in several machine's configuration, the user program speed up and the static effect, i.e., the reduction in the number of instructions and in the number of very large instructions of each test program.

1. Introdução

A exploração do paralelismo no nível de instrução (*Instruction Level Parallelism* - ILP), utilizou inicialmente a técnica *pipelining* [1,2]. Posteriormente surgiram arquiteturas como as super escalares e as *Very Large Instruction Word* -VLIW [3,4,5], com diversas unidades funcionais replicadas. Essas máquinas, ao contrário das arquiteturas *pipelined*, podem completar a execução de mais de uma instrução por ciclo de processador.

A VLIW possui diversas unidades funcionais que podem executar simultaneamente instruções de um mesmo programa de aplicação, fora da ordem preestabelecida pelo programador. Nesse modelo de arquitetura, a compactação (escalonamento) das instruções que serão ativadas durante cada ciclo de máquina é realizada por um algoritmo implementado pelo *software*, que atua durante a fase de geração de código.

O maior problema encontrado na geração eficiente de código para arquiteturas VLIW diz respeito a baixa utilização de suas unidades funcionais: os recursos presentes no *hardware* não são utilizados eficientemente [6]. Apesar dessa baixa utilização, vem ocorrendo um incremento no número de projetos de pesquisa envolvendo processadores VLIW, já que a simplicidade dessa classe de arquiteturas permite que elas operem com relógios de freqüências muito altas. Importantes contribuições no desenvolvimento de tais máquinas incluem: técnicas de compactação local [7] e global [8,9,10] de código, especificação e implementação de processadores VLIW [9] e a inclusão da capacidade de execução condicional [11,12,13,14, 15,16].

O modelo de VLIW com capacidade de execução condicional CONDEX [14,15,16], utiliza um conjunto de instruções limitado e um compilador experimental que reconhece apenas um subconjunto da linguagem Pascal. Essa situação impede a utilização de grandes programas de teste em experimentos, já que a interpretação desses programas no simulador da arquitetura, obriga a alteração de seus fontes.

Utilizando o conjunto de instruções do processador SPARC Versão V7 [17] na concepção original do modelo de arquitetura CONDEX, a geração de código seqüencial pode ser feita através do compilador GNU gcc, permitindo melhor avaliação, tanto do modelo de arquitetura estudado, quanto do código paralelo produzido.

Este artigo trata do CONDEX-I (isto é, do modelo CONDEX, redefinido para o conjunto de instruções SPARC), da inclusão de um novo mecanismo de execução condicional e do novo compactador de código seqüencial. Na próxima seção, fazemos uma breve descrição das máquinas VLIW e do modelo CONDEX. O modelo de processador CONDEX-I é apresentado na Seção 3, e na Seção 4 mostramos o novo compactador de código. Finalmente na Seção 5, temos as considerações finais sobre o CONDEX-I e sobre a pesquisa em curso.

2. As Arquiteturas VLIW e CONDEX

Arquiteturas VLIW são caracterizadas pela existência de um único fluxo de instruções e pela presença de múltiplas unidades funcionais que podem operar em paralelo. Essas máquinas possuem uma unidade de controle bem simples devido ao processo de detecção do paralelismo no nível da instrução, que é realizado por *software*. Esse processo é conhecido como compactação [7,8,9,10]. Assim, torna-se necessária em cada ciclo de processador, somente a busca da instrução larga (ou instrução multifuncional - IMF), que contém o conjunto das instruções que devem ser executadas, conforme especificado em tempo de geração de código. Em toda IMF há um campo associado a cada unidade funcional existente na arquitetura. Cada campo contém o código da operação e os operandos fontes e destino associados à operação.

Técnicas usadas para escalonar instruções para os respectivos dispositivos funcionais, assumem um papel relevante no desempenho das arquiteturas VLIW. O principal objetivo dessas técnicas é gerar código paralelo capaz de manter as unidades funcionais bem utilizadas durante a execução de programas, aumentando desse modo o desempenho do processador. O processo de compactação deve considerar a interdependência das instruções (i.e., as dependências de dados e controle) e a limitação dos recursos existentes na arquitetura [5,18].

As técnicas de escalonamento podem ser classificadas em locais e globais. Técnicas locais [7] são aquelas que agem isoladamente no interior de cada um dos blocos básicos que compõem um programa. Já técnicas globais [8,9,10] escalonam instruções provenientes de diferentes blocos básicos, não se restringindo aos seus limites.

No modelo de VLIW CONDEX, cada instrução é suficientemente larga para acomodar os campos que controlam as unidades funcionais. Além da operação que será obedecida, cada campo indica a condição que deve ser satisfeita para que a respectiva unidade funcional seja ativada. Desse modo, torna-se possível empacotar em uma mesma IMF operações

provenientes de blocos básicos distintos, elevando o número de unidades funcionais que podem ser ativadas em paralelo [14,15,16].

O principal objetivo da arquitetura CONDEX é permitir a realização de compactação global de instruções e, com isso, aumentar a taxa de ocupação dos campos das IMFs, acelerando a execução de programas de aplicação.

Operações de desvio condicional provocam saltos no fluxo de controle de programas e degradam o desempenho das máquinas que exploram o ILP [19]. O compactador de código do modelo CONDEX, empacota em um mesmo conjunto de IMFs instruções provenientes de diversos blocos básicos alcançáveis pelo fluxo de controle.

No modelo CONDEX podem ser destacadas as seguintes características [15]:

- Banco de registradores. Conjunto de registradores de propósito geral que podem conter valores inteiros e valores em ponto flutuante.
- Unidades Funcionais. Múltiplas cópias de ALUs (unidades aritméticas e lógicas para inteiros), FPUs (unidades que realizam operações em ponto flutuante), MEMs (unidades de acesso à memória) e BR (unidade de processamento de desvios). As unidades funcionais devem obter informações do indicador condicional específico, comparar estes valores com os contidos no campo da IMF e avaliar a condição necessária, para decidir se a operação deve ou não ser executada.
- Banco de indicadores de condição. Cada registrador desse banco tem dois bits Z | N (zero | negativo) e tem por finalidade manter as informações produzidas pela execução de instruções de comparação. Essas informações são utilizadas pelas unidades funcionais quando da decisão da execução ou não, de uma operação que lhes é apresentada.
- Compilador. Os programas de teste são submetidos a um compilador experimental que reconhece um subconjunto da linguagem Pascal. Isso impede a utilização de programas de aplicação grandes, durante a validação do modelo.
- Processo de compactação. Processo que compacta as operações oriundas de blocos básicos correspondentes as estruturas em linguagem de alto nível do tipo "THEN e ELSE" ou "THEN e bloco Sucessor" (chamamos de bloco Sucessor, o bloco para onde o fluxo de controle é dirigido quando na linguagem de alto nível temos uma estrutura do tipo IF *condição* THEN..., isto é não existe o ELSE correspondente ao THEN, e em tempo de execução a *condição* é verificada falsa), do programa seqüencial e as compacta em IMFs. Compreende duas fases, a da compactação local e da compactação global.

Na Seção 3 mostramos as principais características do modelo de arquitetura VLIW CONDEX-I, que permite a execução condicional do conjunto de instruções do SPARC.

3. A Arquitetura CONDEX-I

Chamamos de CONDEX-I, ao modelo de arquitetura VLIW com capacidade de execução condicional, que utiliza o conjunto de instruções da arquitetura SPARC [17]. Esse conjunto de instruções foi escolhido pois:

- é amplamente utilizada em centros de pesquisa e universidades;
- corresponde à filosofia RISC (*Reduced Instruction Set Computer*), atualmente adotada pelos fabricantes; e
- permitirá a comparação de diferentes arquiteturas que exploram ILP, já que esse conjunto de instruções é utilizado também em um projeto PROTeM, desenvolvido em comum, pelo Departamento de Ciência da Computação da UFF e outras Universidades.

SPARC é uma arquitetura que apresenta as seguintes características, preservadas no CONDEX-I:

- 24 registradores de propósito geral para inteiros;
- 24 registradores de propósito geral para valores em ponto flutuante;
- utilização de janelas deslizantes; e
- 8 registradores globais.

A utilização desse conjunto de instruções, exigiu a redefinição das características arquiteturais e novas implementações para o simulador do modelo e para o gerador de código paralelo. Entre as modificações realizadas no modelo original CONDEX, temos:

- redefinição do banco de registradores de condição;
- redefinição do acesso ao banco de registradores de condição pelas unidades funcionais;
- inclusão da instrução "combine" no conjunto de instruções do SPARC;
- redefinição do efeito da instrução "compare" do processador SPARC; e
- redefinição do efeito das instruções de desvio do processador SPARC.

O modelo CONDEX-I, ilustrado na Figura 3.1, possui um banco de registradores de trabalho, quatro tipos de unidades funcionais (ALUs, FPUs, MEMs e BR), um registrador de estado e um banco de registradores de condição.

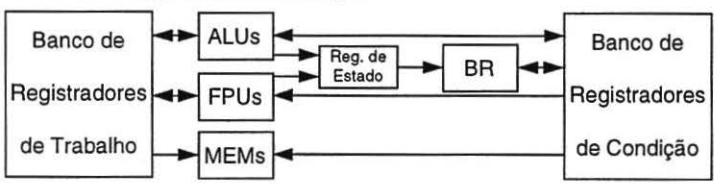

Figura 3.1: A arquitetura CONDEX-I

O banco de registradores de trabalho contém os registradores utilizados pelo programador. Ele fornece às unidades funcionais os valores dos operandos armazenados e recebe os valores resultantes das operações. Como as unidades funcionais operam em paralelo, estas podem obter os valores desejados do banco simultaneamente. Já no momento do armazenamento de resultados nos registradores, o programa de geração de código paralelo (o compactador) não permitirá que operações com dependência de saída compartilhem o mesmo ciclo, isto é, a mesma IMF.

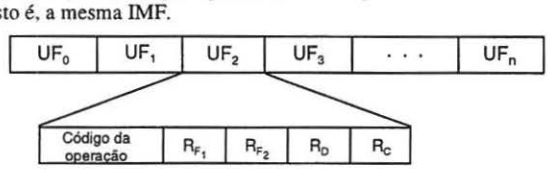

Figura 3.2: IMF da CONDEX-I

A segunda parte do CONDEX-I é constituída pelas unidades funcionais, que são múltiplas cópias de ALUs, FPUs e MEMs e uma única unidade de processamento de desvios (BR). Cada tipo de unidade funcional presente na arquitetura recebe, a cada ciclo de máquina, a operação contida em seu respectivo campo na IMF. Cada campo da IMF do CONDEX-1 (Figura 3.2), contém o código da operação, os operandos fonte e destino, além do registrador de condição que habilita ou não, a execução da operação. Assim, a única tarefa, relativa à execução condicional nas unidades funcionais, é buscar o valor do registrador de condição (R_C) indicado no campo associado da IMF, através do qual pode, de forma simples, decidir se deve (R_C contém 1) ou não (R_C contém 0) executar a operação que lhe é apresentada.

O terceiro elemento do modelo CONDEX-I (Figura 3.1), corresponde ao único registrador de estado formado por dois bits Z | N (zero | negativo). Esse registrador reflete o resultado produzido pela execução de uma instrução de comparação.

A última parte do CONDEX-I, é o banco de registradores de condição, que está intimamente ligado ao esquema de execução condicional de operações. Os registradores de condição são modificados por instruções de desvio e pela instrução "combine" (examinada mais adiante). Cada registrador de condição tem largura de um bit e pode conter apenas os valores 0 ou 1. Em tempo de execução as unidades funcionais utilizam o conteúdo desses registradores para decidir se a operação que lhes é apresentada na IMF, deve ou não ser executada.

A alteração introduzida no banco de registradores de condição, possibilitou a expansão e o aperfeiçoamento do processo de compactação, simplificou o trabalho das unidades funcionais em tempo de execução, e reduziu o número de bits gastos nos campos das IMFs necessários para a execução condicional. O banco de registradores de condição, mostrado na Figura 3.1 e detalhado na Figura 3.3, está organizado em pares registradores. O primeiro registrador do par, condiciona a execução de instruções do bloco THEN e o segundo a das do bloco ELSE. Assim, se a condição testada é verificada verdadeira, o par de registradores de condição correspondente, contém o valor 1-0. Caso contrário, o par contém o valor 0-1.

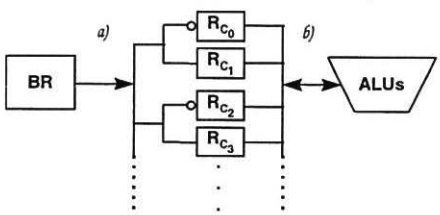

Figura 3.3: Organização detalhada do banco de registradores de condição

A Figura 3.3 ilustra o esquema de funcionamento do banco de registradores de condição. A Figura 3.3a mostra a relação com a unidade de processamento de desvios BR, que modifica os registradores de condição, e a Figura 3.3b mostra as relações com as ALUs, que escrevem no banco, quando executam a operação "combine." Todas as unidades funcionais da arquitetura, lêem os valores dos registradores de condição para decidir se devem ou não executar operações por eles condicionadas.

Com o uso do conjunto de instruções do SPARC, tornou-se necessário redefinir o efeito da execução de algumas instruções e incluir a instrução "combine," nesse conjunto. Assim:

- a execução da instrução "compare" produz a seguinte ação: é realizada a subtração dos dois valores comparados e o único registrador de estado é modificado. Se o resultado da subtração é positivo então ambos os bits Z e N recebem o valor 0. Se o resultado da subtração é negativo então o bits Z recebe o valor 0 e o bit N recebe o valor 1. Se o resultado da subtração é zero então o bits Z recebe o valor 1 e o bit N recebe o valor 0;
- a execução da instrução "branch" e de todas as suas variantes, corresponde a seguinte ação: o registrador de estado (Z | N) é avaliado e um par de registradores de condição é modificado, de acordo com a condição testada (maior ou igual, menor ou igual, diferente, etc). Em outras palavras, no CONDEX-I a execução da instrução "branch", não causa desvios no fluxo de controle do programa, mas modifica um par de registradores de condição; e
- a introdução da instrução "combine" que combina o conteúdo de registradores de condição, permitindo a compactação de código que apresenta desvios condicionais

aninhados. A ação resultante da execução da instrução "combine" corresponde a execução de um "and" onde os operandos são dois registradores de condição.

O modelo CONDEX-I viabiliza o uso do compilador GNU gcc, e a realização de experimentos com um conjunto de programas de teste como o SPEC.

O compilador gera o código objeto seqüencial usado como entrada pelo compactador de código, que por sua vez produz o código paralelo que explora a capacidade de execução condicional do CONDEX-I. Posteriormente, o código paralelo é interpretado pelo simulador da arquitetura, permitindo não apenas a obtenção de estatísticas para a avaliação do modelo de processador, como também a comprovação da equivalência semântica do código.

4. A Compactação para o CONDEX-I

Uma vez redefinido o nosso modelo de arquitetura, é necessário gerar código paralelo a partir do seqüencial. Desenvolvemos então, um novo compactador de código, o Compactador CONDEX-I, que leva em conta as características arquiteturais do modelo de processador CONDEX-I.

Se comparada a atuação do Compactador CONDEX, a ação do Compactador CONDEX-I se dá sobre um número maior de estruturas existentes na linguagem de alto nível. A Figura 4.4, ilustra as estruturas por ele tratadas. Na figura, BL significa bloco Livre; BT, bloco THEN; BE, bloco ELSE; BS, bloco Sucessor; e LOOP, bloco contendo um laço.

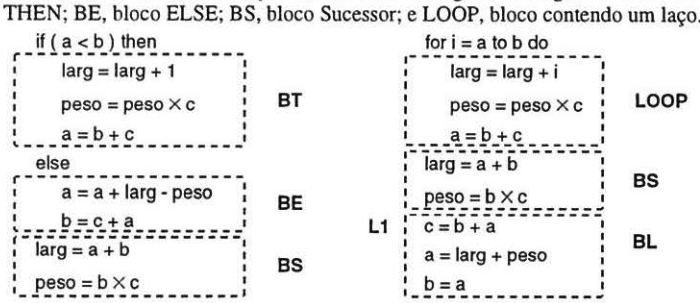

Figura 4.4: Estruturas utilizadas pelo processo de compactação condicional

O processo de compactação preenche IMFs com instruções provenientes das seguintes combinações de blocos:

- BL : bloco Livre não compartilha as IMFs que contêm suas instruções com instruções oriundas de nenhum outro bloco;
- BT-BS : nas IMFs que contêm instruções do bloco THEN são compactadas instruções do bloco Sucessor. Nesse caso o processo de compactação, considera as dependências de dados existentes entre as instruções do bloco THEN e as do bloco Sucessor, pois o bloco Sucessor é executado em qualquer caso e o bloco THEN pode também ser executado, se a condição testada é satisfeita;
- BT-BE-BS : nas IMFs que contêm instruções do bloco THEN são compactadas instruções dos blocos ELSE e Sucessor. Nesse caso, o processo de compactação considera também as dependências de dados existentes entre as instruções do bloco THEN e as do bloco Sucessor, pois o bloco Sucessor é executado em qualquer caso e o bloco THEN pode também ser executado, se a condição testada é satisfeita. A compactação, considera também as dependências de dados existentes entre as instruções do bloco ELSE e as do bloco Sucessor, pois o bloco Sucessor é executado em qualquer caso e o bloco ELSE pode também ser executado, se a condição testada não é satisfeita. As dependências de dados existentes entre as

instruções do bloco THEN e as do bloco ELSE, não são consideradas, já que a execução do bloco THEN garante que o bloco ELSE não é executado, e vice-versa (bloco Sucessor nesse caso, é o bloco para onde o fluxo de controle é dirigido após a execução da estrutura do tipo IF *condição* THEN...ELSE...).

- LOOP-BS : nas IMFs que contêm instruções do bloco LOOP são compactadas instruções do bloco Sucessor. Nesse caso o bloco LOOP é análogo a um bloco THEN, e o Sucessor a um ELSE. Desta forma, o processo de compactação não considera as dependências de dados existentes entre as instruções dos dois blocos.

Todas as instruções pertencentes a cada uma das combinações de blocos, identificadas pelo compactador, são relacionadas a um registrador de condição e compactadas em um único passo no conjunto de IMFs que compõe o programa paralelo.

O Compactador CONDEX-I incorpora os seguintes aperfeiçoamentos, para melhor uso da capacidade de execução condicional do modelo:

- compactação de "if"s aninhados;
- melhor aproveitamento dos campos das IMFs, através da reutilização de campos ocupados por instruções "nops" (*no operate*);
- compactação de blocos BT-BE-BS; e
- compactação de blocos LOOP-BS.

Instruções provenientes de "if"s aninhados são compactados em IMFs comuns devido ao formato e a organização em pares do banco de registradores de condição. O processo exige que, em tempo de geração de código, dois passos sejam realizados. O primeiro passo é feito sempre que um novo desvio condicional ("if") é encontrado no programa seqüencial. Já o segundo é realizado sempre que, após o primeiro passo, seja constatado que o "if," encontrado no primeiro passo, está aninhado dentro de um outro. Em outras palavras, a execução do segundo "if" depende do desvio condicional anterior. Esse dois passos, podem ser resumidamente descritos como se segue:

- **1º passo:** A presença de um "if "corresponde a duas instruções, a "compare" e a "branch," geradas pelo compilador. Em tempo de execução, a operação "compare" compara (subtrai) dois valores e preenche o único registrador de estado (Z | N) da arquitetura. Em seguida, a operação "branch" analisa o registrador de estado e, de acordo com a condição testada (maior ou igual, menor ou igual, diferente, etc), escreve o primeiro par de registradores de condição disponível. O par de registradores de condição é modificado da seguinte forma: se a condição é verdadeira, o primeiro registrador (correspondente ao bloco THEN) recebe valor 1 e o segundo (correspondente ao bloco ELSE) recebe o valor 0; se a condição é falsa, o primeiro registrador (correspondente ao bloco THEN) recebe valor 0 e o segundo (correspondente ao bloco ELSE) recebe o valor 1. O bloco executado é aquele que tem o registrador de condição com valor 1.
- **2º passo:** Esse passo verifica se a instrução de desvio condicional encontrada e processada no passo 1 está aninhada dentro de uma outra. É importante lembrar que a execução de um bloco depende não apenas da última condição, mas também de todas as outras condições correspondentes ao seu aninhamento. Nesse caso, é preciso, em tempo de geração de código, inserir uma instrução "combine" para que, em tempo de execução, seja possível combinar os pares de registradores correspondentes ao desvio condicional atual e ao seu antecessor. A execução da instrução "combine", faz um "and" entre um registrador do par anterior com cada um dos registradores do par atual.

Para ilustrar os dois passos descritos, temos na Figura 4.5 um exemplo de programa em linguagem de alto nível que possui um desvio condicional com outro desvio condicional aninhado. Ao lado do código em alto nível encontra-se o código *assembly* correspondente, com cada uma instrução seguida do registrador de condição que deve ser consultado pela unidade funcional, para a tomada de decisão sobre a execução ou não da instrução. O programa, para um melhor entendimento, não está compactado. Nesse programa, os nomes

das variáveis substituem os endereços de memória. Representamos por r0...r2 os registradores de trabalho e por c0...c5, os de condição (em negrito), sendo que os primeiros são fonte ou destino e os últimos condicionam cada operação.

if (a) then	1.	ld	a, r0, **c0**
	2.	tst	a, **c0**
	3.	be	c2, c3, **c0**
larg = larg + 1	4.	ld	larg, r1, **c2**
	5.	add	r1, 1, r1, **c2**
if (larg = peso) then	6.	ld	peso, r2, **c2**
	7.	cmp	r1, r2, **c2**
	8.	bne	c4, c5, **c2**
	9.	cmb	c4, c2, c4, **c2**
peso = peso + larg	10.	add	r2, r1, r2, **c4**
else	11.	cmb	c5, c2, c5, **c2**
peso = peso - larg	12.	sub	r2, r1, r2, **c4**

Figura 4.5: Exemplo do processo condicional sobre estruturas aninhadas

No primeiro comando "if" é aplicado apenas o primeiro passo, resultando nas operações 2 e 3. As instruções dependentes do segundo "if" também o são do primeiro. Quando esse segundo "if" é encontrado, o segundo passo do processo também deve ser realizado, resultando na inclusão das operações 9 e 11 (duas instruções "combine"). É importante destacar que a presença das instruções 7 e 8, resultantes do primeiro passo, independem do aninhamento. As instruções 9 e 11 combinam o resultado produzido pela instrução 3 (isto é, o valor de c2) com os valores produzidos pela instrução 8 (isto é, os valores de c4 e c5). Em tempo de execução, se o conteúdo de c2 for colocado a 1 pela instrução 3, então as instruções 4 a 9 e 11 são executadas. Porém, a execução das instruções 10 e 12 depende não apenas do conteúdo de c2, mas também do resultado produzido pelas operações "combine" (9 e 11).

O compactador de código do CONDEX-I busca também um melhor aproveitamento dos campos das IMFs através da reutilização de campos ocupados por "nops" (códigos de operação que reservam a unidade funcional) decorrentes das diferentes latências das instruções compactadas. Esse processo se dá em dois passos:

- **1º passo:** As diferentes instruções presentes no modelo CONDEX-I têm diferentes latências. Assim, quando da compactação das instruções do bloco THEN (como também de outro bloco qualquer), os códigos de tais instruções ocupam apenas o campo da primeira IMF, sendo as outras ocupadas por "nops" de acordo com os tamanhos das latências. Esses "nops" asseguram que, quando da execução do bloco THEN, esses ciclos estarão reservados na unidade funcional correspondente até a completa execução da operação. Para ilustrar essa idéia, na Figura 4.6a, existe um trecho de programa compactado, nele estão presentes apenas operações do bloco THEN.
- **2º passo:** Após a compactação do bloco THEN, nas IMFs que contêm suas instruções, compactam-se instruções do bloco ELSE. Embora esses dois blocos compartilhem o mesmo conjunto de IMFs, no momento da execução apenas as operações de um deles serão realizadas pelas unidades funcionais. Assim, esses blocos serão mutuamente exclusivos com relação a execução, tornando possível que os campos que contém os "nops" correspondentes às latências de instruções pertencentes a um bloco, sejam utilizados para abrigar operações do outro, sem prejuízo na equivalência semântica do código. Na Figura 4.6b, temos as operações do bloco THEN da Figura 4.6a, compartilhando IMFs com as operações oriundas de um bloco ELSE (em negrito).

Na Figura 4.6, r0...r4 designam os registradores de trabalho e c2...c3 os registradores que condicionam cada operação. Na Figura 4.6a foram compactadas, através do primeiro

passo, operações "add", "sub" e "mul" pertencentes ao bloco BT. Todas têm latência igual a 1, com exceção da operação "mul", que possui latência de valor 3 e, portanto, preenche dois campos (IMFs 2 e 3) com "nops". Já a Figura 4.6b apresenta o resultado do segundo passo, ou seja, a compactação do BE em IMFs do BT. As operações do bloco BE têm latência igual a 1 e, como serão executadas se as do BT não o forem (e vice-versa), podem ocupar os campos do BT preenchidos por "nops". É possível observar na IMF 2, que o campo reservado por um "nop" do bloco BT na Figura 4.6a recebeu uma instrução do bloco BE na Figura 4.6b.

a)

IMF_0	add $r_0,r_1,r_{2,}c_2$	sub $r_0,r_1,r_{3,}c_2$
IMF_1	mul $r_0,r_2,r_{2,}c_2$	
IMF_2	nop.c2	
IMF_3	nop.c2	

b)

add $r_0,r_1,r_{2,}c_2$	sub $r_0,r_1,r_{3,}c_2$
mul $r_0,r_2,r_{2,}c_2$	**add $r_2,r_0,r_{2,}c_3$**
add $r_1,r_3,r_{3,}c_3$	**sub $r_2,r_4,r_{4,}c_3$**
nop.c2	

Figura 4.6: Exemplo do 1o (a) e do 2o (b) passos da compactação de BT com BE

A terceira diferença do novo processo de compactação, é que o Compactador CONDEX-I, sempre compacta no conjunto de IMFs resultante da compactação dos blocos THEN e ELSE, instruções pertencentes ao bloco Sucessor. Com isso, a execução de dessas instruções é adiantada, o número de IMFs necessárias para abrigar o bloco Sucessor é reduzido e conseqüentemente o desempenho do modelo pode ser aumentado.

A última particularidade do novo processo de compactação diz respeito a união de um bloco LOOP com o bloco Sucessor. Isto é possível porque, ao final da última repetição do LOOP, o teste condicional causará um desvio para o bloco Sucessor, de modo análogo ao que ocorre com um bloco ELSE. Essa situação permite que o processo de geração de código paralelo, beneficie-se das mesmas vantagens da compactação dos blocos THEN com blocos ELSE. Em outras palavras, é possível reutilizar campos ocupados por "nops", introduzidos com a finalidade de reservar as unidades funcionais por ciclos correspondentes às latências das operações. Além disso, ao final da repetição do laço, o bloco Sucessor pode ser executado sem a execução de uma instrução de desvio, melhorando a qualidade do código gerado.

5. Considerações Finais

Especificamos e implementamos o CONDEX-I, um novo simulador para o nosso modelo de arquitetura, e um gerador de código paralelo. O CONDEX-I utiliza conjunto de instruções da arquitetura SPARC, viabilizando experimentos para a validação do modelo de arquitetura VLIW com capacidade de execução condicional de instruções.

O simulador CONDEX-I e seu gerador de código paralelo, estão implementados em linguagem C, e reconhecem até o momento as instruções do processador SPARC que manipulam inteiros. Ambos encontram-se na fase final de testes. Através da variação dos seus parâmetros, permitem a avaliação de diversas configurações de máquina.

Em nossos experimentos iniciais, estamos investigando o efeito provocado pela inserção de diversos dispositivos funcionais no desempenho global do processador, a aceleração na execução de programas, a contribuição oferecida por cada tipo de componente à relação custo x desempenho de diversas configurações da máquina, e a qualidade do código paralelo gerado. Os primeiros resultados apontam para um código paralelo de melhor qualidade e conseqüentemente para um melhor desempenho do modelo de processamento.

O próximo passo a ser realizado, diz respeito a inclusão, no simulador e no compactador do CONDEX-I, das instruções que envolvem a aritmética em ponto flutuante e a otimização da técnica de compactação de instruções. Nosso trabalho prossegue nessa direção.

Referências

[1] John L. Hennessy and David A. Patterson, "Computer Architecture: A Quantitative Approach," Morgan Kaufmann Publisher Inc, USA, 1996, pp. 125-214.

[2] C. V. Ramamoorthy and H. F. Li, "Pipeline Architecture," Computing Surveys, Vol. 9, No.1, March 1977, pp. 61-102.

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

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

[5] Edil S. T. Fernandes e Anna Dolejsi Santos, "Arquiteturas Super Escalares: Detecção e Exploração do Paralelismo de Baixo Nível," VIII Escola de Computação, Gramado, RS, Agosto de 1992, 155 páginas.

[6] Michael A. Schuette and John P. Shen, "An Instruction-Level Performance Analysis of the Multiflow Trace 14/300," Proceedings of the 11th Annual International Symposium on Microarchitecture, November 1991, pp. 2-11.

[7] David Landskov, Scott Davidson, Bruce Shriver, and Patrick W. Mallet, "Local Microcode Compaction Techniques," Computing Surveys, vol. 12, no. 3, September 1980, pp. 261-294.

[8] M. Tokoro, E. Tamura, T. Takizuka and I. Yamamura, "A Technique of Global Optimization of Microprograms," Proceedings of the 11th Annual Microprogramming Workshop, 1978, pp. 41-50.

[9] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," IEEE Transactions on Computers, vol. C-30, no. 7, July 1981, pp. 478-490.

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

[11] J. Labrousse and G. Slavenburg, "A 50 MHz Microprocessor with a VLIW Architecture," Proceedings of the International Solids State Circuits Conference, San Francisco, 1990.

[12] Sue Gray and Rod Adams, "Using Conditional Execution to Exploit Instruction Level Concurrency," Technical Report No. 181, University of Hertfordshire, School of Information Sciences, Division of Computer Science, March 1994, 30 pages.

[13] F. L. Steven, G. B. Steven and L. Wang, "An Evaluation of the iHARP Multiple Instruction Issue Processor," Euromicro 94, September 1994.

[14] Edil S. T. Fernandes, Anna Dolejsi Santos and Claudio L. de Amorim, "Conditional Execution: an Approach for Eliminating the Basic Block Barriers," Microprocessing and Microprogramming The Euromicro Journal, North Holland, vol. 40, no. 10-12, December 1994, pp. 668-692.

[15] Anna Dolejsi Santos, "Efeito da Execução Condicional em Arquiteturas Paralelas," Tese de Doutorado, COPPE/UFRJ, Dezembro de 1994.

[16] Anna Dolejsi Santos, Andrew Wolfe and Edil S. T. Fernandes, "Functional Units Utilization in a Multiple-Instruction Issue Architecture," Proceedings of the 23rd Euromicro Conference, Euromicro (ISBN 0-8186-8215-9), IEEE Computer Society, 1997, pp. 228-233.

[17] Sun Microsystems, "The SPARC Architecture Manual Version 7," Mountain View, CA, 1987.

[18] R. M. Keller, "Look-Ahead Processors," Computing Surveys, Vol. 7, No. 4, December 1975, pp. 177-195.

[19] Scott McFarling and John Hennessy, "Reducing the Cost of Branches," Proceedings of the 13th International Symposium on Computer Architecture, ACM and IEEE Computer Society, Tokio, Japan, June 1986, pp. 396-403.