Projeto e Implementação de Uma Linguagem Intermediária do Compilador ACTUS II para Transputer

Cláudia Linhares Sales

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUACÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA OBTENCÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTACÃO.

Aprovada por:

Prof. Cláudio Luis de Amorim, Ph.D.
(presidente)

Profa. Leila Maria Ripoll Eizirik, D.Sc.

Prof. Edil Severiano Tavares Fernandes, Ph.D.

Prof. Jairo Panetta, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
OUTUBRO DE 1990

SALES, CLÁUDIA LINHARES

Projeto e Implementação de uma Linguagem Intermediária do Compilador ACTUS II para *Transputer* [Rio de Janeiro] 1990
xii, 131 p., 29.7 cm, (COPPE/UFRJ, M. Sc., Engenharia de Sistemas e Computação, 1990)
TESE – Universidade Federal do Rio de Janeiro, COPPE
1 – Linguagens 2 – Processamento Paralelo
I. COPPE/UFRJ II. Título(série).

A memória de meus avós Jardilina, Gabriel e Narciso,
A minha avó Maria,
A meus pais Rita e Manuel,
A meus irmãos Narciso, Verônica, Marta, Sônia e Inês,
A meus tios Nenzinha e Geraldo
e A Ricardo.

Agradecimentos

Esse espaço é pequeno para relacionar as pessoas que têm me ajudado ao longo destes anos. De fato, recebi, embora em formas e freqüências diversas, ajudas de quase todas as pessoas que conheci. Essas ajudas variaram desde palavras encorajadoras, passando por caronas recebidas, lições de vida e passagem de conhecimentos, até favores impágaveis, tantas vezes generosamente prestados, graças às amizades que fiz, e que, sem dúvida, são o meu maior prêmio. Meus sinceros agradecimentos a todas essas pessoas. Abaixo, relaciono as pessoas que me ajudaram de maneira especial.

Aos professores Cláudio e Leila pela excelente orientação, paciência e apoio.

Ao Povo Brasileiro, que através da CAPES e CNPq, possibilitou a execução desse trabalho.

Aos professores Valmir, Pinho, Edil, Nélson e Jayme, pelos conhecimentos adquiridos.

Aos professores Tarcísio, Clécio e José Carlos, da UFC, pelo incentivo recebido para iniciar este curso.

Aos professores Dina e Antônio, pelas palavras de apoio e conforto, nas situações difíceis.

A Delfim, pelo esforço e boa-vontade empregados neste trabalho. Certamente, a sua contribuição foi indispensável para a obtenção dos resultados aqui apresentados.

A Denise e Cláudia pela paciência, já histórica, nos trabalhos de secretaria.

A Edu e Adilson, pelos incontáveis socorros prestados nas situações aflitivas (fitas ruins, falta de papel, defeitos em impressoras, etc.). Além disso, pela amizade de ambos.

A Suely, pelo apoio, ajuda e companhia no NCP.

A D. Prazeres, pela ajuda, carinho e amizade, que tantas vezes ajudou-me a superar as saudades de minha familia.

A Tia Luzia, pelo carinho e pela ajuda recebida, de forma tão generosa, ao chegar em terra estranha.

Ao Wamberto (Bamv's), companheiro de 'àpê", de risos, de "baixo astral" e de esperanças, pelo apoio e amizade.

A Cristina (Titina), Mónica e Clícia (Glícia) pela amizade, apoio, companheirismo e pelos "zilhões" de vezes que me ajudaram. Com muito carinho, obrigada.

A Adelina, Ângela, Anna, Célia, Cláudia, Emilia, Farid, Gilberto, Hércules, Hermes, Inês, Ildemar, Lorena, Lúcia, Luis, Mariela, Mau, Mauricio, Max, Oscar, Philippe, Riverson, Rose e Rui. Não foram poucas as vezes que obtive ajuda dessas pessoas e, sem dúvida, o seu apoio foi fundamental para a conclusão desse trabalho. Obrigada.

Às familias Stelling de Castro, Cordeiro Corrêa e Silva Boeres, pelo apoio e inúmeras ajudas.

Aos ex-companheiros de DelRio, pelo apoio recebido para iniciar este curso. Em especial, agradeço o carinho e amizade de D. Maria e Marta.

A Lurdi, pelo carinho e apoio. Com muito carinho, obrigada.

A toda a minha família, pela "força" e carinho.

Em especial, aos meus pais e irmãos, pelo amor, carinho e apoio, tão grandes, que o tempo todo os senti ao meu lado. Com muito amor, obrigada.

A Ricardo, por **tudo**. Com muito amor, obrigada.

Resumo da Tese apresentada à COPPE como parte dos requisitos necessários para a obtencão do grau de Mestre em Ciências (M. Sc.)

Projeto e Implementação de uma Linguagem Intermediária do Compilador ACTUS II para *Transputer*

Cláudia Linhares Sales

Outubro de 1990

Orientador: Cláudio Luis de Amorim
Programa: Engenharia de Sistemas e Computacão

Como parte da pesquisa e desenvolvimento em processamento vetorial e paralelo, um compilador da Linguagem ACTUS II está sendo implementado. A linguagem ACTUS II tem construções que permitem expressar diretamente o paralelismo de problemas que podem ser mapeados em grades ou malhas.

O presente trabalho consistiu do projeto de uma linguagem intermediária específica para a linguagem ACTUS II e da implementação do gerador de código para o *transputer*. No trabalho, descrevemos como foram contornadas as diferenças entre as linguagens paralelas ACTUS II e OCCAM 2 (OCCAM 2 é a linguagem objeto). Algumas dessas diferenças foram minimizadas pela linguagem intemediária, outras tiveram que ser resolvidas pelo gerador de código. No trabalho, apresentamos a linguagem intermediária, o processo de geração de código e os testes realizados para validação de ambos.

Abstract of Thesis presented to COPPE as partial fulfillment of the requirements for the degree of Master of Science (M. Sc.)

Project and Implementation of a Compiler ACTUS II Intermediate Language for Transputer

Cláudia Linhares Sales
October, 1990

Thesis Supervisor: Cláudio Luis de Amorim
Department: Programa de Engenharia de Sistemas e Computacão

A compiler ACTUS II is being implemented as part of the research and development in vectorial and parallel processing. The ACTUS II language has constructions which allow to express directly the parallelism of problems which can be maped in grids or meshes.

This work envolved the project of a ACTUS II intermediate language and the code generator implementation for transputer. We describe the differences between ACTUS II and OCCAM 2 parallel languages (OCCAM 2 is the object language), and how they were resolved. Many of these differences were minimized by the intermediate language, but other were resolved by the code generator. We also present the intermediate language, the code generation process and the tests performed.

Índice

Lista de Tabelas

Capítulo I

Introdução

Os ambientes de programação utilizados para se fazer uso efetivo da alta velocidade de processamento oferecida pelos supercomputadores são em geral baseados em um dos seguintes métodos. O primeiro corresponde ao uso de linguagens cujas construções refletem diretamente a arquitetura da máquina (p.e. DAP FORTRAN), ou de linguagens que incluem subrotinas escritas em código de máquina para a manipulação do paralelismo. Optando-se por este método, tem-se a oportunidade de escrever programas que usam mais eficientemente a arquitetura disponível, e, além disso, a implementação das linguagens que usam bibliotecas de rotinas paralelas é mais rápida e fácil. Entretando, a portabilidade dos programas é bastante prejudicada, pois, em geral, as rotinas paralelas são feitas especialmente para uma determinada arquitetura. O segundo método transfere ao compilador a responsabilidade de extrair o máximo de paralelismo possível dos programas seqüenciais a ele submetidos (p.e. através de vetorização de *loops*) [15] [14]. Neste caso, para se obter um bom resultado, tem-se que reestruturar o programa ou reprojetá-lo para adaptar-se ao compilador. Este método possibilita o aproveitamento dos vultuosos investimentos feitos em aplicações escritas em linguagens seqüenciais, usualmente FORTRAN.

O terceiro método consiste em explorar o paralelismo do problema diretamente. Nessa abordagem, o programador deve estudar o problema e descobrir, se existir, o seu paralelismo natural e expressá-lo através de uma linguagem apropriada e isenta de referências à estrutura subjacente do *hardware* ou aos mecanismos de detecção de paralelismo do compilador. A Linguagem ACTUS II é uma proposta nessa direção. Ela permite expressar o paralelismo do problema através de construções de alto nível. Assim, ACTUS II se beneficia da confiabilidade que uma linguagem de alto nível oferece e garante um alto nível de portabilidade dos programas.

A linguagem ACTUS II [3] [5] é uma Linguagem de Alto Nível, *pascal-like*, cujas características especiais são as estruturas de dados e de controle oferecidas para a manipulação de paralelismo em ambientes paralelos síncronos, encontrados nas arquiteturas de processadores matriciais e vetoriais.

Este trabalho é parte do estudo de técnicas de programação de su-

percomputadores, e se insere no projeto de construção de um compilador baseado em ACTUS II, para um computador paralelo de memória distribuída, projetado e implementado no Laboratório de Computação Paralela da COPPE/UFRJ.

Embora a linguagem ACTUS II não tenha sido projetada para esse modelo computacional, é possível, no entanto, implementá-la no ambiente de um dos nós de processamento do computador paralelo. Em cada nó de processamento existem dois processadores: um *transputer* T800 e um *intel* i860 que se comunicam através de memória compartilhada. Do ponto de vista do compilador, o i860 funciona como se fosse um "processador vetorial" (o i860 não possui instruções vetoriais, como nos supercomputadores, mas utiliza *pipelines* aritméticos e é capaz de atingir 60 MFLOPS com precisão dupla – 64 bits), enquanto o T800 desempenha o papel de processador central com funções escalares e de controle.

Associada ao *transputer*, encontra-se a linguagem OCCAM 2 [10] que foi projetada para programação de ambientes paralelos assíncronos, baseados em rede de *transputers*. O seu caráter especial está na simplicidade inerente à linguagem para a manipulação de concorrência e comunicação. Ela pode ser considerada como uma linguagem de montagem de uma rede de *transputers* por fornecer um bom nível de eficiência e desempenho do código produzido [11].

Este trabalho compreende o projeto de uma linguagem intermediária (LI) a ser usada no processo de compilação da linguagem ACTUS II [1] em OCCAM 2 [8] e no projeto e implementação do gerador de código do compilador.

No Capítulo II, serão apresentadas as linguagens OCCAM 2 e ACTUS II, no que diz respeito à expressão de paralelismo.

No Capítulo III, é apresentada a LI e no Capítulo IV, é discutido o mapeamento de ACTUS II na LI.

No Capítulo V, o processo de geração de código é tratado, sendo dada ênfase aos problemas encontrados em gerar código OCCAM 2 a partir da LI e, além disso, são descritos os testes executados (100 subrotinas escritas em ACTUS II e LI), para validação do Gerador de Código.

Finalmente, no Capítulo VI são apresentados os resultados mais significativos do trabalho e perspectivas de pesquisa.

Durante todo este trabalho, as linguagens ACTUS II e OCCAM 2 serão referenciadas apenas como ACTUS e OCCAM, respectivamente.

Capítulo II

Expressão do Paralelismo nas Linguagens ACTUS e OCCAM

II.1 Introdução

As linguagem ACTUS e OCCAM são apropriadas para exprimir algoritmos a serem executados em processadores vetoriais/matriciais e em sistemas distribuídos, respectivamente.

Nos processadores matriciais (*arrays processors*), os elementos processadores são conectados por uma única rede de controle. Esta decodifica seqüencialmente as instruções e as transmite para todos os elementos processadores. Os processadores que não estiverem inibidos (mascarados) executarão simultaneamente as instruções recebidas sobre os dados previamente carregados em suas memórias [16].

Os processadores vetoriais (*vector processors*) utilizam a técnica *pipelining*, que introduz concorrência em um sistema de computação, particionando em várias subfunções, as funções que normalmente serão chamadas repetidamente. O *hardware* necessário para avaliar cada uma dessas subfunções é chamado de *estágio*.

Essas arquiteturas são normalmente usadas para solucionar problemas, cujo paralelismo inerente pode ser expresso através de uma estrutura de dados. Para a programação dessas arquiteturas, as linguagens devem permitir a definição de estruturas de dados a serem tratadas como objetos no quais operações podem ser realizadas em paralelo. Em FORTRAN 8x (90) e ACTUS, as estruturas de dados paralelas são os *arrays* e ambas as linguagens oferecem construções para a manipulação destes *arrays*.

Nos sistemas de memória distribuída, cada processador tem sua própria memória e utiliza uma rede de interconexão para se comunicar com os outros processadores via troca de mensagens. Os processadores funcionam de modo assíncrono. Estas arquiteturas normalmente são usadas para resolver problemas, cujo paralelismo inerente pode ser expresso através de paralelismo entre processos.

As linguagens de programação para estas arquiteturas devem possuir estruturas de dados e de controle similares àquelas requeridas no ambiente de programação seqüencial e, ainda, de ferramentas para controlar a interação entre os processos, ou seja, para promover exclusão mútua e sincronização. Neste grupo de linguagens, encaixam-se OCCAM e CSP, por exemplo.

II.2 Características Básicas da Linguagem OCCAM

A linguagem OCCAM é uma implementação do modelo CSP [26] para programação de sistemas distribuídos baseados em uma rede de *transputers*. O *transputer* é um microprocessador de 16 ou 32 bits, memória interna de até 4 kbytes, com quatro unidades de comunicação serial (*links*) e unidade de ponto flutuante. No modelo T800 a taxa de processamento é de 10 milhões de instruções por segundo e 2,25 MFLOPS. A capacidade de cada *link* é de 10 Mbits/segundo e os quatro *links* podem operar em paralelo. Através dos *links*, podem-se formar redes de computadores com qualquer número de *transputers*, onde todos podem operar concorrentemente.

Um programa em OCCAM corresponde a uma coleção de processos concorrentes, que pode, em princípio, ser executada em um número arbitrário de *transputers*. Os processos se comunicam sincronamente, isto é, somente quando um processo estiver pronto para enviar e um outro processo estiver pronto para receber a mensagem através de um canal lógico previamente estabelecido, é que ocorre a comunicação. Se os processos a se comunicarem estão em processadores distintos, os canais lógicos devem ser mapeados nos canais físicos *links* no *transputer*. O número de *transputers* na rede pode ser alterado e o programa em OCCAM pode ainda ser executado sem mudar sua descrição.

As próximas seções descreverão os processos primitivos, seqüenciais, de comunicação, alternados e paralelos de OCCAM.

II.2.1 A Construção de Processos

A linguagem OCCAM define três processos primitivos:

1. Atribuição, que muda o valor de uma variável. Por exemplo,

   ```
   X := X + 1
   ```

   incrementa o valor de X;

2. Entrada, que recebe um valor de um canal. Por exemplo,

   ```
   IN ? X
   ```

onde o símbolo "?" é usado para indicar uma operação de entrada. O efeito desse processo é receber um valor para a variável X através do canal IN;

3. Saída, que envia um valor para um canal. Por exemplo,

```
OUT ! EXPRESSAO
```

onde o símbolo "!" é usado para indicar uma operação de saída. O efeito desse processo é colocar o valor de EXPRESSÃO no canal OUT.

Se é necessário receber ou colocar mais que um valor em um canal, eles são então listados na ordem requerida. Por exemplo,

```
IN ? X1; X2
```

associa o primeiro valor do canal IN à X1 e o segundo valor à X2.

Os processos primitivos podem ser agrupados para compor processos mais complexos. Um programa em OCCAM é feito da combinação de processos primitivos e de construtores de processos seqüenciais (SEQ), paralelos (PAR) e altenativos (ALT). Os contrutores servem para agrupar processos especificando o seu modo de execução. Os processos agrupados podem ser processos primitivos e processos agrupados. Quando processos são agrupados com um construtor, diz-se que os processos compõem aquele construtor. Os construtores são os seguintes:

1. Seqüencial (SEQ): neste caso, os processos que o compõem são executados um após o outro, terminando após a execução do último componente.
2. Paralelo (PAR): neste caso, os processos que o compõem são executados concorrentemente, terminando quando todos os componentes tiverem terminado.
3. Alternativo (ALT): neste caso, um dos processos que o compõem é escolhido para ser executado, terminando após a execução do processo escolhido.

Na linguagem OCCAM, a identação do texto em um programa define o escopo de declarações e construtores. As declarações são válidas para o processo (primitivo ou construtor) definido imediatamente após as suas definições. Os processos que compõem um construtor são todos os processos que seguem a palavra chave (SEQ, ALT ou PAR) que têm uma identação maior.

II.2.2 Processos Seqüenciais (SEQ)

O construtor SEQ deve ser usado para agrupar processos que devem ser executados seqüencialmente.

Para exemplificar, será considerado um processo somador que recebe valores de um canal, soma-os e coloca o valor recebido em outro canal:

```
CHAN OF INT in, out:
INT soma:
SEQ
  soma := 0
  INT item:
  SEQ
    in ?  item
    soma := soma + item
    out !  item
```

II.2.3 Processos Paralelos (PAR)

O construtor PAR é usado para agrupar processos que serão executados em paralelo. Não há restrições quanto ao número de processos agrupados. Os processos componentes de um PAR podem ser processos seqüenciais.

Para exemplificar o construtor PAR, serão definidos dois processos somadores com características similares e que podem operar em paralelo:

```
CHAN OF INT in1, in2:
CHAN OF INT out1, out2:
INT soma1, soma2:
SEQ
  soma1 := 0
  soma2 := 0
  PAR
    -- somador 1
    WHILE TRUE
      VAR item1:
      SEQ
        in1 ?  item1
        soma1 := soma1 + item1
        out1 !  item1
    -- somador 2
    WHILE TRUE
      VAR item2:
      SEQ
        in2 ?  item2
        soma2 := soma2 + item2
        out2 !  item2
```

OCCAM também dispõe de um construtor paralelo PRI PAR, que permite definir prioridades para processos paralelos, quando houver disputa de recursos entre eles. A prioridade dos processos é definida pela sua ordem de definição no texto, da maior para a menor prioridade.

II.2.4 Processos de Comunicação

Em OCCAM, as variáveis não podem ser compartilhadas por processos executados em paralelo. Como conseqüência, os processos devem comunicar-se para trocar valores utilizando os canais de comunicação. Cada canal é usado exclusivamente por dois processos, e um processo pode ter um número qualquer de canais, podendo comunicar-se com vários processos concorrentemente.

Os processos de comunicação são os processos primitivos de entrada (?) e saída (!). A comunicação é síncrona, ou seja, ambos os processos devem estar prontos para a comunicação, afim de que se estabeleça um *rendezvous*, para promover a passagem de informações através de um canal. Após a comunicação os processos prosseguem com as suas execuções de forma independente.

A linguagem permite que protocolos de comunicação sejam definidos para os canais. O protocolo define a quantidade e o tipo dos dados que serão enviados e recebidos pelo canal.

II.2.5 Processos Alternativos (ALT)

O construtor ALT serve para agrupar processos que dependem da realização de uma comunicação para serem executados. Cada processo componente tem como primeiro comando um processo primitivo de entrada, e apenas um desses processos componentes é executado: aquele cuja comunicação pode ser realizada.

O construtor ALT pode ainda ser visto como uma ferramenta que permite que um processo com um número qualquer de canais associados a ele selecione uma ação, dependendo dos processos de comunicação realizados.

Cada processo componente do ALT pode ter associada a si uma **guarda**, para introduzir critérios de escolha diferentes para cada processo. Um processo é considerado pronto para ser executado quando sua expressão booleana (guarda) é verdadeira e o processo de entrada pode ser executado.

O processo alternativo espera até que um de seus processos guardados esteja pronto para ser executado. Um dos processos prontos é então escolhido aleatoriamente e é executado, após o qual a execução da construção termina.

É possível associar uma prioridade ao processo alternativo; então, se mais de uma guarda está pronta, a primeira na seqüência textual é selecionada, ou seja, a prioridade é dada pela ordem de declaração dos processos componentes no texto. A prioridade é indicada pelas palavras PRI ALT.

O exemplo abaixo ilustra o uso de um construtor ALT que tem dois processos componentes:

```
-- algumas declaracoes
ALT
```

```
  -- primeiro processo
(NUMERO > 0) & LINK1 ?  ITEM1
    SEQ
      NUMERO := NUMERO - 1
      SOMA := SOMA + ITEM1
  -- segundo processo
  LINK2 ?  ITEM2
    SOMA := SOMA + ITEM2
```

Para que um processo componente possa ser selecionado, é preciso que exista um processo pronto para comunicar-se com ele.

II.2.6 Processos Replicados

A linguagem OCCAM permite que processos sejam replicados e para isso fornece replicadores para serem usados com os construtores, da seguinte forma:

- ALT i = 0 FOR N
  processo

- SEQ i = 0 FOR N
  processo

- PAR i = 0 FOR N
  processo

Nesses exemplos, o processo será replicado N vezes, onde N deve ser uma constante para PAR e ALT e pode ser uma variável para SEQ. Por exemplo,

```
VAL N IS 4;
CHAN OF INT in, out:
INT soma:
SEQ
  soma := 0
  SEQ i=0 FOR N
    WHILE TRUE
      INT item:
      SEQ
        in ?  item
        soma := soma + item
        out !  item
```

irá criar N processos somadores que serão executados seqüencialmente, isto é, na ordem em que eles foram criados.

Se o replicador é usado com PAR, será criado um *array* de N processos paralelos que serão executados concorrentemente. Se ALT é usado, ao invés de PAR, significa que um dos N processos componentes será selecionado de acordo com as regras especificadas para esse construtor.

II.3 A linguagem ACTUS

A abordagem usada no projeto da linguagem ACTUS II [4] foi a de considerar as áreas de problemas nas quais os supercomputadores são usados e isolar as características relevantes desses problemas. O passo seguinte foi formar as regras sintáticas e semânticas para essas abstrações, e ao mesmo tempo tentar assegurar que um compilador com essas características possa ser implementado com eficiência [2]. A linguagem possibilita ao programador:

1. ignorar as características de *hardware*, liberando o programador para preocupar-se apenas com os algoritmos paralelos a serem implementados.
2. expressar o paralelismo do problema diretamente;
3. projetar a solução do problema em termos de uma extensão de paralelismo variável, ou seja, a quantidade de ações a serem executadas em paralelo pode variar.
4. controlar o processamento paralelo através de estruturas de controle explícitas.

Mais precisamente, ACTUS amplia as estruturas de dados e de programa de PASCAL (com exceção de conjuntos, registros variantes e ponteiros) para a programação de processadores matriciais e vetoriais.

Os computadores matriciais/vetoriais foram desenvolvidos como um meio de realizar a mesma operação em dados independentes em paralelo, portanto, são os dados que devem indicar a extensão de paralelismo. Para expressar esse princípio em uma linguagem de alto nível, é definido em ACTUS que cada declaração de um ítem de dado deve ter associada a si uma extensão máxima de paralelismo.

A extensão de paralelismo em um processador matricial é o número de processadores que poderiam logicamente fazer cálculos ao mesmo tempo com uma determinada estrutura de dados (esse valor pode ser maior, menor ou igual ao número de processadores disponíveis); para um processador vetorial, ela significa o tamanho da estrutura de dados apresentada ao processador para fazer o cálculo. A extensão de paralelismo é, então, explicitamente indicada nas declarações de dados e manipulada pelos comandos da linguagem. Deve ser enfatizado que a extensão de paralelismo é um conceito central na linguagem e possibilita uniformizar as construções da linguagem para programação de processadores vetoriais e matriciais.

As principais características da linguagem são:

1. facilidades para estruturação do programa e construção de tipos de dados pelo usuário;
2. possibilidade de expressar a natureza paralela de um problema a nível de declarações e comandos;
3. controle estático da extensão de paralelismo através dos conjuntos de índices (*index sets*);
4. controle dinâmico de paralelismo;
5. alinhamento de dados (*shift* e *rotate*) e indexação independente. A indexação independente é a especificação de um vetor contendo os índices dos elementos de uma matriz a ser manipulada em paralelo.

II.3.1 Estruturas de Dados

Arrays **Paralelos**

A estrutura de dados matricial é a única estrutura para a qual pode ser declarada uma extensão de paralelismo. A extensão de paralelismo é usada como um meio de expressar a quantidade de ações que podem ser aplicadas naquela estrutura de dados em paralelo. Por exemplo, nas declarações

```
const
  N = 200 ;
var
  PARALELO : array [1:N] of REAL ;
  ESCALAR : array [1..N] of REAL ;
```

o símbolo ":" indica que a extensão máxima de paralelismo para o *array* PARALELO é N, ou seja, no máximo N elementos podem ser operados em paralelo. Os N elementos podem ser manipulados em modo idêntico ao mesmo tempo. Por exemplo, a expressão envolvendo PARALELO[1:200], fará com que todos os 200 elementos do *array* sejam acessados em paralelo, ao contrário dos elementos do *array* ESCALAR que podem ser manipulados somente um elemento a cada vez.

A referência PARALELO[I:J] selecionará os elementos de I até J, inclusive, da extensão máxima de paralelismo declarada. Por exemplo, PARALELO[10:50] seleciona os elementos PARALELO[10], PARALELO[11],..., PARALELO[49], PARALELO[50].

Uma matriz pode ter um número qualquer de dimensões, no entanto, no máximo podem ser definidas duas dimensões paralelas. No exemplo abaixo, o *array* paralelo

```
var
  PARALELO : array [1:P, 1:Q] of INTEGER ;
```

define um *array* paralelo bidimensional com um total de P*Q elementos, onde todos podem ser acessados simultaneamente. A extensão máxima de paralelismo para essa variável é uma grade formada de 1:P e 1:Q. As referências às variáveis paralelas são feitas utilizando-se extensões de paralelismo. A variável PARALELO pode ser acessada usando essa extensão de paralelismo ou uma extensão menor que essa, por exemplo, 2:P-1,2:Q-1, que exclui as linhas e colunas do perímetro do *array* original.

As dimensões de *array* paralelo são válidas para qualquer tipo comum de PASCAL, como REAL, INTEGER, BOOLEAN e outros.

Conjuntos de Índices

Para permitir o acesso simultâneo a todos ou a determinados elementos de uma variável paralela, ACTUS fornece os *conjuntos de índices.* Os membros de um conjunto de índices são normalmente valores inteiros, cada um identificando um elemento particular de uma estrutura de dados que pode ser acessada em paralelo. Existem dois tipos de conjuntos de índices disponíveis:

1. conjunto de índices explícito, que se mantém constante por todo o bloco para o qual foi definido, e
2. conjunto de índices redefinível, que possibilita a seleção de diferentes porções de uma variável paralela. Ele deve ser declarado com um dos tipos primitivos INTEGER, BOOLEAN ou CHAR e ter valores específicos associados a ele em tempo de execução.

A forma de declaração dos conjuntos de índices é:

```
index
  CI_EXPLICITO: 1:15;
  CI_REDEFINIVEL: INTEGER;
```

Um identificador de conjunto de índices pode ser usado para indexar uma variável paralela, por exemplo, dada a declaração

```
var
  PARA : array[1:100] of INTEGER ;
```

então PARA[CI_EXPLICITO] acessará os 15 primeiros elementos de PARA simultaneamente.

A atribuição de valores aos conjuntos de índices redefíniveis é feita através do comando **using**, a ser explicado a seguir.

Em ACTUS, um único conjunto de índices é usado para acessar (em paralelo) os elementos de um *array* paralelo unidimensional ou uma linha, coluna ou diagonal de *array* paralelo bidimensional. Para manipular a grade de elementos de um *array* paralelo bidimensional, dois conjuntos de índices são necessários. Por exemplo,

```
var
  MAT_PARA : array[1:50, 1:50] of INTEGER ;
index
  CI, CJ : 1:50 ;
```

Dada a declaração acima, MAT_PARA[1, CJ] acessará todos os 50 elementos da primeira linha de MAT_PARA simultaneamente; MAT_PARA[CI,3] acessará todos os elementos da terceira coluna de MAT_PARA simultaneamente; e MAT_PARA[CI, CJ] acessará todos os 250 elementos de MAT_PARA simultaneamente.

Para possibilitar definir padrões de processamento paralelo mais flexíveis, podem-se formar combinações de conjuntos de índices usando os operadores de conjuntos união (+), interseção (*) e diferença (−). É também possível definir conjuntos de índices com um incremento regular, definindo o valor apropriado de incremento entre colchetes. Por exemplo,

```
index
  CI1 :  1:[3]15 ;
  CI2 :  0:3 + 7:[2]18 + 23:23 ;
```

Constantes Paralelas

Em ACTUS, podem-se definir constantes que têm como valor uma seqüência de números e conseqüentemente pode-se usá-las em expressões ou atribuições para variáveis paralelas. O número de valores deve ser o mesmo da extensão de paralelismo da variável ou expressão. A forma de constantes paralelas é como segue:

```
parconst
  SEQUENCIA = 1:50 ;
```

Essa expressão faz com que 50 valores seqüenciais 1, 2, 3 .., 50 sejam gravados e identificados pelo identificador SEQUENCIA.

Qualquer valor inteiro para início, fim e incremento pode ser usado para definir uma constante paralela. O valor de incremento constante é colocado entre colchetes. Por exemplo,

```
parconst
  CP1 = 100:[-2]84 ; (* 9 valores *)
  CP2 = -2:[4]14, 1:[3]22, -17:[6]91 ; (* 28 valores *)
```

Pedaços de seqüências podem ser indicados, usando uma vírgula para representar os valores inexistentes. Por exemplo,

```
parconst
  SEQINCOMPLETA = 5:10, 100:104 ;
```

que representa uma constante paralela com 11 valores.

II.3.2 Comandos Paralelos

Atribuição

Os comandos de atribuição (atribuição de expressões a variáveis paralelas), devem ser englobados pelo comando **using** de ACTUS. A estrutura **using** define uma expressão com conjuntos de índices, que é a extensão de paralelismo para os comandos que seguem a palavra **do**. A forma geral para o comando **using** é

```
using especificação de índices do comandos;
```

Esse comando agrupa comandos com referências à *arrays* paralelos, que usam os mesmos conjuntos de índices. Isso faz o programa mais legível e facilita a implementação da linguagem.

A especificação de índices contém um identificador de um conjunto de índices explicito ou o identificador de um conjunto de índices redefinível com o seu conjunto de valores atual. No máximo, duas especificações podem ser associadas a qualquer comando **using**. Resumindo, a especificação em um comando **using** representa um conjunto que é uma máscara unidimensional ou bidimensional para os seus comandos.

A extensão de paralelismo deve ser a mesma em ambos os lados da expressão. Por exemplo,

```
var
  PARA : array[1:10, 1:100] of INTEGER ;
index
  CI : 1:10 ;
  CJ : INTEGER ;
begin
  using CI, CJ := 1:100 do
    PARA[CI, CJ] := 1;
```

```
  using CI, CJ := 1:1 do
    PARA[CI, CJ] := 2;
end;
```

Na primeira atribuição, todos os 1000 elementos de PARA serão operados simultaneamente e, na segunda atribuição, todos os elementos da primeira coluna de PARA serão acessados simultaneamente.

Comandos de Seleção

São dois os comandos de seleção de ACTUS: **if** e **case**. A expressão booleana do comando **if** é avaliada em tempo de execução, produzindo um conjunto de valores *TRUE*. Esse conjunto é então usado como a extensão de paralelismo para as sentenças da parte **then** do *if*. O conjunto complementar a este é usado para executar as sentenças da parte **else**. Por exemplo,

```
var
  PARA1, PARA2 :  array[1:10, 1:100] of INTEGER ;
index
  CI : 1:10 ;
  CJ : 1:100 ;
begin
  using CI, CJ do
    if PARA1[CI, CJ] > 0 then
      PARA2[CI, CJ] := PARA2[CI, CJ] - 1
    else
      PARA2[CI, CJ] := PARA2[CI, CJ] + 1
end;
```

Nesse exemplo, os elementos de PARA2, cujos elementos correspondentes em PARA1 forem positivos (se houver algum), serão decrementados de 1 e o resto dos componentes (se houver algum) de PARA2 terão seus valores incrementados em uma unidade.

Além dos operadores booleanos **and**, **or** e **not** de PASCAL, ACTUS II possui mais dois operadores booleanos **any** e **all**, que podem ser usados para aplicar um teste através de uma extensão de paralelismo. Por exemplo,

```
using CI, CJ do
  if any PARA1[CI, CJ] > 58 then
    PARA1[CI, CJ] := 0;
```

Nesse exemplo, se algum elemento de PARA1 tiver o valor maior que 58, todos os seus elementos receberão o valor 0.

O comando **case** tem o seguinte formato geral

```
case seletor of
  lista1-valores-case :  comandos1;
  lista2-valores-case :  comandos2;
  listan-valores-case :  comandosn
end;
```

Se o seletor é uma variável paralela, a extensão de paralelismo da expressão é distribuída através das opções do **case**, comparando cada elemento do seletor com as listas de valores das opções. Logo, cada opção terá uma extensão de paralelismo diferente associada a ela. As opções serão executadas seqüencialmente na ordem em que foram definidas, cada uma com a extensão de paralelismo apropriada. Por exemplo,

```
var
  PARA1 :  array[1:10, 1:100] of 100..105 ;
index
  CI : 1:10 ;
  CJ : 1:100 ;
begin
  using CI, CJ do
    case PARA1[CI, CJ] of
      100,105:  PARA1[CI, CJ] := -1;
      101..104:  PARA1[CI, CJ] := 1
    end;
end;
```

Esse comando faz com que os elementos de PARA1 especificados na extensão de paralelismo sejam testados, para descobrir qual comando ou grupo de comandos devem ser aplicados para qual elemento do *array*. Sendo assim, os elementos de PARA1 que são iguais a 100 ou 105, recebem o valor -1 e os elementos cujo valor esteja entre 101 e 104 recebem o valor 1.

Comandos de Repetição

A execução de um grupo de comandos algum número de vezes pode ser especificada usando uma das seguintes três construções: **repeat**, **while** e **for**. Essas construções correspondem às construções de PASCAL, sendo que em ACTUS foi associada uma extensão de paralelismo às expressões de testes e variáveis de controle desses comandos. No caso do comando **while**,

```
while expressão booleana do comandos;
```

A expressão booleana pode incluir variáveis paralelas ou escalares ou uma mistura de ambas. O resultado da expressão booleana em cada iteração determina a extensão de paralelismo para os comandos. Por exemplo,

```
var
  PARA1, PARA2 :  array[1:10, 1:10] of INTEGER ;
index
  CI, CJ : 1:10 ;
begin
  using CI, CJ do
    while PARA1[CI, CJ] > 0 do
      begin
        PARA2[CI, CJ] := PARA2[CI, CJ] + PARA1[CI, CJ];
        PARA1[CI, CJ] := PARA1[CI, CJ] - 6
      end;
end;
```

No exemplo acima, o corpo do comando **while** será executado repetidas vezes, enquanto pelo menos um dos componentes de PARA1 é maior do que zero. A extensão de paralelismo antes da primeira avaliação da expressão booleana é 1:10, 1:10. A cada iteração do corpo do *loop*, os componentes de PARA1[1:10, 1:10] que tem um valor menor ou igual a zero serão removidos da extensão de paralelismo. A execução do *loop* termina quando a extensão de paralelismo torna-se vazia, ou seja, quando não há mais elementos para os quais o teste é verdadeiro.

Um meio menos flexível de especificar repetição é usar o comando **repeat**. Ele é menos flexível porque a expressão de teste vem depois dos comandos do corpo, não sendo possível ter uma extensão de paralelismo variável. Dessa forma, a expressão booleana do comando deve gerar apenas um valor booleano, ao contrário do WHILE que gera uma extensão de paralelismo. Sendo assim, as variáveis paralelas, na expressão booleana, são testadas com um dos dois contrutores **all** ou **any**. Por exemplo,

```
var
  PARA1, PARA2 :  array[1:100] of INTEGER ;
index
  CI : 1:100 ;
begin
  using CI do
    repeat
      PARA1[CI] := PARA2[CI];
      if PARA2[CI] < 0 then
        PARA2[CI] := PARA2[CI] - 1;
    until all (PARA1[CI] > 0);
end;
```

Nesse exemplo, todos os elementos de PARA1 e PARA2 especificados pelo conjunto de índices CI serão manipulados dentro do corpo do REPEAT, tantas vezes quantas forem as iterações executadas. O corpo do REAPEAT do exemplo será executado até que **todos** os elementos de PARA1 sejam maiores que zero.

II.3.3 Subprogramas

Em ACTUS, os procedimentos e as funções podem manipular variáveis escalares ou paralelas. As variáveis locais em um bloco do programa não podem ter suas extensões de paralelismo alteradas como resultado de uma chamada a um procedimento ou função. As listas de parâmetros para ambos os tipos de subprogramas permitem variáveis paralelas de uma ou duas dimensões. A extensão máxima de paralelismo para cada variável deve ser conhecida em tempo de compilação.

ACTUS adota o mesmo esquema de PASCAL no que diz respeito a definição de parâmetros formais do tipo *array* cujas dimensões não são especificadas, sendo conhecidas apenas em tempo de execução. Por exemplo:

```
procedure P1 ( var PARA:array[I..J: INTEGER] of INTEGER;
```

Na lista desse tipo de parâmetros, o método pelo qual a extensão de paralelismo corrente associada a um *array* paralelo é definida é mostrado no exemplo abaixo:

```
procedure P2 ( PARA:array[index CI: INTEGER] of INTEGER;
```

A procedure acima poderia ser chamada da seguinte forma:

```
var PARA: array[1:10] of INTEGER;
index CI1 :  1:10;
⋮
using CI1 do P2(PARA[CI1]);
```

As funções de ACTUS podem ser usadas para retornar *arrays* paralelos como resultado, de forma semelhante à definição de funções em PASCAL.

II.3.4 Alinhamento de Dados

Para possibilitar o movimento de dados entre componentes da mesma ou diferentes estruturas paralelas, dois operadores de alinhamento estão disponíveis: **shift** e **rotate**. O operador **shift** causa um movimento de dados dentro dos limites da extensão de paralelismo declarada. O operador **rotate** causa um deslocamento circular dos dados, dentro dos limites da extensão de paralelismo atual. Cada operador é aplicado à extensão de paralelismo com a qual ele está associado para produzir uma nova extensão de paralelismo.

Os seguintes comandos, por exemplo, inicializarão as diagonais maior e menor da grade PARA para zero:

```
var
  PARA : array[1:10, 1:10] of INTEGER ;
index
  CI : INTEGER ;
begin
  using CI:= 1:9 do
    begin
      PARA[CI, CI shift 1] := 0; (* diagonal maior *)
      PARA[CI shift 1, CI] := 0; (* diagonal menor *)
    end;
end;
```

II.3.5 Indexação Independente

Para acessar componentes aleatórios de um *array* com duas ou mais dimensões, um *array* paralelo de uma dimensão pode ser usado. Os índices apropriados são atribuídos para esse *array* unidimensional, que é usado como subscrito de um *array* com duas ou mais dimensões para promover uma indexação independente. Por exemplo,

```
const
  N = 50;
parconst
  CP = N:[-1]1 ;
var
  MATRIZ_PARA : array[1:N,1:N] of INTEGER ;
  VETOR_PARA : array[1:N] of INTEGER ;
index
  CI : 1:N ;
begin
  using CI do
    begin
      VETOR_PARA[CI] := CP;
      MATRIZ_PARA[CI, VETOR_PARA[CI]] := 0;
    end;
end;
```

O *array* VETOR_PARA é usado para possibilitar que os elementos da diagonal secundária do *array* MATRIZ_PARA sejam acessados em paralelo.

Capítulo III

Especificação da Linguagem Intermediária

III.1 Introdução

Os compiladores trabalham na representação de um programa em alguma linguagem de alto nível e a convertem para uma representação em forma de código de máquina ou montagem, ou ainda para código binário relocável. No modelo de análise e síntese de um compilador, o *front end* traduz um programa fonte em uma representação intermediária, da qual o *back end* gera o código alvo. O compilador pode ter vários passos, existindo então várias representações intermediárias, que consistirão de alguma estrutura de dados. Uma linguagem intermediária (LI) consiste de um conjunto relativamente pequeno de operações simples, que, quando combinadas, possibilitam formar uma lista que equivale semanticamente a um programa fonte qualquer [21]. Além disso, elas se caracterizam por fornecerem uma representação independente e padronizada do programa, geralmente em uma forma similar às linguagens de montagem convencionais, sendo normalmente expressadas em forma de caracteres . Embora o programa fonte possa ser traduzido diretamente para a linguagem alvo, existem benefícios em se usar uma forma intermediária independente de máquina, a saber:

1. A tarefa de compilação pode ser dividida em problemas menores e com a introdução da linguagem intermediária, esses problemas tornam-se logicamente independentes.

2. Obtém-se uma maior portabilidade nos sistemas de linguagens – pode-se implementar uma nova linguagem em uma máquina ou uma linguagem em uma nova máquina, bastando para isso rescrever o compilador (*front end*) ou o tradutor (*back end*).

Na prática, a segunda razão é muito mais importante. Uma terceira vantagem obtida pelo uso de LI's, é a facilidade que se tem para aplicar um otimizador de código independente de máquina à representação intermediária.

O projeto desta LI para o Compilador ACTUS teve várias fases. Primeiro, foi feito um estudo de algumas LI's [18] [19] [20]. Depois, estudamos ACTUS e OCCAM, uma vez que o código a ser gerado é OCCAM. Durante esta fase, estudamos também implementações da linguagem ACTUS no CRAY-1 [6] e em uma rede de *transputers* [7]. Neste último, os programas escritos na linguagem usariam toda a rede, isto é, seriam transformados em uma coleção de processos. No projeto, a linguagem objeto também é OCCAM. Por último, dividimos o trabalho em duas fases: projeto de uma LI para parte escalar de ACTUS e de uma LI para a parte paralela de ACTUS. Finalmente, fizemos a unificação de ambas as linguagens, obtendo a LI a ser apresentada neste trabalho. Durante todo o projeto, observamos os problemas de aninhamento, escopo, facilidade de geração de código intermediário pelo compilador e facilidade de geração de código OCCAM pelo gerador de código. Esses cuidados dizem respeito à capacidade de traduzir programas em ACTUS para a LI, preservando a semântica do programa fonte.

A LI aqui proposta é específica da linguagem ACTUS para *transputers*. Nesse contexto, ela foi projetada com duas preocupações básicas em mente:

1. Conservar integralmente o paralelismo expresso no programa fonte;
2. Conservar a estrutura dos comandos de ACTUS que podem ser mapeados quase que diretamente em comandos de OCCAM, sem com isso obrigar o tradutor a fazer qualquer análise que já tenha sido feita pelo compilador.

Essas duas preocupações visam obter uma maior portabilidade para o compilador e uma maior eficiência no processo de compilação.

Como mencionado anteriormente, a natureza precisa da LI dependerá dos objetivos do projeto. O objetivo do projeto em questão é produzir boas implementações de ACTUS em arquiteturas paralelas baseadas em *transputers*. Essas definições dão um caráter especial a LI, que apesar de específica da linguagem ACTUS, sofreu muitas influências de OCCAM. Como conseqüência, o compilador ACTUS pode ser implementado em máquinas que disponham da linguagem OCCAM. Isto não impede que o compilador seja implementado em outras máquinas, porém o tradutor a ser construído deverá ser bem mais complexo.

Para projetar uma LI, algumas decisões que dependem basicamente dos objetivos do projeto devem ser tomadas, tais como o nível da LI, se a LI será específica da linguagem ou da máquina e o grau de dependência da máquina que está embutido na LI.

Ao escolher projetar uma LI específica da linguagem, obtém-se algumas vantagens, tais como:

- é possível projetar esse tipo de LI com quase nenhuma dependência de máquina e se a escrita do tradutor da LI para cada máquina não for muito difícil, a portabilidade da máquina será aumentada;
- a maioria das aplicações bem sucedidas dos métodos de LI, tem usado a técnica de LI específica da linguagem [18]

As LI's específicas da linguagem são normalmente de alto nível. De uma forma geral, a LI proposta pode ser considerada de alto nível, em parte pelo fato de ambas as linguagens fonte e objeto serem de alto nível. Tais LI's geralmente facilitam o trabalho do compilador e dificultam o trabalho do tradutor. No nosso caso, a LI foi projetada visando à tradução para OCCAM, facilitando assim a tarefa do gerador de código.

Na LI proposta, o conjunto de operadores é rico o suficiente para a implementação das operações de ACTUS. No caso das estruturas de controle de ACTUS (*REPEAT, WHILE, etc.*), a LI conserva as características dessas estruturas. Entretanto, para as expressões aritméticas, ela tem as características das LI's mais usuais – as expressões são traduzidas para código de três endereços (*three-address code*) com quatro campos (quádruplas: operação, resultado e dois operandos). Esse tipo de representação facilita o trabalho de aplicação de um otimizador ao código intermediário. Alguns operadores de conversão de tipos de dados são fornecidos na LI (CON, TRU, ROU), pois em ACTUS, as expressões aritméticas podem manipular dados de tipos de diferentes, mas em OCCAM, os operandos de uma expressão aritmética devem ter o mesmo tipo.

Todas as instruções da LI são completamente independentes de máquina, e para expressar a dependência de máquina fornecendo informações sobre o tipo de paralelismo da máquina alvo, foi proposta a instrução **config**. Por ser apenas uma instrução, o trabalho de alteração do compilador e gerador de código é facilitado, caso se deseje implementar ACTUS em outra máquina alvo.

Há três observações importantes a serem feitas a respeito da LI:

1. Os problemas de incompatibilidade das linguagens ACTUS e OCCAM, tais como: comunicação e sincronização (não existentes em ACTUS), não foram resolvidos neste trabalho.

2. Todas as construções de ACTUS são mapeadas em um subconjunto de construções de OCCAM. Por exemplo, a construção ALT e processos de comunicação não são utilizados na tradução dos comandos de ACTUS.

3. A LI usou mnemônicos para representar as instruções, em vez de códigos númericos, para facilitar a codificação manual de todos os testes, uma vez que o *front-end* do compilador ainda não está pronto. A troca de mnemônicos para códigos númericos pode ser feita sem problemas, já o que Gerador de Código foi projetado visando esta futura modificação.

As instruções da LI serão apresentadas em uma forma simbólica e informal. A apresentação das instruções é dita simbólica, pois estão ausentes as informações que visam apenas o aspecto prático da implementação. No Anexo A, tem-se as instruções na sua forma de implementação e a apresentação formal das instruções da LI, por meio de uma BNF.

Na descrição das instruções, os seguintes símbolos foram usados:

fim_bloco $\Rightarrow$ número de instruções do bloco;

fim_cod_teste ⇒ número de instruções do código de teste;

fim_bloco_then ⇒ número de instruções do bloco *then*;

fim_par ⇒ número de declarações de parâmetros;

fim_dec ⇒ número de declarações de **dados**;

nome_dado ⇒ nome de uma variável;

nome_label ⇒ literal;

nome_subrotina ⇒ nome de uma subrotina;

Na terminologia usada para descrever a linguagem, foi determinado que o significado de um campo entre {}, significa ou que o campo é opcional, podendo ocorrer uma ou mais vezes ($\{_0$, campo$\}$), ou que o campo deve ocorrer pelo menos uma vez ($\{_1$, campo$\}$).

III.2 Estruturas de Dados

III.2.1 Tipos de Dados Primitivos

Os tipos de dados definidos na LI são:

- INT (32 bits)
- INT16 (16 bits)
- INT32 (32 bits)
- INT64 (64 bits)
- REAL32 (8 bits para expoente e 23 bits para fração, Padrão ANSI-IEEE (754-1985))
- REAL64 (11 bits de expoente e 52 bits para fração, Padrão ANSI-IEEE (754-1985))
- BYTE (8 bits)
- BOOLEAN

O nome do **dado** deve ser iniciado por uma letra, seguido de letras ou dígitos.

III.2.2 Alocação de Memória

A LI permite a alocação de um único **dado** ou de um grupo de **dados**. Os **dados** dentro de um grupo devem ter o mesmo tipo e um nome é dado ao grupo. A esse grupo chamamos *array*. Um *array* expressa um número qualquer de ocorrências de um tipo de **dado**. Os *arrays* podem ser organizados **n**-dimensionalmente e ocupam uma área contígua de memória.

A alocação é feita através do comando *DEC*. No caso de alocação de um **dado**, tem-se:

DEC (nome_dado, tipo)

onde *tipo* é um dos tipos descritos anteriormente. No caso de alocação de *arrays*, tem-se:

DEC (nome_dado, ↑)

onde ↑ aponta para uma tabela (tabela de *arrays*) onde estão localizadas informações de tipo do *array*, número de dimensões, valor máximo de cada dimensão e indicação de quais dimensões do *array* são paralelas.

A informação de quais dimensões são paralelas refere-se à forma pela qual será processado o *array*. Isso será melhor explicado na seção III.2.3.

III.2.3 Endereçamento

Uma vez alocado o **dado**, o seu endereçamento é feito, em primeira instância, através do nome a ele associado. Em se tratando de *arrays*, o endereçamento dependerá da forma pela qual o *array* será processado. As duas possíveis formas são:

i) Processamento Escalar

Processar um *array* de forma escalar significa executar uma instrução para processar cada elemento (**dado**) do *array*. Nesse caso, as instruções devem endereçar a memória dando o nome do *array* e o(s) índice(s) do **dado** dentro do grupo. O primeiro **dado** dentro de cada dimensão tem o índice de valor 0, o segundo tem o índice de valor 1, e assim sucessivamente.

ii) Processamento Paralelo

Processar um *array* de forma paralela significa executar uma instrução que opere ao mesmo tempo sobre mais de um elemento do *array*. Nesse caso, as instruções devem endereçar à memória dando o nome do *array* e informando quais os **dados** a serem processados.

Para endereçar **um dado**, deve-se antepor ao *nome_dado* uma das **letras gregas** α ou β, cujo significado será explicado adiante, e que indica o modo de processamento dos dados e a forma como eles devem ser endereçados. Os dados processados escalarmente têm um dos seguintes endereçamentos:

$\alpha \Rightarrow$ o endereço é dado apenas pelo *nome_dado* e

$\beta \Rightarrow$ o endereço é dado pelo *nome_dado* e os índices que o seguem.

Nos dados processados paralelamente, os endereços são dados pelo *nome_dado*, seguido pelas definições das extensões de paralelismo. Na realidade, o que se tem após *nome_dado* são ponteiros para as posições de memória, onde se encontram as extensões de paralelismo. Uma extensão de paralelismo diz quais os elementos de uma dimensão de um *array* que serão processados em paralelo e é composta por três campos, a saber:

(elemento_inicial, incremento, número_elementos)

Com esses dados é possível obter os índices dos elementos a serem processados. Para exemplificar o conceito de extensão de paralelismo, seja o seguinte endereço:

$$\mathrm{AA}\ (\ \Uparrow_1, \Uparrow_2\ )$$

onde $\Uparrow_1$ aponta para (1, 2, 3) e $\Uparrow_2$ aponta para (2, 2, 3), na tabela de extensões de paralelismo (tabela de edp's). Logo, a primeira extensão de paralelismo denota os elementos $1, 3$ e 5 na primeira dimensão e a segunda denota os elementos $2, 4$ e 6 na segunda dimensão.

A informação de quais dados serão processados, é dada pelo **tipo de endereçamento** dos dados. Os dois tipos possíveis são:

$\gamma \Rightarrow$ os índices denotados pelas extensões de paralelismo variam independentemente. Isso significa, para o exemplo anterior, que os dados a serem processados em paralelo são: $AA(1,2)$, $AA(1,4)$, $AA(1,6)$, $AA(3,2)$, $AA(3,4)$, $AA(3,6)$, $AA(5,2)$, $AA(5,4)$ e $AA(5,6)$.

$\delta \Rightarrow$ os índices denotados pelas extensões de paralelismo variam dependentemente. Isso significa, para o exemplo anterior, que os dados a serem processados em paralelo são: $AA(1,2)$, $AA(3,4)$, e $AA(5,6)$.

As constantes escalares e paralelas devem ser antecedidas por uma das letras gregas abaixo:

$\varphi \Rightarrow$ indica um literal escalar.

$\epsilon \Rightarrow$ indica um conjunto de valores que podem ser processados em paralelo (constante paralela). Os valores são denotados também por uma extensão de paralelismo.

A tabela III.1 resume os tipos de endereçamentos definidos na LI.

constante escalar	φliteral
variável escalar	αnome_dado
array escalar	βnome_dado $\{_1$, índice$\}$
array paralelo com variação independente de índices	γnome_dado $\{_1, \Uparrow \}$
array paralelo com variação dependente de índices	δnome_dado $\{_1, \Uparrow \}$
constante paralela	$\epsilon\{_1, \Uparrow \}$

Tabela III.1: Tipos de Endereçamento e as suas Notações

III.3 Processamento Escalar e Paralelo

As instruções cujos os operandos são escalares, ou seja, cada operando denota apenas um **dado** da memória ou uma constante, são chamadas instruções escalares. Essas expressões são instruções do tipo *three-address instruction* [20] e têm quatro campos: operação, operando resultado e dois operandos sobre os quais deve ser aplicada a operação. O seu formato geral é:

EXP(operação, operando.result, operando1, operando2)

As instruções que operam sobre grupos de *dados* são chamadas de instruções paralelas, e o seu formato geral é:

EXPP(operação, operando.result, operando1, operando2, máscara)

A máscara identifica quais elementos dos *arrays* especificados pelo endereçamento devem ser operados. A máscara é um *array* de **dados** do tipo booleano e deve ser endereçado na instrução. As operações especificadas em *EXPP* agem sobre cada elemento do(s) *array(s)* endereçado(s) pelo(s) operando(s).

As instruções escalares e paralelas, com exceção daquelas cuja operação é de conversão ou relacional, devem ter todos os operandos do mesmo tipo.

As operações podem ser monádicas (o campo operando2 é ignorado) ou diádicas e foram classificadas em aritméticas, operações sobre bits, operações booleanas, operações de conversão, operações relacionais e operação de atribuição.

As operações apresentadas a seguir, podem ser usadas tanto em instruções escalares com em paralelas. No caso de instruções paralelas, as operações são executadas, na forma como apresentadas, em cada elemento do(s) *array*(s) envolvidos – os elementos de mesmo índice são operados e o resultado é colocado no elemento de mesmo índice no *array* definido em *operand.result.*

III.3.1 Operações Aritméticas

As operações aritméticas diádicas são as seguintes :

- ADD - adição
- SUB - subtração
- MULT - produto
- DIV - divisão
- REM - resto

Os operandos devem ser inteiros ou reais. A única operação monádica nessa classe de operações é a operação *NEG*, que troca o sinal do operando.

III.3.2 Operações sobre Bits

Essas operações são feitas sobre o padrão de bits de valores do tipo inteiro. Os operadores diádicos são:

- BAND - *and*
- BOR - *or*
- BXOR - *or* exclusivo

O operador monádico é *BNOT* (negação).

Um outro tipo de operação feita em bits, são as operações de deslocamento. Essas operações produzem um deslocamento no padrão de bits de um valor inteiro. O *operando1* recebe o valor ou **dado** a ser deslocado e o *operando2* recebe o número de posições em que deve ser deslocado o *operando1.* Os operadores são:

- SHR - deslocamento à direita
- SHL - deslocamento à esquerda

III.3.3 Operações Booleanas

As operações agem sobre operandos booleanos produzindo um resultado booleano *TRUE* ou *FALSE*. As operações diádicas são:

- AND - *and*
- OR - *or*

A única operação monádica desta classe é a operação *NOT* (negação).

III.3.4 Operações Relacionais

As instruções que executam operações relacionais produzem um resultado booleano e devem ter os operandos de um mesmo tipo. Os seguintes operadores podem ser aplicados sobre operandos de qualquer tipo:

- = - igual
- <> - diferente

Os seguintes operadores podem ser aplicados sobre operandos do tipo inteiro, byte ou real:

- < - menor
- > - maior
- <= - menor ou igual
- >= - maior ou igual

III.3.5 Operações de Conversão

As operações de conversão permitem que se converta o tipo de um dado, através de uma atribuição.

O código do tipo desejado é colocado em *operando1* e o **dado** ou valor a ser transformado é colocado em *operando2*. O *operando.result* é do tipo especificado em *operando1* e recebe o valor convertido. As operações são:

- CON - conversão entre inteiros, entre inteiros e *bytes*, entre booleanos e inteiros ou *bytes* (válida para valores 0 e 1) e entre reais.
- ROU - conversão entre inteiros e reais. Na conversão, os valores são arredondados.

- TRU - conversão entre inteiros e reais. Na conversão, os valores são truncados.

No caso do operando *CON*, a conversão só será válida se o valor produzido estiver dentro do tamanho suportado pelo tipo do receptor.

III.3.6 Operação de Atribuição

A operação *COPY* copia um literal ou conteúdo de um **dado** para um **dado**. O endereço do **dado** receptor deve ficar no campo *operando.result*. O endereço do **dado** fonte ou literal deve ficar no *operando1*. O campo *operando2* é ignorado.

A operação também permite a cópia dos valores booleanos *TRUE* ou *FALSE* para operandos booleanos.

III.4 Mais Instruções Paralelas

Existem três tipos de instruções paralelas: expressões (apresentadas anteriormente), diretiva de configuração e inicialização.

III.4.1 Diretiva de Configuração

Essa instrução traz informações sobre as extensões de paralelismo a serem usadas em um grupo de intruções. A sintaxe dessa instrução é:

CONFIG ($\{_1$, ⇑ }, fim_bloco)
bloco_config

onde ⇑ aponta para os valores que definem as extensões de paralelismo necessárias a execução das instruções para as quais a configuração é válida. *fim_bloco* deve conter o número de instruções do *bloco_config*.

III.4.2 Inicialização de Vetores

Através dessa instrução, é possível atribuir, em paralelo, um conjunto de valores a um *array*. A sintaxe dessa instrução é:

CPVEC (nome_dado, ⇑, fim_instrução)
$\{_1$, CPVAL (numero_valores, $\{_1$, valor}}

onde *valor* é um literal representando um número inteiro ou real e ocorre *numero_valores* vezes em cada instruçaõ *CPVAL*. Na extensão de paralelismo apontada por ⇑, está a informação de quais elementos devem ser inicializados. O campo *fim_instrução* deve informar o número de instruções *CPVEC* da instrução *CPVAL*.

III.5 Instruções de Controle

Neste grupo, enquadram-se as instruções que podem modificar o fluxo de execução dos programas.

Além dos símbolos já descritos anteriormente, foram usados os seguintes símbolos na descrição destas instruções:

valor ⇒ valor inteiro;

var_controle ⇒ variável de tipo inteiro ou valor inteiro;

var_bool ⇒ variável de tipo booleano ou valor booleano e

var_seleção ⇒ variável de tipo booleano ou valor booleano.

III.5.1 Estrutura de Blocos

A LI permite a alocação de área de memória em qualquer ponto de um programa. Para qualquer área alocada deve-se definir o escopo de sua validade. Isso equivale a isolar um trecho do programa para o qual é válida aquela alocação. Esse grupo de comandos constitui um **bloco**.

Seguindo a(s) declaração(ões), deve-se definir o seu escopo, que pode ser apenas uma instrução ou grupo de instruções, e no último caso, essas instruções devem ser agrupadas em um bloco. A instrução para definição de um bloco é:

BLOCK (fim_bloco)
bloco

Resumindo, um bloco agrupa instruções e pode conter outros blocos.

III.5.2 Estruturas de Repetição

Estão incluídas nessa classe, todas as estruturas que controlam a repetição da execução de blocos de comandos. Para um número de repetições desconhecido, podem-se usar as estruturas *WHILE* e *REPEAT*. Caso contrário, se existe um número de repetições pré-determinado, a estrutura *FOR* poderá ser usada.

Na apresentação dessas estruturas ficará clara a diferença entre as duas primeiras (*REPEAT* e *WHILE*).

A Estrutura WHILE

A sintaxe do *WHILE* é:

WHILE (var_bool, fim_cod_teste, fim_bloco)
codigo_teste
bloco_while

onde *codigo_teste* consiste de expressões para avaliação de **dados**, que ao final tornam um **dado** booleano *TRUE* ou *FALSE*. O endereço de tal **dado** deve ser posto em *var_bool*. O *codigo_teste* não é obrigatório. O campo *fim_bloco* contém o número de instruções de *código_teste* e *bloco_while*.

O *bloco_while* contém um comando ou um bloco de comandos. Enquanto o **dado** booleano mencionado no parágrafo anterior for *TRUE*, o *bloco_while* será executado.

A Estrutura REPEAT

A sintaxe do **REPEAT** é:

REPEAT (var_bool, fim_bloco)
bloco_repeat
codigo_teste

Nessa estrutura, o bloco de comandos (*bloco_repeat*) será executado pelo menos uma vez. Como na estrutura anterior, as expressões de *código_teste* tornam o **dado** booleano endereçado em *var_bool*, *TRUE* ou *FALSE*. Após a primeira execução, o *bloco_repeat* só será executado, se o **dado** booleano mencionado anteriormente tiver o valor *TRUE*. O campo *fim_bloco* contém o número de instruções de *bloco_repeat* e *código_teste*.

A Estrutura FOR

A sintaxe do *FOR* é:

FOR (var_controle, fim_bloco)
bloco_for

O campo *var_controle* denota o número de vezes que deve ser executado o *bloco_for*. Nesse campo, pode-se ter um literal ou o endereço de um **dado**.

III.5.3 Estruturas de Desvios Condicionais

Estão incluídas nessa classe aquelas estruturas que promovem um desvio do controle de execução para um ponto ou outro do programa, a depender do resultado de um teste. São duas as estruturas dessa classe: *IF* e *CASE*.

A Estrutura IF

A sintaxe do IF é:

codigo_teste
IF (var_bool, fim_bloco_then, fim_bloco)
bloco_then
bloco_else

O campo *var_bool* endereça um **dado** booleano, cujo valor é testado no início da execução da estrutura: se for *TRUE*, o *bloco_then* de comandos será executado; caso contrário, se o valor de *var_bool* for *FALSE*, o *bloco_else* será executado.

A existência do *bloco_else* não é obrigatória. Nesse caso, o valor do campo *fim_bloco_then* é igual a *fim_bloco*. O campo *fim_bloco* denota o número de instruções do *bloco_then* e *bloco_else*, se este foi definido.

A Estrutura CASE

A sintaxe do *CASE* é:

CASE (var_seleção, fim_bloco)
{$_1$, OPTION (VD/VC, {$_1$, valor})
bloco_opção}

Essa estrutura é usada quando se tem vários pontos (opcões) de desvio em função do valor de uma variável (*var_seleção*).

O valor da variável *var_seleção*, é comparado com o(s) valor(es) *valor* especificados em *OPTION*, gerando um valor booleano. Se o resultado for *TRUE*, o *bloco_opção* de comandos é executado e o controle de execução do programa é desviado para a instrução após o endereço *fim_bloco*. Se o valor da comparação for *FALSE*, o dado *var_seleção* é comparado com o(s) valor(es) *valor* da próxima *OPTION* e assim sucessivamente até acabarem as opções ou algum *bloco_opção* ter sido executado.

O tipo de comparação feita entre *var_seleção* e *valor* depende de *VD/VC*, que significam respectivamente valores discretos e contínuos.

Se *OPTION* tem o valor *VD*, a comparação será:

var_selecao = $valor_1$ **ou**
var_selecao = $valor_2$ ...,
dependendo do número de valores especificados em *OPTION*.

Se *OPTION* tem o valor *VC*, existem dois valores e a comparação será:

var_selecao <= $valor_1$ **e** *var_selecao* >= $valor_2$

A estrutura *CASE* deve ter pelo menos uma opção.

III.5.4 Estrutura de Desvio Incondicional

A LI permite o desvio do controle de execução para qualquer ponto do programa. Para tanto, é necessário marcar cada ponto de desvio com um *LABEL* e o desvio para tal ponto é executado através da instrução *GOTO*. A sintaxe do *LABEL* é:

LABEL (nome_label)

O mesmo rótulo *nome_label* não pode marcar mais de um ponto do programa. A sintaxe do *GOTO* é:

GOTO (nome_label)

Esta instrução promove o desvio do controle de execução do programa para a instrução logo após o ponto marcado com *nome_label*.

III.5.5 Subrotinas

A LI permite a definição e uso de subrotinas. A criação de subrotinas é tratada, quanto ao escopo, como uma declaração, ou seja, ela é válida por todo o próximo bloco.

Existem dois tipos de subrotinas, cada uma com instruções distintas para criação e chamada. As duas são expostas nas subseções seguintes.

As subrotinas aceitam passagens de parâmetros, que podem ser passados de duas maneiras: por referência ou valor. No primeiro caso, passa-se o

endereço do parâmetro, permitindo que sejam feitas alterações em seu conteúdo dentro da subrotina. No segundo caso, passa-se apenas o valor do parâmetro, impossibilitando modificações no conteúdo do **dado**.

Por ocasião da criação das subrotinas, os parâmetros são chamados parâmetros formais ou parâmetros e na chamada, eles são chamados argumentos.

Funções

Esse tipo de subrotina retorna um valor resultado, o qual é copiado para um **dado** determinado pelo programador. O uso dessas subrotinas tem severas restrições:

- as instruções permitidas nessa subrotina são as apenas instruções de controle e as expressões escalares.
- a cópia de **dados** só pode ser feita para **dados** alocados dentro da subrotina.
- as subrotinas chamadas dentro desta devem obedecer às restrições anteriores.

Essas subrotinas são chamadas **funções**. Para criar uma **função**, tem-se:

FUNCTION (nome_subrotina, tipo, fim_par, fim_dec, fim_bloco)
$\{_1$, PAR (nome_dado, tipo, VAL)$\}$
$\{_0$, DEC (nome_dado, tipo)$\}$
bloco_function

O tipo de valor retornado pela função é dado pelo campo *tipo* da instrução *FUNCTION*. A função só aceita passagem de parâmetros por valor. É obrigatória a definição de pelo menos um parâmetro na função. A alocação (*DEC*) de **dados** não é obrigatória. Para chamar uma função, tem-se:

CALLFUNC (nome_dado, nome_subrotina, fim_par)
$\{_1$, CPAR (nome_dado)$\}$

O resultado da função será copiado para *nome_dado* de *CALLFUNC*. O nome da função é dado por *nome_subrotina* e *CPAR* descreve os argumentos da função.

Para a LI, existem duas funções pré-definidas particularmente importantes: **ifany** e **ifall**. Essas funções atribuem um valor a uma variável booleana em função dos valores de uma matriz ou vetor booleano. Se todos os valores do argumento são *TRUE*, a função **ifall** retorna *TRUE*. Se há pelo menos um valor do argumento *TRUE*, a função **ifany** retorna *TRUE*. Essas funções podem ser usadas para testar máscaras de comandos paralelos, atribuindo valores à variáveis booleanas usadas nos testes das estruturas de controle. Ambas as funções têm apenas um parâmetro que é a matriz ou vetor booleano sobre o qual serão aplicados os testes.

Procedimentos

Essas subrotinas não retornam um valor resultado e podem ter quaisquer instruções. Além disso, a passagem de parâmetros se dá por valor ou referência. A criação de procedimentos é feita com os seguintes comandos:

PROC (nome_subrotina, recursive, fim_par, fim_bloco)
$\{_0$, PAR (nome_dado, tipo, VAL/REF)$\}$
$\{_0$, DEC (nome_dado, tipo)$\}$
bloco_proc

A chamada de procedimento é feita com as seguintes instruções:

CALLPROC (nome_subrotina, recursive, fim_par)
$\{_1$, CPAR (nome_dado)$\}$

A LI permite que os procedimentos sejam chamados recursivamente. Para tanto o campo booleano *recursive* deve ser *TRUE* na definição do procedimento recursivo e na chamada recursiva, ou seja a chamada do procedimento que causa a recursão.

III.6 Instruções de Entrada e Saída

A LI tem instruções de leitura em teclado ou disco e de escrita em tela ou disco. Para tanto, os arquivos em disco devem ser abertos, antes do uso e fechados após o uso. Os arquivos teclado e tela não precisam ser abertos ou fechados.

Para a LI, um *arquivo* é um conjunto de dados de tipo primitivo, dispostos em qualquer ordem em subconjuntos de qualquer tamanho chamados *registros*. Os dados nos registros são separados por pelo menos um espaço (" "). No máximo, quatro arquivos em disco podem estar abertos ao mesmo tempo.

Cada arquivo possui um (*buffer*), que contém o registro de onde são lidos os dados requisitados pelos comando de leitura e onde são escritos os dados especificados pelos comandos de escrita. O *buffer* pode ser recarregado (leitura) ou descarregado (escrita) através de instruções.

Nas subseções abaixo, as instruções da LI para realizar entrada/saída são apresentadas.

III.6.1 Abertura de Arquivos

A abertura de um arquivo significa a inicialização do ponteiro de registros do arquivo, fazendo-o apontar para o primeiro registro daquele arquivo.

A LI permite que seja associado um nome lógico ao arquivo, e essa associação deve ser feita antes da abertura do arquivo. A instrução que associa nome lógicos a nomes físicos é:

assign (nome_lógico_arq, nome_físico_arq)

Existem dois modos de abertura de um arquivo: leitura e escrita. As instruções apropriadas para cada um deles são:

open (nome_lógico_arq) e create (nome_lógico_arq),

respectivamente.

III.6.2 Fechamento de Arquivos

Após o uso, os arquivos devem ser fechados antes do final da execução do programa em LI. O comando de fechamento dependerá do modo de abertura do arquivo. Se o arquivo foi aberto para leitura, o comando de fechamento é:

close (nome_lógico_arq)

Se, caso contrário, o arquivo tiver sido aberto para escrita, o comando é:

finish (nome_lógico_arq)

III.6.3 Leitura

A execução do comando de leitura equivale à leitura do proximo dado do *buffer* e a atribuição de seu valor a um dado especificado na instrução (este controle, de próximo dado, é feito na implementação da LI).

A LI possui instruções específicas para vários tipos de leitura. Elas são:

rd.char (nome_lógico_arq, nome_dado),
rd.int (nome_lógico_arq, nome_dado, tam),
rd.i64 (nome_lógico_arq, nome_dado, tam),
rd.i32 (nome_lógico_arq, nome_dado, tam),
rd.r64 (nome_lógico_arq, nome_dado, I, D),
rd.r32 (nome_lógico_arq, nome_dado, I, D) e
rd.text (nome_lógico_arq, nome_dado).

onde *nome_lógico_arq* deve ser **kbd**, se a leitura é de teclado. Os tipos dos dados endereçados por *nome_dado* são BYTE, INT, INT64, INT32, REAL64, REAL32 e [ ]BYTE, seguindo a ordem de apresentação acima. Os campos *tam*, *I* e *D* são inteiros e informam o número máximo de dígitos dos dados de tipo inteiro (*tam*) e o tamanho das partes inteira e decimal dos dados de tipo real (*I* e *D*)

III.6.4 Escrita

A execução de um comando de escrita equivale à escrita do conteúdo do dado indicado na instrução, no *buffer* do arquivo, na próxima posição disponível. A LI tem instruções específicas para vários tipos de escrita. Elas são:

wr.char (nome_lógico_arq, nome_dado),
wr.int (nome_lógico_arq, nome_dado, tam),
wr.i64 (nome_lógico_arq, nome_dado, tam),
wr.i32 (nome_lógico_arq, nome_dado, tam),
wr.r64 (nome_lógico_arq, nome_dado, I, D),
wr.r32 (nome_lógico_arq, nome_dado, I, D) e
wr.text (nome_lógico_arq, nome_dado).

onde *nome_lógico_arq* deve ser **scr**, se a escrita é na tela. Os tipos dos dados endereçados por *nome_dado* são BYTE, INT, INT64, INT32, REAL64, REAL32 e [ ]BYTE, seguindo a ordem de apresentação acima. Os campos *tam*, *I* e *D* são inteiros e informam o número máximo de dígitos dos dados de tipo inteiro (*tam*) e o tamanho das partes inteira e decimal dos dados de tipo real (*I* e *D*)

III.6.5 Modificação no *Buffer*

Quando se deseja efetivamente escrever (descarregar) o conteúdo do *buffer* no disco e ler (carregar) do disco um novo registro para o *buffer*, o seguinte comando deve ser usado:

newline (nome_lógico_arq),

Se o arquivo especificado por *nome_lógico_arq* tiver sido aberto no modo leitura, o seu *buffer* será recarregado com o próximo registro do arquivo. Se o arquivo foi aberto no modo escrita, o seu *buffer* será descarregado, ou seja o conteúdo será escrito no próximo registro do arquivo.

Essa instrução também pode ser usada para teclado e tela, substituindo *nome_lógico_arq* por **kbd** e **scr**, respectivamente.

III.7 Aspectos Importantes da Linguagem Intemediária

O conceito de **dado** e de grupo de **dados** (*array*) da LI é considerado exatamente o meio termo que se poderia ter entre ACTUS e OCCAM. Em ACTUS, as estruturas de dados podem ser ou não paralelas e é a sua declaração que define a forma de manipulação da estrutura. Em OCCAM, não existe a definição de uma estrutura de dados paralela. Mais precisamente, em OCCAM não existe esse tipo de paralelismo (SIMD) de ACTUS. A solução foi deslocar o conceito de paralelismo da declaração da estrutura para o modo de processamento da estrutura. Isso significa que na LI, a declaração de estrutura de dados é única, e a sua manipulação em paralelo será determinada pelas operações que forem feitas com a estrutura, que são, executar instruções paralelas ou escalares.

O conceito de extensão de paralelismo em ACTUS, fundamental à estrutura da linguagem, foi mantido na LI, sendo que, se em ACTUS ele denotava os elementos a serem manipulados, na LI, ele denota, na forma que segue, o endereço dos elementos a serem manipulados: o endereço do primeiro elemento, o espaçamento entre os elementos e a quantidade de elementos. Em termos práticos, isso significa aproximar um pouco o conceito de extensão de paralelismo de ACTUS para uma forma em que as informações serão normalmente utilizadas. Os vários tipos de endereçamento permitidos na LI, associados à extensão de paralelismo, permitem traduzir todas as situações de ACTUS que envolvem paralelismo, de uma forma relativamente simples. Esse assunto será detalhadamente abordado no próximo capítulo. Por último, o modo de uso das extensões de paralelismo da LI, que é através de ponteiros, mantém o código mais claro, permite economia no armazenamento do código e simplifica a geração de código pelo compilador.

Como se pode observar na leitura do capítulo, as operações das expressões escalares e paralelas são idênticas. A diferença é que as operações nas expressões paralelas são aplicadas a grupos de **dados** e nas expressões escalares são aplicadas a **dados** simples. As vantagens dessa característica da LI são facilitar a geração de código intermediário, pois o conjunto de mnemônicos é reduzido e as expressões estão padronizadas ao máximo, e facilitar o entendimento da semântica dos operadores nas expressões paralelas. A principal desvantagem dessas instruções é o seu formato de três endereços. Para expressões muito complexas, o código gerado pode se tornar muito grande e a administração da alocação de temporárias dificultar o trabalho do compilador. Por outro lado, esta foi a solução encontrada para adaptar os operadores de ACTUS (mais flexíveis) aos de OCCAM. Por exemplo, OCCAM não define prioridades entre os operadores e não permite operações entre variáveis de tipos diferentes. Sendo assim, optamos por esse formato, para as expressões, pois ele não exige grandes análises do tradutor e permite o uso de otimizadores.

A alocação de temporárias nas expressões pode ser eficientemente resolvida pela estrutura de blocos da LI, que permite reduzir o escopo das temporárias ao estritamente necessário. A LI, adotando o conceito de bloco de OCCAM, permite que sejam feitas declarações de dados em qualquer parte do código e que seja defi-

nido, através da instrução **block** o escopo daqueles dados, que pode ser de qualquer tamanho. Em termos práticos, isso significa que é possível alocar temporárias para a tradução de uma expressão complexa de ACTUS e se livrar dessas temporárias após a tradução.

Um outro aspecto interessante da LI é o tratamento dispensado às máscaras usadas nas expressões paralelas. As máscaras são grupos de **dados** comuns de tipo booleano, que são manipuladas pelas operações booleanas e de atribuição paralelas. Isso também simplifica o trabalho do compilador e tradutor, que trata da mesma forma os *arrays* booleanos, não importando o seu uso.

Nas estruturas de controle de fluxo de execução (*REPEAT, CASE, WHILE e IF*) temos uma das principais vantagens da LI. Em ACTUS, existe uma diferença semântica dessas instruções, se elas são usadas ou não com estruturas de dados paralelas. É como se existissem versões escalares e paralelas desses comandos. Para a LI, existe apenas uma versão de tais comandos, o que simplifica bastante o trabalho de geração de código intermediário. As instruções *REPEAT, WHILE, IF e CASE* são executadas sempre em função do valor de alguma variável booleana, ou seja, o tratamento das expressões booleanas de ACTUS, que gerariam o valor dessa variável booleana, é feito separadamente, conforme a sintaxe descrita nas seções anteriores. No caso da tradução da versão escalar desses comandos, o tratamento das expressões booleanas de ACTUS é feito com expressões escalares. No caso da tradução da versão paralela desses comandos, o tratamento das expressões booleanas com estruturas de dados paralelas de ACTUS é feito com expressões paralelas. Esses comandos de ACTUS em suas versões paralelas geralmente modificam a extensão de paralelismo usada nas estruturas de dados paralelas usadas no comando. Para a LI, a extensão de paralelismo não é modificada, apenas a LI permite que seja aplicada uma máscara á extensão. Então os comandos de ACTUS que usariam a extensão de paralelismo modificada são traduzidos para comandos da LI que usam uma extensão de paralelismo segundo uma máscara.

Capítulo IV

Tradução de ACTUS para a Linguagem Intermediária

IV.1 Introdução

Definido o conjunto de instruções da LI, serão apresentadas aqui sugestões para a tradução de ACTUS nessa linguagem intermediária. A apresentação seguirá a exposição da linguagem ACTUS, isto é, à medida que a linguagem for sendo apresentada serão sugeridas traduções. Não se pretende apresentar a semântica ou a sintaxe de ACTUS, além do necessário para a definição da tradução.

O símbolo ♯ nas instruções da LI, está sendo usado para indicar quando um campo da instrução não está sendo utilizado, por exemplo, nas expressões cujos operandos são monádicos.

IV.2 Tipos de Dados e Constantes

A tradução das declarações de ACTUS envolve três estruturas da LI: a tabela de *arrays*, onde são colocadas as informações sobre *arrays*; a tabela de edp's, onde são colocadas as informações sobre os conjuntos de índices; e a instrução DEC, que nas declarações de tipos primitivos (BOOL, INT, etc.), contém todas as informações sobre os dados, e nas declarações de *arrays* faz referências à tabela de *arrays*.

Nesta seção, serão apresentadas as traduções para os tipos de dados de ACTUS, que se dividem em simples e estruturados, e para constantes, que podem ser escalares ou paralelas.

Tipos Simples

ACTUS possui os seguintes tipos simples:

i) Tipos Primitivos

A tabela IV.1 apresenta os tipos primitivos de ACTUS e as suas traduções para a LI.

Tipo Primitivo em ACTUS	Tipo na LI
integer	INT
short integer	INT16
real	REAL64
short real	REAL32
byte	BYTE
boolean	BOOL
char	BYTE

Tabela IV.1: Tradução de Tipos Primitivos

A declaração em ACTUS de um dado de tipo primitivo é traduzida para:

```
DEC ( nome_dado, tipo )
```

onde *tipo* é um dos tipos de dados existentes na LI. A referência aos dados de tipo primitivo é feita simplesmente antepondo ao *nome_dado* o símbolo α. Por exemplo, em ACTUS, a declaração

```
var AA : integer
```

é traduzida para

```
DEC ( AA, INT )
```

e o dado *AA* é referenciado, na LI, como αAA.

ii) Tipos *Enumerated* e *Subrange*

A tradução desses tipos deve fazer uso das técnicas tradicionais de compilação, pois a LI não tem tipos equivalentes. Logo, eles devem ser traduzidos para os tipos primitivos da LI. Por exemplo, uma variável do tipo *subrange* pode ser traduzida para o tipo inteiro e, antes de seu uso, seus limites devem ser testados.

Tipos Estruturados

Os tipos estruturados de ACTUS são dois:

i) *Record*

Como no caso dos tipos *enumerated* e *subrange*, a tradução desses tipos é a decomposição deles em tipos da LI.

ii) *Array*

Os *arrays* em ACTUS podem ser **paralelos** ou **escalares**. A LI não faz distinção, a nível de declaração, entre esses tipos de *arrays*, mas faz distinção no seu modo de processamento. Os *arrays* podem ser processados escalarmente ou paralelamente. Em outras palavras, a declaração de *arrays* da LI é única:

```
DEC ( nome_dado, ↑ )
```

onde ↑ é o ponteiro do símbolo (*nome_dado*) em uma tabela de *arrays* gerada pelo compilador. Essa tabela contém informações sobre o tipo do *array*, número de dimensões, etc.

Os *arrays* escalares e paralelos de ACTUS devem ser manipulados com instruções escalares e paralelas, respectivamente, da LI, na forma como segue:

- *Arrays* Escalares
  Nas instruções que manipulam os *arrays* escalares, esses devem estar endereçados da seguinte forma:

  βnome_dado $\{_1$, (índice)$\}$

  É oportuno observar que os índices na LI sempre começam em **zero**, devendo o compilador efetuar transformações nos valores dos índices, se for necessário.

- *Arrays* Paralelos
  Os conjuntos de índices de ACTUS (*index set*) são usados para fazer referências a *arrays* paralelos. A tradução dessas referências depende dos valores dos conjuntos usados. Na seção que trata de conjunto de índices, é apresentada a tradução de referências a *arrays* paralelos.

Constantes

As constantes de ACTUS podem ser escalares ou paralelas, se elas representam um valor ou um conjunto de valores, respectivamente.

i) Constantes Escalares

As constantes devem ter o seu valor colocado diretamente no código em LI e devem ser precedidas pelo símbolo φ. Esse símbolo precede as constantes e os literais númericos ou alfabéticos. Por exemplo, em ACTUS, a declaração

```
const N = 100
```

não é traduzida e a expressão

```
AA := N
```

é traduzida para

```
EXP (copy, αAA, φ100, ♯) .
```

Já a expressão com literal númerico

```
AA := 0
```

tem sua tradução similar a tradução anterior:

```
EXP (copy, αAA, φ0, ♯).
```

ii) Constantes Paralelas

As constantes paralelas, assim como as constantes escalares, têm os seus valores colocados diretamente no código. Essas constantes são codificadas como os conjuntos de índices e referenciadas como tal. Na seção que trata dos conjuntos de índices, é apresentada a tradução de referências a constantes paralelas.

IV.3 Conjuntos de Índices e a Manipulação de Paralelismo

Em ACTUS, a manipulação de paralelismo é feita através dos conjuntos de índices (*index set*). Os conjuntos de índices dizem quais os elementos de um *array* que devem ser manipulados. Obviamente os conjuntos de índices são usados apenas com *arrays* paralelos, e as informações que os compõem são:

- elemento inicial ou limite inferior
- incremento (se não declarado, tem o valor 1)
- limite superior

Um conjunto de índices recebe valores, na sua declaração ou em um comando **using**, especificados na seguinte fórmula sintática:

limite inferior : [incremento] limite superior

Cada conjunto de índice definido em ACTUS gera uma entrada na tabela de edp's da representação intermedia'ria (LI) com os seguintes campos:

(elemento inicial, espaçamento, número de elementos)

Como será visto mais detalhadamente nas próximas seções, ACTUS e a LI têm uma visão diferente da manipulação dos conjuntos de índices. Em ACTUS, tem-se dois tipos de indexação de *arrays* paralelos:

1. indexação direta, que é feita diretamente através dos conjuntos de índices, que podem ser aleatórios ou regulares. Nesse tipo de indexação, um mesmo conjunto de índices pode ser usado para as duas extensões de paralelismo, no caso de um *array* paralelo bidimensional, por exemplo, as referências AA[IS, IS] e AA[IS shift 1, IS].

2. indexação independente, que é feita através de um *array* paralelo unidimensional. No caso de *arrays* bidimensionais, um mesmo conjunto de índices deve ser usado para indexar o *array* de índices e a outra dimensão paralela.

A LI, por sua vez, analisa os conjuntos de índices sob o ponto de vista de suas variações. Sendo assim, tem-se na LI:

1. variação independendente de índices, onde as extensões de paralelismo variam independentemente. Aqui estão incluídos os *arrays* paralelos unidimensionais de ACTUS, pois só possuem uma extensão de paralelismo, e os casos de indexação direta de *arrays* bidimensionais, onde são definidas duas extensões de paralelismo distintas.

2. variação dependente de índices, onde as extensões de paralelismo das duas dimensões paralelas variam dependentemente. Aqui, estão incluídos os *arrays* paralelos bidimensionais com indexação direta, se o mesmo conjunto de índices é usado nas duas dimensões, e os *arrays* paralelos com indexação independente.

IV.3.1 Manipulação de Paralelismo em ACTUS

ACTUS reconhece dois tipos de indexação: a indexação feita diretamente com conjuntos de índices, que será chamada **direta** e a indexação feita através de um *array* paralelo unidimensional, que é dita **independente**.

Indexação Direta

A indexação direta é feita com conjuntos de índices, cujos valores podem ter uma regra de formação (conjuntos de índices **regulares**) ou não (conjuntos de índices **aleatórios**). Os exemplos abaixo ilustram cada um desses tipos:

- Exemplos de conjunto de índices regular:

  Exemplo 1 :

```
using IS := 2:4, JS := 4:6 do
    AA [IS, JS] := 0;
```

  Nove elementos de AA (número de elementos expresso na primeira dimensão (IS) multiplicado pelo número de elementos expresso na segunda dimensão (JS)), receberão zero:

  AA[2,4]; AA[2,5]; AA[2,6]
  AA[3,4]; AA[3,5]; AA[3,6]
  AA[4,4]; AA[4,5]; AA[4,6]

Exemplo 2 :

```
using IS := 2:4 do
    AA[IS, IS] := 0;
```

Três elementos de AA receberão zero: AA[2,2], AA[3,3] e AA[4,4].

Exemplo 3 :

```
using IS := 2:4, JS := 3:5 do
      AA[IS, JS] := BB[IS shift 1, JS shift 1];
```

Os elementos

AA[2,3]; AA[2,4]; AA[2,5]
AA[3,3]; AA[3,4]; AA[3,5]
AA[4,3]; AA[4,4]; AA[4,5]

receberão os contéudos de

BB[3,4]; BB[3,5]; BB[3,6]
BB[4,4]; BB[4,5]; BB[4,6]
BB[5,4]; BB[5,5]; BB[5,5]

- Exemplos de conjunto de índices aleatórios:

Exemplo 4 :

```
using IS := 2:4, JS:= 2:6 - 3:4 do
      AA[IS, JS] := 0
```

O operador "-" no comando **using** faz uma diferença entre as extensões de paralelismo especificadas. Logo, o conjunto de índices JS denota os valores $2, 5$ e 6, ou seja, não tem uma regra de formação para os seus valores, por isso é chamado de aleatório. Nesse exemplo, nove elementos receberão zero:

AA[2,2]; AA[2,5]; AA[2,6]
AA[3,2]; AA[3,5]; AA[3,6]
AA[4,2]; AA[4,5]; AA[4,6]

Exemplo 5 :

```
using IS := 2:6 - 3:4 do
      AA[IS, IS] := 0;
```

Três elementos de AA receberão zero: AA[2,2], AA[5,5] e AA[6,6].

Indexação Independente

A indexação independente é feita com um *array* de índices unidimensional paralelo. O número de elementos expresso em cada dimensão deve ser igual. Por exemplo:

Exemplo 6 :

```
using IS := 1:50 do
      AA[ AI[IS], IS] := 0;
```

Nesse exemplo, cinqüenta elementos, cujos índices são informados no *array* AI, receberão zero.

IV.3.2 A Manipulação de Paralelismo na Linguagem Intermediária

Na LI, os elementos de um *array* são especificados através de seu tipo de endereçamento, nome e conjuntos de índices (um para cada dimensão paralela).

Sob o ponto de vista da LI, os endereçamentos são de dois tipos: no primeiro (γ), as dimensões têm os seus índices variando independentemente; no segundo (δ), a variação dos índices das dimensões são dependentes entre si.

Variação Independente de Índices (γ)

A variação independente de índices, representada na LI pelo símbolo γ precedendo o nome do *array*, é usado para a tradução de indexação direta de ACTUS, a qual dependerá do tipo de conjunto de índices, que pode ser aleatório ou regular:

i) Se o conjunto de índices for regular
Seu valores possuem regra de formação e a tradução das referências com esses tipos de índices é quase direta:

nome = γnome

e para cada dimensão paralela a tradução será:

valor inicial	=	limite inferior
espaçamento	=	incremento
número de elementos	=	((limite superior - limite inferior) / incremento) + 1

No caso do exemplo 1, a tradução da referência à variável *AA* seria:

γAA ($\Uparrow_0$, $\Uparrow_1$)

onde as duas extensões estão definidas na tabela de edp's com os seguintes valores:

ponteiro	elemento inicial	espaçamento	tamanho
$\Uparrow_0$	2	1	3
$\Uparrow_1$	4	1	3

ii) Se o conjunto de índices for aleatório

Seus valores não possuem uma regra de formação. Nesses casos, o compilador deve criar um vetor com esses valores e a tradução das referências com esses tipos de índices, é:

nome = γnome

e para a dimensão aleatória, tem-se:

valor inicial	$\Rightarrow$	nulo
espaçamento	$\Rightarrow$	nome do vetor de valores criado pelo compilador
número de elementos	$\Rightarrow$	tamanho do vetor de valores criado pelo compilador

No caso do exemplo 4, a tradução da referência à *AA* seria:

γAA ($\Uparrow_0$, $\Uparrow_1$)

onde as duas extensões estão definidas na tabela de edp's com os seguintes valores:

ponteiro	elemento inicial	espaçamento	tamanho
$\Uparrow_0$	2	1	3
$\Uparrow_1$	♯	CI	3

Nesse exemplo, *CI* é um *array* paralelo unidimensional, criado pelo compilador, cujos valores são os índices da segunda dimensão. Esse *array* tem três elementos de valores 2, 5 e 6.

Variação Dependente de Índices (δ)

A variação dependente de índices, representada na LI pelo símbolo δ precedendo o nome do *array*, pode ser aplicado à tradução de indexação independente ou direta de ACTUS:

i) Se indexação independente

Nesse caso, como explicado anteriormente, a indexação é feita através de um *array* paralelo unidimensional. A LI representa esse *array* como um conjunto de índices aleatórios, tendo o endereçamento com variação dependente de índices. Logo, a tradução é:

nome = δnome

E para a dimensão com indexação independente, tem-se:

valor inicial	$\Rightarrow$	nulo
espaçamento	$\Rightarrow$	nome do vetor de índices
número de elementos	$\Rightarrow$	tamanho do vetor de índices

No exemplo 6, a tradução da referência à *AA* seria:

δAA ($\Uparrow_0$, $\Uparrow_1$)

onde as duas extensões estão definidas na tabela de edp's com os seguintes valores:

ponteiro	elemento inicial	espaçamento	tamanho
$\Uparrow_0$	♯	AI	50
$\Uparrow_1$	1	1	50

ii) Se indexação direta

Nesse caso, a indexação é feita diretamente com os conjuntos de índices e a sua tradução é:

nome = δnome

e para cada dimensão paralela a tradução será:

valor inicial	=	limite inferior
espaçamento	=	incremento
número de elementos	=	((limite superior - limite inferior) / incremento) + 1

No exemplo 2, tem-se indexação direta com variação dependente de índices, e sua tradução seria:

δAA ($\Uparrow_0$, $\Uparrow_0$)

onde as duas extensões estão definidas na tabela de edp's com os seguintes valores:

ponteiro	elemento inicial	espaçamento	tamanho
$\Uparrow_0$	2	1	3

Nesse exemplo, apesar da matriz *AA* possuir duas dimensões paralelas, apenas uma extensão de paralelismo é utilizada.

No exemplo 3, tem-se um caso especial de variação dependente de índices, onde existe a variação dependente de índices nas duas extensões de paralelismo. Para identificar as relações de dependências necessárias à geração de código, uma informação adicional deve ser colocada na tabela de edp's, nos nas extensões de paralelismo dependentes. A tradução das referências é:

γAA ($\Uparrow_0$, $\Uparrow_1$) e
δBB ($\Uparrow_2$, $\Uparrow_3$)

onde as extensões estão definidas na tabela de edp's com os seguintes valores:

ponteiro	elemento inicial	espaçamento	tamanho	dependência
$\Uparrow_0$	2	1	3	
$\Uparrow_1$	3	1	3	
$\Uparrow_2$	3	1	3	0
$\Uparrow_3$	4	1	3	1

Resumindo, na LI, os *arrays* paralelos serão manipulados pelas instruções paralelas. Os três campos que codificam os elementos do *array* a serem manipulados são encontrados na **tabela de extensões de paralelismo**(edp). Na referência ao *array*, o que se tem é um ponteiro ($\Uparrow$) para a tabela, onde se encontram tais campos. Para cada dimensão paralela do *array*, tem-se um ponteiro para a tabela de edp's, onde se encontram os campos que traduzem a extensão de paralelismo para aquela dimensão.

A linguagem ACTUS permite que se faça operações sobre os conjuntos de índices (operações de alinhamento (*shift* e *rotate*) e operações de união, diferença e interseção). Um novo conjunto de índices é gerado após a operação e sendo assim ele deve ser colocado na tabela de edp's.

As constantes paralelas são traduzidas como se fossem conjuntos de índices, podendo como tais, ter valores com ou sem regra de formação. Por exemplo:

```
var
   PARALELO: array[0:99] of integer;
parconst
   SEQUENCIA = 1:50;
index
   CI: 0:[2]98;
⋮
using CI do
   PARALELO[CI] := SEQUENCIA
⋮
```

A constante paralela SEQUENCIA, cujo ponteiro é $\Uparrow_1$, e o conjunto de índices CI, cujo ponteiro é $\Uparrow_0$, serão codificados nos seguintes registros da tabela de edp's:

ponteiro	valor inicial	espaçamento	tamanho
$\Uparrow_0$	0	2	50
$\Uparrow_1$	1	1	50

A constante paralela deve ser referenciada no código através do símbolo ϵ, que identifica constantes paralelas, seguido do ponteiro para a tabela de edp's. Logo, a tradução do exemplo acima seria:

```
dec ( PARALELO, ↑ )
⋮
config( ⇑0, 1 )
expp (copy, γPARALELO[⇑0], ∈⇑1, ♯, ♯)
⋮
```

IV.4 Comandos

Em ACTUS, existem comandos escalares e comandos paralelos. Os primeiros são sintática e semanticamente equivalentes aos comandos da linguagem PASCAL. Os comandos paralelos são sintaticamente muito parecidos com PASCAL, mas totalmente diferentes quanto à semântica. A LI foi projetada de forma a unificar as traduções para as versões escalares e paralelas das construções ACTUS, tornando mais fácil o trabalho de geração de código e tornando o código mais uniforme.

IV.4.1 Atribuição

Os comandos de atribuição de ACTUS deverão ser traduzidos para instruções de três endereços da LI. Em outras palavras, comandos de atribuição serão desdobrados em grupos de instruções da LI, sempre que as expressões do lado direito do comando tenham mais de um operador.

O desdobramento de expressões complexas em expressões mais simples possibilita a aplicação de um otimizador ao código intermediário, se for o caso.

Se o comando envolve apenas variáveis escalares, a instrução (ou grupo de instruções) na LI é:

EXP (operação, operando.result, operando1, operando2)

Um exemplo de tradução de expressões escalares:

ACTUS		LI
var		dec(a,INT)
a,b : integer;		dec(b,INT)
c: real;	⟹	dec(c,REAL32)
⋮		dec(T1,REAL32)
b := a + c*3		dec(T2,INT)
		⋮
		exp(*, αT1, αc, φ3)
		exp(TRU, αT2, αT1, ♯)
		exp(+, αb, αa, αT2)

A operação TRU usada na representação intermediária, serve para fazer a conversão de tipos entre as variáveis T1 e T2. A LI não permite que os operandos de uma expressão tenham tipos diferentes e por isso fornece operadores para fazer conversões.

Se o comando envolve variáveis paralelas, a instrução (ou grupo de instruções) na LI é:

EXPP (operação, operando.result, operando1, operando2, máscara)

Nas expressões EXP e EXPP em LI, *operando.result*, *operando1*, *operando2* são endereços de um dado, ou endereço de um *array* em EXPP. As operações de ACTUS têm equivalentes na LI. A única exceção fica pela operação de **potenciação** que na LI é uma função. O campo *máscara* de EXPP pode ser ou não usado. Ele serve para promover uma seleção entre os elementos especificados pelos operandos. Em EXPP, bem como nas expressões em ACTUS, todos os operandos devem ter a mesma extensão de paralelismo. O exemplo abaixo mostra a tradução de uma expressão paralela:

Em ACTUS :

```
var
   p1, p2, total :  array[1:p] of integer;
index
   IS : integer;
⋮
using IS := 1:p do
   total := (p1[IS] + p2[IS]) * 1024;
```

Em LI:

```
dec(p1, ↑)
dec(p2, ↑)
dec(total, ↑)
⋮
config (⇑)
dec(t1, ↑)
expp (+, γt1[⇑], γp1[⇑], γp2[⇑], ♯)
expp (*, γtotal[⇑], γt1[⇑], φ1024, ♯)
```

onde ⇑ aponta para as seguintes informações na tabela de edp's:
(1 (valor inicial), 1 (incremento), p (número de elementos))

IV.4.2 Comando USING

A instrução *USING* de ACTUS engloba um bloco de comandos e especifica qual a extensão de paralelismo válida dentro daquele bloco. Com essas informações, o compilador pode gerar uma **diretiva de configuração** na LI, informando qual

a extensão de paralelismo necessária para executar um grupo de instruções. A instrução na linguagem intermediária é:

CONFIG ({$_1$, ⇑}, fim_bloco)
bloco_config

onde ⇑ aponta para a(s) extensão(ões) de paralelismo a ser(em) usada(s) na de execução dos comandos para qual o *CONFIG* é válido.

IV.4.3 Comandos Condicionais

O comando *IF*

O *IF* em ACTUS tem a seguinte sintaxe:

```
IF teste THEN bloco_then
         ELSE bloco_else
```

Para a tradução, se *teste* não é uma variável booleana, um conjunto de instruções – com uma ou mais instruções – deve ser gerado, antecedendo a geração de código do comando *IF*. Uma variável booleana deve ser criada para conter o resultado da avaliação da expressão de teste.

No *IF* escalar, apenas um dos blocos (*then* ou *else*) será executado. No *IF* paralelo, ambos os blocos podem ser executados. No *teste* desse tipo de *IF*, as instruções para o tratamento de *teste* não apenas atribuem um valor a uma variável booleana, mas também devem atribuir valores às máscaras. A máscara é testada, e se estiver vazia, o resultado é *FALSE*, caso contrário, é *TRUE*. Se *TRUE*, o *bloco_then* é executado. Após essa execução, um complemento à máscara deve ser feito, a máscara complementada será testada se está ou não vazia e novamente um valor será atribuido à variável booleana. Se *TRUE*, o *bloco_else* será executado.

Logo, tem-se que o *IF* escalar do ACTUS é traduzido quase que diretamente para a instrução *IF* da LI. O *IF* paralelo deverá ser traduzido para dois *IF's* da LI, uma vez que os dois blocos de comandos (*then* e *else*) podem vir a ser executados. No *IF* paralelo, os comandos paralelos seriam executados sob a máscara gerada no tratamento de *teste*.

A sintaxe do *IF* da linguagem intermediária é:

codigo_teste
IF (var_bool, fim_bloco_then, fim_bloco)
bloco_then
bloco_else

onde *fim_bloco* tem o número de instruções do *bloco_then* e *bloco_else*. No exemplo abaixo, é dada a tradução de um *IF* paralelo.

```
using IS:=1:50, JS:=1:50 do
      if AA[IS,JS] > 0 then
        BB[IS,JS] := BB[IS,JS] - 0.5
      else
        BB[IS,JS] := BB[IS,JS] + 0.5;
```

Nesse exemplo, os 250 elementos de **AA** denotados pelos conjuntos de índices **IS** e **JS** são comparados a zero. Os **índices** dos elementos de **AA** para os quais esse teste é verdadeiro serão usados para denotar quais os elementos de **BB** a serem operados em *then*. Os índices dos elementos de **AA** para os quais o teste é falso serão usados para denotar quais os elementos de **BB** a serem operados em *else*.

Na realidade, os comandos de *then* e *else* não são mutuamente exclusivos, não se podendo gerar para eles o mesmo código de um *if* escalar. O código gerado para o exemplo acima é:

```
dec(MAA, ↑)
dec(AA, ↑)
dec(BB, ↑)
dec(tb, BOOL)
block(9)
config ([⇑0,⇑1], 8)
expp (>, γMAA[⇑0,⇑1], γAA[⇑0,⇑1], ♯, ♯)
callfunc (ifany, αtb, 1)
cpar (γMAA[⇑0,⇑1])
if (αtb, 9, ♯, 9)
expp (−, γBB[⇑0,⇑1], γBB[⇑0,⇑1], φ0.5)
expp (not, γMAA[⇑0,⇑1], γMAA[⇑0,⇑1], ♯, ♯)
callfunc (ifany, αtb, 1)
cpar (γMAA[⇑0,⇑1])
if (αtb, 13, ♯, 13)
expp (+, γBB[⇑0,⇑1], γBB[⇑0,⇑1], φ0.5)
```

onde $\Uparrow_0$ e $\Uparrow_1$ apontam para as seguintes informações :

(1 (valor inicial), 1 (incremento), 50 (número de elementos)) e
(1 (valor inicial), 1 (incremento), 50 (número de elementos)),

respectivamente, na tabela de edp's.

O comando *CASE*

O formato do *CASE* em ACTUS é:

```
CASE var_selecao OF
   case_lista_valores_1 : bloco_opcao_1
   case_lista_valores_2 : bloco_opcao_2
   ⋮
   case_lista_valores_n : bloco_opcao_n
END;
```

Se o *CASE* é escalar, o valor de *var_selecao* é comparado aos valores de *case-lista-valores* e apenas um *bloco_opção* é executado. Os valores em *case-lista-valores* podem ser discretos (valor_1, valor_2, ···, valor_n) ou contínuos (valor_1 ··· valor_2). Esse *CASE* pode ser traduzido para o *CASE* da linguagem intermediária, cuja sintaxe é:

```
CASE ( var_selecao, fim_bloco )
{ 1, OPTION ( VD/VC, { 1, valor}
bloco_opcao }
```

Por exemplo, o trecho de programa

```
⋮
case A of
    58..79 :  X := A + X;
    80, 97, 138 :  X := A + Y;
end
⋮
```

é traduzido para

```
⋮
case (A, 4)
option (VC, 58, 79)
exp (+, αX, αA, αX);
option (VD, 80, 97, 138)
exp (+, αX, αA, αX);
⋮
```

Se o *CASE* é paralelo, cada *bloco_opcao* pode vir a ser executado. O valor de *var_selecao* é comparado a cada *case-lista-valores*, e uma máscara é gerada. Se a máscara tem algum elemento *TRUE*, o *bloco_opcao* correspondente é executado. O *CASE* paralelo deve ser traduzido para tantos *IF's* da linguagem intermediária quantos forem os *case-lista-valores*.

É importante lembrar que como a variável *var_seleção* pode ser alterada na execução dos *bloco_opcao*, o compilador deve gerar uma variável temporária para preservar os seus valores e gerar os *IF's* testando essa variável.

IV.4.4 Comandos de Repetição

O Comando *REPEAT*

O formato geral do *REPEAT* em ACTUS é:

REPEAT bloco_repeat UNTIL teste;

A tradução para a linguagem intermediária é praticamente direta:

REPEAT (var_bool, fim_bloco)
bloco_repeat
codigo_teste

onde *fim_bloco* contém o número de instruções do *bloco_repeat* e *codigo_teste*. *codigo_teste* contém as instruções para tratamento de *teste*. A última dessas instruções deve fazer uma atribuição a uma variável booleana cujo endereço é posto em *var_bool*. Por exemplo, o trecho de programa

```
using CI := 1:100 do
   repeat
      BB[CI] := AA[CI];
      AA[CI] := AA[CI] + 2;
   until any(AA[CI] < 0);
```

pode ser traduzido para a seguinte seqüência de comandos em LI:

```
config(⇑, 5)
repeat(tb, 4)
expp(copy, γBB[⇑], γAA[⇑], #, #)
expp(+, γAA[⇑], γAA[⇑], α2, #)
expp(<, γMAA[⇑], γAA[⇑], #, #)
```

```
callfunc (ifany, αBB[⇑], 1)
cpar (γAA[⇑])
```

onde ⇑ aponta para as seguintes informações

(1 (valor inicial), 1 (incremento), 100 (número de elementos))

na tabela de edp's.

O Comando *WHILE*

O formato geral do *WHILE* em ACTUS é:

WHILE teste DO bloco_while ;

A semântica do comando *WHILE* seqüencial é a usual. Se o *WHILE* é paralelo, a primeira execução de *teste* será feita com todos os elementos especificados na extensão de paralelismo. As instruções de tratamento de *teste* geram uma máscara para os elementos da extensão de paralelismo para os quais *teste* é *TRUE*. As instruções de *bloco_while* serão executadas segundo essa máscara. A próxima execução de *teste* será feita com os elementos que tiveram o valor *TRUE* no teste anterior; isso significa executar *teste* com a máscara gerada pelo *teste* anterior, e assim sucessivamente. O *bloco_while* será executado enquanto houver elementos *TRUE* na máscara. Devido ao primeiro teste, a máscara deve ser inicializada com *TRUE* antes de sua execução.

A sintaxe do *WHILE* na linguagem intermediária é:

```
WHILE ( var_bool, fim_cod_teste, fim_bloco )
codigo_teste
bloco_while
```

onde *fim_bloco* contém o número de instruções de *codigo_teste* e *bloco_while*.

O exemplo abaixo ilustra a tradução da versão paralela do *while*.

```
var
    AA : array[1:5,1:5] of integer;
index
    IS1,IS2 :  1:5;
Begin
    using IS1,IS2 do
          while AA[IS1,IS2] > 0 do
          AA[IS1,IS2] := AA[IS1,IS2] - 4;
End;
```

Nesse exemplo, o *array* **AA** será testado, e os elementos que forem maiores que zero serão subtraídos de 4. Apenas esses elementos serão testados na segunda interação do *loop*, e assim sucessivamente, até que nenhum dos elementos testados seja maior que zero. A nível de LI, isso significa gerar máscaras sucessivas de **AA** e executar a expressão aritmética paralela sob essas máscaras. A LI a ser gerada é:

```
dec(AA, ↑)
dec(MAA, ↑)
dec(tb, BOOL)
block(6)
config([⇑0, ⇑1], 5)
expp(copy, γMAA[⇑0, ⇑1], TRUE, ♯, ♯)
while(tb, 2, 3)
expp(>, γMAA[⇑0, ⇑1] , γAA[⇑0, ⇑1] , ♯, γMAA[⇑0, ⇑1])
callfunc (ifany, αtb, 1)
cpar (γMAA[⇑0, ⇑1])
expp(−, γAA[⇑0, ⇑1], γAA[⇑0, ⇑1], φ4, γMAA[⇑0, ⇑1])
```

onde $\Uparrow_0$ e $\Uparrow_1$ apontam para as seguintes informações :

(1 (valor inicial), 1 (incremento), 5 (número de elementos)) e
(1 (valor inicial), 1 (incremento), 5 (número de elementos)),

respectivamente, na tabela de edp's.

O Comando *FOR*

O formato geral do *FOR* em ACTUS é:

FOR var_controle := val_inicial
BY incremento
TO/DOWNTO val_final DO
bloco_for;

A tradução para a LI é dada por:

FOR (num_vezes, fim_bloco)
bloco_for

calculando-se

$$num\_vezes = \frac{val\_final - val\_inicial + incremento}{incremento} \qquad \text{(IV.1)}$$

onde *num_vezes* é o número de vezes que serão executados os comandos do *bloco_for*.

Desvio Incondicional

A sintaxe do *GOTO* em ACTUS é:

GOTO nome_label

A colocação de um *nome_label* em um comando exige sua declaração prévia.

Na LI não existe declaração para *nome_label*. A tradução de *GOTO* é bastante simples. Ao se deparar com um *nome_label*, o compilador deve gerar a seguinte intrução:

LABEL (nome_label)

onde *nome_label* é um literal. Ao se deparar com o comando *GOTO*, a seguinte instrução deve ser gerada:

GOTO (nome_label)

IV.4.5 Subprogramas – Função

As funções em ACTUS podem ser traduzidas em funções da linguagem intermediária, desde que sejam obedecidas as restrições impostas pela LI, mencionadas na seção III.5.5. Em outras palavras, funções de ACTUS que manipulem variáveis paralelas, ou que sejam recursivas ou que façam alguma entrada/saída, não podem ser traduzidas para funções da LI. A declaração da função deve ser traduzida para:

FUNCTION (nome_subprograma, tipo, fim_par, fim_dec, fim_bloco)
{ 1, PAR (nome_dado, tipo, VAL) }
{ 0, DEC (nome_dado, tipo) }
bloco_funcao

A chamada à função deve ser traduzida para:

CALLFUNCTION (nome_dado_resultado, nome_subprograma, fim_bloco
)
{ 1, CPAR (nome_dado) }

Quando não for possível a tradução da função de ACTUS devido às restrições existentes em *FUNCTION* da linguagem intermediária, a primeira deve ser transformada em *procedure*. O formato geral da função em ACTUS é:

```
FUNCTION nome_subprograma ( parametros formais ) :  tipo ;
(* identificadores *)
  Begin
     bloco_funcao
     nome_subprograma := resultado
  End;
⋮
⋮
nome_dado_resultado := nome_funcao ( argumentos )
⋮
```

Se transformado, o procedimento seria:

```
PROCEDURE nome_subprograma ( nome_dado_resultado,
parametros formais ) :  tipo ;
(* identificadores *)
  Begin
     bloco_funcao
     nome_dado_resultado := resultado
  End;
⋮
⋮
nome_subprograma ( nome_dado_resultado, argumentos )
⋮
```

As funções padrão de ACTUS são fornecidas pela LI na biblioteca de rotinas. A tradução da chamada à função padrão é igual à tradução da chamada à função definida pelo programador.

IV.4.6 Subprogramas – Procedimentos

A declaração de procedimento deve ser traduzida para:

```
PROC ( nome_subprograma, recursividade, fim_par, fim_bloco )
{ 0, PAR ( nome_dado, tipo, VAL/REF ) }
{ 0, DEC ( nome_dado, tipo }
bloco_procedure
```

Se o procedimento for recursivo, o campo *recursividade* terá o valor *TRUE*, caso contrário, terá o valor *FALSE*.

A chamada do procedimento deve ser traduzida para:

CALLPROC (nome_subrotina, recursive, fim_bloco)
{ 0, CPAR (nome_dado) }

Se o procedimento é recursivo, a **chamada recursiva** deve ter o campo *recursive* com o valor *TRUE*.

IV.5 Manipulação de Arquivos

Nessa versão do compilador, ACTUS manipula arquivos como PASCAL manipula, ou seja, os comandos de ACTUS são sintatica e semanticamente semelhantes aos comandos de PASCAL. Na LI, os arquivos também são tratados de maneira semelhante à PASCAL, e as instruções de ACTUS têm um mapeamento praticamente direto nas instruções da LI, sendo que as últimas são de mais baixo nível, tendo instruções específicas para cada tipo de dado, e não oferecendo instruções compostas (p.e. READLN, que lê uma variável e incrementa o apontador de arquivo).

Além dessas instruções, as instruções de ACTUS que consultam o *status* de um arquivo, têm instruções correspondentes na LI.

IV.6 Escopo de Estruturas de Dados e Subrotinas

Como as linguagens ACTUS e OCCAM têm as mesmas regras de escopo e a LI é específica da linguagem ACTUS, a LI tem as mesmas regras de escopo de ambas as linguagens, poupando o compilador do trabalho adicional de tradução de referências não locais. No capítulo V, esse assunto será tratado com mais detalhes.

Capítulo V

Geração de Código OCCAM

V.1 Introdução

Este capítulo tem por finalidade mostrar como o código em LI foi traduzido para código OCCAM. Uma vez que a LI é específica da linguagem ACTUS, alguns comandos de ACTUS têm tradução quase direta para a LI, deixando para o gerador de código o trabalho de traduzí-los para OCCAM. Nesses casos, mencionaremos o mapeamento ACTUS em OCCAM. Mas, existem comandos de ACTUS, cujo trabalho de tradução ficou dividido entre os processos de compilação e geração de código. Esses comandos serão mostrados em termos da tradução de LI em OCCAM.

Neste capítulo, os testes realizados para validação do gerador de código serão descritos, e tecidas algumas considerações a respeito da qualidade de código gerado.

V.2 Mapeamento de ACTUS em OCCAM

V.2.1 Regras de Escopo

O nosso compilador ACTUS, ao contrário dos compiladores tradicionais, terá facilidades na tradução de declarações de estruturas de dados e procedimentos e no controle do uso dessas declarações, devido ao fato de as regras de escopo de ACTUS e OCCAM serem as mesmas. Obviamente, a LI manteve as mesmas regras. Nesta seção, as regras de escopo de ambas as linguagens serão apresentadas e comparadas. As regras de escopo aqui mencionadas para ACTUS são iguais às regras de PASCAL [22]. As regras de OCCAM foram retiradas de [9].

ACTUS é uma linguagem estruturada em blocos na qual as regras de escopo estático são usadas para determinar o significado de referências não locais a nomes. As regras de escopo estático associadas com os programas estruturados em blocos são como se segue:

(1) As declarações no topo do bloco definem as referências locais do bloco. Qualquer referência para um identificador dentro do corpo do bloco (não incluindo qualquer bloco aninhado) é considerada uma referência para a declaração local do identificador, se ela existe.

(2) Se o identificador é referenciado dentro do corpo do bloco e nenhuma declaração local existe, a referência é procurada nas declarações locais do bloco ao qual aquele bloco pertence e assim sucessivamente até achá-la, ou concluir que ela não existe.

(3) Se um bloco contém outro, as declarações do bloco interior são invisíveis para o bloco que o contém.

(4) Um bloco pode receber uma identificação. Essa identificação tornar-se parte do ambiente de referências locais do bloco que o engloba.

Um programa em OCCAM é chamado de processo. Sejam as seguintes definições:

(1) processo = SKIP | STOP
| ação
| construção
| bloco
| instância

(2) bloco = especificação :
escopo

(3) especificação = declaração
| definição

(4) escopo = processo

(5) definição = PROC nome ({, formal})
corpo

(6) corpo = processo

(7) instância = nome ({, argumento})

(8) Sejam os nomes X e Y. Sejam os processos similares P(X) e P(Y), exceto que, P(X) contém X sempre que P(Y) contém Y e vice-versa. Sejam S(X) e S(Y) especificações similares, exceto que, S(X) define X e S(Y) define Y. Então:

S(X):	=	S(Y):
P(X)		P(Y)

Com essa regra é possível expressar o processo em uma forma canônica onde nenhum nome é especificado mais que uma vez.

Pelas definições de OCCAM relacionadas acima, tem-se que:

1. Com a definição (2) de OCCAM, as declarações no topo do bloco são as declarações locais ao bloco, cujo escopo é o processo que o compõe. Qualquer referência para um identificador dentro do corpo do bloco é considerada uma referência local, se a declaração existe. Se assim não fosse, a regra (8) não poderia ser aplicada ao bloco.

2. Se o identificador é referenciado dentro do corpo do bloco e nenhuma declaração local existe, a referência é procurada nas declarações locais do bloco que o engloba e assim sucessivamente., pois pela definição (2) de OCCAM, as declarações no topo do bloco são válidas para todo o corpo do bloco, inclusive para os blocos contidos nele (pelas definições (4) e (1) é possível ter ninhos de blocos).

3. Se as declarações de um bloco não fossem invisíveis para o bloco que o contém, a regra (8) de OCCAM não poderia ser aplicada.

4. Pelas definições (3), (5), (6) e (7), um bloco pode ser chamado e torna-se parte do ambiente de referências locais do bloco que o engloba.

Pode-se concluir então, que as regras de escopo de ACTUS e OCCAM são equivalentes. Isso significa que o compilador pode gerar código em LI para declarações de procedimentos e dados, na mesma ordem em que ele as encontra no programa ACTUS. Obviamente, ele deve fazer *check* de escopo, mas a geração de código é simplificada.

V.2.2 Declarações

Na LI, as declarações são de dados de tipos primitivos ou de *arrays*. No primeiro caso, a tradução é fácil, pois existe uma correspondência direta entre os tipos primitivos da LI e de OCCAM. No segundo caso, o gerador de código usa a tabela de *arrays* gerada pelo compilador para gerar a declaração de *array* em OCCAM. No Apêndice B, na subrotina **s242**, tem-se exemplos de tradução de declarações. No Capítulo IV, é demostrado como se podem traduzir declarações de ACTUS para a LI.

V.2.3 Comandos de Controle de Fluxo de Execução

Em ACTUS, a semântica dos comandos WHILE, REPEAT e FOR é diferente nas versões escalares e paralelas. Na LI, esses comandos têm suas versões unificadas, ou seja, as instruções da LI usadas pelo compilador para a tradução são as mesmas para ambas as versões. A distinção entre os comandos paralelos e seqüenciais reside apenas nas expressões paralelas e seqüenciais dos comandos WHILE, REPEAT e FOR. Logo, o gerador de código tem um único algoritmo de geração de código para esses comandos. No Apêndice B, as subrotinas **s242**, **s481** e **s482** têm exemplos

dos comandos FOR, WHILE e REPEAT, respectivamente, nas suas traduções de ACTUS para a LI e depois para OCCAM.

V.2.4 Comandos Condicionais

Em ACTUS, assim como nos comandos de controle de fluxo de execução, a semântica dos comandos IF e CASE é diferente para as versões paralelas e escalares desses comandos.

Mas, novamente, na LI, o comando IF tem suas versões unificadas, ou seja, as instruções da LI usadas pelo compilador para a tradução são as mesmas para ambas as versões. A distinção entre os comandos paralelos e sequüenciais reside apenas nas expressões paralelas e seqüenciais do comando IF. O IF de OCCAM, pode ter várias expressões booleanas associadas à varios blocos de comandos. Na ordem em que são definidas, essas expressões são avaliadas, e a primeira que resultar *TRUE*, terá o bloco de comandos executado. O Gerador de código traduz o IF da LI diretamente para o IF de OCCAM.

O código gerado pelo compilador para o comando CASE na sua versão escalar é diferente do código gerado para a sua versão paralela. O CASE escalar é mapeado diretamente no CASE da LI. O gerador de código traduz o CASE da LI para o IF com várias expressões booleanas de OCCAM. O CASE paralelo é mapeado em uma seqüência de IF's da LI, que são traduzidos conforme mencionado acima. No Apêndice B, as subrotinas **s279** e **s442** têm exemplos dos comandos IF e CASE, respectivamente, nas suas traduções de ACTUS para LI e depois para OCCAM.

V.2.5 Recursividade

Em ACTUS, podem-se fazer chamadas recursivas de procedimentos. Em OCCAM, isso não é possível. A LI, sendo específica da linguagem, conserva essa facilidade de ACTUS, deixando para o gerador de código a tarefa de contornar essa diferença.

Várias soluções foram examinadas em baixo nível, tentando driblar OCCAM no que diz respeito a alocação e ativação dinâmica de processos, usando a linguagem de montagem do *transputer* [23]. A solução adotada usa os recursos de OCCAM, para simular a ativação dinâmica de processos.

Cada chamada recursiva a um procedimento significa a ativação de um processo, sem que o processo ativado anteriormente tenha acabado. Isso significa que, em um dado instante, tem-se vários processos ativos. Em OCCAM, a única construção que permite a existência de vários processos ativos em um dado momento é a construção **PAR**, que cria processos que serão executados em paralelo. A idéia é criar através dessa construção, um número de processos qualquer (esse número diz a profundidade possível de chamadas recursivas e pode ser aumentado através de uma diretiva), que serão ativados ao início da execução da construção, mas em que

a execução do código de cada processo só começará, depois que cada um receber um sinal de um dos outros processos. Esse recurso faz com que seja simulada a execução seqüencial das chamadas recursivas, como em ACTUS. O código do procedimento recursivo é o código a ser executado na construção **PAR** e o número de replicações (ver Capítulo II) é a profundidade da recursão. Quando é iniciado o procedimento de retorno, os processos não "usados" recebem uma ordem de **fim**. Uma descrição do código gerado para os procedimentos recursivos é dado abaixo:

```
PROC recursiva (novas definições de parâmetros formais
     iguais aos do programa ACTUS II)
  -- definição de parâmetros de ativação
  --    iguais aos parâmetros formais.
  -- definição de um protocolo, cujos tipos são iguais e têm
  --    mesma ordem dos tipos dos parâmetros formais.
  -- definição dos canais início e retorno,
  --    cujo protocolo foi definido acima.
  -- definição de um array booleano fim
  --    cuja tamanho é igual à profundidade
  --    possível da recursão.
  SEQ
    -- inicialização dos parâmetros de ativação com
    --com os valores dos parâmetros formais.
    -- inicialização do array booleano fim com
    --valor FALSE.
  PAR
    retorno[0] ?  parâmetros formais
    início[1] !  parâmetros de ativação
    PAR i=1 FOR profundidade
      -- definição dos parâmetros formais verdadeiros,
      --ou seja, os parâmetros formais de ACTUS II.
      ALT
        fim[i] & SKIP
          fim[i+1] := TRUE
        início[i] ?  parâmetros formais verdadeiros
          -- todo o código do procedimento a partir das
          -- declarações.
          retorno[i-1] !  parâmetros formais verdadeiros
```

No código do procedimento, a chamada recursiva deve ser substituída pelo seguinte código:

```
SEQ
  início[i+1] !  parâmetros da chamada recursiva
  retorno[i] ?  parâmetros da chamada recursiva
```

V.2.6 Desvio Incondicional

Em ACTUS e na LI, diferentemente de OCCAM, existe o comando de desvio incondicional (**goto**) para algum ponto do programa identificado por um **label**. O gerador de código recorre à linguagem de montagem do *transputer*, inserindo código nessa linguagem, no programa objeto, para execução desses desvios.

O uso de **goto** em ACTUS tem restrições. Não se pode dar um **goto** para comandos pertecentes a um bloco **using**, por exemplo. Consideremos o seguinte trecho de um programa em ACTUS:

```
using CI := 1:100 do
   100:  AA[CI] := 0;
```

Se fosse possível um desvio para o **label** 100, o conjunto de índices CI não seria inicializado. Em OCCAM, o uso de **goto** é também bastante dificultado: só se pode usá-lo em código *assembly* e tem-se que assumir o risco dos eventuais problemas que surgirem, pois o compilador OCCAM é liberado de fazer alguns *checks*, quando se tem código *assembly* no programa. O desvio em OCCAM não causa problemas se for feito para algum comando do mesmo processo seqüencial. As restrições em ACTUS garantem que em OCCAM não se tenha desvios para processos paralelos – não existe desvio para dentro das construções **using**, que é a única construção de ACTUS que gera processos paralelos em OCCAM.

V.2.7 Manipulação de Paralelismo

As estruturas de dados paralelas de ACTUS não encontram em OCCAM uma tradução direta. Na realidade, o que fizemos foi manter no programa em LI e nas tabelas que o acompanham, todas as informações sobre as estruturas, e no caso de constantes e conjunto de índices, os valores que eles representam. Quando o gerador de código é aplicado à representação intermediária, ele utiliza essas informações para gerar código apropriado para a execução das instruções que fazem uso dessas estruturas. Sendo assim, para as estruturas de dados apresentadas a seguir, apresenta-se sua codificação na LI, e quando falarmos de comandos paralelos, será explicado o uso dessas informações.

Arrays

Em ACTUS, os *arrays* são declarados paralelos ou seqüenciais. Na LI e em OCCAM não existe distinção entre ambas as declarações, e a tradução é feita facilmente usando-se os dados disponíveis na **tabela de** *arrays* gerada pelo compilador, na tradução de um programa em ACTUS para a LI.

A tabela de *arrays* possui a informação de tipo, número de dimensões e valor de cada dimensão do *array*. Com essas informações é possível gerar uma

declaração de *array* em OCCAM. As informações de quais dimensões são paralelas, também presentes na tabela, serão utilizadas na geração de código para as expressões paralelas que usam o *array*.

Conjunto de Índices

Cada conjunto de índices equivale a um elemento em uma tabela chamada de *tabela de edp's* (tabela de extensões de paralelismo). Cada elemento da tabela possui três campos: valor base, incremento e tamanho. Um conjunto de índices explícitos possui informações para uma tradução quase que direta para a tabela de edp's. A tradução de um conjunto de índices redefinível, envolve a criação de três variáveis e em vez de valores na tabela de edp's, têm-se essas variáveis, que devem ser inicializadas quando o compilador toma conhecimento dos valores que serão atribuidos ao conjunto de índices (os conjunto de índices redefíniveis são inicializados nos comandos **using**).

No programa OCCAM não existe declaração de conjunto de índices; as informações da tabela de edp's são usadas na tradução dos comandos paralelos, ou seja, comandos que fazem referências a *arrays* paralelos. As subrotinas **s111**, **s425** e **s4113**, do Apêndice B, mostram exemplos de conjuntos de índices explícitos, redefíniveis e de indexação independente.

Constantes Paralelas

As constantes paralelas são traduzidas como os conjuntos de índices, ou seja, cada constante paralela equivale a um elemento da tabela de edp's. Igualmente aos conjuntos de índices, a representação das constantes na tabela de edp's dependerá de seus valores. Conforme eles tenham ou não regra de formação, serão codificados nos campos da tabela ou serão armazenados em um vetor de índices, respectivamente.

O programa objeto em OCCAM, aloca área para um vetor e o inicializa com os valores regulares da constante paralela; é esse vetor que substituirá a constante nas expressões aritméticas paralelas.

Expressões Paralelas e o Comando CONFIG

Nesta seção, será apresentada as traduções das expressões paralelas (**expp**) e das diretivas de configuração (**config**).

A linguagem ACTUS permite o uso de até duas dimensões paralelas. A nossa máquina alvo consiste de um nó de uma rede de *transputers* com um co-processador vetorial. Assim sendo, só podemos executar operações em paralelo sobre vetores. A estratégia usada para execução dos comandos paralelos, consistiu, então, de escolher, se duas dimensões paralelas, a dimensão com maior extensão de paralelismo e fazer um *loop* com os dados da outra dimensão, fazendo operações paralelas sobre a dimensão de maior extensão de paralelismo, a cada iteração do *loop*.

Na geração de código para as expressões, são usadas as construções SEQ e PAR (com replicadores) de OCCAM. A variável que controla a replicação poderia ser um indexador natural para os *arrays*, se fosse permitido incrementá-las com valores diferentes de uma unidade. Entretanto, é possível definir qualquer valor de espaçamento entre os valores dos conjuntos de índices. A solução adotada foi gerar, para as dimensões de maior extensão de paralelismo, um vetor de índices (*gc.cilarge*), que guarda os valores da extensão (todas as variáveis criadas pelo gerador de código têm o prefixo *gc.*).

Outras variáveis criadas pelo gerador de código são *gc.const*, que guarda os valores de uma constante paralela, e **gc.cidep**, que guarda os valores de uma extensão de paralelismo que deve variar dependendo de outra (esse é o caso, em ACTUS, de uma referência como aquela apresentada no exemplo 3, na seção IV.3).

No código gerado para expressões que manipulam duas extensões paralelas, o *loop* externo, que controla a menor extensão de paralelismo, a variável de controle do replicador é nomeada *i*. No *loop* interno, a variável de controle é nomeada *j*.

Ao encontrar uma ordem de configuração (comando **config**), o gerador de código analisa os conjuntos de índices do **config** e:

1. se é utilizado apenas um conjunto de índices, a dimensão que usar esse conjunto de índices será a dimensão a ser executada em paralelo.

2. se são utilizados dois conjuntos de índices, eles devem ser comparados. Se há algum conjunto de índices com valores aleatórios, ou um conjunto de índices cuja extensão de paralelismo só seja conhecida em tempo de execução, a dimensão que usa esse conjunto de índices, será a dimensão a ser executada escalarmente. Se os dois conjuntos de índices são regulares e têm as extensões de paralelismo conhecidas, a dimensão que usar o conjunto de índices de menor extensão, será a dimensão executada escalarmente.

3. se a execução do programa é no modo escalar e se o conjunto de índices de maior extensão tem valores regulares, gerar um vetor com os seus índices (*gc.cilarge*).

A configuração válida para uma expressão paralela, é aquela especificada na instrução *config* que a engloba. Logo, quando uma expressão paralela é encontrada, são feitos os seguintes procedimentos:

1. análise dos operandos. Se há alguma constante paralela (ϵ), um vetor é alocado (*gc.const*), e seus elementos são inicializados com os valores da constante.

   Se há algum operando cujos índices variem dependentemente (δ) e não seja o caso de indexação independente, os conjuntos de índices são analisados, e, se necessário, um vetor é alocado (*gc.cidep*). Os elementos de *cidep* são inicializados com os valores do conjunto de índices. Se as extensões de paralelismo do operando são iguais, (esse é o caso, em ACTUS, de uma referência como

aquela apresentada no exemplo 2, na seção IV.3) não é necessária a alocação do vetor *cidep*.

2. se a expressão tem apenas uma extensão de paralelismo e o modo de execução do programa é escalar, um *loop* (SEQ) é gerado em OCCAM (a replicação é controlada por j).

3. se a expressão tem duas extensões de paralelismo, um *loop* (SEQ) de OCCAM é gerado com os dados da menor extensão. Nesse *loop*, se a extensão de paralismo é regular, uma variável *gc.small* é criada pelo gerador de código, e vai recebendo os valores dos índices da extensão, a cada iteração.

   Se o modo de execução do programa é escalar, um outro *loop* (SEQ) é gerado com os dados da maior extensão de paralelismo.

4. se a expressão deve ser executada segundo uma máscara e a execução é escalar, um teste de máscara é gerado.

5. o código da expressão é gerado dependendo da execução ser escalar ou paralela. Se escalar, para cada operando, tem-se:

   - Se o operando é escalar (φ, α ou β), o código é gerado normalmente.
   - Se o operando é paralelo com endereçamento γ, A dimensão da menor extensão de paralelismo é substituída por *gc.small*, se esta é regular. Caso contrário, a dimensão é substituída pelo vetor indicado na tabela de edp's, indexado por i.

     A dimensão da maior extensão de paralelismo é substituída pelo vetor *gc.cilarge*, indexado por j.
   - Se o operando é paralelo com endereçamento δ, os procedimentos para as dimensões de menor e maior extensão de paralelismo são iguais àqueles descritos no ítem anterior. Para as dimensões diferentes destas, tem-se que examinar as suas relações de dependências. Se a dimensão depende da menor extensão, o vetor *gc.cidep* indexado por i, deve ser usado. Caso contrário, se a dimensão varia com a maior dimensão, o vetor *gc.cidep* indexado por j, deve ser usado.
   - Se o operando é uma constante paralela (ϵ), ele deve ser substituído pelo vetor *gc.const* indexado por j.

No Apêndice B, são mostrados vários exemplos de expressões paralelas e as suas traduções para LI e OCCAM.

V.2.8 Subrotinas - Procedimentos e Funções

A declaração e uso de **procedimentos** OCCAM é semanticamente igual a ACTUS. Em ACTUS, os parâmetros de um procedimento podem ser escalares ou paralelos. No caso de parâmetros escalares, a tradução do procedimento para a LI e OCCAM é quase direta. No caso de parâmetros paralelos, deve-se acrescentar, para cada

parâmetro paralelo, um parâmetro que representa o conjunto de índices associado à variavel. Logo, na chamada ao procedimento, o conjunto de índices que expressa a extensão de paralelismo corrente de cada parâmetro paralelo deve ser um argumento.

A extensão de paralelismo da variável paralela deve, como em todos os outros casos, estar contida na tabela de edp's. Assim sendo, o compilador deve:

1. Alocar um elemento na tabela de edp's para conter as informações a respeito do conjunto de índices parâmetro. Como essas informações só serão conhecidas em cada chamada ao procedimento, os campos do conjunto na tabela devem conter nomes de variáveis.

2. Aumentar a lista de parâmetros formais do procedimento com mais três parâmetros, cujos nomes são os das variáveis da tabela de edp's, que representam o conjunto de índices.

3. Na tradução da chamada do procedimento, passar valores sobre a extensão de paralelismo corrente para os três novos parâmetros.

As subrotinas **s151** e **152** têm exemplos da tradução de procedimentos, cujos parâmetros são variáveis paralelas.

Uma questão em aberto sobre procedimentos, é a possibilidade de se ter uma chamada a um procedimento com variáveis paralelas dentro de comando do tipo **if** e **while**, em que a extensão de paralelismo varia dinamicamente.

Na definição da linguagem ACTUS não foi encontrada nenhuma informação a esse respeito. O uso de procedimentos com variáveis paralelas exige do programador uma série de cuidados. Por exemplo, na definição de variáveis paralelas locais ao procedimento, elas devem ter uma extensão de paralelismo tal, que nunca produza incompatibilidade nas eventuais operações aritméticas, entre essas variáveis e as variáveis paralelas, cujos valores e extensões foram recebidos como parâmetros.

Por falta de uma definição e pelo uso cuidadoso que se deve fazer de procedimentos com variáveis paralelas, decidimos que não seriam permitidas chamadas a procedimentos, com variáveis paralelas como parâmetros, dentro de comandos que provoquem uma variação dinâmica da extensão de paralelismo. Obviamente, essa restrição é perfeitamente contornável a nível lógico pelo programador. Suponhamos que ele deseje testar elementos de um vetor paralelo e para os elementos maiores que zero, por exemplo, ele deseje aplicar operações definidas em um procedimento. Isso pode ser contornado, passando como parâmetro o vetor paralelo e fazendo o teste dentro do procedimento.

As funções em ACTUS são análogas às de PASCAL. Essa maneira de usar funções traz uma dificuldade clássica: os efeitos colaterais. Isso acontece, onde não apenas a função retorna um valor, mas modifica os valores de outras variáveis que estão no seu escopo. Um outro tipo de efeito colateral é a possibilidade, em uma linguagem de programação concorrente, da função ter concorrência interna. A avaliação de uma simples expressão pode, nesse caso, levar à criação e execução de

processos, surgindo a possibilidade de ocorrer *deadlock*. OCCAM define a função de uma maneira controlada, que proíbe efeitos colaterais mas é expressiva a ponto de permitir um uso eficiente. Na linguagem, as funções não podem ter:

1. construções **PAR**.
2. construções **ALT**.
3. operações de entrada e saída em canais.
4. atribuições a variáveis que não são locais a função.

Isso provoca as restrições impostas à função da LI, definidas no capítulo III. Em virtude dessa definição de OCCAM e da LI, o compilador deve traduzir para procedimentos as funções de ACTUS que operam sobre variáveis paralelas ou que são recursivas.

V.2.9 Comandos de Entrada e Saída

A linguagem OCCAM define que os programas devem utilizar canais específicos para usar os periféricos. Isso faz com que a entrada/saída de OCCAM seja muito trabalhosa. Por exemplo, a leitura de um dígito do teclado envolve um comando de entrada (leitura) no canal do teclado e a conversão do caracter ASCII lido para um caracter numérico. O Sistema de Desenvolvimento do Transputer (TDSII), que é o sistema operacional utilizado no desenvolvimento do gerador de código, fornece uma biblioteca de procedimentos que facilitam muito o uso dos periféricos. Ainda assim, no que diz respeito à entrada/saída, OCCAM é uma linguagem de baixo nível, se comparada à PASCAL.

O TDSII organizou os arquivos em registros e a leitura e escrita compreende ler e escrever um registro completo. Assim, o usuário deve tomar conhecimento do número e dos tipos de campos em um registro, para, depois de lê-lo, extrair, com as rotinas apropriadas, os campos desejados. Além disso, o usuário deve administrar os canais especificados pelo Sistema para acessar os arquivos em disco, pois o arquivo em OCCAM é associado a um canal e não a um nome lógico, como ocorre em PASCAL. As características acima mencionadas são as mais simples da entrada/saída do TDSII. Na realidade, cada procedimento da biblioteca do TDSII envolve muitos parâmetros (canais de disco, número do arquivo, tipo do arquivo, atributos, comentários, nome do registro, tamanho do registro e outros), e eles estão documentados em [24].

Resumindo, a complexidade da Entrada e Saída do Sistema contrasta enormemente com a simplicidade de PASCAL. A solução encontrada foi desenvolver um conjunto de procedimentos de um nivel mais alto que o oferecido pelo TDSII e gerar código para que o programa objeto em OCCAM utilize esses procedimentos. A biblioteca desenvolvida por nós, possibilitou a obtenção de uma LI com instruções de entrada e saída simples, facilitando o processo de compilação e de geração de

código. Uma única restrição do sistema operacional em relação à entrada/saída não pode ser contornada: o número de arquivos em disco abertos em um dado instante é no máximo quatro (o TDSII oferece apenas quatro canais para discos). Conseqüentemente, na LI, no máximo quatro arquivos em discos podem estar abertos ao mesmo tempo.

Os procedimentos da biblioteca simulam uma entrada/saída de PASCAL em OCCAM. Essas rotinas associam, através de tabelas, cada canal a um nome lógico e físico, e possibilitam que, como em PASCAL, o arquivo seja acessado pelo nome lógico. Além disso, administram as variáveis passadas para o TDSII, e fornecem, como em PASCAL, funções para consultar o *status* dos arquivos (**eof** e **eoln**). As leituras e escritas de campos são também administradas de forma transparente para o usuário, pela biblioteca, que lê e escreve registros, simulando PASCAL.

A desvantagem dessa biblioteca é que ela é um pacote fechado e, como tal, colocado completo dentro do programa objeto. Uma versão melhorada do TDSII poderia possibilitar aperfeiçoamentos dessa biblioteca.

V.3 O Programa Gerador de Código

O programa gerador de código (*back end*) foi implementado na linguagem OCCAM em um *transputer* T800, instalado em um PC XT.

O *transputer* usa os periféricos do PC, e por isso a entrada e saída torna-se bastante lenta em relação a velocidade de processamento do *chip*. Para tentar contornar esse problema, o programa foi projetado para funcionar como um **pipeline** de três estágios: leitura das intruções da LI, identificação das instruções e geração/gravação de código OCCAM.

O programa lê um arquivo tipo texto em LI e gera um objeto compilável na linguagem OCCAM. As tabelas de edp's e *arrays* geradas pelo compilador devem vir antes das instruções em LI, no mesmo arquivo.

Quando se mede a qualidade de uma representação intermediária, costuma-se levar em consideração a eficiência do algoritmo de geração de código. Como já foi mencionado, a LI foi projetada tentando evitar análises que já tivessem sido feitas anteriormente no processo de compilação. Como conseqüência, o Gerador de Código deve apenas identificar as intruções, pois depois de identificá-las, os seus formatos (número e tipo de campos) são conhecidos. O código em LI é gerado automaticamente pelo compilador, logo está livre de erros de sintaxe ou semânticos. Devido ao projeto da LI, o código para as instruções é gerado tão logo elas são lidas, ou seja, o Gerador de Código, ao processar uma instrução, não precisa processar outras instruções para gerar código para a primeira. Tudo isso, torna o algoritmo de geração de código simples. No caso das instruções paralelas, a geração de código já não é tão simples, pois a Linguagem ACTUS, como pode ser visto no capítulo II, pode acessar os *arrays* paralelos de várias maneiras, fazendo com que o Gerador de Código analise o tipo de enderaçamento feito e decida qual o código a ser gerado.

Resumindo, com exceção da geração de código das expressões paralelas, o algoritmo de geração de código é simples, embora seja extenso, devido ao grande conjunto de instruções da LI.

V.4 Testes Realizados

O gerador de código foi testado com um *benchmark* de 100 subrotinas (*loops*), que foram originalmente escritas em FORTRAN, para testar a eficiência de compiladores vetorizadores automáticos. Um compilador vetorizador automático é aquele que traduz o código escrito em uma linguagem serial (usualmente FORTRAN), para instruções vetoriais [25]. Essas subrotinas foram re-escritas em ACTUS e em LI e foram submetidas ao gerador de código. No processo de tradução para ACTUS, foi possível testar o poder de expressão da linguagem. Os resultados obtidos foram muito bons, ou seja, ACTUS na maioria dos casos (65 %) expressa completamente o paralelismo das subrotinas e na quase totalidade dos casos (81 %) expressa parcialmente ou totalmente o paralelismo da subrotinas, que deveria ser detectado por um super-compilador.

A tradução das subrotinas para a LI foi feita a partir do código em ACTUS e pode-se constatar, pelo menos manualmente, que a LI consegue expressar programas escritos em ACTUS de forma satisfatória, ou seja, atendendo a um objetivo de projeto, não é difícil traduzir ACTUS para a LI. As subrotinas originais escritas em FORTRAN podem ser encontradas em [25] e no Apêndice B podem ser encontrados, seguindo a identificação de [25], a tradução das subrotinas para ACTUS e LI, e o código gerado em OCCAM.

A validação do gerador de código consistiu na execução das rotinas em FORTRAN e do código gerado em OCCAM, para comparação de resultados. Nas subrotinas de FORTRAN e da LI foram acrescentadas chamadas a procedimentos que imprimem as matrizes resultantes das operações. Para executar as subrotinas de OCCAM mais rapidamente, foi criada uma biblioteca de subrotinas especializada em imprimir e inicializar as matrizes utilizadas nos testes. Além de se ter um código gerado menor (são 100 subrotinas), quando os testes foram iniciados, a LI para os comandos de entrada/saída de ACTUS , ainda estava em discussão.

V.5 Considerações sobre o Código Objeto Gerado

Um outro parâmetro para se medir a eficiência de uma representação intermediária, além da eficiência do algoritmo de geração de código, é a qualidade e tamanho do código gerado. A proporção entre os códigos em LI e o código em OCCAM, para as subrotinas testadas, foi aproxidamente de 1 (LI) para 1,5 (OCCAM) Porém, é difícil determinar se um código gerado automaticamente é ou não "inteligente". De concreto, tem-se que esse código é comprovadamente possível de ser gerado pela

tradução da LI. As Tabelas V.1 e V.5 resumem os resultados obtidos, e foram divididas em quatro grupos, conforme a divisão de [25]. A Tabela V.1 compara os tamanhos dos códigos fontes de FORTRAN e ACTUS, informando o grau de vetorização conseguido ao expressar os *loops* de FORTRAN na linguagem ACTUS. A Tabela V.5 compara os tamanhos dos códigos de ACTUS, LI e OCCAM.

	Compilador				
Loop	FORTRAN		ACTUS		Paralelização
	Fonte	Objeto	Fonte	Objeto	
111	182	628	196	269	vetorizado
112	191	644	175	271	vetorizado
113	187	624	150	256	vetorizado
114	265	807	388	718	parcial. vetorizado
115	284	773	258	820	parcial. vetorizado
116	384	839	530	655	parcial. vetorizado
117	382	821	591	1158	não vetorizado
121	245	674	200	269	vetorizado
122	298	1007	413	398	vetorizado
123	330	771	286	834	parcial. vetorizado
124	342	781	280	759	vetorizado
125	296	831	440	546	parcial. vetorizado
126	341	808	307	470	parcial. vetorizado
127	290	746	284	606	vetorizado
131	287	719	254	484	vetorizado
132	287	731	296	411	vetorizado
151	293	860	351	575	vetorizado
152	334	877	318	578	vetorizado
161	290	933	296	739	vetorizado
171	207	676	278	328	vetorizado
172	199	682	211	356	vetorizado
173	206	679	205	434	vetorizado
174	213	654	252	364	vetorizado
175	217	688	216	366	vetorizado

Tabela V.1: Comparação dos Tamanhos dos Códigos de FORTRAN e ACTUS

	Compilador				
Loop	FORTRAN		ACTUS		Paralela
	Fonte	Objeto	Fonte	Objeto	
211	276	937	250	554	vetorizado
212	291	104	316	492	vetorizado
221	316	1114	397	803	parcial. vetorizado
222	314	962	316	777	parcial. vetorizado
231	278	790	299	465	parcial. vetorizado
232	277	822	246	574	não vetorizado
233	460	1224	369	856	parcial. vetorizado
234	564	1457	343	867	não vetorizado
241	288	922	290	629	vetorizado
242	439	1585	455	942	não vetorizado
243	309	961	285	681	vetorizado
244	311	963	221	693	vetorizado
245	242	822	349	724	não vetorizado
251	246	760	242	412	vetorizado
252	298	857	288	586	vetorizado
253	339	955	293	823	vetorizado
254	243	714	180	434	vetorizado
255	285	744	198	733	vetorizado
271	265	824	234	537	vetorizado
272	404	972	392	1292	parcial. vetorizado
273	435	1492	380	1154	vetorizado
274	451	1272	441	1014	vetorizado
275	315	829	348	581	não vetorizado
276	323	949	266	567	vetorizado
277	380	1214	438	826	não vetorizado
278	378	1174	314	721	vetorizado
279	444	1224	427	1157	vetorizado
2710	636	1302	598	1502	vetorizado
2711	238	730	206	682	vetorizado
2712	219	680	188	516	vetorizado
281	261	907	522	486	vetorizado
291	233	701	179	419	vetorizado
292	233	701	199	716	vetorizado
293	194	624	151	255	vetorizado

Tabela V.2: Comparação dos Tamanhos dos Códigos de FORTRAN e ACTUS (Cont.)

	Compilador				
Loop	FORTRAN		ACTUS		Paralela
	Fonte	Objeto	Fonte	Objeto	
311	234	702	231	396	vetorizado
312	234	702	232	397	vetorizado
313	212	668	257	281	vetorizado
314	198	610	255	469	vetorizado
315	205	387	248	393	não vetorizado
316	237	808	279	351	vetorizado
317	158	601	270	207	vetorizado
318	275	716	472	1198	vetorizado
321	214	679	253	432	não vetorizado
322	236	730	204	588	não vetorizado
323	283	956	246	611	não vetorizado
331	192	587	186	355	não vetorizado
332	277	693	247	497	não vetorizado
341	305	913	338	513	não vetorizado
342	305	912	278	493	não vetorizado

Tabela V.3: Comparação dos Tamanhos dos Códigos de FORTRAN e ACTUS (Cont.)

	Compilador				
Loop	FORTRAN		ACTUS		Paralela
	Fonte	Objeto	Fonte	Objeto	
411	288	727	172	284	vetorizado
412	298	729	196	376	não vetorizado
413	334	929	235	466	vetorizado
414	458	1429	338	832	parcial. vetorizado
421	230	741	230	323	vetorizado
422	310	813	250	598	não vetorizado
423	283	916	248	333	vetorizado
424	288	920	111	158	vetorizado
425	421	1188	252	324	parcial. vetorizado
426	571	1396	322	363	parcial. vetorizado
427	270	923	210	254	vetorizado
441	361	975	352	1046	vetorizado
442	444	1122	328	1315	vetorizado
451	225	753	182	431	vetorizado
452	210	708	180	539	parcial. vetorizado
461	266	1008	259	445	vetorizado
471	407	1196	369	326	parcial. vetorizado
481	244	782	224	388	não vetorizado
482	267	735	208	418	não vetorizado
4101	214	763	191	243	vetorizado
4111	215	705	198	349	vetorizado
4112	215	705	198	349	vetorizado
4113	219	710	202	349	vetorizado
4114	312	1081	422	544	vetorizado
4115	256	880	468	779	vetorizado
4116	200	662	291	505	vetorizado
4117	216	711	312	471	vetorizado

Tabela V.4: Comparação dos Tamanhos dos Códigos de FORTRAN e ACTUS (Cont.)

Loop	ACTUS	LI	OCCAM
111	196	269	578
112	175	271	581
113	150	256	371
114	388	718	850
115	258	820	1096
116	530	655	1862
117	591	1158	916
121	200	269	579
122	413	398	613
123	286	834	596
124	280	759	1097
125	440	546	847
126	307	470	784
127	284	606	973
131	254	484	739
132	296	411	672
151	351	575	574
152	318	578	791
161	296	739	1138
171	278	328	635
172	211	356	464
173	205	434	488
174	252	364	442
175	216	366	664

Tabela V.5: Comparação dos Tamanhos dos Códigos de ACTUS, LI e OCCAM

Loop	ACTUS	LI	OCCAM
211	250	554	1034
212	316	492	808
221	397	803	819
222	316	777	963
231	299	465	546
232	246	574	434
233	369	856	987
234	343	867	600
241	290	629	1097
242	455	942	738
243	285	681	1179
244	221	693	1183
245	349	724	550
251	242	412	580
252	288	586	938
253	293	823	1203
254	180	434	921
255	198	733	2134
271	234	537	794
272	392	1292	1912
273	380	1154	1524
274	441	1014	1444
275	348	581	506
276	266	567	978
277	438	826	743
278	314	721	1112
279	427	1157	1858
2710	598	1502	2404
2711	206	682	1035
2712	188	516	745
281	522	486	1850
291	179	419	898
292	199	716	1869
293	151	255	381

Tabela V.6: Comparação dos Tamanhos dos Códigos de ACTUS, LI e OCCAM (Cont.)

Loop	ACTUS	LI	OCCAM
311	231	396	533
312	232	397	553
313	257	281	558
314	255	469	662
315	248	393	351
316	279	351	577
317	270	207	445
318	472	1198	3708
321	253	432	319
322	204	588	429
323	246	611	418
331	186	355	331
332	247	497	340
341	338	513	423
342	278	493	409

Tabela V.7: Comparação dos Tamanhos dos Códigos de ACTUS, LI e OCCAM (Cont.)

Loop	ACTUS	LI	OCCAM
411	172	284	461
412	196	376	420
413	235	466	511
414	338	832	1025
421	230	323	619
422	250	598	451
423	165	333	1157
424	111	158	536
425	252	324	736
426	322	363	933
427	142	254	978
441	352	1046	1517
442	328	1315	1870
451	182	431	510
452	180	539	740
461	259	445	571
471	369	326	782
481	224	388	409
482	208	418	373
4101	191	243	415
4111	198	349	440
4112	198	349	440
4113	202	349	444
4114	422	544	829
4115	468	779	1140
4116	291	505	374
4117	312	471	740

Tabela V.8: Comparação dos Tamanhos dos Códigos de ACTUS, LI e OCCAM (Cont.)

Capítulo VI

Conclusões

VI.1 Resultados Obtidos

Ao término de nosso trabalho, consideramos atingidos os objetivos de projeto da Linguagem Intermediária. A LI apresentada mantém completamente o paralelismo expresso em programas codificados em ACTUS, e a sua tradução para OCCAM é possível de ser feita eficientemente. A LI e o gerador de código, como já foi mencionado, foram testados através de 100 subrotinas, escritas originalmente em FORTRAN, documentadas em [25]. As subrotinas foram programadas em ACTUS e depois, manualmente, elas foram traduzidas para a LI, pois o *front-end* do compilador ainda não está disponível. Já em LI, essas subrotinas foram submetidas ao gerador de código. O código OCCAM gerado foi executado, e os resultados foram comparados com os resultados obtidos na execução das subrotinas em FORTRAN (as subrotinas em FORTRAN foram executadas no *transputer*). Os resultados obtidos por OCCAM foram iguais aos de FORTRAN. Com isso, testamos o entendimento da semântica dos comandos de ACTUS, provamos que é possível traduzir ACTUS em LI, e provamos que o código gerado é semanticamente correto.

Com os testes realizados, foi possível também medir o poder de expressão de paralelismo da linguagem ACTUS. Como já foi mencionado no Capítulo V, pode-se expressar totalmente o paralelismo de 65% das subrotinas testadas, e parcialmente/ totalmente o paralelismo de 81% das subrotinas. Uma informação interessante seria saber quantas dessas subrotinas teriam o seu paralelismo detectado pelos compiladores vetorizadores. Em [25], essas subrotinas escritas em FORTRAN foram submetidas à compiladores vetorizadores de vários fabricantes, e os resultados apresentados foram em relação à cada compilador vetorizador. Tais resultados foram comparados aos resultados obtidos em ACTUS. Nessa comparação, pode-se observar que muitas da subrotinas não paraleli- zadas pelos compiladores vetorizadores usavam o comando EQUIVALENCE de FORTRAN ou tinham comandos GOTO nos *loops*. Por outro lado, em ACTUS, pode-se observar que muitas das subrotinas não paralelizadas em ACTUS, o poderiam ter sido se a linguagem oferecesse funções que retornassem informações sobre conjuntos de índices, como menor valor ou maior valor de um conjunto de índices. Nas figuras VI.1 e VI.1, apresentamos os resultados

da comparação entre os compiladores vetorizadores e ACTUS.

Os próximos trabalhos a serem feitos dizem respeito à máquina alvo e ao sistema operacional do *transputer*. Tão logo esteja disponível a máquina alvo, esperamos incluir no gerador de código, que já está preparado para esta modificação, a tradução das expressões paralelas da LI para instruções a serem executadas pelo co-processador vetorial (o *i860*), uma vez que atualmente só foi possível implementar a geração de código seqüencial, em OCCAM, para estas expressões. Além disso, tendo em vista a aquisição de sistemas operacionais mais poderosos para *transputers*, planejamos construir um ambiente mais eficiente e amigável para o programador de ACTUS.

VI.2 Direcionamento da Pesquisa

Na pesquisa de linguagens paralelas, o término da implementação do compilador ACTUS vai nos possibilitar medir o quanto a linguagem satisfaz ou não as necessidades dos programadores de máquinas paralelas. Com esses resultados, podemos sugerir uma extensão da linguagem, por exemplo, uma linguagem ACTUS com construções para especificar e manipular paralelismo entre processos. Nesse caso, a LI também precisaria ser estendida.

Em relação ao compilador ACTUS, já estão sendo estudadas técnicas de otimização, cuja maior utilidade seria detectar paralelismo em estruturas de dados não expresso pelo programador, apesar da capacidade da linguagem de exprimí-lo, e paralelismo entre processos, uma vez que a linguagem OCCAM permite exprimir esse tipo de paralelismo.

	Máquina	Compilador	V
1	Alliant FX/8	FX/Fortran V3.0.14	65
2	Amdahl 1200/1400	Fortran 77/VP V10L20	59
3	Ardent Titan-1	Fortran prerelease	51
4	CDC Cyber 205	VAST-2 V2.21	62
5	CDC Cyber 990E/995E	VFTN V2.1	25
6	Convex C series	FC4.0	67
7	CRAY X-MP	CFT V1.15	50
8	CRAY X-MP	CFT77 V2.0	51
9	CRAY-2	CFT2 V3.1a	27
10	ETA-10	FTN77 V1.0	62
11	Gould NP1	VAST-2 prerelease	46
12	Hitachi S-810/820	FORT77/HAP V20-2B	67
13	IBM 3090/VF	VS Fortran V2.3.0	38
14	Intel iPSC-VX	VAST-2 V2.22	56
15	NEC SX/2	FORTRAN77/SX	54
16	SCS-40	CFT x13g	24
17	Stellar GS 1000	F77 prerelease	48
18	Unisys ISP	UFTN 3.2.14	58
19	Transputer T800	Actus	65

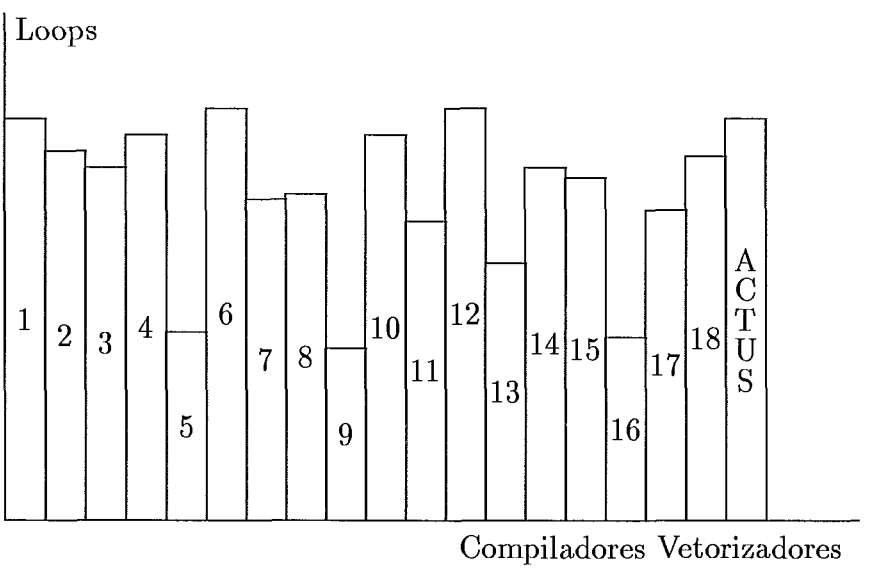

Figura VI.1: Comparação com os *Loops* Totalmente Vetorizados

	Máquina	Compilador	V + P
1	Alliant FX/8	FX/Fortran V3.0.14	71
2	Amdahl 1200/1400	Fortran 77/VP V10L20	71
3	Ardent Titan-1	Fortran prerelease	58
4	CDC Cyber 205	VAST-2 V2.21	67
5	CDC Cyber 990E/995E	VFTN V2.1	36
6	Convex C series	FC4.0	72
7	CRAY X-MP	CFT V1.15	51
8	CRAY X-MP	CFT77 V2.0	52
9	CRAY-2	CFT2 V3.1a	28
10	ETA-10	FTN77 V1.0	69
11	Gould NP1	VAST-2 prerelease	51
12	Hitachi S-810/820	FORT77/HAP V20-2B	71
13	IBM 3090/VF	VS Fortran V2.3.0	42
14	Intel iPSC-VX	VAST-2 V2.22	64
15	NEC SX/2	FORTRAN77/SX	59
16	SCS-40	CFT x13g	25
17	Stellar GS 1000	F77 prerelease	59
18	Unisys ISP	UFTN 3.2.14	70
19	Transputer T800	Actus	81

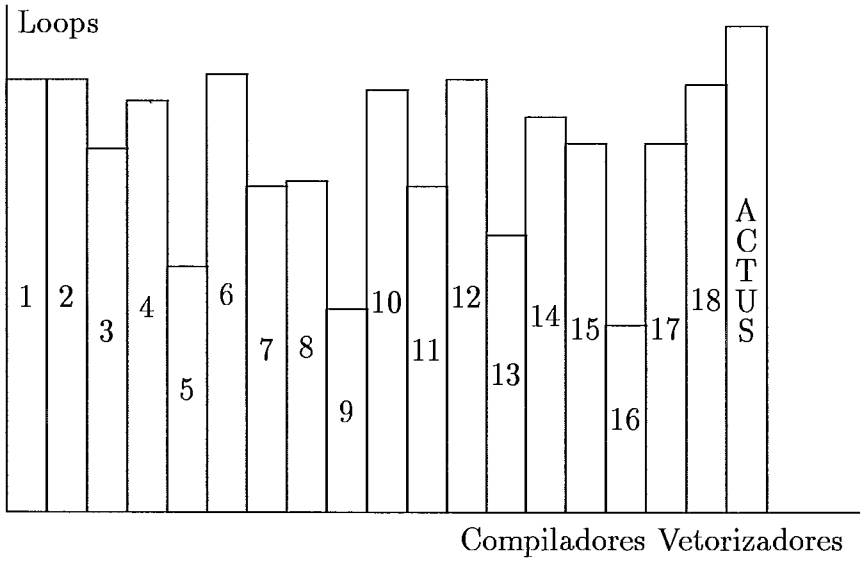

Figura VI.2: Comparação com os *Loops* Parcialmente e Totalmente Vetorizados

Referências Bibliográficas

[1] Perrot, R.H., Lyttle, R.W., e Dhillon., P.S., "The Design and Implementation of a Pascal-based language for Array Processor Architectures", *Journal of Parallel and Distributed Computing*, 4 (1987), 266-287.

[2] Perrot, R.H., *Parallel Programming*, Addison-Wesley, 1987.

[3] Perrot, R.H., Crookes, D. e Milligan, P., "The Programming Language ACTUS", *Software - Practice and Experience*, Vol. 13, 305-322 (1983).

[4] Perrot, R.H., *ACTUS, A Language for Array and Vector Processors - User Manual*, CS023, Department of Computer Science, The Queen's University, Fevereiro, 1983.

[5] Perrot, R.H., "A Language for Array and Vector Processors", *ACM Transactions on Programming Languages and Systems*, Vol. 1, No. 2, 177-195 (Outubro, 1979).

[6] Perrot, R.H., Crookes, D. e Milligan, P., "Implementing a Parallel Language on The CRAY-1", *Computer Physics Communications*, Vol. 37(1985), 119-124.

[7] Crookes D., Morrow, P.J., Milligan, P. e Kilpatrick, P.L., "Notes on Implementing a Language for Transputers Networks", *Microprocessing and Microprogramming*, 21(1987), 559-566, Noth-Holland.

[8] Burns, Allan, *Programming in OCCAM 2*, Addison-Wesley Publishing Company, 1988.

[9] Inmos Limited, *OCCAM 2 - Reference Manual*, Prentice Hall, 1988.

[10] Pountain, D. e May, D., *A Tutorial Introduction of OCCAM 2*, BSP Professional Books, 1987.

[11] Elizabeth, M. e Hull, C., "OCCAM - A Programming Language for Multiprocessor System", *Computer Languages*, Vol. 12, No. 1, 27-37 (1987).

[12] Hwang, K. e Briggs, F.A., *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984.

[13] Inmos Limited, *Preliminary Data - IMS T800 Transputer*, Março 1988.

[14] Polychronopoulos, C.D., *Parallel Programming and Compilers*, Kluwer Academic Publishers, 1988.

[15] Padua, D.A. e Wolfe, M.J., "Advanced Compiler Optimizations for Supercomputers", *Communications of The ACM*, Dezembro 1986, Vol. 29, No. 12.

[16] Amorim, C.L., Barbosa, V.C. e Fernandes, E.S.T., *Uma Introdução à Computação Paralela e Distribuída*, VI Escola de Computação, Campinas-SP, 1988.

[17] Bohrland, *Pascal 3.0 - Reference Manual*, Editora, 1983.

[18] Elsworth, E.F., "Compilation via an intemediate language", *The computer Journal*, Vol. 22, No. 3, 1978.

[19] Tanenbaum, A.S., Staveren, H.V., Keizer, E.G. e Stevenson, J.W., "A Practical Tool Kit for Making Portable Compilers", *Communications of The ACM*, Vol. 26, No. 9, Setembro, 1983.

[20] Aho, A.V., Sethi, R. e Ullman, J.D., *Compilers – Principles, Techniques and Tools*, Addison Wesley, 1986.

[21] Barret, W.A. e Couch, J.D., *Compiler Construction: Theory and Practice*, Science Research Associates, INC, 1979.

[22] Pratt, T.W., *Programming Languages, Design and Implementation*, Prentice-Hall, Segunda Edição, 1984.

[23] Inmos Limited, *The Transputer Instruction Set – A Compiler Writer's Guide*, 1987.

[24] Inmos Limited, *Transputer Development System*, Prentice-Hall, 1988.

[25] Callaham, D., Dongarra, J. e Levine, D., *Vectorizing Compilers: A Test Suit and Results*, Argonne National Laboratory, Mathematics and Computer Science Division, Technical Memorandum No. 109, Março 1988.

[26] Hoare, C.A.R., "Communicating Sequential Processes", *Communications of the ACM*, Vol. 21, No. 8, Agosto 1978.

Apêndice A

A Implementação da Linguagem Intermediária

A.1 Introdução

Durante esse trabalho, a LI foi apresentada em um formato relativamente livre, pois o objetivo era apresentar as idéias usadas na concepção da LI. Este apêndice destina-se a apresentar as instruções da LI, na forma em que elas foram implementadas. Algumas convenções serão usadas. Os campos escritos na forma $\{_0$, campo$\}$ são opcionais, podendo ocorrer zero ou mais vezes. Os campos $\{_1$, campo$\}$ devem ocorrer pelo menos uma vez, ou seja, eles podem ocorrer uma ou mais vezes. Os campos $\{\text{campo}\}_n$ devem ocorrer exatamente n vezes. As palavras reservadas são escritas com letra maiúscula. O símbolo "|" indica **ou**, os símbolos ":" e ";" fazem parte da LI, os comentários são precedidos por "–" e o símbolo "*" indica final de linha.

A.2 Formato do Arquivo em LI

O arquivo em LI é composto da tabela de *arrays*, da tabela de edp's e das instruções em LI, dispostas na seguinte ordem e formato:

tabela de *arrays*
::
tabela de edp's
::
instruções em LI
::

A tabela de *arrays* é composta de registros com o seguinte formato:

: tipo ; num_dimensões ; $\{\text{dados\_dimensão}\}_{num\_dimensões}$
tipo = um dos tipos primitivos da LI
num_dimensões = um valor inteiro que é o número de
dimensões do *array*
dados_dimensão = tipo_dimensão ; número_elementos ;
tipo_dimensão = 0 – dimensão escalar
| 1 – dimensão paralela
número_elementos = um valor inteiro que é o número de
elementos naquela dimensão do *array*

O formato em que o **tipo** deve ser codificado é especificado na seção A.3, que descreve as instruções da LI.

A tabela de edp's é composta de registros que têm o seguinte formato:

: início ; incremento ; tamanho ;
início = 0 ; valor_inteiro
| 1 ; nome_dado
incremento = 0 ; valor_inteiro
| 1 ; nome_dado
tamanho = 0 ; valor_inteiro
| 1 ; nome_dado

O campo **nome_dado** de **incremento** pode significar o nome de uma variável, se a edp é regular, ou o nome de um vetor de índices, se a edp é irregular. No último caso, o campo **início** é nulo, que é quando ele tem os seguintes valores:

início = 1 ; 0

A.3 Instruções da LI

Inicialmente, alguns símbolos usados nas descrição das instruções serão definidos, e depois ser ao mostradas as instruções.

- fim_bloco = número de instruções do bloco
- fim_código_teste = número de instruções do **código_teste**
- código_teste = código gerado para o tratamento de expressões booleanas, nos comandos condicionais e de controle
- fim_bloco_then = número de instruções do do bloco de instruções que compõem o **then** da instrução **if-then-else**.
- fim_instru,c ao = náumero de componentes da instrução.
- nome_dado = nome de uma variável
- nome_label = nome de uma posição no texto, a ser referenciada por um comando **goto**
- dígitos = valores reais ou inteiros
- valor_inteiro = valor inteiro positivo
- operando 0 = 0 ; dígitos
- operando 1 = 1 ; nome_dado
- operando 2 = 2 ; dado_array_escalar
- operando 3 = 3 ; dado_array_paralelo_variação_indepte_índices
- operando 4 = 4 ; dado_array_paralelo_variação_depte_índice
- operando 5 = 5 ; constante_paralela
- número_dimensões = valor_inteiro
- dado_array_escalar = nome_dado ; número_dimensões ; {dimensão_escalar ;}$_{número\_dimensões}$
- dimensão_escalar = 0 ; valor_inteiro
  | 0 ; nome_dado
- dado_array_paralelo = nome_dado ; número_dimensões ; {dimensão_paralela ; | dimensão_escalar ;}$_{número\_dimensões}$
- dimensão_paralela = 1 ; valor_edp
- valor_edp = valor_inteiro – ponteiro para tabela de edp's

- dado_array_paralelo_variação_indepte_índices = dado_array_paralelo
- dado_array_paralelo_variação_depte_índices = dado_array_paralelo
- constante_paralela = 1 ; dimensão_paralela

(1) dec = DEC ; nome_dado ; tipo ; *
tipo = ↑
| tipoprimitivo
↑ = valor_inteiro – ponteiro para tabela de *arrays*
tipoprimitivo = 30000 – INT
| 30001 – INT16
| 30002 – INT32
| 30003 – INT64
| 30004 – REAL32
| 30005 – REAL64
| 30006 – BYTE
| 30007 – BOOL

(2) block = BLOCK ; fim_bloco ; *
bloco
bloco = block
| while
| repeat
| for
| label
| goto
| if
| case
| exp
| config
| expp
| cpvec
| proc
| callproc
| function
| callfunction
| assign
| open
| create
| close
| finish
| read
| write
| newline

(3) while = WHILE ; vb ; fim_codigo_teste ; fim_bloco ; *
codigo_teste

bloco
codigo_teste = { $_0$, exp | expp }
vb = operando 0 – tipo booleano
| operando 1 – tipo booleano
| operando 2 – tipo booleano

(4) repeat = REPEAT ; vb ; fim_bloco ; *
bloco
codigo_teste

(5) for = FOR ; vi ; fim_bloco ; *
bloco
vi = operando 0 – tipo inteiro
| operando 1 – tipo inteiro
| operando 2 – tipo inteiro

(6) label = LABEL ; nome_label ; *

(7) goto = GOTO ; nome_label ; *

(8) if = codigo_teste
IF ; vb ; fim_bloco_then ; fim_bloco ; *
bloco_then
bloco_else
| codigo_teste
IF ; vb ; fim_bloco_then ; 0 ; *
bloco_then
bloco_then = bloco
bloco_else = bloco

(9) case = CASE ; fim_bloco ; *
bloco_case
bloco_case = { $_1$, opção
bloco_opção }
opção = OPTION ; tipo_valores ; número_valores ; { vi ; }$_{número\_valores}$ *
tipo_valores = 0 – valores contínuos
| 1 – valores discretos
número_valores = valor_inteiro
bloco_opção = block

(10) exp = EXP ; operação ; operandoR ; operando1 ; operando2 ; *
operandoR, operando1, operando2 = operando 0
| operando 1
| operando 2
operação = COPY – atribuicao
| ADD – +
| SUB – –
| MULT – *
| DIV – /

| REM –
| MINUS – –
| BAND – *and* bit a bit
| BOR – *or* bit a bit
| BXOR – *xor* bit a bit
| SHR – deslocamento a direita
| SHL – deslocamento a esquerda
| AND – *and* logico
| OR – *or* logico
| NOT – *not* logico
| BNOT – *not* bit a bit
| EQUAL – =
| NOTEQ – <>
| LESS – <
| GREAT – >
| LESEQ – <=
| GTREQ – >=
| CON – conversão
| ROU – arredondamento
| TRU – truncamento

(11) expp = EXPP ; operacao; operandoPR; operandoP1;
operandoP2; máscara; *

operandoPR = operando 1
| operando 2
| operando 3
| operando 4

operandoP1, operandoP2 = operando 0
| operando 1
| operando 2
| operando 3
| operando 4

máscara = operando 3 – tipo booleano

(12) config = CONFIG ; número_edps ; { ⇑ ; }$_{número\_edps}$ *
número_edps = valor_inteiro – valor maximo = 2

(13) cpvec = CPVEC ; nome_dado ; ⇑ ; fim_instrução ; *
{$_1$, CPVAL ; número_valores ; { valor }$_{número\_valores}$ } *
número_valores = valor_inteiro
valor = operando 0

(14) par = PAR ; nome_dado ; tipo ; by_value ; *
by_value = 1 – se por valor
| 0 – se por referência

(15) cpar = CPAR ; operando ; *
operando = operando 0
| operando 1
| operando 2
| operando 3

(16) proc = PROC; nome_subrotina; recursive; fim_par; fim_bloco; *
bloco_parametros
bloco_proc
recursive = 0 – procedimento nao recursivo
| 1 – procedimento recursivo
bloco_parametros = { $_1$, par}
bloco_proc = bloco

(17) function = FUNCTION; nome_subrotina; tipo; fim_par;
fim_dec; fim_bloco; *
bloco_parametros
bloco_dec
bloco_function
bloco_dec = { $_0$, dec}
bloco_function = bloco
result
result = RESULT ; resultado
resultado | operando 0
| operando 1
| operando 2

(18) callproc = CALLPROC; nome_subrotina; recursive; fim_par; *
bloco_cpar
bloco_cpar = { $_0$, cpar }

(19) callfunc = CALLFUNC; nome_subrotina; nome_dado; fim_par; *
bloco_cpar

(20) assign = ASSIGN ; nome_arquivo_físico ; nome_arquivo_lógico ; *

(21) open = OPEN ; nome_arquivo_lógico ; *

(22) create = CREATE ; nome_arquivo_lógico ; *

(23) close = CLOSE ; nome_arquivo_lógico ; *

(24) finish = FINISH ; nome_arquivo_lógico ; *

(25) newline = NEWLINE ; nome_arquivo_lógico ; *

(26) read = READ ; nome_arquivo_lógico ; read_type ; nome_dado ; *
read_type = 0 – char
| 1 – int
| 2 – int64

| 3 – int32
| 4 – real64
| 5 – real32
| 6 – text

(27) write = WRITE ; nome_arquivo_lógico ; write_type ; nome_dado ; *
write_type = 7 – char
| 8 – int
| 9 – int64
| 10 – int32
| 11 – real64
| 12 – real32
| 13 – text

Apêndice B

Código das Subrotinas dos Testes

B.1 Introdução

Neste apêndice, apresentamos alguns dos testes realizados para validação da LI e do Gerador de Código. O nosso principal objetivo, com os testes, além da depuração do Gerador de Código, foi verificar a isenção de erros semânticos do código OCCAM gerado. Isso foi verificadao através da comparação dos resultados obtidos na execução das subrotinas em FORTRAN e OCCAM. As subrotinas, nas duas linguagens, foram executadas com os mesmos dados (matrizes e vetores). Nas subrotinas de FORTRAN e OCCAM, incluímos subrotinas para inicialização das matrizes usadas e subrotinas para a impressão dos resultados. As seções seguintes apresentam as matrizes usadas e os códigos das subrotinas nas quatro linguagens, FORTRAN, ACTUS II, LI e OCCAM 2.

Os códigos OCCAM têm, nas primeiras linhas, as chamadas das bibliotecas de inicialização e impressão específicas para os testes, a chamada da biblioteca de I/O geral, construída por nós, e a chamada da biblioteca de funções pré-definidas.

B.2 Códigos das Subrotinas

```
        SUBROUTINE S111(a,N,itam)
        DIMENSION a(1000)
        DO 400 i = 2,itam,2
           a(i) = a(i-1)
400     CONTINUE
        WRITE (6,100) (a(i),i=1,n)
100     FORMAT (e12.6)
        RETURN
        END
```

```
PROGRAM S111;

CONST
  N = 100;
 N1 = 999;
VAR
  a : ARRAY[1:1000] OF REAL;
INDEX
  PAR : INTEGER;
BEGIN
  USING PAR := 2:[2]N DO
    a[PAR] := a[PAR SHIFT -1];
  WRITE(a[N]);
END.
```

s111.tsr

```
:30004;1;1;4;
::
:0;1;0;2;0;2;
:0;0;0;2;0;2;
::
DEC;VA;0;
BLOCK;8;
  CALLPROC;iniciaVA;0;1;
  CPAR;1;VA;
  CONFIG;1;0;1;
    EXPP;COPY;4;VA;1;1;0;4;VA;1;1;1;0;0;0;0;
  CALLPROC;imprimevetor;0;3;
  CPAR;1;VA;
  CPAR;1;screen;
  CPAR;1;keyboard;
```

```
 0111

#USE imprime
#USE dados
#USE pfdma
#USE userio
[4]REAL32 VA:
SEQ
  iniciaVA ( VA)
  [2]INT gc.cilarge:
  INT    gc.value:
  SEQ
    gc.value := 1
    SEQ i0   = 0 FOR 2
      SEQ
        gc.cilarge[i0  ] := gc.value
        gc.value := gc.value + (2)
    [2]INT gc.cidep1:
    INT    gc.value:
    SEQ
      gc.value := 0
      SEQ i0   = 0 FOR 2
        SEQ
          gc.cidep1[i0  ] := gc.value
          gc.value := gc.value + (2)
      PAR j = 0 FOR 2
        VA[gc.cilarge[j]] := VA[gc.cidep1[j]]
    imprimevetor (VA, screen, keyboard)
```

```
        SUBROUTINE S151(a,b,n)
        INTEGER n
        REAL a(*),b(*)
        CALL S151s(a,b,n-1,1)
        RETURN
        END
        SUBROUTINE S151s(a,b,n,m)
        INTEGER n,m
        REAL a(*),b(*)
        DO 730 i = 1,n
           a(i) = a(i+m) + b(i)
730     CONTINUE
        WRITE (6,100) (a(i),i=1,n)
100     FORMAT (e12.6)
        RETURN
        END
```

```
PROGRAM S151;

CONST
  N = 1000;
VAR
  a,
  b :  ARRAY[1:N] OF REAL;
  m :  INTEGER;

PROCEDURE S151s(VAR a,
                    b   : ARRAY[1:N] OF REAL;
    n,m : INTEGER);

  INDEX
    IS : INTEGER;
  BEGIN
    USING IS:=1:N DO
      a[IS] := a[IS SHIFT m] + b[IS];
  END;

BEGIN
 S151s(a,b,N,m);
END.
```

s151.tsr

```
:30004;1;1;4;
::
:0;0;0;1;1;N;
:1;M;0;1;0;3;0;
::
DEC;VA;0;
DEC;VB;0;
DEC;M;30000;
DEC;N;30000;
PROC;S151s;0;4;7;
PAR;VA;0;0;
PAR;VB;0;0;
PAR;N;30000;0;
PAR;M;30000;0;
  BLOCK;
    CONFIG;1;0;1;
      EXPP;ADD;3;VA;1;1;0;4;VA;1;1;1;3;VB;1;1;0;0;0;
BLOCK;15;
  CALLPROC;iniciaVA;0;1;
  CPAR;1;VA;
  CALLPROC;iniciaVB;0;1;
  CPAR;1;VB;
  EXP;COPY;1;M;0;1;0;0;
  EXP;COPY;1;N;0;3;0;0;
  CALLPROC;S151s;0;4;
  CPAR;1;VA;
  CPAR;1;VB;
  CPAR;1;N;
  CPAR;1;M;
  CALLPROC;imprimevetor;0;3;
  CPAR;1;VA;
  CPAR;1;screen;
  CPAR;1;keyboard;
```

```
 0151

#USE imprime
#USE userio
#USE dados
#USE pfdma
[4]REAL32 VA:
[4]REAL32 VB:
INT    M:
INT    N:
PROC S151s([4]REAL32 VA, [4]REAL32 VB, INT    N, INT    M)
  SEQ
    INT    gc.lgval:
    INT    gc.dpval0:
    SEQ
      gc.lgval := 0
      gc.dpval0 := M
      SEQ j = 0 FOR N
        SEQ
          VA[gc.lgval] := VA[gc.dpval0] + VB[gc.lgval]
          gc.lgval := gc.lgval + (1)
          gc.dpval0 := gc.dpval0 + (1)
:
SEQ
  iniciaVA ( VA)
  iniciaVB ( VB)
  M := 1
  N := 3
  S151s ( VA, VB, N, M)
  imprimevetor ( VA, screen, keyboard)
```

```
      SUBROUTINE S152(a,b,n)
      INTEGER n
      REAL a(*),b(*)
      DO 750 i = 1,n
         b(i) = b(i) + 2
             CALL S152s(a,b,i)
750   CONTINUE
      WRITE (6,100) (a(i),i=1,n)
100   FORMAT (e12.6)
      RETURN
      END
      SUBROUTINE S152s(a,b,i)
      INTEGER n,m
      REAL a(*),b(*)
      a(i) = a(i) + b(i)
      RETURN
      END
```

```
PROGRAM S152;

CONST
  N = 1000;
TYPE
 ARRAY1 = ARRAY[1:N] OF REAL;
VAR
  a,
  b  :  ARRAY1;
INDEX
  ID : INTEGER;

PROCEDURE  S152s ( VAR a,b : ARRAY1);

  BEGIN
    USING ID := 1:N DO
      a[ID] := a[ID]  + b[ID];
  END;

BEGIN
  USING ID := 1:N DO
    b[ID] := b[ID] + 2;
  S152s(a,b);
END.
```

s152.tsr

```
:30004;1;1;4;
::
:0;0;0;1;0;4;
::
DEC;VA;0;
DEC;VB;0;
PROC;S152s;0;2;5;
PAR;VA;0;0;
PAR;VA;0;0;
BLOCK;2;
  CONFIG;1;0;1;
      EXPP;ADD;3;VA;1;1;0;3;VA;1;1;0;3;VB;1;1;0;0;
BLOCK;17;
  CALLPROC;iniciaVA;0;1;
  CPAR;1;VA;
  CALLPROC;iniciaVB;0;1;
  CPAR;1;VB;
  CONFIG;1;0;1;
    EXPP;ADD;3;VB;1;1;0;3;VB;1;1;0;0;2.0(REAL32);0;0;
  CALLPROC;S152s;0;2;
  CPAR;1;VA;
  CPAR;1;VB;
  CALLPROC;imprimevetor;0;3;
  CPAR;1;VA;
  CPAR;1;screen;
  CPAR;1;keyboard;
  CALLPROC;imprimevetor;0;3;
  CPAR;1;VB;
  CPAR;1;screen;
  CPAR;1;keyboard;
```

```
 0152

#USE imprime
#USE userio
#USE dados
#USE pfdma
[4]REAL32 VA:
[4]REAL32 VB:
PROC S152s([4]REAL32 VA, [4]REAL32 VB)
  SEQ
    [4]INT gc.cilarge:
    INT    gc.value:
    SEQ
      gc.value := 0
      SEQ i0   = 0 FOR 4
        SEQ
          gc.cilarge[i0  ] := gc.value
          gc.value := gc.value + (1)
      PAR j = 0 FOR 4
        VA[gc.cilarge[j]] := VA[gc.cilarge[j]] + VB[gc.cilarge[j]]
:
SEQ
  iniciaVA ( VA)
  iniciaVB ( VB)
  [4]INT gc.cilarge:
  INT    gc.value:
  SEQ
    gc.value := 0
    SEQ i0   = 0 FOR 4
      SEQ
        gc.cilarge[i0  ] := gc.value
        gc.value := gc.value + (1)
    PAR j = 0 FOR 4
      VB[gc.cilarge[j]] := VB[gc.cilarge[j]] + 2.0(REAL32)
  S152s ( VA, VB)
  imprimevetor ( VA, screen, keyboard)
```

```
        SUBROUTINE S242(a,b,c,n)
        dimension a(100),b(100),c(100),d(100)
        dimension abc1(100),abc2(100),bcd1(100)
        DO 510 i = 1,n
           abc1(i-1) = abc2(i-1) + 1
           bcd1(i-1) = abc1(i-1) ** 2
           a(i) = bcd1(i-1) + d(i)
           abc2(i) = b(i+1) + b(i-1)
           b(i) = c(i) + 1
           c(i+1) = b(i) + 1
510     continue
        write(6,12) (a(i),i=1,n)
        write(6,12) (b(i),i=1,n)
        write(6,12) (c(i),i=1,n)
12      format (e12.6)
        return
        end
```

```
PROGRAM S242;
  CONST
    N = 100;
  VAR
    a,
    b,
    c,
    d,
    abc1,
    abc2,
    bcd1  : ARRAY[1..N] OF REAL;
    i     : INTEGER;
  BEGIN
    FOR i := 2 TO n-1 DO
      BEGIN
        abc1[i-1] := abc2[i-1] + 1;
        bcd1[i-1] := Power(Abc1[i-1],2);
        a[i] := bcd1[i-1] + d[i];
        abc2[i] := b[i+1] + b[i-1];
        b[i] := c(i) + 1;
        c[i+1] := b[i] + 1;
      END;
    WRITE(c[N]);
    WRITE(b[N]);
  END.
```

s242.tsr

```
:30004;1;0;4;
::
::
DEC;ABC1;0;
DEC;BCD1;0;
DEC;ABC2;0;
DEC;VA;0;
DEC;VB;0;
DEC;VC;0;
DEC;VD;0;
DEC;I;30000;
BLOCK;32;
  CALLPROC;iniciaVA;0;1;
  CPAR;1;VA;
  CALLPROC;iniciaVB;0;1;
  CPAR;1;VB;
  CALLPROC;iniciaVC;0;1;
  CPAR;1;VC;
  CALLPROC;iniciaVD;0;1;
  CPAR;1;VD;
  EXP;COPY;1;I;0;1;0;0;
  FOR;0;2;10;
    BLOCK;9;
      EXP;ADD;2;ABC1;1;1;I-1;2;ABC2;1;1;I-1;0;1.0(REAL32);
      CALLFUNC;POWER;2;BCD1;1;1;I-1;2;
      CPAR;2;ABC1;1;1;I-1;
      CPAR;0;2.0(REAL32);
      EXP;ADD;2;VA;1;1;I;2;BCD1;1;1;I-1;2;VD;1;1;I;
      EXP;ADD;2;ABC2;1;1;I;2;VB;1;1;I+1;2;VB;1;1;I-1;
      EXP;ADD;2;VB;1;1;I;2;VC;1;1;I;0;1.0(REAL32);
      EXP;ADD;2;VC;1;1;I+1;2;VB;1;1;I;0;1.0(REAL32);
      EXP;ADD;1;I;1;I;0;1;
  CALLPROC;imprimevetor;0;3;
  CPAR;1;VA;
  CPAR;1;screen;
  CPAR;1;keyboard;
  CALLPROC;imprimevetor;0;3;
  CPAR;1;VB;
  CPAR;1;screen;
  CPAR;1;keyboard;
  CALLPROC;imprimevetor;0;3;
  CPAR;1;VC;
  CPAR;1;screen;
  CPAR;1;keyboard;
```

æ

```
  0242

#USE imprime
#USE userio
#USE dados
#USE pfdma
#USE snglmath
[4]REAL32 ABC1:
[4]REAL32 BCD1:
[4]REAL32 ABC2:
[4]REAL32 VA:
[4]REAL32 VB:
[4]REAL32 VC:
[4]REAL32 VD:
INT    I:
SEQ
  iniciaVA ( VA)
  iniciaVB ( VB)
  iniciaVC ( VC)
  iniciaVD ( VD)
  I := 1
  SEQ i0   = 0 FOR 2
    SEQ
      ABC1[I-1] := ABC2[I-1] + 1.0(REAL32)
      BCD1[I-1] := POWER ( ABC1[I-1], 2.0(REAL32))
      VA[I] := BCD1[I-1] + VD[I]
      ABC2[I] := VB[I+1] + VB[I-1]
      VB[I] := VC[I] + 1.0(REAL32)
      VC[I+1] := VB[I] + 1.0(REAL32)
      I := I + 1
  imprimevetor ( VA, screen, keyboard)
  imprimevetor ( VB, screen, keyboard)
  imprimevetor ( VC, screen, keyboard)


æ
```

```
      SUBROUTINE S279(a,b,c,n)
      INTEGER n
      REAL a(*),b(*),c(*)
      DO 810 i = 1,n
         IF (a(i).GT.0) GOTO 811
         b(i) = -b(i)
         if (abs(b(i)).LE.a(i)) GOTO 812
         c(i) = ABS(c(i))
         GOTO 812
811      CONTINUE
         c(i) = -c(i)
812      CONTINUE
         a(i) = b(i) + c(i)
810   CONTINUE
      WRITE (6,11) (a(i),i=1,n)
      WRITE (6,11) (b(i),i=1,n)
      WRITE (6,11) (c(i),i=1,n)
11    FORMAT (e12.6)
      RETURN
      END
```

```
PROGRAM S279;
  CONST
    N = 100;
  VAR
    a,
    b,
    c  : ARRAY[1:N] OF REAL;
  INDEX
    IS : INTEGER;
  BEGIN
    USING IS := 1:N DO
      BEGIN
        IF a[IS] <= 0 THEN
          c[IS] := -c[IS]
        ELSE
          BEGIN
            b[IS] := -b[IS];
            IF (ABS(B[IS]) > A[IS]) THEN
              c[IS] := ABS(C[IS]);
          END;
        a[IS] := b[IS] + c[IS];
      END;
  END.
```

s279.tsr

```
:30004;1;1;4;
:30007;1;1;4;
::
:0;0;0;1;0;4;
::
DEC;VA;0;
DEC;VB;0;
DEC;VC;0;
DEC;AUX;0;
DEC;MASC;1;
DEC;MASC2;1;
DEC;TESTE;30007;
DEC;TESTE2;30007;
BLOCK;22;
  CALLPROC;iniciaVA;0;1;
  CPAR;1;VA;
  CALLPROC;iniciaVB;0;1;
  CPAR;1;VB;
  CALLPROC;iniciaVC;0;1;
  CPAR;1;VC;
  CONFIG;1;0;21;
  BLOCK;20;
    EXPP;GREAT;3;MASC;1;1;0;3;VA;1;1;0;0;0.0(REAL32);0;0;
    CALLFUNC;ifanyD1;1;TESTE;1;
    CPAR;2;MASC;
    IF;1;TESTE;1;15;
      EXPP;MULT;3;VC;1;1;0;3;VC;1;1;0;0;(-1.0(REAL32));3;MASC;1;1;0;
        BLOCK;13;
          EXPP;NOT;3;MASC;1;1;0;3;MASC;1;1;0;0;0;0;0;
          EXPP;MULT;3;VB;1;1;0;3;VB;1;1;0;0;(-1.0(REAL32));3;MASC;1;1;0;
          EXPP;COPY;3;AUX;1;1;0;3;VB;1;1;0;0;0;0;0;
          CALLPROC;absD1;0;1;
          CPAR;1;AUX;
          EXPP;GREAT;3;MASC2;1;1;0;3;AUX;1;1;0;3;VA;1;1;0;3;MASC;1;1;0;
          CALLFUNC;ifanyD1;1;TESTE2;1;
          CPAR;2;MASC2;
          IF;1;TESTE2;4;0
            BLOCK;3;
              EXPP;COPY;3;VC;1;1;0;3;VC;1;1;0;0;0;3;MASC2;1;1;0;
              CALLPROC;absD1;0;1;
              CPAR;1;VC;
    EXPP;ADD;3;VA;1;1;0;3;VB;1;1;0;3;VC;1;1;0;0;0;
```

```
  0279

#USE imprime
#USE userio
#USE dados
#USE pfdma
[4]REAL32 VA:
[4]REAL32 VB:
[4]REAL32 VC:
[4]REAL32 AUX:
[4]BOOL   MASC:
[4]BOOL   MASC2:
BOOL   TESTE:
BOOL   TESTE2:
SEQ
  iniciaVA ( VA)
  iniciaVB ( VB)
  iniciaVC ( VC)
  [4]INT gc.cilarge:
  INT    gc.value:
  SEQ
    gc.value := 0
    SEQ i0   = 0 FOR 4
      SEQ
        gc.cilarge[i0  ] := gc.value
        gc.value := gc.value + (1)
    SEQ
      PAR j = 0 FOR 4
        MASC[gc.cilarge[j]] := VA[gc.cilarge[j]] > 0.0(REAL32)
      TESTE := ifanyD1 ( MASC)
      IF
        TESTE
          PAR j = 0 FOR 4
            IF
              MASC[j]
                VC[gc.cilarge[j]] := VC[gc.cilarge[j]]
                                     * (-1.0(REAL32))
              TRUE
                SKIP
        TRUE
          SEQ
            PAR j = 0 FOR 4
              MASC[gc.cilarge[j]] :=  NOT MASC[gc.cilarge[j]]
            PAR j = 0 FOR 4
              IF
                MASC[j]
                  VB[gc.cilarge[j]] := VB[gc.cilarge[j]]
                                       * (-1.0(REAL32))
```

```
          TRUE
            SKIP
        PAR j = 0 FOR 4
          AUX[gc.cilarge[j]] := VB[gc.cilarge[j]]
        absD1 ( AUX)
        PAR j = 0 FOR 4
          IF
            MASC[j]
              MASC2[gc.cilarge[j]] := AUX[gc.cilarge[j]]
                                      > VA[gc.cilarge[j]]
            TRUE
              SKIP
        TESTE2 := ifanyD1 ( MASC2)
        IF
          TESTE2
            SEQ
              PAR j = 0 FOR 4
                IF
                  MASC2[j]
                    VC[gc.cilarge[j]] := VC[gc.cilarge[j]]
                  TRUE
                    SKIP
              absD1 ( VC)
          TRUE
            SKIP
PAR j = 0 FOR 4
  VA[gc.cilarge[j]] := VB[gc.cilarge[j]] + VC[gc.cilarge[j]]
```

```
      SUBROUTINE S4111(a,b,ip,n)
      INTEGER n,ip(*)
      REAL a(*),b(*)
      DO 560 i = 1,n
         a(ip(i)) = b(i)
560   CONTINUE
      write(6,12) (a(i),i=1,n)
12    format (e12.6)
      RETURN
      END
```

```
PROGRAM S4111;

CONST
  N = 100;
VAR
  a,
  b  :  ARRAY[1:100] OF REAL;
  ip :  ARRAY[1:100] OF INTEGER;
INDEX
  ID : INTEGER;
BEGIN
  USING ID := 1:N DO
    a[ip[ID]] := b[ID];
END.
```

s4111.tsr

```
:30004;1;1;4;
:30000;1;1;4;
::
:0;0;0;1;0;4;
:1;0;IP;0;4;
::
DEC;VA;0;
DEC;VB;0;
DEC;IP;1;
BLOCK;12;
  CALLPROC;iniciaVB;0;1;
  CPAR;1;VB;
  CALLPROC;iniciaIP;0;1;
  CPAR;1;IP;
  CONFIG;1;0;1;
    EXPP;COPY;4;VA;1;1;1;4;VB;1;1;0;0;0;0;0;
  CALLPROC;imprimevetor;0;3;
  CPAR;1;VA;
  CPAR;1;screen;
  CPAR;1;keyboard;
```

```
  04111

#USE imprime
#USE userio
#USE dados
#USE pfdma
[4]REAL32 VA:
[4]REAL32 VB:
[4]INT    IP:
SEQ
  iniciaVB ( VB)
  iniciaIP ( IP)
  [4]INT gc.cilarge:
  INT    gc.value:
  SEQ
    gc.value := 0
    SEQ i0   = 0 FOR 4
      SEQ
        gc.cilarge[i0  ] := gc.value
        gc.value := gc.value + (1)
    PAR j = 0 FOR 4
      VA[IP[gc.cilarge[j]]] := VB[gc.cilarge[j]]
  imprimevetor ( VA, screen, keyboard)
```

```
      SUBROUTINE S4113(a,b,ip,n)
      INTEGER n,ip(*)
      REAL a(*),b(*)
      DO 570 i = 1,n
         a(ip(i)) = b(ip(i))
570   CONTINUE
      write(6,12) (a(i),i=1,n)
12    format (e12.6)
      RETURN
      END
```

```
PROGRAM S4113;

CONST
  N = 100;
VAR
  a,
  b  :  ARRAY[1:100] OF REAL;
  ip :  ARRAY[1:100] OF INTEGER;
INDEX
  ID : INTEGER;
BEGIN
  USING ID := 1:N DO
    a[ip[ID]] := b[ip[ID]];
END.
```

s4113.tsr

```
:30004;1;1;4;
:30000;1;1;4;
::
:0;0;0;1;0;4;
:1;0;IP;0;4;
::
DEC;VA;0;
DEC;VB;0;
DEC;IP;1;
BLOCK;12;
  CALLPROC;iniciaVB;0;1;
  CPAR;1;VB;
  CALLPROC;iniciaIP;0;1;
  CPAR;1;IP;
  CONFIG;1;0;1;
    EXPP;COPY;4;VA;1;1;1;4;VB;1;1;1;0;0;0;0;
  CALLPROC;imprimevetor;0;3;
  CPAR;1;VA;
  CPAR;1;screen;
  CPAR;1;keyboard;
```

```
 04113

#USE imprime
#USE userio
#USE dados
#USE pfdma
[4]REAL32 VA:
[4]REAL32 VB:
[4]INT    IP:
SEQ
  iniciaVB ( VB)
  iniciaIP ( IP)
  [4]INT gc.cilarge:
  INT    gc.value:
  SEQ
    gc.value := 0
    SEQ i0   = 0 FOR 4
      SEQ
        gc.cilarge[i0  ] := gc.value
        gc.value := gc.value + (1)
    PAR j = 0 FOR 4
      VA[IP[gc.cilarge[j]]] := VB[IP[gc.cilarge[j]]]
  imprimevetor ( VA, screen, keyboard)
```

```
        SUBROUTINE S425(n)
        DIMENSION b(100)
C       COMMON /comm1 / cc1(100),cc2(100)
        COMMON /comm1 / cc1(100),cc2(100),cc3(100),cc4(100,100)
        DIMENSION eqv1(100),eqv2(90)
        EQUIVALENCE (eqv1(1),cc2(1))
        EQUIVALENCE (eqv2(1),cc1(1))
        DO 920 i = 1,85
           eqv1(i) = cc1(i+8) + b(i)
920     CONTINUE
        WRITE(6,100) (eqv2(i),i=1,n)
100     FORMAT (e12.6)
        RETURN
        END
```

```
PROGRAM S425;

CONST
  N = 1000;
VAR
   b,
 cc1  :  ARRAY[1:N] OF REAL;
   i  :  INTEGER;
INDEX
  IS : INTEGER;
BEGIN
  FOR i:= 1 TO 11 DO
    USING IS := i:i+4 DO
      cc1[IS] := cc1[IS SHIFT 4] + b[IS];
  cc1[89] := cc1[93] + b[85];
END.
```

s425.tsr

```
:30004;1;1;200;
::
:1;I;0;1;0;4;
:1;I+4;0;1;0;4;
::
DEC;VB;0;
DEC;VC;0;
DEC;I;30000;
BLOCK;7;
  EXP;COPY;1;I;0;0;0;0;
  FOR;0;11;4;
    BLOCK;3;
      CONFIG;1;0;1;
        EXPP;ADD;4;VC;1;1;0;4;VC;1;1;1;4;VB;1;1;0;0;0;
      EXP;ADD;1;I;1;I;0;1;
  EXP;ADD;2;VC;1;0;88;2;VC;1;0;92;2;VB;1;0;84;
```

```
 o425.tsr
#USE imprime
#USE userio
#USE dados
#USE pfdma
[200]REAL32 VB:
[200]REAL32 VC:
INT    I:
SEQ
  I := 0
  SEQ i0   = 0 FOR 11
    SEQ
      [4]INT gc.cilarge:
      INT    gc.value:
      SEQ
        gc.value := I
        SEQ i1   = 0 FOR 4
          SEQ
            gc.cilarge[i1  ] := gc.value
            gc.value := gc.value + (1)
        [4]INT gc.cidep0:
        INT    gc.value:
        SEQ
          gc.value := I+4
          SEQ i1   = 0 FOR 4
            SEQ
              gc.cidep0[i1  ] := gc.value
              gc.value := gc.value + (1)
          PAR j = 0 FOR 4
            VC[gc.cilarge[j]] := VC[gc.cidep0[j]]
                                 + VB[gc.cilarge[j]]
      I := I + 1
  VC[88] := VC[92] + VB[84]
```

```
      SUBROUTINE S442(a,c,n)
      DIMENSION a(1000),b(1000),c(1000)
      DIMENSION d(1000),aa(200),ii(1000)
      DO 810 i = 1,n
         GOTO (815,820,830,840) ii(i)
815      c(i) = aa(i)
         GOTO 850
820      c(i) = a(i)
         GOTO 850
830      c(i) = b(i)
         GOTO 850
840      c(i) = d(i)
850      CONTINUE
810   CONTINUE
      write(6,100) (c(i),i=1,n)
100   FORMAT (e12.6)
      RETURN
      END
```

```
PROGRAM S442;

CONST
  N = 1000;
VAR
  a,
  b,
  c,
  d,
  ii  :  ARRAY[1:N] OF REAL;
  aa  :  ARRAY[1:200] OF REAL;
INDEX
  ID : INTEGER;
BEGIN
  USING ID := 1:100 DO
    CASE d[ID] OF
      1 : c[ID] := aa[ID];
      2 : c[ID] := a[ID];
      3 : c[ID] := b[ID];
      4 : c[ID] := d[ID];
    END;
END.
```

s442.tsr

```
:30004;1;1;4;
:30007;1;1;4;
:30000;1;1;4;
::
:0;0;0;1;0;4;
::
DEC;VA;0;
DEC;VB;0;
DEC;VC;0;
DEC;VD;0;
DEC;VE;0;
DEC;MASC;1;
DEC;IP;2;
DEC;TESTE;30007;
BLOCK;38;
  CALLPROC;iniciaVA;0;1;
  CPAR;1;VA;
  CALLPROC;iniciaVB;0;1;
  CPAR;1;VB;
  CALLPROC;iniciaVC;0;1;
  CPAR;1;VC;
  CALLPROC;iniciaVD;0;1;
  CPAR;1;VD;
  CALLPROC;iniciaVE;0;1;
  CPAR;1;VE;
  CALLPROC;iniciaIP;0;1;
  CPAR;1;IP;
  CONFIG;1;0;21;
    BLOCK;20;
      EXPP;EQUAL;3;MASC;1;1;0;3;IP;1;1;0;0;1(INT);0;0;
      CALLFUNC;ifanyD1;1;TESTE;1;
      CPAR;2;MASC;
      IF;1;TESTE;1;0;
        EXPP;COPY;3;VC;1;1;0;3;VE;1;1;0;0;0;3;MASC;1;1;0;
      EXPP;EQUAL;3;MASC;1;1;0;3;IP;1;1;0;0;2(INT);0;0;
      CALLFUNC;ifanyD1;1;TESTE;1;
      CPAR;2;MASC;
      IF;1;TESTE;1;0;
        EXPP;COPY;3;VC;1;1;0;3;VA;1;1;0;0;0;3;MASC;1;1;0;
      EXPP;EQUAL;3;MASC;1;1;0;3;IP;1;1;0;0;3(INT);0;0;
      CALLFUNC;ifanyD1;1;TESTE;1;
      CPAR;2;MASC;
      IF;1;TESTE;1;0;
        EXPP;COPY;3;VC;1;1;0;3;VB;1;1;0;0;0;3;MASC;1;1;0;
      EXPP;EQUAL;3;MASC;1;1;0;3;IP;1;1;0;0;4(INT);0;0;
      CALLFUNC;ifanyD1;1;TESTE;1;
```

```
    CPAR;2;MASC;
    IF;1;TESTE;1;0;
      EXPP;COPY;3;VC;1;1;0;3;VD;1;1;0;0;0;3;MASC;1;1;0;
CALLPROC;imprimevetor;0;3;
CPAR;1;VC;
CPAR;1;screen;
CPAR;1;keyboard;
```

```
  0442

#USE imprime
#USE userio
#USE dados
#USE pfdma
[4]REAL32 VA:
[4]REAL32 VB:
[4]REAL32 VC:
[4]REAL32 VD:
[4]REAL32 VE:
[4]BOOL   MASC:
[4]INT    IP:
BOOL   TESTE:
SEQ
  iniciaVA ( VA)
  iniciaVB ( VB)
  iniciaVC ( VC)
  iniciaVD ( VD)
  iniciaVE ( VE)
  iniciaIP ( IP)
  [4]INT gc.cilarge:
  INT    gc.value:
  SEQ
    gc.value := 0
    SEQ i0   = 0 FOR 4
      SEQ
        gc.cilarge[i0  ] := gc.value
        gc.value := gc.value + (1)
    SEQ
      PAR j = 0 FOR 4
        MASC[gc.cilarge[j]] := IP[gc.cilarge[j]] = 1(INT)
      TESTE := ifanyD1 ( MASC)
      IF
        TESTE
          PAR j = 0 FOR 4
            IF
              MASC[j]
                VC[gc.cilarge[j]] := VE[gc.cilarge[j]]
              TRUE
                SKIP
        TRUE
          SKIP
      PAR j = 0 FOR 4
        MASC[gc.cilarge[j]] := IP[gc.cilarge[j]] = 2(INT)
      TESTE := ifanyD1 ( MASC)
      IF
        TESTE
```

```
        PAR j = 0 FOR 4
          IF
            MASC[j]
              VC[gc.cilarge[j]] := VA[gc.cilarge[j]]
            TRUE
              SKIP
      TRUE
        SKIP
    PAR j = 0 FOR 4
      MASC[gc.cilarge[j]] := IP[gc.cilarge[j]] = 3(INT)
    TESTE := ifanyD1 ( MASC)
    IF
      TESTE
        PAR j = 0 FOR 4
          IF
            MASC[j]
              VC[gc.cilarge[j]] := VB[gc.cilarge[j]]
            TRUE
              SKIP
      TRUE
        SKIP
    PAR j = 0 FOR 4
      MASC[gc.cilarge[j]] := IP[gc.cilarge[j]] = 4(INT)
    TESTE := ifanyD1 ( MASC)
    IF
      TESTE
        PAR j = 0 FOR 4
          IF
            MASC[j]
              VC[gc.cilarge[j]] := VD[gc.cilarge[j]]
            TRUE
              SKIP
      TRUE
        SKIP
imprimevetor ( VC, screen, keyboard)
```

```
        SUBROUTINE S481(a,b,c,n)
        REAL a(1000),b(1000),c(1000)
        DO 110 i = 1,n
             if (a(i).LT.0) stop 'stop 1'
             b(i) = c(i)
110     CONTINUE
        write(6,100) (b(i),i=1,n)
100     FORMAT (e12.6)
        RETURN
        END
```

```
PROGRAM S481;

CONST
  N = 100;
VAR
  a,
  b,
  c  :  ARRAY[1:N] OF REAL;
  i  :  INTEGER;
BEGIN
  i := 1;
  WHILE (a[i] >= 0) and (i <= N)  DO
    BEGIN
      b[i] := c[i];
      i := i + 1;
    END;
END.
```

```
s481.tsr

:30004;1;0;4;
::
::
DEC;VA;0;
DEC;VB;0;
DEC;VC;0;
DEC;I;30000;
DEC;TESTE;30007;
DEC;TESTE1;30007;
DEC;TESTE2;30007;
BLOCK;14;
  CALLPROC;iniciaVA;0;1;
  CPAR;1;VA;
  CALLPROC;iniciaVC;0;1;
  CPAR;1;VC;
  EXP;COPY;1;I;0;0;0;0;
  WHILE;1;TESTE;3;6;
  EXP;LESS;1;TESTE2;1;I;0;4(INT);
  EXP;GRTEQ;1;TESTE1;2;VA;1;1;I;0;0.0(REAL32);
  EXP;AND;1;TESTE;1;TESTE1;1;TESTE2;
    BLOCK;2;
      EXP;COPY;2;VB;1;1;I;2;VC;1;1;I;0;0;
      EXP;ADD;1;I;1;I;0;1;
  CALLPROC;imprimevetor;0;3;
  CPAR;1;VB;
  CPAR;1;screen;
  CPAR;1;keyboard;


æ
```

```
  0481

#USE imprime
#USE userio
#USE dados
#USE pfdma
[4]REAL32 VA:
[4]REAL32 VB:
[4]REAL32 VC:
INT    I:
BOOL   TESTE:
BOOL   TESTE1:
BOOL   TESTE2:
SEQ
  iniciaVA ( VA)
  iniciaVC ( VC)
  I := 0
  TESTE2 := I < 3(INT)
  TESTE1 := VA[I] >= 0.0(REAL32)
  TESTE := TESTE1 AND TESTE2
  WHILE TESTE
    SEQ
      SEQ
        VB[I] := VC[I]
        I := I + 1
      TESTE2 := I < 3(INT)
      TESTE1 := VA[I] >= 0.0(REAL32)
      TESTE := TESTE1 AND TESTE2
  imprimevetor ( VB, screen, keyboard)
```

æ

```
      SUBROUTINE S482(a,b,c,n)
      INTEGER n
      REAL a(*),b(*),c(*)
      DO 520 i = 1,n
           c(i) = a(i)
           if (a(i).LT.b(i)) GOTO 521
520   CONTINUE
      RETURN
521      CONTINUE
      write(6,12) (c(i),i=1,n)
12    format (e12.6)
      RETURN
      END
```

```
PROGRAM S482;

CONST
  N = 100;
VAR
  a,
  b,
  c  :  ARRAY[1:N] OF REAL;
  i  :  INTEGER;
BEGIN
  i := 1;
  REPEAT
    c[i] := a[i];
    i := i + 1;
  UNTIL (a[i] > b[i]) and (i > N);
END.
```

s482.tsr

```
:30004;1;0;4;
::
::
DEC;VA;0;
DEC;VB;0;
DEC;VC;0;
DEC;I;30000;
DEC;TESTE;30007;
DEC;TESTE1;30007;
DEC;TESTE2;30007;
BLOCK;16;
  CALLPROC;iniciaVA;0;1;
  CPAR;1;VA;
  CALLPROC;iniciaVB;0;1;
  CPAR;1;VB;
  EXP;COPY;1;I;0;0;0;0;
  REPEAT;1;TESTE;6;
    BLOCK;2;
      EXP;COPY;2;VC;1;1;I;2;VA;1;1;I;0;0;
      EXP;ADD;1;I;1;I;0;1;
  EXP;GREAT;1;TESTE1;2;VA;1;1;(I-1);2;VB;1;1;(I-1);
  EXP;GREAT;1;TESTE2;1;I;0;3(INT);
  EXP;AND;1;TESTE;1;TESTE1;1;TESTE2;
  CALLPROC;imprimevetor;0;3;
  CPAR;1;VC;
  CPAR;1;screen;
  CPAR;1;keyboard;
```

æ

```
  0482

#USE imprime
#USE userio
#USE dados
#USE pfdma
[4]REAL32 VA:
[4]REAL32 VB:
[4]REAL32 VC:
INT    I:
BOOL   TESTE:
BOOL   TESTE1:
BOOL   TESTE2:
SEQ
  iniciaVA ( VA)
  iniciaVB ( VB)
  I := 0
  SEQ
    TESTE := TRUE
    WHILE TESTE
      SEQ
        SEQ
          VC[I] := VA[I]
          I := I + 1
        TESTE1 := VA[(I-1)] > VB[(I-1)]
        TESTE2 := I > 3(INT)
        TESTE := TESTE1 AND TESTE2
    imprimevetor ( VC, screen, keyboard)

æ
```