Implementação do Processador da Linguagem Δ

Relatório Final: POC2

Orientando: Eudes da Costa Cândido
Orientadora: Profa. Mariza A. S. Bigonha

Área: Compiladores
Projeto: Implementação do Processador da Linguagem Δ

Sumário

Resumo

1 Introdução

A Linguagem Δ é uma linguagem de programação utilizada como estudo de caso por David A. Watt [2], da Universidade de Glasgow em seus livros texto. Δ é uma linguagem Pascal-like, entretanto mais simples e mais regular.

O processador da linguagem Δ é um processador de linguagem integrado, consistindo de um editor, compilador, interpretador e desmontador.

TAM(Triangule Abstract Machine) é uma máquina abstrata, implementada por um interpretador. Está máquina foi projetada para facilitar a implementação da linguagem Δ , embora possa ser igualmente adequada para implementar Algol, Pascal e linguagens similares. As operações primitivas em TAM são mais similares às operações de uma linguagem de alto nível do que muitas operações primitivas de uma máquina real. Como consequência, a tradução de Δ para código TAM é muito rápida e direta [2].

O Compilador de Δ para TAM juntamente com o interpretador TAM constituem um compilador interpretativo.

O projeto que estamos propondo consiste na implementação do compilador, interpretador e máquina abstrata de tal forma que ao final deste projeto teremos implementado um processador de linguagem integrado, no qual um programador poderá abrir um programa fonte em Δ , editá-lo e compilá-lo. Podendo ainda executar e/ ou desmontar o programa objeto. Todas essas operações serão selecionadas por menus em um ambiente gráfico construído em Delphi, utilizando o sistema operacional Windows NT.

No caso deste projeto, será construído um compilador bottom-up [1] que traduzirá programas fonte escritos na linguagem Δ para código TAM.

Como subproduto, além de oferecer ao aluno a possibilidade de desenvolver estudos e trabalhos que contribuirão para a sua formação, este projeto proverá uma ferramenta de estudo e avaliação para a disciplina Compiladores.

2 Objetivo e Motivação

O projeto proposto tem como principal objetivo disponibilizar uma ferramenta de avaliação e teste para alunos e professores da disciplina Compiladores. O aluno, a medida que for contruindo os módulos do seu próprio compilador, tais como analisador léxico e o analisador sintático, poderá acoplá-lo ao compilador deste projeto, substituindo-o pelo módulo correspondente, devidamente extraído pelo professor, testá-lo, de modo a verificar se seu módulo está funcionando a contento.

Esta proposta é válida na medida que hoje é difícil para os alunos terem uma visão global do projeto que devem construir, a não ser no final do curso quando terão então, um compilador completo. Ademais, pode também ser utilizado como ferramenta de avaliação para o professor da disciplina, que poderá verificar mais rapidamente se os módulos contruídos pelo aluno funcionam ou não.

3 Descrição da Máquina Abstrata TAM

3.1 Memória e Registradores

A máquina possui duas memórias separadas:

- código, 32 bits, apenas leitura.
- Dados, 16 bits, leitura e escrita.

Os *layouts* das memórias estão ilustrados na Figura 1

Cada memória está dividida em segmentos, cujos limites são apontados por registradores dedicados. Dados e instruções são sempre endereçados relativamente a um destes.

Enquanto um programa é executado, o segmento da memória de Código é fixado, como se segue:

- o segmento de código contém as instruções do programa. Os registradores CB e CT apontam para a base e topo do segmento. O Registrador CP aponta para a próxima instrução a ser executada e inicialmente é igual a CB.
- O segmento primitivo contém "microcódigo" para aritmética elementar, lógica, entrada-saída, *heap* e operações de finalidade geral. Os registradores PB e PT apontam para a base e topo do segmento primitivo.

Enquanto um programa é executado, o segmento de memória de Dados pode variar:

- a pilha cresce a partir do endereço mais baixo. Os registradores SB e ST apontam para a base e topo da pilha, e ST é inicialmente igual a SB.
- O *heap* cresce a partir do endereço mais alto. Os registradores HB e HT apontam para a base e topo do *heap*, e HT é inicialmente igual a HB.

Pilha e *heap* podem expandir-se e contrair-se. A memória de dados será esgotada quando ST e HT se cruzarem.

A pilha, em particular, consiste de um ou mais segmentos:

- segmento Global: é a base da pilha e contém dados globais usados pelo programa.
- *Frames*: Cada *frame* contém dados locais para serem ativados em alguma rotina. Ao chamar uma rotina, um novo *frame* é empilhado na pilha; no retorno da rotina o quadro superior é desempilhado. O *frame* mais superior pode expandir ou contrair, mas os outros em frames têm temporariamente tamanho fixo. O registrador LB aponta para a base do *frame* superior.

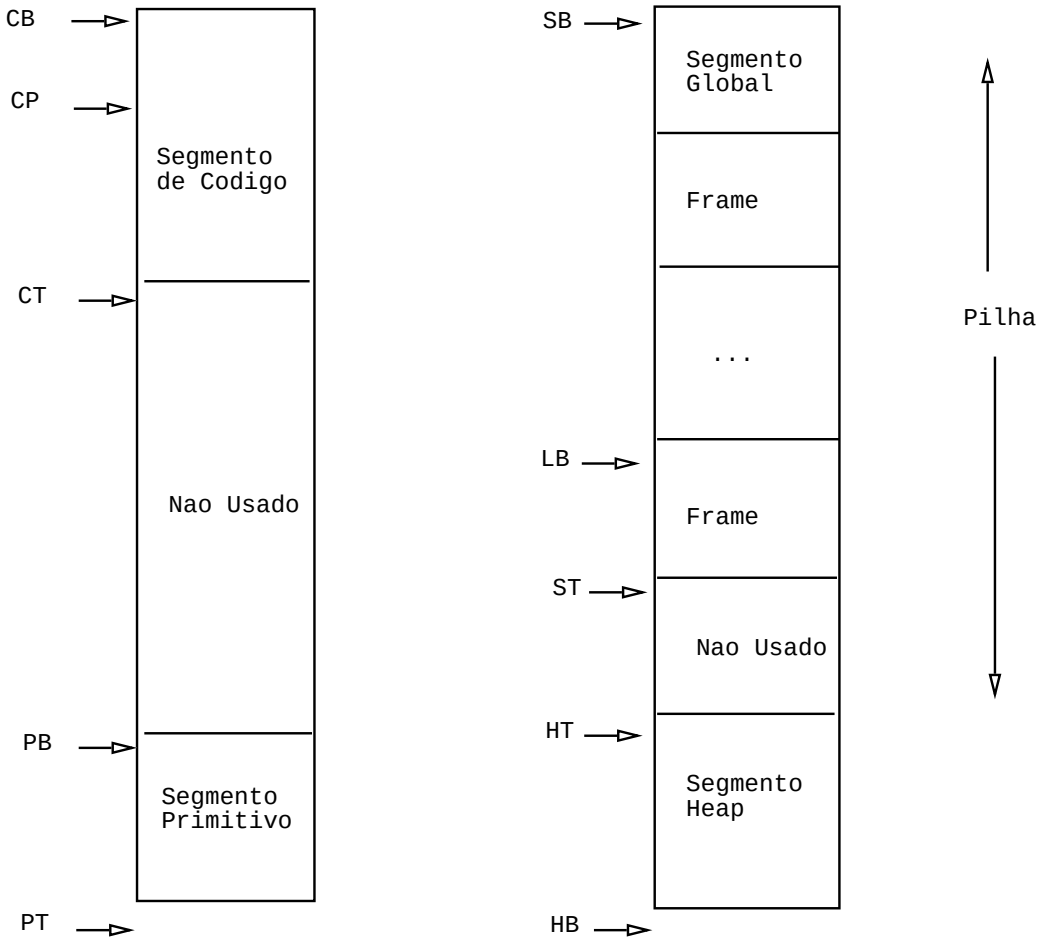

Figura 1: *Layout* das Memórias de Código e Dados

A Figura 2 mostra o esboç o de um programa fonte em alguma linguagem estruturada por bloco.

- O programa principal chamou o procedimento P. O registrador LB aponta para o *frame* superior, associado a P.

- O procedimento P chamou o procedimento S. O registrador LB aponta para o *frame* superior associado com S; o registrador L1 aponta para o *frame* superior associado com P.

- O procedimento S chamou o procedimento Q. O registrador LB aponta o *frame* superior, associado com Q. O registrador L1 ainda aponta para o *frame* associado com P.

- O procedimento Q chamou o procedimento R. O registrador LB aponta para o *frame* superior associado com R; o registrador L1 agora aponta para o *frame* associado com Q; o registrador L2 aponta para o *frame* associado com P.

```
...                          ....... programa principal
proc P () ~
   ...                     ......... corpo de P
  proc Q () ~
    ...                  ........... corpo de Q
   proc R () ~
     ...              ............... corpo de R
  ...
  proc S () ~
     ...                 ........... corpo de S
  ...
...
```

Figura 2: Esboço de um programa fonte.

Dados globais, locais e não locais são acessados do seguinte modo:

LOAD (n) d[SB] - para qualquer procedimento carregar dados globais
LOAD (n) d[LB] - para qualquer procedimento carregar seus próprios dados locais.
LOAD (n) d[L1] - para Q ou S carregar dados locais a P.
LOAD (n) d[L2] - para R carregar dados locais a P.

Em cada caso um objeto de n-palavras é carregado a partir do endereço d relativo a base do segmento apropriado. Armazenar é análogo a carregar.

Em geral, o registrador LB aponta para o *frame* superior que está sempre associado com a rotina R cujo código está sendo atualmente executado; o registrador L1 aponta para o *frame* associado com a rotina R' que textualmente encobre R no programa fonte; o registrador L2 aponta para o *frame* associado com a rotina R" que textualmente encobre R'; e assim por diante.

Todos os dados acessíveis na pilha podem ser endereçados relativamente aos registradores SB, LB, L1, L2, etc., como segue:

d[SB] - para qualquer rotina que acesse dados globais
d[LB] - para qualquer rotina que acesse seus próprios dados locais
d[L1] - para qualquer rotina R que acesse dados locais a R'
d[L2] - para qualquer rotina R que acesse dados locais a R"

Em cada caso um objeto é acessado no endereço d relativo a base do segmento apropriado. Na prática os registradores L1, L2, etc., são usados com uma frequência menor que LB e SB.

O *layout* de um *frame* está ilustrado na Figura 3. Considere um *frame* associado a R:

- O *static link* aponta para um outro *frame* associado com a rotina que textualmente enconbre R no programa fonte.
- O *dynamic link* aponta para um *frame* subordinado imediatamente a este na pilha.
- *return address* é o endereço da instrução imediatamente seguinte a instrução call que ativou R.

A máquina abstrata TAM possui 16 registradores, sumarizados a seguir. É importante destacar que todos os registradores possuem uma função particular, portanto nenhuma instrução usa registradores para operar com dados. Alguns dos registradores são constantes.

Os registradores L1, L2, etc., são apenas pseudo-registradores , necessários para endereçar dados não locais, que são dinamicamente avaliados a partir de LB usando as invariantes:

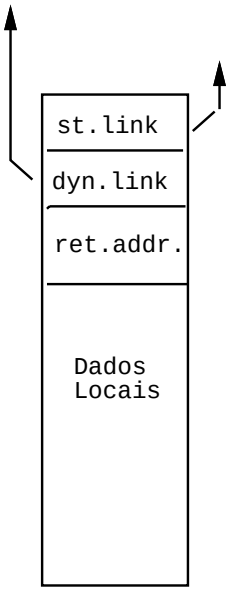

Figura 3: *Layout* de um *frame* em TAM.

- L1 = conteúdo(LB).
- L2 = conteúdo(conteúdo(LB)), etc., onde conteúdo(a) significa a palavra que está no endereço a.

Isto funciona porque LB aponta para a primeira palavra do *frame*, que contém seu *link* estático, que por sua vez aponta para a primeira palavra de um *frame* subordinado, assim por diante.

N Reg.	Mne.	Nome	Comportamento
0	CB	Code Base	constante
1	CT	Code Top	constante
2	PB	Primitive Base	constante
3	PT	Primitive Top	constante
4	SB	Stack Base	constante
5	ST	Stack Top	modificado pela maioria das instruç oes
6	B	Heap Base	constante
7	HT	Heap Top	variável para rotinas de Heap
8	LB	Local Base	modificado por instruções de chamada e retorno
9	L1	Local Base 1	L1 = conteúdo(LB)
10	L2	Local Base 2	L2 = conteúdo(L1)
11	L3	Local Base 3	L3 = conteúdo(L2)
12	L4	Local Base 4	L4 = conteúdo(L3)
13	L5	Local Base 5	L5 = conteúdo(L4)
14	L6	Local Base 6	L6 = conteúdo(L5)
15	CP	Code Pointer	modificado por todas as instruções

3.2 Instruções

Todas as instruções TAM têm um formato comum. O campo op contém o código da operação. O campo r contém o número de um registrador e o campo d geralmente contém um deslocamento de endereço (possivelmente negativo); juntos eles definem o endereço do operando (d+registrador r). O campo n normalmente contém o tamanho do operando.

código da operação	op	4 bits
número do registrador	r	4 bits
tamanho do operando	n	8 bits
valor do deslocamento	d	16 bits (com sinal)

3.3 Rotinas

Toda rotina TAM deve respeitar estritamente o procolo ilustrado na Figura 4. Assuma que uma rotina R aceita d palavras de argumentos e retorna um resultado de n-palavras. Imediatamente antes de R ser chamado, seus argumentos devem ser colocados no topo da pilha. Se R não tem argumentos, então d será zero. Ao retorno de R, seus argumentos são substituídos no topo pelo resultado. Se R não retorna nada, então n será zero.

Existem dois tipos de rotinas em TAM:

- rotinas de código.
- Rotinas primitivas.

Uma rotina de código consiste de uma sequência de instruções armazenadas no segmento de código. O controle é transferido para a primeira instrução da sequência por

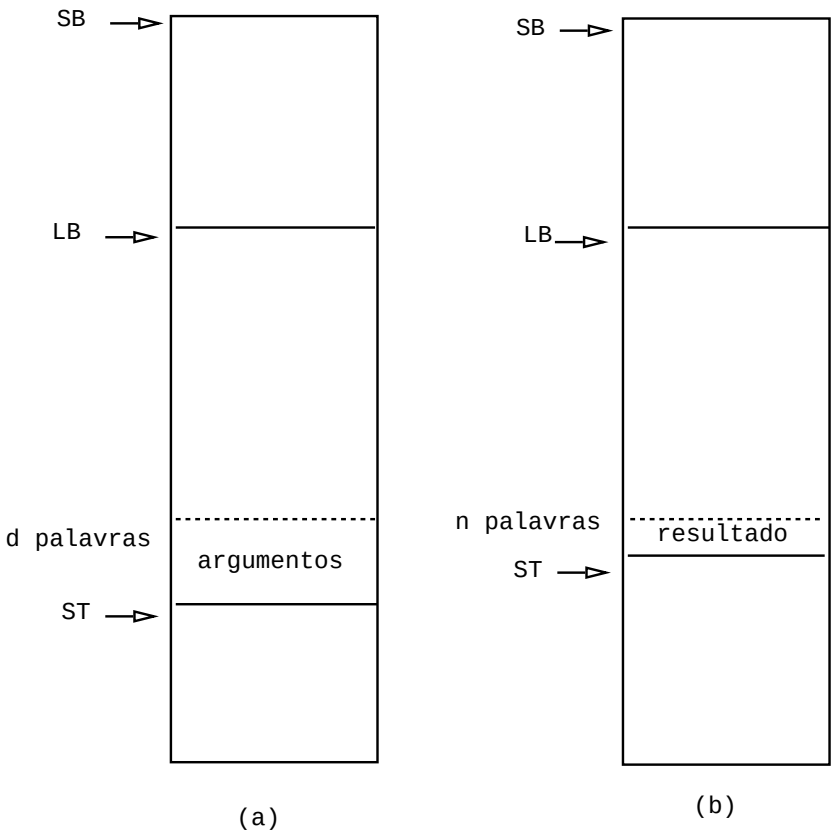

Figura 4: (a) *Layout* da pilha e (b) Após a chamada de uma rotina TAM

uma instrução CALL ou CALLI e em seguida transferido de volta através da instrução RETURN.

Uma rotina primitiva é aquela que realiza operações de aritmética, lógica, entrada–saída, *heap* ou qualquer outra operação de finalidade geral. As rotinas primitivas estão sumarizadas na tabela a seguir. Cada rotina primitiva tem um endereço fixo no segmento primitivo. TAM "traps" toda chamada a um endereço naquele segmento e realiza a operação correspondente diretamente.

OpCode	Mnemônico	Efeito
0	LOAD(n) d[r]	Busca um objeto de n palavras a partir do endereço (d + r) e empilha na pilha.
1	LOADA d[r]	Empilha um endereço de dado que está no endereço [d + r].
2	LOADI (n)	Desempilha um endereço de dado da pilha, busca um objeto de n palavras a partir daquele endereço e empilha-o na pilha.
3	LOADL d	Empilha o valor literal d de uma palavra na pilha.
4	STORE (n) d[r]	Desempilha um objeto de n-palavras da pilha e armazena-o no endereço de dado (d+r).
5	STOREI (n)	Desempilha um endereço da pilha, então desempilha um objeto de n-palavras da pilha e armazena-o naquele endereço.
6	CALL(n) d[r]	Chama a rotina que está no endereço (d+r) usando o endereço no registrador n como static *link*.
7	CALLI	Desempilha um closure (static *link* e endereço de código) da pilha, então chama a rotina naquele endereço.
8	RETURN (n) d	Retorna da rotina corrente: desempilha um resultado de n palavras da pilha, desempilha o *frame* mais superior, desempilha d palavras de argumentos e empilha o resultado de volta na pilha.
9	-	(não usado)
10	PUSH d	Empilha d palavras (não inicializadas) na pilha.
11	POP (n) d	Desempilha um resultado de n palavras da pilha, desempilha mais d palavras e então empilha o resultado de volta na pilha.
12	JUMP d[r]	Desvia para o endereço de código (d + r)
13	JUMPI	Desempilha um endereço de código da pilha, então desvia para esse endereço.
14		Desempilha uma palavra da pilha, então desvia para endereço de código (d + r) se e somente se o valor é igual a n.
15	HALT	Para a execução do programa

Endereço	Mne	Arg.	Resultado	Efeito
PB+1	id	w	w	w4 = w
PB+2	not	t	t4	t4 = not t
PB+3	and	t1,t2	t	t = t1 and t2
PB+4	or	t1, t2	t	t = t1 or t2
PB+5	succ	i	i4	i4 = i + 1
PB+6	pred	i	i4	i4 = i -1
PB+7	neg	i	i4	i4 = -i
PB+8	add	i1, i2	i4	i4 = i1 + i2
PB+9	sub	i1, i2	i4	i4 = i1 - i2
PB+10	mult	i1, i2	i4	i4 = i1 * i2
PB+11	div	i1, i2	i4	i4 = i1 / i2 (truncado)
PB+12	mod	i1, i2	i4	i4 = i1 módulo i2
PB+13	lt	i1, i2	t4	t4 = verdadeiro se i1 ¡ i2
PB+14	le	i1, i2	t4	t4 = verdadeiro se i1 ¡= i2
PB+15	ge	i1, i2	t4	t4 = verdadeiro se i1 ¿= i2
PB+16	gt	i1, i2	t4	t4 = verdadeiro se i1 ¿ i2
PB+17	eq	v1, v2,i	t	t = verdadeiro se v1 = v2
PB+18	ne	v1, v2,i	t	t = verdadeiro se v1 ¡¿ v2 (v1 e v2 são valores de i palavras)
PB+19	eol	-	t	t4 = verdadeiro se o próximo caractere a ser lido é um final de linha.
PB+29	eof	-	t	t4 = verdadeiro se não há mais caracteres a serem lidos (fim de arquivo).
PB+21	get	a	-	Leia um caractere e carregue no endereço a
PB+22	put	c	-	Escreva o caractere c.
PB+23	geteol	-	-	Leia caracteres até incluir o próximo fim de linha.
PB+24	puteol	-	-	Escreva um fim de linha.
PB+25	getint	a	-	Leia um literal inteiro (opcionalmente precedido por brancos a/ou sinalado), e carregue seu valor no endereço a.
PB+26	putint ‘	i	-	Escreva o literal inteiro i.
PB+27	new	i	a	a = endereço de objeto recentemente alocado no *heap*.
PB+28	dispose	i, a	-	Desaloca o objeto de i palavras do endereço a no *heap*.

Significado das letras

Letra	Significado
a	endereço de dado
c	caractere
i	inteiro
t	valor verdade (0 para falso, 1 para verdadeiro)
v	valor de qualquer tipo
w	valor de tamanho igual a uma palavra

4 Compilador para a Linguagem Triângulo

4.1 Introdução

4.2 Especificação Informal da Linguagem de Programação triângulo

4.2.1 Comandos

4.2.2 Expressões

4.2.3 Nomes

4.2.4 Declarações

4.2.5 Parâmetros

4.2.6 Denotadores de Tipo

4.2.7 Especificação Léxica

4.3 Implementação do Compilador

4.3.1 Análise Léxica

4.3.2 Análise Sintática

4.3.3 Análise Semântica

4.3.4 Geração de Código

4.4 Interface e Uso dos Módulos

5 Conclusão

Referências

[1] A. V. Aho, J. D. Ullman, and R. S. Sethi. *Compilers, Principles, Techniques and Tools.*

Addison Wesley Publishing Company Co., 1986.

[2] David A. Watt. *Programming Language Processors*. C.A.Hoare Series Editor, Prenticel Hall International Series in Computer Science, 1993.

A Descrição da Máquina de Pilha

A máquina de pilha, diferentemente dos processadores que utilizam registradores, faz uso de uma pilha. A pilha é uma memória LIFO (último a entrar, primeiro a sair) com duas operações abstratas: push, pop. Push empilhá um item no topo da pilha. Pop busca um item no topo da pilha.

A.1 Cálculos Usando Pilha

Em função da pilha ser LIFO, toda operação acessa o item de dado que está no topo. A pilha não necessita de endereçamento, uma vez que o mesmo está implícito nos operadores que ela utiliza. Assim, qualquer expressão pode ser transformada para a forma posfixada, avaliando a expressão sem a necessidade de localizar explicitamente qualquer variável. Por exemplo,

B + C - D
B C + D - (forma posfixada)
push val B, push val C, add, push val D, sub.

A = B
A B =
push ads A, push val B, store.

Neste exemplo, `add` pega dois itens da pilha, efetua a operação de adição e empilha o resultado no topo da pilha, a instrução sub opera de forma similar. `Store` pega um valor e um endereço da pilha e armazena o valor no endereço.

Vamos comparar as expressões acima usando registradores.

B + C - D
load r0, B
load r1, C
add r0, r1
load r2, D
sub r0, r2

A = B
load r0, ads A
load r1, val B
store r1, (r0)

Pode-se observar que a principal diferença reside na alocação de registradores, por

exemplo, r0 é usado para armazenar o resultado temporário enquanto que em uma máquina à de pilha, a memória temporária fica implícita.

A.2 Chamada de Subrotinas

A estrutura da pilha também desempenha um importante papel na chamada de subrotinas ou chamada de funções. Quando um programa transfere o controle para outra seção no código, o estado corrente da máquina, composto pelo PC, variáveis locais e outros valores, são salvos. O local onde o estado da computação é salvo é chamado "registro de ativação". Quando a execução de uma subrotina é completada o estado anterior da máquina é restaurado.Esta ação é chamada de "retorno" da subrotina. Como grande parte das linguagens atuais são "estruturadas" ou "orientada a blocos", a criação e remoção de registros de ativação se comporta como LIFO. A pilha é usada para armazenar registros de ativação.

A.3 Pilha da Máquina de Pilha

Este tipo de máquina usa pilha para armazenar valores temporários durante o cálculo e também para armazenar os registros de ativação durante as transferências de controle entre subrotinas. Nas máquinas de registradores deve-se alocar explicitamente registradores para armazenar valores temporários e alguma manipulação explicita do tipo LIFO deve ser feita, via algum tipo de apontador, para lidar com os registros de ativação.

Há inúmeras linguagens de programação comteporâneas que usam abstração de pilha, por exemplo, Forth e Postscript. Também muitas outras usam o modelo de "máquina virtual" para implementar suas representações executáveis. Pascal tem o P-system, Smaltalk usa pilha para efetuar cálculos, JAVA tem "byte-code" que usa modelo de pilha.

A.4 Exemplo da Arquitetura de Conjunto de Instruções de uma Máquima de Pilha

Para facilitar o entendimento vamos abstrair dos detalhes envolvidos na criação de um registro de ativação, vamos ignorar variáveis locais e outras informações e ilustrar uma ISA (*Instruction Set Arquiteture*) que é baseada em uma máquina de pilha. Nós precisamos de `load`, `store` ,operadores aritméticos, `call` , `return` e para o fluxo de controle usaremos as instruções convencionais `jump` e `branch`. No treçho de código a seguir: TOP é o item do topo da pilha, NEXT é o item abaixo do TOP, portanto nós podemos chamar dois operandos da pilha por TOP e NEXT, M[ads] o valor de memória em ads. POP a é TOP − a, pop 2 desempilha dois itens da pilha.

lia #a	empilha o valor imediato a.
load	desempilha a, empilha M[a].
store	NEXT - M[TOP] , pop2.
add	NEXT + TOP-a, pop2, push 1.
cmp	if NEXT TOP a =1 else a = 0, pop2, push a.
call	pop a, cria um novo registro de ativação, vá para a.
return	deleta um registro de ativação, vá para pc'.

Observe que lit #a tem #a como argumento, outras instruções tem argumentos implícitos na pilha. O estado de computação é formado do apontador de pilha e do contador de programa.

Se houver duas pilhas disponíveis , uma para cálculo e outra para registro de ativação, chamada de pilha de controle, precisamos somente do PC, endereço de retorno no registro de ativação e não há a necessidade de se fazer qualquer coisa com a pilha de cálculo nas chamadas de subrotina. Ao chamar uma subrotina necessitaríamos apenas empilhar o PC corrente(endereço de retorno) na pilha de controle. No retorno basta desempilhar a pilha de controle e restaurar o PC anterior.

A.5 Máquina de Pilha Pura

Se não temos variáveis locais, como podemos acessar os operandos na pilha de cálculo ? Com exceção dos dois itens no topo da pilha, acessar outros itens é bem mais complicado. É preciso ser capaz de reordenar e copiar inúmeros itens da pilha para usá-los. Algumas instruções que fazem isso:

dup ;; duplica TOP
swap ;; troca TOP e NEXT
rot ;; 1,2,3 - 3,2,1 coloca o terçeiro elemento no topo
over ;; copia NEXT para TOP, 1,2 - 2,1,2

Estas são apenas algumas das instruções. Um exemplo de seu uso:
f(X,Y) = X*X + Y*Y
X e Y são os dois itens do topo da pilha de cálculo quando f()é chamada.
dup mul ;; X*X
swap ;; Y, X*X
dup mul add ;; Y*Y + X*X

Pensar sobre rearranjo de itens na pilha tornar difícil o uso de instruções puras. O uso de variáveis evita a reordenação de itens na pilha, porque uma variável pode ser acessada usando seu nome. Entretanto, as técnicas de compilação atuais pode lidar com a ordenação de itens da pilha, liberando o programador deste nível mais baixo de detalhe. Exemplo de programa: "BUBBLE SORT"
Este exemplo mostra como um programa em linguagem de alto nível pode ser convertido no ISA de uma máquina de pilha.
Dado a[n] um vetor de inteiros, o programa bubble sort ordena os itens em um a[] em ordem ascendente.

```
    while( i < n ) {
while( j < n ) {
if( a[j] < a[j+1] ) {
t = a[j];
a[j] = a[j+1];
a[j+1] = t;
}
j = j+1;
}
i = i+1;
}
```

Segmento de Dados(palavra)

```
1: t
2: i
3: j
4: n
5: a[0]
6: ...
```

Segmento de Código(byte)

```
68: rval 3 rval 4 lt
75: jz 187
78: rval 2 rval 4 lt
88: lval 5 rval 2 index load
```

```
lval 1 rval 2 lit 1 add index load gt
109: jz 159
112: lval 1 lval 5 rval 2 index load store
124: lval 5 rval 2 index
lval 5 rval 2 lit 1 add index load store
144: lval 5 rval 2 lit 1 add index rval 1 store
159: lval 2 rval 2 lit 1 add store
170: jmp 78
173: lval 3 rval 3 lit 1 add store
184: jmp 68
187:
```

onde,
lval ads é lit #ads
rval ads é lit #ads load
index é add
jz é jump se TOP = 0
jmp é jump incondicional

A.6 Discussão

Argumenta-se que as máquinas de pilha são o tipo mais simples de arquitetura. Sua estrutura LIFO é adequada a linguagens estruturadas ou orientada a bloco. O tamanho de código para uma máquina de pilha pode ficar muito compacto, uma vez que a maioria das instruções não tem campo operando. Arquitetura de pilha tem sido um método muito popular para implementar linguagem de máquina de alto nível. Contudo, argumenta-se que a maioria das máquinas modernas baseadas em registradores são mais rápidas, mas existe um esforço "renovado" de melhorar a arquitetura de pilha. Notavelmente, o processador Picojava da SUN que pretende executar a máquina virtual JAVA. A arquitetura de pilha pode provar no futuro que é apropriada para uma máquina.

B Documentação sobre o Interpretador

B.1 Descrição dos Módulos

B.2 Testes do Interpretador

B.3 Listagem do Interpretador