Sistema Gerador de Geradores de Código para Arquiteturas Superescalares

Mariza Andrade da Silva Bigonha – DCC/UFMG

José Lucas Mourão Rangel Netto – PUC/RJ

VII Simpósio Brasileiro de Arquitetura de Computadores e Processamento Paralelo

Agosto/1995

Roteiro da Apresentação

I - Arquiteturas RISC

II - Problemas da Geração e Otimização de Código

III Objetivos

IV - Sistema Gerador de Geradores de Código (GGCO)

- Escalonamento de Instruções
- Alocação de Registradores
- Interdependência entre o Escalonamento e Alocação
- Generalização dos Algoritmos de Escalonamento e Alocação
  - Pares de Registradores
- Linguagem de Descrição de Arquitetura (LDA)
  - Registradores de Código de Condição
  - Janela de Registradores
  - Esquema de Prioridade na Arquitetura M88000

V - Formalização da LDA

VI - Trabalhos Realizados

VII - Conclusões

I - Arquiteturas RISC

- Execução de qualquer instrução em um único ciclo.
- As únicas instruções que podem referenciar a memória são LOAD e STORE. As demais devem operar sobre registradores.
- Relativamente poucas instruções.
- Relativamente poucos modos de endereçamento.
- Formato de instrução fixo.
- Exigência de compiladores mais "inteligentes".
- Uso intensivo de mecanismos de PIPELINING.

Arquiteturas Superescalares

- Segunda Geração de Arquiteturas RISC
- Processadores com várias unidades funcionais.
- Emissão de mais de uma instrução por ciclo

Geradores de Código para RISC

- Implementam a maior parte das operações de uma única forma.

- Expõem ao gerador e otimizador a estrutura do *pipeline* e custos das unidades funcionais.

Necessidade de Otimização

- Otimização é indispensável para usufruir de todas as vantagens das características da arquitetura.

- O fato de possuir um grande número de registradores impõe ao compilador a tarefa de usá-los eficientemente.

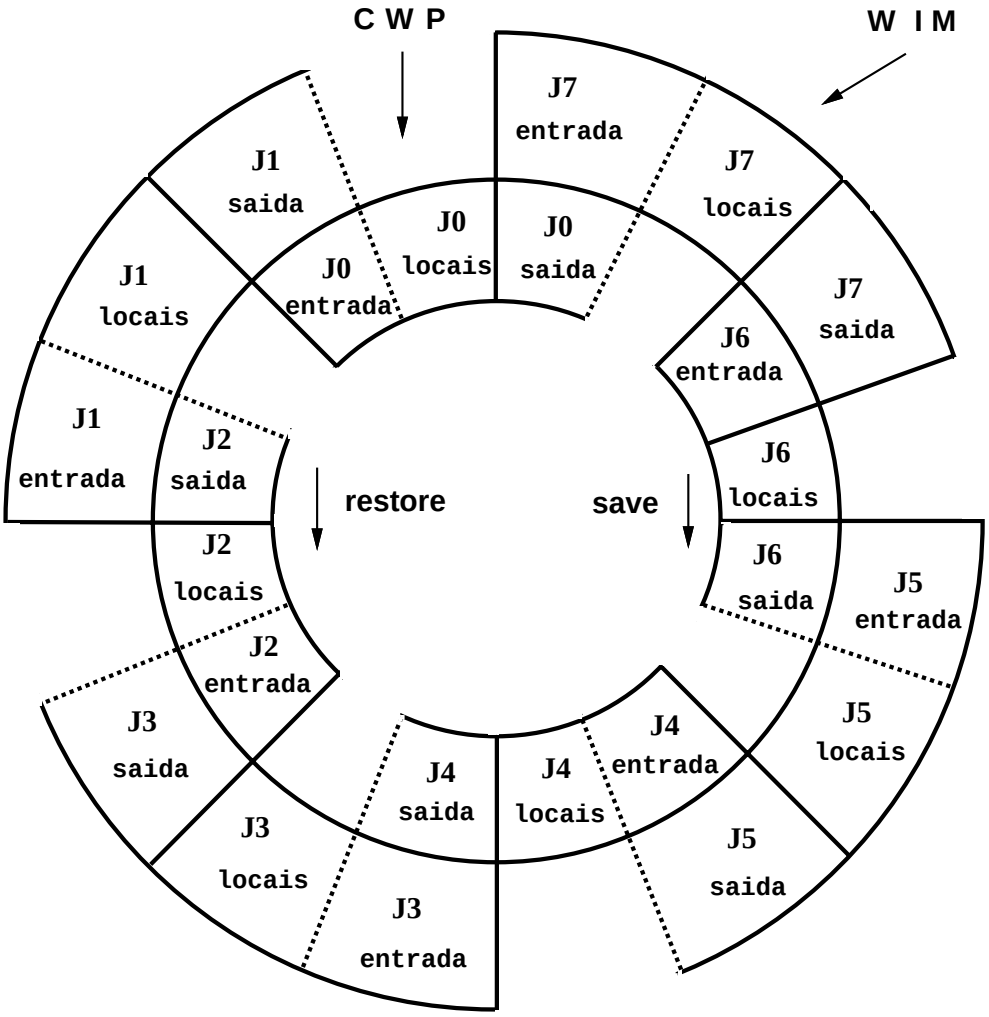
- Algoritmos de escalonamento são utilizados para reduzir os atrasos atribuídos ao tempo de execução no código compilado.

II - Problemas da Geração e Otimização

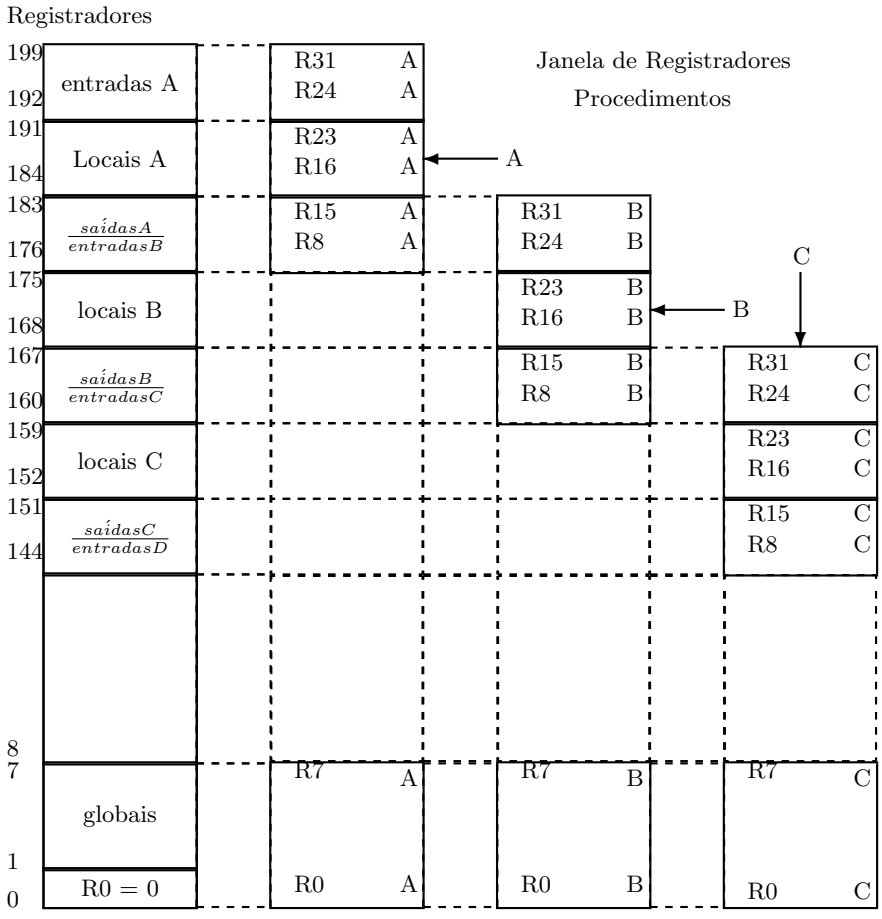
- Redirecionamento auxiliado por computador.
  - Linguagem de Descrição de Arquitetura.
- Alocação de Registradores.
- Escalonamento de Instruções.
  - Atrasos causados por dependências de dados.
  - Atrasos causados pelas instruções de desvios.
  - Presença de restrições impostas pelos processadores *pipelines*, como, por exemplo, a presença ou não de *interlocks* por *hardware*.
- Interdependência entre a alocação de registradores e o escalonamento de instruções.

Continuação: Problemas da Geração e Otimização

- **Janela de Registradores**



Exemplo de três Procedimentos utilizando Janela de Registradores



Continuação: Problemas da Geração e Otimização

- **Pares de Registradores**

- **Registradores de Código de Condição**

- **Esquema de Prioridade do MC88000**

  Disputa para ter acesso a reg. no barramento destino

  - Unidade de Ponto-flutuante
  - Unidade de Inteiro
  - Unidade de Dados

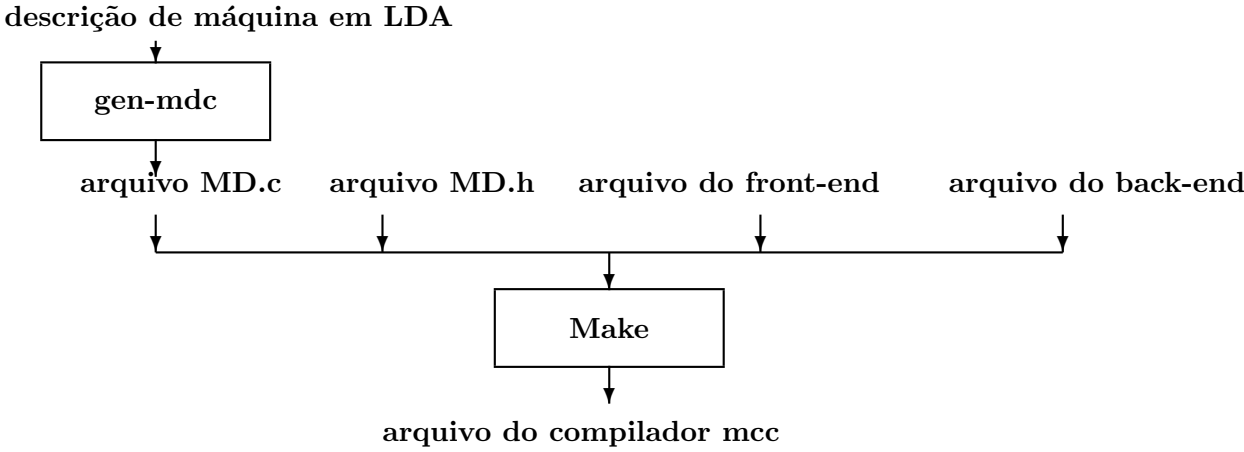
III - Objetivos do Trabalho

- O estudo dos problemas relacionados com a geração de código para arquiteturas mais complexas, tais como as superescalares.

- O estudo das características de sistemas geradores de geradores de código existentes, e das caracteristicas de sistemas adequados para estas arquiteturas;

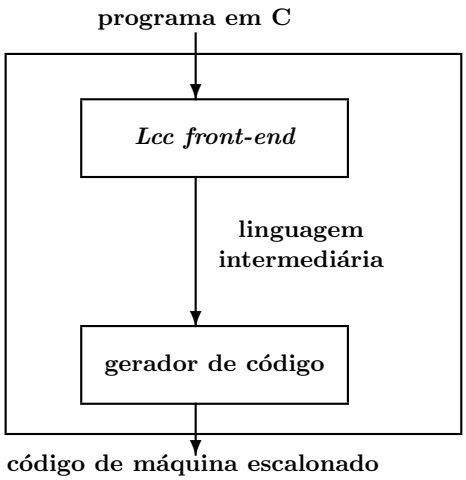
- O estudo das características de linguagens formais de descrição de arquiteturas, apropriadas para uso com geradores de geradores de código.

IV - Sistema Gerador de Geradores de Código

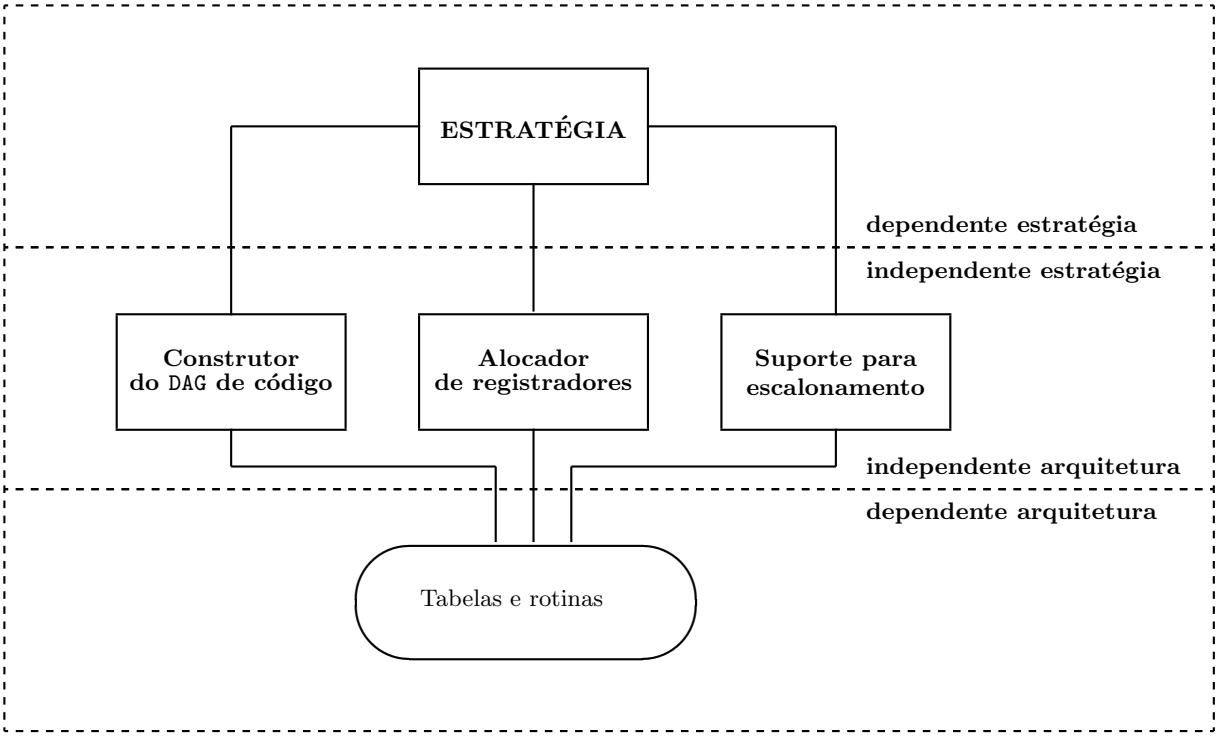


Compilador mcc

- Fase de transformação.
- Fase de construção de blocos básicos.
- Fase de transformações dependentes de arquitetura.
- Fase de casamento de padrões (*match*) e geração de código.
- Fase de estratégia de escalonamento e alocação de registradores.



Estrutura do Módulo Gerador de Código



• Escalonamento de Instruções

- Blocos Básicos.
- Grafo Acíclico Dirigido (DAG).
- Dependências de Dados.
- Escalonamento de Instruções.
- Algoritmos de Escalonamento.

Blocos básicos: *sequência de comandos consecutivos no qual o fluxo de controle entra no seu início e o deixa no final sem interrupções ou possibilidades de desvios, exceto em seu final.*

Grafo Acíclico Dirigido (DAGs)

O grafo de escalonamento: $G_s = (V_s, E_s)$

- Todo vértice $u \in V_s$ corresponde a uma instrução em uma descrição do programa baseada em registradores.

- Existe uma aresta dirigida $(u, v) \in E_s$, de u para v, se u deve ser executado antes de v.

  1. Existe uma dependência de dados de v em u.

  2. Existe uma dependência de controle de u para v.

  3. Existe uma restrição de recurso de máquina que impõe a precedência de u sobre v.

Outras propriedades do grafo de escalonamento

- Arestas são rotuladas com a latência de operação existente entre a aresta de origem e os vértices destinos.
- A *raiz* do DAG.
- Uma *folha*.
- Uma *aresta (u,v) com rótulo l*.
- O DAG é ligado por um encadeamento especial que representa a ordem inicial das instruções dentro do bloco básico.
  - O encadeamento é construído percorrendo o bloco básico da última até a primeira instrução, anotando cada definição ou uso de um operando, e relacionando os usos e as definições que devem precedê-los.

Dependências de Dados

- Dependências verdadeiras
- Dependências falsas
  - anti-dependência
  - dependência de saída.

1. Dependência verdadeira ou de fluxo de dados$\rightarrow$ registrador definido em u é usado em v.

2. Anti-dependência$\rightarrow$ registrador usado em u é redefinido em v e destrói o valor usado em u.

3. Dependência de saída $\rightarrow$ registrador definido em u é redefinido em v, destruindo o valor definido anteriormente em u.

Pipeline

- Janela onde instruções em execução dentro do pipeline não podem depender uma da outra.

Processador pipelined

Várias instruções seqüenciais executam simultaneamente, normalmente, em fases distintas.

- Instruções de desvios e dependência de dados restringem eficácia do pipeline.
- Surge necessidade de mudar a ordem de execução das instruções para manter pipeline cheio.

Escalonamento de Instruções

Técnica de *software* que rearranja seqüências de código durante a compilação com o objetivo de reduzir possíveis atrasos de execução

Algoritmos de Escalonamento

- Abordagem mais utilizada:
  - Lista de escalonamento.
  - Uso de heurísticas - auxiliam atribuir prioridades as instruções.

Heurísticas mais utilizadas

$\Downarrow$

- Distância máxima.
- Vértices possuindo mais sucessores.
- Vértices possuindo o maior tempo de latência em relação aos sucessores.

• Alocação de Registradores

- Pseudo-registradores criados para conter valores intermediários de expressões.
- Número ilimitado de pseudo-registradores.
- Não existe dependência de *hardware*

alocação de registradores

$\Downarrow$

pseudo-registradores $\rightarrow$ registradores físicos

- A alocação é considerada ótima se as variáveis permanecem nos registradores durante todo seu ciclo de vida.
- Alocação adequada de registradores reduz o número de referência à memória.

Grafo de Interferência

Grafo de Interferência: $G_r = (V_r, E_r)$

- 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.

Método mais utilizado para Alocação

- Coloração de grafos (Chaitin,1982; Briggs, 1989).
  - Coloração de grafos modela o problema de alocação de registradores como um grafo de interferência.
  - Atribui cores diferentes para nodos que interferem.
  - Colorir o grafo $\rightarrow$ atribuir registradores físicos aos pseudo-registrador

Interdependência entre Escalonamento e Alocação

- Alocação de registradores ANTES.

Problema:

pode introduzir dependências atribuindo o mesmo registrador físico para instruções sem nenhuma relação

Desvantagem:

se os registradores forem referenciados no mesmo bloco básico, o escalonador não pode superpor operações que os usam.

pr3	←	pr1 * pr1
store	pr3	
pr4	←	pr1 * pr2
store	pr4	

			1^a escolha	2^a escolha
pr1		pr1	r1	r1
pr2		pr2	r2	r2
pr3		pr3	r3	r3
pr4		pr4	r3	r4
pr5		pr5	r4	spill

ciclos	instruções	ciclos	instruções
0	r3 ← r1 * r1	0	r3 ← r1 * r1
4	store r3	1	r4 ← r1 * r2
5	r3 ← r1 * r2	4	store r3
9	store r3	5	store r4

Alocação Registradores *antes* Escalonamento Instruções

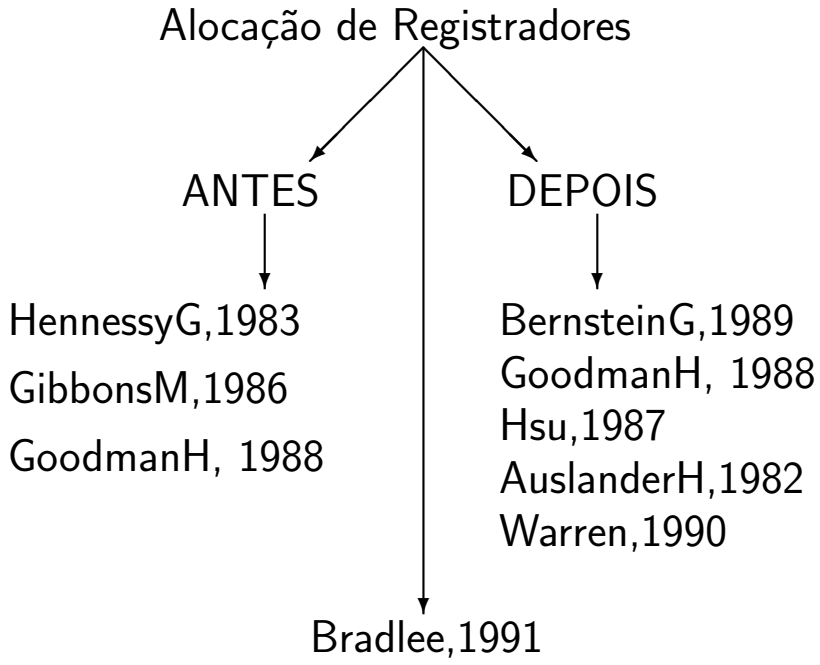
ciclos	instruções
0	load r1, 12(fp)
1	load r2, 16(fp)
2	load r3, 40(fp)
3	load r4, 44(fp)
4	r1 $\leftarrow$ r1 + r2
5	r2 $\leftarrow$ r3 + r4

ciclos	instruções
0	load r1, 12(fp)
1	load r2, 16(fp)
2	load r3, 40(fp)
3	r1 $\leftarrow$ r1 + r2
4	load r2, 44(fp)
6	r2 $\leftarrow$ r3 + r2

Dois escalonamentos para *loads* e *adds* independentes

Escalonamento X Alocação

Soluções Existentes



• Generalização do Algoritmo de Alocação e Escalonamento

Objetivo da Estratégia

- Encontrar um algoritmo ótimo tal que ele faça uso de um número mínimo de registradores físicos
- Que não derrame valores ainda vivos para a memória.
- Cujo grafo de escalonamento não possua dependências falsas.

O ponto principal da estratégia proposta é o desenvolvimento de um alocador de registradores que não restringe a ação do escalonador.

Suposições Básicas

- Grande número de registradores de propósito geral, $r_1, r_2, \cdots, r_n$,
- Células de memória $m_1, m_2, \cdots, m_n$,
- Coleção de unidades funcionais $\rightarrow$ cada uma pode executar uma instrução a cada ciclo de máquina,
- As operações:
  - Operações de carga: $(m_i) \rightarrow (r_j)$.
  - Operações de armazenamento: $(r_i) \rightarrow (m_j)$.
  - Operações aritmeticas: $A((r_{ij}), \cdots, r_{ik})) \rightarrow (r_j)$, onde, $k >= 1$.
- Algoritmo baseado em listas de escalonamento.
- Algoritmo baseado no método de Chaitin.
- Grafos utilizados: G_s e G_r.

Exemplo

$$
\begin{array}{lcl}
\mathsf{x} & := & \mathsf{a[i]} \\
\mathsf{y} & := & \mathsf{z} + \mathsf{z} \\
\mathsf{z} & := & \mathsf{x} * 5 + \mathsf{z}
\end{array}
$$

(1)

$$
\begin{array}{lcl}
\mathsf{s1} & := & \mathsf{load\ z} \\
\mathsf{s2} & := & \mathsf{load\ i} \\
\mathsf{s3} & := & \mathsf{a[s2]} \\
\mathsf{s4} & := & \mathsf{s1} + \mathsf{s1} \\
\mathsf{s5} & := & \mathsf{s3} * 5 + \mathsf{s1}
\end{array}
$$

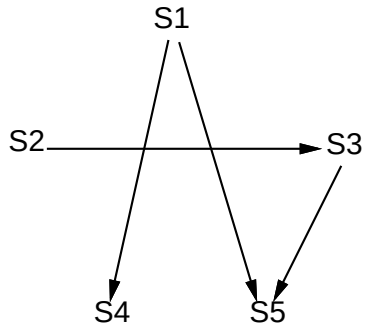
(2)

$$
\begin{array}{lcl}
 & \mathsf{r1} & := \quad \mathsf{load\ z} \\
\Rightarrow & \mathsf{r2} & := \quad \mathsf{load\ i} \\
 & \mathsf{r3} & := \quad \mathsf{a[r2]} \\
\Rightarrow & \mathsf{r2} & := \quad \mathsf{r1} + \mathsf{r1} \\
 & \mathsf{r1} & := \quad \mathsf{r3} * 5 + \mathsf{r1}
\end{array}
$$

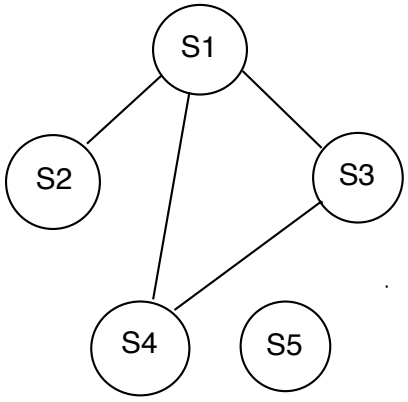
(3)

s1 := load z
s2 := load i
s3 := a[s2]
s4 := s1 + s1
s5 := s3 * 5 + s1

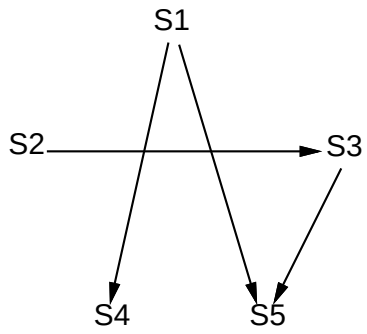
Grafo de Escalonamento G_s



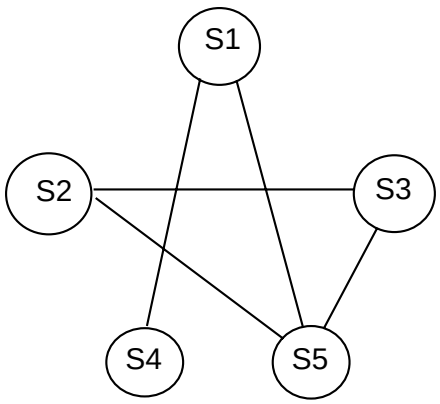
Grafo de Interferência G_r



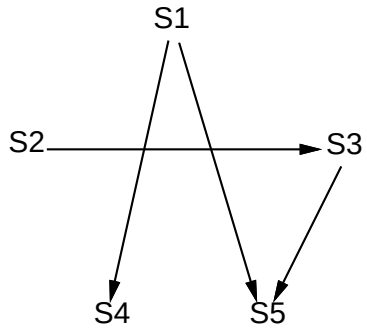
```
s1 := load z
s2 := load i
s3 := a[s2]
s4 := s1 + s1
s5 := s3 * 5 + s1
```



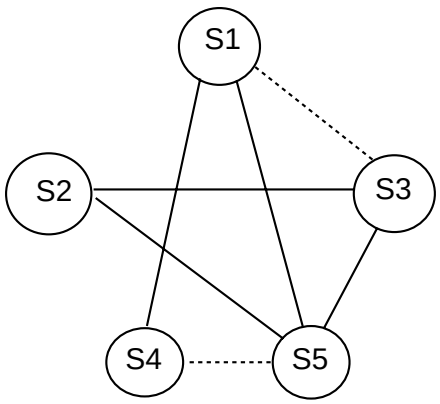
Fecho Transitivo



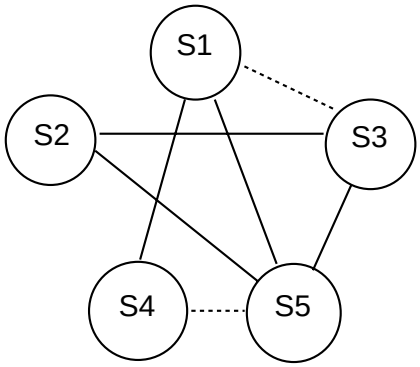
s1 := load z
s2 := load i
s3 := a[s2]
s4 := s1 + s1
s5 := s3 * 5 + s1



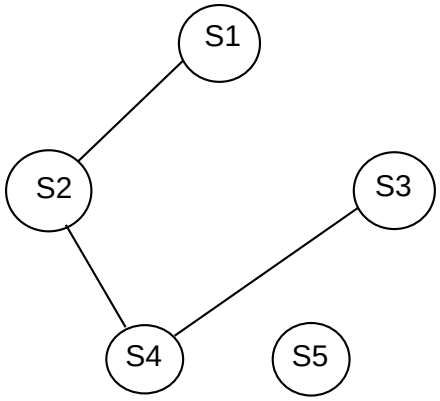
Vértices no Conjunto E_t.



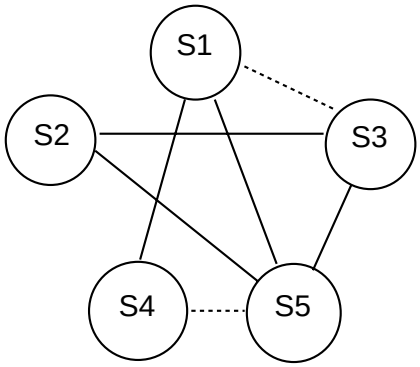
Vértices no Conjunto E_t.



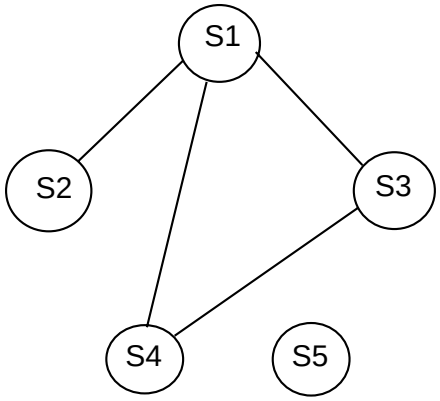
Complemento do Grafo



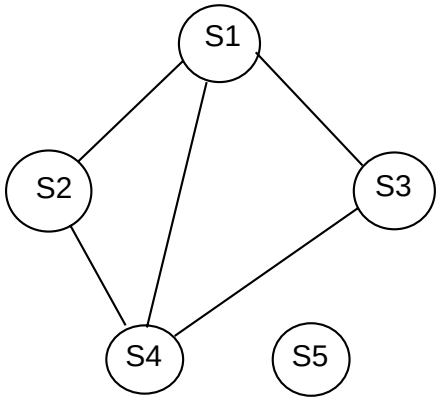
Vértices no Conjunto E_t.



Grafo de Interferência



Grafo de Interferência Paralelizável



Código: Grafo de Interferência Paralelizável

```
     r1  :=  load z
⇒    r2  :=  load i
     r2  :=  a[r2]
⇒    r3  :=  r1 + r1
     r2  :=  r2 * 5 + r1
```

Código: Grafo de Interferência

```
     r1  :=  load z
⇒    r2  :=  load i
     r3  :=  a[r2]
⇒    r2  :=  r1 + r1
     r1  :=  r3 * 5 + r1
```

Derramamento de Código

Alocação de registradores $\rightarrow$ encontrar um mapeamento de registradores tal que o custo de derramamento seja mínimo.

$\Downarrow$

Aplicam-se sobre o *grafo de interferência paralelizável* as mesmas heurísticas usadas tanto na alocação de registradores como no escalonamento de instruções.

Heurísticas $\rightarrow$ eliminam arestas do grafo.

- Arestas que evitam dependências falsas
  algumas opções de paralelismos são perdidas.

- Arestas de interferência as quais podem ocasionar derramamentos de código.

Regras para Derramamento

- Remova todas as arestas de $E - E_r$ para as quais o escalonamento paralelo de duas instruções possui a menor prioridade para o escalonamento.

- Evite a remoção das arestas em $E_f \cap E_r$. Estas arestas são usadas pelo escalonador e pelo alocador.

Outra abordagem

Função heurística $h \rightarrow$

$$h(v) = cost(v)/degree(v)$$

Pares de Registradores

- A necessidade por mais de um registrador para um dado vértice v muda a a definição de *degree*.

$$degree_p \rightarrow \textit{vizinhos de } p \textit{ no grafo de interferência.}$$

- Um vértice v é considerado sem restrições se:
$degree(v)+need(v)$ for menor que o número de registradores alocáveis.

- Um vértice v pode ser removido do *grafo de interferência paralelizável* durante a fase de simplificação se o predicado *is-safe(v)* é verdadeiro;

$$\textit{is-safe(v)} = degree(v) + need(v) \leq r$$

1. $degree(v) =$ soma das necessidades de registradores físicos de seus vizinhos.

2. r representa os registradores físicos disponíveis.

3. $need(v)$ representa o número de registradores físicos necessários à v.

Continuação: Pares de Registradores

Como pares de registradores não possuem nehuma relação com o objetivo do algoritmo proposto, que é manter o paralelismo de instruções evitando dependências falsas no *grafo de interferência paralelizável*, eles podem ser incorporados neste algoritmo como em qualquer outro esquema.

Pontos Positivos desta Abordagem

- O escalonador não é afetado pelo alocador de registradores.
- O grafo de escalonamento continua sendo G_c.
- O grafo para alocar registradores é o *grafo de interferência paralelizável*.
- O *grafo de interferência paralelizável estendido* pode ser usado para as duas funções de escalonamento e alocação.
- O algoritmo de escalonamento não precisa ser tão complicado como RASE, podendo inclusive usar o algoritmo proposto por Gibbon cujas técnicas heurísticas empregadas são simples, contudo capazes de prover um bom escalonamento.

• A Linguagem de Descrição de Arquitetura

1. Características gerais das arquiteturas:
    - Registradores.
    - Estágios de *pipeline*.
    - Unidades funcionais.
    - Memória.
2. Modelo da máquina virtual.
    - Definição do uso dos registradores.
    - Passagem de parâmetros.

3. Lista de Instruções:

   - Instruções padrão

     Para cada instrução de máquina deve ser especificado:
     - Mnemônico da instrução.
     - Restrições de tipo dos operandos.
     - Semântica da instrução.
     - Recursos de *pipeline* necessários.
     - Outras propriedades de escalonamento, como por exemplo. latên custo e número de *slots* associado às operações.
   - Instruções especiais.

4. Detalhes específicos de arquitetura:

   - Transformações necessárias para auxiliar no mapeamento da linguagem intermediária com o conjunto de instruções da máquina alvo.
   - Especificações de latência especiais.

Estrutura da LDA

- Seção de Declarações.
- Seção com as características da Máquina Alvo.
- Seção de Instruções.

Escopo de Aplicação de LDA

- Conjunto de registradores compartilhados
  diretiva: %reg
  arquiteturas: M88000, i860, i960, SPARC, RS/6000

- Pares de registradores
  diretivas: %reg, %equiv
  arquiteturas: M88000, M88110, i860, i960, SPARC

- Janela de registradores
  diretivas: %arg, %par, %calleesave, %callersave
  darquitetura: SPARC

- Registradores com valores pré-definidos por *hardware*
  diretiva: %hard
  darquiteturas: M88000, M88110, i860, i960, SPARC

- Dependência Estrutural
  diretiva: %resource
  arquiteturas: RS2000, M88000, M88110, i860, i960, SPARC, RS/60

- Carga de ponto flutuante de precisão dupla usando duas instruções
  diretiva: %func
  arquiteturas: RS2000

- Desvios com *delay slots*
  diretiva: delay
  arquiteturas: RS2000, M88000, i860, i960, SPARC

- Desvios com *delay slots* executados condicionalmente
  diretiva: delay - valor 1 - a instrução só é executada se o desvio acontecer.
  arquiteturas: i860, i960, SPARC

- Latência de operação dependente do destino da instrução
  diretiva: %aux
  arquiteturas: M88000, M88110, i860, i960

- Múltiplos despachos de instruções
  cada instrução pode ser despachada a cada ciclo.
  arquiteturas: i860, RS/6000

- Múltiplos modos de endereçamentos

diretiva: %reg
arquiteturas: M88000, M88110, i860, i960, RS/6000

- *Pipelines* com avanço explícito
  diretivas: %reg, temporal, %clock, %element, %class
  arquitetura: i860

- Registradores de código de condição
  diretivas: %reg, temporal, %clock
  arquiteturas: i860, i960, RS/6000, SPARC

- Esquema de prioridade por *hardware* para contenção de recursos
  diretiva: %resource
  arquiteturas: M88000, M88110.

Elementos Não-descritíveis em LDA

- Instruções *set-on-condition*
  arquitetura: RS2000

- Instrução de chamada condicional *conditional call instruction*
  arquitetura: RS2000

- Múltiplas unidades funcionais idênticas
  arquiteturas: i960, M88110

- Decremento em *loops* e instruções de desvios
  arquitetura: i860

- Modo de endereçamento com auto-incremento
  arquiteturas: i860, RS/6000

- Carga através de *bypass cache*
  arquitetura: i860

Solução para o Esquema de prioridade para controlar o uso do *write-back bus* do MC88000

- Permitir que uma série de prioridades seja associada com a **declaração de recursos** na descrição da máquina
  IF[10..15] — FI[4..6] — DA[0..3]

- Permitir que um elemento do vetor de recursos seja associado com uma **instrução** para indicar sua prioridade
  %instr add r, r, r (int)
  {$1 = $2 + $3;}
  $[IF.10; ID.11; IE.12; IW.13;]$ (1,1,0)

- Examinar prioridades durante a verificação por
  dependências estruturais e permitir que escalonamentos contenham estas dependências, se eles forem causados por recursos prioritizados.

Solução para o Registrador de Código de Condição

- Declarar o registrador de código de condição como sendo um registrador temporal na Seção de Declarações da LDA.
- Declarar na Seção de Instruções as instruções que o modificam.
- Examinar o registrador de código de condição durante o escalonamento de instruções garantindo que o mesmo está ligado e corretamente referenciado.

V - Formalização de LDA

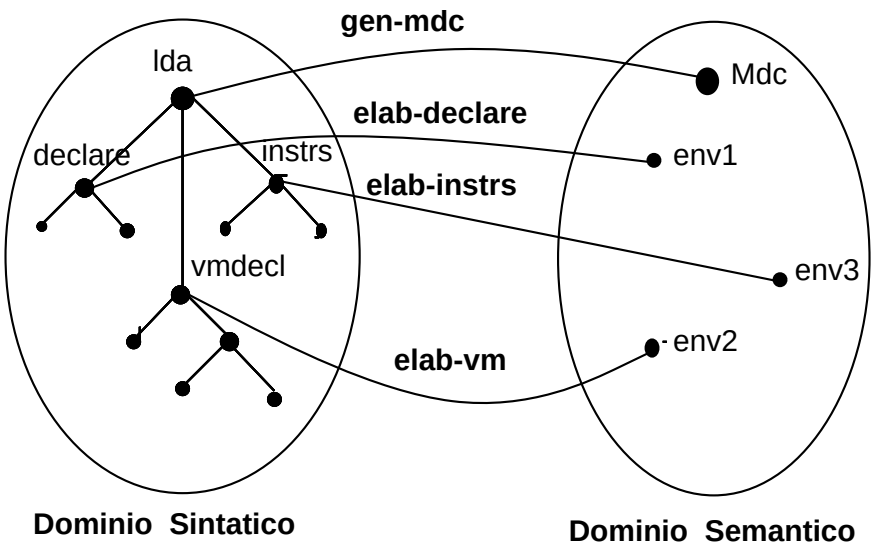
- A definição formal da linguagem de descrição de arquitetura LDA segue o método denotacional para especificação de semântica.

- A definição mostra o mapeamento da descrição de arquitetura de máquinas para o seu respectivo gerador de código.

- O resultado final deste mapeamento é um trecho de programa em linguagem C, que constitui a parte dependente de máquina do gerador de código para a arquitetura descrita.

Módulo gen-mdc

```
DEF gen-mdc(lda) : Mdc =
    LET [declare vmdecl instrs] = lda
    LET env1        = elab-declare(declare)(env0)
    LET env2        = elab-vm(vmdecl) (env1)
    LET env3        = elab-instrs(instrs) (env2)
    LET (env4, mdc) = elab-tables(env3)
    IN  mdc
END Lda
```



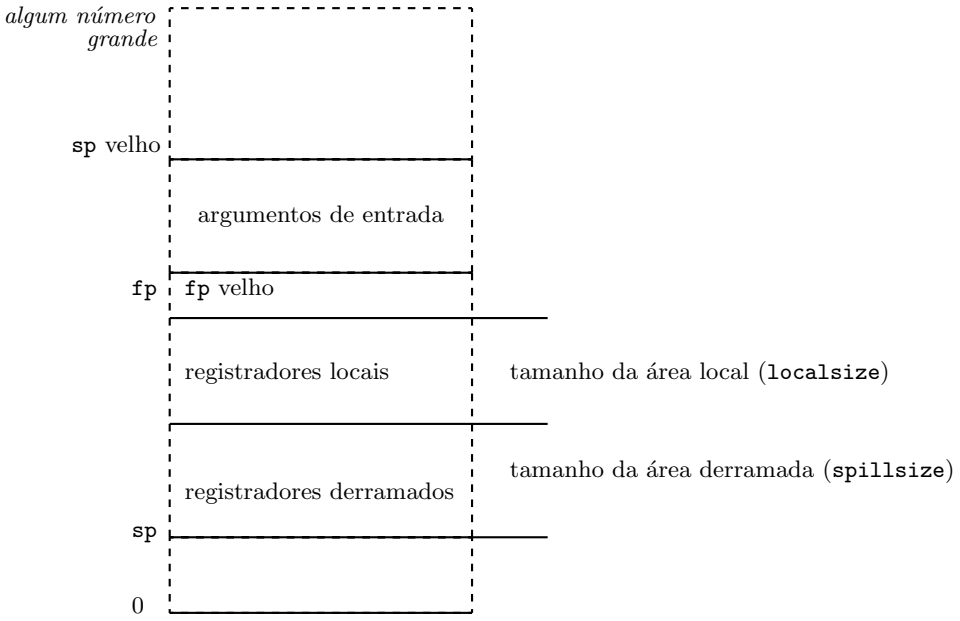
Tabelas Geradas

1. Tabela de Nomes.
2. Tabela de Declarações.
3. Tabela com as Convenções de Chamadas de Procedimentos.
4. Tabela de Recursos Utilizados pelas Instruções.
5. Tabela de Associação dos Elementos (instruções) a Relógios.
6. Tabela de Classes.
7. Tabela de Produções.
8. Tabela de Operadores.
9. Tabela de Registradores usados pelas Instruções.
10. Tabela de Funções Especiais.
11. Tabela de Instruções Especiais.
12. Tabela de Registradores Especiais.
13. Tabela de Instruções `glue` .
14. Tabela de Latências Auxiliares.

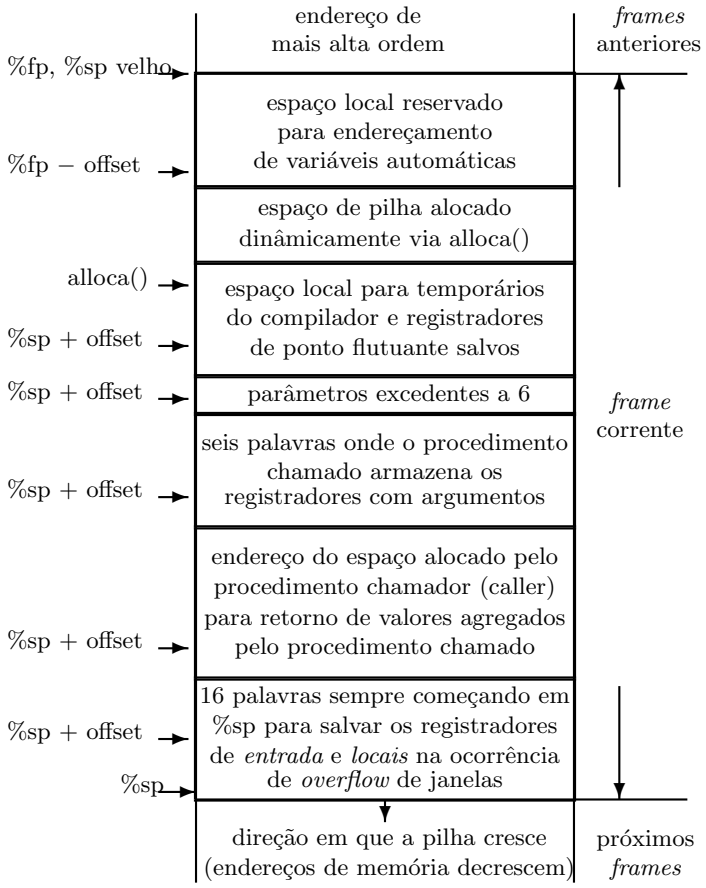
Funções Geradas

1. Função reg_overlap( ).
2. Função cginit( ).
3. Função maptype_declMD( ).
4. Função tsize( ).
5. Função typesizeMD( ).
6. Função regsizeMD( ).
7. Função relocMD( ).

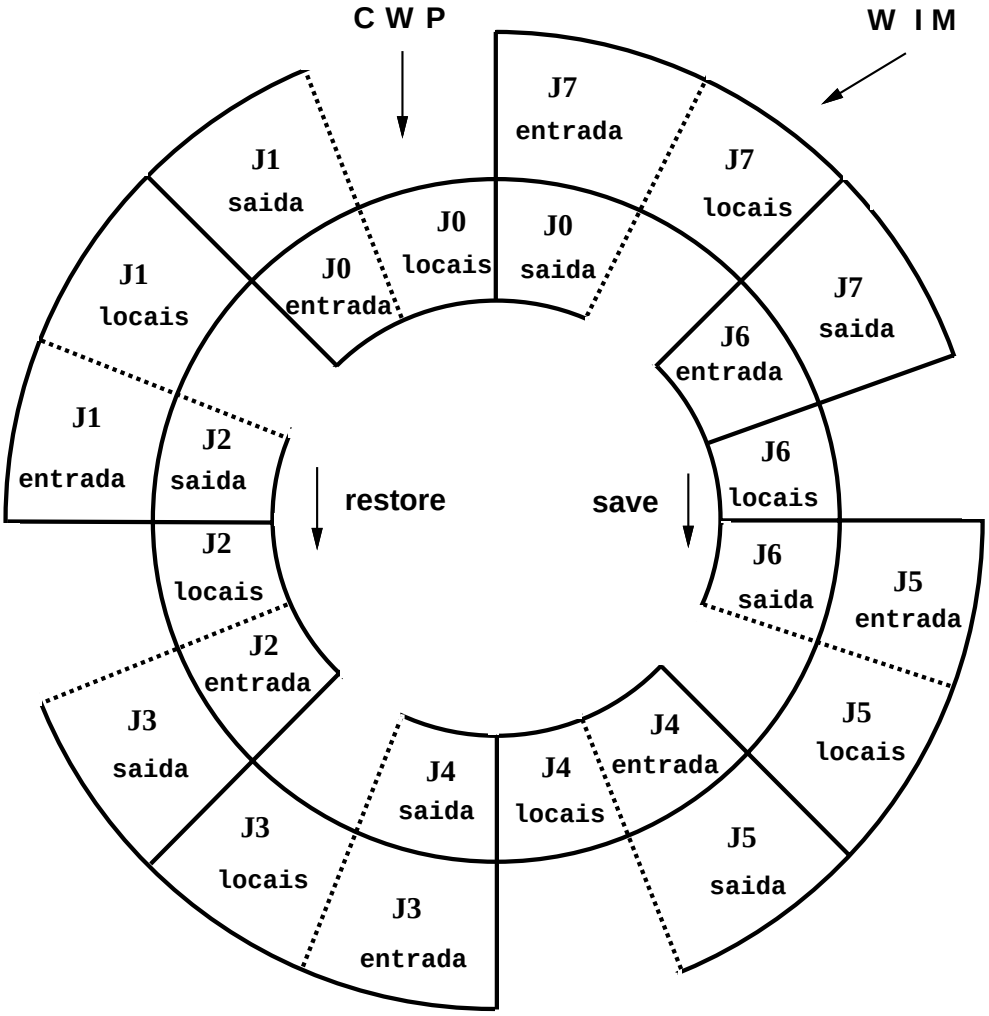
Um Modelo de Pilha de GGCO



Pilha usada na Arquitetura SPARC



Pilha Circular de Janela de Registradores



Vantagens no uso de Janelas de Registradores em um Sistema

- O sistema garante a reutilização dos registradores automaticamente quando possível, ao mesmo tempo associa registradores diferentes quando necessário.

- O sistema de tratamento de estouros é transparente ao compilador.

- Um maior número de registradores pode estar disponível sem a necessidade de mais *bits* nas instruções.

- Experiencias realizadas por Carlo Sequin,*Strategies for managing the register file in RISC*, comprovam que não há grandes seqüências de chamadas sem retornos, e vice-versa
  - Portanto o acesso às janelas tem comportamento similar a acessos ao *cache*.
  - Esta semelhança garante que estouros são relativamente raros.

Desvantagens no uso de Janelas de Registradores em um Sistema

- É difícil determinar o número ideal de registradores por janelas. Normalmente é feito um superdimensionamento.
- O custo de tratar estouro é alto. Muitos armazenamentos desnecessários podem ser executados, mesmo estando esta informação disponível a compiladores que fazem análise de fluxo de dados.
- É difícil quantificar o efeito de janelas na implementação de uma arquitetura, embora em arquiteturas, como SPARC, o tempo de ciclo seja determinado pelo acesso ao *cache*.
- O sistema operacional gasta mais tempo na troca de contexto dos processos mesmo quando poucos registradores estão realmente em uso por determinado programa (processo).
  - Isto é porque todos os registradores são armazenados e restaurados durante a troca do processo corrente.

Representação da Pilha na Memória

Esta pilha é usada durante as chamadas de procedimentos em linguagens do tipo C ou Pascal.

Um compilador gera código para procedimentos:

(1) Gera-se código para alocar espaço (*um stack frame*) para o procedimento decrementando o `sp`.

(2) Gera-se código para o próprio procedimento e ao final, o compilador gera código para liberar o espaço alocado incrementando o `%sp`.

Um compilador gera código para passar parâmetros usando o modelo tradicional de convenções de chamadas de procedimento.

Em um modelo clássico, cada registro de ativação consiste em três áreas: parâmetros de entrada, variáveis locais e parâmetros de saída.

Exemplo: como parâmetros e resultado são passados entre procedimento chamador e procedimento chamado

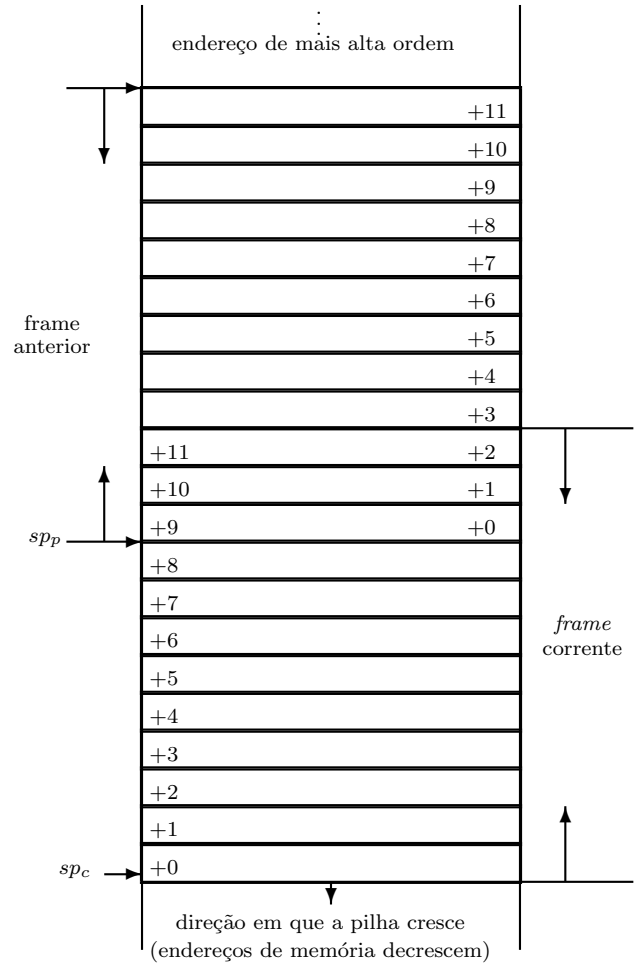


Figure 1: Pilha de Execução

o registro de ativação dos dois procedimentos envolvidos (*frame* anterior e o *frame* corrente) são alocados de tal forma que eles se sobrepõem. A área sobreposta identifica onde parâmetros do procedimento chamador são alocados e acessados pelo procedimento chamado.

Usando esta convenção, o compilador pode compilar cada um dos procedimentos completamente independentes.

Ele gera código para referenciar as variáveis na pilha utilizando os deslocamentos apropriados para o procedimento que está sendo compilado.

Por exemplo, para referenciar o primeiro parâmetro, um deslocamento de $+9$ deve ser usado pelo procedimento chamado (registro de ativação corrente) enquanto um deslocamento de $+0$ deve ser usado pelo procedimento chamador (registro de ativação anterior).

Pilha nos Registradores

Janela de registradores simula esta estrutura de pilha com duas diferenças:

(1) a estrutura é implementada nos registradores do `chip`;

(2) o tamanho das três áreas descritas (parâmetros de entrada, variáveis locais e parâmetros de saída) são fixa.

Janela de Registradores

A SPARC não possui instruções explícitas ou registradores sobre pilhas, o gerenciamento da pilha é resultado de convenções de software.

(1) na criação de um processo, o sistema operacional reserva uma área para a pilha.

(2) a pilha cresce a partir do endereço de mais alta ordem para o endereço de mais baixa ordem;

(3) os registradores `r16` até `r31` são preservados entre trocas de janelas;

(4) a área reservada para tratar estouros deve iniciar na posição indicada por `%sp` mesmo quando a altura da pilha variar devido a alocação dinâmica de memória;

(5) extensão da pilha é de 64 *bytes*, ou seja, `[sp] - [sp + 63]`. Os tratadores de estouro pressupõem um alinhamento em múltiplos de oito *bytes*.

Os parâmetros de uma função são encontrados a partir do endereço `fp + 64`, as variáveis locais e automáticas presentes na pilha são endereçadas em relação ao `%fp`.

Temporários e parâmetros de saída são endereçados especificando um deslocamento positivo a partir do `%sp` corrente (`%sp + 64`).

Problemas associados ao Uso de Janelas de Registradores

Em sistemas como UNIX existem vários processos executando simultaneamente (pseudo-paralelismo).

Para que cada processo tenha acesso aos recursos físicos do ambiente, por exemplo, CPU, o sistema operacional realiza a troca do contexto de um processo para o de um outro.

Caso o processo que estava executando anteriormente, tenha utilizado todas as janelas de registradores, é necessário salvar e invalidar todas janelas, inclusive aquelas que tiveram que ser alocadas na memória para serem utilizadas por outros processos.

Desta forma pode ocorrer um aumento no *overhead* de troca de contexto, o que caracteriza uma grande desvantagem do uso de janelas de registradores.

Uma solução de utilização das janelas de registradores em tais sistemas seria na base de uma por processo

Neste caso, em um sistema de multiprocesso poderia ter até oito processos sendo executados, simultaneamente, cada um deles com sua janela de registradores.

Se o processador parasse de executar um processo e passasse a executar outro, não seria necessário recarregar todo o seu contexto, o sistema operacional simplesmente trocaria a janela de registradores para a daquele processo e continuaria executando.

Compiladores existentes hoje não suportam este modo de operação.

Dificuldades encontradas para a Implementação do Sistema de Janelas no Sistema Marion

(1) A área para alocar memória dinâmicamente durante a execução de uma função, facilidade do sistema operacional UNIX que a pilha da SPARC utiliza mas que Marion não prevê esta hipótese.

(2) A necessidade de interagir com o sistema dado que as rotinas introduzidas para mainipulação de janelas vão ser utilizadas nele. Isto significa trabalhar no ambiente que Bradlee estabeleceu para Marion.

(3) A metodologia utilizada em Marion não é satisfatória. Variáveis globais, são, basicamente, usadas o tempo todo, não existe modularidade, é pouco comentado. O que significa que para substituir as rotinas de chamada de função implicaria em alterar várias partes do sistema e não seria possível mexer nisto entendendo somente da interface das rotinas que manipulam a pilha, precisaria entender o sistema como um todo nos mínimos detalhes.

Solução Adotada em GGCO

Em GGCO pensou-se em utilizar a pilha da SPARC mas, chegou-se a conclusão de que se o fizesse, estaria projetanto um sistema totalmente preso a sua arquitetura.

Supondo uma implementação nestas bases, os algoritmos projetados de acordo com estas regras manipulariam janelas de registradores com 8 entradas, 8 saídas e 8 locais.

Para executar uma especificação, por exemplo, da arquitetura MC88000 que não possui a facilidade de janela, estes algoritmos não serveriam para nada.

Portanto, um sistema com este modelo de pilha não é genérico, pelo contrário, é totalmente ligado a uma máquina específica, inclusive se isto traz um ganho, traz um ganho somente naquele processador porque em máquinas que não possuem esta característica, o ganho seria nulo.

Para implementar um sistema que mantenha estas duas estruturas paralelas, uma forma seria aproveitar a parte existente de Marion e projetar a pilha da SPARC de tal forma que quando fosse compilar para a SPARC, poderia ser utilizado recurso de pré-processamento para ter acesso ao arquivo onde estaria definido sua pilha.

O Sistema seria ineficiente porque não daria, simplesmente para pegar o que está pronto e substituir parte da implementação dado que Marion não possui uma boa definição de módulos.

Ideal Projetar um novo sistema que seja um superconjunto do sistema existente.

Produto final → interface (ou seja, as funções globais)

O sistema GGCO projetado não modela a pilha que engloba janela de registradores, contudo LDA possui as diretivas para tal.

Para incorporar janela de registradores: opção → implementar o algoritmo de renomeaçã

A troca de janelas salva e renomeia os registradores: os registradores declarados como (saída) $\%\mathbf{arg}$ *passam a ser novos (entradas)* $\%\mathbf{par}$.

Solução para o Esquema de Janelas

Em GGCO pensou-se em utilizar a pilha da SPARC mas, chegou-se a conclusão de que se o fizesse, estaria projetanto um sistema totalmente preso a sua arquitetura.

Supondo uma implementação nestas bases, os algoritmos projetados de acordo com estas regras manipulariam janelas de registradores com 8 entradas, 8 saídas e 8 locais.

Para executar uma especificação, por exemplo, da arquitetura MC88000 que não possui a facilidade de janela, estes algoritmos não serveriam para nada.

Portanto, um sistema com este modelo de pilha não é genérico, pelo contrário, é totalmente ligado a uma máquina específica, inclusive se isto traz um ganho, traz um ganho somente naquele processador porque em máquinas que não possuem esta característica, o ganho seria nulo.

Para implementar um sistema que mantenha estas duas estruturas paralelas, uma forma seria aproveitar a parte existente de Marion e projetar a pilha da SPARC de tal forma que quando fosse compilar para a SPARC, poderia ser utilizado recurso de pré-processamento para ter acesso ao arquivo onde estaria definido sua pilha.

O Sistema seria ineficiente porque não daria, simplesmente para pegar o que está pronto e substituir parte da implementação dado que Marion não possui uma boa definição de módulos.

Ideal Projetar um novo sistema que seja um superconjunto do sistema existente.

Produto final → interface (ou seja, as funções globais)

O sistema GGCO projetado não modela a pilha que engloba janela de registradores, contudo LDA possui as diretivas para tal.

Para incorporar janela de registradores: opção → implementar o algoritmo de renomeaçã

A troca de janelas salva e renomeia os registradores: os registradores declarados como (saída) $\%\mathbf{arg}$ *passam a ser novos (entradas)* $\%\mathbf{par}$.

VI - Trabalho Realizado

- Projeto de uma ferramenta de auxílio à implementação de compiladores de boa qualidade.

- Definicão de uma LDA capaz de especificar de forma satisfatória diferentes famílias de processadores.

- Formalização da definição da LDA.

- Especificação dos aspectos mais importantes da geração de código de maneira prática, utilizando-se da LDA.

- Identificação de técnicas para aumentar a eficiência do escalonamento de instruções e o nível de comunicação que deve existir entre o alocador e o escalonador de instruções.

VII - Conclusões e Trabalhos Futuros

A continuação deste trabalho compreende:

1. Implementação e avaliação do algoritmo de alocação de registradores proposto.

2. Um estudo da viabilidade de implementação de algumas sugestões apresentadas neste trabalho, como por exemplo:

   - implementação do esquema para modelar múltiplas unidades funcionais idêntica

   - implementação do algoritmo de Gross e Henessy para preencher os *delay slots* com instruções válidas durante o escalonamento;

3. Efetuar as modificações propostas neste trabalho para RASE e incorporá-lo em GGCO como uma segunda opção para escalonamento.