**AMBIENTE PARA PROGRAMAÇÃO PARALELA NO PROCESSADOR i860**

Cristiana Bentes Seidel

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

Aprovada por:

Prof. Édil Severiano Tavares Fernandes, Ph. D.
(presidente)

Prof. Valmir Carneiro Barbosa, Ph. D.

Prof. Cláudio Luís Amorim, Ph. D.

Prof. Ageu Cavalcante Pacheco Junior, Ph. D.

RIO DE JANEIRO, RJ - BRASIL
JULHO DE 1991

SEIDEL, CRISTIANA BENTES

Ambiente para Programação Paralela no Processador i860 [Rio de Janeiro] 1991

XI, 117 p., 29.7 cm, (COPPE/UFRJ, M. Sc., ENGENHARIA DE SISTEMAS E COMPUTAÇÃO, 1991)

TESE – Universidade Federal do Rio de Janeiro, COPPE

1 – Sistemas Operacionais 2 – Processamento Paralelo

I. COPPE/UFRJ II. Título(Série).

*A Paulo e a meus pais*

# Agradecimentos

Ao professor Edil S. T Fernandes pelo interesse e dedicação com que orientou este trabalho.

Ao professor Valmir Carneiro Barbosa pela contribuição que deu ao trabalho na ausência do professor Edil.

Ao grupo de processamento paralelo da COPPE/UFRJ, Ricardo Citro, Eliseu, Alberto, José Queiroz, Juliana, Júlio, Delfim, Emerson, Edu e Adilson pela construção da máquina e pelo auxílio constante na utilização da mesma.

A Gilberto Yamashiro pelo desenvolvimento do montador e pela ajuda na utilização do i860.

A meus pais Vera e José Flávio a quem devo esta realização, pelo carinho e pelo apoio em todos os momentos da minha vida.

E a meu marido Paulo um agradecimento muito especial para quem compartilhou dos bons e dos maus momentos, transmitindo todo amor, amizade e compreensão de que necessitei.

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

Ambiente para Programação Paralela no Processador i860

Cristiana Bentes Seidel

Julho de 1991

Orientador: Edil Severiano Tavares Fernandes
Programa: Engenharia de Sistemas e Computação

Essa tese descreve o desenvolvimento do Núcleo de um Sistema Operacional para o processador i860 da Intel. O Núcleo tem como objetivo principal tornar homogêneo o ambiente para programação paralela de cada nó de processamento de uma arquitetura com topologia hipercúbica. Essa máquina, desenvolvida na COPPE-UFRJ, inclui em cada um de seus nós um processador T-800 da Inmos e um i860. O ambiente de programação paralela desenvolvido reproduz no i860 as facilidades para processamento paralelo implementadas diretamente pelo nível de $\mu$-código do T-800.

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

Environment for Parallel Programming on i860 Processor

Cristiana Bentes Seidel

July, 1991

Thesis Supervisor: Edil Severiano Tavares Fernandes

Department: Programa de Engenharia de Sistemas e Computação

This thesis describes the development of an Operating System's Kernel for the Intel i860 processor. The main goal of the Kernal is to create a homogeneous environment suitable for parallel programming in the processing nodes of a hypercube machine wich was developed at COPPE-UFRJ. Each processing node of this hypercube includes an Inmos T -800 and an Intel i860 processors. The environment developed in the i860 provides the same facilities as those realized by the T-800's $\mu$-code.

# Índice

# Lista de Figuras

# Lista de Tabelas

# Capítulo I

# Introdução

A necessidade de resolver classes de problemas num curto intervalo de tempo constitui um grande desafio para a Ciência da Computação. A medida que aumenta a complexidade dos programas de aplicação, precisamos de máquinas mais rápidas. Avanços tecnológicos proporcionaram um aumento expressivo no desempenho dos computadores, entretanto, a tecnologia esbarra em um obstáculo: a velocidade da luz. Como seria possível, dada esta limitação, reduzir ainda mais o tempo de execução dos programas de aplicação? A resposta a essa pergunta está no processamento paralelo.

O processamento paralelo explora a seguinte estratégia: se não podemos aumentar a desempenho individual de um componente do sistema, o desempenho do sistema como um todo pode ser melhorado substancialmente se utilizarmos diversos componentes operando em paralelo.

Em sistemas de computação, o processamento paralelo pode estar presente em diversos níveis. Por exemplo, é possível projetar um processador com diversas unidades funcionais operando em paralelo. Em um nível mais alto, podemos ter uma arquitetura constituída de diversos processadores interconectados que podem executar, simultaneamente, diferentes partes de uma mesma tarefa. Este tipo de arquitetura é denominado de **multiprocessador**.

Multiprocessadores podem ser diferenciados, primariamente, pela forma de conexão entre os elementos de processamento e a memória, sendo classificados

como forte ou fracamente acoplados.

Máquinas fortemente acopladas são caracterizadas pela existência de uma memória global, ou seja, a memória é compartilhada por todos os elementos de processamento. Nessas máquinas, a comunicação entre os elementos de processamento pode ser levada a cabo através de primitivas do tipo **P** e **V**.

Nas máquinas fracamente acopladas a memória encontra-se distribuída pelos diversos elementos de processamento, cada elemento possui o seu segmento de memória privativo e troca informações com os outros elementos através de mensagens. A conexão dos elementos de processamento é realizada por ligações (*links*) dedicadas, por barramentos globais ou por outras estruturas formando uma rede de interconexão.

Dentre as diversas topologias de rede de interconexão podemos destacar a topologia hipercúbica [9,14,15] que é capaz de mapear inúmeras estruturas como: grade, anel e árvore.

Essa forma de interconexão é caracterizada pela presença de $N = 2^d$ elementos de processamento, onde cada elemento é conectado a outros $d$ elementos adjacentes. Os elementos de processamento são vistos como nós, ou vértices, de um hipercubo em um espaço $d$-dimensional, a conexão dos nós, ou *links*, são as arestas do cubo. A Figura I.1 ilustra algumas topologias hipercúbicas, onde os círculos representam os elementos de processamento e as arestas os *links*; $d$ representa a dimensão do hipercubo.

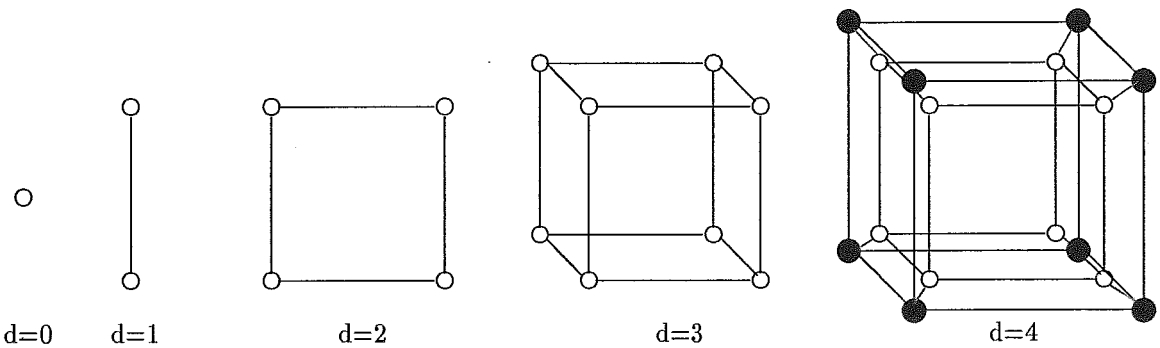



Figura I.1: Definição recursiva da topologia hipercúbica

Como parte do projeto de pesquisa em processamento paralelo, foi desenvolvida, pelo grupo de processamento paralelo da COPPE/UFRJ, uma máquina paralela, denominada de NCP-1. É um multiprocessador do tipo MIMD [11] que incorpora diversos nós de processamento interconectados segundo uma topologia hipercúbica (sendo possível a reconfiguração para outras topologias). A Figura I.2 mostra a topologia do protótipo da máquina que está em funcionamento.

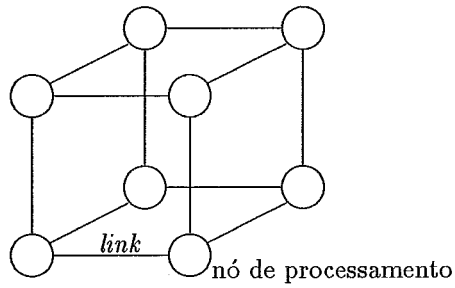



Figura I.2: Topologia da máquina hospedeira

A memória do sistema encontra-se distribuída através da rede e a comunicação entre os nós se dá diretamente através dos *links* que formam a rede de interconexão.

Cada nó de processamento pode ser considerado como um computador. Ele é constituído de dois processadores: um processador i860 da Intel [19,20] e um processador T-800 da família Transputer da Inmos [35,36]. A Figura I.3 mostra o diagrama do nó de processamento.

Conforme ilustrado na Figura I.3, existem três segmentos de memória: um segmento de memória dinâmica local do T-800, um segmento de memória estática compartilhado pelos dois processadores e um segmento de memória dinâmica em que uma parte é compartilhada pelos dois processadores e a outra é memória local do i860. A comunicação entre os processadores se dá através dos dois segmentos de memória compartilhada; a memória estática (de acesso mais rápido) pode ser utilizada para agilizar a troca de mensagens.

As arestas do hipercubo correspondem aos *links* do *Cross-Bar-Switch*. É através desse módulo que podemos configurar a rede de interconexão dos nós de processamento. As mensagens procedentes de outros nós são passadas do *Cross-*

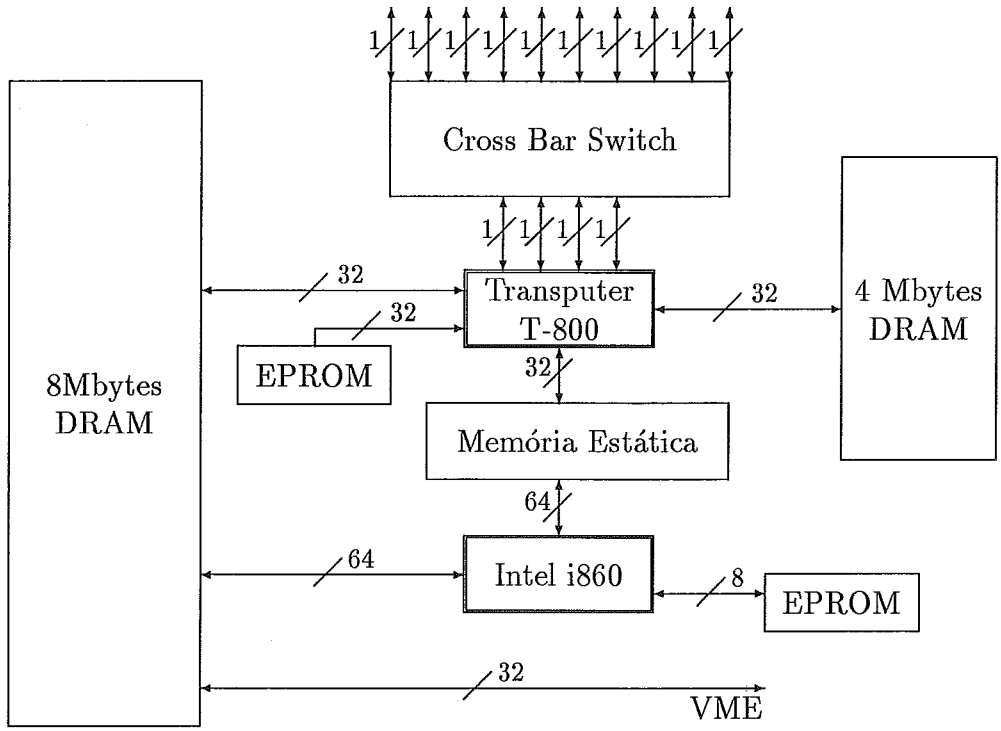



Figura I.3: Nó de processamento da máquina hospedeira

*Bar-Switch* para o Transputer através de um dos quatro *links* desse processador.

Além dessa rede de interconexão (controlada diretamente pelos processadores T-800), a arquitetura possui uma outra via alternativa conectando todos os nós de processamento. Essa via de comunicação é um barramento VME interconectando os processadores i860 do sistema.

Nessa estrutura, cada nó de processamento pode executar processos provenientes de um mesmo programa ou não. Por exemplo, partes de um mesmo programa podem estar sendo executadas por uma coleção de processadores T-800 e i860.

Como geralmente ocorre com as máquinas paralelas, um dos maiores problemas para explorar adequadamente os recursos do NCP-1 reside na dificuldade de programá-lo [12]. A tarefa de desenvolvimento de programas (explorando o poder computacional da máquina subjacente) será bastante facilitada se dispusermos de um ambiente adequado à programação paralela.

O objetivo fundamental desse trabalho é o de tornar homogêneo o

ambiente de programação paralela em cada um dos nós do hipercubo. Para tal, especificamos e implementamos o Núcleo de um Sistema Operacional que executa no processador i860 e provê, além das primitivas para comunicação e sincronização de processos em execução nos diversos processadores do sistema, procedimentos para reproduzir no processador i860 as facilidades para processamento paralelo que são diretamente realizadas pelo nível de $\mu$-código do T-800. Assim, fica facilitada a tarefa de migração de processos entre os dois tipos de processadores, bem como a execução de processos cooperantes.

Essa tese está organizada em seis capítulos. No Capítulo II é apresentado o processador i860 da Intel no qual o Núcleo do Sistema Operacional irá executar.

O Capítulo III descreve o ambiente que o Núcleo deverá reproduzir. Serão vistas as principais características da linguagem Occam e como elas são implementadas no processador T-800 da Inmos.

No Capítulo IV descrevemos a estrutura geral do Núcleo, destacando as facilidades que ele oferece ao usuário, como é realizado o escalonamento de processos e como o Núcleo pode ser ativado.

O Capítulo V está diretamente ligado à implementação do Núcleo no processador i860, nele é encontrada a estrutura de dados utilizada, e todas as primitivas que compõem o Núcleo. Para cada primitiva apresenta-se o respectivo algoritmo. No final deste Capítulo são expostas as medidas de desempenho das primitivas.

No Capítulo VI apresentamos nossas conclusões do trabalho e mostramos algumas possíveis evoluções futuras.

# Capítulo II

# O Microprocessador i860

O microprocessador i860 da Intel [19,20] é um processador RISC de 64 bits, com arquitetura paralela utilizando técnicas de *pipeline*. Sua arquitetura inclue em um único circuito integrado operações com inteiros, operações com ponto-flutuante, operações gráficas, suporte para gerenciamento de memória e duas memórias *caches*, uma para dados e outra para instruções. A incorporação de todas as unidades funcionais em um único *chip* reduz o *overhead* de comunicação entre *chips*.

## II.1 Arquitetura

A arquitetura do i860 está ilustrada na Figura II.1. A CPU i860 contém as seguintes unidades funcionais:

. Unidade central para operações em inteiros

. Unidade de gerenciamento de memória

. Unidade de controle de ponto-flutuante

. Unidade de soma de ponto-flutuante

. Unidade de multiplicação de ponto-flutuante

. Unidade gráfica

. Memória *cache* de instruções

. Memória *cache* de dados

. Unidade de controle do barramento e das memórias *caches*

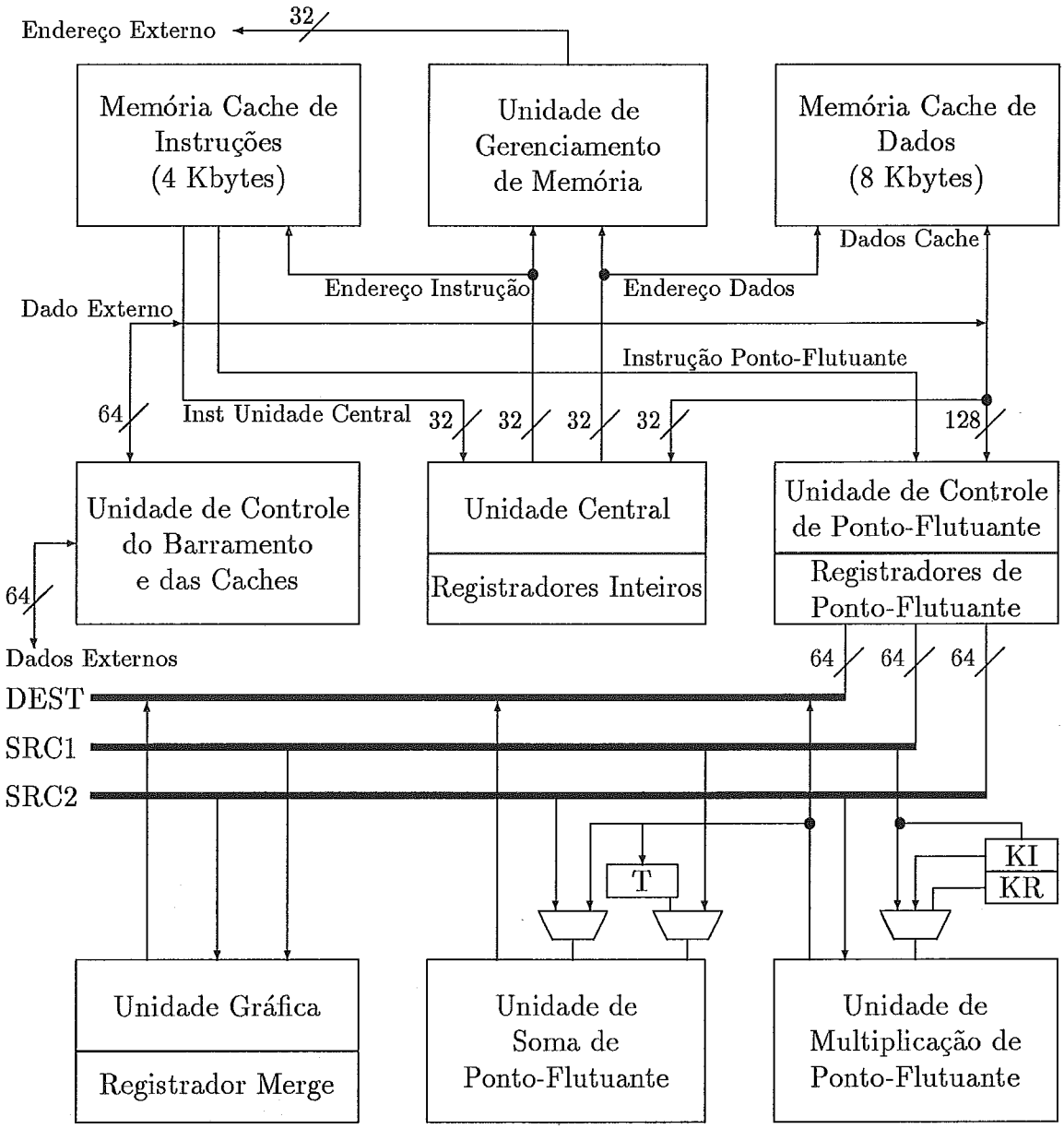



Figura II.1: O processador i860

## II.2 Tipos de dados

O microprocessador i860 provê operações atuando sobre dados inteiros e em ponto-flutuante. Basicamente, operações com inteiros empregam operandos de 32 bits,

embora existam algumas operações que manipulam operandos de 64 bits. As instruções do tipo *load* e *store* podem referenciar operandos de 8, 16, 32, 64 ou 128 bits. Operações com ponto-flutuante manipulam números de 32 ou 64 bits. Instruções gráficas manipulam *arrays* de *pixels* constituídos de 8, 16 ou 32 bits.

Um inteiro é representado na forma de complemento a dois e os números reais podem estar representados em precisão simples ou em precisão dupla, de acordo com o padrão IEEE 754.

## II.3 Memória Local

Conforme mostra a Figura II.2, o microprocessador i860 possui os seguintes registradores:

a. Um "arquivo" de registradores inteiros : são trinta e dois registradores de 32 bits (r0 a r31) utilizados para cálculo de endereços e em operações escalares. O registrador **r0** retorna zero quando lido, independentemente do que for armazenado nele;

b. Um "arquivo" de registradores de ponto-flutuante : são trinta e dois registradores de 32 bits, denominados de **f0** a **f31** e utilizados nas operações em ponto-flutuante. Estes registradores podem ser operados individualmente como trinta e dois registradores de 32 bits, como dezesseis registradores de 64 bits, ou como oito registradores de 128 bits. Os registradores **f0** e **f1** retornam zero quando lidos;

c. Seis registradores de controle (PSR, EPSR, DB, FIR, DIRBASE e FSR): estes registradores são acessíveis somente por instruções do tipo *load* e *store*. O registrador PSR(*Processor Status Register*) contém informações referentes ao estado do processo corrente. O registrador EPSR(*Extended Processor Status Register*) contém informações adicionais do estado do processo corrente, além de outras informações, como por exemplo, tipo do processador e tamanho da memória *cache*. O registrador DB(*Data Breakpoint*) é usado na geração de

um *trap* quando o processador acessa um operando cujo endereço está armazenado nesse registrador, permitindo a implementação de facilidades para a monitoração/depuração de programas. O registrador FIR(*Fault Instruction Register*) armazena o endereço da instrução que causou o *trap*. O registrador DIRBASE(*DIRectory BASE*) contém informações de controle para a *cache*, para a tradução de endereços e para as opções do barramento. Finalmente o registrador FSR(*Floating-point Status Register*) contém informações referentes ao estado do processo corrente quando ocorre um *trap* por erro em uma operação de ponto-flutuante e sobre os modos de arredondamento;

d. Quatro registradores de propósito especial (KR, KI, T e Merge). Os registradores KR, KI e T são utilizados por operações duais, conforme será visto posteriormente. O registrador Merge é utilizado pela unidade gráfica.

## II.4 Endereçamento

A memória é acessada em unidades de bytes com um espaço de memória virtual paginado de $2^{32}$ bytes. Instruções e dados podem localizar-se em qualquer posição do espaço de endereçamento desde que respeitado o alinhamento. Dados só podem ser acessados em endereços múltiplos de seu tamanho. Por exemplo, o endereço de um dado de dois bytes deve ser divisível por dois, o de um dado de quatro bytes deve ser divisível por quatro e assim por diante.

Normalmente, dados com mais de um byte são armazenados no formato *little endian*, isto é, o byte menos significativo localizado no endereço de memória mais baixo. Opcionalmente, pode-se selecionar dinamicamente, por *software*, o formato *big endian*, onde o byte mais significativo encontra-se no endereço de memória mais baixo. O acesso ao código é feito sempre por endereçamento *little endian*.

Registradores Inteiros

| 31 | 0 |
|---|---|
| R0 | |
| R1 | |
| R2 | |
| R3 | |
| R4 | |
| R5 | |
| R6 | |
| R7 | |
| R8 | |
| R9 | |
| R10 | |
| R11 | |
| R12 | |
| R13 | |
| R14 | |
| R15 | |
| R16 | |
| R17 | |
| R18 | |
| R19 | |
| R20 | |
| R21 | |
| R22 | |
| R23 | |
| R24 | |
| R25 | |
| R26 | |
| R27 | |
| R28 | |
| R29 | |
| R30 | |
| R31 | |

Registradores de Ponto-Flutuante

| 63 | 32 | 31 | 0 |
|---|---|---|---|
| F1 | | F0 | |
| F3 | | F2 | |
| F5 | | F4 | |
| F7 | | F6 | |
| F9 | | F8 | |
| F11 | | F10 | |
| F13 | | F12 | |
| F15 | | F14 | |
| F17 | | F16 | |
| F19 | | F18 | |
| F21 | | F20 | |
| F23 | | F22 | |
| F25 | | F24 | |
| F27 | | F26 | |
| F29 | | F28 | |
| F31 | | F30 | |

Registradores de Propósito Especial

| | |
|---|---|
| KR | |
| KI | |
| T | |
| Merge | |

Registradores de Controle

| |
|---|
| FSR |
| DB |
| DIRBASE |
| EPSR |
| PSR |
| FIR |

Figura II.2: Conjunto de registradores do processador i860

## II.5 Unidade Central

A unidade central realiza a busca de instruções inteiras e de ponto-flutuante. Ela é responsável pela execução de instruções do tipo: *load*, *store*, operações aritméticas com inteiros, operações de manipulação de bits, desvio do fluxo de controle, transferência de registradores inteiros para registradores de ponto-flutuante, teste e modificação dos registradores de controle, *flush* da *cache* e armazenamento de *pixel* (esta em conjunto com a unidade gráfica). Todas as instruções são de 32 bits. Somente as instruções *load* e *store* operam com a memória, todas as outras instruções operam com registradores. A maioria das instruções permite que o usuário especifique dois registradores fonte e um terceiro registrador destino onde será armazenado o resultado.

A característica principal da unidade central está na sua capacidade de executar a maioria das instruções em um mesmo ciclo de relógio. Ela possui um *pipeline* de quatro estágios: busca, decodificação, execução e escrita.

Algumas otimizações foram incluídas no conjunto de instruções de desvio. As instruções de desvio condicional podem ser executadas com ou sem "atraso". As instruções com atraso permitem que o processador execute a instrução localizada na posição de memória seguinte à instrução de desvio, enquanto o endereço alvo do desvio é buscado.

A unidade central (*Core Unit*) é responsável pela execução das instruções *load* e *store* de ponto-flutuante. Existem dois tipos de *load* para ponto-flutuante: FLD(*Floating-point Load*) e PFLD(*Pipelined Floating-point Load*). A instrução FLD transfere um dado da memória *cache* para um registrador de ponto-flutuante, ou traz o dado da memória, armazenando-o em uma linha da *cache* (se o dado não estiver na *cache*), e carrega-o para um registrador de ponto-flutuante. Um dado de 128 bits pode ser acessado, na *cache* de dados, em um ciclo de relógio. A capacidade de realizar um carregamento de um dado de 128 bits em um ciclo de relógio é fundamental para manter a unidade de ponto-flutuante executando continuamente.

Para acessar um dado muito grande (que não caiba na *cache*), a unidade central usa a instrução PFLD. O carregamento pelo *pipeline* transporta o dado diretamente para o registrador de ponto-flutuante sem colocá-lo na *cache* na ocorrência de uma falha. O *pipeline* de carregamento possui três estágios.

A unidade central provê, também, instruções *lock* e *unlock*. A instrução *lock* é utilizada para travar o barramento até a próxima instrução *unlock*. Entre uma instrução *lock* e a instrução *unlock* correspondente podem ser executadas no máximo 32 instruções.

## II.6 Unidade de Gerenciamento de Memória

A unidade de gerenciamento de memória implementa as características básicas de memória virtual paginada incluindo a proteção de memória a nível de páginas. As técnicas de gerenciamento de memória são as mesmas dos microprocessadores 386 e 486 da Intel.

O processador realiza a tradução de endereços utilizando tabelas de páginas. São utilizados dois níveis hierárquicos de tabelas de páginas. No nível mais alto está o diretório de páginas que contém entradas para 1024 tabelas de páginas. A tabela de páginas, no segundo nível, endereça até 1024 páginas.

Para diminuir o *overhead* causado pelos acessos às tabelas de páginas, o processador guarda as informações de tradução referentes às 64 páginas mais recentemente utilizadas em uma memória associativa chamada de TLB (*Translation Lookaside Buffer*). Esta memória atua como uma *cache* de tradução de endereços. Quando o processador não encontra a informação de tradução para uma determinada página no TLB, ele procura esta informação na tabela de páginas, localizada na memória principal, e atualiza o TLB.

A proteção de memória é atingida através de dois bits da tabela de páginas. São eles W(*Writable*) e U(*User*). O processador realiza a proteção durante a tradução de endereços. Para proteger a área do sistema operacional de acessos indevidos de programas de usuários, o i860 utiliza o conceito de privilégio (ou modo

de operação). Este conceito é implementado atribuindo-se a cada página um dos dois níveis: Nível Supervisor(U = 0) e Nível Usuário(U = 1). Quando o processador está executando no Nível Supervisor, todas as páginas são endereçáveis, mas, quando ele está executando no Nível Usuário, apenas as que pertecem ao Nível Usuário são endereçáveis. No Nível Supervisor, todas as páginas podem ser lidas e apenas as que estiverem com W=0 podem ser alteradas (é importante observar que o i860 possui um modo que, estando no Nível Supervisor, permite que todas as páginas, seja qual for o valor de W, sejam modificadas). No Nível Usuário todas as páginas do processo corrente podem ser lidas e apenas as que estiverem com W=0 podem ser modificadas.

## II.7 Unidade de Ponto-Flutuante

A unidade de ponto-flutuante pode operar com números reais com precisão simples e dupla. Ela possui duas unidades *pipeline* de três estágios, uma para soma outra para multiplicação.

As operações de soma e multiplicação podem ser executadas em dois modos: escalar e *pipelined.* No modo escalar novas operações de ponto-flutuante só podem ser iniciadas quando a operação anterior estiver terminada. Para cada operação de soma (independente da precisão) e para cada multiplicação com precisão simples são necessários três ciclos do relógio. A multiplicação com precisão dupla requer quatro ciclos do relógio.

No modo *pipelined* para cada nova operação o *pipeline* é avançado de um estágio. Desta forma, é possível produzir um resultado a cada ciclo.

No exemplo da Figura II.3, a unidade *pipeline* de soma inicia a soma de f2 com f7. Como esta é a primeira instrução da série e o *pipeline* ainda não está cheio, o resultado obtido é descartado (f0 é sempre zero). Em cada ciclo sucessivo, uma instrução de soma avança o *pipeline.* No quarto ciclo, o resultado da soma da primeira instrução estará disponível no estágio de resultado e é armazenado no registrador especificado pela quarta instrução.

Conteúdo dos Registradores Antes da Soma

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| f2 | 2 | f7 | 7 | f12 | | | |
| f3 | 3 | f8 | 8 | f13 | | | |
| f4 | 4 | f9 | 9 | f14 | | | |
| f5 | 5 | f10 | 10 | f15 | | | |
| f6 | 6 | f11 | 11 | f16 | | | |

| Sequência de Instruções | *Pipeline* de Soma | | | Destino |
|---|---|---|---|---|
| pfadd.ss f2, f7, f0 | 2+7 | - | - | Nenhum |
| pfadd.ss f3, f8, f0 | 3+8 | 2+7 | - | Nenhum |
| pfadd.ss f4, f9, f0 | 4+9 | 3+8 | 2+7 | Nenhum |
| pfadd.ss f5, f10, f12 | 5+10 | 4+9 | 3+8 | f12 ← 9 |
| pfadd.ss f6, f11, f13 | 6+11 | 5+10 | 4+9 | f13 ← 11 |
| pfadd.ss f0, f0, f14 | 0 | 6+11 | 5+10 | f14 ← 13 |

Figura II.3: Exemplo de utilização do *pipeline* de soma

Após a quarta instrução, obtemos um resultado por ciclo. Quando um número muito grande de operações é realizada, o *overhead* para preencher e esvaziar o *pipeline* torna-se desprezível. Caso apenas uma ou duas operações forem realizadas o modo escalar é mais indicado.

Os *pipelines* de soma e multiplicação permitem a sobreposição (*overlap*) de instruções sequenciais quando múltiplas instruções estão sendo executadas, ao mesmo tempo, em vários estágios diferentes. Contudo, o i860 vai além em termos de sobreposição de instruções. Ele possui um modo de processamento, chamado modo dual, que inicia duas instruções ao mesmo tempo: uma para a unidade central (*core unit*) e outra para a unidade de ponto-flutuante. No modo dual uma instrução tem 64 bits, com os 32 bits mais baixos representando a instrução de ponto-flutuante e os 32 bits mais significativos representam a instrução a ser executada na unidade central.

Dentro da unidade de ponto-flutuante existe, ainda, outra forma de sobreposição de instruções: as operações duais. Estas operações permitem que o usuário especifique que uma soma e uma multiplicação sejam executadas simultaneamente. Com apenas uma instrução de 32 bits é iniciada uma operação de soma em conjunto com uma operação de multiplicação. Apesar dessas duas operações

necessitarem de seis operandos, o formato da instrução especifica apenas três. Os outros operandos são providos pela utilização dos registradores de propósito especial KR, KI, e T.

Com o modo dual e as operações duais o processador é capaz de executar três operações ao mesmo tempo.

O exemplo seguinte mostra a aceleração obtida pelo uso do paralelismo na arquitetura.

$$for\ i := 1\ to\ 100\ do$$

$$X[i] := A[i] * B[i] + C$$

O código relativo ao corpo desse loop no modo escalar poderia ser:

$$FMUL \quad A[i],\ B[i],\ temp$$

$$FADD \quad temp,\ C,\ X[i]$$

Cada multiplicação e cada soma requerem três ciclos de relógio. Portanto este loop precisaria de um total de 600 ciclos de relógio.

Utilizando o *pipeline* e as operações duais, podemos obter um novo resultado a cada ciclo, se empregarmos a seguinte codificação:

$$M12TPM \quad A[i],\ B[i],\ X[i-6]$$

A instrução M12TPM realiza uma multiplicação com os dois primei-

ros operandos (A[i] e B[i]) e realiza uma soma com o resultado da multiplicação e o conteúdo do registrador T. O resultado da soma é armazenado no terceiro operando (X[i-6]). Devido aos três estágios do *pipeline* de soma e de multiplicação, o resultado disponível vem da operação que iniciou 6 ciclos de relógio anteriormente. Com a utilização do *pipeline* e a sobreposição das operações de soma e multiplicação o loop que no outro modo requeria 600 ciclos, nesse modo necessita de somente 100 ciclos de relógio (excluindo o preenchimento e esvaziamento do *pipeline*).

Apesar de ser uma máquina superescalar, o i860 é um processador bastante eficiente na realização de operações vetoriais. As características de *pipeline* das operações com a memória principal e das operações de ponto-flutuante permitem que um alto desempenho seja atingido. Dessa forma, um possível uso da conjunção T-800/i860 pode ser: o T-800 solicita ao i860 a execução de rotinas vetoriais.

## II.8 Unidade Gráfica

A unidade gráfica possui uma lógica especial que manipula inteiros de 64 bits além de dar suporte a algoritmos para processamento gráfico em 3-D. Esta unidade pode operar em paralelo com a unidade central. Ela contém o registrador Merge e realiza múltiplas adições em inteiros que são armazenados nos registradores de ponto-flutuante.

## II.9 Memória Cache

Além do TLB (*cache* utilizada para tradução de endereços), o i860 incorpora duas memórias *caches*: uma para o armazenamento de dados e outra para instruções. A memória *cache* de dados tem 8 Kbytes de capacidade e a memória *cache* de instruções 4 Kbytes, ambas possuem uma organização associativa por conjuntos, onde cada conjunto possui dois blocos de 32 bytes.

Ambas as *caches* utilizam endereços virtuais. Desta forma, dentro de um determinado contexto, cada endereço virtual só pode referenciar um endereço físico. Quando do acesso à *cache* de dados o TLB é consultado para fins de proteção

das páginas. Durante a troca de contexto a *cache* de instruções deve ser invalidada pelo programador e a *cache* de dados esvaziada (*flushed*).

A atualização dos dados na memória *cache* é feita segundo a política de *write-back*, ou seja, o processador faz as atualizações na *cache*, e a atualização da memória principal é retardada até que a linha alterada seja descartada da *cache*. Nesse momento, o dado atualizado é "escrito de volta" na memória principal. A utilização desta política impede a propagação de todas as escritas para o barramento externo, reduzindo assim a formação de um gargalo. Uma política mais sofisticada de coerência da *cache* deve ser implementada por *software*.

Cada processador pode armazenar na *cache* código e dados privativos de cada processo, enquanto que os dados compartilhados não podem ser armazenados em memória *cache*. Existe um bit (*cacheable bit*) na entrada da tabela de páginas que permite o controle, por *software*, do que é, ou não, armazenado nas *caches*.

# Capítulo III

# A Linguagem Occam e o Transputer

Visando facilitar a apresentação das características do Núcleo, vamos descrever o ambiente para programação paralela que ele deve reproduzir. Neste capítulo faremos uma breve descrição da linguagem Occam [26] e como algumas de suas construções são implementadas pelo processador T-800 da linha Transputer da Inmos.

## III.1 Uma Máquina Occam

A especificação da linguagem Occam e dos processadores da linha Transputer se deu de uma maneira um tanto peculiar. As características de *hardware* do Transputer foram escolhidas para dar suporte ao modelo Occam de concorrência e o seu conjunto de instruções foi projetado para otimizar a execução de programas Occam [29]. Apesar do Transputer ser capaz de executar código derivado de programas escritos em C, FORTRAN, Pascal ou em *Assembly*, ele é uma "máquina Occam". Em outras palavras, o Transputer incorpora em *hardware* o modelo Occam de programação, com diversas construções da linguagem sendo diretamente realizadas pelas primitivas do processador.

## III.2 A Arquitetura do Transputer

O termo **Transputer** se refere a uma família de dispositivos VLSI programáveis incluindo controladoras de discos, processadores de ponto-flutuante, processadores gráficos, dispositivos de processamento de sinal e processadores de propósito geral de 16 e 32 bits [36]. Estamos interessados na arquitetura do processador T-800 de propósito geral de 32 bits. A Figura III.1 mostra os principais componentes do processador T-800.

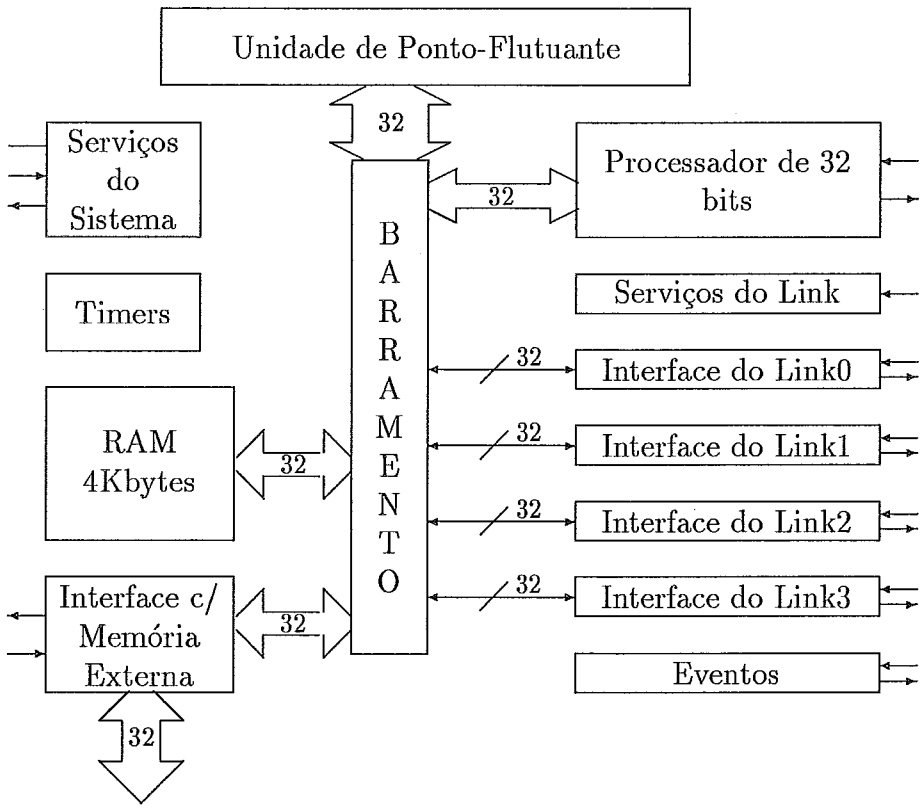



Figura III.1: O processador T-800

O processador contém uma memória interna de 4K bytes e quatro *links* para comunicação direta com outros Transputers. O barramento é conectado a uma memória externa e é implementado de modo que o espaço de endereçamento da memória externa seja uma continuação do da memória interna.

Os quatro *links* estão conectados ao processador central via quatro *interfaces de link*. Essas interfaces podem, idependentemente, manipular os *links* de comunicação (incluindo o acesso direto à memória). Por conseguinte, o T-800 pode se comunicar simultaneamente através dos quatro *links* (em ambas as direções) e realizar processamento interno. A comunicação é feita no modo ponto-a-ponto.

## III.3 A Linguagem Occam

O modelo básico de um programa Occam é o de um conjunto de processos concorrentes que se comunicam. A linguagem Occam foi influenciada pela linguagem CSP (*Communicating Sequential Processes*) de C.A.R. Hoare [17]. A linguagem CSP se baseia na seguinte idéia: computações paralelas podem ser implementadas por um número de máquinas sequenciais que se comunicam umas com as outras através de uma rede de canais.

### III.3.1 Processos Primitivos

Em Occam o elemento fundamental de um programa é o processo. Por definição, um processo inicia, realiza algumas ações e termina. Um programa Occam consiste de vários processos concorrentes. Existem apenas cinco processos primitivos, são eles:

- STOP
- SKIP
- Atribuição
- Entrada
- Saída

### STOP

É um processo que não realiza nenhuma ação e nunca termina. É utilizado comumente para casos de erro, impedindo que os processos seguintes sejam executados.

### SKIP

Também não realiza nenhuma ação. É um processo que está sempre pronto para executar, terminando imediatamente após seu início. Representa um processo nulo e é usado quando nenhuma ação deve ser realizada.

### ATRIBUIÇÃO

A atribuição, em Occam, é identica à atribuição nas linguagens C, Pascal ou BASIC:

$$V := s;$$

onde $V$ é uma variável e $s$ é uma expressão do mesmo tipo de $V$. Após este comando, $V$ conterá o valor de $s$.

### ENTRADA e SAÍDA (Comunicação)

As primitivas de entrada e saída da linguagem implementam um mecanismo troca de mensagens através de um **canal**. Canais são entidades de um

programa Occam que se parecem com variáveis. Um canal tem um identificador, um tipo, e deve ter sido declarado antes do uso. Um canal conecta exatamente dois processos em apenas uma direção, uma conexão bidirecional requer dois canais, um para cada direção. Os comandos de saída (envio de uma mensagem) e entrada (recebimento de uma mensagem) são da forma:

$ch\ !\ s$ – saída

$ch\ ?\ V$ – entrada

onde $ch$ é um canal, $s$ uma expressão, e $V$ uma variável. Esses comandos especificam, respectivamente, "envie o valor da expressão $s$ pelo canal $ch$", e "receba pelo canal $ch$ um valor e o armazene em $V$". Combinando os comandos acima teremos o seguinte efeito:

$V := s$; (estando $V$ em um processo e $s$ em outro)

O compartilhamento de variáveis globais é proibido em Occam. A única maneira possível de se transferir um valor de um processo para outro é através de um canal.

A troca de mensagens é feita de modo bloqueante. Se um canal é utilizado para entrada em um processo $X$ e para saída em um outro processo $Y$, a comunicação só é realizada quando ambos os processos estiverem preparados para trocar uma mensagem, ou seja, o processo $Y$ espera até que $X$ esteja preparado para receber a mensagem, e vice-versa (i.e., o processo $X$ espera até que $Y$ esteja preparado para enviar a mensagem).

Canais são utilizados também para implementar a interação de um programa Occam com o ambiente externo. Como em qualquer linguagem de alto nível, as operações de E/S dependem dos detalhes da implementação da linguagem. Entretanto, o modelo básico de E/S em Occam é muito simples. Um programa que deseja se comunicar com a *VDU* e com o teclado, por exemplo, deve utilizar dois canais. Através de um desses canais o programa envia (executando um comando de saída) caracteres para a *VDU* e, através do outro canal o programa recebe (executando um comando de entrada) os caracteres do teclado.

Existem canais especiais para temporização que retornam o valor do relógio local de tempo real. Estes canais são declarados como *TIMER* e cada processo só pode declarar um canal desse tipo. Sobre canais *TIMER* é possível a executar apenas comandos de entrada. Este canal tem como característica principal o fato de estar sempre preparado para saída (i.e., um processo nunca fica bloqueado quando realiza uma entrada em um canal *TIMER*). Para que um processo fique bloqueado esperando por um intervalo de tempo, a linguagem Occam provê a seguinte construção:

*chrel* ? AFTER $T$

onde *chrel* é um canal *TIMER* declarado previamente, e $T$ um número inteiro. O processo que executar esse comando de entrada sobre *chrel* ficará bloqueado até que o valor do relógio de tempo-real seja maior ou igual ao conteúdo de $T$.

### III.3.2 Construções

Processos primitivos podem ser combinados formando uma construção. Uma construção também é um processo e pode ser usada como componente de outra construção. Além das construções usuais IF, CASE e WHILE, existem outras três classes de construções:

### Sequencial

A definição da construção Sequencial é:

```
SEQ
  P1
  P2
  P3
  ...
```

Os processos componentes (`P1, P2, P3, ...`) da construção são executados sequencialmente. Cada componente é iniciado depois que o outro termina, e a construção termina quando o último componente terminar.

### Paralela

A definição da construção Paralela é:

```
PAR
  P1
  P2
  P3
  ...
```

Os processos componentes (`P1, P2, P3, ...`) da construção são executados concorrentemente. A construção termina quando todos os processos componentes terminarem.

**Alternativa**

A definição da construção Alternativa é:

```
ALT
  G1
    P1
  G2
    P2
  G3
    P3
  ...
```

onde `Gi` é um **guarda** e `Pi` um subprocesso chamado de **lista de comandos**, o conjunto {`Gi`,`Pi`} é chamado de **comando guardado.**

A construção ALT deriva dos comandos guardados de Dijkstra [7]. Ela introduz na linguagem o não-determinismo na comunicação. Através dessa construção, é possível selecionar qual o processo concorrente (dentre diversos) que trocará mensagem com o processo executando a construção.

A sintaxe de um guarda é:

guarda = comando de entrada
| (expressão booleana) & comando de entrada
| (expressão booleana) & SKIP

O guarda é dito **pronto** quando a expressão booleana é verdadeira e o comando de entrada associado pode executar. É importante observar que o processo SKIP está sempre pronto para executar, e que um comando de entrada só estará preparado para a troca de mensagens quando existir um outro processo

preparado para realizar a saída correspondente. O comando de entrada pode representar também a espera por um intervalo tempo, neste caso, o guarda estará **pronto** após o transcurso intervalo de tempo (i.e., o *delay*) especificado.

A execução da construção ALT inclui a escolha de um comando guardado (cujo guarda esteja **pronto**) e a execução da respectiva lista de comandos. Quando não há nenhum guarda **pronto**, o processo ALT fica suspenso até que o estado de um dos guardas seja modificado. Quando há mais de um guarda **pronto** a escolha é arbitrária (i.e., a definição da linguagem não especifica nenhum critério de seleção).

## III.4 A Implementação da Linguagem Occam no Transputer

O Transputer contém facilidades para prover uma implementação eficiente do modelo Occam de concorrência e comunicação. Essas facilidades incluem:

- Suporte para processos sequenciais;
- Suporte para processos concorrentes;
- Suporte para comunicação;
- Suporte para implementação da construção ALT.

### III.4.1 Suporte para Processos Sequenciais

Para execução de processos sequenciais são usados seis registradores do processador:

- ponteiro para área de trabalho - aponta para a área de memória onde as variáveis locais do processo estão armazenadas;
- ponteiro de instrução - aponta para a próxima instrução a ser executada;
- registrador de operandos - utilizado na formação de operandos de instruções (imediatos ou endereços);

- registradores A, B e C - utilizados como uma pilha de avaliação.

### III.4.2 Suporte para Processos Concorrentes

O processador possui um escalonador implementado em $\mu$-código permitindo que um número arbitrário de processos esteja sendo executado, compartilhando o tempo do processador. Com isso, não há necessidade de um núcleo em *software*. A alocação de espaço para processos concorrentes é tarefa do compilador, pois o processador não dá suporte à alocação dinâmica de memória.

Cada processo concorrente possui na memória uma área de trabalho chamada *workspace*. Ela armazena variáveis locais e valores temporários manipulados pelo processo.

Um processo concorrente pode estar em um dos seguintes estados:

**ativo**
- sendo executado
- pronto, esperando para ser executado

**inativo**
- pronto para uma entrada (esperando por uma saída no mesmo canal)
- pronto para uma saída (esperando por uma entrada no mesmo canal)
- esperando por um intervalo de tempo

Os processos ativos são mantidos em duas listas, cada uma delas associada a um nível de prioridade. Uma lista para processos urgentes (nível 0) e outra para os não-urgentes (nível 1). O processador dará prioridade para um processo do nível zero em detrimento de um processo do outro nível.

Um processo urgente é executado até que não seja possível prosseguir porque está esperando por uma entrada (ou saída), ou porque está aguardando o término de um intervalo tempo. Um processo não-urgente pode ter sua execução interrompida por esses dois motivos e ainda porque: terminou sua fatia de tempo, ou um processo urgente tornou-se pronto para executar. Assim, podemos dizer que a política de escalonamento de processos implementada no $\mu$-código do Transputer

é a de filas em dois níveis, sendo que na fila mais prioritária é utilizado a política *FIFO*, e na fila menos prioritária a política *Round-Robin*.

O processador provê algumas instruções de máquina para suportar o modelo Occam de processos. Essas incluem:

**iniciar um processo** (*startp*)

**terminar um processo** (*endp*)

Na realização da construção PAR, instruções *startp* são utilizadas para a criação dos processos concorrentes. A instrução *startp* cria um processo concorrente inserindo o endereço do seu *workspace* no final de uma das listas de escalonamento (conforme sua prioridade).

A instrução *endp* garante o término correto de uma construção PAR utilizando um contador de processos ativos. No *worspace* do processo "pai" é inicializado um contador com o número de processos "filhos". Cada processo "filho", ao terminar, executa a instrução *endp* que se encarregará de decrementar o contador de uma unidade. Para todos os "filhos", menos o último, o contador é diferente de zero e o processo é desescalonado. Para o último, o contador é zero e o processo "pai" continua.

### III.4.3 Suporte para Comunicação

A comunicação entre processos é efetuada por meio de canais. Em Occam ela é ponto-a-ponto, bloqueante e não utiliza *buffers*. Como resultado, não há necessidade de filas de processos, nem de filas ou *buffers* de mensagens.

Um canal entre dois processos executando no mesmo Transputer é implementado por uma palavra na memória. Um canal entre processos que executam em processadores diferentes é implementado pelo *link* entre os dois processadores.

O processador provê as seguintes instruções para dar suporte à troca de mensagens:

**receba mensagem** (*in*)

**envie mensagem** (*out*)

As instruções *in* e *out* usam o endereço do canal para determinar se o canal é interno ou externo. Isto significa que a mesma instrução pode ser usada para canais de *software* ou de *hardware*, permitindo que os processos sejam escritos e compilados sem que se precise saber como o canal está implementado.

Como no modelo Occam, a comunicação só ocorre quando ambos os processos estão **preparados para trocar mensagens**. Consequentemente, o primeiro processo que fica **preparado** deve esperar (inativo) até que o outro também esteja **preparado**.

### Comunicação Interna

A qualquer momento, um canal interno (uma palavra de memória), armazena a identificação de um processo, ou um valor especial chamado "vazio". Os canais são inicializados com "vazio" assim que são declarados.

A troca de mensagens por um determinado canal é realizada da seguinte maneira:

- O primeiro processo a tornar-se preparado para a comunicação tem a sua identificação armazenada no canal e é desescalonado;
- Quando o segundo processo torna-se preparado, a comunicação é efetuada, ou seja, a mensagem é copiada;
- O processo cujo identificador estava armazenada no canal é reescalonado;

- O canal é novamente inicializado com "vazio".

### Comunicação Externa

Quando a mensagem é passada através de um canal externo compete à *interface do link* (elemento processador autônomo) a tarefa de transferir a mensagem e de desescalonar os processos envolvidos na comunicação. Após a troca da mensagem, a *interface do link* requer do processador o reescalonamento dos processos inativos. Isso permite que o processador continue executando um outro processo enquanto é feita a transferência externa da mensagem.

As seguintes atividades são levadas a cabo quando a comunicação é realizada através dos *links*:

- O processo de entrada e o processo de saída são sempre desescalonados;
- A *interface do link* do processador destino fica aguardando;
- A *interface do link* do processador que enviará a mensagem envia um byte e espera por um *acknowledge*;
- Os *links* trocam dados e *acks* até que toda a mensagem tenha sido transferida;
- Após essas atividades os processos são reescalonados.

## III.4.4 Temporização

O Transputer tem um relógio de tempo real cujo *tick* é de 1 $\mu$segundo. Um comando de entrada, utilizando um canal do tipo *TIMER*, pode fazer com que um processo só fique pronto para executar após um determinado intervalo de tempo. Essa instrução requer a especificação de um intervalo de tempo. Se este intervalo já transcorreu (i.e. Valor-do-relógio AFTER Hora-especificada), então a instrução não tem efeito algum. Caso contrário, (Hora-especificada AFTER Valor-do-relógio) o processo é desescalonado, permanecendo nesse estado até que a hora especificada seja alcançada.

### III.4.5 Suporte para a Construção ALT

A construção ALT da linguagem Occam permite que um processo espere por uma entrada de um dentre vários canais, ou espere por um intervalo de tempo. Para tal, são necessárias instruções especiais, uma vez que as instruções normais de entrada causam o desescalonamento do processo até que a saída seja realizada, ou um intervalo de tempo específico seja ultrapassado. As instruções utilizadas são:

**Inicio de um ALT** (*alt*)

**Inicio de um ALT com** *timer* (*talt*)

**Habilita canal** (*enabc*)

**Habilita Tempo** (*enbt*)

**Habilita SKIP** (*enabs*)

**Espera da Alternativa** (*altwt*)

**Espera da Alternativa com** *timer* (*taltwt*)

**Desabilita canal** (*disc*)

**Desabilita Tempo** (*dist*)

**Desabilita SKIP** (*diss*)

**Fim do ALT** (*altend*)

A seleção de um comando guardado é iniciada por uma instrução de início do ALT (*alt* ou *talt*), seguida pela sequência de instruções de habilitação (*enabc*, *enabt* ou *enabs*, uma para cada guarda), uma instrução *altwt*, uma sequência de instruções de inabilitação (*disc*, *dist* ou *diss*, uma para cada guarda) e uma instrução *altend*.

As instruções de habilitação são usadas para definir os guardas que estão prontos. A instrução *altwt* ou *taltwt* é utilizada para desescalonar o processo se nenhum dos guardas estiverem prontos. Nesse caso, o processo será reescalonado quando algum dos guardas tornar-se pronto. Para cada instrução de habilitação há uma instrução de inabilitação associada. As instruções de inabilitação são usadas para determinar, dentre os comandos guardados que possuem o guarda pronto, qual deve executar e para inabilitar as comunicações dos guardas associados a comandos guardados que não foram escolhidos.

Usando as instruções de habilitação e inabilitação, uma construção ALT do tipo:

```
ALT
  G1
    P1
  ...

  Gn
    Pn
```

Pode ser implementada como:

```
alt
enable(G1);...; enable(Gn)
altwt; (ou taltwt se algum Gi for um guarda de temporizacao)
disable(G1);...; disable(Gn)
altend;
```

As instruções acima são utilizadas apenas para a seleção do comando guardado que será executado. A execução da lista de comandos (`P1`, ..., `Pn`) do comando guardado escolhido é realizada após a instrução `altend`, o controle é desviado para o subprocesso correspondente.

A sequência em que os comandos guardados são habilitados não é importante, mas a sequência na qual eles são inabilitados determina a prioridade dos comandos guardados. O primeiro comando guardado pronto, a ser inabilitado, é selecionado.

# Capítulo IV

# Características do Núcleo

Neste capítulo descreveremos os objetivos de projeto do Núcleo; os serviços oferecidos ao usuário; o seu fluxo de controle (mostrando como podemos ativá-lo); como ele realiza o escalonamento de processos; e, por fim, veremos como é feita a interface com o usuário, ou melhor, quais as primitivas oferecidas ao usuário.

## IV.1 Objetivos

O objetivo fundamental do Núcleo é o de prover um meio ambiente no qual os processos concorrentes possam existir [10]. Isso implica em manipular as interrupções, alocar CPU para os processos e oferecer outros tipos de serviços de forma que o ambiente para programação paralela do T-800 seja reproduzido no i860. É da competência do Núcleo ainda, atuar como interface entre os programas em execução nos diferentes processadores (i.e. i860 e T-800) do hipercubo.

## IV.2 Serviços Oferecidos ao Usuário

O Núcleo, além de tornar invisível os detalhes de *hardware* do sistema, oferece ao usuário algumas facilidades para a implementação de processamento concorrente, principalmente no que diz respeito às primitivas e construções da linguagem Occam.

Essas facilidades incluem a comunicação entre processos, o escalonamento, a criação e destruição dinâmica de processos, a espera de um processo

durante um intervalo de tempo e a implementação da construção ALT da linguagem Occam.

## IV.3 Pontos de Ativação

O Núcleo é dirigido por interrupções, ou seja, ele somente é ativado quando da ocorrência de uma interrupção.

O processador i860 possui dois tipos de interrupções:

- interrupções internas;
- interrupções externas.

As interrupções internas são geradas pelo processo em execução quando ocorre um erro (divisão por zero, endereço inválido, etc ...), ou por uma instrução *trap*. Esse tipo de instrução requer três operandos e constitui-se num eficiente mecanismo para a implementação de *extra-codes* como veremos posteriormente.

As possíveis fontes geradoras de interrupções externas são: o processador T-800 ou o relógio de tempo real.

Tradicionalmente existem dois tipos de ferramentas para implementar a comunicação programa do usuário x Sistema Operacional: instruções especiais e *extra-codes*. Algumas máquinas (como por exemplo, o Mitra 15 [25] e algumas máquinas IBM [18]) possuem instruções especiais para o usuário ativar funções do Sistema Operacional (CSV no Mitra e SVC no IBM).

Um importante ponto no projeto de uma nova arquitetura diz respeito ao conjunto de instruções que será implementado. Uma vez especificado, esse conjunto permanece fixado, e é a partir da expansão dessas instruções que o usuário desenvolve complexas estruturas de dados e os correspondentes algoritmos para resolver os problemas de aplicação.

Com o objetivo de fornecer ao usuário a capacidade de definir instruções de máquina (virtuais) orientadas para problemas específicos, o conceito de

*extra-code* é incorporado em alguns processadores. Através desse conceito, o repertório da máquina pode ser expandido pelo próprio usuário: quando da execução de um código de operação específico (i.e., uma instrução do tipo *trap*), uma interrupção interna é gerada, e o controle da máquina pode ser transferido para a sequência de instruções responsável pela realização da instrução virtual desejada.

Além de produzir programas mais compactos, a implementação de *extra-code* através de instruções do tipo *trap* possui potencial equivalente ao oferecido pelas instruções do tipo *CSV*. No projeto do nosso Núcleo, empregamos instruções do tipo *trap* do i860 para implementar a interface 'usuário x Núcleo.'

As primitivas do Núcleo podem ser consideradas como uma espécie de *extra-code*. O usuário gera um *trap*, provocando assim uma transferência de controle para o Núcleo. Uma vez ativado, o Núcleo irá verificar qual a fonte responsável pela interrupção. Ao verificar que trata-se de uma interrupção gerada pelo usuário (i.e um *trap*), os registradores empregados na comunicação usuário x Núcleo são consultados e o controle é transferido para a rotina do Núcleo responsável pela realização do serviço requerido pelo usuário. Após a execução da primitiva, o Núcleo irá reativar o processo que gerou o *trap*, ou, se for o caso, um outro processo da fila de prontos.

A instrução *trap* do i860 requer três operandos. No primeiro operando o usuário informa qual a função do Núcleo que deseja executar (os outros dois operandos não são utilizados na nossa implementação). A cada função está associado um número inteiro conforme será visto mais tarde. Algumas funções do Núcleo necessitam de parâmetros que são passados através de registradores inteiros.

Na ativação do Núcleo, i.e. na sua entrada, o processo que chamou uma de suas funções (executando um *trap*), ou o que estava em execução quando ocorreu uma interrupção, é suspenso. O Núcleo se encarrega, então, de salvar a parte do contexto do processo que reside nos registradores que o Sistema Operacional vai utilizar (o restante do contexto será salvo se, durante a execução de alguma primitiva, a fila de prontos for alterada). Em seguida, a origem da interrupção é identificada.

Para cada tipo de interrupção o Núcleo realiza um tratamento diferente:

**Interrupção originada por uma instrução trap:** Examinando o operando da instrução, o Núcleo determina qual é a função que o usuário deseja executar e, de acordo com a função escolhida, chama uma primitiva para executá-la.

**Interrupção por erro de execução:** O erro pode ser causado por acesso indevido a uma instrução, acesso indevido a um dado, ou ainda por erro em uma operação de ponto-flutuante. Em todos os casos o Núcleo envia uma mensagem de erro ao usuário e termina o processo corrente.

**Interrupção externa:** Para identificar qual das interrupções externas ocorreu o Núcleo consulta o "processador de interrupções". No caso de uma interrupção gerada pelo relógio, o Núcleo verifica se o processo corrente esgotou sua fatia de tempo e testa se algum processo que estava esperando por um intervalo de tempo já pode ser reescalonado. No caso de uma interrupção gerada pelo processador T-800 é ativada a primitiva responsável pela comunicação com o processador T-800 conforme veremos mais tarde.

Na saída do Núcleo, após o tratamento da interrupção, o controle é retornado ao processo que foi escolhido para executar. É escolhido o primeiro processo da fila de urgentes ou, caso esta esteja vazia, o primeiro da fila de não-urgentes. Existe, sempre, pelo menos um processo na fila de prontos, pois assim que o Núcleo é inicializado, um "processo nulo" com baixa prioridade é criado e inserido na fila de prontos. Este processo realiza um *loop* eterno executando uma instrução *nop*.

Antes de transferir o controle para o processo selecionado, o Núcleo precisa restaurar seu contexto. Considerando que o contexto do processo é muito grande (376 bytes de informação), a restauração é realizada em duas partes. Primeiro são restaurados os registradores que o Núcleo utilizou. Caso durante o tratamento da interrupção ocorra uma comutação de processos, então, além dos registradores usados pelo Núcleo, deve ser restaurado o restante do contexto do processo. Quando não há comutação, a parte da memória local não usada pelo Núcleo estará intacta e, portanto, não há necessidade de restaurá-la.

Após a restauração do contexto, a única operação do Núcleo é desviar para a próxima instrução a ser executada pelo processo escalonado.

## IV.4 Escalonamento de Processos

A política de escalonamento de processos executando no processador i860 é semelhante à política empregada pelo processador T-800. A diferença básica entre elas está na implementação. O escalonamento de processos no processador T-800 é implementado em *hardware* por micro-código, enquanto que o escalonamento de processos no processador i860 é implementado em *software* pelo Núcleo.

A política de escalonamento da CPU emprega filas em dois níveis prioritários, sendo que na fila mais prioritária é utilizado o algoritmo *FIFO*, e na menos prioritária o algoritmo *Round-Robin*.

É da competência do usuário atribuir prioridades aos processos, ou, no caso de processos criados dinamicamente, compete ao processo que o criou. Tanto para o Núcleo quanto para o usuário, a prioridade é representada por um número, zero (para processos mais prioritários) ou um (para processos menos prioritários). Processos mais prioritários são chamados de **urgentes** e processos menos prioritários são chamados de **não-urgentes**.

O Núcleo mantém duas filas de processos prontos para executar, a fila de processos urgentes e a fila de processos não-urgentes. Para cada fila estão associados dois ponteiros: para o primeiro e para o último processo componente; sendo que, o último processo da fila de urgentes aponta para o primeiro da fila de não-urgentes. Se a fila de urgentes estiver vazia, então os ponteiros de início e fim da fila apontam para o primeiro processo da fila de não-urgentes. Se a fila de não-urgentes estiver vazia, então os ponteiros de início e fim de fila conterão o valor $\lambda$. O ponteiro de início da fila de urgentes servirá também como ponteiro para o processo corrente.

Um processo não-urgente só executa quando a fila de processos urgentes estiver vazia. A chegada de um processo urgente interrompe a execução de

um processo não-urgente. Para a fila de não-urgentes (onde é utilizada a política *Round-Robin*) a fatia de tempo é de dois *ticks* do relógio de tempo real.

## IV.5 Interface com o Usuário

A seguir apresentaremos as primitivas que o Núcleo oferece para realizar os serviços citados anteriormente. Os detalhes de implementação de cada primitiva, assim como seus parâmetros e sua ativação são temas abordados no próximo Capítulo.

### IV.5.1 Comunicação entre Processos

O Núcleo implementa a comunicação entre:

a) processos em execução num mesmo processador i860 - **comunicação interna**;

b) processos do processador i860 com processos sendo executados no processador T-800 e vice-versa - **comunicação externa**;

c) cópias do Núcleo residentes em diferentes nós do hipercubo - **comunicação inter-nó.**

Além das primitivas para comunicação entre processos que executam num mesmo i860, foram implementados no processador T-800 dois processos, denominados servidores, para a troca de mensagens entre os processadores T-800 e i860.

#### a) Comunicação Interna

A comunicação entre dois processos do processador i860 envolve a troca de mensagens através de canais. Ela foi implementada (seguindo os mesmos princípios da comunicação interna do T-800) de modo bloqueante, ponto-a-ponto,

não havendo, portanto, a necessidade da utilização de filas de processos ou *buffers* de mensagens.

A comunicação só acontece quando ambos os processos (entrada e saída) estão preparados para trocar mensagens. Consequentemente, o primeiro processo a ficar preparado deve esperar, bloqueado, até que o segundo processo esteja também preparado. Quando os dois processos estiverem preparados, a comunicação é realizada, com a mensagem sendo transferida do processo fonte para o processo destino.

Cada canal é mapeado numa área de memória que deve ser alocada pelo usuário. Como requerido pela linguagem Occam, no nosso meio ambiente os canais são unidirecionais, i.e. através de um canal a comunicação só pode ser realizada em um sentido. A utilização indevida dos canais (por exemplo, mais de um processo tentando enviar, ao mesmo tempo, pelo mesmo canal) constitui um erro de programação, não sendo tratado pelo Núcleo.

O Núcleo oferece duas primitivas para comunicação entre processos em execução no i860:

- ENVIA
- RECEBE

Um processo que deseja enviar ou receber uma mensagem deve executar as primitivas do Núcleo ENVIA ou RECEBE, informando qual o endereço de memória alocado para o canal, quantos bytes serão transmitidos, o endereço da área de memória onde a mensagem a ser enviada está localizada (no caso da primitiva ENVIA) ou o endereço da área onde a mensagem será armazenada (no caso da primitiva RECEBE).

A área de memória alocada para o canal possui um campo que indica se o canal está ocupado ou vazio, e outro que armazena a identificação de um processo. Todos os canais devem estar inicialmente vazios. Quando um processo executa a primitiva RECEBE ou ENVIA ele está preparado para trocar a mensagem.

O primeiro processo a tornar-se preparado encontra o canal vazio e é desescalonado. O canal é marcado como ocupado e tão logo o segundo processo torne-se preparado a comunicação é então realizada. O processo que encontrava-se bloqueado é reescalonado e o canal é novamente marcado como vazio.

### b) Comunicação Externa

Este tipo de comunicação trata da interface entre os processadores i860 e T-800. As primitivas para troca de mensagens oferecidas pelo processador T-800 (instruções de máquina *in* e *out*) permitem somente a comunicação entre processadores da família Transputer. Por esse motivo, além das primitivas do Núcleo, implementamos no T-800 processos capazes de trocar mensagens com o i860. São os **processos servidores**.

A comunicação é realizada através de segmentos de memória compartilhada pelo dois processadores, por um número limitado de canais. Cada canal é mapeado numa área reservada da memória estática e eles estão assim divididos:

- Dois canais para comunicação do T-800 com o i860;
- Dois canais para comunicação do i860 com o T-800.

Um processo em execução no T-800 pode estabelecer comunicação com o i860 por dois motivos:

- Quando um processo do T-800 deseja trocar mensagens com um outro processo em execução no i860;
- Quando um processo do T-800 faz uma **requisição de serviço** ao Núcleo do Sistema Operacional do i860 (por exemplo, para criação de um processo).

Um processo em execução no i860 se comunica com o T-800 apenas quando:

- Um processo do i860 deseja trocar mensagens com um processo em execução no T-800.

A troca de mensagens entre os dois processadores é realizada pelo Núcleo do i860 e pelos processos servidores do T-800 (existe um processo servidor para entrada e outro para saída). Para utilizar essa facilidade o usuário do i860 deve ativar as primitivas do Núcleo de envio/recebimento de mensagens (respectivamente ENVT800 e RECT800). O usuário do T-800, por sua vez, deve ativar os procedimentos ENVi860 e RECi860. Esses procedimentos ativam os processos servidores devidos.

Tanto os procedimentos do T-800, como as primitivas do Núcleo requerem três parâmetros:

- **Mensagem** – endereço do primeiro byte da mensagem a ser enviada, ou endereço do primeiro byte da área de memória onde a mensagem será armazenada;
- **Canal** – cada canal de comunicação entre os dois processadores é representado por um número inteiro (um ou dois);
- **Número de bytes** – número de bytes da mensagem.

Para que os serviços do Núcleo estivessem disponíveis aos processos em execução no T-800, ou num outro nó de processamento do NCP-1, funções de **requisição de serviço** foram incorporadas no Núcleo. Através delas, é possível solicitar remotamente a criação e destruição de processos, ou que um processo permaneça em estado de hibernação durante um intervalo de tempo. Por exemplo, o programa de um usuário em execução no T-800 pode solicitar que uma operação vetorial seja executada no i860. Nesse caso, ele deve requerer a criação remota do processo apropriado no i860.

Considerando que o mecanismo de requisição de serviço foi desenvolvido através de um tipo especial de troca de mensagens, então será trivial adicionar

outros serviços no repertório de funções oferecidas, conforme previsto para as futuras versões do Núcleo.

Para utilizar tal facilidade, o usuário do T-800 deve chamar o procedimento SERVi860. Este procedimento ativa o processo servidor de saída e requer as seguintes informações:

- **Serviço** – número do serviço a ser executado;
- **Parâmetros** – são os parâmetros requeridos pelas primitivas do Núcleo.

Para manter a compatibilidade com a comunicação realizada internamente nos processadores i860 e T-800, a troca de mensagens entre os dois processadores funciona de modo bloqueante. A requisição de um serviço ao i860 bloqueia o processo do T-800 até que o serviço tenha terminado.

Para implementar ambas as formas de comunicação (troca de mensagens e requisição de serviço) utilizamos interrupções de *hardware* para permitir que um processador suspenda a execução do outro.

Através de segmentos de memória compartilhados e de interrupções de *hardware*, a troca de mensagens entre processos executando em processadores diferentes (chamados indistintamente de i e j) é realizada da seguinte maneira:

- O primeiro processo a pedir o envio/recebimento da mensagem no processador i é desescalonado;
- O processador i envia uma interrupção ao processador j;
- O processador j recebe a interrupção e quando o pedido simétrico for executado ele envia uma interrupção ao processador i;
- O processador i ao receber a interrupção realiza a comunicação e reescalona o processo;

Já a **requisição de serviço** ao Núcleo do i860 é realizada da seguinte maneira:

- O processo que requisita no T-800 um serviço ao i860 é desescalonado;
- O T-800 interrompe o i860;
- O i860, ao receber a interrupção, verifica qual o serviço requerido e ativa a rotina correspondente;
- Ao término do serviço o i860 interrompe o T-800;
- O T-800, ao receber a interrupção, reescalona o processo.

Quando da troca de mensagens e da **requisição de serviços** no i860, os servidores do T-800 fazem o tratamento de um interrupção externa e a comutação de processos.

No tratamento das interrupções do T-800, utilizamos uma facilidade presente na implementação da linguagem Occam. A linguagem associa um canal especial às interrupções externas, e quando da ocorrência de um desses eventos esse canal torna-se preparado para realizar a primitiva *input* da linguagem. Esse mecanismo foi empregado na implementação dos nossos servidores, provendo dessa forma, a comutação de processos requerida.

### b) Comunicação Inter-nó

A requisição de serviços entre Núcleos foi implementada de maneira bastante simples devido às facilidades encontradas na máquina hospedeira: a ligação entre os nós de processamento feita através do barramento VME (que conecta os processadores i860 do hipercubo). O barramento VME quando utilizado, conecta exatamente dois nós de processamento e através dele é possível enviar dados e interrupções.

Nesse caso a requisição de serviços tem um comportamento assíncrono. O Núcleo solicitando o serviço interrompe o i860 destino e envia o pedido. O serviço é executado e uma interrupção é gerada avisando do término. Enquanto um Núcleo

executa os serviços o outro pode continuar em execução pois a interrupção indicará a conclusão da tarefa solicitada.

### IV.5.2 Criação e Destruição Dinâmica de Processos

Como ocorre no T-800, associado a cada processo do i860 existe uma área de trabalho na memória denominada de *workspace*. Nesta área são armazenadas variáveis utilizadas pelo Núcleo (conforme será visto no próximo Capítulo) em conjunto com o contexto do processo.

A alocação do *workspace* na memória é feita pelo usuário, pois a versão corrente do Núcleo não provê suporte para alocação dinâmica de memória.

Para criação e destruição dinâmica de processos o Núcleo emprega as seguintes primitivas:

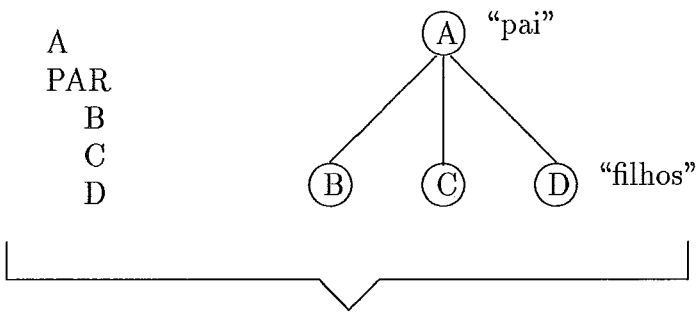
- INIC-PAR - cria um processo com a mesma prioridade do processo corrente;
- BLOQUEIA - retira o processo corrente da fila de prontos;
- FIM-PAR - termina um processo iniciado por INIC-PAR;
- DESESCALONA - retira o processo corrente da fila de prontos, armazenando em seu *workspace* o endereço da próxima instrução a ser executada quando ele for reescalonado.

Para criar um processo dinamicamente, o usuário deve ativar a primitiva INIC-PAR, informando o endereço de seu *workspace*. Existem três maneiras para terminar um processo. A primeira delas, através da ativação da primitiva BLOQUEIA, é utilizada para terminar um processo iniciado estaticamente pelo usuário. A segunda, através da primitiva FIM-PAR, está diretamente relacionada a implementação da construção PAR da linguagem Occam. A terceira, por intermédio da primitiva DESESCALONA, suspende um processo que poderá ser reescalonado posteriormente.

A Figura IV.1 mostra um exemplo de uma construção PAR da linguagem Occam e como ela é implementada através das primitivas do Núcleo.

Construção PAR em Occam:

```
A
PAR
  B
  C
  D
```




Implementação no Núcleo:

| Processo A | Processo B | Processo C | Processo D |
|---|---|---|---|
| ⋮ | ⋮ | ⋮ | ⋮ |
| CONTADOR := 3 | | | |
| INIC-PAR(B) | FIM-PAR | FIM-PAR | FIM-PAR |
| INIC-PAR(C) | | | |
| INIC-PAR(D) | | | |
| DESESCALONA | | | |

Figura IV.1: Implementação da construção PAR

Para implementar a construção PAR as primitivas são utilizadas da seguinte maneira:

- O primeiro passo consiste na inicialização no *workspace* do processo "pai" do contador de processos "filhos" que serão criados. Os *wokspaces* dos processos "filhos" devem estar carregados na memória.

- Para cada processo "filho", o usuário ativa a primitiva INIC-PAR, passando como parâmetro o endereço do *workspace* correspondente.

- Quando todos os "filhos" estiverem iniciados, o processo "pai" deve ser desescalonado. Para tal, o usuário deve executar a primitiva DESESCALONA. Esta primitiva retira o descritor do processo "pai" da fila de processos prontos.

- No final da execução, cada "filho" ativa a primitiva FIM-PAR que retira o processo da fila de prontos e decrementa o contador no *workspace* do processo

"pai" de uma unidade. Quando a primitiva FIM-PAR encontra o contador com zero, o processo "pai" é reescalonado.

Existe ainda outra maneira de se criar um processo dinamicamente. Isso ocorre quando um usuário do processador T-800 requisita o serviço de criação de um processo no processador i860, conforme mencionado anteriormente.

### IV.5.3 Espera por Um Intervalo de Tempo

O Núcleo oferece ao usuário do i860 facilidades para que seu processo fique bloqueado durante um intervalo de tempo. A implementação desse mecanismo difere bastante de como é implementado na linguagem Occam. Em Occam, o pedido de espera por um intervalo de tempo é um caso especial do comando de Entrada quando canais especiais são utilizados para temporização.

Em nossa implementação do Núcleo, o pedido de espera por intervalo de tempo é feito através da ativação da primitiva DELAY, com o usuário informando qual o intervalo de tempo que o processo deve ficar "hibernando". Essa primitiva retira o processo da fila de prontos, colocando-o na fila de processos esperando por intervalo de tempo.

A cada interrupção do relógio é realizada uma busca nesta fila para verificar se algum processo já terminou o seu tempo de "hibernação", e pode, assim, retornar à fila de prontos. Para que esta busca torne-se mais simples, a fila de espera por tempo é organizada em ordem crescente da "hora a ser alcançada". Após o relógio ter alcançado esta "hora", o processo é transferido de volta para a fila de prontos.

### IV.5.4 Realização da Construção ALT da Linguagem Occam

O Núcleo inclui algumas primitivas para que o usuário possa implementar o equivalente a uma construção ALT da linguagem Occam.

Conforme visto na definição da linguagem Occam, uma construção ALT é formada de um conjunto de comandos guardados do tipo:

**{guarda, lista de comandos}**

Um **guarda** é formado por um comando de entrada (que pode incluir espera por um intervalo de tempo), por um processo SKIP e possivelmente por uma expressão booleana. O guarda representa a "condição" que será avaliada durante a execução do ALT; ele é dito **pronto** se a expressão booleana que o compõe é verdadeira e se o comando de entrada a ele associado está preparado para trocar mensagens.

A execução de uma construção ALT consiste em escolher um dos comandos guardados para ser executado. O comando guardado é escolhido, de forma arbitrária, dentre aqueles que possuem seu guarda **pronto**. Uma vez escolhido o comando guardado, sua lista de comandos é executada.

A implementação dessa construção requer um mecanismo para avaliar as condições especificadas por cada um dos guardas e um outro mecanismo para selecionar um dos comandos.

Esses mecanismos seriam facilmente implementados se a condição especificada no guarda não envolvesse um comando de Entrada. (A avaliação da expressão booleana deve ser feita pelo usuário). Um comando de entrada pode estar, ou não, preparado para a troca de mensagens. Ele é dito preparado quando o comando de saída respectivo já foi executado.

Para avaliar se um comando de entrada encontra-se preparado verificamos se o canal por ele utilizado já está ocupado (por um comando de saída):

- Estando o comando de entrada preparado, o guarda está **pronto** (apesar dos comandos de entrada e saída estarem preparados para executar, a comunicação não é realizada);
- Se o comando de entrada não estiver preparado, o guarda não está pronto e é necessário marcar o canal por ele utilizado como "habilitado para uma

entrada" (o canal continua marcado como vazio). Quando da execução do comando de saída, o canal estará vazio (a comunicação não será realizada).

Em Occam a espera por um intervalo de tempo é um caso especial do comando de entrada. Na nossa implementação eles são distintos. Por conseguinte, a avaliação da condição especificada no guarda possui mais um caso: espera por intervalo de tempo.

Quando a "hora" que deveria ser alcançada pelo guarda já foi atingida, ele está **pronto**. Se o guarda não estiver **pronto** a "hora" a ser alcançada é armazenada em seu *workspace*. Este valor será utilizado se o processo for transferido para a fila de processos esperando por intervalo de tempo.

Conforme especificado pela semântica da construção ALT, se nenhum guarda estiver **pronto**, a construção fica suspensa até que pelo menos um deles torne-se **pronto**. Nesse caso, o processo executando a construção ALT é desescalonado (se em algum guarda existir uma espera por intervalo de tempo, o *workspace* do processo é inserido na fila de processos esperando por intervalo de tempo).

O reescalonamento do processo (i.e. sua inserção na fila de prontos) depende ou da execução de um comando de saída em um canal "habilitado para uma entrada", ou de uma hora desejada ser atingida.

Existindo pelo menos um guarda **pronto**, a seleção do comando guardado pode ser realizada. Por simplicidade, decidimos escolher o primeiro comando cujo guarda esteja **pronto**.

Antes da execução do comando selecionado é preciso inabilitar a execução de todos os outros comandos guardados. Os canais que estavam habilitados para uma entrada são inabilitados, i.e. as trocas de mensagens que poderiam ser realizadas não serão mais processadas.

No último passo transfere-se o controle para o comando guardado escolhido: é realizada a troca de mensagem especificada no guarda; e em seguida executa-se a lista de comandos correspondente.

Para a implementação da construção ALT, o Núcleo incorpora um total de nove primitivas. Estas primitivas se assemelham com algumas instruções de máquina que o processador T-800 possui para implementar a construção ALT.

A Figura IV.2 ilustra o exemplo de uma construção ALT em Occam e sua implementação segundo as primitivas do Núcleo.

As primitivas são:

- INIC-ALT - inicia uma construção ALT;
- HAB-CANAL e HAB-TMP - verificam se um guarda, que possui, respectivamente, um comando de entrada ou uma espera por um intervalo de tempo, está pronto;
- HAB-SKIP - marca um guarda como **pronto**;
- SUSP-ALT e SUSP-ALTTMP - retiram o processo da fila de prontos caso não haja nenhum guarda **pronto**. A primitiva SUSP-ALTTMP realiza, ainda, a inserção do processo na fila de processos esperando por um intervalo de tempo;
- DESB-CANAL, DESB-TMP e DESB-SKIP - desabilitam o guarda que possui respectivamente, uma Entrada, uma espera por um intervalo de tempo, ou um processo SKIP;
- FIM-ALT - finaliza uma construção ALT.

Para iniciar uma construção ALT a primitiva INIC-ALT deve ser executada, ela será responsável pelas inicializações necessárias. Para cada guarda ativa-se uma das primitivas de habilitação: HAB-CANAL se o guarda possui um comando de entrada; HAB-TMP se o guarda possui uma espera por determinado tempo; HAB-SKIP se o guarda possui um processo do tipo SKIP. Essas primitivas testam se o guarda já está **pronto** e, caso não esteja, o habilitam a se tornar. O tratamento da expressão booleana de um guarda deve ser feito pelo usuário, ou melhor, antes executar uma primitiva de habilitação o usuário deve testar se a

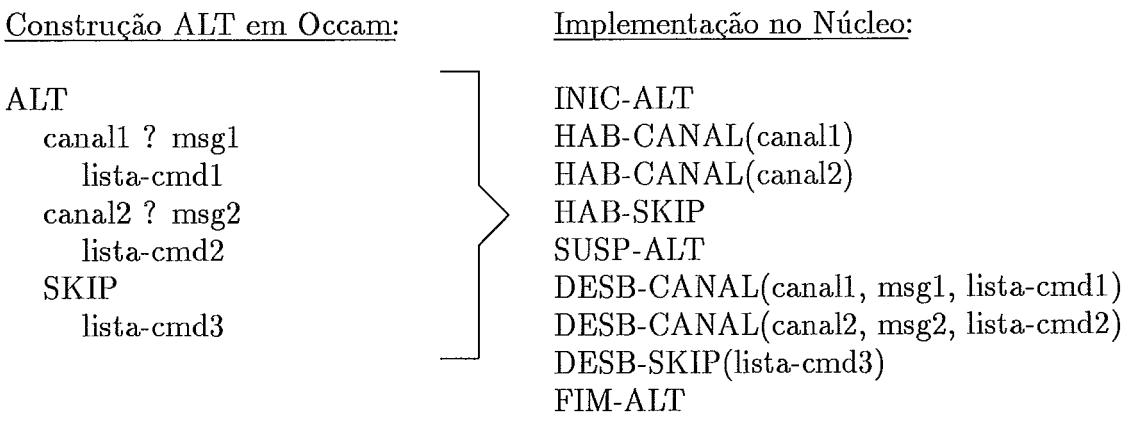



Figura IV.2: Implementação da construção ALT

expressão booleana do guarda é verdadeira. Caso todas as expressões booleanas sejam falsas, o processo deve ser terminado, pois representa um erro de execução.

Após a habilitação de todos os guardas, uma das duas primitivas: SUSP-ALT, ou SUSP-ALTTMP (caso exista em pelo menos um guarda uma espera por determinado tempo), deve ser executada. Elas se encarregam de desescalonar o processo caso nenhum guarda esteja pronto depois das habilitações. O processo será reescalonado automaticamente quando algum guarda tornar-se pronto.

A seguir, para cada comando guardado, ativa-se uma das primitivas do Núcleo de desabilitação: DESB-CANAL para os guardas que possuem um comando de entrada; DESB-TMP para os guardas que possuem uma espera por intervalo de tempo; DESB-SKIP para os guardas que possuem um processo do tipo SKIP. Essas primitivas têm duas funções. A primeira consiste em inabilitar uma possível troca de mensagens (dos guardas associados a comandos guardados que não irão executar). A segunda refere-se à escolha do comando guardado que será executado. Para cada comando guardado é testado se o guarda respectivo está **pronto**, caso esteja e se nenhum dos comandos guardados anteriores foi selecionado, então este comando guardado é o escolhido. A troca de mensagens do guarda é realizada juntamente com a execução de sua lista de comandos. A ordem na qual o usuário ativa as primitivas de desabilitação determina a prioridade dos comandos guardados.

No final da construção ALT a primitiva FIM-ALT deve ser executada, ela é responsável pela finalização da construção.

# Capítulo V

# Implementação do Núcleo

Este capítulo descreve a implementação do Núcleo no processador i860. Serão apresentadas a estrutura de dados utilizada e todas as rotinas que o compõem. Cada rotina será descrita em termos de sua função básica, dos parâmetros requeridos e do algoritmo correspondente.

## V.1 Estrutura de Dados

Cada processo possui uma área de trabalho na memória, o *workspace*. Ele armazena o contexto do processo e outras informações referentes ao processo que são utilizadas pelo Núcleo. A Figura V.1 mostra o *layout* de um *workspace*.

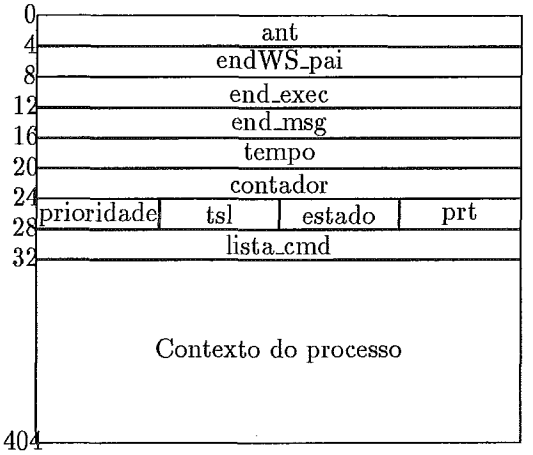



Figura V.1: Estrutura de um *workspace*

O *workspace* é equivalente a um descritor de processos. Conforme

ilustrado na Figura V.2, os *workspaces* são encadeados, formando a lista de descritores de processos que é manipulada pelas rotinas do Núcleo. Cada *workspace* inclui os seguintes campos:

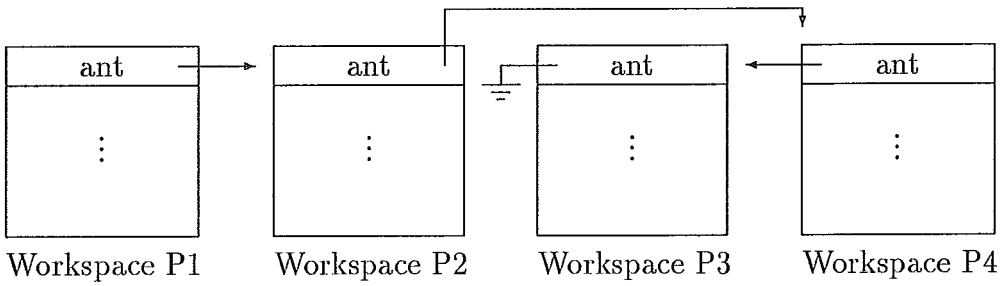



Figura V.2: Lista de descritores de processos

- O campo **ant** é usado no encadeamento dos processos de cada fila. Ele aponta para o *workspace* do processo precedente;
- O campo **endWS_pai** aponta para o *workspace* do processo "pai";
- O campo **end_retorno** indica o endereço da próxima instrução a ser executada pelo processo, quando este for escalonado;
- O campo **end_msg** armazena o endereço de memória que será utilizado na troca de mensagens entre dois processos. Ele aponta para o primeiro byte da mensagem a ser transferida (no caso de envio da mensagem), ou para o início da área onde a mensagem deve ser copiada (no caso de recebimento da mensagem);
- O campo **tempo** indica até quando o processo deve permanecer bloqueado;
- O campo **contador** armazena o número atual de processos concorrentes iniciados por um processo "pai";
- O campo **prioridade** especifica qual a prioridade do processo. Este campo conterá o valor zero para processos urgentes e o valor um para processos não-urgentes;
- O campo **tsl** indica, nos processos não-urgentes, a fatia de tempo remanescente;

- O campo **estado** armazena o estado do processo, i.e. se o processo está ativo (valor = 0), ou se está bloqueado (valor = 1);
- O campo **prt** é usado como *flag* durante a execução da construção ALT;
- O campo **lista-cmd**, também usado na execução da construção ALT, aponta para a primeira instrução (da lista de comandos) do comando guardado selecionado pela construção.

A estrutura de um canal utilizado na comunicação interna está ilustrada na Figura V.3. Cada canal é representado por uma área na memória que armazena as seguintes informações:

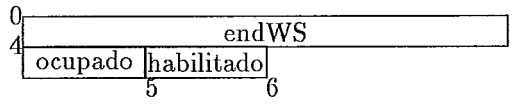



Figura V.3: Estrutura de um canal para comunicação interna

- O campo **endWS** armazena o endereço do *workspace* de um processo que está bloqueado a espera de uma comunicação;
- O campo **ocupado** atua como *flag* indicando se já existe um processo esperando pela comunicação;
- O campo **habilitado** indica se a comunicação faz parte de um guarda da construção ALT.

Na comunicação externa (i.e., i860xT-800), os canais apresentam uma estrutura diferente, conforme mostra a Figura V.4. A estrutura da dados referente ao canal inclui os seguintes campos:

- **endWS** - armazena o endereço do *workspace* do processo que está bloqueado a espera de uma comunicação;
- **mens** - armazena o endereço destino da mensagem (no caso de recebimento) ou o endereço fonte da mensagem (no caso de envio);

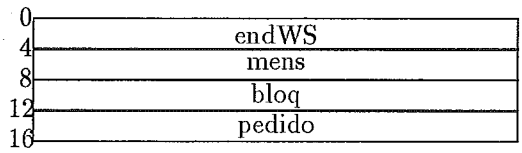



Figura V.4: Estrutura de um canal para comunicação externa

- **bloq** - indica se existe, no i860, algum processo bloqueado esperando pela comunicação;
- **pedido** - *flag* consultado pelo i860 e T-800 para que possam identificar se o processador oposto já fez o pedido pela comunicação.

As variáveis locais ao Núcleo **Inic_0**, **Fim_0**, **Inic_1**, **Fim_1** e **Inic-tmp** são usadas como apontadores na implementação das filas de processos. A Figura V.5 ilustra o emprego desses apontadores numa possível configuração. Neste exemplo a fila de urgentes é formada pelo processo **A** e a fila de não-urgentes é formada pelos processos **B**, **D** e **C**.

A função desempenhada por cada ponteiro segue-se:

- **Inic_0** - aponta para o primeiro *workspace* da fila de processos urgentes. Utilizado como ponteiro para o processo corrente;
- **Fim_0** - aponta para o último *workspace* da fila de processos urgentes;
- **Inic_1** - aponta para o primeiro *workspace* da fila de processos não-urgentes;
- **Fim_1** - aponta para o último *workspace* da fila de processos não-urgentes;
- **Inic_tmp** - aponta para o primeiro *workspace* da fila de processos esperando por um intervalo de tempo.

Obs: Apesar de existirem duas filas de prontos, "o primeiro processo da fila dos prontos" representa o primeiro processo da fila de urgentes (caso esta não esteja vazia) ou o primeiro processo da fila de não-urgentes quando a fila de urgentes estiver vazia.

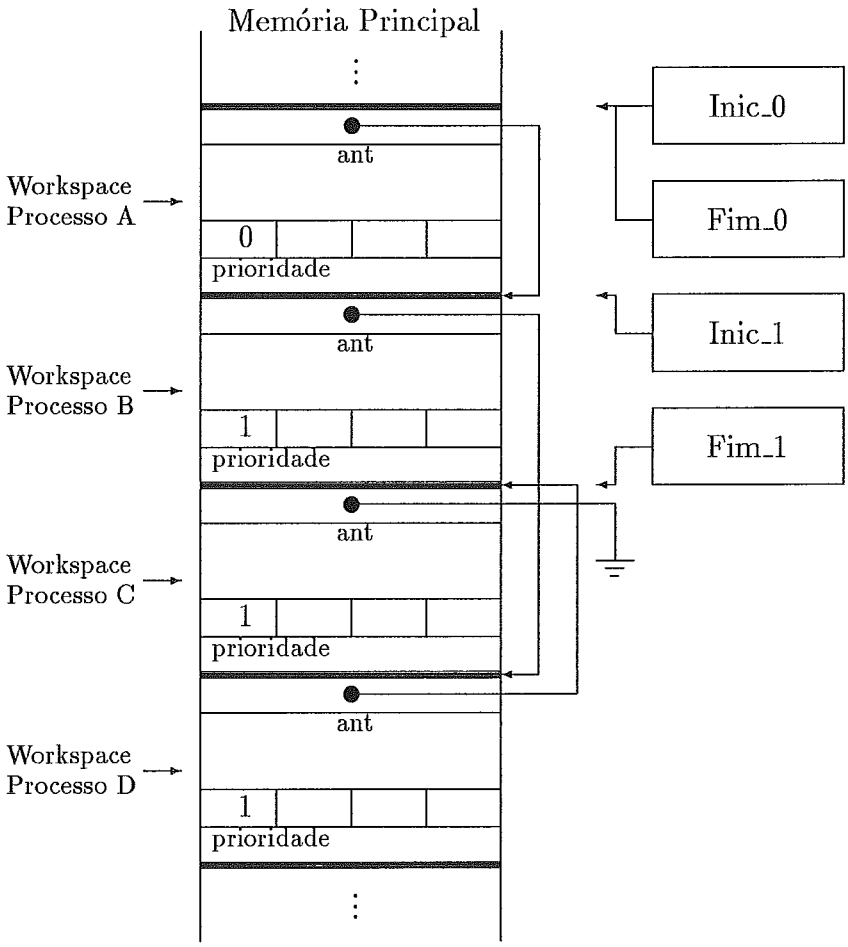



Figura V.5: Filas de prontos

## V.2 Primitivas do Núcleo

A seguir serão apresentadas as primitivas e as sub-rotinas que compõem o Núcleo. As primitivas representam as facilidades oferecidas pelo Núcleo e que podem ser ativadas diretamente pelo usuário. Sub-rotinas são os procedimentos que não podem ser ativados pelo usuário, pois representam as rotinas de serviço ou manutenção, usadas internamente pelo Núcleo. Empregou-se a Linguagem *Assembly* [30], na codificação das primitivas e sub-rotinas, exceto o código relacionado com os processos servidores que foram implementados em Occam. Os algoritmos serão descritos em linguagem de alto nível: Pascal para aquelas implementadas em *Assembly* e Occam para os processos servidores.

Quando da ativação das primitivas do Núcleo, os registradores inteiros do i860 são usados na passagem de parâmetros. Compete ao usuário (ou ao tradutor da linguagem de programação) a tarefa de passagem de parâmetros. Na Tabela A.1 estão listados, para cada primitiva, os parâmetros com a identificação dos respectivos registradores e como eles devem ser interpretados.

Conforme mencionado no Capítulo anterior, para o usuário ativar uma das primitivas do Núcleo ele deve executar a instrução *trap* indicando, no campo referente ao primeiro operando dessa instrução, um número inteiro (que especifica o número da primitiva que ele deseja executar), no segundo e no terceiro operando pode ser passado o valor zero. Os números identificando cada uma das primitivas são estão também listados na Tabela 1 do Apêndice A.

As primitivas do Núcleo podem ser divididas em sete grupos. Os grupos são formados pelas primitivas usadas:

- para salvar e resguardar o contexto do processo;
- como subrotinas de manutenção;
- na realização da construção PAR;
- no escalonamento de processos;

- para realizar o mecanismo de espera por intervalo de tempo;

- na comunicação entre processos;

- na realização da construção ALT.

### V.2.1 Salvamento e Restauração do Contexto

O contexto de um processo inclui o conteúdo dos registradores do i860 (com exceção do DB) e dos estágios dos dois *pipelines* (soma e multiplicação de ponto-flutuante). Toda vez que um processo é desescalonado (i.e. deixa a fila de prontos), é necessário resguardar, em seu *workspace*, todo o seu contexto. Quando ele for novamente escalonado, carrega-se nos registradores do i860 os valores utilizados pelo processo.

Como o contexto de um processo é muito grande, quando da ativação do Núcleo são resguardados apenas alguns registradores inteiros, r1 a r11 (aqueles que o Núcleo irá utilizar como rascunho). O restante do contexto do processo só é salvo se o processo, por algum motivo, for desescalonado.

**SALVA-CONTEXTO**

- Função: Salva no *workspace* do processo corrente o restante do contexto que não foi salvo na entrada do Núcleo.
- Parâmetros: Nenhum.
- Algoritmo:

```
begin
  Salva, no workspace do processo corrente, todos os
   registradores do i860;(menos os registradores DB e r0 a r11)
  Salva o estado dos pipelines;
end;
```

Essa subrotina é chamada apenas por outras primitivas do Núcleo.

Toda vez que uma primitiva do Núcleo alterar a fila de prontos, modificando a primeira posição, a primitiva SALVA-CONTEXTO é chamada.

Para salvar o estado de cada *pipeline*, o Núcleo realiza três operações *dummy*. Após a execução de cada uma delas, os estados e os resultados intermediários do *pipeline* são resguardados.

**REST-CONTEXTO**

- Função: Restaura o contexto do processo escalonado.
- Parâmetros: Nenhum.
- Algoritmo:

```
begin
  Restaura registradores r1 a r11;
  If (houve desescalonamento) then
    Restaura o restante do contexto;
end;
```

Essa primitiva é utilizada na saída do Núcleo.

Ao retornar o controle ao processo escalonado, o Núcleo precisa restaurar seu contexto. Primeiro são restaurados os registradores r1 a r11 que o Núcleo utilizou. Caso durante a execução das primitivas do Núcleo tenha ocorrido a comutação de processo, o restante do seu contexto é também restaurado.

### V.2.2 Subrotinas de Manipulação de Filas

As primitivas apresentadas a seguir são utilizadas como subrotinas de manutenção, ou seja, são chamadas apenas por outras primitivas do Núcleo.

Elas são usadas para manipular a fila de prontos e a fila de processos esperando por um intervalo de tempo. São primitivas para inserção e retirada de descritores da fila de prontos e para inserção na fila de *delay*.

**INS-PRONTOS**

- Função: Insere o apontador do *workspace* de um processo numa das filas de prontos. O campo prioridade indica em que fila o processo será incluído.
- Parâmetros:
  1. endWS -> apontador para o *workspace* do processo que será inserido na fila de prontos.
- Algoritmo:

```
begin
  endWS^.ant := nil;
  endWS^.tsl := 0;
  if endWS^.prioridade = 0 then (* insere na lista de urgentes *)
    if Inic_0 = Inic_1 then
      begin (* lista de processos urgentes esta vazia *)
        if Inic_1 <> nil  then
          SALVA_CONTEXTO; (* do processo que estava em execucao *)
        endWS^.ant := Inic_0;
        Inic_0 := endWS;
        Fim_0 := endWS;
      end
    else
      begin (* lista de processos urgentes nao esta vazia *)
        endWS^.ant := Fim_0^.ant;
        Fim_0^.ant := endWS;
```

```
        Fim_0 := endWS;
      end
  else
    begin (* insere na lista de nao-urgentes *)
      if Inic_1 = nil then
        begin (* lista de nao-urgentes esta vazia *)
          Inic_1 := endWS;
          if Inic_0 = nil then
            begin
              Inic_0 := endWS;
              Fim_0 := endWS;
            end
          else
            Fim_0^.ant := Inic_1;
        end
      else (* lista de nao-urgentes nao esta vazia *)
        Fim_1^.ant := endWS;
        Fim_1 := endWS;
    end;
end;
```

Se a prioridade do processo for igual a zero, o *workspace* será inserido na fila de processos urgentes, caso contrário, será inserido na fila de processos não-urgentes.

Caso a prioridade seja igual a zero e se a fila de processos urgentes estiver vazia, o processo será o único da fila, interrompendo, portanto, o processo não-urgente que estava sendo executado (o contexto deste processo deve ser salvo, pois ele será desescalonado). Se já houver processos urgentes na fila, então o apontador do *workspace* do processo será inserido na última posição da fila de processos urgentes.

Se a prioridade for igual a um, o processo será inserido no final da

fila de não-urgentes. Quando a fila de não-urgentes estiver vazia, o processo será o primeiro e último processo da fila e, ainda, se a fila de urgentes também estiver vazia, o processo é escalonado para execução.

**TIRA-PRONTOS**

- Função: Retira o primeiro processo da fila dos prontos
- Parâmetros: Nenhum
- Algoritmo:

```
var
  prim: ponteiro p/ um workspace
begin
  prim := Inic_0^.ant;
  if Inic_0 = Inic_1 then
    begin (* fila de urgentes esta vazia *)
      Inic_1 := prim;
      Fim_0 := prim;
      if Inic_1 = nil then
        Fim_1 := nil;
    end
  else
    if Inic_0 = Fim_0 then
      Fim_0 := prim;
  Inic_0 := prim;
end;
```

Para retirar o primeiro processo da fila dos prontos, precisamos atualizar a variável Inic_0 de modo que ela aponte para o sucessor.

Se a fila de processos urgentes estiver vazia atualiza-se as variáveis Inic_1 e Fim_0. Se o processo a ser retirado for o único componente da fila de não-urgentes, a variável Fim_1 é atualizada.

Se a fila de processos urgentes não estiver vazia, mas o processo a ser retirado é o único processo da fila de urgentes, a variável Fim_0 deve ser atualizada.

**INS-TMP**

- Função: Insere o apontador do *workspace* de um processo na fila de processos esperando por um intervalo de tempo.
- Parâmetros:
  1. endWS -> apontador para o *workspace* que será inserido na fila.
- Algoritmo:

```
begin
  endWS^.estado := 1;
  tempo := endWS^.tempo + Hora_Atual
  if (Inic_tmp <> nil) and (tempo > Inic_tmp^.tempo) then
    begin
      j := nil;
      i := Inic_tmp;
      while (i <> nil) and (tempo > i^.tempo)
        begin
          j := i;
          i := Inic_tmp^.ant;
        end
      j^.ant := endWS;
      endWS^.ant := i;
    end
```

```
    else
      begin
        endWS^.ant := Inic_tmp;
        Inic_tmp := endWS;
      end;
  end;
```

A fila de processos aguardando por intervalo de tempo é organizada cronologicamente: os processos que serão reativados mais cedo ocupam as primeiras posições da fila. Por essa razão, antes de inserir um processo na fila, o algoritmo percorre a fila, examinando o campo indicativo do *delay* de cada processo, até que seja localizada a posição relativa do processo em questão. O *workspace* do processo corrente é então transferido da fila de processos prontos para a fila de processos aguardando intervalo de tempo.

### V.2.3 Implementação da Construção PAR

As funções do Núcleo, INIC-PAR, DESESCALONA e FIM-PAR, são usadas na realização da construção PAR da linguagem Occam. Elas são responsáveis pela criação e destruição dinâmica de processos e pelo desescalonamento do processo que especificou a construção PAR. A primitiva BLOQUEIA é usada para terminar o processo corrente.

#### INIC-PAR

- Função: Cria um processo concorrente ("filho") com a mesma prioridade do processo corrente ("pai").
- Parâmetros:
  1. endWS -> endereço do *workspace* do processo "filho" a ser criado.

2. prim_inst -> endereço da primeira instrução a ser executada pelo processo "filho".

- Algoritmo:

```
begin
  endWS^.prioridade := Inic_0^.prioridade;
  endWS^.endWS_pai := Inic_0;
  endWS^.end-exec := prim_inst;
  INS_PRONTOS (endWS);
end;
```

O *workspace* do processo "filho" é inserido na fila de processos com prioridade idêntica ao processo corrente (processo "pai").

Antes de inserí-lo, entretanto, é preciso armazenar, em seu *workspace*, o conteúdo de algumas variáveis que serão utilizadas posteriormente. A variáveis são:

- prioridade: a subrotina INS-PRONTOS necessita dessa informação para incluir o processo numa das filas;

- endWS_pai: quando o processo terminar, é preciso acessar o *workspace* do processo "pai" para decrementar o contador que indica o número de filhos ativos;

- end-exec: indica qual é a primeira instrução a ser executada.

Após a atualização dessas variáveis no *workspace* do processo "filho", a subrotina INS-PRONTOS irá inseri-lo na fila de prontos.

## DESESCALONA

- Função: Retira o processo corrente da fila de prontos, armazenando, no seu *workspace*, o endereço da próxima instrução a ser executada, quando ele for reescalonado.
- Parâmetros:
  1. end-retorno -> endereço da próxima instrução a ser executada pelo processo quando ele for reescalonado.
- Algoritmo:

```
begin
  SALVA_CONTEXTO;
  endWS^.end-exec := end_retorno;
  TIRA-PRONTOS;
end;
```

O objetivo dessa primitiva é retirar o *workspace* do processo corrente da fila de prontos. Inicialmente ela resguarda o contexto do processo em seu *workspace*. Em seguida o endereço da próxima instrução a ser executada. Por fim, a sub-rotina TIRA-PRONTOS é chamada para que o *workspace* do processo seja retirado da fila de prontos.

DESESCALONA é utilizada quando da geração de processos "filhos" concorrentes. Após o processo "pai" ter criado todos os processos "filhos" ele ativa a primitiva, saindo da fila de prontos. O último "filho" ao terminar sua execução reescalona o "pai".

**FIM-PAR**

- Função: Termina um processo concorrente.
- Parâmetros: Nenhum.
- Algoritmo:

```
begin
  Inic_0^.endWS_pai^.contador := Inic_0^.endWS_pai^.contador - 1;
  if Inic_0^.endWS_pai^.contador = 0 then
    INS_PRONTOS (Inic_0^.endWS_pai);
  TIRA-PRONTOS;
end;
```

Para garantir o término correto de diversos processos criados pela construção PAR, o *workspace* do processo "pai" contém um contador que é inicializado com o número de processos "filhos". Toda vez que um processo "filho" que termina, a primitiva FIM-PAR decrementa esse contador de uma unidade e retira seu *workspace* da fila de prontos. Quando o contador for igual a zero, significa que o último "filho" terminou, e, portanto, é preciso reescalonar o "pai" (ou seja, inseri-lo na fila de prontos).

**BLOQUEIA**

- Função: Retira o processo corrente da fila de prontos
- Parâmetros: Nenhum

- Algoritmo:

```
begin
  TIRA_PRONTOS;
end;
```

### V.2.4 Escalonamento de Processos

De acordo com a política de escalonamento de processos implementada no Núcleo, processos não-urgentes recebem uma fatia de tempo de CPU equivalente a dois *ticks* do relógio de tempo real. Portanto, após cada *tick* do relógio é preciso testar se o processo (não-urgente) que está usando o processador pode continuar ou deve retornar para o final da fila de prontos. Assim, quando ocorre uma interrupção do relógio de tempo real (i.e., ao término de um *tick*), o Núcleo chama a sub-rotina FATIA-TEMPO para efetivar o escalonamento dos processos não-urgentes.

**FATIA-TEMPO**

- Função: Verifica se um processo não-urgente já esgotou sua fatia de tempo.
- Parâmetros: Nenhum
- Algoritmo:

```
begin
  if Inic_0 <> nil then (* fila de prontos nao vazia *)
    if Inic_0^.prioridade = 1 then
      if Inic_0^.tsl = 0 then
        Inic_0^.tsl := 1
      else
```

```
            begin
              SALVA_CONTEXTO;
              if Inic_0 <> Fim_1 then
                begin (* existem mais processos nao-urgentes *)
                  Inic_0 := Inic_0^.ant;
                  Fim_1^.ant := Inic_1;
                  Fim_1 := Inic_1;
          Inic_1 := Inic_0;
          Fim_0 := Inic_0;
                  Fim_1^.ant := nil;
                end;
              Fim_1^.tsl := 0;
            end;
    end;
```

Se a fila de prontos não estiver vazia e se o primeiro processo da fila de prontos é um processo não-urgente, então no *workspace* do processo corrente o campo **tsl** é verificado.

Se **tsl** contiver zero, significa que o processo usou apenas um *tick*. Dessa forma, o campo recebe o valor um, indicando que metade da fatia de tempo já foi consumida, e o processo continua na primeira posição da fila de prontos.

Caso contrário (**tsl** =1), o processo esgotou sua fatia de tempo e deve ser desescalonado. O contexto é salvo e a fila de prontos é reorganizada de modo que o processo corrente ocupe a última posição da mesma.

Se o processo corrente não for o único componente da fila de não-urgentes, as variáveis Inic_0, Fim_0 e Inic_1 são atulizadas com o endereço do *workspace* do segundo processo da fila e a variável Fim_1 com o endereço do *workspace* do processo corrente. Se o processo corrente for o único componente da fila de não-urgentes, ele não é desescalonado. Sendo o processo desescalonado ou não, o

campo **tsl** de seu *workspace* é reinicializado com 0 para que, da próxima vez ele possa utilizar mais dois *ticks* do relógio.

Obs: Toda vez que um processo for inserido na fila de prontos por qualquer outra primitiva do Núcleo, o campo **tsl** de seu *workspace* é zerado.

### V.2.5 Espera por um Intervalo de Tempo

O Núcleo possui facilidades para que o processo do usuário possa ficar suspenso durante um certo intervalo de tempo. A primitiva DELAY permite manter o processo bloqueado por um determinado tempo. A rotina BUSCA-FILATMP é usada pelo Núcleo para que o processo possa retornar à fila de prontos assim que o intervalo de tempo de espera tenha sido esgotado.

**DELAY**

- Função: Permite que o processo do usuário espere, bloqueado, durante um determinado intervalo de tempo.
- Parâmetros:
  1. tempo -> intervalo de tempo que o processo ficará bloqueado.
- Algoritmo:

```
var
  endWS,i,j : ponteiro p/ um workspace;
begin
  SALVA_CONTEXTO;
  Inic_0^.tempo := tempo + Hora_atual;
  endWS := Inic_0;
  TIRA_PRONTOS;
  INS_TMP(endWS);
end;
```

O *workspace* do processo corrente será retirado da fila de prontos e inserido na fila de processos esperando por um intervalo de tempo. A cada interrupção de relógio, a fila de processos esperando por um intervalo de tempo é percorrida para testar se algum dos processos bloqueados já pode ser reescalonado (i.e, se o intervalo de tempo já transcorreu).

Como o processo será desescalonado, é necessário resguardar seu contexto. Em seguida, é armazenada, no *workspace* do processo, a hora que ele deve ser desbloqueado (esta hora é obtida somando-se o valor do relógio de tempo real — Hora_atual — com o intervalo de tempo que o processo deve esperar).

## BUSCA-FILATMP

- Função: A cada interrupção de relógio é feita uma busca na fila de processos esperando por intervalo de tempo para verificar se algum processo já pode ser reescalonado.
- Parâmetros: Nenhum
- Algoritmo:

```
var
  endWS : ponteiro p/ um workspace;
begin
  if Inic_tmp <> nil then
    if Inic_tmp^.tempo < Hora_atual then
      while (Inic_tmp <> nil) and
            (Inic_tmp^.tempo < Hora_atual) do
        begin
          endWS := Inic_tmp;
          Inic_tmp := Inic_tmp^.ant;
```

```
            if endWS^.estado = 1 then
              begin
                endWS^.estado := 0;
                INS_PRONTOS (endWS);
              end;
          end;
    end;
```

A fila de processos esperando por intervalo de tempo é percorrida e, cada processo cuja com a hora a ser alcançada é menor (mais cedo) do que a hora do relógio de tempo real, será transferido para a fila de prontos. Quando é encontrado um processo cujo campo tempo possui valor maior que do relógio de tempo real, a busca é interrompida (pois a fila encontra-se ordenada).

Antes da inserção do processo na fila de prontos, verificamos se o campo **estado** de seu *workspace* possui o valor um (i.e., processo está bloqueado). Este teste, embora pareça redundante, é realizado porque na implementação da construção ALT (descrita posteriormente) muitas vezes é possível que um mesmo processo fique bloqueado por diversos motivos e, a partir do momento que um desses motivos desaparece, o processo sai do estado de bloqueado e torna-se pronto. Assim, o campo **estado** recebe o valor zero, e mesmo que o *workspace* continue em uma das filas de processos bloqueados, ele não será reescalonado mais de uma vez.

## V.2.6 Comunicação

### Comunicação Interna

Para implementar a troca de mensagens entre processos em execução no i860, o Núcleo oferece duas primitivas RECEBE e ENVIA. Essas primitivas realizam a comunicação entre processos como definido na linguagem Occam, de modo bloqueante e através de canais.

Os canais são previamente alocados pelo usuário. Dois processos que desejam trocar mensagens devem ter alocado o mesmo canal. O primeiro processo a ativar a primitiva RECEBE/ENVIA fica bloqueado até que o segundo processo invoque a primitiva simétrica, quando, então, a comunicação é realizada. Examinando o conteúdo do canal utilizado pelos dois processos é possível saber qual dos dois processos foi o primeiro a ativar a primitiva RECEBE/ENVIA, tornando-se possível desescaloná-lo. O segundo processo, ao ativar a outra primitiva, reescalona o primeiro.

**COPIA**

- Função: Copia um conjunto de bytes de uma posição de memória para outra posição de memória.
- Parâmetros:
  1. nbytes -> número de bytes do bloco a ser copiado.
  2. end_fnt -> endereco inicial onde está localizado o bloco.
  3. end_dst -> endereco inicial da área destino da cópia.

**RECEBE**

- Função: Recebe uma mensagem proveniente de um processo em execução no processador i860
- Parâmetros:
  1. canal -> endereço do canal envolvido na troca da mensagem.
  2. end_destino -> endereço inicial da área de memória para onde a mensagem será copiada.
  3. nbytes -> número total de bytes a ser transferido.

- Algoritmo:

```
begin
  if canal^.ocupado = 0 then
    begin
      Inic_0^.estado := 1;
      canal^.endWS := Inic_0;
      canal^.ocupado := 1;
      Inic_0^.end_msg := end_destino;
      TIRA-PRONTOS;
    end
  else
    begin
      COPIA (nbytes, canal^.endWS^.end_msg, end_destino);
      canal^.ocupado := 0;
      INS_PRONTOS (canal^.endWS);
    end;
end;
```

Quando o campo **ocupado** do canal contiver zero, significa que o processo responsável pelo envio da respectiva mensagem ainda não está preparado para a troca. Por essa razão, o processo corrente será bloqueado. O endereço de seu *workspace* é armazenado no canal e o endereço inicial da área de memória para onde a mensagem deverá ser copiada é armazenado no *workspace* do processo. O campo **ocupado** do canal recebe o valor um (indicando que já existe um processo tentando se comunicar através dele). O campo **estado** do *workspace* do processo também recebe o valor um pois o processo será bloqueado. Em seguida a sub-rotina TIRA-PRONTOS é chamada para retirar o *workspace* do processo da fila de prontos.

Se o conteúdo do campo **ocupado** do canal for igual a um, significa que já existe um processo preparado para a troca. Assim, a mensagem pode ser copiada a partir do endereço armazenado no canal para o endereço que o processo

corrente passou como parâmetro. Após a mensagem ter sido copiada, o campo **ocupado** do canal é reinicializado com zero e o *workspace* do processo bloqueado é inserido novamente na fila de prontos.

**ENVIA**

- Função: Envia uma mensagem para um outro processo em execução no processador i860
- Parâmetros:
  1. canal -> endereço do canal envolvido na troca da mensagem.
  2. end -> endereço inicial onde está localizada a mensagem a ser enviada.
  3. nbytes -> número total de bytes a ser transferido.
- Algoritmo:

```
begin
  if canal^.ocupado = 0 then
    begin
      if canal^.habilitado = 1 then
        if canal^.endWS^.estado = 1 then
          begin
            canal^.endWS^.estado := 0;
            INS_PRONTOS (canal^.endWS);
          end;
      Inic_0^.estado := 1;
      canal^.endWS := Inic_0;
      canal^.ocupado := 1;
      canal^.endWS^.end_msg := end;
      TIRA-PRONTOS;
    end
```

```
    else
      begin
        COPIA (nbytes, end, canal^.endWS^.end_msg);
        canal^.ocupado := 0;
        INS_PRONTOS (canal^.endWS);
      end;
  end;
```

A implementação da primitiva ENVIA é simétrica à da primitiva RECEBE com uma única diferença que será vista a seguir.

A diferença na realização das primitivas RECEBE e ENVIA está na implementação da construção ALT. Conforme a definição dessa construção (vide Capítulo anterior), o guarda de um comando pode incluir uma entrada. Consequentemente deve existir um outro processo responsável pelo envio da mensagem para esse guarda. Esse tipo de troca de mensagens não pode ser realizado diretamente pelas primitivas RECEBE e ENVIA, uma vez que a decisão de realizar ou não a comunicação é tomada durante a execução da construção ALT. Por essa razão, foram introduzidas algumas modificações na primitiva ENVIA.

Quando um processo vai enviar uma mensagem para um guarda, duas situações podem ocorrer: as condições do guarda já foram avaliadas, ou não.

- Se a condição já foi avaliada, então quando da avaliação o guarda não foi considerado **pronto**, é possível, portanto, que a construção ALT esteja suspensa. A execução da primitiva ENVIA torna o guarda **pronto** e se a construção ALT estava suspensa (campo **estado** do *workspace* = 1), o processo que a estava executando pode ser reescalonado (i.e. seu *workspace* será inserido na fila de prontos). Mesmo se o guarda estiver pronto, a comunicação não é realizada de imediato e o processo que executou a primitiva ENVIA é bloqueado. A realização da comunicação e o desbloqueamento do processo ficam a cargo da primitiva DESB-CANAL (explicada na próxima seção).

- Se a condição do guarda ainda não foi avaliada, o processo é bloqueado e o canal marcado como ocupado para que, quando da avaliação o guarda possa ser considerado **pronto**.

### Comunicação Externa

A comunicação entre processos executando em processadores diferentes (i.e., i860 e T-800) pode ocorrer porque os processos desejam trocar mensagens, ou porque um processo do T-800 necessita de um serviço do Núcleo do i860.

Primeiramente veremos como foi implementada a troca de mensagens entre os dois processadores.

Tal qual a comunicação interna, a comunicação externa é realizada de modo bloqueante e através de canais. Os quatro canais para comunicação externa estão mapeados na memória estática compartilhada. Ainda nesta memória, reservamos duas palavras, canal-Ti e canal-iT, para armazenar o número do canal utilizado na comunicação, no sentido i860—T-800 e T-800—i860.

As interrupções de *hardware* (utilizadas para que um processador possa interromper a execução do outro) foram implementadas de tal forma que, junto com o sinal de interrupção, podemos enviar alguns bits de comando (quatro bits para a interrupção que o T-800 gera no i860 e dois bits para a interrupção gerada pelo i860 no T-800). Assim, quando um processador interrompe o outro consideramos tipos diferentes de interrupções:

- Interrupção gerada no i860:
  - Interrupção classe 0 (comando = 0): gerada quando o T-800 tenta enviar uma mensagem;
  - Interrupção classe 1 (comando = 1): gerada quando o T-800 tenta receber uma mensagem;
  - Interrupção classe 2 (comando = 2): gerada quando o T-800 requer a execução de um serviço do Núcleo do i860.

- Interrupção gerada no T-800:

  - Interrupção classe 0 (comando = 0): gerada quando o i860 tenta enviar uma mensagem;
  - Interrupção classe 1 (comando = 1): gerada quando o i860 tenta receber uma mensagem;
  - Interrupção classe 2 (comando = 2): gerada para avisar ao T-800 do término de um serviço;
  - Interrupção classe 3 (comando = 3): gerada para avisar ao T-800 do término de uma rotina.

A a troca de mensagens entre os dois processadores é realizada da seguinte forma:

1) T-800 envia e i860 recebe:

   a) Processo do T-800 chegou primeiro -> O processo do T-800 é bloqueado. Quando o processo do i860 executa o *receive*, o Núcleo verifica que o *send* correspondente já foi executado. A cópia da mensagem é feita e uma interrupção (classe 1) é gerada para o T-800. Este, ao receber a interrupção, reescalona o processo que estava bloqueado.

   b) Processo do i860 chegou primeiro -> O processo do i860 é desescalonado e o i860 envia uma interrupção (classe 1) para o T-800. O T-800 recebe a interrupção e posiciona um *flag* (indicando que o i860 já fez o *receive* através daquele canal). Quando a primitiva ENVIA for executada no T-800, o endereço da mensagem é armazenado no canal e ele interrompe o i860 (classe 1). Ao chegar uma interrupção classe 1, o i860 reescalona o processo que estava bloqueado e a mensagem é copiada.

2) i860 envia e T-800 recebe: Este tipo de comunicação é realizado da mesma forma análoga ao caso anterior. Sendo que as interrupções geradas são de classe 0.

A Figura V.6 mostra o esquema geral da comunicação entre os dois processadores.

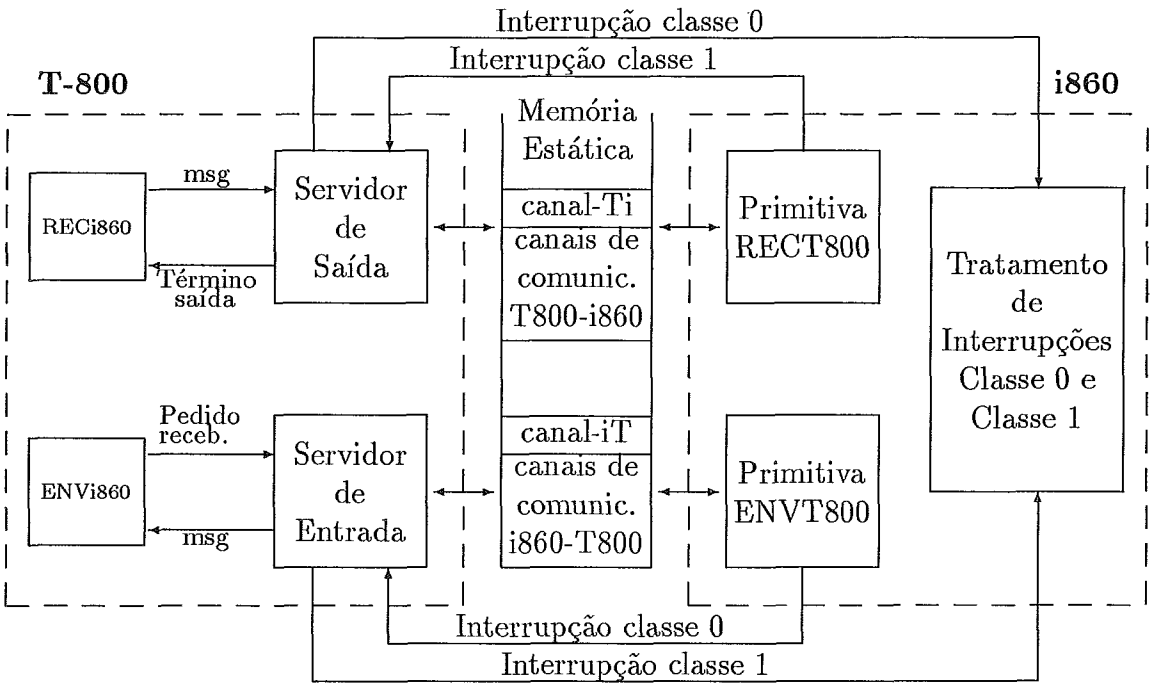



Figura V.6: Comunicação Externa

O usuário do T-800 que deseja trocar mensagens com um processo em execução no i860 deve chamar, respectivamente, as macros ENVi860 ou RECi860 do T-800, passando-lhes os devidos parâmetros (número de bytes, canal e endereço da mensagem). Essas macros trocam mensagens com os processos servidores de saída e entrada (respectivamente) e mantêm os processos que as chamaram bloqueados até que a comunicação tenha sido realizada.

A Figura V.7 apresenta as rotinas da macro ENVi860 juntamente com o processo servidor de saída.

A macro ENVi860 armazena, no devido canal, o número de bytes e o endereço da mensagem, além de posicionar o *flag* **pedido** indicando ao i860 que o T-800 já executou o *send*. Em seguida, ela envia a mensagem ao processo servidor de saída (pelo canal `echmsg`) e espera (pelo canal `echok`) o recebimento de uma mensagem qualquer do servidor (a chegada desta significa que a comunicação já foi realizada e assim o processo pode continuar). A comunicação somente é realizada

ENVi860(canal,nbytes,msg)

```
IF
  canal = 1
    canal1.pedido := 1
    canal1.nbytes := nbytes
    canal1.mens := msg
    echmsg1 !  msg
    echok1 ?  sinal
  canal = 2
    canal2.pedido := 1
    canal2.nbytes := nbytes
    canal2.mens := msg
    echmsg2 !  msg
    echok2 ?  sinal
```

Processo Servidor de Saída

```
SEQ
  flag1 := FALSE
  flag2 := FALSE
  WHILE TRUE
    ALT
      Chegou interrupcao(classe 1)
       IF
         canal-Ti = 1
           flag1 := TRUE
         canal-Ti = 2
           flag2 := TRUE
      flag1 & echmsg1 ?  msg
       echok1 !  0
       flag1 := FALSE
       canal-Ti := 1
       IF
         canal1.bloq = 1
           Interrompe o i860(classe 0)
      flag2 & echmsg2 ?  msg
       echok2 !  0
       flag2 := FALSE
       canal-Ti := 2
       IF
         canal2.bloq = 1
           Interrompe o i860(classe 0)
```

Figura V.7: Rotinas do T-800 para Envio de Mensagem do i860

após o pedido de envio da mensagem pelo T-800 (uma mensagem pelo canal `echmsg`) e a chegada de uma interrupção classe 1 proveniente do i860. Depois disso, o T-800 libera a macro ENVi860 (enviando pelo canal `echok`) e interrompe o i860 caso o *flag* **bloq** seja verdadeiro, i.e., existe um processo no i860 bloqueado esperando pela comunicação.

As rotinas da macro RECi860 e do processo servidor de entrada estão na Figura V.8

RECi860(canal,msg)

```
IF
  canal = 1
    canal1.pedido := 1
    rchreq1 !  0
    rchmsg1 ?  msg
  canal = 2
    canal2.pedido := 1
    rchreq2 !  0
    rchmsg2 ?  msg
```

Processo Servidor de Entrada

```
SEQ
  flag1 := FALSE
  flag2 := FALSE
  WHILE TRUE
    ALT
      Chegou interrupcao(classe 0)
       IF
         canal-iT = 1
           flag1 := TRUE
         canal-iT = 2
           flag2 := TRUE
      flag1 & rchreq1 ?  sinal
       rchmsg1 !  msg
       flag1 := FALSE
       canal-iT := 1
       IF
         canal1.bloq = 1
           Interrompe o i860(classe 1)
      flag2 & rchreq2 ?  sinal
       rchmsg2 !  msg
       flag2 := FALSE
       canal-iT := 2
       IF
         canal2.bloq = 1
           Interrompe o i860(classe 1)
```

Figura V.8: Rotinas do T-800 para Recebimento de Mensagem do i860

A macro RECi860 posiciona o *flag* pedido do canal (para indicar ao i860 que o T-800 já tentou receber a mensagem), envia uma mensagem qualquer (pelo canal `rchreq`) para o processo servidor de entrada (informando que o pedido de recebimento da mensagem já foi realizado) e fica esperando o recebimento da mensagem pelo canal `rchmsg`, mantendo, assim, o processo bloqueado até que a mensagem do i860 tenha chegado. O servidor de entrada tem funcionamento idêntico ao servidor de saída.

O usuário do i860 que deseja trocar mensagens com um processo em execução no T-800 deve ativar, respectivamente, as primitivas do Núcleo ENVT800 ou RECT800. Como o T-800 avisa ao i860 do envio e recebimento de mensagens através de interrupções classe 0 e classe 1 (respectivamente), o Núcleo deve estar preparado para tratar essas duas classes de interrupção também.

## TRATAMENTO DE UMA INTERRUPÇÃO CLASSE 0

```
begin
  case canal-Ti
    1: canal := endereco do primeiro canal;
    2: canal := endereço do segundo canal;
  if canal^.bloq <> 0 then
    begin
      COPY(canal^.nbytes, canal^.mens, canal^.WS^.end_msg);
      INS-PRONTOS(canal^.WS)
      canal^.bloq := 0;
    end;
  canal^.pedido := 0;
end;
```

Pelo conteúdo de canal-Ti sabemos em qual dos dois canais o T-800 está tentando enviar. Se o *flag* **bloq** do canal for verdadeiro é porque existe, no i860, um processo bloqueado esperando pela comunicação. Como a interrupção chegou, significa que o T-800 já tentou enviar. Nesse caso, portanto a mensagem pode ser copiada e o processo, que estava bloqueado, pode ser reescalonado. A seguir, os *flags* **bloq** e **pedido** são reinicializados.

## TRATAMENTO DE UMA INTERRUPÇÃO CLASSE 1

```
begin
  case canal-iT
    1: canal := endereco do primeiro canal;
    2: canal := endereco do segundo canal;
  if canal^.bloq <> 0 then
    begin
      INS-PRONTOS(canal^.WS)
      canal^.bloq := 0;
    end;
  canal^.pedido := 0;
end;
```

Pelo conteúdo da variável canal-iT sabemos qual o canal em questão. O *flag* **bloq** do canal sendo verdadeiro ligado significa que exite um processo bloqueado, esperando pela comunicação. Este processo é então reescalonado (a cópia da mensagem é tarefa do T-800) e os *flags* **bloq** e **pedido** são reinicializados.

## ENVT800

- Função: Envia uma mensagem a um processo em execução no processador T-800.
- Parâmetros:
  1. canal -> número do canal envolvido na troca da mensagem.
  2. end-fonte -> endereço inicial onde está localizada a mensagem a ser enviada.
  3. nbytes -> número total de bytes a ser transferido.

- Algoritmo:

```
begin
  canal^.nbytes := nbytes;
  canal^.mens := end_fonte;
  if canal^.pedido = 0 then
    begin
      canal^.WS := Inic_0;
      canal^.bloq := 1;
      BLOQUEIA;
    end
  else
    begin
      canal^.pedido := 0;
      canal^.bloq := 0;
    end;
  canal-iT := canal;
  Interrompe o T-800 (classe 0);
end;
```

O endereço da mensagem e o número de bytes a serem enviados são passados para a variável canal. Se o *flag* **pedido** do canal for falso, significa que o processo do T-800 responsável pelo recebimento da mensagem ainda não executou, portanto, armazena-se no canal o endereço do *workspace* do processo (do i860) e o *flag* **bloq** é posicionado indicando que existe um processo bloqueado nesse canal. O processo é, então, desescalonado.

Se o *flag* **pedido** for verdadeiro então o T-800 já realizou o pedido pelo recebimento da mensagem. A cópia da mensagem é tarefa do T-800 e os *flags* podem ser reinicializados.

Em ambos os casos o i860 armazena em canal-iT o número do canal

em questão e interrompe o T-800 (classe 0) para avisá-lo que o pedido pelo envio da mensagem já foi realizado.

### RECT800

- Função: Recebe uma mensagem proveniente de processo em execução no processador T-800.
- Parâmetros:
  1. canal -> número do canal envolvido na troca da mensagem.
  2. end-destino -> endereço inicial onde será localizada a mensagem a ser recebida.
  3. nbytes -> número total de bytes a ser transferido.
- Algoritmo:

```
begin
  if canal^.pedido = 0 then
    begin
      canal^.WS := Inic_0;
      canal^.mens := end-destino;
      canal^.bloq := 1;
      BLOQUEIA;
    end
  else
    begin
      canal^.pedido := 0;
      canal^.bloq := 0;
    end;
  canal-Ti := canal;
  Interrompe o T-800 (classe 1);
end;
```

A primitiva RECT800 tem funcionamento análogo a primitiva ENV-T800, sendo que, no caso do *flag* **pedido** estar posicionado, ela é responsável pela cópia da mensagem. E, ainda, o endereço da mensagem só será armazenado no canal se o flag for falso (i.e., i860 executou primeiro).

Veremos agora como foi implementada a requisição de serviço ao Núcleo do i860.

A estrutura de dados utilizada é formada por duas palavras da memória estática compartilhada. São elas:

- SERVIÇO - armazena o número relativo ao serviço requerido:
  - **1** -> iniciar um processo;
  - **2** -> terminar um processo;
  - **3** -> manter um processo bloqueado durante um determinado intervalo de tempo.
- ENDWS - armazena o endereço do *workspace* do processo do i860 sobre o qual o serviço será executado.

Os serviços um e três requerem, ainda, que o usuário armazene no *workspace* do processo os seguintes valores:

**Serviço 1** - a prioridade com que o processo vai executar (deve ser armazenada no campo **prioridade** do *workspace*).

**Serviço 3** - o intervalo de tempo que o processo deve permanecer bloqueado (deve ser armazenado no campo **tempo** do *workspace*).

A Figura V.9 apresenta o esquema da requisição de serviços ao Núcleo do i860.

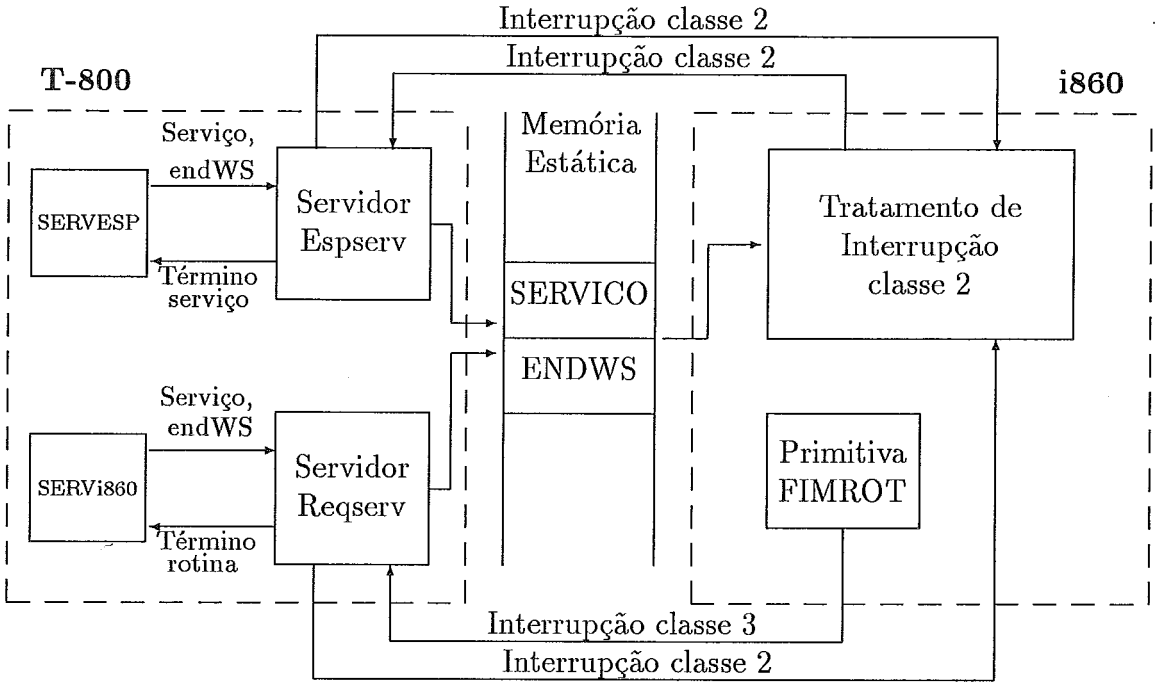



Figura V.9: Requisição de Serviços

O usuário do T-800 pode requerer um serviço do Núcleo do i860 através da chamada da macro SERVi860 ou da macro SERVESP. Para ambas os parâmetros são: número do serviço e endereço do *workspace* do processo no i860.

A macro SERVi860 se comunica com o processo servidor Reqserv e mantém o processo que a chamou bloqueado enquanto o Núcleo do i860 executa o serviço requerido.

A macro SERVESP se comunica com o processo servidor Espserv e é utilizada para o seguinte caso: o usuário do T-800 solicita a criação de um processo no i860 e deseja permanecer bloqueado até que o processo, iniciado no i860, tenha terminado a sua execução. Para isso é preciso que a rotina a ser executada no i860, ao terminar, ative a primitiva FIMROT do Núcleo. Esta se encarrega de interromper o processador T-800.

As rotinas da macro SERVi860 e do processo servidor Reqserv são apresentadas na Figura V.10.

A macro SERVi860 envia ao servidor Reqserv o número do serviço

```
SERVi860(servico,endWS)
chserv !  serv
chserv !  endWS
chterm ?  sinal
```

```
Processo Servidor Reqserv
ALT
  chserv ?  numero
    SERVICO := numero
    chserv ?  endWS
    ENDWS := endWS
    Interrompe o i860(classe 2)
  Chegou interrupcao(classe 2)
    chterm !  sinal
```

Figura V.10: Rotinas do T-800 para Requisição de Serviços

requerido e o endereço do *workspace* do processo relacionado com o serviço que será executado. A seguir, ela espera uma mensagem qualquer pelo canal `chterm`. O processo servidor, por sua vez, ao receber pelo canal `chserv` escreve os valores dos parâmetros na memória compartilhada e interrompe o i860 (classe 2). Ao chegar uma interrupção classe 2 do i860, o serviço requerido já foi executado, e a macro SERVi860 será liberada (pelo envio de uma mensagem pelo canal `chterm`).

As rotinas da macro SERVESP e do processo servidor Espserv são apresentadas na Figura V.11.

```
SERVESP(servico,endWS)
chepserv !  serv
chepserv !  endWS
chepterm ?  sinal
```

```
Processo Servidor Espserv
ALT
  chepserv ?  numero
    SERVICO := numero + 3
    chepserv ?  endWS
    ENDWS := endWS
    Interrompe o i860(classe 2)
  Chegou interrupcao(classe 3)
    chepterm !  sinal
```

Figura V.11: Rotinas do T-800 para Esperar o Término de um Processo no i860

O funcionamento da macro SERVESP e do processo servidor Espserv é análogo ao da macro SERVi860 com o servidor Reqserv. A diferença entre eles está na chegada da interrupção. O servidor Espserv espera uma interrupção classe 3 (significa que uma rotina terminou sua execução no i860). Em SERVIÇO é armazenado um valor maior do que três para que o i860 não gere uma interrupção classe 2 no final da execução do serviço.

Pelo lado do i860, para que o Núcleo possa atender os serviços requeridos pelos usuários do T-800, ele deve oferecer a primitiva FIMROT (para que um processo possa notificar seu término ao T-800) e realizar o tratamento da interrupção do T-800 classe 2, ativando as primitivas responsáveis pela execução do serviço requerido:

- Serviço 1: a primitiva INICPROC insere um *workspace* na fila de prontos.
- Serviço 2: a primitiva TERMPROC retira da fila de prontos um *workspace* dado.
- Serviço 3: a primitiva ESPPROC retira um dado *workspace* da fila de prontos e o insere na fila de processos esperando por um intervalo de tempo.

## TRATAMENTO DE UMA INTERRUPÇÃO CLASSE 2

```
begin
  if (SERVICO = 1) or (SERVICO = 4) then
    INICPROC(ENDWS)
  else if (SERVICO = 2) or (SERVICO = 5) then
         TERMPROC(ENDWS)
       else if (SERVICO = 3) or (SERVICO = 6) then
              ESPPROC(ENDWS);
  if SERVICO < 4 then
    Interrompe o T-800 (classe 2);
end;
```

O valor armazenado em SERVIÇO indica o número do serviço requerido (se SERVICO $>$ 3, o número do serviço requerido é dado por SERVICO - 3). Para cada valor de SERVIÇO o Núcleo ativa a primitiva respectiva. Ao terminar a execução da primitiva, o Núcleo somente gerará a interrupção (classe 2) se SERVIÇO $<$ 4, i.e, o usuário do T-800 está esperando pelo fim do serviço.

## INICPROC

- Função: Insere, a partir de uma requisição de um processo em execução no T-800, um *workspace* na fila de prontos.
- Parâmetros:
  1. end_WS -> endereço do *workspace* do processo a ser iniciado.
- Algoritmo:

```
begin
  INS-PRONTOS(end_WS);
end;
```

Como o usuário do T-800 já armazenou no *workspace* do processo a prioridade com que ele vai executar, basta chamar a rotina INS-PRONTOS para inserir o *workspace* na fila devida.

## TERMPROC

- Função: Retira, a partir de uma requisição de um processo em execução no T-800, um *workspace* da fila de prontos.
- Parâmetros:
  1. end_WS -> endereço do *workspace* do processo a ser retirado.
- Algoritmo:

```
var
  i, frente: pontws;
```

```
    achou: boolean;
  begin
    i := Inic_0;
    frente := nil;
    achou := false;
    while (i <> nil) and (NOT achou)
      if i = end_WS then
        achou := true
      else
        begin
          frente := i;
          i := i^.ant;
        end;
    if achou then
      if i = Inic_0 then
        TIRA-PRONTOS
      else
        begin
          if i = Inic_1 then
            Inic_1 := i^.ant;
          if i = Fim_1 then
            Fim_1 := frente;
          if i = Fim_0 then
            Fim_0 := frente;
          frente^.ant := i^.ant;
        end;
  end;
```

Para retirar um *workspace* qualquer da fila de prontos, precisamos primeiro percorrê-la para descobrir qual é a posição que ele ocupa.

Se o processo a ser retirado é o primeiro da fila, então basta chamar a rotina TIRA-PRONTOS.

Caso contrário, a retirada do processo $i$ é feita da seguinte maneira: o campo **ant** do processo que precede $i$ deve apontar para o antecessor de $i$. É testado ainda se o processo a ser retirado ocupa a posição de início da fila de não-urgentes ou de fim de uma das duas filas, pois, em caso afirmativo, as variáveis Inic_1, Fim_1 e Fim_0 devem ser atualizadas.

## ESPPROC

- Função: Mantém um processo bloqueado durante um intervalo de tempo, a partir de uma requisição de um processo em execução no T-800.
- Parâmetros:
  1. endWS -> endereço do *workspace* do processo.
- Algoritmo:

```
begin
  endWS^.tempo := endWS^.tempo + Hora_atual;
  TERMPROC(endWS);
  INS-TMP(endWS);
end;
```

Primeiramente, o *workspace* do processo deve ser retirado da fila de prontos. Isso é feito pela primitiva TERMPROC. Em seguida, o *workspace* é inserido na fila de processos esperando por um intervalo de tempo.

## FIMROT

- Função: Avisa ao T-800 do término do processo.
- Parâmetros: Nenhum.

- Algoritmo:

```
begin
  Interrompe o T-800 (classe 3)
end
```

Para que um processo em execução no T-800 possa permanecer bloqueado durante a execução de um processo no i860, o Núcleo oferece uma primitiva que, quando chamada no fim do processo, avisa ao T-800 do seu término.

### V.2.7 Implementação do ALT

A implementação da construção ALT envolve a avaliação dos guardas e a escolha de um comando guardado cuja lista de comandos será executada.

A avaliação dos guardas é feita pelas primitivas de habilitação:

- HAB-CANAL (para guardas que possuem um comando de entrada).
- HAB-TMP (para guardas que possuem uma espera por um intervalo de tempo).
- HAB-SKIP (para guardas que possuem um processo SKIP).

Essas primitivas verificam se o guarda está pronto. Caso esteja, um *flag* é posicionado indicando que existe pelo menos um guarda pronto e assim o processo que executou a construção ALT não precisa ficar suspenso. Se o guarda não estiver pronto, as primitivas de habilitação comportam-se assim:

- Um guarda que possui um processo SKIP está sempre pronto;
- A primitiva HAB-CANAL marca o canal utilizado pelo guarda como "habilitado para uma entrada";
- A primitiva HAB-TMP armazena no *workspace* do processo o intervalo de tempo que se deseja esperar (esse valor será utilizado se o processo for inserido na fila de processos esperando por intervalo de tempo).

As primitivas SUSP-ALT e SUSP-ALTTMP são utilizadas para suspender o processo caso nenhum dos guardas avaliados esteja pronto (*flag* = falso). A primitiva SUSP-ALTTMP é utilizada no lugar da primitiva SUSP-ALT se algum dos guardas estiver esperando que uma hora seja alcançada, pois além de suspender o processo, ela insere seu *workspace* na fila de processos esperando por um intervalo de tempo.

Assim que o processo for reescalonado (caso ele tenha sido suspenso), as primitivas de inabilitação (DESB-CANAL, DESB-TMP, DESB-SKIP) fazem a seleção do comando guardado (que será executado) e inabilitam os outros comandos. O primeiro comando guardado, cujo guarda estiver pronto, a ser inabilitado é o escolhido (o endereço da primeira instrução da lista de comandos é armazenado no *workspace* do processo). Para o comando escolhido a primitiva DESB-CANAL realiza, ainda, a comunicação requerida pelo guarda. Para os outros comandos guardados a primitiva DESB-CANAL impede que a comunicação do guarda seja realizada. As outras primitivas não têm qualquer efeito.

A primitiva FIM-ALT termina uma construção ALT desviando o controle para a lista de comandos da alternativa selecionada.

**INIC-ALT**

- Função: Inicia uma construção ALT
- Parâmetros: Nenhum
- Algoritmo:

```
begin
  Inic_0^.prt := 0;
end;
```

No *workspace* do processo, que deseja iniciar a execução de uma construção ALT, o campo **prt** é inicializado com zero. Este campo indicará se há pelo menos um guarda pronto (nesse caso, não há necessidade de se retirar o processo da fila de prontos).

**HAB-CANAL**

- Função: Avalia um guarda que possui um comando de entrada.
- Parâmetros:
  1. canal -> endereço do canal envolvido na troca da mensagem.
- Algoritmo:

```
begin
  if canal^.ocupado = 0 then
    begin
      canal^.habilitado := 1;
      canal^.endWS := Inic_0;
    end
  else
    Inic_0^.prt := 1;
end;
```

Se o campo **ocupado** do canal contiver zero, então não existe, por enquanto, nenhum processo desejando enviar uma mensagem por este canal. O campo **habilitado** do canal recebe o valor um para indicar que o canal está "habilitado para uma entrada" e o endereço do *workspace* do processo corrente é armazenado no canal. O campo **ocupado** do canal continua com zero de modo a retardar uma troca de mensagens através desse canal.

Se o campo **ocupado** do canal em questão estiver com o valor um, então existe um processo, que encontra-se bloqueado, preparado para enviar uma mensagem pelo mesmo canal. Neste caso, o guarda está pronto e o campo **prt** do *workspace* do processo corrente deve receber o valor um, mas ainda assim a comunicação não é realizada imediatamente.

**HAB-TMP**

- Função: Avalia um guarda que possui uma espera por intervalo de tempo.
- Parâmetros:
  1. tempo -> intervalo de tempo que se deseja esperar.
- Algoritmo:

```
begin
  if tempo > Inic_0^.tempo then
    Inic_0^.tempo := tempo;
end;
```

Um guarda que contém uma espera por um intervalo de tempo, nunca se encontra, de imediato, pronto. O valor do intervalo de tempo de espera é armazenado no campo **tempo** do *workspace* do processo corrente. Para o caso de uma construção ALT com mais de um guarda contendo espera por intervalo de tempo, permanece armazenado no *workspace* do processo somente o valor do menor intervalo de tempo.

**HAB-SKIP**

- Função: Avalia um guarda que possui um processo SKIP.
- Parâmetros: Nenhum.
- Algoritmo:

```
begin
  Inic_0^.prt := 1;
end;
```

Como o processo SKIP não realiza qualquer ação, o guarda está pronto e assim o campo **prt** do *workspace* do processo corrente recebe o valor um.

**SUSP-ALT**

- Função: Retira o processo da fila de prontos caso não haja nenhum guarda pronto
- Parâmetros: Nenhum.
- Algoritmo:

```
begin
  if Inic_0^.prt = 0 then
    begin
      Inic_0^.estado := 1;
      DESESCALONA (Inic_0, end_volta);
    end
```

```
  else
    Inic_0^.prt := 0;
end;
```

Se o valor armazenado no campo **prt** do *workspace* do processo corrente for igual a zero, é porque nenhum dos guardas envolvidos na construção ALT está pronto. O processo corrente deve ficar bloqueado até que algum guarda torne-se pronto. Portanto, o campo **estado** de seu *workspace* recebe o valor um, seu contexto é salvo e o *workspace* é retirado da fila de prontos através da primitiva DESESCALONA.

Se o valor armazenado no campo **prt** for igual a um, é porque pelo menos um guarda envolvido na construção está pronto. A primitiva SUSP-ALT, então, não tem qualquer efeito senão o de zerar o campo **prt** para que ele possa ser usado como *flag* novamente.

**SUSP-ALTTMP**

- Função: Retira o processo da fila de prontos, transferindo-o para a a fila de processos esperando por um intervalo de tempo, caso não haja nenhum guarda pronto.
- Parâmetros: Nenhum.
- Algoritmo:

```
var
  aux : integer;
begin
  if Inic_0^.prt = 0 then
    aux := Inic_0^.tempo
```

```
      Inic_0^.tempo := Hora_atual
      DELAY (aux)
    else
      Inic_0^.prt := 0;
end;
```

Essa primitiva é utilizada quando existe pelo menos um guarda esperando por intervalo de tempo.

A função dessa primitiva é análoga à da primitiva SUSP-ALT, com uma diferença no que diz respeito à retirada do processo da fila de prontos (quando prt=0). A primitiva TATLWAIT retira o processo corrente da fila de prontos e insere seu *workspace* na fila de processos esperando por intervalo de tempo, através da primitiva DELAY (o intervalo de tempo de espera está armazenado no campo tempo do *workspace* do processo corrente).

**DESB-CANAL**

- Função: Inabilita um guarda que possui uma entrada.
- Parâmetros:
  1. canal -> endereço do canal envolvido na troca da mensagem.
  2. nbytes -> número de bytes da mensagem.
  3. end_msg -> endereço inicial da área de memória para onde a mensagem será copiada.
  4. end_lista -> endereço da primeira instrução da lista de comandos do guarda.

- Algoritmo:

```
begin
  if Inic_0^.prt = 0 then
    begin
      if canal^.ocupado = 1 then
        begin
          Inic_0^.prt := 1;
          Inic_0^.lista-cmd := end_lista;
          COPIA (nbytes, canal^.endWS^.end_msg, end_msg);
          canal^.habilitado := 0;
          canal^.ocupado := 0;
          INS_PRONTOS (canal^.endWS);
        end;
    end;
end;
```

Se o campo **prt** do *workspace* do processo corrente estiver com zero, então, por enquanto, nenhum comando guardado foi escolhido. Para que este comando guardado possa ser escolhido, o guarda respectivo deve estar pronto. Como essa primitiva trata de guardas que envolvem comunicação, o guarda estará pronto se o valor do campo **ocupado** do canal em questão for igual a um (pois significa que existe um outro processo preparado para realizar uma troca de mensagem por este canal). Sendo este o primeiro comando guardado inabilitado (cujo guarda está pronto), ele é o escolhido. O campo **prt** do *workspace* do processo recebe o valor um indicando que já existe uma alternativa escolhida. No *workspace* é armazenado o endereço da primeira instrução da lista de comandos que será executada. A comunicação do guarda é realizada (a mensagem é copiada) e o processo que estava bloqueado, esperando para realizar a saída, é reescalonado, ou melhor, é inserido na fila de prontos.

Se a alternativa não foi escolhida (porque ao inabilitar o canal um

comando guardado anterior já havia sido escolhido), então mesmo que o guarda esteja pronto, a comunicação deste guarda não é realizada.

**DESB-TMP**

- Função: Inabilita um guarda que inclui uma espera por intervalo de tempo.
- Parâmetros:
  1. tempo -> hora a ser alcançada pelo processo.
  2. end_lista -> endereço da primeira instrução da lista de comandos do guarda.
- Algoritmo:

```
begin
  if Inic_0^.prt = 0 then
    if Hora_atual > tempo then
      begin
        Inic_0^.prt := 1;
        Inic_0^.lista-cmd := end_lista;
      end;
end;
```

Se o campo **prt** do *workspace* do processo corrente estiver com zero, então, por enquanto, nenhum comando guardado foi escolhido. Se o valor do campo tempo (hora a ser alcançada pelo processo) for menor que o valor do relógio de tempo real (Hora_atual) então o guarda está pronto e o comando guardado do qual este guarda faz parte é o escolhido. O campo **prt** do *workspace* do processo corrente recebe o valor um indicando que já existe um comando guardado escolhido. No *workspace* é armazenado o endereço da lista de comandos que será executada.

**DESB-SKIP**

- Função: Inabilita um guarda que possui um processo SKIP.
- Parâmetros:
  1. end_lista -> endereço da primeira instrução da lista de comandos do guarda.
- Algoritmo:

```
begin
  if Inic_0^.prt = 0 then
    begin
      Inic_0^.prt := 1;
      Inic_0^.lista-cmd := end_lista;
    end;
end;
```

Como um guarda que possui um SKIP está sempre pronto, se nenhum comando guardado tiver sido escolhido anteriormente, então a alternativa que inclui este guarda é a escolhida. O campo **prt** do *workspace* do processo recebe, então, o valor um. No mesmo é armazenado, também, o endereço da primeira instrução da lista de comandos que será executada.

**FIM-ALT**

- Função: Término de uma construção ALT.
- Parâmetros: Nenhum.

- Algoritmo:

```
begin
  goto Inic_0^.lista-cmd;
end;
```

Realiza o desvio de controle para a primeira instrução da lista de comandos da alternativa escolhida para executar.

## V.3 Medidas de Desempenho

### V.3.1 Número de Instruções Executadas

As tabelas a seguir mostram o número de instruções executadas por cada primitiva do Núcleo.

| INIC-PAR | |
|---|---|
| As filas estão cheias ou vazias | 152 instruções |
| O processo a ser iniciado tem prioridade maior que o processo corrente | 404 instruções |

Tabela V.1: Número de Instruções da Primitiva INIC-PAR

| FIM-PAR | | |
|---|---|---|
| Término de todos menos o último processo de uma construção PAR | a fila de urgentes está vazia | 126 instruções |
| | a fila de urgentes não está vazia | 123 instruções |
| Término do último processo de uma construção PAR | a fila de urgentes está vazia | 229 instruções |
| | a fila de urgentes não está vazia | 226 instruções |
| | o processo a ser iniciado interrompe o processo corrente | 481 instruções |

Tabela V.2: Número de Instruções da Primitiva FIM-PAR

| DESESCALONA | |
|---|---|
| A fila de urgentes está vazia | 368 instruções |
| A fila de urgentes não está vazia | 365 instruções |

Tabela V.3: Número de Instruções da Primitiva DESESCALONA

| FATIA-TEMPO | | |
|---|---|---|
| O processo corrente é urgente | | 90 instruções |
| O processo corrente é não-urgente | a fatia de tempo não terminou | 99 instruções |
| | a fatia de tempo terminou e só há um processo na fila | 99 instruções |
| | a fatia de tempo terminou e há mais de um processo na fila | 368 instruções |

Tabela V.4: Número de Instruções da Sub-rotina FATIA-TEMPO

| DELAY | | |
|---|---|---|
| A fila de delay está vazia | | 390 instruções |
| A fila de delay não está vazia | o processo será o primeiro | 393 instruções |
| | o processo não será o primeiro | 390 instruções mais 8 instruções para cada posição na fila |

Tabela V.5: Número de Instruções da Primitiva DELAY

| BUSCA-TMP | | |
|---|---|---|
| A fila de delay está vazia | | 86 instruções |
| A fila de delay não está vazia | o primeiro processo não pode ser reescalonado | 92 instruções |
| | há processos a serem reescalonados | 102 instruções mais 32 instruções para cada processo |

Tabela V.6: Número de Instruções da Sub-rotina BUSCA-TMP

| Implementação da Construção ALT | | |
|---|---|---|
| INIC-ALT | | 109 instruções |
| HAB-CANAL | o guarda está pronto | 117 instruções |
| | o guarda não está pronto | 119 instruções |
| HAB-TMP | | 127 instruções |
| HAB-SKIP | | 125 instruções |
| SUSP-ALT | existe um guarda pronto | 118 instruções |
| | não há guardas prontos | 414 instruções |
| SUSP-ALTTMP | exite um guarda pronto | 122 instruções |
| | não há guardas prontos | 414 instruções |
| DESB-CANAL | já existe um comando guardado escolhido | 123 instruções |
| | não há comando guardado escolhido e o guarda não está pronto | 127 instruções |
| | não há comando guardado escolhido e o guarda está pronto | 183 instruções mais 7 instruções para cada byte da messagem |
| DESB-TMP | já existe um comando guardado escolhido | 126 instruções |
| | não há comando guardado escolhido e o guarda não está pronto | 131 instruções |
| | não há comando guardado escolhido e o guarda está pronto | 134 instruções |
| DESB-SKIP | | 129 instruções |
| FIM-ALT | | 112 instruções |

Tabela V.7: Número de Instruções das Primitivas para a Construção ALT

| Comunicação Interna | | |
|---|---|---|
| RECEBE | O processo foi o primeiro a tentar a comunicação | 384 instruções |
| | O processo foi o segundo a tentar a comunicação | 164 instruções mais 7 instruções para cada byte da mensagem |
| ENVIA | O processo foi o primeiro a tentar a comunicação | 385 instruções |
| | O processo foi o segundo a tentar a comunicação | 164 instruções mais 7 instruções para cada byte da messagem |
| | A messagem será enviada a um guarda | 149 instruções |

Tabela V.8: Número de Instruções das Primitivas para a Comunicação Interna

| Comunicação Externa | | |
|---|---|---|
| RECi860 | T800 executa primeiro | 233 instruções mais 7 instruções para cada byte da menssagem |
| | i860 executa primeiro | 564 instruções mais 7 instruções para cada byte da mensagem |
| ENVi860 | T800 executa primeiro | 244 instruções |
| | i860 executa primeiro | 544 instruções |

Tabela V.9: Número de Instruções das Primitivas para a Comunicação Externa

### V.3.2 Tempo de Execução

A seguir são apresentados os tempos de execução de algumas primitivas. Comparando estas medidas com as apresentadas anteriormente, verificamos que as primitivas envolvendo o desescalonamento do processo corrente (incluindo, portanto, o salvamento de todo o seu contexto e a restauração do contexto de um outro processo) apresentaram um número de instruções bem mais elevado do que as outras primitivas. Por esse motivo, o tempo de execução da maioria das primitivas do Núcleo é bastante afetado pelas operações de salvamento e restauração de contextos.

| Implementação da Construção PAR | | |
|---|---|---|
| INIC-PAR | as filas estão vazias | 10.43 $\mu$s |
| | as filas não estão vazias | 9.23 $\mu$s |
| FIM-PAR | último processo concorrente a terminar | 30.45 $\mu$s |
| | outros processos | 29.5 $\mu$s |
| DESESCALONA | | 21.27 $\mu$s |
| BLOQUEIA | | 28.90 $\mu$s |

Tabela V.10: Tempo de Execução das Primitivas para a Construção PAR

| Espera por um Intervalo de Tempo | | |
|---|---|---|
| DELAY | a fila de delay está vazia | 30.5 $\mu$s |
| | a fila de delay não está vazia | 28.7 $\mu$s |

Tabela V.11: Tempo de Execução da Primitiva DELAY

| Comunicação Interna | | |
|---|---|---|
| RECEBE | O processo foi o primeiro a tentar a comunicação | 18.41 $\mu$s |
| | O processo foi o segundo a tentar a comunicação | 27.59 $\mu$s |
| ENVIA | O processo foi o primeiro a tentar a comunicação | 18.77 $\mu$s |
| | O processo foi o segundo a tentar a comunicação | 27.60 $\mu$s |

Tabela V.12: Tempo de Execução das Primitivas para Comunicação Interna

| Implementação da Construção ALT | | |
|---|---|---|
| INIC-ALT | | 6.85 $\mu$s |
| HAB-CANAL | | 7.33 $\mu$s |
| HAB-TMP | | 7.33 $\mu$s |
| HAB-SKIP | | 6.97 $\mu$s |
| SUSP-ALT | existe um guarda pronto | 7.09 $\mu$s |
| | não há guardas prontos | 17.58 $\mu$s |
| SUSP-ALTTMP | exite um guarda pronto | 7.56 $\mu$s |
| | não há guardas prontos | 19.25 $\mu$s |
| DESB-CANAL | já existe um comando guardado escolhido | 15.19 $\mu$s |
| | não há comando guardado escolhido e o guarda não está pronto | 8.04 $\mu$s |
| | não há comando guardado escolhido e o guarda está pronto | 28.90 $\mu$s |
| DESB-TMP | já existe um comando guardado escolhido | 7.33 $\mu$s |
| | não há comando guardado escolhido e o guarda não está pronto | 7.56 $\mu$s |
| | não há comando guardado escolhido e o guarda está pronto | 7.8 $\mu$s |
| DESB-SKIP | | 7.53 $\mu$s |
| FIM-ALT | | 7.56 $\mu$s |

Tabela V.13: Tempo de Execução das Primitivas para a Construção ALT

# Capítulo VI

# Conclusões

Essa tese apresentou a especificação e implementação de um Núcleo provendo facilidades para processamento paralelo. Esse Núcleo é responsável pelo gerenciamento do processador i860, que em conjunto com o Transputer T-800 fornecem a capacidade de processamento de cada um dos nós do NCP-1: máquina paralela organizada segundo um hipercubo, que foi desenvolvida pela equipe de processamento paralelo da COPPE-Sistemas. A versão corrente do Núcleo foi implementada em linguagem *Assembly* do processador i860 enquanto que os servidores residentes no T-800 em linguagem Occam.

As facilidades disponíveis ao usuário incluem a criação e destruição dinâmica de processos concorrentes, primitivas para comunicação entre processos, a construção Alternativa da linguagem Occam e um mecanismo permitindo que um processo fique bloqueado durante intervalos de tempo por ele especificados. Sob o ponto de vista funcional, todas essas facilidades são semanticamente compatíveis com as correspondentes primitivas que são implementadas no processador Transputer.

O objetivo fundamental do trabalho —tornar homogêneo o ambiente para programação paralela em cada um dos nodos de processamento da arquitetura hipercúbica— foi atingido através da criação de primitivas que reproduzem o comportamento das funções implementadas a nível de $\mu$-código no processador T-800. Através dessa compatibilidade funcional, a migração de processos de um tipo de processador para o outro torna-se factível a nível de código fonte. O desenvolvi-

mento de tradutores de linguagens de alto nível para os dois processadores extenderá essa capacidade para o nível de código objeto, permitindo assim que técnicas de balanceamento dinâmico de carga de processamento sejam implementadas no nosso Núcleo.

A implementação no i860 de primitivas de comunicação e sincronização baseadas no modelo adotado pelo Transputer, permite executar processos cooperantes ao longo dos processadores do NCP-1. A facilidade para criação remota de procedimentos no i860, a partir do T-800, é um típico exemplo do potencial para processamento paralelo provido pelo Núcleo. Por exemplo, a partir do T-800, o usuário pode ativar rotinas de uma biblioteca vetorial implementada no i860. Durante a execução dessas rotinas, o programa no T-800 pode continuar sendo processado em paralelo pois existem primitivas no Núcleo capazes de assinalar, através de um *signal* ao respectivo processo do T-800, que a tarefa solicitada já foi concluída.

Os procedimentos responsáveis pelo gerenciamento do barramento VME —canal de comunicação alternativo interconectando todos nodos de processamento do hipercubo— serão codificados e traduzidos para o código objeto do i860. As atividades futuras incluem a validação desses procedimentos tão logo o correspondente projeto de *hardware* esteja concluído. Está previsto também a definição e implementação de rotinas para o gerenciamento dos segmentos de memória compartilhados pelos dois processadores.

Conforme descrito no Capítulo V, as medidas de desempenho das primitivas do Núcleo que foram obtidas pelos nosso experimentos, evidenciam as características do processador i860: uma arquitetura RISC com razoável número de registradores e outros elementos de armazenamento no interior do processador. Quando da comutação de processos, o *overhead* requerido para resguardar/restaurar o conteúdo dos registradores e dos diversos estágios dos *pipelines* é elevado pois envolve a execução de inúmeras instruções do tipo RISC.

É importante enfatizar que muitos obstáculos tiveram que ser superados para que o corrente estágio do Núcleo aqui apresentado fosse atingido. Considerando o aspecto pioneiro do projeto de desenvolvimento de uma arquitetura

de alto desempenho, empregando processadores recentemente lançados no mercado, então fomos forçados a conviver com a falta de informações precisas acerca de detalhes de implementação do i860, manuais omissos, inexistência de um sistema de desenvolvimento, etc. Por essa razão, construímos —em conjunto com o grupo de processamento paralelo da COPPE— algumas ferramentas de *hardware/software* formando o embrião de um sistema de desenvolvimento.

A decisão de definir e implementar um montador (*Assembler*), um simulador e um monitor primitivo para o processador i860 foi duplamente vantajosa: além de atuarem como ferramentas de suporte para o desenvolvimento do nosso ambiente para programação paralela, a implementação desse *software* foi de extrema relevância para que dominássemos os conceitos incorporados nesse novo processador. Fomos igualmente beneficiados com a decisão de construir uma placa i860 compatível com os micro-computadores do tipo IBM-PC. As primitivas do Núcleo foram desenvolvidas e parcialmente depuradas e validadas nessa placa antes de serem transportadas para o nó de processamento do hipercubo.

Extensões do Núcleo para torná-lo compatível com outros padões industriais, como o Sistema Operacional UNIX por exemplo, é uma outra atividade de pesquisa que pode ser conduzida. Duas estratégias poderiam ser empregadas para atingir esse objetivo. A primeira consiste em introduzir no Núcleo novas primitivas reproduzindo a interface requerida pelo UNIX. A outra estratégia seria incorporar no Sistema UNIX as primitivas para programação paralela especificadas nessa tese. Nesse caso, seria necessário alterar um Sistema do tipo UNIX e transportá-lo para o i860 do nó de processamento do hipercubo. Por exemplo, o Sistema MINIX descrito em [33] poderia ser traduzido para o código de máquina do i860 após tê-lo adaptado para processamento paralelo.

# Referências Bibliográficas


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

[2] Babb II, R.G. (Ed.), *Programming Parallel Processors*, Addison-Wesley Publishing Company, 1988.

[3] Bradley, D.K., Nazief, B.A.A.,Grunwald, D.C. e Reed, D.A., "Picasso: An Experiment in Hypercube Operating System Design", *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications* - 1988, pp 364-373.

[4] Burns, A., *Programming in Occam2*, Addison-Wesley Publishing Company, 1988.

[5] Carlile, B.R. e Miles, D.,"Structured Asynchronous Communication Routine for the FPS T Series", *Proceedings of the 3rd Conference on Hypercube Concurrent Computers e Applications* - 1988, pp 550-559.

[6] *Communicating Process Architecture*, Inmos Limited, Prentice Hall, 1988.

[7] Dijkstra, E.W., "Guarded Commands, Non-Determinacy and Formal Derivation of Programs", *Communications of the ACM*, Volume 18, N. 8, Agosto 1975, pp 453-457.

[8] Ezzat, A.K. e Agrawal, R., "Making Oneself Known in a Distributed World", *Proccedings of the International Conference on Parallel Processing* - 1985, pp 139-142.

[9] Fazzari, R.J. e Lynch, J.D., "The Second Generation FPS T Series: A Enhanced Parallel Vector Supercomputer", *Proceedings of then 3rd Conference on Hypercube Computers and Applications* - 1988, pp 61-70.

[10] Fernandes, E.S.T, "Sistemas Operacionais", Notas de Aula do Curso de Informática, UFRJ, Março 1982.

[11] Flynn, M.J., "Very High-Speed Computing Systems", *Proceedings of the IEEE 54*, 1966, pp 1901-1909.

[12] Fox, G.C., Jonhson, M.A., Lyzenga, G.A., Otto, S.W., Salmon, J.K. e Walker, D.W., *Solving Problems On Concurrent Processors*, Volume 1, Prentice Hall, 1988.

[13] Gehringer, E.F. e Harry, B.D., "Rapid Prototyping of a Parallel Operating System for a Generalized Hypercube", *The Second Conference on Hypercube Concurrent Computers and Applications* - 1987, pp 377-380.

[14] Gustafson, H.L., Hawkinson, S. e Scott, K., "The Architecture of a Homogeneous Vector Supercomputer", *Proceedings of the International Conference on Parallel Processing* - 1986, pp 649-652.

[15] Hayes, J.P., Mudge, T., Stout, Q.F., Colley, S. e Palmer J.A., "A Microprocessor-Based Hypercube Supercomputer", *IEEE MICRO* - Outubro 1986, pp 6-17.

[16] *Helios Operating System*, Perihelion Software Limited, Prentice Hall, 1989.

[17] Hoare, C.A.R., "Communicating Sequential Processes", *Communications of the ACM*, Volume 21, N. 8, Agosto 1978, pp 666-677.

[18] *IBM System /360 Operating System Concepts and Facilities*, IBM Corporation, Manual C28-6535.

[19] *i860 64-Bit Microprocessor*, (Número de ordem 240296-002), Intel Corporation, Santa Clara, CA, 1989.

[20] *i860 64-Bit Microprocessor Programmer's Reference Manual*, (Número de ordem 240329-002), Intel Corporation, Santa Clara, CA, 1989.

[21] Kohn, L. e Margulis, N., "Introducing the Intel i860 64-Bit Microprocessor", *IEEE MICRO*, Agosto 1989, pp 15-30.

[22] Krumme, D.W., "The SIMPLEX Operating System", *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications* - 1988, pp 381-383.

[23] Maekawa, M., Oldehoeft, A.E. e Oldehoeft, R.R., *Operating Systems Advanced Concepts*, The Benjamin/Cummings Publishing Company, 1988.

[24] Margulis, N., "The Intel 80860", *BYTE*, Dezembro 1989, pp 333-340.

[25] Mitra-15, *Manuel de Reference–Tome 1*, CII, Junho 1975.

[26] *Occam 2 Reference Manual*, Inmos Limited, Prentice Hall, 1988.

[27] Peterson, J.L. e Silberchatz A., *Operating Systems Concepts*, Addison-Wesley Publishing Company, 1985.

[28] Peterson, J.C., Tuazon, J.O., Lieberman, D. e Pniel, M., "The Mark III Hypercube-Ensemble Concurrent Computer", *Proceedings of the International Conference on Parallel Processing* - 1985, pp 71-73.

[29] Pountain, D., "Occam II", *BYTE*, Outubro 1989, pp 279-284.

[30] Seidel, C.B. e Yamashiro, G., "Um Montador para o i860", Relatório Técnico COPPE/Sistemas UFRJ (em fase de preparação).

[31] Sietz, C.L., "The Cosmic Cube", *Communications of The ACM*, Janeiro 1985, pp 23-25

[32] Tabak, D., *RISC Systems*, Research Studies Press, 1990.

[33] Tanembaum, A.S., *Operating Systems: Design and Implementation*, Prentice Hall, New Jersey, 1987.

[34] *Transputer Development System, User Guide and Reference Manual*, Inmos Limited, Prentice Hall, 1988.

[35] *Transputer Instruction Set: a compiler writer's guide*, Inmos Limited, Prentice Hall, 1988.

[36] *Transputer Technical Notes*, Inmos Limited, Prentice Hall, 1988.

[37] Yamashiro, G., "Projeto e Implementação de uma Biblioteca de Rotinas Vetoriais para o i860", Tese de Mestrado COPPE/Sistemas UFRJ (em fase de preparação).

# Apêndice A

| Primitiva | Número de Chamada | Função | Parâmetros | Registrador |
|---|---|---|---|---|
| INIC-PAR | 7 | Cria um processo concorrente | EndWS | R3 |
| | | | Prim_Inst | R5 |
| BLOQUEIA | 9 | Termina um processo | — | — |
| DESESCALONA | 8 | Termina um processo que mais tarde será reescalonado | end_retorno | R3 |
| FIM-PAR | 10 | Termina um processo concorrente | — | — |
| DELAY | 11 | Bloqueia um processo durante um intervalo de tempo dado | tempo | R3 |
| RECEBE | 12 | Recebe uma mensagem de um processo em execução no i860 | canal | R6 |
| | | | end_destino | R4 |
| | | | nbytes | R5 |
| ENVIA | 13 | Envia uma mensagem p/ um processo em execução no i860 | canal | R6 |
| | | | end_fonte | R4 |
| | | | nbytes | R5 |
| RECi860 | 25 | Recebe uma mensagem de um processo em execução no T-800 | canal | R4 |
| | | | end_destino | R9 |
| | | | nbytes | R5 |
| ENVi860 | 25 | Envia uma mensagem p/ um processo em execução no T-800 | canal | R4 |
| | | | end_fonte | R6 |
| | | | nbytes | R5 |
| FIMROT | 30 | Avisa ao T-800 do término | — | — |
| INIC-ALT | 14 | Inicia uma construção ALT | — | — |
| HAB-CANAL | 15 | Avalia guarda com entrada | canal | R3 |
| HAB-TMP | 18 | Avalia guarda com espera | tempo | R3 |
| HAB-SKIP | 21 | Avalia guarda com SKIP | — | — |
| SUSP-ALT | 16 | Mantém a construção ALT suspensa | — | — |
| SUSP-ALTTMP | 19 | Idem, para guardas c/ espera | — | — |
| DESB-CANAL | 17 | Desabilita guarda que possui uma entrada | canal | R4 |
| | | | nbytes | R5 |
| | | | end_msg | R9 |
| | | | end_comp | R6 |
| DESB-TMP | 20 | Desabilita guarda c/ espera | tempo | R3 |
| | | | end_comp | R4 |
| DESB-SKIP | 22 | Desabilita guarda com SKIP | end_comp | R5 |
| FIM-ALT | 23 | Termina a construção ALT | — | — |

Tabela A.1: Primitivas Oferecidas pelo Núcleo