Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Relatório Parcial do Projeto Orientado II

Implementação de um Algoritmo para Alocação de Registradores com Informações sobre Escalonamento de Instruções Embutidas

Área: Compiladores
Projeto: Implementação de um Algoritmo para Alocação de Registradores com Informações sobre Escalonamento de Instruções Embutidas
Orientadora: Professora Mariza Andrade da Silva Bigonha
Orientando: Nizam de Abreu Pfeilsticker
Cronograma: Início : Novembro de 1998
Término : Julho de 1999

Belo Horizonte, 20 de Julho de 1999

Resumo

Computadores superescalares executam mais de uma instrução por ciclo e possuem várias unidades funcionais. Além disto, como são originários da arquitetura *RISC* e, portanto, caracterizam-se pela simplicidade de seu conjunto de instruções. Cabe ao compilador, então, a solução do problema de geração e otimização de código.

Um dos problemas existentes consiste em determinar a melhor maneira de integrar o escalonamento de instruções e a alocação de registradores de forma eficiente. Para resolução deste problema, Pinter sugere um algoritmo.

Este documento tem por objetivo descrever e implementar o algoritmo de Pinter, utilizando como alvo o processador μRisc, desnvolvido no Departamento de Ciência da Computação da Universidade Federal de Minas Gerais pelo professor Claudionor Coelho. O processador foi modificado para se adequar aos objetivos do trabalho.

Sumário

1 Introdução **1**

2 Processador μRisc **2**
2.1 Introdução 2
2.2 Pipeline 3
2.2.1 Problemas com Pipeline 3
2.3 Conjunto de Instruções 5
2.3.1 ALU 5
2.3.2 Constantes 6
2.3.3 Transferência de Controle 6
2.3.4 Memória 7
2.3.5 NOP 7
2.3.6 HALT 7
2.4 Arquivos de Entrada 7

3 Especificação do Algoritmo Proposto **8**
3.1 Construção do Grafo de Escalonamento 8
3.2 Construção do Grafo de Interferência 9
3.3 Construção do Grafo de Complemento 10
3.4 Construção do Grafo de Interferência Paralelizável 12

4 Metodologia de Implementação do Algoritmo de Pinter **13**
4.1 Escalonamento de Instrução 13
4.2 Alocação de Registradores 15
4.2.1 Problemas do Método de Chaitin 16
4.3 Estruturas de Dados 17
4.4 Implementação do Grafo de Interferência Paralelizável 17
4.4.1 Exemplos 19

5 Conclusão **20**

A Código do Algoritmo de Pinter **21**

Lista de Figuras

1 Grafo de Escalonamento 8
2 Construção do Grafo de Interferência 9
3 Fecho transitivo do Grafo de Escalonamento 11
4 Grafo de Escalonamento com Informações de Dependência de Máquina 11
5 Grafo Complemento 12
6 Integração dos Grafos de Interferência e Complemento 12
7 *Grafo de Interferência Paralelizável* 13
8 Escalonamento de Instrução 14
9 Coloração de um grafo simples 16
10 Um grafo simples que requer duas cores 17
11 Estrutura de Dados Utilizada para Construção do Grafo 17

Lista de Tabelas

1 Intruções de ALU 5
2 Instruções de Constantes 6
3 Instruções de Salto 6
4 Condições de Salto 6
5 Instruções de acesso à Memória 7

1 Introdução

Computadores modernos têm a capacidade de executar mais de uma instrução por ciclo e possuem várias unidades funcionais que podem operar em paralelo, tais máquinas são denominadas superescalares.

As máquinas superescalares são uma evolução das arquiteturas RISC (*Reduced Instruction Set Computers*), que se caracterizam pela simplicidade de seu conjunto de instruções, em contraposição às arquiteturas CISC (*Complex Instruction Set Computers*), cujo conjunto de instruções se caracteriza por sua complexidade.

Como computadores superescalares são originados da tecnologia RISC, mais instruções são utilizadas para a execução de um programa, cabendo ao compilador a solução de problemas complexos de geração e otimização de código. Um destes problemas está relacionado com a alocação eficiente de registradores e o escalonamento de instruções. Surgem as seguintes questões: qual destas duas tarefas deve ser feita primeiro? Deve existir algum tipo de relacionamento entre elas?

Uma ótima técnica, até recentemente, para resolver o problema de uma alocação eficiente de valores a registradores físicos em um compilador é a coloração do grafo de interferência.[2][4] Neste grafo, os vértices, representando pseudo-registradores, e as arestas, representando conflitos entre os pseudo-registradores ativos em um intervalo de tempo, eram coloridos por um certo número de cores registradores físicos, derramando para a memória alguns valores quando necessário. Com o advento dos processadores superescalares, permitindo o paralelismo entre instruções, algoritmos de coloração de grafos, considerados ótimos outrora, já não se correlacionam com uma boa utilização dos recursos desta máquina. Para explorar o paralelismo existente entre instruções, as mesmas devem ser reordenadas pelo escalonador de instruções.

Quando a reordenação é feita após a alocação de registradores, a seleção de registradores pode limitar as possibilidades de escalonamento, diminuindo o paralelismo, devido a introdução de dependências falsas com o reuso dos registradores. Por outro lado, quando o escalonamento é feito antes da alocação de registradores o número de registradores ativos aumenta dando origem a sérias implicações como: o aumento da vida de registradores, mais registradores físicos são necessários e ocorrem mais derramamentos para a memória[1].

Então torna-se necessária a especificação de um algoritmo que aloque eficientemente registradores e que faça uma ótima utilização do paralelismo existente entre as instruções, considerando, por exemplo, os recursos das máquinas superescalares. Tal algoritmo é proposto por Pinter, em 1993, por meio da coloração do Grafo de Interferência Paralelizável[4].

A idéia básica de seu algoritmo é prover uma única estrutura de dados, o Grafo de Interferência Paralelizável, com todas as informações necessárias tanto para a alocação de registradores quanto para o escalonamento, de tal forma que estas duas tarefas possam ser executadas usando os algoritmos clássicos para alocação de registradores, coloração de grafos[2], escalonamento e lista de escalonamento[3], mas sem restringir a ação de cada um.

O objetivo deste trabalho foi a implementação do algoritmo de Pinter. O processador μRisc, detalhado na Seção 2 foi usado como a máquina alvo. Este texto relata a experiência e os resultados advindos de tal implementação.

2 Processador μRisc

2.1 Introdução

O processador μRisc possui 16 bits e baseia-se na arquitetura RISC. Esse processador possui 8 registradores de uso geral e 42 instruções.

Cada instrução demora 4 ciclos para completar, embora 8 instruções possam estar sendo executadas paralelamente. Esses ciclos de execução são denominados: IF, ID, EX/MEM e WB.

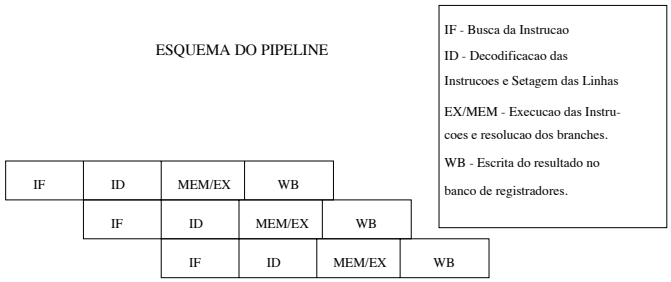
- IF (*instruction fetch*): a próxima instrução a ser executada é buscada na memória e armazenada no registrador de instruções(IR);
- ID (*instruction decode*): a instrução sendo executada é decodificada e os operandos são buscados no banco de registradores;
- EX/MEM (*execute and memory*): a instrução é executada e as condições dos resultados são "setados"(condition codes) para operações de ALU. Loads, stores, o cálculo do endereço efetivo e a resolução de branches são feitos também nesse ciclo;
- WB (*write back*): os resultados são escritos no banco de registradores.

O processador μRisc possui as seguintes características:

- 16 bits;
- 8 registradores de uso geral de 16 bits de largura;
- 42 instruções;
- instruções de 3 operandos;
- submissão de 2 instruções por ciclo;
- *big endian*;
- memória endereçada a nível de palavra, ou seja, cada endereço de memória deve se referir a dois bytes. No total, o processador deverá possuir 64k(2^16) endereços. Então, a memória total do processador será de 128k(64k de endereços x 2 bytes).

2.2 Pipeline

Ao contrário das versões anteriores deste processador, que se baseavam em uma máquina de estados finitos, onde cada instrução caminha através da máquina até o seu término, esta nova versão implementa uma *pipeline* simples. Em um processador com *pipeline* existe uma busca de instrução por ciclo, e a execução das instruções é realizada de forma concorrente. Portanto, em condições ideais, todas as unidades funcionais do processador estão em funcionamento, pois cada instrução se encontra em um estado da máquina, não havendo assim recursos ociosos, o que resulta em maior paralelismo e conseqüentemente maior desempenho. A maioria dos processadores comerciais hoje existentes recorrem a *pipelines* como recurso para aumento de desempenho. A *pipeline* a ser implementada apresenta profundidade de quatro estágios, que se dividem em IF(instruction fetch - busca da instrução), ID(instruction decode - decodificação da instrução), MEM/EX(execução da instrução e busca de operandos na memória) e WB(Write-Back).



2.2.1 Problemas com Pipeline

Apesar da simplicidade da *pipeline* implementada, ela provoca uma série complicações na arquitetura. Como são executadas várias instruções em paralelo, é necessário que, após a decodificação da instrução, as linhas de controle ligadas sejam armazenadas em registradores especiais de forma que o processador ajuste as linhas de controle durante a passagem da instrução de um determinado estágio para outro. Isto é feito acrescentando uma série de registradores entre cada estágio de forma que as informações pertinentes à instrução avancem no pipeline até que não sejam mais úteis (veja Datapath na Seção **??**). A principal dificuldade em se implementar uma *pipeline* eficiente é lidar com as dependências de dados e recursos estruturais entre instruções, que podem resultar em *hazards* e execução errônea das instruções. Uma descrição detalhada do que são *hazards* e de como eles ocorrem na *pipeline* está além do escopo deste documento, mas podem ser encontradas em [6] .

Na arquitetura proposta, existem três situações que podem causar tais problemas. São elas:

Hazard Estrutural - Instruções submetidas simultâneamente que utilizam recursos comuns como ALU ou o porto de memória geram uma parada na *pipeline* em uma das instruções.

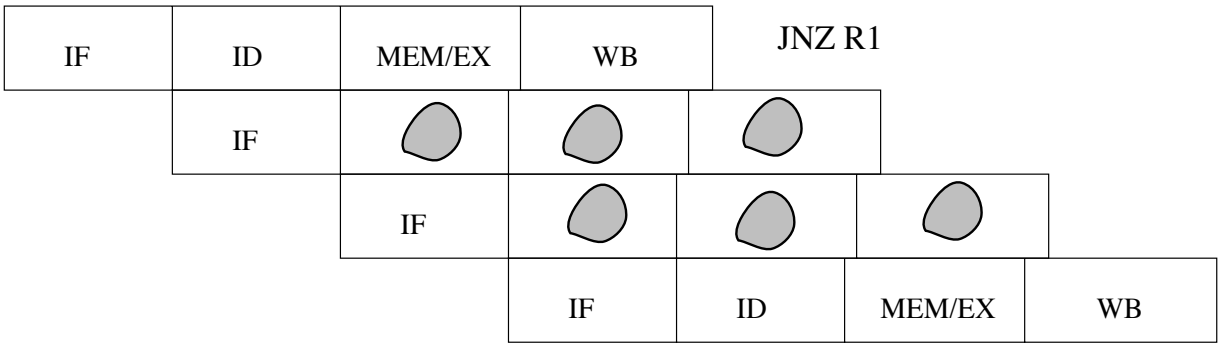
Hazard de Controle - Uma das maiores limitações de desempenho na *pipeline* são devidos à *hazards* de controle, ou seja, paradas na *pipeline* devido a execução de desvios. Nesta versão do processador foi implementado a política de

em Predict Not Taken onde o processador supõe que o desvio não vai ser tomado e continua a operação normalmente. O tratamento de desvios nesta arquitetura se dá no estágio MEM/EX, portando ao descobrir que o desvio será tomado se faz necessário que o pro

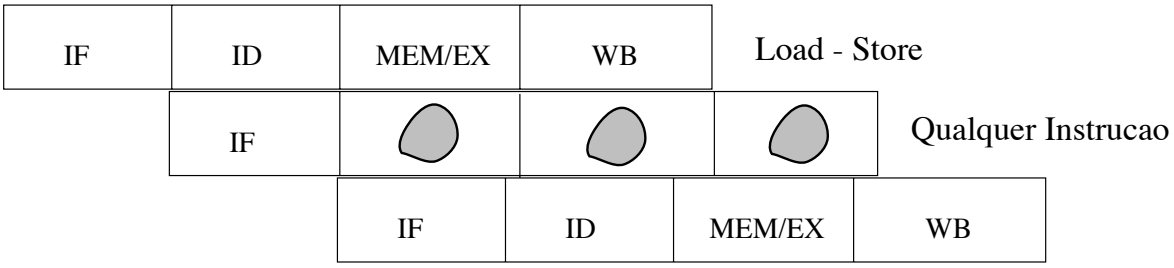
cessador insira duas bolhas na *pipeline*, anulando as duas instruções que já foram buscadas antes da resolução do desvio.

ESQUEMA DE STALLS

Branch tomado



Mem - Stall



Hazard de Dados - Outro problema que aflige a implementação da arquitetura são os Hazard de Dados, que aparecem quando existe uma dependência de dados entre operandos de instrução vizinhas.E é este o principal problema que nos motiva ao escalonamento de instruções.

2.3 Conjunto de Instruções

O processador μRisc possui o seguinte conjunto de instruções:

- instruções aritméticas que envolvem ALU;
- instruções que envolvem o uso de constantes;
- instruções de transferência de controle; instruções que acessam a memória.

2.3.1 ALU

As instruções de ALU seguem o formato dado pela Tabela 1 abaixo:

Operação	**Mnemônico**
$C = 0$	zeros c
$C = A\&B$	and c,a,b
$C = !A\&B$	andnota c,b,a
$C = A$	passa c,a
$C = A \wedge B$	xor c,a,b
$C = A\|B$	or c,a,b
$C = !A\&!B$	nor c,a,b
$C = !A \wedge B$	xnor c,a,b
$C = !A$	passnota c,a
$C = A\|!B$	ornotb c,a,b
$C = !A\|!B$	nand c,a,b
$C = 1$	ones c
$C = A + B$	add c,a,b
$C = A + B + 1$	addinc c,a,b
$C = A + 1$	inca c,a
$C = A - B - 1$	subdec c,a,b
$C = A - B$	sub c,a,b
$C = A - 1$	deca c,a
$C =$shift lógico para esquerda	lsl c,a
$C =$shift lógico para direita	lsr c,a
$C =$shift aritmético para direita	asr c,a
$C =$shift aritmético para esquerda	asl c,a

Tabela 1: Intruções de ALU

2.3.2 Constantes

O processador μRisc possui somente instruções para a carga de constantes com sinal, representadas pelas instruções indicadas pela Tabela 2:

Operação	**Mnemônico**
C =Constante	loadlit c, Const11 lc c, Const11
C =Const8 | (C & 0xff00)	lcl c, Const8
C =(Const8 $\ll$ 8) | (C & 0x00ff)	lch c, Const8

Tabela 2: Instruções de Constantes

2.3.3 Transferência de Controle

O processador μRisc possui 5 instruções para a transferência de controle codificadas de três maneiras diferentes que são indicadas na Tabela 3. A primeira codificação é utilizada para instruções de desvios condicionais e a segunda codificação é utilizada para a instrução de desvios incondicionais. Em ambas as instruções o desvio é relativo ao PC, contudo o comprimento da instrução de desvios condicionais é de 8 bits enquanto que nas instruções de desvios incondicionais o comprimento é de 12 bits.

Operação	**Mnemônico**
Jump False	jf.cond Destino
Jump True	jt.cond Destino
Jump Incondicional	j Destino
Jump and Link	jal b
Jump Register	jr b

Tabela 3: Instruções de Salto

As condições de transferência de controle são definidas na Tabela 4:

Condição	**Mnemônico**
Resultado ALU Negativo	.neg
Resultado ALU Zero	.zero
Carry da ALU	.carry
Resultado da ALU negativo ou zero	.negzero
TRUE	.true
Resulato da ALU Overflow	.overflow

Tabela 4: Condições de Salto

2.3.4 Memória

Operação	Mnemônico
C = Mem[A]	load c,a ld c,a
Mem[A] = B	store a,b st a,b

Tabela 5: Instruções de acesso à Memória

2.3.5 NOP

Esta é uma instrução que não faz nada.

2.3.6 HALT

Esta é uma instrução que determina a parada de sistema.

2.4 Arquivos de Entrada

O arquivo de entrada para o processador μRisc deve conter as instruções em hexadecimal em código ASCII, uma instrução por linha. Pode conter a instrução *address Endereço* que indica que as instruções subsequentes devem ser guardadas no *Endereço* definido.

Para geração de tal arquivo pode-se utilizar o Montador e Editor ligador que se encontram na máquina Turmalina do Departamento de Ciência da Computação da Universidade Federal de Minas Gerais.

- **Montador:** /pkg/cursos/sis/urasm
- **Editor Ligador:** /pkg/cursos/sis/urlink

Estes arquivos aceitam como entrada o conjunto de instruções do processador μRisc em formato Mnemônico.

O programa implementado neste projeto recebe como entrada uma seqüência de código mnemônico representando os blocos básicos do programa a ser executado. Para cada uma das instruções de seqüência dada é fornecido um novo pseudo-registrador. Assume-se que existe um número infinito do mesmo. O programa produz como saída um novo conjunto de código mnemônico que reflete todo o paralelismo existente e uma "ótima"atribuição de registradores.

3 Especificação do Algoritmo Proposto

Pinter[4] propõe um algoritmo que realiza a integração do grafo de interferência, utilizado para a alocação de registradores, e o grafo de escalonamento, usado para escalonamento de instruções dando origem ao Grafo de Interferência Paralelizável.

Com este algoritmo é possível uma alocação de registradores com um número mínimo de registradores sem diminuição do paralelismo inerente ao programa.

O algoritmo de Pinter será explicado usando o exemplo abaixo:

Suponha o seguinte trecho de programa, retirado de [4]:

(a) X := a[i]
Y := Z + Z
Z := X*5 + Z

Cujo código utilizando pseudo-registradores é:

(b) S1 := load Z
S2 := i
S3 := a[S2]
S4 := S1 + S1
S5 := S3 * 5 + S1

3.1 Construção do Grafo de Escalonamento

O código mostrado em (b) possui dependências de escalonamento ilustradas no grafo de escalonamento da Figura 1. Neste grafo, cada nó representa um pseudo-registrador, que define uma instrução e a aresta direcionada representa a utilização do pseudo-registrador no lado direito da instrução. Para construí-lo basta apontar o nó com o pseudo-registrador aos nós em que este parece do lado direito.

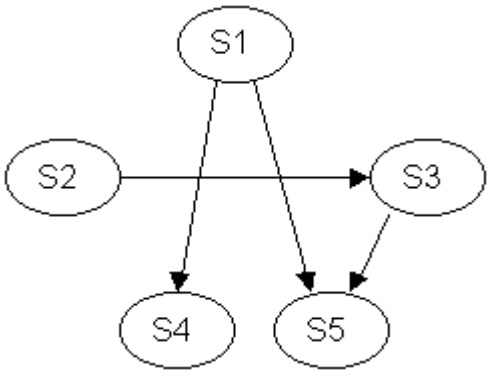


Figura 1: Grafo de Escalonamento

3.2 Construção do Grafo de Interferência

O grafo de interferência para o mesmo trecho de programa que aparece em (a) apresentado na Figura 2.

Para construí-lo é necessário que se observe as regras abaixo:

- Uma variável está viva até ela ser utilizada do lado direito de uma instrução, ou seja, enquanto ela for necessária sem redefinição, ela está viva.
- Todo vértice $v \in V_r$ corresponde a um intervalo distinto do programa no qual a definição da variável está viva.
- Existe uma aresta não dirigida $(u, v) \in E_r$ se uma definição está viva em u e é necessária em v, ou seja, seu valor é usado, ou subseqüentemente usado, em um comando onde o outro é definido, os dois intervalos se interceptam.

Na prática, em muitos compiladores o ponto final do intervalo de uma variável viva,ou seja, o comando correspondendo ao seu último uso, não é considerado como parte do intervalo, isto permite o reuso do registrador no mesmo comando de seu último uso.

No exemplo dado deve ser adicionadas a S1, como seu último uso foi em S5, as arestas S2, S3 e S4; a S2 não é necessário adicionar novas arestas; a S3 deve ser adicionada a aresta $(S3, S4)$; a S4 e S5 não deve ser adicionado arestas novas. O grafo de interferência fica, conforme mostrado na Figura 2:

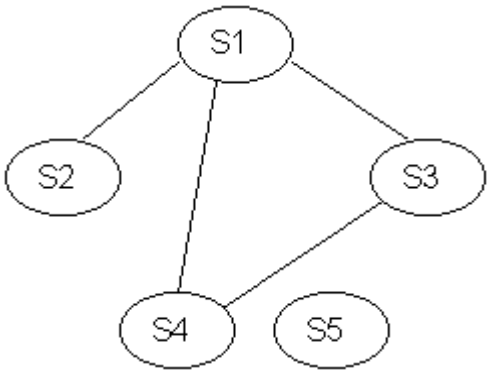


Figura 2: Construção do Grafo de Interferência

O que deve ser feito antes: o escalonamento de intruções ou a alocação de registradores? Este é um dos pontos chaves ao gerar código para arquiteturas mais modernas.

Considerando que a alocação de registradores fosse realizado primeiro, uma possível atribuição de valores a registradores físicos poderia ser:

R1 := load Z
R2 := i
R3 := a[R2]
R2 := R1 + R1
R1 := R3 * 5 + R1

A alocação proposta é muito boa pois utiliza um número mínimo de registradores físicos. Porém, como pode ser visto a seguir, há a inserção de uma dependência falsa [1] entre a segunda e a quarta instrução, o que restringe o escalonamento de intruções, perdendo o paralelismo antes existente entre estas duas instruções.

Para resolver tal problema será introduzido o algoritmo proposto por Pinter. Tal algoritmo trata do problema a partir do grafo de escalonamento e do grafo de interferência, onde as instruções ainda se utilizam de pseudo-registradores. Segue-se ainda os seguintes passos:

1- Construção do grafo de complemento.

2- Integração do grafo de interferência e complemento.

3.3 Construção do Grafo de Complemento

Para a construção do grafo complemento os seguintes passos são necessários:

Passo 1:

Insere-se arestas dirigidas entre instruções com dependências de dados.

Para caracterizar dependência de dados observe as regras abaixo:

Sejam u e v duas instruções. A dependência de dados entre u e v existe se pelo menos uma das afirmações for verdadeira:

- Dependência de controle de dados: registrador definido em u é usado em v.
- Anti-dependência de dados: um registrador usado em u é redefinido mais tarde em v, destruindo o valor utilizado em u.
- Dependência de saída de dados: o registador definido em u é redefinido em v, destruindo o valor definido previamente em u.

Passo 2:

Faz-se o fecho transitivo do grafo de escalonamento e removem-se as direções das arestas. Portanto, no exemplo utilizado, insere-se a aresta $(S2, S5)$ conforme mostra a Figura 3

[1] dependência falsa é uma dependência introduzida pelo reúso de registradores em instruções antes independentes

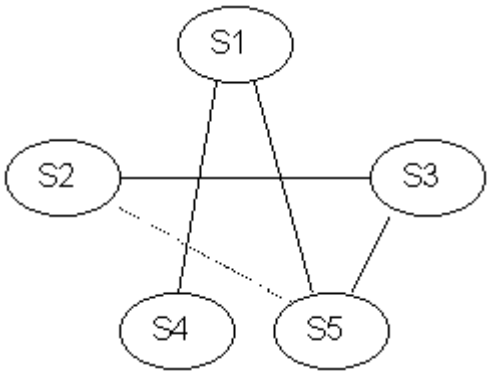


Figura 3: Fecho transitivo do Grafo de Escalonamento

Passo 3:

Insere-se arestas entre instruções que são dependentes da máquina, que não de precedência. Supõe-se a título de exemplo que a máquina possua apenas uma unidade de memória e uma unidade de ponto fixo. Portanto para o exemplo apresentado na Seção 3, insere-se duas arestas devido as limitações da máquina:

Arestas entre $(S1, S3) \rightarrow$ unidade de memória

Arestas entre $(S4, S5) \rightarrow$ unidade de ponto fixo

A Figura 4 apresenta o resultado da inclusão destas arestas.

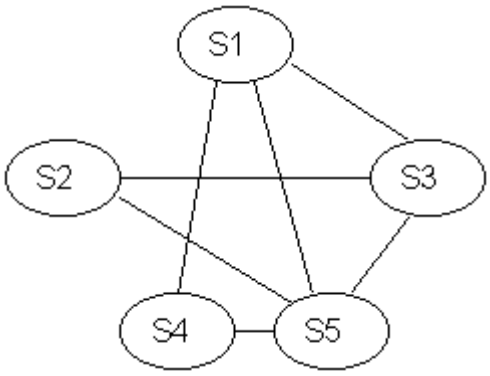


Figura 4: Grafo de Escalonamento com Informações de Dependência de Máquina

Passo 4:

Preenche-se as arestas que ainda não estão preenchidas e retiram-se as demais. O novo grafo é o grafo de complemento, que apresenta o paralelismo existente na máquina para um dado programa. A Figura 5 apresenta, à direita, este grafo.

Realmente as arestas do grafo à direita representam as únicas operações que podem ser executadas em paralelo de acordo com o modelo da máquina usada.

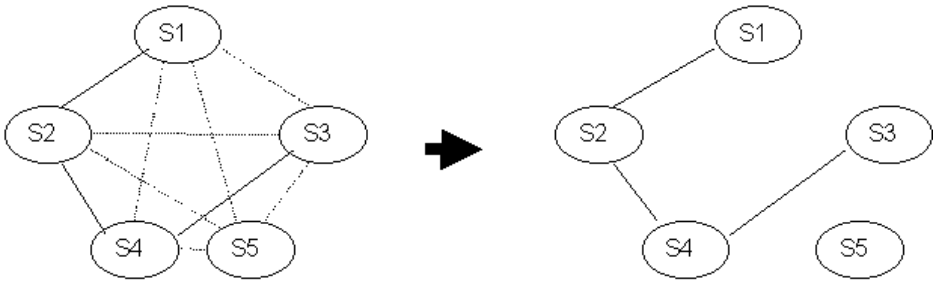


Figura 5: Grafo Complemento

3.4 Construção do Grafo de Interferência Paralelizável

Agora basta integrar os grafos de interferência apresentado na Figura 2 e o grafo de complemento mostrado na Figura 5 e ilustrados na Figura 6 para obter o *Grafo de Interferência Paralelizável*.

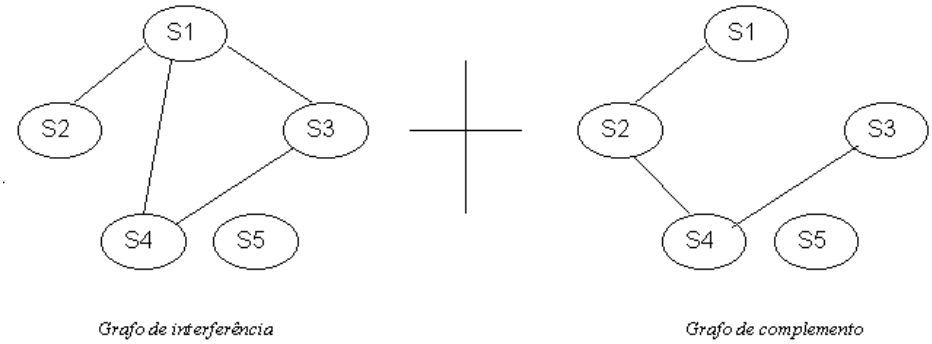


Figura 6: Integração dos Grafos de Interferência e Complemento

Após a construção do *Grafo de Interferência Paralelizável* ilustrado na Figura 7, o algoritmo de coloração de grafos [2] pode ser aplicado ao mesmo. O resultado esperado é uma ótima alocação de registradores que não limite as possibilidades de escalonamento, isto é, o paralelismo da máquina. Para o escalonamento de instruções pode ser usado o grafo de escalonamento original. E o ponto principal é que agora a alocação de registradores pode ser feita primeiro sem prejudicar o paralelismo existente.

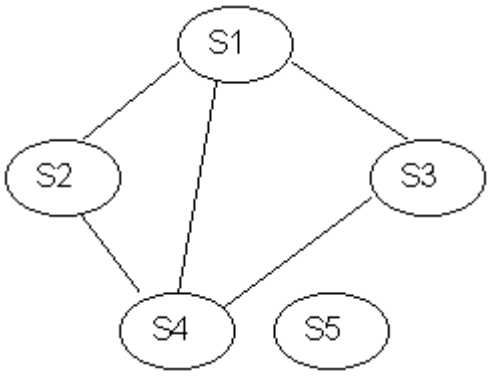


Figura 7: *Grafo de Interferência Paralelizável*

4 Metodologia de Implementação do Algoritmo de Pinter

A implementação do algoritmo de Pinter, que é o objetivo deste projeto teve várias dificuldades. Dentre elas, a coloração do grafo de interferência e escalonamento. As Seções 4.1 e 4.2 introduzem os algoritmos de coloração dos grafos de escalonamento e interferência, respectivamente. As Seção 4.3 e 4.4 apresentam, respectivamente, a estrutura de dados utilizada na implementação do grafo de interferência paralelizável e a implementação propriamente dita .

4.1 Escalonamento de Instrução

Os algoritmos de escalonamento de instruções considerados ótimos, excetos em alguns casos, são NP-completos, mesmo não levando em consideração os recursos computacionais e restrições de registradores. Conseqüentemente, heurísticas são utilizadas com freqüência.

A abordagem mais utilizada para escalonamento de instruções, denominada *lista de escalonamento* [5],[3] funciona da seguinte forma: dado um DAG de código, o escalonador mantém uma lista de instruções que estão aptas para serem escalonadas sem provocar atrasos. A cada iteração utiliza-se uma heurística para selecionar uma instrução pronta para ser escalonada e então atualiza-se a lista. Em geral, esta abordagem possui, no pior caso, um tempo de execução equivalente a $O(e)$, onde e refere-se ao número de arestas no DAG, mas a heurística pode crescer em complexidade.

Em uma lista de escalonamento, normalmente escolhe-se o vértice que possui a prioridade mais elevada em relação aos outros na lista. A heurística mais utilizada para atribuir prioridades denomina-se *distância máxima* e é definida como o comprimento do caminho mais longo através do DAG de código, partindo do vértice da instrução até o vértice folha. O comprimento de um caminho é a soma de todos os valores de rótulos das arestas ao longo do caminho. Muitas vezes, a *distância máxima* é também denominada *peso* do vértice dentro do DAG. O raciocínio utilizado é que o vértice mais longe de ser atingido é o mais crítico, vértices menos importantes podem ser escalonados mais tarde.

A distância máxima é considerada uma heurística inicial relativamente satisfatória, mas existem situações onde heurísticas alternativas ou complementares são mais vantajosas. Por exemplo, dado o DAG da Figura 8, o escalonamento b;a;c;d;e é ótimo. Entretanto, utilizando a heurística da distância máxima, o escalonamento a;c;b;-;d;e é igualmente selecionado porque a,b e c possuem a mesma prioridade. O atraso ocorre porque b possui uma latência de dois ciclos. Um segundo nível de heurística que atribui uma prioridade mais alta para *vértices possuindo mais sucessores* escolheria corretamente b neste caso. A ídeia por trás desta heurística é que escalonando um vértice com mais sucessores, criam-se mais oportunidades para o escalonador nos ciclos subsequentes e permite-se que mais vértices estejam prontos para serem escalonados mais cedo.

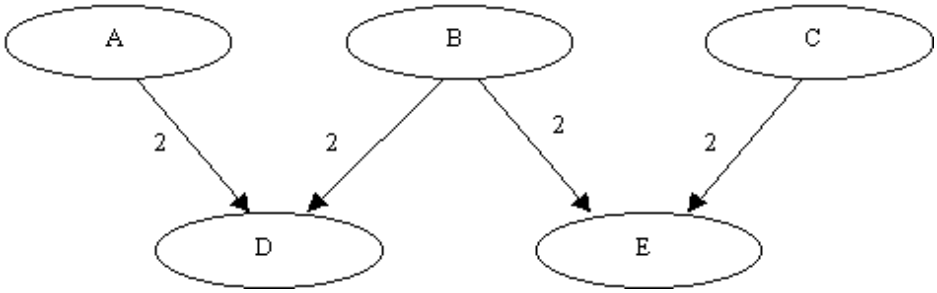


Figura 8: Escalonamento de Instrução

4.2 Alocação de Registradores

Uma boa alocação de registradores é difícil. Alocadores robustos devem lidar bem com programas complexos e números inadequados de registradores. A construção e coloração de grafo de interferência representa um passo essencial para a elaboração de um algoritmo para alocação de registradores. Coloração representa atribuir a cada nó um valor de cor que seja diferente de todos os seus vizinhos. Isto significa que se houver menos registradores que vizinhos será necessário um derramamento de variáveis para a memória. Os nós do grafo representam as variáveis e as arestas, a interferência entre estas variáveis. Logo se o grafo puder ser colorido por até K cores, e o processador tiver K registradores então não haverá derramamento de valores para memória.

O objetivo da coloração do grafo é minimizar o derramamento de valores para memória. Pois estas instruções necessariamente provocam uma bolha no *pipeline* do processador.

Coloração de grafo é um problema NP-Completo e portanto não há uma solução de ordem polimonial, a menos que P=NP. Normalmente são usadas heurísticas, porém não há garantia que a coloração será ótima.

Para a implementação da alocação serão utilizadas as heuristicas abaixo, proposta por Chaitin [2]:

- Remove-se os nós repetidamente do grafo e os insere na pilha;
- O número de vizinhos é contado somente entre os nós restantes no grafo;
- Se o número de vizinhos do nó retirado for maior ou igual ao número de registradores físicos, então marca-se este nó como sendo um candidato ao futuro derramamento para memória;
- Quando o grafo estiver vazio, os nós são retirados da pilha e recolocados no grafo;
- Quando este nó voltar da pilha para o grafo deve-se colorí-lo com uma cor diferente de seus vizinhos. Caso este nó estiver marcado para derramamento, então deve-se gerar código para tratamento no momento de sua devolução.
- A coloração dos nós é trivial pois pode se utilizar a primeira cor livre diferente dos vizinhos. A escolha da cor não altera a complexidade do problema.

Para ajudar a compreensão do algoritmo devemos observar a Figura 9, que em seu lado esquerdo representa a fase de empilhamento das instruções e em seu lado direito o retorno das instruções ao grafo, já se colorindo os nós com cores diferentes das de seus vizinhos.

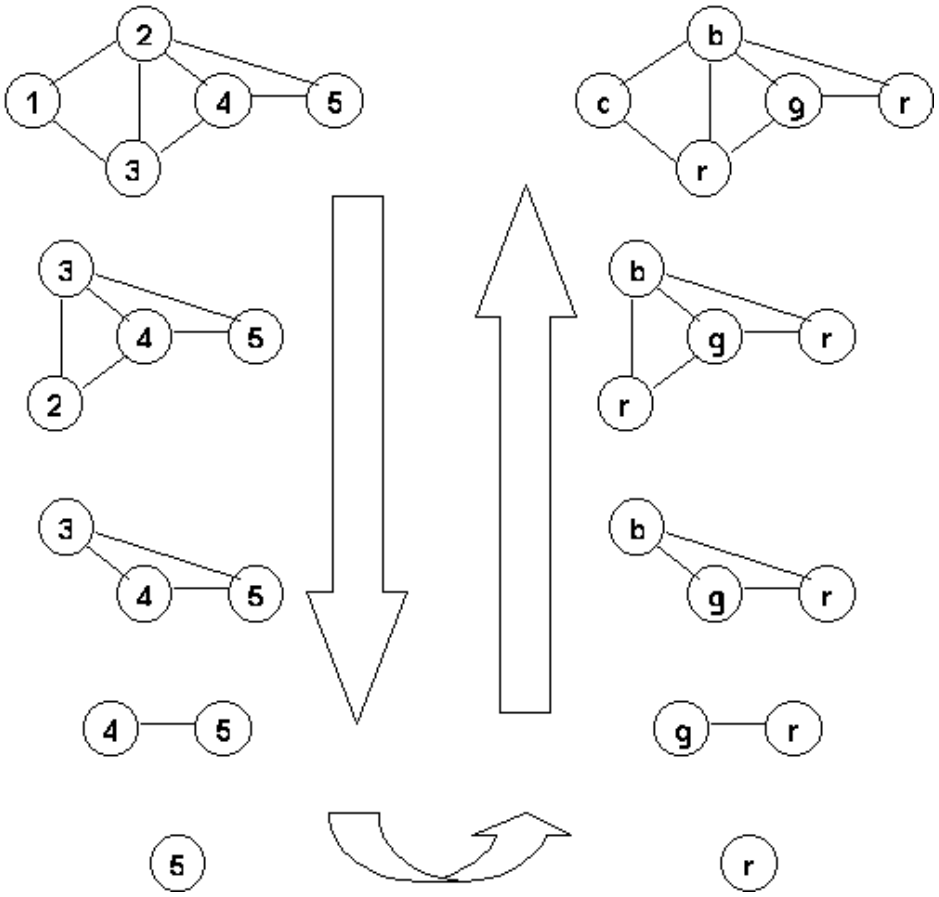


Figura 9: Coloração de um grafo simples

4.2.1 Problemas do Método de Chaitin

Suponha um grafo simples, como o da Figura 10, e suponha ainda que seu processador tenha somente dois registradores físicos. O grafo da Figura 10 pode ser facilmente colorido atribuindo a mesma cor para x e para y e outra cor para w e z. Porém utilizando o método de Chaitin, por qualquer um dos nós que se inicie, o algoritmo falha porque os nós sempre terão o mesmo número de ligações com os vizinhos que o número de registradores físicos disponíveis. Neste exemplo, o algoritmo de Chaitin não acha uma coloração ótima quando a mesma existe. É claro que não é surpreendente, pois o problema é NP-Completo e isto é somente uma heurística.

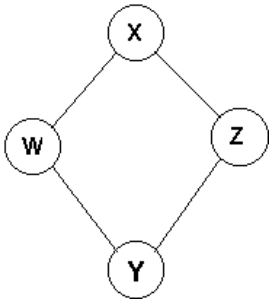


Figura 10: Um grafo simples que requer duas cores

4.3 Estruturas de Dados

A mesma estrutura de dados foi utilizada para a construção do grafo de interferência e o grafo de escalonamento. O grafo implementado pressupõe arestas direcionadas. Porém o grafo de interferência não possui direção nas arestas. O problema foi resolvido colocando duas arestas, apontando de um nó a outro e vice-versa. A estrutura do grafo possui um nó que contém informações sobre cor, instrução, pseudo-registrador, apontador para estrutura de ligação, apontador para o próximo nó. A estrutura de ligação aponta para os nós os quais o nó em questão se liga e para o próximo campo da estrutura de ligação. Esta estrutura está ilustrada na Figura 11.

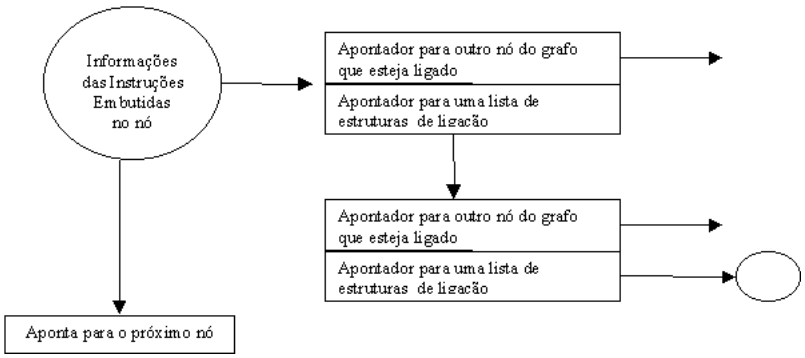


Figura 11: Estrutura de Dados Utilizada para Construção do Grafo

4.4 Implementação do Grafo de Interferência Paralelizável

A implementação do algoritmo de Pinter foi elaborada seguindo os passos que foram explicados na Seção 3. A seguir as funções que o implementaram:

Fecho Transitivo

```
static void FechoTransitivo (struct Indicador *Ind,int
```

```
NumEstados) { int aux,aux2,aux3;

   for (aux=0;aux<NumEstados;aux++)
      for (aux2=0;aux2<NumEstados;aux2++)
     if (SeLigado(RetornaEstado(aux,Ind),RetornaEstado(aux2,Ind),1))
        for (aux3=0;aux3<NumEstados;aux3++)
           if (SeLigado(RetornaEstado(aux2,Ind),RetornaEstado(aux3,Ind),1))
          LigaEstados(RetornaEstado(aux3,Ind),RetornaEstado(aux,Ind),1);
    Envelhece(Ind,NumEstados);
};
```

O fecho transitivo consiste em ligar nós que estajam ligados transitivamente. Exemplificando, se há uma aresta $(S1, S2)$ e uma aresta $(S2, S3)$, então deve existir uma aresta direcionada $(S1, S3)$. A construção do Grafo de Complemento, Seção 3.3 requer o fecho transitivo do grafo no passo 2.

Retirar Direção das Arestas

```
static void RetirarDirecao(struct Indicador *Ind, int NumEstados)
{ int aux,aux2;

   for (aux=0;aux<NumEstados;aux++)
      for (aux2=0;aux2<NumEstados;aux2++)
     if (SeLigado(RetornaEstado(aux,Ind),RetornaEstado(aux2,Ind),1))
        LigaEstados(RetornaEstado(aux2,Ind),RetornaEstado(aux,Ind),1);
   Envelhece(Ind,NumEstados);
}
```

Para a execução do passo 2 da Contrução do Grafo de Complemento, visto na Seção 3.3 é preciso retirar a direção das arestas.

Troca de Arestas

```
static void TrocarArestas(struct Indicador *Ind, int NumEstados) {
int aux,aux2;

   for (aux=0;aux<NumEstados;aux++)
      for (aux2=0;aux2<NumEstados;aux2++)
     if (SeLigado(RetornaEstado(aux,Ind),RetornaEstado(aux2,Ind),1))
        DesligaEstados(RetornaEstado(aux,Ind),RetornaEstado(aux2,Ind));
     else if (aux!=aux2)
        LigaEstados(RetornaEstado(aux,Ind),RetornaEstado(aux2,Ind),1);

}
```

Esta função troca as arestas do grafo, isto é, dois nós anteriormente ligados são desligados e vice-versa. Isto é necessário para o passo 4 da construção do grafo de complemento, conforme visto na Seção 3.3.

União dos Grafos

```
static void UnirGrafos(struct Indicador *Grafo1, struct Indicador
*Grafo2, struct Indicador *Grafo3, int NumEstados) { int aux,
aux2;

   for (aux=0;aux<NumEstados;aux++)
      for (aux2=0;aux2<NumEstados;aux2++)
     if ((SeLigado(RetornaEstado(aux,Grafo1),RetornaEstado(aux2,Grafo1),0))
        || (SeLigado(RetornaEstado(aux,Grafo2),RetornaEstado(aux2,Grafo2),0)))
           LigaEstados(RetornaEstado(aux,Grafo3),RetornaEstado(aux2,Grafo3),0);

}
```

Finalmente deve-se unir os Grafos de Complemento de Interferência para a obtenção do Grafo de Interferência Paralelizável. Este passo está explicado na Seção 3.4

Demais Considerações

- Foi utilizada uma função envelhece que é um marcador se a aresta já existia no grafo anteriormente ou foi colocada na fase corrente.
- Demais funções da biblioteca *GRAPH.H* se refere a manipulação dos grafos ou da interface referente a elaboração destes.

4.4.1 Exemplos

O seguinte exemplo foi gerado utilizando o programa que implementa o algoritmo de Pinter. É um exemplo simples mas demonstra a eficiência do algoritmo que até diminuiu uma instrução e utilizou o menor número de registradores possíveis.

O código mnemônico de entrada foi:
load R1,300
loadlit R2,100
load R3,R2
add R4,R1,R1
add R5,R3,R1
O codigo de saída foi o mostrado abaixo:
load S1,300
loadlit S2,100
add S3,S1,S1
add S2,S2,S1

Observe que o arquivo de entrada possui R# que representa os pseudo-registradores, e o arquivo de saída contém S# que referencia os registradores físicos

5 Conclusão

Tendo por objetivo resolver o problema da interdependência entre alocador de registradores e o escalonador de instruções, Pinter [4] propôs uma nova estratégia. Esta estratégia foi descrita neste documento e diz que utilizando algoritmos bem estabelecidos na literatura para coloração de grafos e para escalonamento de instruções, como por exemplo, os propostos por Chaitin [2], é possível efetuar a alocação de registradores antes do escalonamento de instruções sem perda do paralelismo e sem introduzir dependências falsas no grafo final de escalonamento.

Neste trabalho foi implementado uma versão do processador μRisc. Além disto foi implementado o algoritmo de Pinter **??** e o algoritmo de coloração de grafo proposto por Chaitin **??**. Portanto o objetivo do trabalho foi alcançado.

Como trabalho futuro seria interessante integrar esta abordagem em um sistema de geradores de código para poder fazer medidas de desempenho em relação as outras abordagens existentes.; por exemplo abordagens onde não há nenhuma iteração entre as funções para alocar registradores e o escalonamento de instruções.

A Código do Algoritmo de Pinter

Arquivo Graph.H

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

typedef struct Indicador {
   int               Valor;
   struct Indicador  *Next;
   struct Grafo      *apGrafo;
} Indi;

typedef struct Estado {
   int Codigo;
} Est;

typedef struct Campos {
   struct Estado Valor;
   struct Grafo  *apGrafo;
   struct Campos *Next;
   int NOVO;
} Camp;

typedef struct Grafo {
   struct Estado Valor;
   struct Campos *Ligantes;
} Gra;


static struct Grafo *CriaEstado(int S1) { struct Grafo *Estado1;

    Estado1=malloc(sizeof(struct Grafo));
    if (Estado1)
    {
       Estado1->Ligantes=malloc(sizeof(struct Campos));
       Estado1->Ligantes->apGrafo=NULL;
       Estado1->Ligantes->Next=NULL;
       Estado1->Valor.Codigo=S1;
       return(Estado1);
    }
    else
    {
    printf("ERRO: Memria insuficiente para alocar um novo Estado");
    return(0);
    }
}

static void LigaEstados(struct Grafo *S1, struct Grafo *S2, int
```

```
Novo) { struct Campos *AuxLigantes;

   AuxLigantes=S1->Ligantes;
   while (S1->Ligantes->apGrafo!=NULL) S1->Ligantes=S1->Ligantes->Next;
   S1->Ligantes->apGrafo=S2;
   S1->Ligantes->Next=malloc(sizeof(struct Campos));
   S1->Ligantes->Valor=S2->Valor;
   S1->Ligantes->NOVO=Novo;
   S1->Ligantes=S1->Ligantes->Next;
   S1->Ligantes->Next=NULL;
   S1->Ligantes->apGrafo=NULL;
   S1->Ligantes=AuxLigantes;
}

static struct Grafo *RetornaEstado(int Chave, struct Indicador
*Ind) {
   while (Ind->Next)
   {
      if (Ind->Valor==Chave) return(Ind->apGrafo);
      Ind=Ind->Next;
   }
   return(NULL);
};

static struct Grafo *SeLigado(struct Grafo *S1, struct Grafo *S2,
int Verifica) { struct Campos *AuxLigantes;

   AuxLigantes=S1->Ligantes;
   while (S1->Ligantes->apGrafo!=NULL)
   {
      if (S1->Ligantes->Valor.Codigo==S2->Valor.Codigo)
     if (Verifica)
     {
        if (!(S1->Ligantes->NOVO))
        {
           S1->Ligantes=AuxLigantes;
           return(S1->Ligantes->apGrafo);
        }
     }
     else
     {
        S1->Ligantes=AuxLigantes;
        return(S1->Ligantes->apGrafo);
     }
      S1->Ligantes=S1->Ligantes->Next;
   }
   S1->Ligantes=AuxLigantes;
   return(NULL);
}
```

```
static void Envelhece(struct Indicador *Ind, int NumEstados) { int
aux; struct Grafo *S1; struct Campos *AuxLigantes;

   for (aux=0;aux<NumEstados;aux++)
   {
      S1=RetornaEstado(aux,Ind);
      AuxLigantes=S1->Ligantes;
      while (S1->Ligantes->Next!=NULL)
      {
     S1->Ligantes->NOVO=0;
     S1->Ligantes=S1->Ligantes->Next;
      }
      S1->Ligantes=AuxLigantes;
   }
}

static void FechoTransitivo (struct Indicador *Ind,int
NumEstados) { int aux,aux2,aux3;

   for (aux=0;aux<NumEstados;aux++)
      for (aux2=0;aux2<NumEstados;aux2++)
     if (SeLigado(RetornaEstado(aux,Ind),RetornaEstado(aux2,Ind),1))
        for (aux3=0;aux3<NumEstados;aux3++)
           if (SeLigado(RetornaEstado(aux2,Ind),RetornaEstado(aux3,Ind),1))
          LigaEstados(RetornaEstado(aux3,Ind),RetornaEstado(aux,Ind),1);
    Envelhece(Ind,NumEstados);
};

static void RetirarDirecao(struct Indicador *Ind, int NumEstados)
{ int aux,aux2;

   for (aux=0;aux<NumEstados;aux++)
      for (aux2=0;aux2<NumEstados;aux2++)
     if (SeLigado(RetornaEstado(aux,Ind),RetornaEstado(aux2,Ind),1))
        LigaEstados(RetornaEstado(aux2,Ind),RetornaEstado(aux,Ind),1);
   Envelhece(Ind,NumEstados);
}

static void LimpaTela() { int aux;

   for (aux=0;aux<80;aux++) printf("\n");
}

static void DesligaEstados (struct Grafo *S1, struct Grafo *S2) {
struct Campos *AuxLigantes; struct Campos *Prior; struct Campos
*Next;

   AuxLigantes=S1->Ligantes;
   Prior=NULL;
   while (S1->Ligantes->Valor.Codigo!=S2->Valor.Codigo)
```

```
   {
      Prior=S1->Ligantes;
      S1->Ligantes=S1->Ligantes->Next;
   }
   if (!Prior)
   {
      Prior=S1->Ligantes;
      S1->Ligantes=S1->Ligantes->Next;
      free(Prior);
   }
   else
   {
      Next=S1->Ligantes->Next;
      free(S1->Ligantes);
      S1->Ligantes=Prior;
      S1->Ligantes->Next=Next;
      S1->Ligantes=AuxLigantes;
   }
}

static void TrocarArestas(struct Indicador *Ind, int NumEstados) {
int aux,aux2;

   for (aux=0;aux<NumEstados;aux++)
      for (aux2=0;aux2<NumEstados;aux2++)
     if (SeLigado(RetornaEstado(aux,Ind),RetornaEstado(aux2,Ind),1))
        DesligaEstados(RetornaEstado(aux,Ind),RetornaEstado(aux2,Ind));
     else if (aux!=aux2)
        LigaEstados(RetornaEstado(aux,Ind),RetornaEstado(aux2,Ind),1);

}

static void ImprimeGrafo(struct Indicador *Ind, int NumEstados) {
int aux; struct Grafo *S1; struct Campos *AuxLigantes;

   printf("Grafo com %d estados --> [0-%d].\n",NumEstados, NumEstados-1);
   for (aux=0;aux<NumEstados;aux++)
   {
      S1=RetornaEstado(aux,Ind);
      AuxLigantes=S1->Ligantes;
      while (S1->Ligantes->Next)
      {
     if (aux < S1->Ligantes->Valor.Codigo)
        printf("%d-%d ",aux,S1->Ligantes->Valor);
     S1->Ligantes=S1->Ligantes->Next;
       }
       printf("\n");
       S1->Ligantes=AuxLigantes;
    }
}
```

```
static void UnirGrafos(struct Indicador *Grafo1, struct Indicador
*Grafo2, struct Indicador *Grafo3, int NumEstados) { int aux,
aux2;

   for (aux=0;aux<NumEstados;aux++)
      for (aux2=0;aux2<NumEstados;aux2++)
     if ((SeLigado(RetornaEstado(aux,Grafo1),RetornaEstado(aux2,Grafo1),0))
        || (SeLigado(RetornaEstado(aux,Grafo2),RetornaEstado(aux2,Grafo2),0)))
           LigaEstados(RetornaEstado(aux,Grafo3),RetornaEstado(aux2,Grafo3),0);

}
```

Referências

[1] Mariza A. Silva Bigonha. *Otimização de Código em Máquinas Superescalares.* PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informàtica-PUC/RJ, 1994.

[2] Gregory J. Chaitin. Register allocation and spilling via graph coloring. *ACM Sigplan Notices*, 17(6), June 1982. ACM Sigplan Symposium on Compiler Construction, 1982.

[3] James R. Goodman and Wei-Chung-Hsu. Code Scheduling and register allocation in large basic blocks. In *International Conference on Supercomputing - Conference ACM-PRESS Proceedings*, St. Malo France, July 1988.

[4] Shlomit S. Pinter. Register Allocation with instruction scheduling: a New Approach. In *Proccedings of the ACM Sigplan'93 Conference on Programming Language Design and Implementation*, Albuquerque NM, June 1993. ACM Sigplan Notices - Vol. 28 number 6 june-1993

[5] Bradlee, David G. and Eggers, Susan J. and Henry, Robert R., The Marion System for Retargetable Instruction Scheduling, ACM Sigplan Conference on Programming Language Design and Implementation, 1991, July, ACM Sigplan Notices 26(7)

[6] D.A., Patterson and J.L., Henessy. In *Computer Organization & Design - The Hardware/Software Interface* Morgan Kaufmann Publishers, Inc., 1994

Belo Horizonte, 20 de Julho de 1999.

Nizam de Abreu Pfeilsticker

Mariza Andrade da Silva Bigonha