UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

JÚLIO CESAR GAVILAN

SÍNTESE EM ALTO NÍVEL DE UMA REDE DE INTERCONEXÃO DINÂMICA PARA MULTICOMPUTADOR

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação

Orientador - Prof. José Mazzucco Jr.,Dr.
Co-orientador – Prof. César Albenes Zeferino, M.Sc.

Florianópolis, Fevereiro de 2000

Síntese em Alto Nível de uma Rede de Interconexão Dinâmica para Multicomputador

Júlio Cesar Gavilan

Esta dissertação foi julgada adequada para a obtenção do Título de

Mestre em Ciência da Computação

Área de Concentração: Sistemas de Computação, e aprovada em sua forma final pelo Curso de Pós-Graduação em Ciência da Computação – CPGCC

Prof. Fernando A. Osthuni Gauthier, Dr. – Coordenador

Banca Examinadora:

Prof. José Mazzucco Jr., Dr. – Orientador – INE - UFSC

Prof. Cesar Albenes Zeferino, M.Sc. – Co-Orientador – FACIC

Prof. Luis Fernando Friedrich, Dr. – INE, UFSC

Prof. Thadeu Botteri Corso, Dr. – INE, UFSC

Prof. João Bosco Alves, Dr. – INE, UFSC

SUMÁRIO

A evolução da tecnologia da microeletrônica na última década proporcionou a redução de preços de hardware, o que motivou a popularização dos microcomputadores pessoais, permitindo a construção de máquinas paralelas com baixo custo e de alto desempenho baseadas em microcomputadores interligados por meio de uma rede de interconexão - os multicomputadores.

No Departamento de Ciência da Computação da Universidade Federal de Santa Catarina, está sendo desenvolvido um projeto, denominado Projeto Nó// (lê-se nó paralelo), que tem como objetivo a implementação de um ambiente completo para processamento paralelo, incluindo a construção de um multicomputador baseado numa rede de interconexão dinâmica. O sistema de comunicação do multicomputador Nó// é baseado no protocolo Transputer Link.

Neste trabalho é apresentado um projeto de uma rede de interconexão experimental a ser utilizada nesse sistema de comunicação, especificada em VHDL e validada através de simulação, estando pronta para ser sintetizada em FPGA. No texto são descritos os problemas que motivaram o desenvolvimento dessa rede, a metodologia adotada e a solução implementada.

SUMMARY

The evolution of the technology of the microelectronic in the last decade provided the reduction of hardware prices, what motivated the popularization of the personal microcomputers, allowing the construction of parallel machines with low cost and of high performance based on interlinked microcomputers by means of an interconnection network - the Multicomputer System.

In Department of Computer Science of Federal University of Santa Catarina, a project is being developed, called Nó // (Parallel node), that has as goal is to build a Parallel/Distributed Programming Environment, including the construction of a Multicomputer System with a dynamic interconnection network. The system of communication of the Multicomputer System of the Nó // it is based on the protocol Transputer Link.

In this work a project is presented of a experimental interconnection network to be used in that communication system, specified in VHDL and validated through simulation, being ready to be synthesized in FPGA. In the text they are described the problems that motivated the development of that net, the adopted methodology and the implemented solution.

AGRADECIMENTOS

Aos Professores Cláudio Kirner e Goro, da UFSCar e IPEN respectivamente, pelos conselhos sinceros.

A todos os professores, amigos e funcionários do CPGCC que apoiaram, influenciaram e ajudaram na realização deste trabalho.

Ao Sr. Itamar Coelho e aos amigos da INFOTECNICA, onde meus conhecimentos foram consolidados. Em especial ao Daniel Warmling, pela amizade e conhecimentos transmitidos.

Aos coordenadores dos cursos de Engenharia de Computação e Ciência da Computação, Prof. José Leomar Todesco e Fernanda dos Santos Cunha, pela oportunidade oferecida e paciência e aos amigos professores da UNIVALI.

Ao Prof. Rogério Antônio Casagrande da UNESC, pelo apoio recebido.

Um especial agradecimento ao Prof. Luiz Fernando Friedrich – INE/UFSC, por ter iniciado o trabalho de orientação deste trabalho.

Ao meu orientador Prof. José Mazzucco Jr. – INE/UFSC, pela continuação na orientação deste trabalho, paciência e estímulo nas horas mais difíceis.

Um agradecimento muito especial ao Prof. César Albenes Zeferino da FACIC, pela excelente co-orientação, mostrando-me os caminhos corretos e direcionando meu trabalho.

Agradeço aos meus irmãos, pelos seus exemplos que sempre me orientaram e influenciaram.

Agradeço aos meus pais, por todo carinho, dedicação, esforço, proteção e orientação.

A DEUS.

ÍNDICE ANALÍTICO

LISTA DE FIGURA

INTRODUÇÃO

O desenvolvimento da microeletrônica tem permitido o aparecimento de microprocessadores cada vez mais velozes, o que tem resultado em computadores com alto poder de processamento. Porém, os computadores baseados no modelo de von Neumann, sofrem algumas limitações. Todos os acessos a dados e controle são realizados através de um barramento de comunicação entre processador e memória[1], o que causa o aparecimento do "gargalo de von Neumann". Outra característica desse modelo, é a execução das instruções de um programa, em uma ordem seqüencial, o que é conhecido por "processamento seqüencial".

O Princípio da Incerteza[2], postulado por Heisenberg, impõe um limite para a tecnologia de compactação dos circuitos, limitando dessa forma, o crescente aumento da velocidade dos microprocessadores.

Esse motivo tem estimulado a busca por novos modelos de arquiteturas ou então direcionado o estudo para o processamento paralelo[3], que surgiu como uma solução promissora para aumentar o poder de processamento e atender as crescentes necessidades computacionais nas áreas de simulação de fenômenos físicos, projetos de engenharia, processamento de imagens, sistemas de banco de dados e previsão meteorológica.

O conceito de processamento paralelo é tão antigo quanto o surgimento do computador. Porém, devido ao elevado custo inicial do *hardware* e às dificuldades iniciais encontradas na programação paralela, as pesquisas se voltaram exclusivamente para o processamento seqüencial.

---

[1] HWANG, K. , BRIGGS, F. A., **Computer Architecture and Parallel Processing**, Mcgraw-Hill, 1985

[2] RUGGIERO, Carlos Antônio, **Arquitetura não-Convencionais**, curso ministrado no IV Simpósio Brasileiro de Arquiteturas de Computadores - Processamento de Alto Desempenho, São Paulo, 1992

[3] Idem

Novas teorias e técnicas de programação paralela[4], além da crescente queda nos preços do *hardware* têm favorecido a investigação no sentido de tentar desenvolver máquinas paralelas, com um grande número de processadores. Essa solução, porém, não é simples, pois além de existirem várias formas de colocar as unidades processadoras em paralelo, surgem problemas quanto a forma de gerenciamento das máquinas e de como manter a coerência na informação.

Pesquisadores do Departamento de Informática da Universidade Federal de Santa Catarina tem se empenhado no desenvolvimento de uma máquina paralela (projeto Nó// - lê-se nó paralelo), onde será possível processar algoritmos paralelos com um ótimo desempenho.

A concepção do projeto Nó//, determina a utilização de uma rede de interconexão dinâmica para interligar os vários processadores no multicomputador.

Neste trabalho será apresentado a implementação lógica de uma rede de interconexão dinâmica (*crossbar*) a ser utilizada no desenvolvimento do projeto em questão, utilizando-se dispositivos lógicos programáveis.

Este trabalho está dividido em seis partes, sendo que no primeiro capítulo, serão apresentados conceitos básicos envolvendo o Paralelismo, um histórico do desenvolvimento dos computadores eletrônicos, Modelos Computacionais e Taxonomia.

O segundo capítulo apresentará uma visão geral sobre as Redes de Interconexão, topologias das redes de interconexão e considerações em relação ao projeto de uma rede.

O terceiro capítulo apresentará as primícias do projeto Nó//, buscando descrever com detalhes os elementos que foram definidos para compor o seu sistema de comunicação.

No quarto capítulo, será apresentada as metodologias para o desenvolvimento do projeto. Uma visão geral sobre os Dispositivos Lógicos Programáveis, mostrando o ambiente de desenvolvimento de projetos de *hardware* e linguagem de descrição de *hardware* e modelagem em máquina de estado.

Será apresentado no quinto capítulo, o detalhamento do projeto lógico de um crossbar com as mesmas características funcionais do crossbar definido para ser utilizado na implementação da máquina paralela e no sexto capítulo será apresentado os resultados obtidos a partir de simulação sobre o projeto lógico.

Na conclusão, serão realizados considerações sobre o projeto do crossbar e o multicomputador e perspectivas de trabalhos futuros

---

[4] Idem

CAPÍTULO 1 - PARALELISMO

INTRODUÇÃO

O motivo fundamental para o estudo do paralelismo é o desenvolvimento de máquinas que atinjam um alto desempenho[5], utilizando, para isso, vários processadores trabalhando concorrentemente.

O uso de vários processadores, como os utilizados em máquinas seqüenciais, executando um algoritmo simultaneamente, possibilita uma solução mais veloz de um problema do que somente com um processador resolvendo o mesmo algoritmo.

Mesmo nos supercomputadores atuais, vários algoritmos são lentos, como por exemplo, os algoritmos de resolução do problema da previsão meteorológica, bancos de dados, sistemas de arquivos, simulação de fenômenos de aerodinâmica.

Uma outra motivação para o estudo do paralelismo é aumentar a produtividade dos programadores no desenvolvimento de algoritmos paralelos.

Esse capítulo apresenta conceitos e terminologias que envolvem o paralelismo, uma visão geral das Arquiteturas de Computadores e Modelos Computacionais existentes e um histórico do desenvolvimento dos Computadores Eletrônicos.

1.1 - CONCEITOS

Definição

Segundo HWANG e BRIGGS[6]:

---

[5] RUGGIERO, Carlos Antônio, **Arquitetura não-Convencionais**, curso ministrado no IV Simpósio Brasileiro de Arquiteturas de Computadores - Processamento de Alto Desempenho, São Paulo, 1992

> "Processamento paralelo é a uma forma eficiente de processamento da informação que enfatiza a exploração dos eventos concorrentes no processo de computação. Concorrência implica em paralelismo, simultaneidade e pipeline. Eventos paralelos podem ocorrer em vários dispositivos durante o mesmo intervalo de tempo; eventos simultâneos podem ocorrer no mesmo instante de tempo; e pipeline podem ocorrer em tempos sobrepostos. Esses eventos concorrentes são alcançados em um sistema de computação em vários níveis de processamento. Processamento paralelo demanda concorrência na execução de vários programas no computador. É um contraste em relação ao processamento seqüencial. Através de atividades concorrentes é possível melhorar a relação custo-desempenho dos sistemas computacionais."

Granularidade

A granularidade de um computador paralelo corresponde ao tamanho das unidades de trabalho ou, então, à quantidade de instruções de um segmento de programa submetida aos processadores. Essa granularidade pode ser fina, grossa ou média, e a dimensão do grão varia de acordo com o tamanho das unidades de trabalho.

Se está previsto para uma arquitetura paralela alocar grandes processos a um pequeno número de processadores, dizemos que a arquitetura tem granularidade grossa. São exemplos desse tipo de definição, os multiprocessadores ditos convencionais.

Em contrapartida, se uma arquitetura prevê pequenas quantidades de unidades de trabalho sendo alocadas para um grande número de processadores, dizemos que a arquitetura tem uma granularidade fina. São exemplos dessa arquitetura, as máquinas de fluxo de dados. Potencialmente, a granularidade fina implicaria em um processamento mais veloz devido ao maior paralelismo, porém isso pode não ocorrer devido a latência (veja definição do termo a seguir) na comunicação entre os processadores, o que compromete o desempenho.

Latência[7]

É a medida de tempo de sobrecarga (*overhead*) de comunicação que ocorre entre subsistemas de uma máquina. Por exemplo, latência de memória é o tempo necessário para o processador acessar a memória e latência de sincronização é o tempo necessário para dois processos sincronizarem suas operações um com o outro.

---

[6] HWANG, K. , BRIGGS, F. A., **Computer Architecture and Parallel Processing**, Mcgraw-Hill, 1985. pg. 6-7.
[7] DECEGAMA, Angel L., **Technology of Parallel Processing: Parallel Processing Architectures and VLSI *Hardware***, Prentice Hall, 1989. Pg.12.

A latência impõe um fator limitante na dimensão da escalabilidade (definição do termo a seguir) de um sistema. Por exemplo, a latência de memória aumenta com o aumento da capacidade de memória, limitando o contínuo aumento no tamanho da mesma.

Exploração do Paralelismo

Um ponto importante a ser discutido são as possíveis formas que podemos explorar o paralelismo. Podemos dividir em quatro níveis:

- Programa ou job
- Subrotina ou procedimento
- Instrução
- Intra-instrução

A nível de programa ou job, a exploração do paralelismo é uma árdua tarefa desempenhada pelo programador, que deve desenvolver algoritmos processáveis paralelamente. Nesse nível é muito grande a dependência do *software* na alocação eficiente dos recursos de *hardware*. Nesse caso, "um grão" é composto de milhares de instruções (granularidade grossa). Na prática, isso acontece nos supercomputadores com pequeno número de poderosos processadores.

Já a nível de intra-instrução, podemos implementar o paralelismo diretamente no *hardware*, explorando, dessa forma, um alto grau de paralelismo e eliminando do programador a tarefa de paralelizar o algoritmo. Isso acontece em máquinas que alocam instruções a processadores, como as de fluxo de dados, que são consideradas de granularidade bastante fina.

Nos níveis intermediários (subrotina ou procedimento e instrução), há um balanço entre a dependência do *software* e *hardware* na exploração do paralelismo.

Máquinas Paralelas em Relação à Aplicação

Quanto aos tipos de programas que cada uma das máquinas paralelas pode executar, podemos definir as máquinas paralelas como:

- de propósito geral: máquinas que utilizam grande variedade de programas em vários tipos de aplicações distintas;
- dedicadas: podem ser utilizadas somente em uma aplicação ou em um conjunto restrito de aplicações.

Máquinas dedicadas normalmente apresentam uma relação custo-desempenho melhor que as máquinas de propósito geral. Porém é muito caro construir máquinas dedicadas para cada tipo de aplicação. Dessa forma, a tendência é desenvolver máquinas de propósito geral.

Distribuição de Trabalho

Esse termo está relacionado com a distribuição das tarefas igualmente entre os vários processadores. É um dos itens mais importantes e pesquisados no estudo de paralelismo entre a máquinas paralelas.

É simples verificar que a distribuição do trabalho é mais eficiente em máquinas com granularidade fina do que em máquinas de granularidade grossa, pelo fato de existir mais trabalho a ser distribuído.

Localidade

Podemos distinguir dois tipos de localidade: localidade espacial, que está relacionada com o fato de que, em determinada arquitetura, instruções e processos relacionados ente si são executados em processadores próximos ou não, e localidade temporal, onde instruções e processos são executados em tempos próximos.

Em arquiteturas que exploram a localidade espacial e temporal encontramos uma diminuição na comunicação entre os processadores, o que aumenta o desempenho da máquina.

Comunicação[8]

Uma análise importante a ser realizada em arquiteturas paralelas é a quantidade de comunicação entre os processadores que ela impõe. A taxa de comunicação numa unidade de tempo é chamada de largura de banda (*bandwidth*).

Em máquinas de granularidade fina, têm-se uma maior quantidade de comunicação entre os processadores do que em máquinas de granularidade grossa. Igualmente, em arquiteturas onde é explorado o paralelismo a nível de instrução, existe uma maior quantidade de comunicação entre os processadores do que em arquiteturas que exploram o paralelismo a nível de programa.

Escalabilidade

A escalabilidade de uma máquina paralela é a capacidade de efetivamente utilizar um número crescente de unidades processadoras. Se não existe degradação no desempenho da máquina, podemos dizer que é escalável.

Sincronismo

Em relação ao sincronismo, podemos classificar os sistemas paralelos como síncronos e assíncronos. Os sistemas síncronos mantém uma relação fixa de velocidade de processamento dos diversos componentes. Os sistemas assíncronos permitem que cada componente tenha sua própria velocidade, devendo ter sua execução paralisada no caso de necessitar de informações ainda não disponíveis de algum outro componente.

"*Speedup*"

"Speedup" é o aumento de velocidade observado entre um computador paralelo com **P** processadores e outro com somente um processador. Pode ser definido como:

$$Sp = T1/Tp,$$

[8] RUGGIERO, Carlos Antônio, **Arquitetura não-Convencionais**, curso ministrado no IV Simpósio Brasileiro de Arquiteturas de Computadores - Processamento de Alto Desempenho, São Paulo, 1992

onde T1 é o tempo de execução de um programa em um único processador e Tp é o tempo de execução do mesmo programa num computador com **P** processadores.

Se o programa que foi executado apresenta taxas elevadas de paralelismo e não há latência de comunicação entre os processadores, então Tp tende a T1/p. Isso faz com que o "Speedup" tenda a p, que é o valor máximo. Essa é a situação ideal. Porém, na prática, sempre ocorre latência na comunicação, impedindo que o "Speedup" tenha valor máximo.

1.2 - PROBLEMAS DA COMPUTAÇÃO PARALELA

O desenvolvimento efetivo de máquinas paralelas têm esbarrado em problemas que ainda não foram exaustivamente explorados e solucionados. Aqui, é apresentado e discutido os mais importantes.

Desempenho de um Computador Paralelo

A suposição de que um computador paralelo com **P** processadores iguais trabalhando concorrentemente, realizará uma determinada tarefa **P** vezes mais rapidamente que um computador com somente um processador não é verdadeira, pois o *overhead* de comunicação e o conflito no acesso à memória também aumenta, o que diminui a desempenho. Portanto, um problema inicial é a definição do número ideal de processadores a serem utilizados para atingir-se uma boa relação custo-desempenho. Outro problema a ser resolvido é a ineficiência dos algoritmos paralelos em explorar a natureza concorrente de alguns problemas.

Comunicação

Em qualquer arquitetura paralela encontraremos o problema da comunicação entre os processadores. É necessário definir-se que tipos de informações serão trocadas entre os mesmos, quais os níveis de paralelismo explorado e que tipo de comunicação deve ser adotado, assíncrona ou síncrona.

Natureza das Aplicações

Não é claro que tipos de aplicações devem ser executadas em ambientes paralelos. Assim, deve-se resolver os seguintes problemas:

- que algoritmos devem ser utilizados em arquiteturas paralelas?
- qual a importância relativa de máquinas de propósito geral e de máquinas dedicadas?
- como o programador verá a máquina? Quais as facilidades que ele encontrará na implementação de determinada aplicação (compiladores, editores de textos)

Confiabilidade

Em computadores paralelos com várias unidades funcionais (de processamento) é possível que algumas dessas unidades falhem. Seria interessante que a máquina não parasse completamente como ocorre com máquinas seqüenciais com somente um processador. De fato, a máquina deveria ser capaz de continuar executando uma aplicação (obviamente, com certo prejuízo no desempenho) mesmo que fosse detectado que algumas unidades funcionais não estivessem funcionando corretamente.

Medição do Paralelismo

Não existe forma de se avaliar o paralelismo existente nos vários estágios da preparação de um *software* (fase de implementação de um código fonte, compilação, relação com o Sistema Operacional, produção do código de máquina), sendo que um algoritmo paralelo pode acabar se transformando em um código seqüencial.

1.3 - DESENVOLVIMENTO DA TECNOLOGIA DE *HARDWARE*

Um dos fenômenos mais surpreendentes é que praticamente todos os computadores construídos desde o final da década de quarenta obedecem ao mesmo princípio base. Trata-se de máquinas inteiramente automáticas, que dispõem de uma memória ampliada e de uma unidade de comando interno, que efetuam operações lógicas de cálculo e de processamento da informação graças a algoritmos armazenados em sua memória[9].

Desde os tempos mais remotos, o homem reconheceu suas limitações em relação a cálculos mentais. O evidente sentimento de frustração no tempo consumido efetuando longos, repetitivos e fatigantes cálculos, levou-o a desenvolver dispositivos artificias que o auxiliassem no exercício de tal atividade.

---

[9] Breton, Philippe, **Histoire de L'informatique**, pg.. 89

A criação desses dispositivos permitiu que pessoas que não tivessem domínio na arte de calcular, pudessem, pelo seu uso, efetuar operações aritméticas[10]. Assim, a utilização do Ábaco na China, passando pelas Réguas de Cálculo e os vários computadores eletromecânicos desenvolvidos a partir dos trabalhos de Blaise Pascal, são exemplos de ferramentas de cálculo desenvolvidas pelo homem. Porém, somente com o desenvolvimento da tecnologia dos dispositivos eletrônicos é que foi possível atingir um alto grau de velocidade e confiabilidade nos cálculos que observamos no presente momento.

1.3.1 - O Computador Eletrônico

As etapas do desenvolvimento dos atuais computadores eletrônicos foram descritas em termos de "geração", onde o tipo de tecnologia eletrônica utilizada na construção do computador é o parâmetro classificatório.

Primeira geração (1945-1954): caracterizada pela construção de computadores que utilizavam a tecnologia dos tubos de vácuo e memórias de relês, sendo que os dispositivos eram interconectados por fios isolados. Computadores desenvolvidos nessa época caracterizavam-se pelo grande volume que ocupavam, como é o caso do ENIAC, o primeiro computador eletrônico plenamente operacional. Outros representantes importante dessa geração são o EDVAC e o IBM 701.

Segunda geração (1955-1964): o grande salto no campo de componentes, ocorreu quando as válvulas foram substituídas pelos transistores e pelos circuitos impressos. Com isso, de uma velocidade de 5 mil operações por segundo, passou-se para uma velocidade de 200 mil. Isso permitiu a realização de máquinas mais potentes e menos volumosas. Exemplos dessa geração, são o IBM 7090 e o Univac LARC.

Terceira geração (1965-1974): iniciaram a utilização da tecnologia dos circuitos integrados de estado sólido, que são dispositivos de silício, onde foi possível combinar vários componentes eletrônicos para as unidades de lógica e memória. Os circuitos integrados tinham inicialmente uma pequena e média escala de integração (SSI e MSI). Os principais representantes desta geração são o IBM/360-370, Texas Instruments ASC (Advanced Scientific Computer) e o PDP-8 da Digital Equipament

[10] Idem, pg.: 63-64

Quarta geração (1975-1991): os computadores dessa geração caracterizam-se pela utilização de circuitos integrados com larga escala e muita larga escala de integração dos seus componentes (LSI e VLSI). Os circuitos integrados LSI continham até 50 mil transistores cravados em uma peça de silício de 1 cm^2 e os VLSI até 100 mil. Como representantes desta geração podemos citar o VAX 9000, o IBM 3090 e o Cray X-MP.

Quinta geração (1992-presente): os atuais sistemas computacionais se caracterizam pela alta densidade e alta velocidade dos seus processadores e memórias resultante da utilização da tecnologia proporcionada pelos circuitos VLSI. Sistemas computacionais dessa geração são o Fujitsu VPP500, o Intel Paragon e o Cray/MPP.

1.3.2 - Arquitetura Seqüencial dos Computadores

Genericamente, o termo arquitetura é utilizado para referenciar tanto os aspectos arquiteturais como os aspectos organizacionais. Os primeiros referem-se principalmente ao conjunto de instruções, enquanto os últimos dizem respeito a como esse conjunto de instruções é implementado. Um mesmo conjunto de instruções pode ser implementado sob diferentes organizações, como é o caso da família x86, onde, a cada geração, uma nova organização é utilizada para incrementar o desempenho de uma mesma arquitetura. Neste texto, o termo arquitetura é utilizado em seu sentido mais amplo, englobando tanto os aspectos arquiteturais como os organizacionais[11]

Ao longo dos anos, os projetistas procuraram aperfeiçoar esses diferentes elementos bem como propor novas formas de organização interna mais racionais, porém predomina a contribuição dada por von Neumann.[12]

Principalmente os computadores da primeira e da segunda geração eram baseados no que é conhecido como arquitetura de von Neumann, onde quatro elementos aparecem como básicos e fundamentais em um computador: um dispositivo de memória, que armazena os dados e as instruções, uma única unidade de lógica e aritmética, que processa a informação, uma única unidade de controle, que organiza o funcionamento interno da máquina e os diferentes elementos de entrada e saída de dados. As informações são tratadas

---

[11] STALLINGS, William. **Computer Organization and Architecture:** principles of structures and function. 3 ed. New York: Macmillan, 1993. pg. 3

[12] Breton, Philippe. **Histoire de L'informatique**, pg.. 187

uma após as outras na unidade de lógica. A unidade de lógica e aritmética e a unidade de controle compõe a unidade central de processamento (CPU). Essa estrutura já tinha sido proposta por Babbage no século passado. Na figura 1.1 ilustramos o modelo de von Neumann.

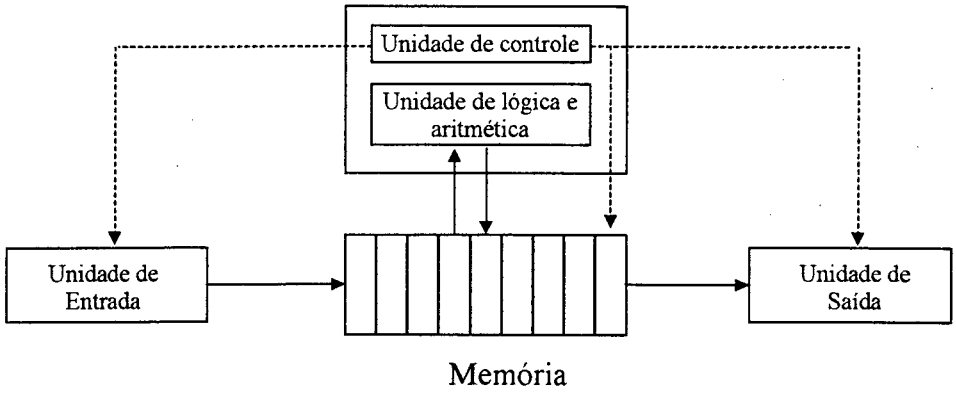


Figura 1.1 - Modelo de von Neumann. Os traços cheios indicam o fluxo de dados e os tracejados indicam o fluxo de controle

1.3.3 - Arquitetura Paralela

O conceito de diferentes partes de um computador operar em paralelo já tinha sido proposto por Babbage no século passado[13]. Porém, o desenvolvimento da tecnologia de computadores paralelos não se desenvolveu em função das dificuldades encontradas por projetistas e programadores. Os projetistas esbarraram no alto custo do *hardware* e os programadores nas dificuldades em desenvolver programas paralelos.

Com a crescente demanda por maiores velocidades de processamento, fez-se necessário o desenvolvimento de máquinas que utilizam outros conceitos diferentes do tradicional modelo de von Neumann, onde a exploração do paralelismo é a chave para o aumento na desempenho da máquina.

A crescente queda dos custos do *hardware* também é um fator que tem estimulado o desenvolvimento de novas tecnologias.

Muitos sistemas de uso geral compostos de múltiplos processadores, hoje disponíveis, podem ser classificados em três grandes grupos em função do grau de integração dos seus componentes: multiprocessadores, multicomputadores e redes locais.

[13] PERROT, Ronald H., **Parallel Programing**, pg. 6

Esses grupos se distinguem pelo grau e velocidade das interações possíveis entre seus componentes. Os multiprocessadores (ou máquinas paralelas com memória compartilhada) são computadores individuais com um número reduzido de processadores que acessam memórias compartilhadas. Os multicomputadores (ou máquina paralela com memória distribuída) são compostos de nós largamente autônomos cada um dos quais dispondo de um processador, de uma memória privativa e de canais de comunicação ligados através de redes de interconexão compostas por múltiplos canais de comunicação bipontuais. As redes locais são compostas de computadores independentes completos interconectados através de redes concebidas como canais de comunicação compartilhados.

Esses três grupos podem formar um sistema integrado no qual os nós de uma rede local sejam representados por multiprocessadores e multicomputadores além de estações de trabalho individuais.

1.3.3.1 - *Pipeling*

Pode-se definir o termo *pipeline* como a capacidade de sobrepor um conjunto de estágios, explorando, dessa forma, o paralelismo temporal.

Por exemplo, de modo geral, os estágios de uma instrução em uma Arquitetura de von Neumann envolvem: busca da instrução na memória (IF), decodificação da instrução (ID), busca dos operandos na memória (OF) (se necessário) e execução da instrução (EX). Em um computador sem *pipeline*, uma nova instrução é realizada somente após a finalização desses quatro estágios da instrução anterior. Em um computador pipeline, cada estágio de uma instrução é sobreposto aos estágios de outras instruções, fazendo com que instruções sejam executadas em conjunto em um número menor de ciclos de máquina. No diagrama da figura 1.2. são mostradas as diferenças entre a execução de um conjunto de instruções em um esquema *pipeline* e em um esquema sem *pipeline*.

Pela figura, é observado, que enquanto pelo esquema b) são realizadas apenas duas instruções, no esquema apresentado em a) é realizada uma sobreposição no ciclo de execução de instruções, o que permite a execução de cinco instruções ao mesmo tempo.

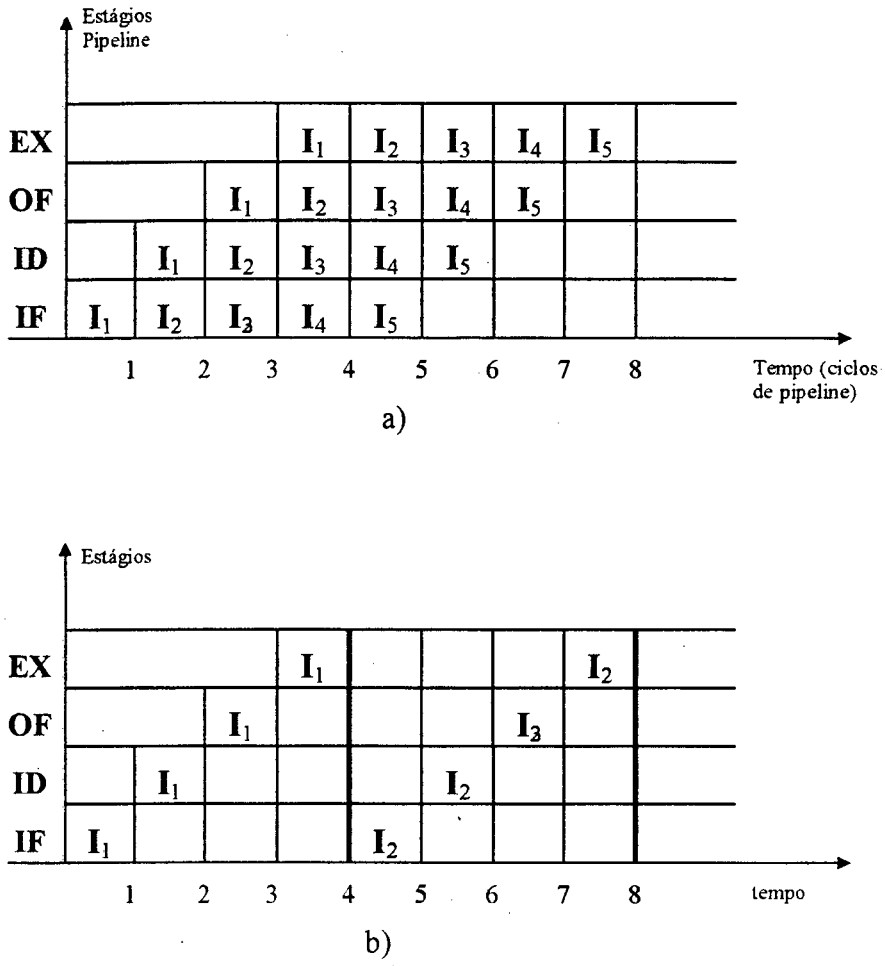


Figura 1.2 - Diagrama da execução de uma instrução: a) utilizando o esquema *pipeline* e b) sem *pipeline*;

1.3.3.2 - Computadores Vetoriais

São caracterizados por conseguirem a elevação da velocidade de processamento com o uso de "*pipeline*". O supercomputador CRAY-1, é um exemplo de máquina que tem esta característica em sua arquitetura. A organização básica desses processadores consiste de: um processador de instrução, que busca e decodifica instruções utilizando um esquema *pipeline*, uma unidade de processamento vetorial, para executar fluxos de dados, e um processador escalar, para executar a parcela não vetorizável do programa.

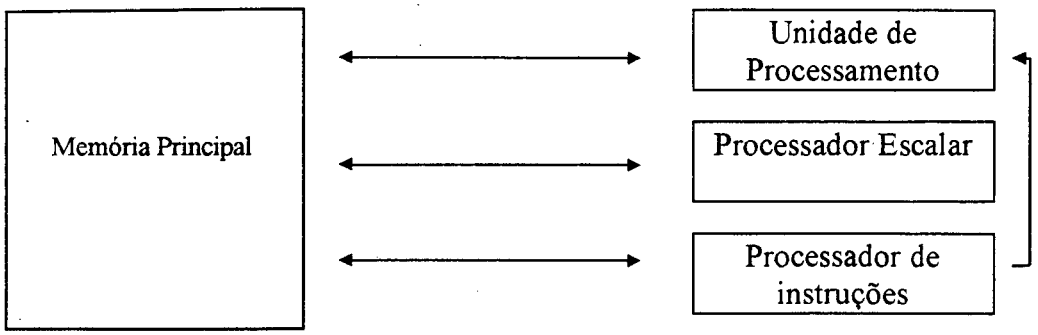


Figura 1.3 - Esquema de um processador vetorial

1.3.3.3 - Processadores Matriciais

Os processadores matriciais possuem um paralelismo espacial do tipo síncrono. Esta arquitetura é caracterizada por vários elementos de processamento (EP), supervisionadas por uma unidade de controle, de onde emanam as instruções que operam sincronamente sobre os diferentes dados. Para a comunicação entre os elementos de processamento, do ponto de vista da transmissão de dados e resultados, os processadores matriciais possuem uma estruturas de interconexão entre os elementos de processamento permitindo o roteamento dos dados. Normalmente, as instruções escalares são executadas diretamente na unidade de controle, enquanto as instruções vetoriais são executadas na matriz de elementos de processamento.

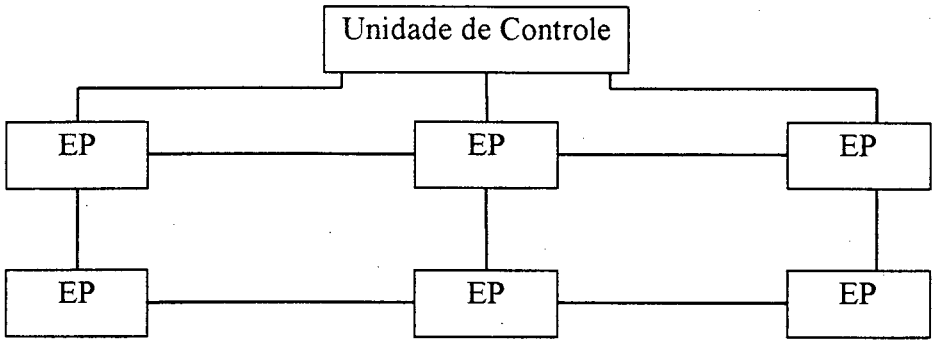


Figura 1.4 - Esquema de um processador matricial

1.3.3.4 - Multiprocessadores

São máquinas com várias unidades processamento que compartilham uma única unidade de memória, onde estão armazenados os dados e as instruções a serem executadas. Devido ao compartilhamento da memória, os multiprocessadores são limitados na quantidade de processadores que podem ser conectados ao sistema (poucas dezenas). A escalabilidade dos multiprocessadores é pequena, pois a inserção de um novo elemento processador pode acarretar em uma saturação da utilização da memória. O compartilhamento da memória é realizado através de uma rede de interconexão

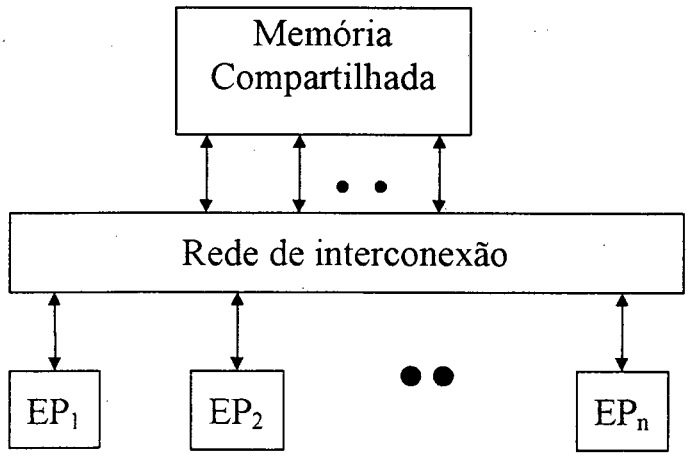


Figura 1.5 - Esquema de uma multiprocessador. Legenda: EP, elemento processador.

1.3.3.5 - Multicomputadores

São caracterizados pela utilização de várias unidades processadoras, cada uma com sua própria memória local, interligadas através de um sistema de interconexão.

Em um multicomputador com N elementos interligados por um sistema de interconexão, cada elemento dessa rede é um computador seqüencial tradicional (von Neumann), isto é, processador mais memória. Esses elementos são designados por nós.

A conjunção das seguintes características permite distinguir os multicomputadores das outras máquinas paralelas[14]:

[14] REED, D. A. & FUJIMOTO, R. M., **Multicomputer Network: Message-Based Parallel Processing**, MIT Press, 1978

- grande número de elementos processadores homogêneos - o baixo custo dos microprocessadores e a facilidade de montagem permitem a construção de máquinas com centenas de processadores.
- grande número de canais de comunicação bipontuais - as redes de interconexão se dispõem em topologias regulares com muitos canais exclusivos conectandos aos pares os nós dessas máquinas.
- interação baseada em troca de mensagens - os processos de um programa paralelo que executam em nós diferentes podem interagir exclusivamente através de troca de mensagens, sendo que normalmente, as mensagens não possuem tamanho fixo.
- alta velocidade de comunicação - as redes de interconexão dos multicomputadores constituem meios confiáveis de comunicação entre nós com velocidades de uma ordem de grandeza superior às das redes locais.
- granularidade média de processamento - os multicomputadores encorajam a exploração do paralelismo real pela decomposição de programas em vários processos cooperantes. Entretanto, uma granularidade muito fina pode se tornar ineficiente pelo predomínio das comunicações.

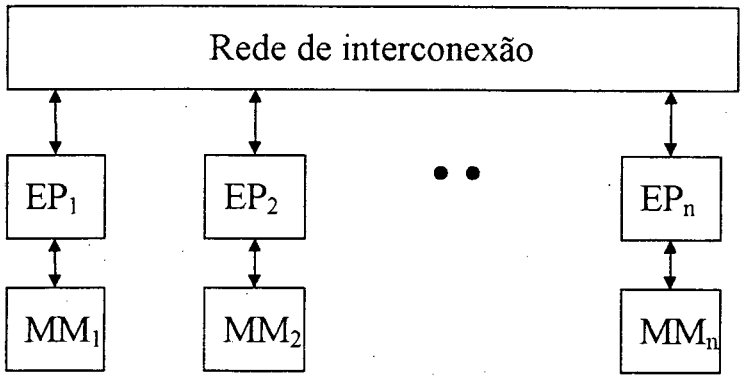


Figura 1.6 - Figura esquemática de um multicomputador. Legenda: EP, elemento processador; MM, módulo de memória privativa.

1.3.3.6 - Arquitetura DATAFLOW

Os computadores convencionais são baseados no mecanismo de fluxo de controle na qual a ordem de execução do programa determina a ordem de execução das instruções e é explicitado pelo programador. Dessa forma, as instruções sempre são realizadas de forma seqüencial. Além disso, as arquiteturas baseadas no modelo de von Neumann sofrem algumas limitações, pois todo o acesso a dados e instruções é realizado através de um único barramento de comunicação entre o processador e a memória.

A característica fundamental do modelo de arquitetura Dataflow é a execução do paradigma no qual instruções são avaliadas para a execução tão logo todos os operandos necessários tornem-se disponíveis. Portanto, a seqüência de execução de uma instrução é baseada na dependência dos dados, permitindo desta forma, a exploração de um alto grau de paralelismo a nível de instrução.[15] Os dados gerados, são multiplicados em várias cópias e passados a todas as instruções que necessitem desse dado. Dessa forma, os dados consumidos por alguma instrução, podem ser usados por outras instruções.

Essa arquitetura não requer nenhuma memória compartilhada, mas sim um mecanismo especial para detectar os dados avaliados e enviá-los para as instruções necessárias, e um mecanismo para sincronizar a execução de instruções assíncronas.

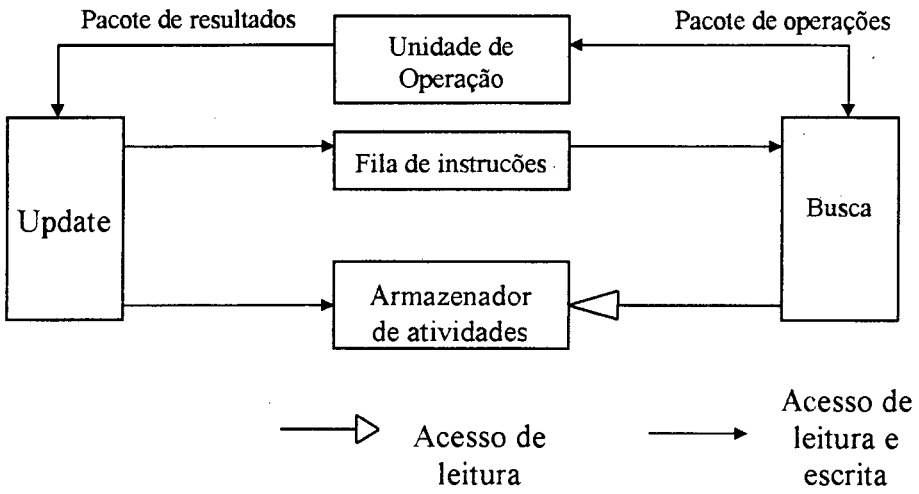


Figura 1.7 - Mecanismo básico de uma arquitetura Dataflow

1.4 - MODELOS COMPUTACIONAIS

Esforços no sentido de classificar as várias propostas de arquiteturas de computadores têm auxiliado atuais projetistas de *hardware* no desenvolvimento dos seus projetos. Pode-se separar cada proposta de arquitetura em um conjunto de regras que são seguidas na execução do projeto.

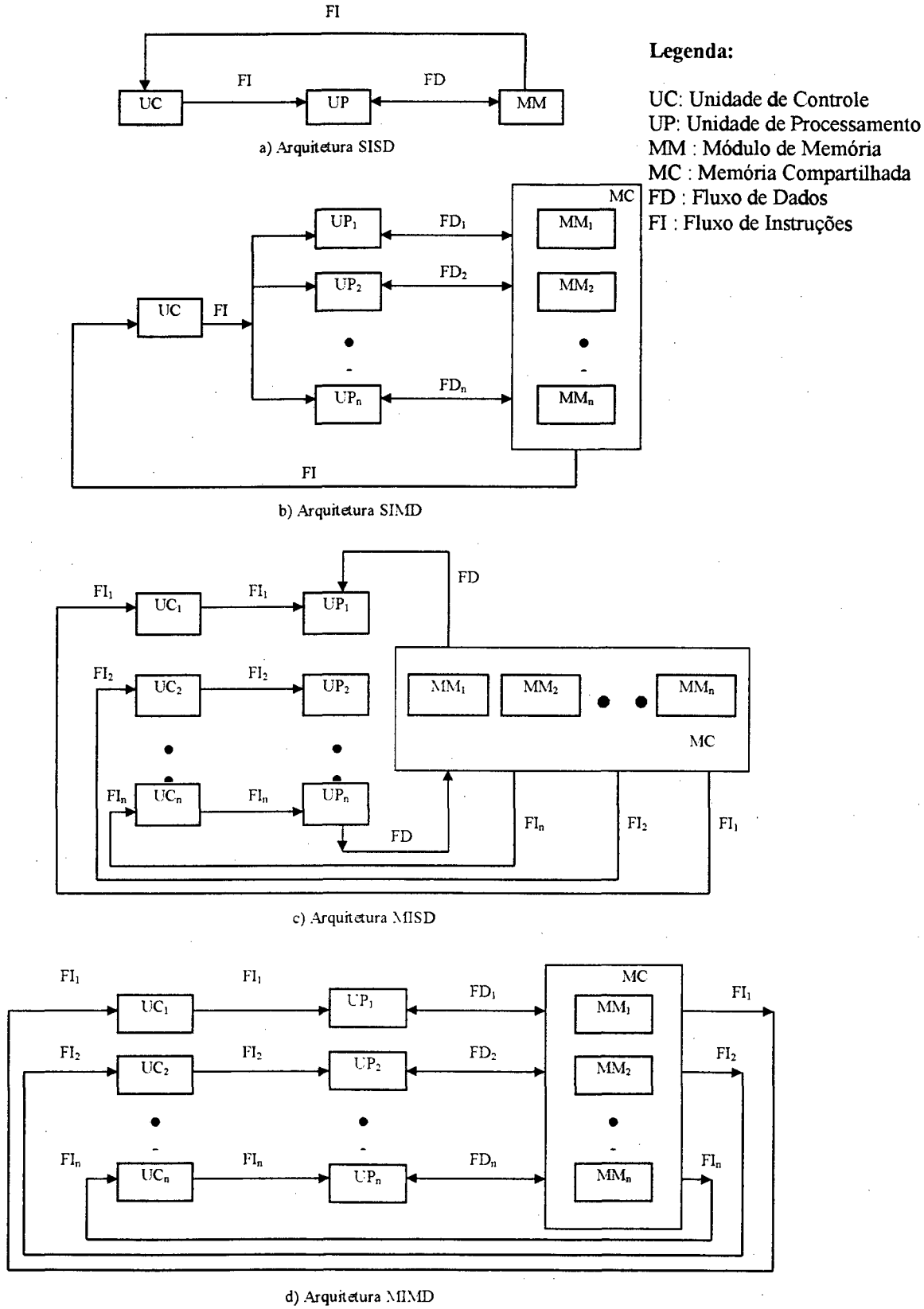
---

[15] DUNCAN, Ralph. **A Survey on Parallel Computer Architectures**, in: IEEE Computer, Feb. 1990, pg. 12

1.4.1 - Classificação de Flynn

A classificação mais aceita é a proposta por Flynn em 1966 baseada nos fluxos de dados e instruções, que considerou o processo computacional como uma execução de uma seqüência de instruções sobre um conjunto de dados. Dessa forma, foi possível separar as arquiteturas em 4 categorias de máquinas conforme a multiplicidade do fluxo de dados e instruções:

- SISD (Single Instruction Single Data): tratam-se de máquinas baseadas no modelo de von Neumann, onde somente uma única instrução é executada seqüencialmente.
- SIMD (Single Instruction Multiple Data): corresponde aos processadores matriciais paralelos. Vários elementos processadores sendo supervisionados por uma mesma unidade de controle, isto é, todos os dados recebem a mesma instrução para operarem sobre diferentes dados.
- MISD (Multiple Instruction Single Data): corresponderia a uma máquina que tivesse vários elementos processadores recebendo o mesmo fluxo de dados. O resultado de um processador seria a entrada do próximo. Essa estrutura não existe na prática, apesar de alguns autores classificam as máquinas pipeline como representantes dessa categoria.
- MIMD (Multiple Instruction Multiple Data): representantes dessa categoria seriam os multiprocessadores e multicomputadores. Cada unidade de processamento possui sua unidade de controle executando instruções sobre um conjunto de dados, e o que mantém a interação entre os diversos processadores é a memória. Pesquisas envolvendo o desenvolvimento de arquiteturas paralelas se concentram nessa categoria.



FONTE: Hang,K.& Briggs, F.A., "Computer Architecture and Parallel Processing", pg.33

Figura 1.8 - Classificação dos modelos computacionais propostos por Flynn.

1.4.2 - Classificação de Sistemas de Processamento Paralelo

Há três propostas para computadores paralelos: von Neumann, Dataflow e máquinas de redução. Uma outra proposta é um híbrido da proposta Dataflow e de redução.

A proposta de von Neumann também referenciada como processamento dirigido pelo controle (control-driven), pois a execução do programa é dirigida pela unidade de controle, obedecendo a ordem seqüencial de instruções estabelecida pelo programa, consiste na conexão de dois ou mais processadores do tipo de von Neumann.

A proposta Dataflow (item 1.3.6), também referenciada como processamento dirigido por fluxo de dados (data-driven), é baseada no conceito de execução das instruções tão logo os seus operandos estejam disponíveis, o contrário da proposta de von Neumann que segue uma seqüência ditada pelo ordem das instruções de um programa.

A proposta de Redução, também referenciada como processamento dirigido por demanda (demand-driven), consiste no reconhecimento de expressões que possam ser reduzidas, substituindo-as por seus valores calculados até que aconteça a redução de todo o programa, gerando o resultado final. Um programa é visto como um conjunto de aplicações, e a execução procede por sucessivas reduções da aplicação mais interna, de acordo com a semântica dos seus respectivos operadores, até a realização da aplicação mais externa (ou seja, quando não houverem mais aplicações). Uma instrução é realizada somente quando os seus resultados são necessários para outros e não quando os operandos estão disponíveis, como na máquinas Dataflow.

Na proposta Híbrido, os processadores, ao invés de realizarem instruções que já estão prontas para a execução, como na arquitetura Dataflow, deverão, inicialmente, realizar instruções necessárias se os seus operandos estiverem prontos. Se os operandos ainda não tiverem sido avaliados, o processador deverá requerê-los de outros processadores enquanto trabalha com itens de menor prioridade. Isso evita que os dados circulem pela máquina, o que é uma característica da arquitetura Dataflow.

Na figura 1.8, tem-se uma visão geral das diferentes propostas para arquitetura de processamento paralelo. As pesquisas ainda tem se direcionado em grande parte ao desenvolvimento de computadores baseados na proposta de von Neumann.

Observa-se, pela figura, que essa classificação engloba a taxonomia de Flynn (item 1.4.1), o que é um requerimento colocado por DUNCAN [16], pois os sistemas baseados na

[16] DUNCAN, Ralph. **A Survey on Parallel Computer Architectures**, in: Computer, February, 1990. pg. 6

proposta de von Neumann estão classificados de acordo com os fluxos de instruções e de dados.

Não faz parte do escopo deste trabalho a análise de cada um dos sistemas mostrados na figura 1.8. Para este trabalho, são importantes somente os sistemas MIMD baseados na utilização de redes de interconexão do tipo *crossbar*.

1.4.2.1 - Sistemas MIMD - *Crossbar*

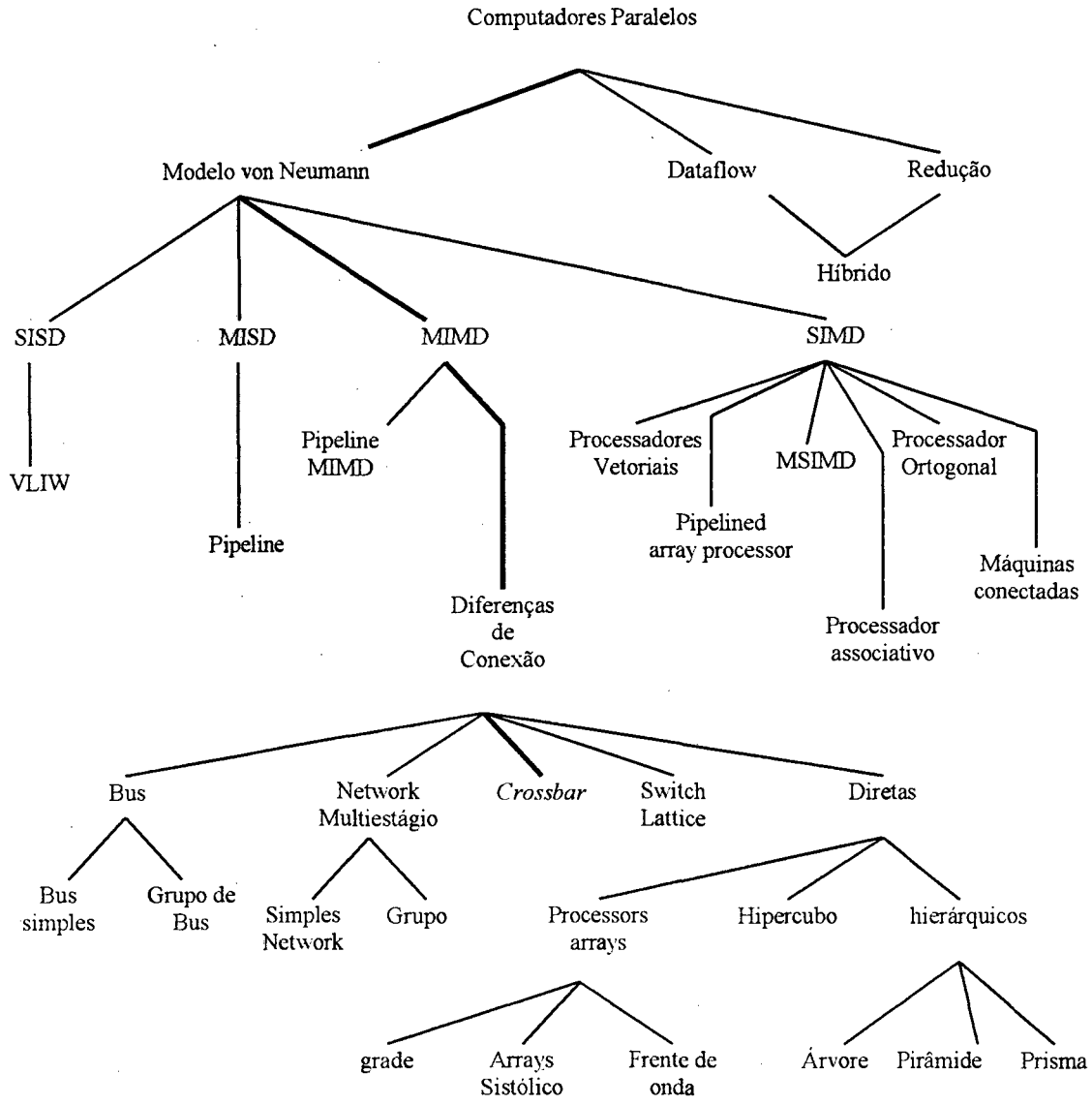
A utilização de uma rede de interconexão do tipo *crossbar* permite a conexão de dois ou mais processadores, módulos de memória e dispositivos de entrada e saída. O *crossbar* proporciona um desempenho melhor do que os sistemas organizados no compartilhamento de barramento. Porém, são mais complexos, maiores (em termos de volume) e mais caros (um estudo sobre as redes de interconexão será realizado no capítulo 2). Basicamente, são dois os sistemas que fazem uso de *crossbar*: os multiprocessadores (item 1.3.3.4) e os multicomputadores (item 1.3.3.5).

1.5 - SUPERCOMPUTADORES

Supercomputadores são definidos como os computadores mais rápidos existentes numa determinada época.

Atualmente, os supercomputadores apresentam um desempenho de centenas a milhares de Mflops (milhões de operações de ponto flutuante por segundo), usam palavras de 64 bits, possuem uma memória principal com dezenas de palavras e custam cerca de 2 a 20 milhões de dólares.

Todos os supercomputadores conseguem um elevado desempenho pelo uso de componentes de alta velocidade e pela execução de múltiplas operações simultaneamente.



FONTE: Decegama, Angel L., "Technology of Parallel Processing : Parallel Processing Architectures and VLSI Hardware", pg. 64.

Figura 1.9 - Classificação das Arquiteturas de Processamento Paralelo. Em negrito, as ramificações que levam aos sistemas que utilizam redes de interconexão do tipo *crossbar*.

CAPÍTULO 2 - REDES DE INTERCONEXÃO

INTRODUÇÃO

Com o crescente desenvolvimento de arquiteturas paralelas, os subsistemas de comunicação, interligando os vários elementos processadores, módulos de memórias e unidades de entrada e saída, tornaram-se uma importante característica no projeto e desenvolvimento de uma arquitetura, pois têm um impacto direto na capacidade, desempenho, dimensão e custo de todo sistema. Duas alternativas de subsistemas de comunicação podem ser utilizadas: os barramentos e as redes de interconexão.

Para sistemas com um pequeno número de elementos conectados, os barramentos tem um desempenho adequado, e proporciona um simples subsistema de comunicação. Porém para um grande número de elementos conectados, observa-se uma sobrecarga na taxa de requisição do barramento. Nesse caso, o uso de uma rede de interconexão, que consiste em múltiplos caminhos entre uma fonte e um destino, elimina esse problema, proporcionando desejável conectividade e desempenho. Porém, o custo e a complexidade de uma grande rede de interconexão, além da dimensão, crescem com o aumento do número de elementos conectados, apresentando uma queda na relação custo-desempenho com o aumento do número de conexão.

Neste capítulo, será apresentada uma discussão dos mecanismos de comunicação em um sistema de processamento paralelo, apresentando as topologias de barramento e redes de interconexão. Apresenta também os aspectos necessários para o projeto, implementação e utilização de uma rede de interconexão do tipo *crossbar*.

2.1 - TOPOLOGIA

O termo topologia de um sistema de interconexão está relacionado com a forma a pela qual as várias unidades funcionais de um sistema estão interligadas umas as outras. As topologias[17] tendem a ser agrupadas em duas categorias: estáticas e dinâmicas. As redes estáticas são formadas por conexões ponto-a-ponto fixas, as quais não mudam durante a execução do programa. As redes dinâmicas são implementadas através de canais chaveados, que podem ser reconfigurados dinamicamente configurando-se as chaves.

2.2 - SISTEMA DE COMUNICAÇÃO

Pode-se dividir os sistemas de comunicação em dois tipos: os barramentos, onde há um meio físico comum a todos os processadores, e as redes bipontuais, que possuem canais que são responsáveis pela conexão entre pares de nós processadores.

2.2.1 - Barramentos

Os barramentos consistem em um sistema de comunicação onde há compartilhamento de um caminho entre mais de um par de elementos que compõe o sistema, para a troca de mensagem. Pode ser implementado por um simples conjunto de fios. Na figura 2.1, é mostrado um esquema de sistema de barramento.

As operações de transferência de dados são controladas por um mecanismo de resolução do conflito, que avalia se o barramento pode ser utilizado. Ou seja, antes de ser enviada uma mensagem pelo barramento, deve ser avaliado se ele já não está sendo utilizado por outro elemento do sistema. Esse mesmo mecanismo também é responsável em avaliar se o destino está apto a receber a mensagem.

Os barramentos são constituídos por um número de diferentes linhas, denominadas de sinais: (a) linha de endereço, especifica o endereço de um processador, módulo de memória ou unidade de entrada e saída; (b) linha de dados é utilizada para a transferência de dados em paralelo (deve ter o tamanho da palavra do computador); (c) linha de controle, por onde passam todos os sinais de controle do tipo estado do barramento, indicação de leitura ou escrita, reconhecimentos e clock, entre outros que podem ser implementados conforme a necessidade do sistema; (d) sinais de interrupção, por onde podem ser

[17] HWANG, Kai. **Advanced Computer Architecture**. 2. Ed. New York: Mcgraw Hill, 1987. pg. 76

transmitidos sinais com prioridade maior dentro do sistema, por exemplo o sinal de reset. Na figura 2.2 são mostradas as várias linhas que pode-se encontrar dentro de um barramento.

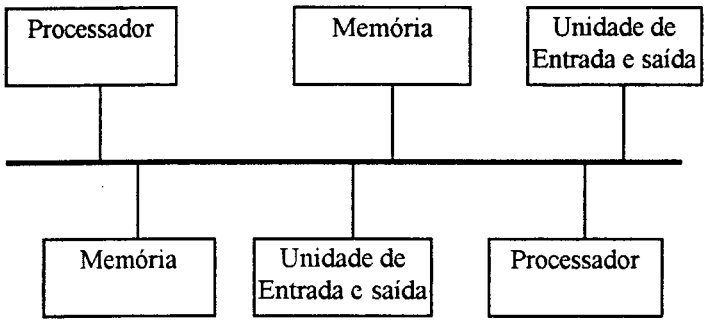


Figura 2.1 - Sistema de Barramento.

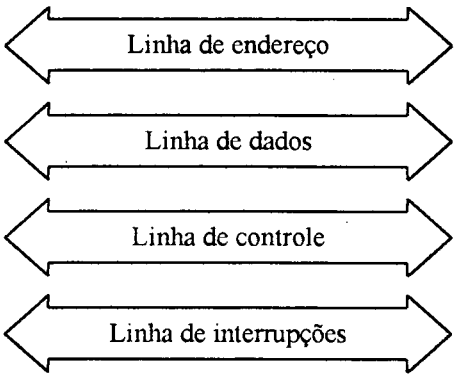
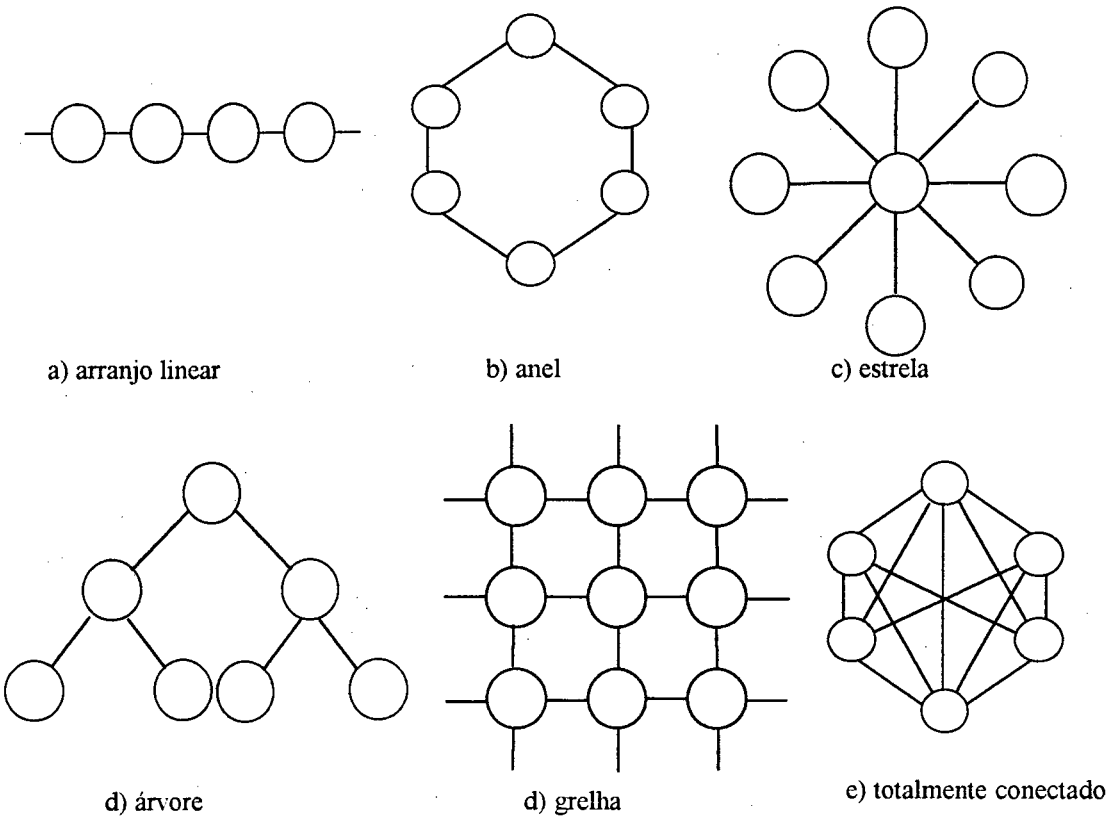


Figura 2.2 - Linhas em um barramento.

As redes bipontuais se dividem por sua vez em estáticas e dinâmicas.

2.2.2 - Redes Bipontuais Estáticas

Se caracterizam por não ser possível alterar a sua configuração. Podem ser classificadas de acordo com a dimensão do seu *layout*, mais especificamente, são unidimensionais, bidimensionais, tridimensionais e hipercubo. Exemplo de redes unidimensionais são as que incluem arranjos lineares como algumas arquiteturas *pipeline*. Exemplos de bidimensionais são as redes do tipo anel, grelha, estrela e árvore. As redes onde todos os nós estão conectados (completamente conectados) e o cubo são exemplos de redes estáticas tridimensionais. Na figura 2.3, são mostradas os exemplos citados.



FONTE: Feng, T., "A Suvey of Interconnection Network", IEEE Computer, pg. 7

Figura 2.3 - Redes bipontuais estáticas

2.2.3 - Redes Bipontuais Dinâmicas

Possuem o poder de alterar sua topologia dinamicamente, conforme a necessidade da aplicação, através de comutadores de conexão. As redes dinâmicas podem ser classificadas em três categorias: *crossbar*, simples-estágio e multi-estágios.

2.2.3.1 - *Crossbar*

Em um *crossbar* existem várias linhas de entrada e saída. No cruzamento de cada linha há um elemento chaveador, que permite a conexão das linha de entrada com a linha de saída de um lado e com as linhas de saída com as de entrada do outro. Os elementos chaveadores podem ser configurados por algum componente do sistema externo. O *crossbar*

proporciona vários caminhos separados de comunicação entre dois componentes do sistema. Pode-se conectar vários componentes ao mesmo tempo. Na figura 2.4, é mostrado o esquema de um *crossbar*. Um *crossbar* pode atingir um alto desempenho, porém, apresenta problemas com a aumento do número de componentes conectados, pois a relação custo-desempenho torna-se ruim, além da tendência da dimensão e da complexidade do *crossbar* crescerem.

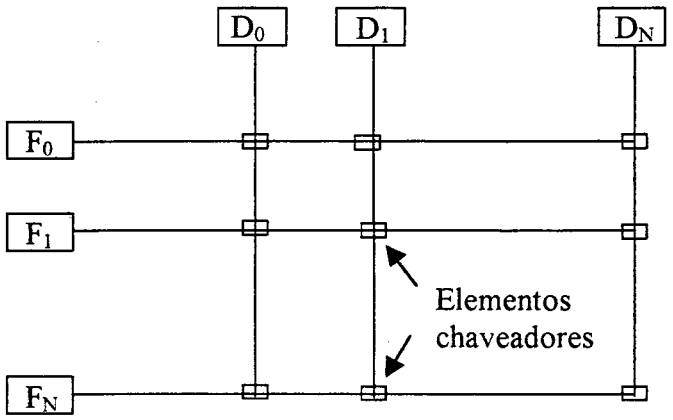


Figura 2.4 – Esquema de um *crossbar* com N fontes e N destinos

2.2.3.2 - Simples-estágio

É composta de um estágio de elementos chaveados em cascata que servem de canais de comunicação.

2.2.3.3 - Multi-estágio

Consiste de mais de um estágio de elementos chaveados e é usualmente capaz de conectar um número arbitrário de entradas com um número arbitrário de saídas. Trata-se de uma combinação de elementos com capacidade de conectar, simultaneamente, duas entradas (ou mais) e duas saídas (ou mais) . É possível construir uma rede dinâmica multi-estágio que permita a conexão entre quaisquer elementos.

2.3 - PROJETO DE UMA REDE DE INTERCONEXÃO

Na escolha de uma arquitetura de uma rede de interconexão, quatro itens devem ser considerados: modo de operação, estratégia de controle, método de chaveamento e topologia da rede.

Modo de operação, do ponto de vista de comunicação, uma rede pode operar de modo síncrono ou assíncrono. Comunicação síncrona é necessária para processamentos nos quais os caminhos de comunicação são estabelecidos sincronamente para cada função de manipulação de dados ou transmissão de instruções e dados. Comunicação assíncrona é necessária para multiprocessamento nos quais conexões são requisitadas dinamicamente. Um sistema também pode ser projetado para trabalhar de modo síncrono e assíncrono.

Estratégia de controle, consiste no número de elementos chaveadores e links de intercomunicação. As conexões são realizadas através de um devido conjunto de controle dos elementos chaveadores. A rede pode ser gerenciada por um controle central denominado "controle centralizado" ou, então, por elementos chaveadores individuais, sendo denominado "controle distribuído".

Metodologia de chaveamento, podem ser de dois tipos: circuito chaveado, que consiste em estabelecer um caminho físico entre fonte e destino; e pacote, onde os dados são empacotados e roteados através da rede de interconexão sem estabelecer um caminho físico. Em geral, circuito chaveado é mais útil para um grande volume de dados transmitidos, e pacote é mais eficiente para pequenos dados. Outra opção e integrar os dois tipos, capacitando a realizar os dois tipos de chaveamento.

Topologia da rede, uma rede pode ser descrita por um grafo nos quais os nós representam os pontos chaveadores e as arestas representam os links de comunicação. A topologia tende a ser regular e pode ser agrupada dentro de duas categorias: dinâmicas e estáticas. Na topologia estática, links entre dois elementos processadores não podem ser reconfiguráveis e na topologia dinâmica os links podem ser reconfiguráveis (ou seja, podem ser alterados).

CAPÍTULO 3 – O PROJETO NÓ//

INTRODUÇÃO

O projeto Nó// (lê-se nó paralelo) prevê o projeto e implementação de um multicomputador que apresenta um conjunto processadores, representados por nós, conectados entre si por intermédio de dois dispositivos distintos: uma rede de interconexão dinâmica do tipo *crossbar* e um barramento de serviço. Esse projeto está sendo desenvolvido por pesquisadores do Departamento de Informática e Estatística (INE) da Universidade Federal de Santa Catarina (UFSC), e tem o objetivo de servir como instrumento de ensino, pesquisa e desenvolvimento de novas tecnologias.

Neste capítulo, serão descritas a arquitetura do projeto Nó// e os componentes mais importantes que já foram definidos para serem utilizados na implementação do projeto.

3.1 - O MULTICOMPUTADOR NÓ//

Segundo Brich Hansen[18], uma máquina de propósito geral deve ter as seguintes características:

1. Arquitetura escalável para milhares de processadores;
2. As unidades processadoras devem ser de propósito geral;
3. computador deve suportar diferentes tipos de estruturas de processos (como *pipeline*, árvore, e outras) de maneira transparente;
4. A criação de processos, comunicação e finalizações devem ser operações de *hardware*;
5. computador paralelo deve distribuir automaticamente os processos computacionais, carregando e roteando as mensagens entre os processadores.

[18] BRINCH HANSEN, Per, **Studies In Computation Science – Parallel Programming Paradigms**, pg. 5

A máquina proposta satisfaz as condições citadas e consiste em um multicomputador, ou seja, uma máquina paralela com memória distribuída, que utiliza uma rede de interconexão dinâmica. Na sua primeira versão, ela é projetada para a interligação de 16 até 32 computadores com CPU x86 (ou *Pentium*). As comunicações são realizadas através de *transputer-links*. Cada nó representa um computador com CPU x86 (ou *Pentium*) e memória local, com total autonomia e canais de conexão com um *crossbar*. Um nó especial (nó de controle) é ligado a todos os outros nós (nós de trabalho) e é responsável por comandar todas as operações de controle. Existem dois tipos de conexão: conexão por demanda, via *crossbar*, que permite a interconexão independente ponto-a-ponto entre pares de nós, e conexão de serviço. A implementação da conexão de serviço pode ser realizada de duas formas distintas, o que levou a concepção de modelos distintos de arquitetura para o multicomputador.

Modelo de Arquitetura Baseada em um Barramento Comum, consiste na ligação do nó de controle com os nós de trabalho através de um barramento compartilhado, denominado barramento de serviço, como mostra a figura 3.1. O nó de controle é o responsável pelo comando do *crossbar* e do barramento de serviço. Através de um processo de *polling*, este nó identifica se um nó de trabalho solicita algum serviço de conexão ou desconexão, e se encarrega de configurar o *crossbar*. Em função do compartilhamento do barramento de serviço, essa arquitetura apresenta uma sobrecarga de comunicação no barramento e um atraso na execução do processo de *polling*. Por essa razão foi concebido um outro modelo de arquitetura onde não há o compartilhamento de um barramento.

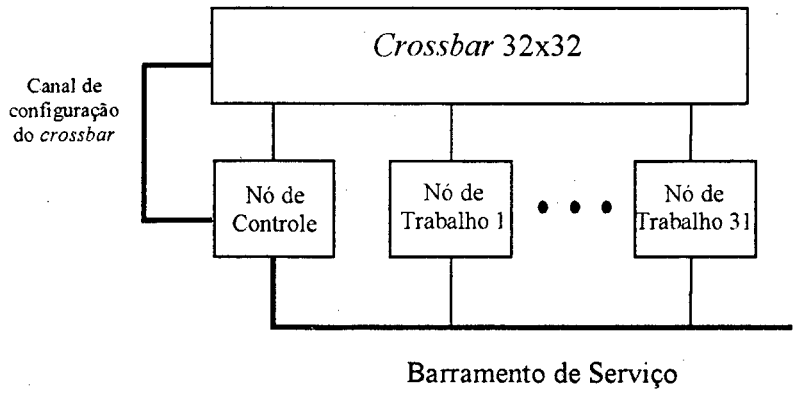


Figura 3.1 – Modelo de Arquitetura baseado no Compartilhamento de Barramento

Modelo de Arquitetura Baseada em um Sistema de Interrupção, consiste na ligação do nó de controle com os nós de trabalho através de linhas de interrupção exclusivas. Cada linha é formada por um par de linhas unidirecionais (figura 3.2) por onde, através de um sinal de interrupção, um nó de trabalho solicita algum serviço de conexão ou desconexão para o nó de controle (INTRtc). O nó de controle identifica a origem do sinal de interrupção e configura o *crossbar* e conecta-se com este o nó de trabalho. Logo a seguir o nó de controle interrompe o nó de trabalho (INTRct) para o recebimento do serviço através do *crossbar*. Após o recebimento do serviço realiza a desconexão com o nó de trabalho e realiza o serviço solicitado. Novamente interrompe o nó de trabalho (INTct), confirmando a realização do serviço solicitado. Nesse modelo, o atendimento a um serviço requisitado por algum nó tem seu tempo reduzido em relação ao modelo apresentado anteriormente, pois depende somente do atendimento da interrupção e do chaveamento do *crossbar*.

A utilização combinada do *crossbar* e do sistema de interrupção permite estabelecer rotas entre os nós de trabalho por onde é realizada a troca de mensagens, o que oferece uma flexibilidade maior em relação às redes de interconexão estática. Isso se deve ao fato de o sistema de interrupção ser utilizado como meio permanentemente confiável para a transmissão de requisitos de conexão e desconexão de canais. O estabelecimento de uma conexão individual através do *crossbar* é uma operação muito rápida que pode ser efetuada a qualquer momento sem afetar outras conexões já existentes.

Este modelo elimina a sobrecarga no barramento de serviço do modelo anterior, porém a complexidade do *hardware* associado ao sistema de interrupção cresce proporcionalmente à quantidade de nós da máquina paralela, o que torna este modelo pouco escalável.

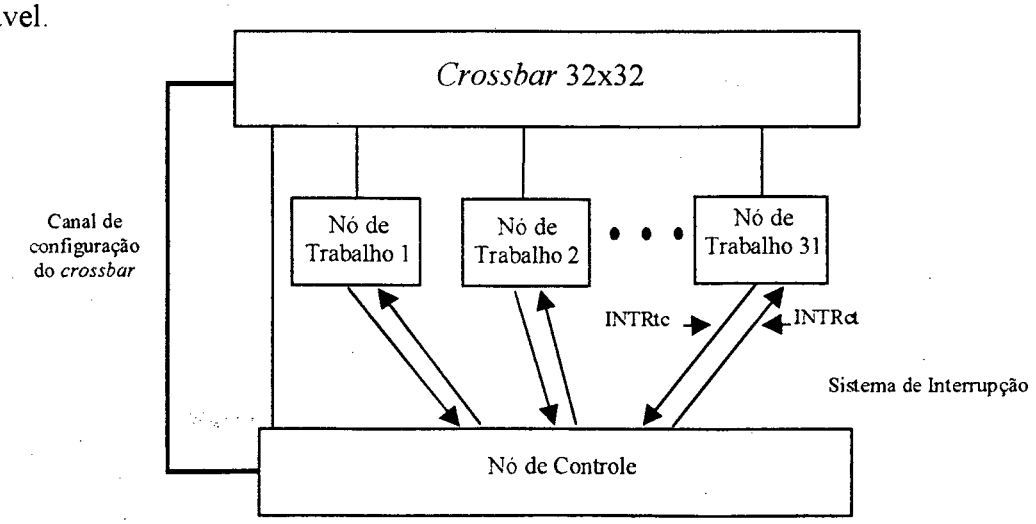


Figura 3.2 – Modelo de Arquitetura baseado no Sistema de Interrupção

Modelo de Arquitetura com Linhas de Interrupção e Barramento Compartilhado, esse modelo reúne as características dos dois modelos já apresentado visando reduzir a complexidade e custo do *hardware* e atingir um bom desempenho.

Consiste na ligação de cada nó de trabalho com o nó de controle por meio de uma linha de interrupção e de um barramento de serviço compartilhado, como mostra a figura 3.3. As linha de interrupção são canais unidirecionais que são utilizadas pelos nós de trabalho para interromper o nó de controle quando houver a necessidade de um serviço.

Quando um nó de trabalho deseja solicitar algum serviço ao nó de controle, gera uma interrupção e envia através da linha de interrupção. Ao receber uma interrupção, o nó de controle inicia um ciclo de *polling* endereçada àquele nó de trabalho e estabelece uma conexão através do barramento de serviço. Com a chegada do comando de *polling*, o nó de trabalho envia sua requisição de serviço ao nó de controle que busca atender o serviço requisitado. Através do mesmo barramento de serviço o nó de controle confirma ao nó de trabalho, o atendimento do serviço requisitado.

O *hardware* desse modelo torna-se mais complexo devido à necessidade de uma interface para as linhas de interrupção, para o barramento de serviço e para a comunicação via *crossbar* em cada nó. Porem, o uso combinado das linhas de interrupção e do barramento de serviço elimina a sobrecarga do primeiro modelo e apresenta uma boa relação custo-desempenho.

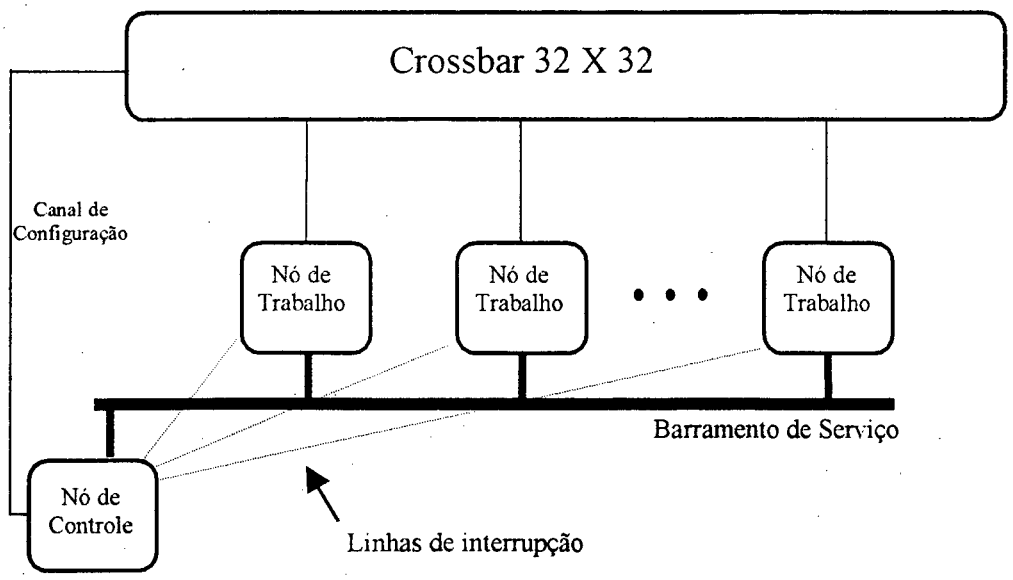


Figura 3.3 – Modelo de Arquitetura com linhas de Interrupção e Barramento Compartilhado

3.2 - ELEMENTOS DO MULTICOMPUTADOR NÓ//

A máquina paralela descrita acima têm como elementos principais aqueles que pertencem ao seu sistema de comunicação. Abaixo são caracterizados cada um desses elementos definidos anteriormente[19] para serem utilizados na implementação do projeto Nó//.

3.2.1 - Nós de Trabalho

Na figura 3.3, é mostrada a estrutura de um nó de trabalho. Cada nó de trabalho deve possuir um processador com memória RAM privativa, uma memória ROM para armazenar o núcleo do sistema operacional, um sistema de comunicação com o *crossbar* e um sistema de comunicação com o nó de controle. O sistema de comunicação com o *crossbar* deve possuir um conjunto de canais para permitir a transmissão eficiente de um grande volume de mensagens. O sistema de comunicação com o nó de controle depende da arquitetura escolhida e será usado para troca de pequenas mensagens de controle.

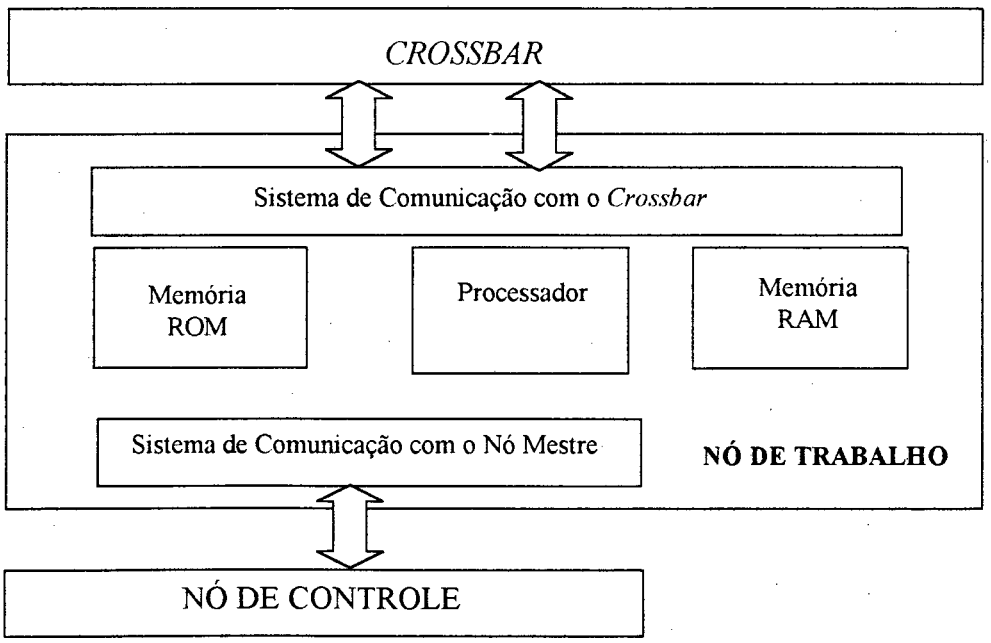


Figura 3.4 – Estrutura de um Nó de trabalho

[19] ZEFERINO, Cesar A., **Projeto do Sistema de Comunicação de um Multicomputador**. Dissertação de Mestrado, Curso de Pos-Graduação em Ciências de Computação, Universidade Federal de Santa Catarina, Florianópolis, SC, 1996, pg. 51-59

3.2.2 - Nó de Controle

Na figura 3.4, é mostrada a estrutura do nó de controle. O nó de controle é responsável pelo controle do sistema comunicação e portanto deve armazenar todo o sistema operacional e uma tabela com os códigos dos serviços que podem ser requisitados por cada nó de trabalho. Deve possuir, então, um processador com memória RAM privativa, uma memória ROM, um sistema de comunicação com o os nós de trabalho e com o *crossbar* e um sistema de configuração do *crossbar*.

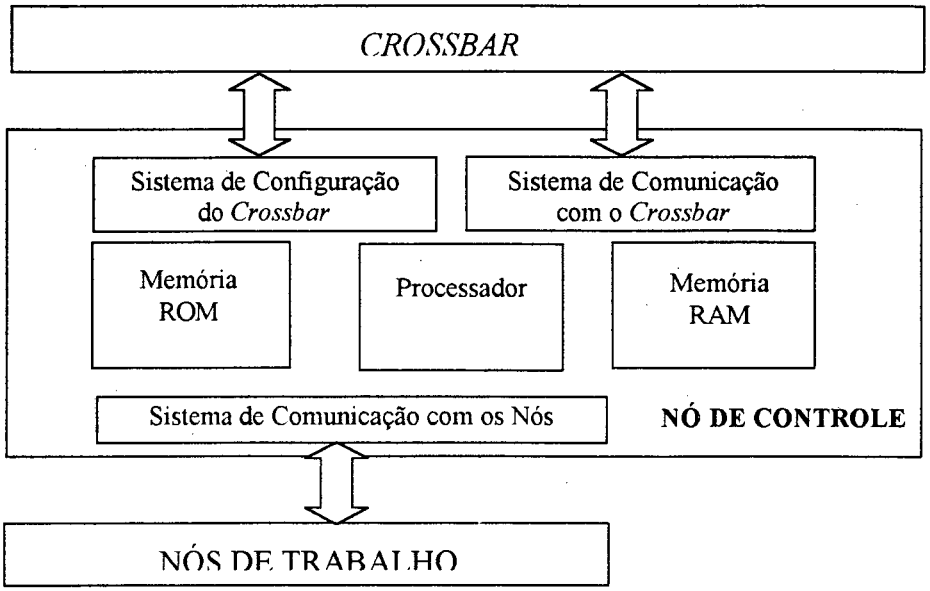


Figura 3.5 – Estrutura do Nó Mestre

3.2.3 - *Link Adapter* C011[20]

É um sistema de interconexão de alta velocidade que proporciona comunicação *full-duplex* entre os membros da família transputer com microprocessadores padrão e subsistemas arquiteturais pela conversão de dados seriais bidirecionais em um feixe de dados paralelos.

O protocolo de comunicação utilizado é o protocolo de enlace serial INMOS, que consiste na transmissão de pacote de dados e uma confirmação de recebimento. O pacote de dado é formado por um *start* bit alto, seguido por mais um bit e oito bit com os dados e um *stop* bit baixo. Após a transmissão do pacote de dados é esperado um pacote de reconhecimento, que consiste em um *start* bit alto e um *stop* bit baixo (Figura 3.2). A

confirmação do reconhecimento indica que um novo processo de transmissão de dados pode iniciar.

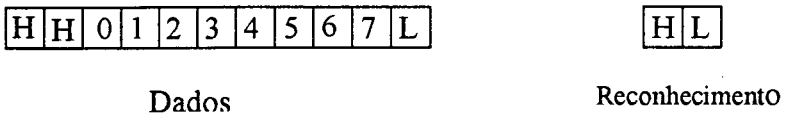
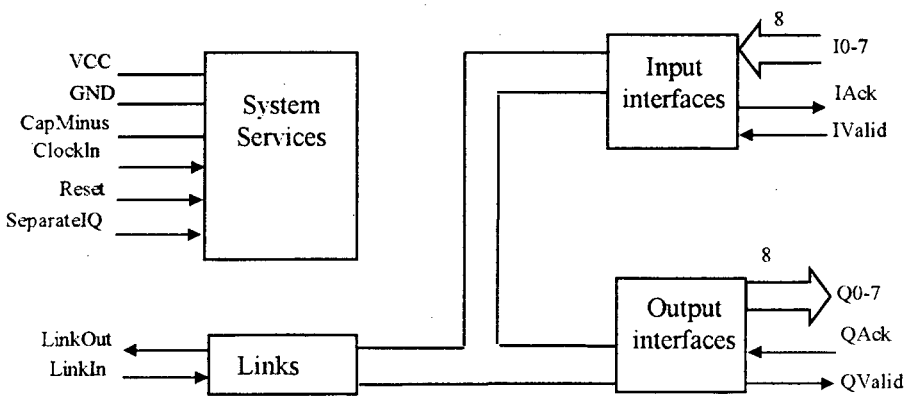


Figura 3.6 – Pacote de dados e reconhecimento dos produtos INMOS

O C011 tem freqüência de comunicação de 10 a 20 Mbits/seg, com recepção assíncrona, permitindo que a comunicação seja independente da fase do clock. Possui dois modos de operação, sendo que para este trabalho, interessa apresentar somente o modo 1.

No modo 1, o *link-adapter* é configurado para converter um link serial em duas interfaces de 1 byte independentes com *handshake*[21], uma de entrada e outra de saída. Pode ser usado na comunicação de um periférico com um transputer, um processador e periférico INMOS ou outro *link-adapter*. A figura 3.2 mostra o diagrama de bloco do C011 no modo 1.



FONTE: INMOS. IMS C011: Link Adaptor., In: INMOS Engineering Data, 1988, pg. 390

Figura 3.7 - Diagrama de blocos do IMS C011 no modo 1

3.2.3.1 - Links

Os links bi-direcionais INMOS proporcionam uma comunicação assíncrona entre produtos INMOS e algum outro produto. Cada *link* compreende um canal de entrada e um canal de saída. Cada byte enviado pelo *link* é reconhecido (ACK) na entrada do mesmo *link*. Dessa forma, cada linha final transporta dados e informação de controle.

---

[20] Fonte: INMOS. IMS C011: Link Adaptor, In: **INMOS Engineering Data**, 1988, pg. 391

3.2.3.2 - Input

Os 8 bits paralelos na porta de entrada I0-7 podem ser lidos pelo dispositivo da família transputer via links seriais. IValid e IAck proporcionam um simples *handshake* para essa porta. Quando o dado está válido em I0-7, IValid é colocado alto pelo periférico indicando que iniciou o *handshake*. O *link-adapter* transmite os dados presentes em I0-7 através dos links seriais. Quando o pacote de reconhecimento é recebido no link de entrada, o C011 seta IAck alto. Para completar o *handshake*, o dispositivo periférico retorna IValid baixo. O *link-adapter* então seta IAck baixo completando a comunicação.

3.2.3.3 - Output

Os 8 bits paralelos na porta de saída podem ser controlados por um dispositivo da família transputer via os links seriais. QValid e QAck proporcionam os sinais de handshake para essa porta. O pacote de dado recebido no link serial é colocado em Q0-7. O link-adapter então faz QValid alto para inicializar o handshake. Após, os dados são lidos de Q0-7, o dispositivo periférico seta QAck alto. O C011 envia, então, um pacote de reconhecimento para o link serial de modo a indicar o fim da transação e seta QValid baixo para completar o handshake.

3.2.3.4 – Descrição da Pinagem

Na Tabela 3.1, é apresentada a descrição dos pinos referentes ao serviços do sistema e interface paralela no modo de operação 1.

---

[21] confirmação de conexão

Tabela 3.1 – Descrição dos Pinos do INMOS C011 no modo de operação 1

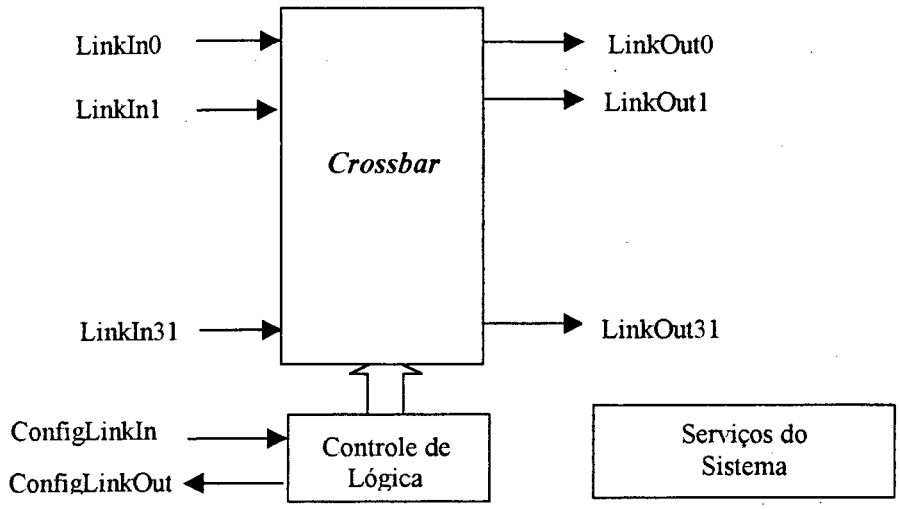
Pino	In/Out	Função
VCC, GND		Fonte de alimentação
CapMinus		Capacitor externo para alimentar relógio interno
ClockIn	In	Relógio de entrada
Reset	In	Reinicialização do sistema
SeparateIQ	In	Seleciona modo de operação e a velocidade dos canais no modo 1
LinkIn	In	Canal de entrada dos dados seriais
LinkOut	Out	Canal de saída dos dados seriais
I0-7	In	Barramento paralelo de entrada
IValid	In	Dado válido em I0-7 (handshake)
IAck	Out	Reconhecimento do dado recebidos em I0-7
Q0-7	Out	Barramento paralelo de saída
QValid	Out	Dado válido em Q0-7
QAck	In	Reconhecimento do dispositivo: dado em Q0-7 foi lido.

FONTE: INMOS. IMS C011: Link Adaptor., In: INMOS Engineering Data, 1988, pg. 391

3.2.4 - *Crossbar* IMS C004[22]

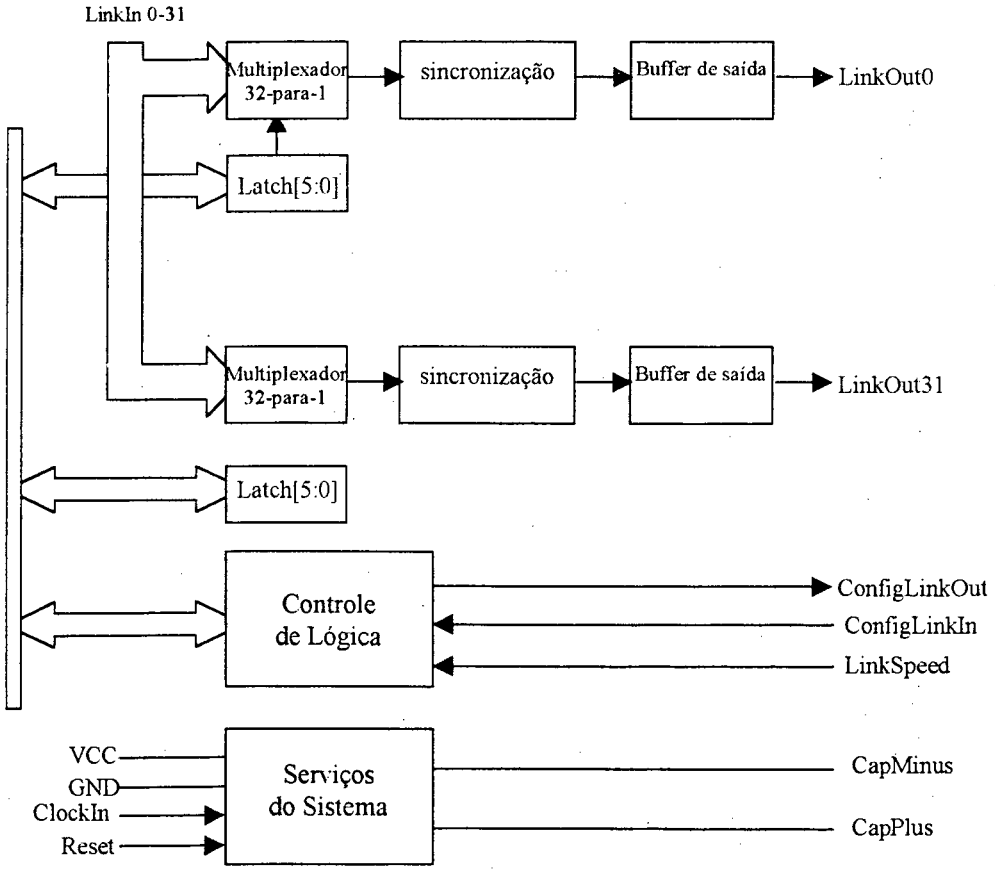
Os canais de comunicação INMOS são sistemas de interconexão de alta velocidade que proporcionam comunicação *full-duplex* entre membros da família transputer, de acordo com o protocolo serial INMOS (item 3.2.1). O C004 é um canal de chaveamento programável projetado para proporcionar um *crossbar* com 32 canais de entrada e 32 canais de saída, como mostra a figura 3.4. Os canais do C004 podem operar a 10 mbits/s ou 20 mbits/s e em média e introduzem um atraso na propagação do sinal da entrada para a saída de 1,75 bits. As chaves do *crossbar* são programadas via um canal especial serial denominado configuration link.

[22] INMOS. IMS C044: Programmable Link Switch, In: **INMOS Engineering Data**, 1988, pg. 479-501



FONTE: INMOS. IMS C044: Programmable Link Switch, In: INMOS Engineering Data, 1988, pg. 479

Figura 3.8 - Diagrama de blocos do IMS - C004



FONTE: Idem, pg. 480

Figura 3.9 – Organização interna do IMS - C004

3.2.4.1 – Programação das Chaves do *Crossbar* IMS C004

O *crossbar* é organizado internamente como um conjunto de 32 multiplexadores 32-para-1 (Figura 3.5). Cada multiplexador tem associado a ele um *latch* de 6 bits, dos quais, cinco bits selecionam uma entrada como origem dos dados para a correspondente saída. O sexto bit é usado para conectar ou desconectar a saída. Esses *latch* podem lidos ou escritos por mensagens enviadas pelo canal de configuração via ConfigLinkIn a ConfigLinkOut.

A saída de cada multiplexador é sincronizada com um clock interno de alta velocidade e essa sincronização gera um atraso de 1,75 bit. O sinal não é eletricamente degradado, o que permite que seja utilizado um número arbitrário de canais.

Cada entrada e cada saída é identificada por um número entre 0 e 31. A mensagem de configuração consiste de um, dois ou três bytes transmitidos pelo canal de configuração. As mensagens de configuração enviadas para as chaves são mostradas na Tabela 3.2.

Tabela 3.2 – Mensagens de configuração do IMS C004

Mensagem de configuração	Função
[0] [input] [output]	Conexão de input com output.
[1] [canal 1] [canal 2]	Liga o canal 1 com o canal 2 conectando a entrada do canal 1 à saída do canal 2 e a entrada do canal 2 com a saída do canal 1.
[2] [output]	Questiona qual entrada está conectada à output. O IMS C004 responde com a entrada. O bit mais significativo do byte indica se output está conectado (bit setado alto) ou desconectado (bit setado baixo).
[3]	Esse byte de comando deve ser enviado no final de cada seqüência de configuração. O IMS C004 está pronto para aceitar dados nas entradas conectadas.
[4]	Reinicializa o *crossbar*. Todas as saídas são desconectadas e mantidas em nível baixo. O mesmo ocorre quando o IMS C004 é resetado.
[5] [output]	Saída output é desconectada e mantida em nível baixo.
[6] [canal 1] [canal 2]	Desconecta a saída do canal 1 e a saída do canal 2.

3.2.4.2 – Descrição da Pinagem

Na Tabela 3.3 é apresentada a descrição da pinagem do IMS C004.

Tabela 3.3 – Pinagem do INMOS C004

Pino	In/Out	Função
VCC, GND		Fonte de alimentação e terra
CapPlus, CapMinus		Capacitor externo para alimentar relógio interno
ClockIn	In	Relógio de entrada
Reset	In	Reinicialização do sistema
DoNotWire		Não deve ser ligado
ConfigLinkIn	In	Configuração dos canais de entrada
ConfigLinkOut	Out	Configuração dos canais de saída
LinkInI0-31	In	Canais de entrada para o *crossbar*
LinkOutI0-31	Out	Canais de saída do *crossbar*
LinkSpeed	In	Seleção da velocidade de comunicação

Fonte: INMOS. IMS C004: IMS C004, Programmable Link Switch, In: INMOS Engineering Data, 1988, pg. 381

3.3 - RESUMO

Nesse capítulo, foram apresentados, de forma sucinta, o projeto Nó// e as propostas de arquitetura para a máquina paralela, com ênfase na descrição dos elementos do sistema de comunicação já definidos, para serem utilizados na implementação da máquina paralela.

O sistema de comunicação da máquina paralela prevê a utilização de uma rede de interconexão dinâmica - do tipo *crossbar*, para a comunicação entre os vários processadores que deverão trocar grandes pacotes de mensagens.

CAPÍTULO 4 – METODOLOGIAS DE PROJETO

INTRODUÇÃO

Este capítulo apresenta as metodologias utilizadas no desenvolvimento de uma rede de interconexão do tipo *crossbar* que segue as mesmas características funcionais do *crossbar* programável C004 da INMOS, apresentada no capítulo anterior, adaptado especificamente para o sistema de comunicação do multicomputador paralelo.

Inicialmente, faz-se uma discussão sobre os dispositivos lógicos programáveis (PLD), utilizados na implementação física de sistemas digitais. Logo em seguida, apresentamos o VHDL – uma linguagem de programação permite a descrição lógica de sistemas digitais em alto nível. Na seção 4.3, é apresentado o ambiente de programação MAX+PLUS II utilizado como ferramenta para o desenvolvimento do projeto. Na última seção, é descrito a modelagem em máquina de estado, utilizada na descrição comportamental dos circuitos lógicos seqüências do *crossbar*.

4.1 - LÓGICA PROGRAMÁVEL[23]

Dispositivos de lógica programável (PLD) são circuitos integrados que projetistas de *hardware* podem programar para realizar funções lógicas específicas. O FPGA (Field Programmable Gate Array) é um exemplo desses dispositivos e consiste de um circuito integrado que pode ser configurado por *software* para implementar circuitos digitais, como processadores, controladores, multiplexadores, etc. Internamente é organizado como um arranjo com um alto grau de compactação de blocos idênticos de pequenos circuitos (portas

lógicas e *flip-flops*) com poucos sinais de interface. As conexões entre a saída de determinados blocos com a entrada de outro são programáveis eletricamente através de um protocolo simples e de fácil implementação.

Existem quatro tecnologias de programação disponíveis[24]: RAM estática, transistores de passagem, EPROM e EEPROM. Dependendo da aplicação, uma determinada tecnologia FPGA pode apresentar melhor desempenho para o objetivo desejado.

RAM estática implementa as conexões entre blocos lógicos através de portas de transmissão ou multiplexadores controlados por células SRAM. Essa tecnologia tem a vantagem de permitir uma rápida reconfiguração do circuito. Em contrapartida, exige um considerável *hardware* auxiliar, a ser integrado junto com os blocos lógicos, para que se permita a configuração das conexões.

Os transistores de passagem exigem grande concentração de transistores que podem ser configurados em modo corte ou condução. Em corte, introduz alta impedância entre dois nós internos. Em condução, implementa a conexão entre eles. É mais barata que as RAM estática.

As tecnologias EPROM e EEPROM têm a vantagem de permitir a reprogramação sem armazenamento de configuração externa, embora os transistores EPROM não possam ser reprogramados no próprio circuito. As EEPROM podem ser reprogramadas eletricamente.

A reprogramação da funcionalidade de um FPGA, ou seja, a forma como o circuito irá operar, é realizada através de uma matriz de dados por um protocolo simples, via comunicação serial ou paralela. Em FPGAs baseados em tecnologias SRAM isso é conseguido através de uma memória externa auxiliar que deve ser lida quando se inicializa o sistema em questão, pois o FPGA é de configuração volátil. Com isso, é possível programar ilimitadamente um FPGA, permitindo que um mesmo protótipo possa ser utilizado para diferentes fins.

Como vantagens, na utilização de lógica programável no desenvolvimento de um projeto de *hardware*, pode-se citar: redução do tempo e custo de desenvolvimento de um projeto, alta flexibilidade para mudanças e atualizações no projeto (manutenção), maior confiabilidade do produto final, quando comparadas ao uso de dispositivos discretos.

---

[23] COLI, Vicent J., "**Introduction to Programmable Array Logic**", Byte, Jan, 1987, pg.207

[24] MENDONÇA,Alexandre & ZELENOVSKI, Ricardo & PINHO, André & ALVARES, Marco., "**FPGA: uma Nova Alternativa para a Criação de Protótipos**", Developers, Jan, 1998, pg. 12

Os principais fornecedores da tecnologia de FPGAs[25] são: Altera e Plus Logic usando a tecnologia baseada nos EEPROM, Xilinx, Toshiba e Altera utilizando a tecnologia baseada em SRAM. Pela disponibilidade, sugere-se nesse trabalho, optar-se pelos os elementos fornecidos pela Altera.

A Altera fabrica diversos modelos de FPGAs, agrupados em famílias, como, por exemplo, as famílias MAX 7000, FLEX 8000 e FLEX 10K. A família MAX 7000 utiliza tecnologia EEPROM e é adequada a projetos baseados em lógica combinacional intensiva e/ou que requerem freqüências de operação elevadas. Alguns modelos operam a mais de 178 MHz. Já a família FLEX 8000 e FLEX 10K utiliza a tecnologia de programação SRAM e é apropriada para o uso em projetos com lógica seqüencial intensiva (*flip-flops*, registradores, etc). Abaixo está relacionada as principais características da duas últimas famílias:

4.1.1 – Característica da família FLEX 8000[26]

A família 8000 foi a primeira da linha da Altera que era programada por memória RAM interna. Cada *chip* possui diversos encapsulamentos, variando de 78 pinos até 208. Tem como principais caraterísticas:

- custo: US$ 7 para 8282;
- portas disponíveis: entre 5K (8282) até 32K (81500);
- portas utilizáveis: entre 2K5 (8282) até 16K (81500);
- flip-flops: de 282 até 1500;
- pinos: de 78 até 208;
- Logic Elements (Elementos Lógicos): de 208 até 1296;
- Freqüência: até 100 MHz.

4.1.2 – Característica da família FLEX 10K

A família 10K foi a primeira da linha da Altera que era permitia o uso de blocos de RAM, seja como elementos de memória ou como elemento lógico. Cada *chip* possui

---

[25] ROSE, Jonathan & EL GAMAL, Abbas & SANGIOVANNI-VINCENTELLI, Alberto. **Architecture of Field-Programmable Gate Array**, Proceeding of the IEEE, Vol. 81, Num. 07, Jul, 1993.

[26] CARRO, Luigi. **Síntese em FPGAs.** V-ERI – Escola Regional de Informática. Maio, 1997

diversos encapsulamentos, variando de 104 pinos até 406. Tem como principais caraterísticas:

- custo: US$ 60 para 10K10;
- portas disponíveis: entre 10K e 100K;
- portas utilizáveis: entre 7K e 62K;
- flip-flops: de 720 até 5392;
- pinos: de 104 até 406;
- Logic Elements (Elementos Lógicos): de 576 até 4992;
- Bits de RAM: de 6144 até 24576;
- Freqüência: até 80 MHz.

Na figura 4.1 é apresentado a arquitetura interna de um FPGA da família FLEX 8000 e FLEX 10K. A arquitetura de um FPGA é composta de pinos de entrada e saída (células de I/O) que acessam os canais de interconexão, que por sua vez ligam os diversos LABs (*Logic Array Blocks*) entre si.

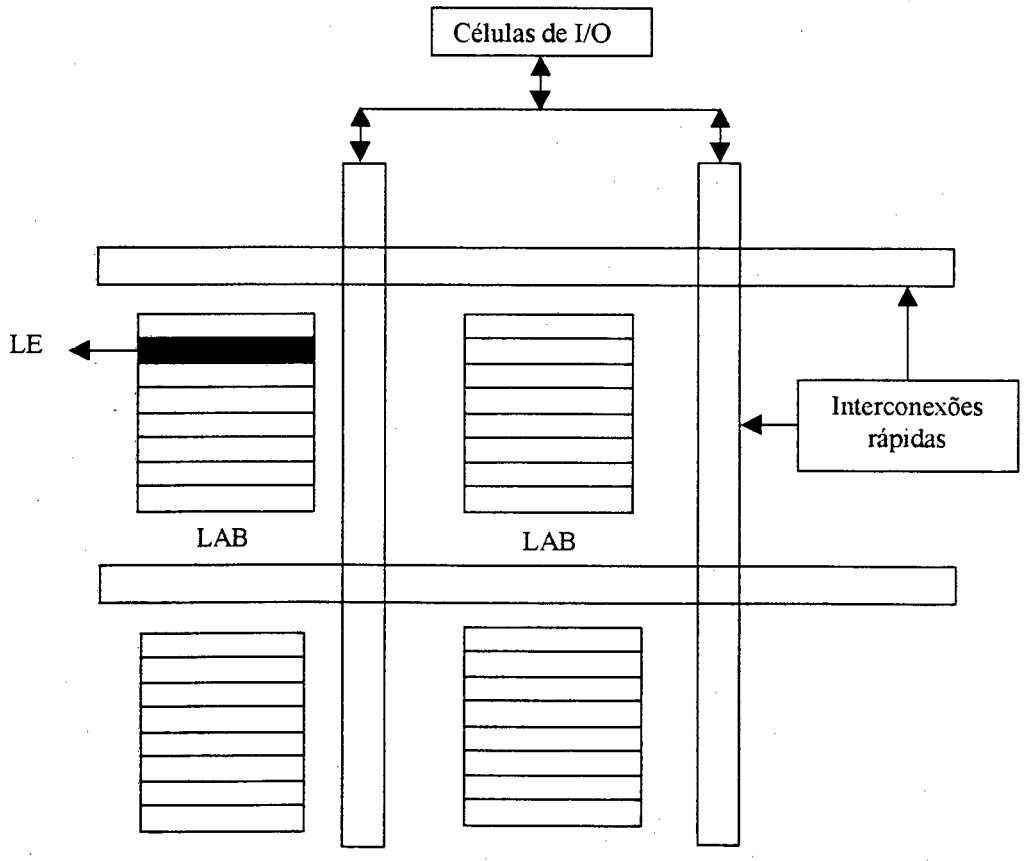


Figura 4.1 – Arquitetura interna de FPGA da família 8000

Cada LAB é formado por oito elementos lógicos (*Logic Element*) (fig. 4.2). Cada elemento lógico possui uma memória para a programação da função lógica (*look-up table* ou LUT), um *flip-flop*, uma cadeia de *carry* e uma cadeia de expansão lógica. A linha de *carry* é utilizada para programar funções lógicas de um LE para outro, como a cadeia de *carry* de um somador. Desta maneira, a ligação economiza recursos de conexão externos aos LEs. A linha de expansão é utilizada para implementar funções com mais de 4 variáveis de entrada.

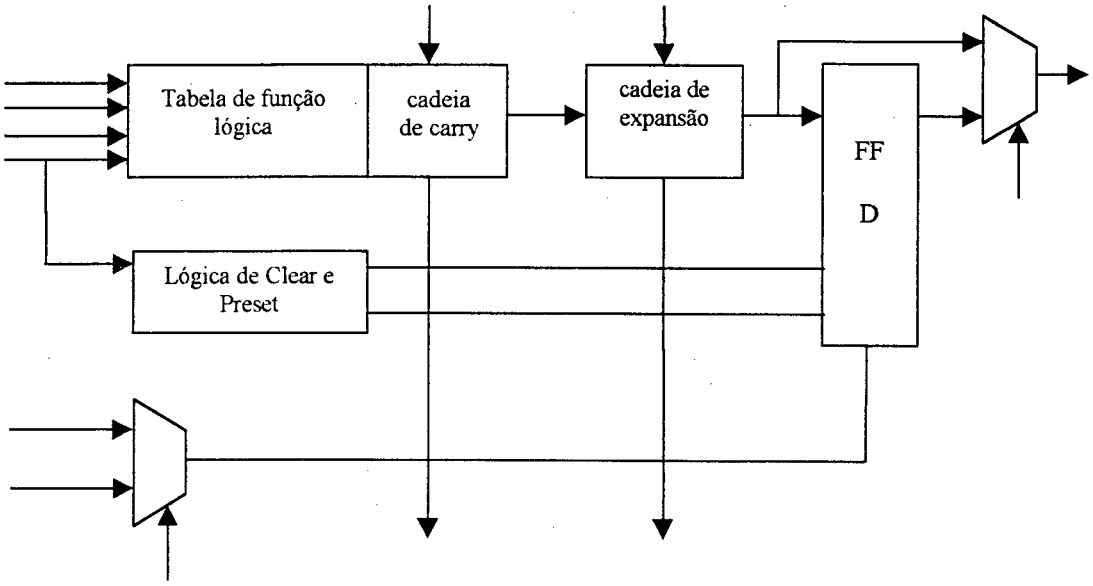


Figura 4.2 – Arquitetura de um LE da família 8000

Além dessas, a Altera fabrica outras famílias de FPGAs, com diferentes custos, densidades, arquiteturas e tecnologias.

4.2 – VHDL – LIGUAGEM DE DESCRIÇÃO DE HARDWARE[27]

No decorrer da história dos sistemas eletrônicos, projetistas têm usado linguagens de descrição de *hardware* (HDLs) para a construção de circuitos integrados (ICs). VHDL é uma linguagem para a descrição de sistemas eletrônicos digitais, resultado de pesquisas do Programa de Circuitos Integrados de Altíssima Velocidade (VHSIC) iniciado em 1980, projeto encomendado pelo Departamento de Defesa dos Estados Unidos. Mais tarde, em 1983, pela necessidade de uma linguagem padrão para projeto de circuitos integrados, que se tornavam cada vez mais complexos, VHDL (VHSIC *Hardware Description Language*) foi

[27] ASHENDEN, Peter J., **The VHDL Cookbook**, Dept. Computer Science – University of Adelaide, First Edition, 1990

desenvolvida e adotada como padrão pelo *Institute of Eletrical and Electronic Engineers* (IEEE).

VHDL permite a descrição hierárquica de projetos; suporta o desenvolvimento concorrente em módulos nos projetos de sistemas complexos; além de possibilitar a ampliação de sua biblioteca de componentes pré-definidos, que podem ser incluídos em projetos sem a necessidade de reescrevê-los.

O VHDL permite a descrição de sistemas digitais em dois diferentes níveis de abstração: a descrição a nível de portas lógicas, detalhando a estrutura interna de um módulo lógico e a descrição em alto nível, descrevendo informações sobre as funções desempenhadas de um módulo lógico.

4.2.1 – Descrição Estrutural

Um sistema digital pode ser descrito como um módulo com entradas e/ou saídas. Os valores das saídas são função de alguma função dos valores de entradas. Por este ponto de vista, são considerados os detalhes da composição interna dos componentes. São realizadas declarações e instanciações, onde as declarações definem as interfaces dos subcomponentes utilizados na entidade, e as instanciações permitem a criação de instâncias deste componente, quantas forem necessárias Usando a terminologia VHDL, nos temos um módulo digital ou uma *entity*, e as entradas e saídas são denominadas *ports*. Desta forma, um multiplexador pode ser descrito como segue:

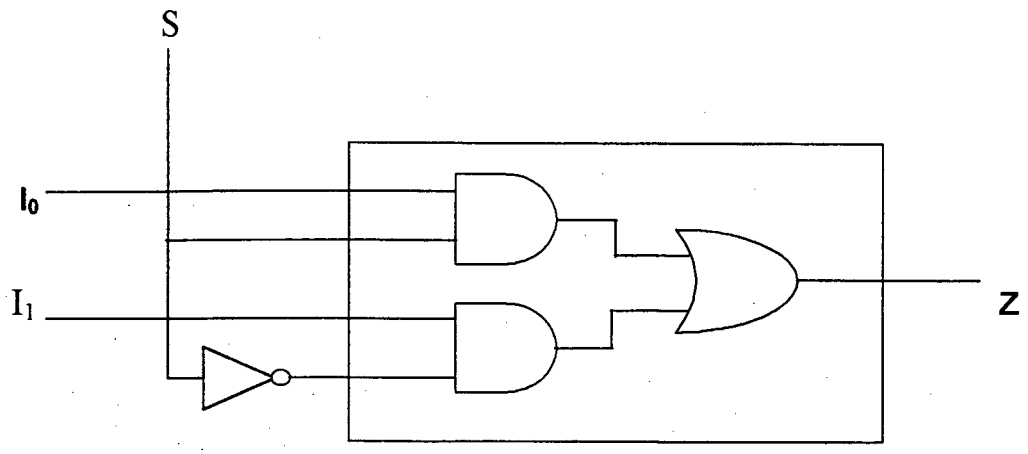


Figura 4.3 – Descrição estrutural de um MUX 2x1

4.2.1 – Descrição Comportamental

Em muitas situações, não existe a necessidade de detalhar a estrutura interna de um circuito integrado, informações sobre as funções desempenhadas são suficientes para seu projeto. Esse é o estilo de descrição funcional ou comportamental. A descrição comportamental, consiste na interpretação funcional do sistema. Nesse estilo de descrição de sistemas digitais, são declaradas estruturas de dados e definidas as operações que serão executadas sobre estas estruturas. Desta forma, o mesmo multiplexador pode ser descrito como:

$$Z = S I_0 + \overline{S} I_1$$

Ao descrever um sistema digital, se desempenho for o fator mais importante, o estilo estrutural é recomendado. No entanto, o estilo comportamental proporciona maior simplicidade e eficiência em curto espaço de tempo. No mesmo projeto, pode-se optar pela descrição pura, em um único estilo, ou então combinar aspectos comportamentais e estruturais.

4.2.3 – Primitivas VHDL

1. ***Entity***: A entidade é a parte mais simples na construção de sistemas digitais. Uma entidade representa um componente de *hardware*. Cada componente tem um conjunto de sinais ou pinos que são a interface com o ambiente externo. Estes sinais são denominados de *ports*.

Na declaração de cada *port*, é necessário informar o tipo e modo de operação. O tipo indica o domínio dos valores que passarão por ele e o modo de operação indica se o sinal é de entrada (*in*), de saída (*out*) ou bidirecional (*inout*). A declaração de entidade é apresentada abaixo:

```
ENTITY entity_name IS
  PORT( input_name          : IN STD_LOGIC;
        input_vector_name   : IN STD_LOGIC_VECTOR;
        bidir_name          : INOUT STD_LOGIC;
        output_name         : OUT STD_LOGIC);
END entity_name;
```

2. ***Architecture***: A arquitetura define a implementação do projeto. Descreve a estrutura ou comportamento interno da entidade. Uma mesma entidade pode conter mais de uma implementação de arquitetura.

Um corpo arquitetural também define itens, como variáveis, constantes e sinais, que são usado na implementação do projeto. Um componente pode ser simulado desde que tenha sua arquitetura declarada. O modelo de declaração de uma arquitetura é como segue:

```
ARCHITECTURE a OF entity_name IS
SIGNAL signal_name1 : STD_LOGIC;
SIGNAL signal_name2 : STD_LOGIC;
BEGIN
    (Corpo arquitetural)
END a;
```

3. ***Signal*** (sinal global): O conceito de sinais globais é similar ao conceito de variáveis globais nas linguagens de programação convencionais. Esse recurso de VHDL é usado para descrever a comunicação interprocessos, ou seja, trocar informações entre os vários processos de um sistema. No entanto, é necessário cuidado extremo para que num mesmo instante em que um processo esteja alterando o valor de um *signal*, esse não seja acessado por outro processo. Devido a esse fato, são necessários, mecanismos de sincronização, que são obtidos através do comando *wait*. A forma de declaração de um *signal* é conforme apresentado:

```
SIGNAL signal_name : type_name;
```

4. ***Variables***: Em VHDL, um variável é uma representação ou agregação de valores de um determinado domínio. As variáveis são declaradas e usadas somente em processos e subprogramas, ou seja, não podem ser usadas para comunicação entre processos. O valor de uma variável pode ser alterado através de uma instrução <= (*gets*), semelhante à de atribuição. Uma variável é declarada:

```
VARIABLE variable_name : type_name;
```

5. ***Process***: : Um processo é uma unidade básica de execução em VHDL. Define um conjunto de comandos seqüenciais (*if, case, wait*, etc.) que representam

comportamentos específicos adotados pelo sistema quando estiver ativo. Em um projeto de sistemas digitais, os processos são executados concorrentemente entre si.

```
Process_name: PROCESS (lista de sensitivos)
VARIABLE variable_name : type_name;
BEGIN
<instruções seqüenciais>
END PROCESS process_name;
```

6. ***Package***: A descrição de um sistema digital em VHDL, as estruturas de dados, constantes, sinais e processos são visíveis apenas pela própria entidade e por sua arquitetura. Quando outras entidades necessitarem destas declarações, estas devem ser feitas em forma de pacotes (*packages*). Um pacote é uma coleção de tipos, constantes, subprogramas, agrupados de forma que possam ser usados por diversas entidades. Sua aplicação é semelhante às funções pré-definidas nas linguagens de programação convencionais.

Um pacote pode ser visto como duas partes diferentes, semelhante a uma entidade. A interface, que contém as declarações visíveis pelo projetista, e o corpo que não é visível ao mundo externo. O corpo do pacote pode ser omitido se não houver necessidade de defini-lo. Como exemplo, é apresentada a declaração de um pacote que contém tipos de dados:

```
package data_types is
subtype address is bit_vector(24 downto 0);
subtype data is bit_vector(15 downto 0);
constant vector_table: address;
function data_to_int(v:data) return integer;
end data_types;
```

7. **Subprogramas**: VHDL também oferece recursos de subprogramação na forma de funções e procedimentos. Um subprograma é um conjunto de declarações e parâmetros que pode ser usado em diferentes partes de uma descrição VHDL.

A execução de um subprograma (procedimento) é ativada pela chamada de seu identificador, e a troca de valores é feita pela lista de parâmetros, que podem ser variáveis ou sinais. No caso das funções, é possível retornar somente valores do tipo pré-definido do função. A declaração de procedimentos e funções são realizadas conforme os seguintes modelos:

```
procedure id_procedure [<lista_de_parâmetros>]
<parte_declarativa>
begin
<comandos_seqüenciais>
end id_procedure;

Function id_function [<lista_de_parâmentros>]
return tipo_do_valor_retornado
<parte_declarativa>
begin
<comando_seqüenciais>
end id_function
```

8. **Tipos de Dados**: A definição de tipo de dados, especifica o conjunto de valores manipulados por variáveis, sinais ou constantes. Os tipos básicos simples: escalar, inteiro, quantidades físicas, literais; e compostos (*array, record*). Além dos tipos básicos, novos tipos de dados podem ser criados, declarando-se nome, domínio e operações possíveis. Exemplos de declaração de tipos são apresentados abaixo:

```
TYPE type_state IS (state1, state2);
TYPE array_type IS ARRAY (15  downto 0) OF type_name;
```

9. **Instruções Seqüenciais**: As mudanças de estado dos objetos e o controle de fluxo na execução de um sistema digital são feitas através de instruções seqüenciais.

9.1. A estrutura IF-THEN-ELSE, permite a seleção das instruções a serem executadas, dependendo de uma ou mais condições. A sintaxe imita a sintaxe em outras linguagens de programação como por exemplo, o PASCAL:

```
 if <condição> then
instruções
{ elsif <condição> then
instruções}
[ else
instruções]
end if ;
```

9.2. A estrutura CASE, permite a implementação de máquinas de estado:

```
case expressão is
   when choices =>
        instruções1;
   others
        instruções2;
end case ;
```

9.3 O Loop em VHDL tem como instruções básicas de repetição definidas na forma de While e Loop, comuns na maioria das linguagens de programação, declarados como:

```
loop_label :[ iteration_scheme ] loop
sequence_of_statements
end loop [ loop_label ] ;

iteration_scheme ::= while condition
| for loop_parameter_specification
parameter_specification ::=
identifier in discrete_range
```

4.3 – O AMBIENTE DE PROJETO DE FPGA

O Departamento de Engenharia Elétrica da UFSC participa, desde de 1994, de um convênio com a Altera, chamado de Programa Universitário Altera. Através desse convênio, essa empresa fornece a Universidade licenças de *software*, *hardware* de programação e circuitos FPGAs para o uso em ensino e pesquisa. A infra-estrutura laboratorial para a condução desse convênio é fornecida pelo LINSE (Laboratório de Instrumentação de Eletrônica), que, através do Prof. Márcio Schneider, responsável pelo Programa Universitário Altera na UFSC, disponibilizou uma licença do *software* para o uso no ambiente do LMA-INE (Laboratório de Microprocessadores e Automação do Departamento de Informática e Estatística).

4.3.1 – O ambiente de desenvolvimento Altera MAX+PLUS II

Para o desenvolvimento de projetos baseados em seus componentes, a Altera fornece um ambiente computacional chamado MAX+PLUS II. Esse ambiente consiste de um conjunto de ferramentas para entrada de projeto, compilação, simulação, análise de

"*timing*" e programação, entre outras. O fluxo de projeto utilizando o MAX+PLUS II é ilustrado na figura 4.2 .

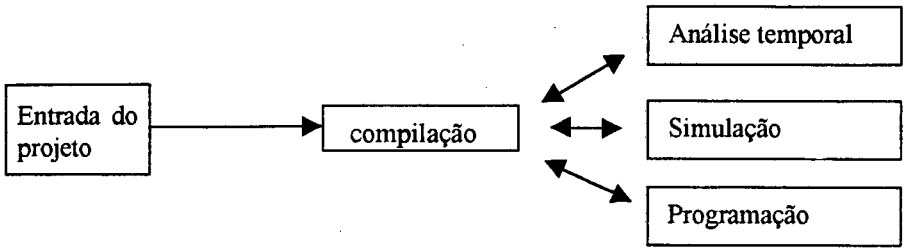


Figura 4.4 – O fluxo do projeto

A entrada do projeto pode ser feita através da captura do esquemático, edição das formas de ondas de entrada e saída ou pelo uso de linguagens de descrição de *hardware*, como o VHDL ou AHDL.

A compilação envolve etapas como minimização lógica, otimização e mapeamento tecnológico (colocação e roteamento) do projeto no FPGA escolhido. As etapas seguintes permitem a realização de simulação funcional ou temporal, análise de tempos de atraso, de estabilização e de manutenção, para a determinação de caminhos críticos e, por fim, a programação em FPGA.

4.3.1.1 – Simulação em VHDL

Entre os diversos recursos e facilidades oferecidas por VHDL, a simulação está entre as mais importantes. Tendo definida a arquitetura de um componente, é possível realizar a simulação desse componente.

O tempo de simulação é dividido em intervalos discretos, e nesse intervalo o modelo adotado segue o paradigma de respostas a estímulos, no momento em que é estimulado, responde através de eventos definidos na sua arquitetura, de acordo com os sinais de entrada e/ou o estado em que se encontra.

O principal objetivo de uma simulação é obter informações sobre as ações do sistema em determinadas condições num intervalo de tempo definido. Com base nas informações obtidas, o projetista pode avaliar os resultados do projeto, dispensando métodos de prototipação.

Quando um processo reage a um estímulo, após um breve retardo, responde ao estímulo alterando os valores dos sinais de saída, registrando um novo evento.

4.4 – MODELAGEM EM MÁQUINA DE ESTADOS

O modelo de máquinas de estados é um modelo abstrato de mecanismos de controle, que podem ser encontrados em qualquer dispositivo determinístico de entrada e saída, como em sistemas digitais.

Ele consiste de um conjunto de estados, que representam a situação em que se encontra um determinado sistema, e de transições, determinadas pelo estado corrente e pelos sinais de entrada do sistema.

Um sistema digital é visto como um conjunto de passos ou transições e como um conjunto de estados, pelos quais esse sistema transita durante seu funcionamento. Um estado, em sistemas digitais, é definido por sinais de voltagem, de forma discreta, como sinais altos e baixos. Sendo que o estado atual é determinado pelo estado anterior e pelos sinais de entrada do sistema, podendo gerar um conjunto de sinais discretos como saída.
Aqui abordaremos dois tipos básicos de máquinas de estados: a máquina de Moore e a máquina de Mealy. Ambas têm as características básicas de uma máquina de estados, no entanto, têm diferenças marcantes, sendo que as funções de controle das máquinas de leitura e escrita foram implementadas através de máquinas de estados baseadas nos conceitos de máquina de Moore.

4.4.1 – Máquina de Moore

Os sinais de saída são determinados somente pelo estado anterior, ou seja, independe dos sinais de entrada. Isso significa que os sinais de saída são sempre os mesmo para um determinado estado. Para cada estado, a máquina de Moore gera uma única saída possível. O diagrama desse modelo de máquina de estados é mostrado na figura 4.3.

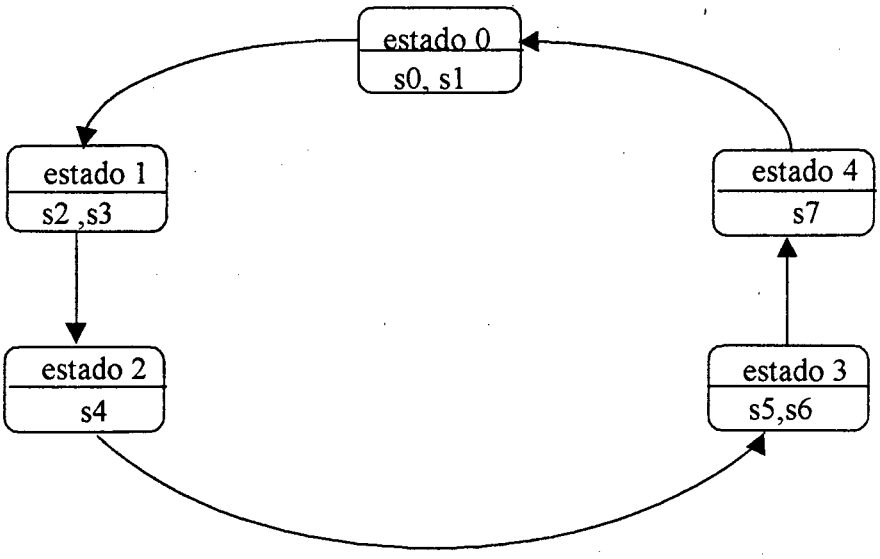


Figura 4.5 – Máquina de Estados de Moore

4.4.2 – Máquina de Mealy

No modelo de máquina de estados de Mealy os sinais de saída são gerados a partir do estado presente e a partir dos sinais de entrada da máquina. Desse modo, é possível gerar diferentes saídas para um mesmo estado. O diagrama de estados da máquina de Mealy é mostrado na figura 4.4.

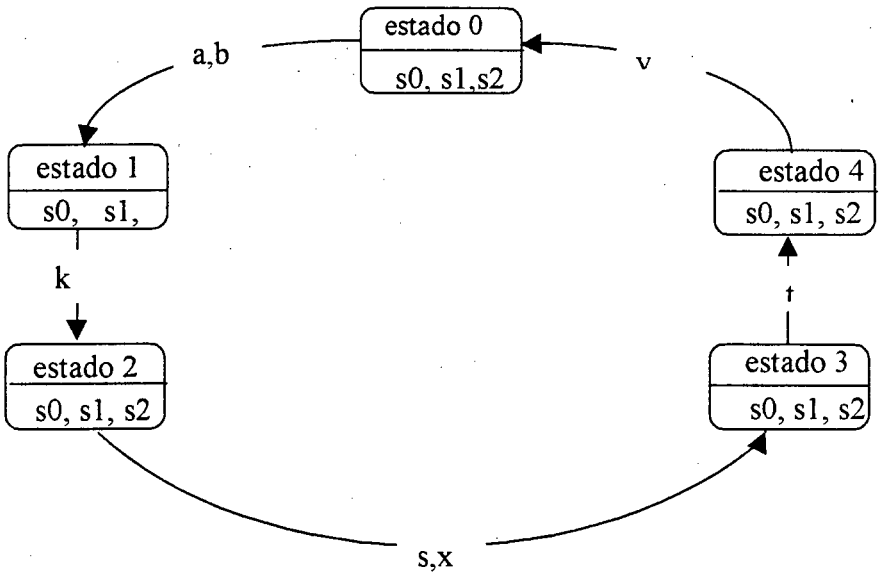


Figura 4.6 – Máquina de Estados de Mealy

CAPÍTULO 5 – PROJETO DA REDE DE INTERCONEXÃO

INTRODUÇÃO

No capítulo 3 foi apresentado os elementos definidos para a implementação do Projeto Nó//, no sistema de comunicação, foi definido a utilização da rede de intercomunicação dinâmica do tipo *crossbar* – o C004 da INMOS.

Pelas dificuldades comerciais na obtenção deste dispositivo eletrônico, pois no mercado nacional ele não é mais encontrado, sendo encontrado somente em fornecedores internacionais, o que dificulta a sua aquisição, optou-se pelo desenvolvimento de um componente similar implementado fisicamente em FPGA e logicamente em VHDL. O ambiente de programação utilizado foi o MAX+PLUS II.

Outra justificativa para o projeto é o domínio e a disponibilização da tecnologia de projeto e implementação utilizando dispositivos lógicos programáveis, que possibilita ainda, a customização do *crossbar*, em função das necessidades atuais do Nó//.

Este capítulo apresenta a implementação lógica de um componente com as mesmas características funcionais do C004.

5.1 - PROTÓTIPO DO *CROSSBAR*

Inicialmente, propõe-se a implementação de um *crossbar* com somente quatro canais de entradas e quatro canais de saídas, como mostrado na figura 5.1, número de entradas e saídas suficiente para a validação do modelo de *hardware*.

Como ferramenta de projeto e implementação, foram utilizados os dispositivos da Altera, o ambiente de desenvolvimento MAX+PLUS II e entrada de projeto via linguagem de descrição de *hardware* – VHDL.

O projeto consiste no desenvolvimento de um crossbar que se comporte como o *crossbar* IMS C004 da INMOS, ou seja, para a configuração das chaves de entrada e saída utilize as mensagens de configuração da Tabela 3.2.

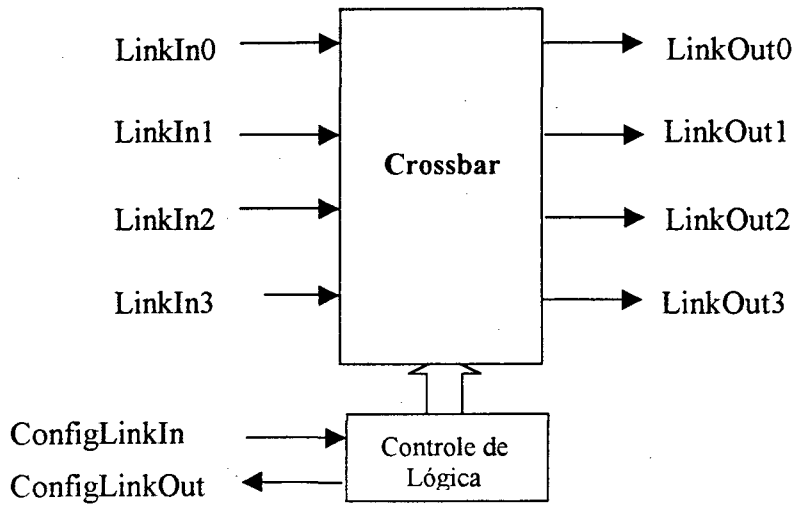


Figura 5.1 – Protótipo do crossbar

Este modelo inicial foi proposto para efeito de simplificação no desenvolvimento e permite testar as conexões entre as entradas e saídas em função das mensagens de configuração. Esta simplificação nos permite incluir uma nova mensagem, que determina que uma entrada especificada seja conectada a todas as saídas simultaneamente (*broadcasting*). Cada mensagem consiste de 1, 2, ou 3 bytes que são transmitidos pelo link serial "*configuration link*" (ConfigLinkIn). Então a Tabela 5.1 apresenta as mensagens de configuração implementadas no protótipo, com o conjunto de bits que formam o byte de cada instrução. A cada byte recebido, e enviado um sinal de reconhecimento pelo canal ConfigLinkOut. Este mesmo canal é utilizado como meio de transmissão do *status* de um canal de saída (mensagem 2). Input e output correspondem os endereços das entradas e saídas respectivamente, que neste caso variam de "00000000" a "00000011".

Tabela 5.1 – Mensagens de configuração

Mensagem de configuração	Função
00000000 [input] [output]	Conexão de input com output.
00000001 [canal 1] [canal 2]	Liga o canal 1 com o canal 2 conectando a entrada do canal 1 à saída do canal 2 e a entrada do canal 2 com a saída do canal 1.
00000010 [output]	Questiona qual entrada está conectada à output. O IMS C004 responde com a entrada. O bit mais significativo do byte indica se output está conectado (bit setado alto) ou desconectado (bit setado baixo).
00000011	Este byte de comando deve ser enviado no final de cada seqüência de configuração. O IMS C004 está pronto para aceitar dados nas entradas conectadas.
00000100	Reinicializa o crossbar. Todas as saídas são desconectadas e mantidas em nível baixo. O mesmo ocorre quando o IMS C004 é resetado.
00000101 [output]	Saída output é desconectada e mantida em nível baixo.
00000110 [canal 1] [canal 2]	Desconecta a saída do canal 1 e a saída do canal 2.
00000111 [input]	Conecta input a todas as saídas

5.2 – PROJETO DE HARDWARE

Adotando o modelo de caixa preta para representar cada módulo, inicialmente podemos dividir o hardware em questão em dois módulos, como apresentado na Figura 5.2. O módulo 1 corresponde a lógica de controle, formado por circuitos combinacionais e seqüenciais, responsáveis pela geração de sinais que controlam o módulo 2, um circuito exclusivamente combinacional. que corresponde a um conjunto de multiplexadores onde estão conectados os canais de entrada e saída. O módulo 1 tem como sinal de entrada o canal serial ConfigLinkIn e como sinal de saída o canal serial ConfigfLinkOut. O módulo 1 gera o conjunto de sinais MU3..0[3..0], que faz a conexão entre os canais de entrada LinkIn[3..0] e os canais de saída LinkOut[3..0] no módulo 2.

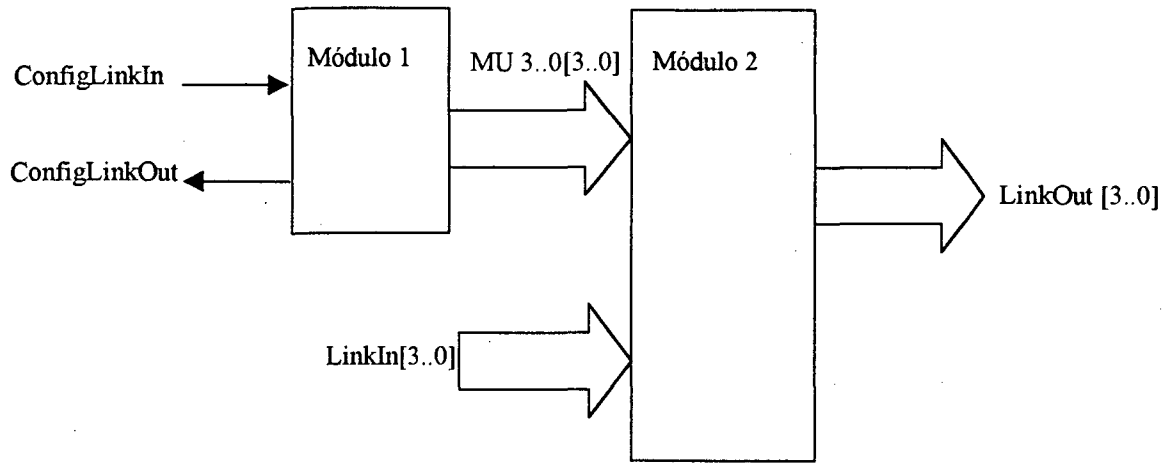


Figura 5.2 – Representação lógica do crossbar em dois módulos

5.2.1 – Módulo 1 – Lógica de controle

O módulo 1 corresponde a lógica de controle composta por um conjunto de circuitos combinacionais representados pelos decodificador e multiplexadores e circuitos seqüenciais representados pelas máquinas de estados, latchs e registradores de deslocamento. Na Figura 5.3 é apresentado os vários sub-módulos que compõe a lógica de controle.

5.2.1.1 – Módulo 1.1 – Máquinas de leitura

O módulo 1.1 da Figura5.3 é representado por duas máquinas de estados: 1) a máquina de estado que controla a leitura do byte pelo canal ConfigLinkIn, verifica o validade do byte pela análise dos *start* bits e *stop* bits, envia o sinal para um registrador deslocar os bits para a direita e montar o byte e envia o sinal de reconhecimento (Ack) pelo canal ConfigLinkOut e 2) a máquina de estado que controla a montagem da mensagem de configuração, que desvia o byte montado para o decodificador de instruções ou decodificador de endereços 1 ou 2, em função da quantidade de bytes que formam a mensagem de configuração.

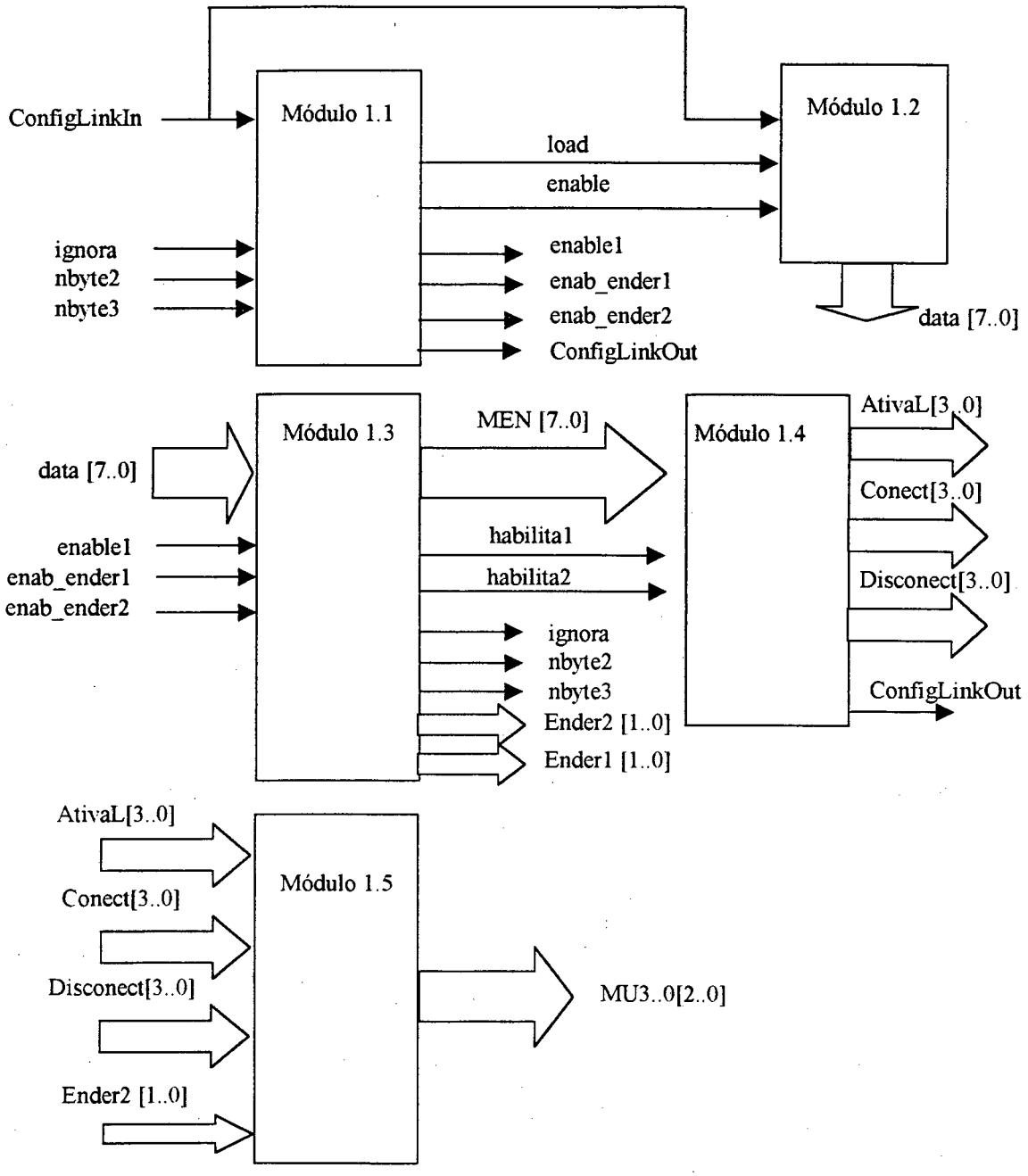


Figura 5.3 – Diagrama de blocos da lógica de controle

1. Controle da leitura do byte: esta máquina de estados inicia no estado *idle* onde é testado o canal de entrada ConfigLinkIn. Ao subir ao nível alto, é testado uma próxima entrada alta (*start* bit) no estado rec_start e passa a leitura do byte até o estado rec_data7. Logo em seguida é esperado um bit com nível baixo (*stop* bit) para confirmação do dado recebido. Caso não seja confirmado o dado, todo byte é ignorado e retorna ao estado idle. No estado data_ok, é enviado o reconhecimento do dado pelo canal ConfigLinkOut. O

estado *shoot* envia o sinal *trigger* alto para a próxima máquina de estado, indicando o fim de um pacote de dado recebido com sucesso. Esta máquina de estado apresenta os seguintes sinais de saída, em ordem da esquerda para a direita:

a) ClinkOut1: sinal de confirmação do byte recebido
b) *load* : sinal que habilita o deslocamento dos bits para a direita no registrador de deslocamento.
c) *enable*: descarrega o byte pronto para os decodificadores
d) *trigger*: dispara a leitura do próximo byte

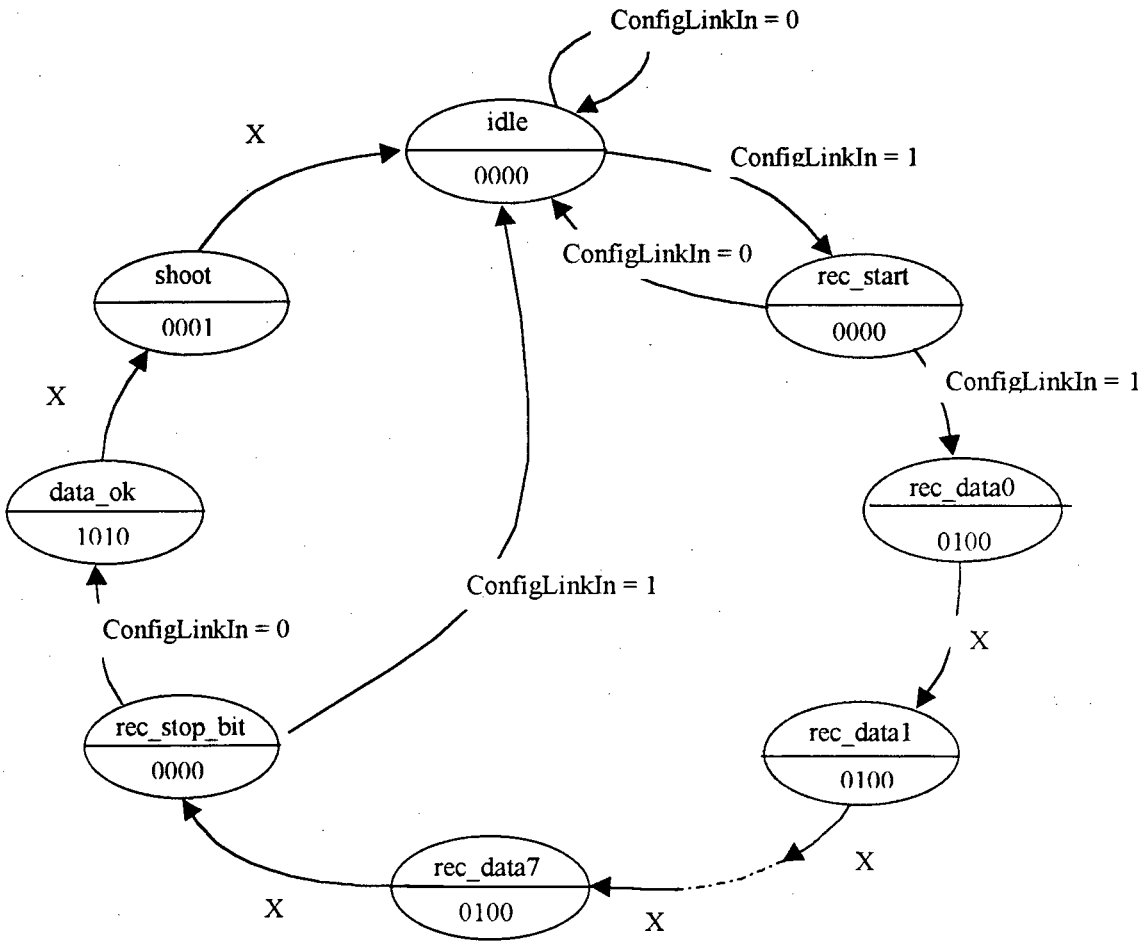


Figura 5.4 – Máquina de estados para controle da leitura do byte

2. Controle da mensagem de configuração: esta máquina de estado inicia no estado idle2 e é disparado também pelo nível alto no canal ConfigLinkIn. No estado rec_byte1 aguarda o sinal de *trigger* enviado pela máquina de estado 1 para passar para o estado wait_1 onde envia o sinal para o decodificador de instruções. No estado confirm1, espera os sinais nbyte2 e nbyte3 do decodificador, que indicam o tamanho da mensagem. Caso seja uma mensagem de apenas 1 byte ou então se o byte não for decodificado (ignora = 1), retorna ao estado idle2 para a leitura de uma próxima mensagem. Se a mensagem possuir dois byte (nbyte2 = 1 e nbyte3 = 0) é desviado para o estado rec_byte3 e se possuir três bytes (nbyte2 = 1 e nbyte3 = 1), é desviado para o estado rec_byte2.

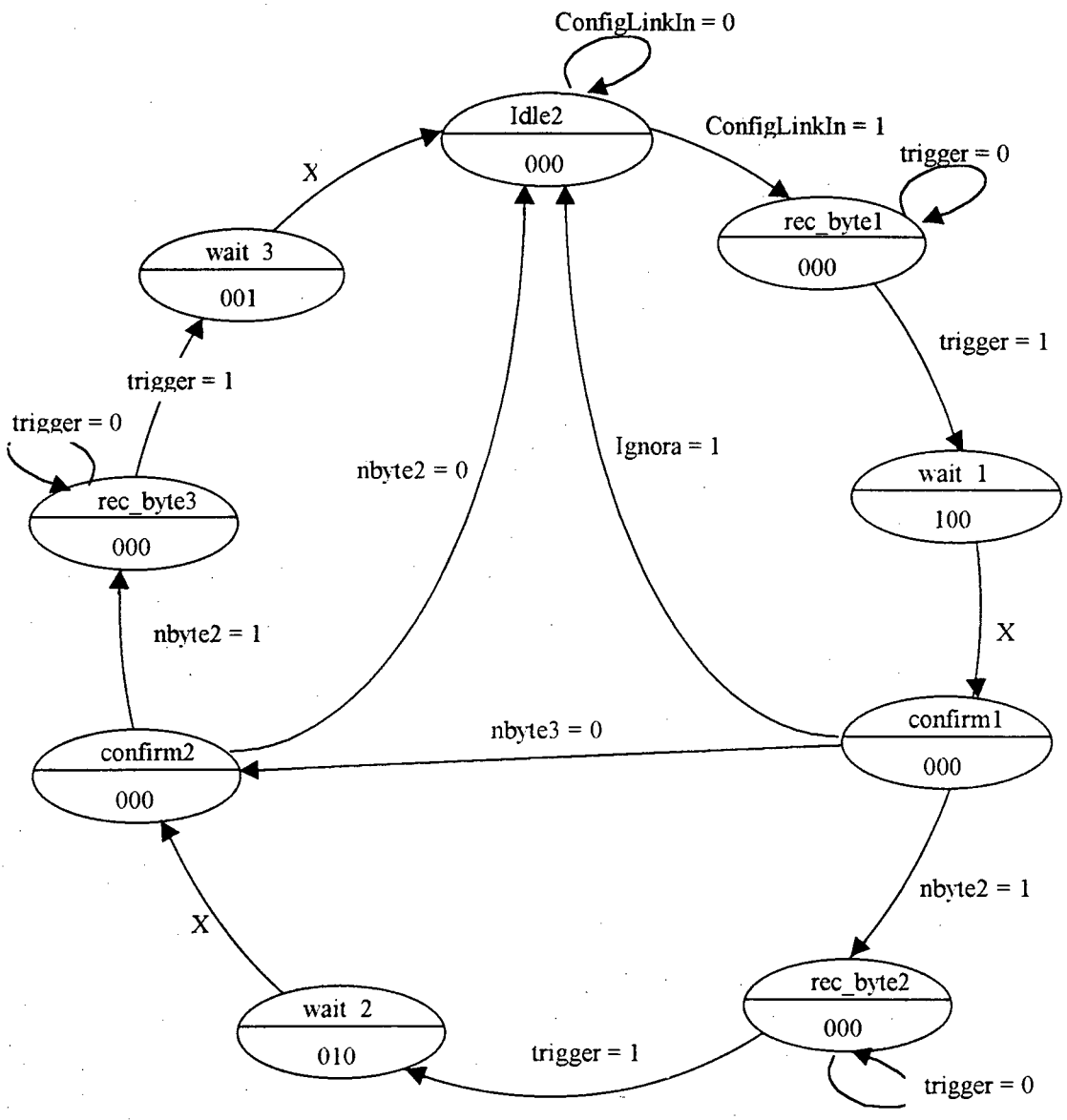


Figura 5.5 – Máquina de estados para controle do tamanho da mensagem de configuração

Esta máquina tem as seguintes saídas:

a) enable1: habilita o decodificador de instruções
b) enab_ender2 : habilita o decodificador de endereços de entrada
c) enab_ender1: habilita o decodificador de endereços de saída

5.2.1.2 – Módulo 1.2 – Registrador de deslocamento

O módulo 1.2 é formado por um registrador de deslocamento para a direita. O registrador de deslocamento tem como função receber os bits e desloca-los à direita, com o sinal alto de load, montando desta forma o byte. Com o sinal alto de *enable*, descarrega o byte para os respectivos decodificadores.

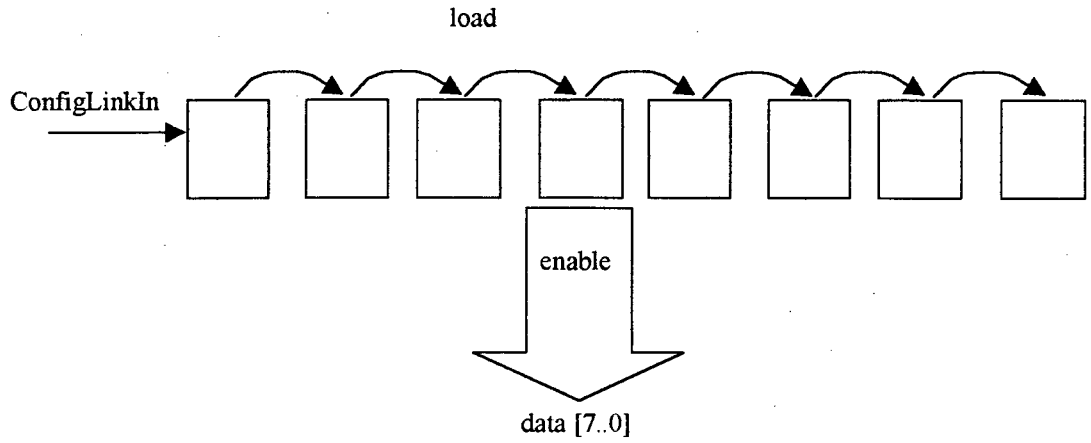


Figura 5.6 – Registrador de deslocamento

5.2.1.3 – Módulo 1.3 – Decodificadores

O módulo 1.3 é formado pelo conjunto dos três decodificadores: o decodificador de instruções, o decodificador de endereços de saída (ender1) e o decodificador de endereços de entrada (ender2), como é mostrado na Figura 5.7. O sinal enable1, desvia o byte montado no registrador de deslocamento para o decodificador de instruções, que gera os sinal nbyte2, nbyte3 e ignora e os sinais que identificam as mensagens de configuração (MEN[7..0]) que dispara as respectivas máquinas de estado. O sinal enab_ender1 desvia o byte para o decodificador de endereços do canal de saída, e gera o sinal habilita1 e o sinal enab_ender2, desvia o byte para o decodificador de endereços do canal de entrada e gera o sinal de habilita2.

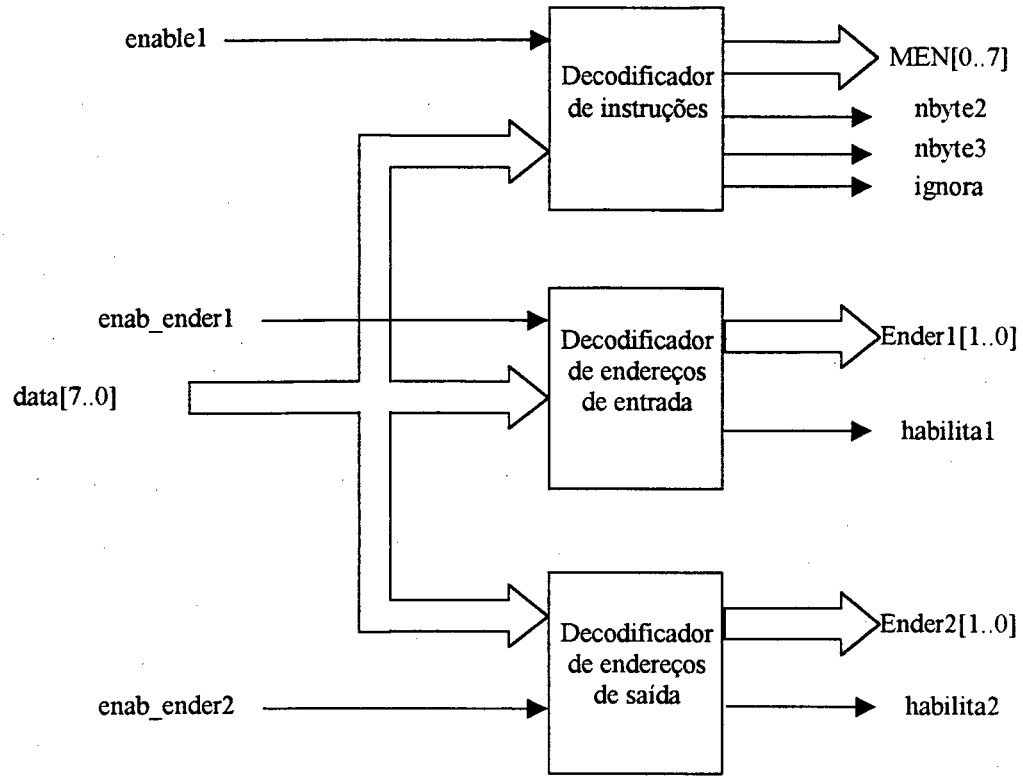


Figura 5.7 - Decodificadores

5.2.1.4 – Módulo 1.4 – Execução das instruções

O módulo 1.4 é formado por um conjunto de 5 máquinas de estados, ativadas com nível alto dos sinais MEN[7..0], que executam uma determinada instrução representada pela mensagem de configuração. Além dos sinais MEN[7..0], recebe do módulo 1.3 os sinais ender1 e ender2, habilita1 e habilita2. Gera os sinais responsáveis pela conexão e desconexão dos canais e leitura do conteúdo do *latch[3..0]* (mensagem 2).

1. máquina de estado para executar uma conexão - mensagem 0: inicia no estado idle0 e MEN0 alto ativa esta máquina de estado. No estado decodC1 espera o endereço de saída estar disponível (habilita1 = 1) e desvia o fluxo para setar somente os sinais relacionados ao endereço de saída.

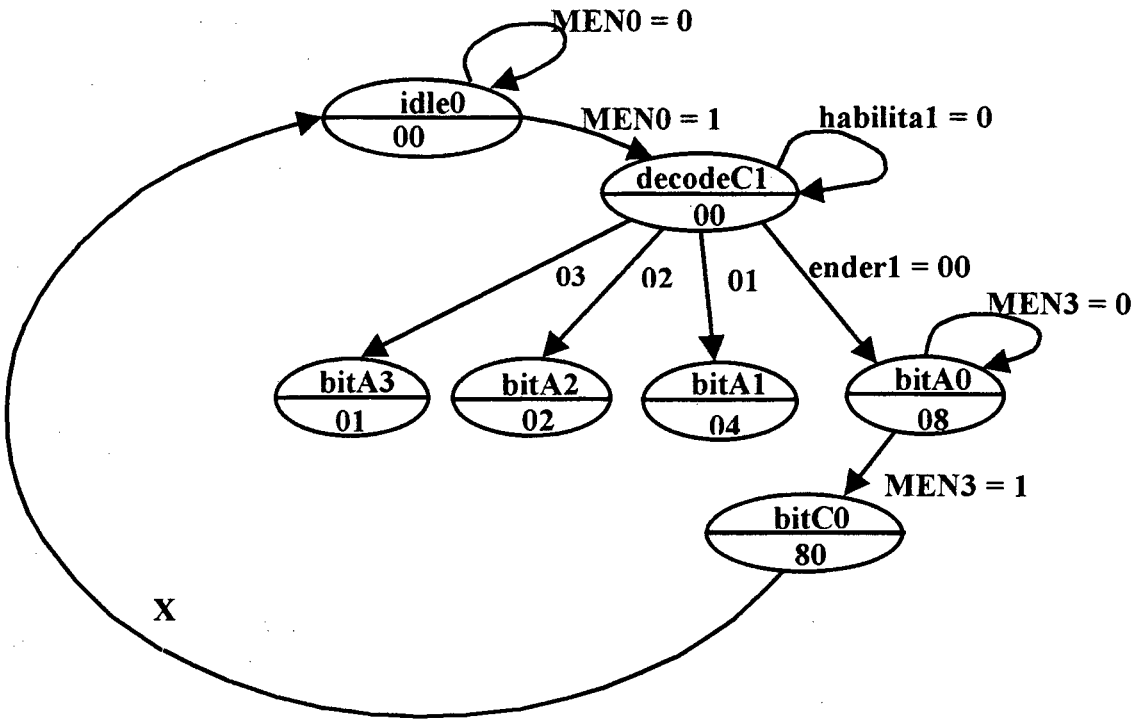


Figura 5.8 – Máquina de estado para gerar sinais de conexão. Os sinais de saída em cada estado estão representados em hexadecimal.

Esta máquina tem os seguintes sinais de saídas:

a) ConectC[0..3] : sinal que estabelece a conexão

b) AtivaLC[0..3]: habilita a copia do endereço de saída no *latch[3..0]*.

2. máquina de estado para exibir o *status* de uma saída – mensagem 2: esta máquina de estado inicia no estado idler e espera o sinal alto de MEN2, passando para o estado wait_ender1 que espera o endereço da saída estar disponível para carregar o dado contido no *latch* no registrador. Os próximos 12 estados deslocam o dado no registrador para a saída ConfigLinkOut bit a bit, iniciando pelos dois bits altos de *start* bit, o dado finalizando com o *stop* bit. O estado endM2 exigi que o sinal de MEN2 se torne baixo antes de iniciar uma nova consulta. Desta forma, não é possível fazer-se duas consultas seguidas. O bit mais significativo do byte representa a conexão (bit alto) ou uma desconexão (bit baixo) do canal em questão. Na Figura 5.8 mostramos o diagrama de transição de estados para a mensagem de configuração 2. Temos os seguintes sinais de saída:

a) loadr : carrega o registrador com os bit de start bit, stop bit e copia dado do *latch*.

b) discharge: descarrega o dado armazenado no registrador bit a bit para o canal ConfigLinkOut, deslocando os dados para esquerda.

c) sel: sinal para habilitar a saída do canal ConfigLinkOut para a saída do registrador.

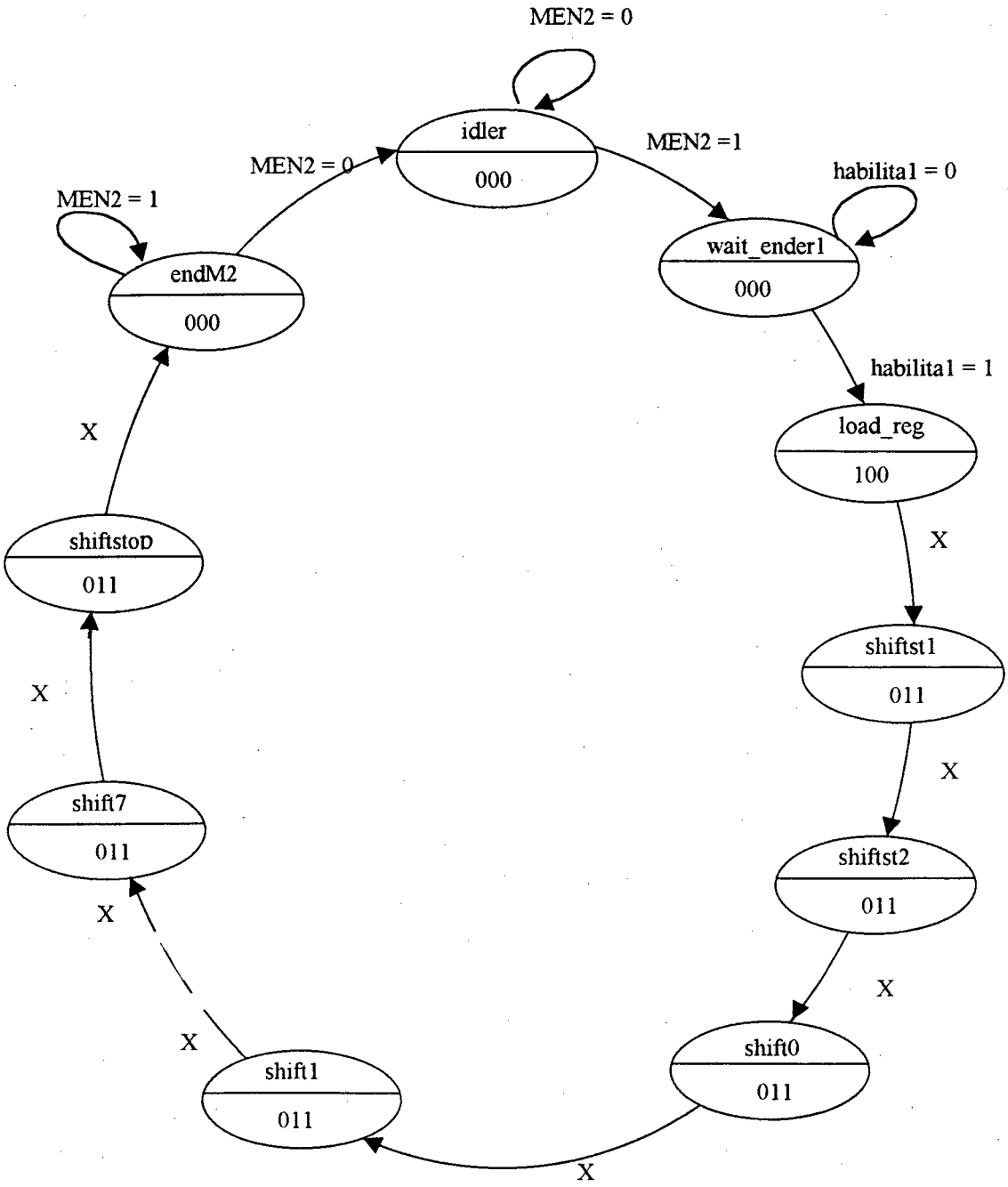


Figura 5.9 – Máquina de estado para a verificação do *status* de uma saída

3. máquina de estados para a desconexão de uma saída - mensagem 5: esta máquina (Figura 5.10) inicia no estado idle5 e é ativada com sinal alto em MEN5. Passa para o estado decodeD1 que espera o endereço de saída estar disponível (habilita1 = 1), decodifica o endereço e desvia o fluxo para setar somente os sinais de desconexão relacionadas ao endereço de saída decodificado. Temos as seguintes saídas:

a) disconect5 [3..0]: sinal de desconexão enviado para o *latch* endereçado.

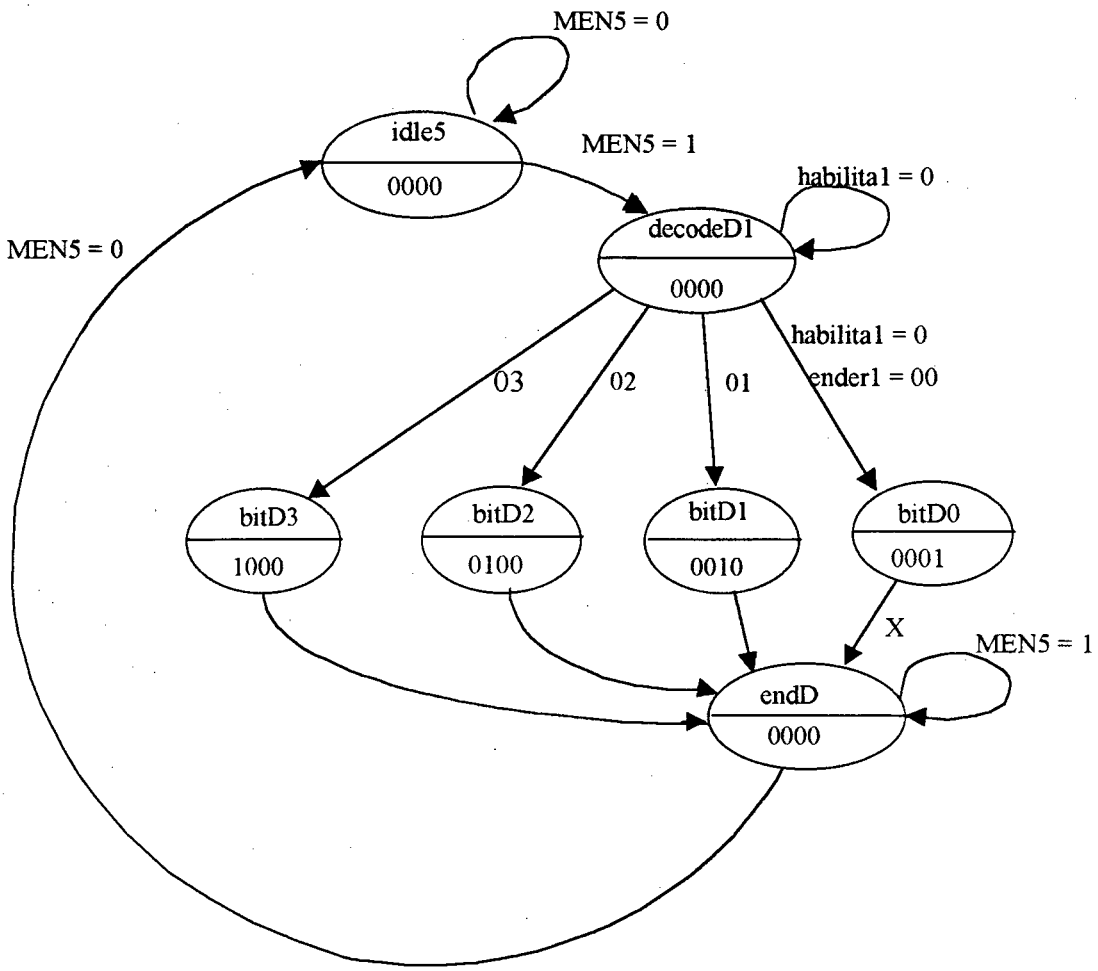


Figura 5.10 – Máquina para desconexão de uma saída endereçada

4. Máquina para desconectar um canal de entrada de um canal de saída e o canal de saída da primeira do canal de entrada da segunda – mensagem 6: esta máquina necessita de dois estados adicionais em relação a máquina anterior para decodificação dos dois endereços de saída – decodeD2 e decodeD1, que aguardam a disponibilização dos endereços – habilita2 e habilita1. Temos as seguintes saídas:

a) disconect6 [3..0]: sinal de desconexão enviado para o *latch* endereçado. Em cada ciclo desta máquina é enviado dois sinais de desconexão.

b) sel6: sinal que habilita as linhas de desconexão para esta máquina.

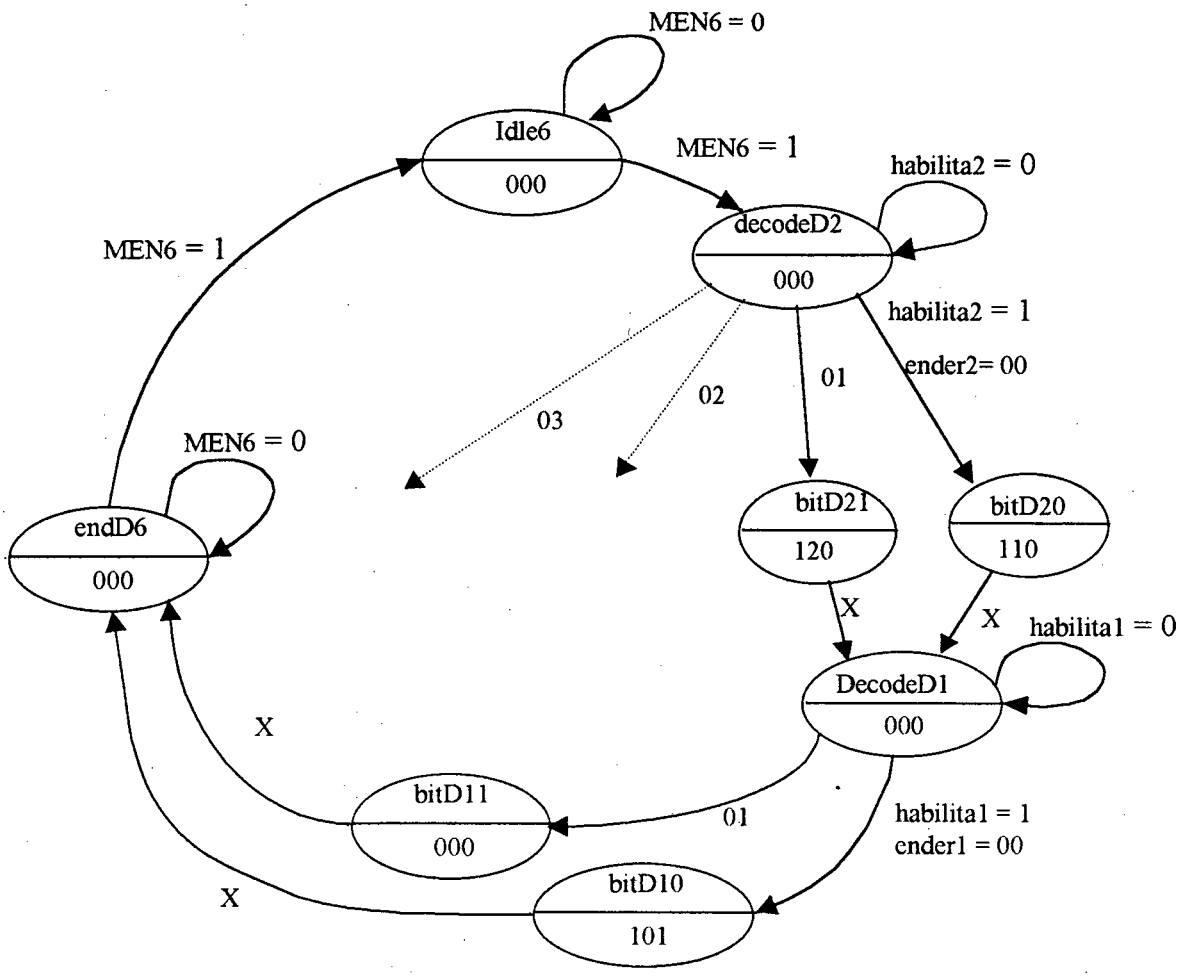


Figura 5.11 – Máquina de estado para desconexão de duas saídas

5. Máquina para conectar uma entrada a todas as saídas (*broadcasting*) – mensagem 7: esta máquina inicia no estado idle7 e é ativada com sinal alto em MEN7. Aguarda no estado bitBA o endereço da entrada estar disponível onde envia os sinais para cópia do endereço de entrada nos *latch* e no estado bitBC envia o sinal de conexão para todas as saídas. Temos as seguintes saídas:

a) AtivaLB[3..0] : habilita a cópia do endereço de entrada em todos os latchs

b) ConectB[3..0] : sinal de conexão para todos os canais de saída.

c) Sel_conect: sinal que habilita as linhas de conexão para esta máquina.

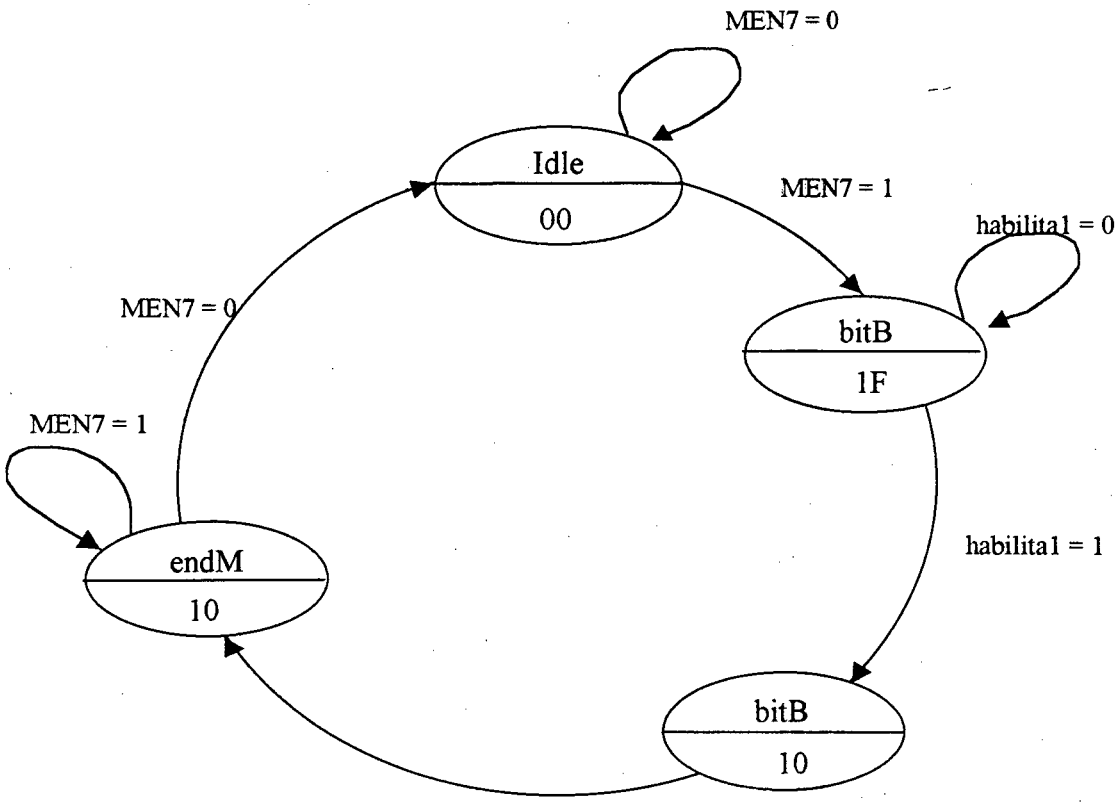


Figura 5.12 – Máquina de estado para gerar sinais de conexão de uma entrada em todas as saídas. Os sinais de saída em cada estado estão representados em hexadecimal.

5.2.1.5 – Módulo 1.5 – Latchs

Este módulo é composto por um conjunto de *latchs* que estão associados aos multiplexadores de saída. Tem como entrada os sinais AtivaL[3..0], sinal que habilita a cópia do outro sinal de entrada, ender2 [1..0], no *latch* , Conect[3..0] que habilita a conexão do canal de entrada com o canal de saída e Disconect [3..0] que habilita a desconexão do canal de saída.

5.2.2 – Módulo 2 – Multiplexadores

O módulo 2 é constituído por um conjunto de multiplexadores, que através das linhas MU, seleciona qual canal de entrada aparecerá no canal de saída. A linha MU(2) = 1 habilita a conexão e MU(2) = 0 desabilita a conexão.

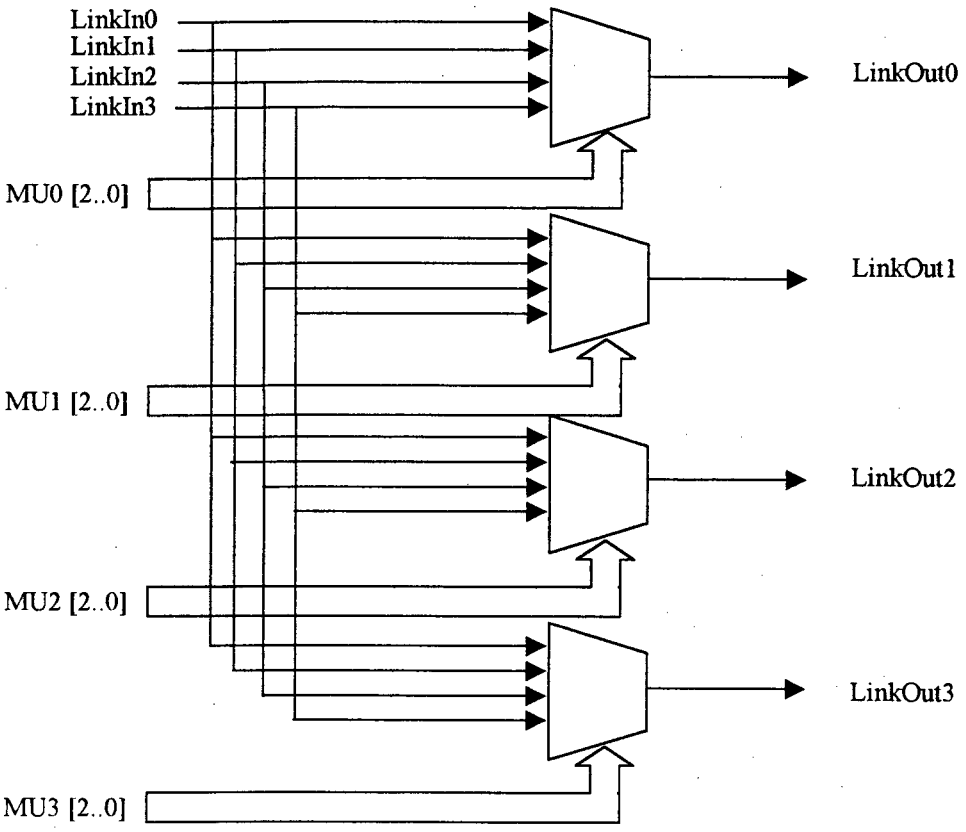


Figura 5.13 – Circuito Combinacional do módulo2

5.3 - RESUMO

Nesse capítulo foi apresentado o projeto de hardware do crossbar com as características funcionais do C004 da INMOS. O projeto foi dividido em módulos para maior visualização e entendimento dos vários componentes de hardware que compõe o projeto. Foi apresentado um detalhamento de cada módulo, as suas entradas, as máquinas de estado que controlam e geram um conjunto de sinais e suas saídas. A implementação em VHDL de cada módulo se encontra no Anexo I.

CAPÍTULO 6 – SIMULAÇÃO DO PROJETO LÓGICO

INTRODUÇÃO

Este capítulo apresenta os resultados obtidos da implementação em VHDL do *crossbar*, através da simulação de um conjunto de sinais de entrada e análise dos sinais de saída. A interface gráfica do ambiente permite que os sinais de entrada e sinais de saída sejam visualizados através de formas de ondas. As simulações foram realizadas no ambiente de desenvolvimento MAX+PLUS II da Altera – nas versões 7.21 e 8.2. A implementação em VHDL é apresentada no Anexo I.

As simulações buscam verificar e validar o projeto lógico. Desta forma, é introduzido um conjunto de dados pelo canal ConfigLinkIn, que corresponde a uma mensagem de configuração válida apresentada na Tabela 5.1, e um conjunto de dados, que correspondem a um múltiplo do sinal de clock, nos canais de entrada LinkIn[3..0] e verifica-se a resposta nos canais de saída LinkOut[3..0] e ConfigLinkOut. A seguir são apresentados os diagramas de tempo das seguintes simulações:

1. Conexão do canal de entrada 3 com o canal de saída 1 e o canal de entrada 1 com canal de saída 2.
2. Verificação do status dos canais de saída 1 e 3
3. Desconexão do canal de saída 1
4. Reinicialização de um conjunto de conexões
5. *Broadcasting* – conexão do canal de entrada 3 com todas as saídas

6.1 – ESTABELECIMENTO DE UMA CONEXÃO

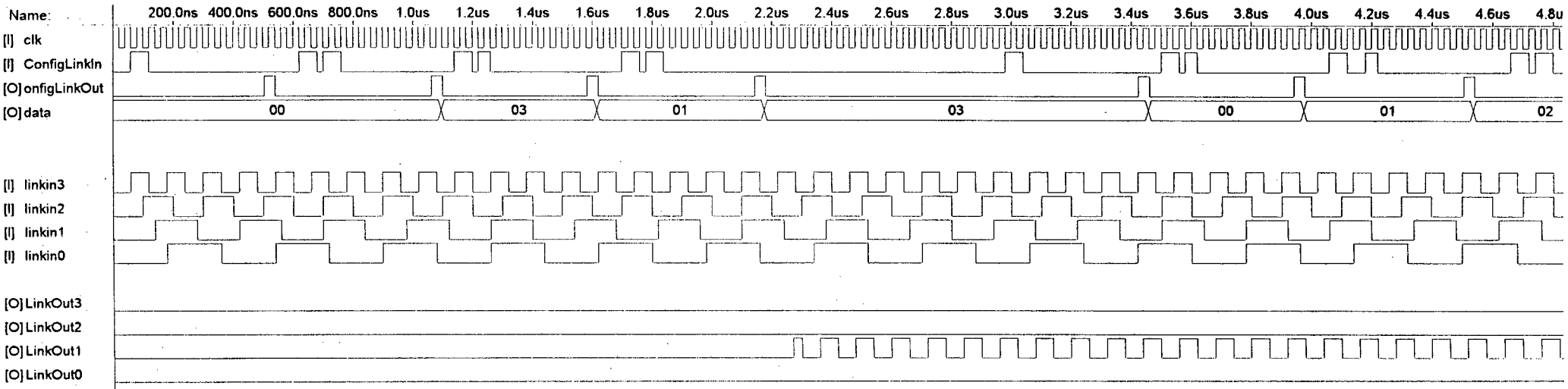
Para o estabelecimento de uma conexão, é necessário seguir o procedimento descrito abaixo. É realizada a conexão do canal LinkIn3 com LinkOut1 e o canal LinkIn1 com LinkOut2:

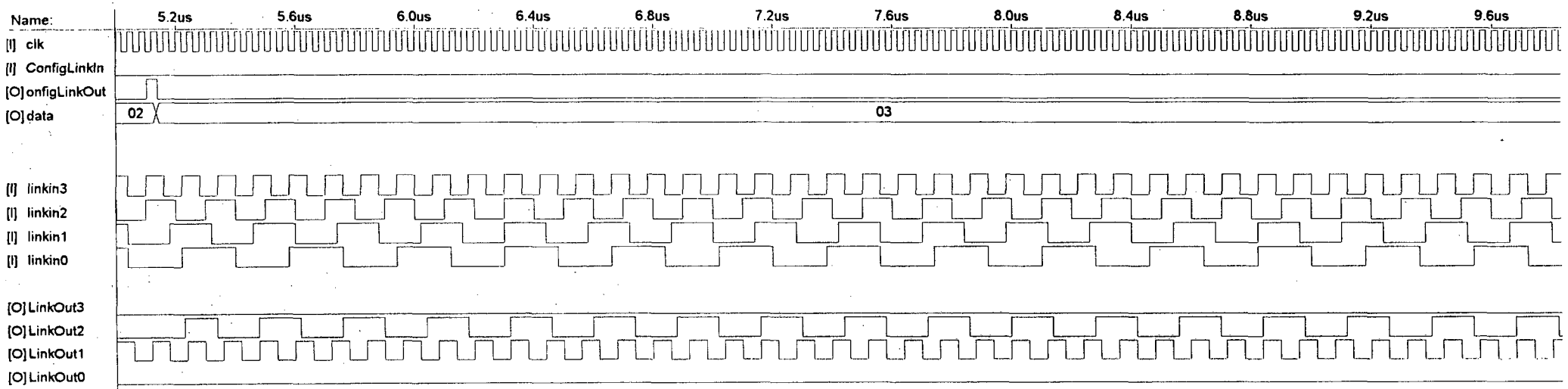
1. Envia-se pelo canal de entrada ConfigLinkIn a mensagem 0, ou seja, o conjunto de bits que representam a mensagem 0. Logo após dois bits alto (*start bit*), Envia-se um conjunto de 8 sinais baixo (mensagem 0) e mais um bit baixo (*stop bit*), seguindo o protocolo de comunicação INMOS.
2. De 0,7 us a 1,2 us, envia-se pelo canal ConfigLinkIn o endereço do LinkIn3.
3. De 1,4 us a 1,9 us, envia-se pelo canal ConfigLinkIn o endereço do LinkOut1.
4. Envia-se pelo canal ConfigLinkIn a mensagem 3, que estabelece a conexão.
5. Iniciar uma nova conexão seguindo o mesmo procedimento a partir do passo 1, trocando-se o endereços dos *links* de entrada e saída.

É verificada uma latência no diagrama das paginas 71 e 72 de 125.5 ns, após o reconhecimento da mensagem 3, para o estabelecimento da conexão.

Nos diagramas apresentados, observa-se um atraso de 18.7 ns entre os sinais aplicados nos canais de entrada LinkIn e os sinais de resposta nos canais de saída LinkOut, o que está de acordo com o atraso especificado no manual do C004. A defasagem ocorre devido ao atraso de propagação dentro do crossbar provocado pelos pinos de I/O (E/S) e à lógica dos multiplexadores.

Nos diagramas que seguem foi mantido um sinal que visualiza o dado em ConfigLinkIn

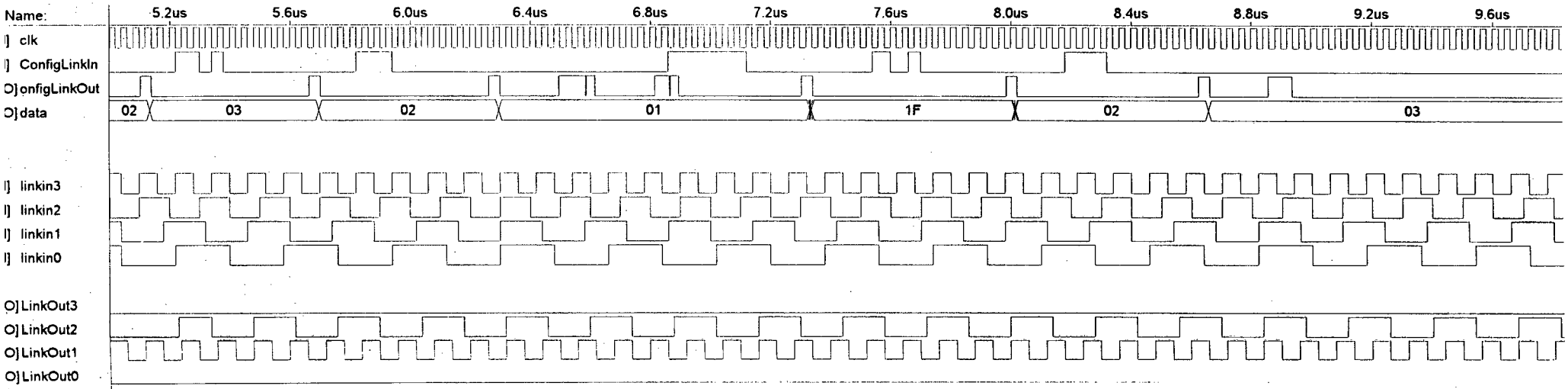

Name:
200.0ns 400.0ns 600.0ns 800.0ns 1.0us 1.2us 1.4us 1.6us 1.8us 2.0us 2.2us 2.4us 2.6us 2.8us 3.0us 3.2us 3.4us 3.6us 3.8us 4.0us 4.2us 4.4us 4.6us 4.8u
[I] clk
[I] ConfigLinkIn
[O] onfigLinkOut
[O] data
00
03
01
03
00
01
02
[I] linkin3
[I] linkin2
[I] linkin1
[I] linkin0
[O] LinkOut3
[O] LinkOut2
[O] LinkOut1
[O] LinkOut0


Name:
5.2us
5.6us
6.0us
6.4us
6.8us
7.2us
7.6us
8.0us
8.4us
8.8us
9.2us
9.6us
[I] clk
[I] ConfigLinkIn
[O] onfigLinkOut
[O] data
02
03
[I] linkin3
[I] linkin2
[I] linkin1
[I] linkin0
[O] LinkOut3
[O] LinkOut2
[O] LinkOut1
[O] LinkOut0

6.2 – *STATUS* DE UM CANAL DE SAÍDA

O procedimento descrito abaixo permite a verificação do *status* dos canais de saída 1 e 3:

1. Envia-se pelo canal ConfigLinkIn, a mensagem de configuração 2.
2. Envia-se pelo canal ConfigLinkIn, o endereço do LinkOut1.
3. Obtém-se a resposta bit a bit pelo canal serial ConfigLinkOut.
4. Para verificação do status de um outro canal de saída envia-se pelo canal ConfigLinkIn um dado qualquer, como é apresentado no diagrama da página 74, antes de enviar pelo mesmo canal a mensagem 2.
5. Inicia-se o procedimento a partir do passo 1 enviando o endereço do LinkOut3.

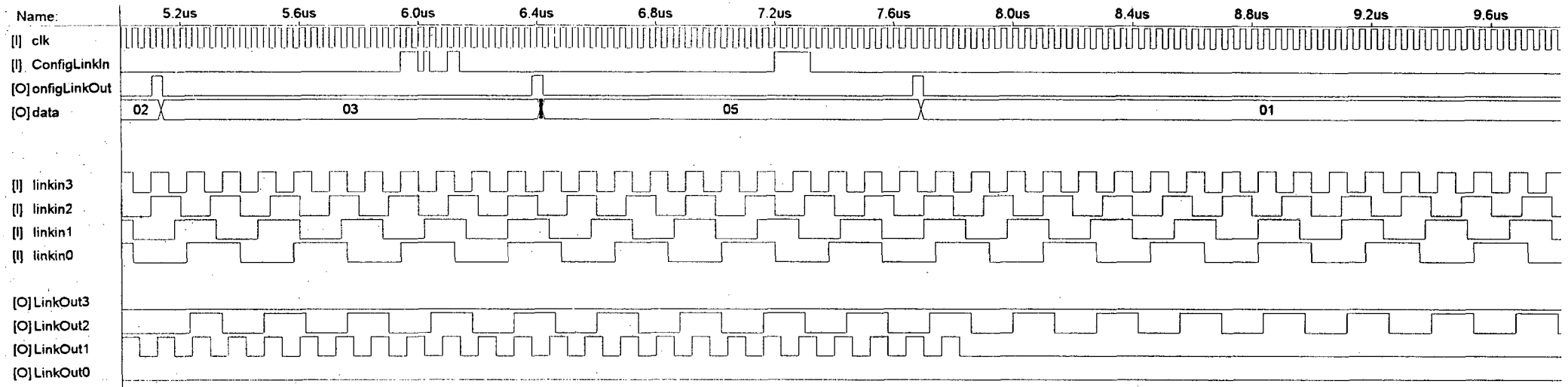

Name:
5.2us
5.6us
6.0us
6.4us
6.8us
7.2us
7.6us
8.0us
8.4us
8.8us
9.2us
9.6us
clk
ConfigLinkIn
onfigLinkOut
data
02
03
02
01
1F
02
03
linkin3
linkin2
linkin1
linkin0
LinkOut3
LinkOut2
LinkOut1
LinkOut0

6.3 - DESCONEXÃO

Para desconexão do canal de saída 1 (LinkOut1) segue-se o procedimento abaixo:

1. Envia-se pelo canal ConfigLinkIn, a mensagem de configuração 5
2. Envia-se pelo canal ConfigLinkIn, o endereço do LinkOut1.

Verifica-se pelo diagrama da página 76 que após uma latência de 154.6 ns em relação ao reconhecimento do endereço, é realizada a desconexão.

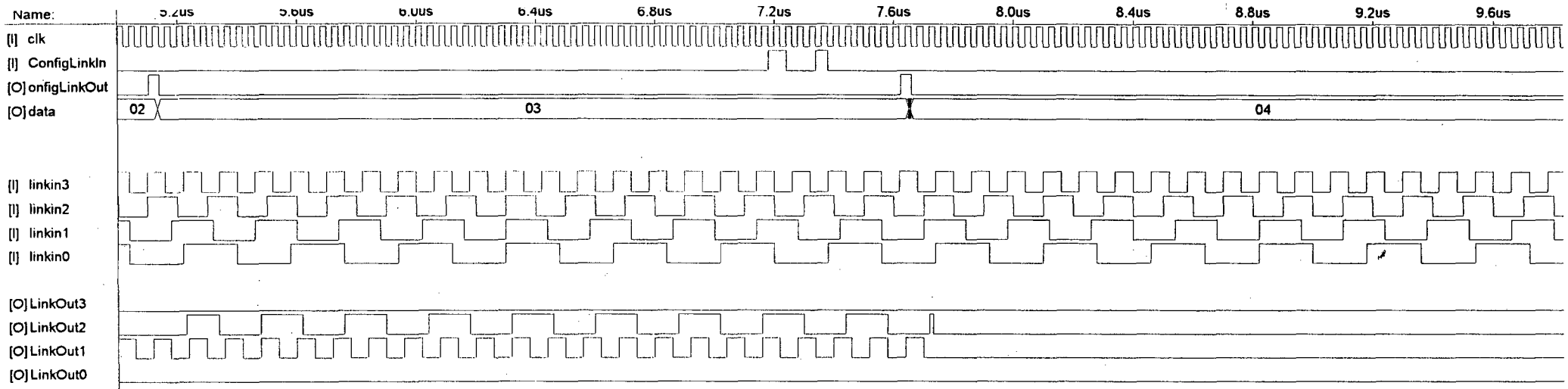

Name:
5.2us
5.6us
6.0us
6.4us
6.8us
7.2us
7.6us
8.0us
8.4us
8.8us
9.2us
9.6us
[I] clk
[I] ConfigLinkIn
[O] onfigLinkOut
[O] data
02
03
05
01
[I] linkin3
[I] linkin2
[I] linkin1
[I] linkin0
[O] LinkOut3
[O] LinkOut2
[O] LinkOut1
[O] LinkOut0

6.4 – *RESET* DAS SAÍDAS

Para reinicializar todas as saídas é necessário seguir o procedimento abaixo:

1. Envia-se pelo canal ConfigLinkIn, a mensagem de configuração 4.

Verifica-se pelo diagrama da pagina 78, que após um atraso de 109.0 ns, todas as saídas são desconectadas e mantidas em nível baixo.

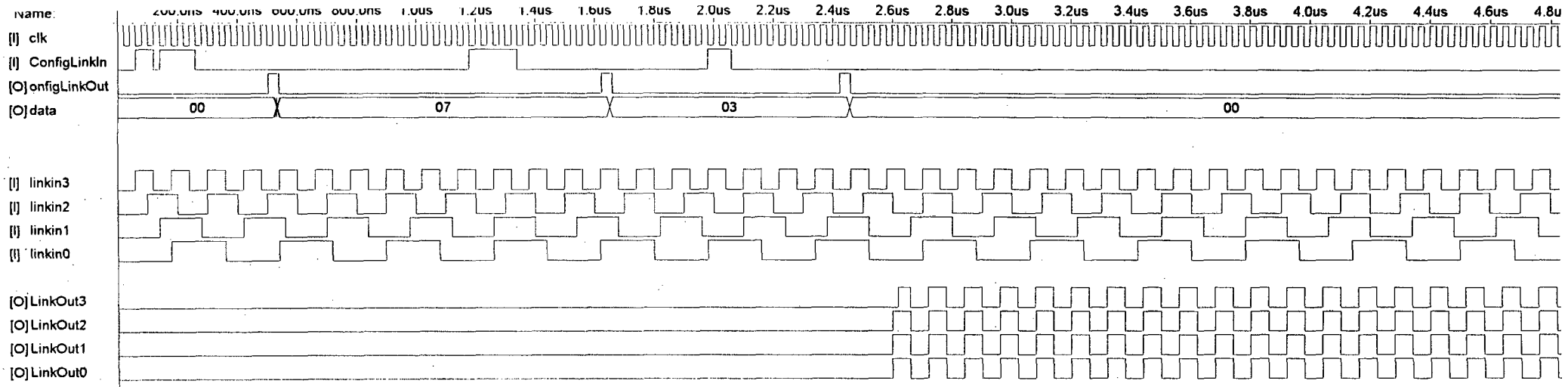

Name:
5.2us
5.6us
6.0us
6.4us
6.8us
7.2us
7.6us
8.0us
8.4us
8.8us
9.2us
9.6us
[I] clk
[I] ConfigLinkIn
[O] onfigLinkOut
[O] data
02
03
04
[I] linkin3
[I] linkin2
[I] linkin1
[I] linkin0
[O] LinkOut3
[O] LinkOut2
[O] LinkOut1
[O] LinkOut0

6.5 - *BROADCASTING*

No diagrama da pagina 80, é apresentado os sinais para *broadcasting*. Para conexão do canal de entrada LinkIn3 com todos os canais de saída é necessário seguir o procedimento abaixo:

1. Envia-se pelo canal ConfigLinkIn a mensagem 7 .
2. Envia-se pelo canal ConfigLinkIn o endereço do canal de entrada (LinkIn3).
3. Envia-se mais um byte para completar o tamanho da mensagem pois é uma mensagem de três bytes.

Após 192,6 ns do reconhecimento do segundo byte é realizada a conexão, como verificamos no diagrama da página 80.


Name
[I] clk
[I] ConfigLinkIn
[O] onfigLinkOut
[O] data
00
07
03
00
[I] linkin3
[I] linkin2
[I] linkin1
[I] linkin0
[O] LinkOut3
[O] LinkOut2
[O] LinkOut1
[O] LinkOut0
1.0us
1.2us
1.4us
1.6us
1.8us
2.0us
2.2us
2.4us
2.6us
2.8us
3.0us
3.2us
3.4us
3.6us
3.8us
4.0us
4.2us
4.4us
4.6us
4.8u

CONCLUSÃO

Nesse trabalho foi apresentado o projeto de uma rede de interconexão do tipo *crossbar* com as mesmas características funcionais do *crossbar* C004 da INMOS, específica e customizada para o multicomputador do projeto Nó//.

No primeiro capítulo foi realizada uma revisão um estudo dos principais conceitos e termos relacionadas com o paralelismo, modelos de arquitetura paralela e seqüencial e classificação dos vários modelos de arquitetura propostos.

No segundo capítulo foi realizado um estudo sobre as redes de interconexão, modelos de interconexão, importância em um sistema de comunicação de uma máquina paralela e primícias do projeto de uma rede de interconexão.

O terceiro capítulo descreveu o ambiente do Multicomputador No//, os modelos de arquitetura propostos e componentes definidos para a implementação.

A seguir, no quarto capítulo, apresentou-se as metodologias que foram utilizadas no transcorrer do desenvolvimento do projeto do *crossbar*, dispositivos lógicos programáveis (FPGA), linguagem de descrição de hardware – VHDL utilizada para a implementação lógica do projeto, ambiente de programação e simulação – MAX+PLUS II e a modelagem em máquina de estados.

No quinto capítulo foi apresentado o projeto lógico do *crossbar*, o diagrama de blocos e as máquinas de estados utilizadas no projeto.

No sexto capítulo foi apresentado os resultados obtidos através da simulação no MAX+PLUS II de cada mensagem de configuração.

A integração dos módulos no projeto lógico, foi realizado no editor gráfico do MAX+PLUS II, porém os resultados permitem validar o modelo de *hardware* proposto, restando a implementação dos módulos repetitivos, como os multiplexadores e *latch*, na forma de pacotes (*package*) e a implementação hierárquica do projeto em VHDL. Não foi implementado também a mensagem de configuração 1, mas pode-se obter as conexões cruzadas utilizando a mensagem 0, como pode ser verificado na simulação do item 6.1.

Os atraso de propagação obtidos estão conforme os atrasos especificados no manual do crossbar da INMOS.

Do ponto de vista do *crossbar*, foi possível implementar a mensagem 7 – *broadcasting* – porém do ponto de vista do multicomputador, será necessário resolver o

problema do pacote de reconhecimento (ack) que deve ser enviado pelos vários nós que estarão recebendo ao mesmo tempo os pacotes de dados. Há duas propostas, criar uma lógica adicional para recebimento e contagem dos vários sinais de reconhecimento ou criar uma lógica para aceitar somente o sinal de reconhecimento do dispositivo mais lento.

Outra proposta de implementação é uma instrução de *multicasting*, que conecta um canal de entrada em um determinado número de canais de saída.

Foi realizado um estudo sobre a taxa de ocupação dos componentes internos de um dispositivo ALTERA da família FLEX 10 K, para uma chave com 32 entradas e 32. Obteve-se uma taxa de ocupação de 4 % de células lógicas, o que viabiliza a implementação deste projeto também para as necessidades do Multicomputador Nó //. Como não foi realizado o mesmo estudo para outros dispositivos de outras famílias ALTERA, é sugerido como pesquisa para futuros trabalhos.

REFERÊNCIAS BIBLIOGRÁFICAS

1. ALVES, Valéria A., **Multicomputador Nó//: Implementação de Primitivas Básicas de Comunicação e Avaliação de Desempenho**., Dissertação de Mestrado, Curso de Pos-Graduação em Ciências de Computação, Universidade Federal de Santa Catarina, Florianópolis, SC, 1996.

2. BRETON, Philippe., **Histoire de L'insformatique**., Tradução: Elcio Fernandes. Editora Universidade Estadual Paulista, São Paulo, 1991.

3. CAMPOS, Rodrigo A. **Um Sistema Operacional Fundamentado no Modelo Cliente-Servidor e um Simulador Multiprogramado para Multicomputador**., Dissertação de Mestrado, Curso de Pos-Graduação em Ciências de Computação, Universidade Federal de Santa Catarina, Florianópolis, SC, 1995.

4. COLI, Vicent J., **Introduction to Programmable Array Logic.**, Byte, Jan, 1987.

5. CORSO, Thadeu B., **Ambiente para Programação Paralela em Multicomputador.**, Florianópolis: Relatório Técnico, UFSC-CTC-INE, n.° 1, Nov, 1993.

6. DECEGAMA, Angel L., **Technology of Parallel Processing : Parallel Processing Architectures and VLSI *Hardware.***, Prentice Hall, 1989.

7. DUNCAN, Ralph., **A Survey on Parallel Computer Architectures**, in: IEEE Computer., Feb. 1990.

8. FENG, T., **A Survey Of Interconnection Network.,** in: IEEE Computer, Dec. 1981.

9. HANSEN, Per Brich., **Studies In Computation Science – Parallel Programming Paradigms.,** Prentice Hall, New Jersey, 1995.

10. HWANG, Kai., **Advanced Computer Architecture**., 2. Ed. New York: Mcgraw Hill, 1987.

11. HWANG, Kai & BRIGGS, Fayé A., **Computer Architecture and Parallel Processing**., New York: Mcgraw Hill, 1985.

12. HWANG, Kai & DEGROOT, Douglas., **Parallel Processing for Supercomputers & Artificial Intelligence**., New York: Mcgraw Hill, 1989.

13. INMOS. IMS C011: Link Adaptor., In: **INMOS Engineering Data.**, 1988.

14. INMOS. IMS C004: Programable Link Switch., In: **INMOS Engineering Data.**, 1988.

15. MENDONÇA, Alexandre & ZELENOVSKI, Ricardo & PINHO, André & ALVARES, Marco., **FPGA: uma Nova Alternativa para a Criação de Protótipos.**, Developers, Jan, 1998.

16. MERKLE, Carla., **Ambiente para Execução de Programas Paralelos Escritos na Linguagem SuperPascal em um Multicomputador com Rede de Interconexão Dinâmica.**, Dissertação de Mestrado, Curso de Pos-Graduação em Ciências de Computação, Universidade Federal de Santa Catarina, Florianópolis, SC, 1996.

17. MONTEZ, Carlos B., **Um Sistema Operacional com Micronúcleo distribuído e um Simulador Multiprogramado de Multicomputador.**, Dissertação de Mestrado, Curso de Pos-Graduação em Ciências de Computação, Universidade Federal de Santa Catarina, Florianópolis, SC, 1995.

18. PERROT, Ronald H., **Parallel Programing.**, Great Britain: Addilson-Wesley, 1987

19. REED, D. A. & FUJIMOTO, R. M., **Multicomputer Network: Message-Based Parallel Processing.**, MIT Press, 1978.

20. ROSE, Jonathan & EL GAMAL, Abbas & SANGIOVANNI-VINCENTELLI, Alberto., **Architecture of Field-Programmable Gate Array.**, Proceeding of the IEEE, Vol. 81, Num. 07, Jul, 1993.

21. RUGGIERO, Carlos Antônio., **Arquitetura não Convencionais.**, curso ministrado no IV Simpósio Brasileiro de Arquiteturas de Computadores - Processamento de Alto Desempenho, São Paulo, 1992.

22. STALLINGS, William., **Computer Organization and Architecture: principles of structures and function**., 3 ed. New York: Macmillan, 1993.

23. ZEFERINO, Cesar A., **Projeto do Sistema de Comunicação de um Multicomputador**., Dissertação de Mestrado, Curso de Pos-Graduação em Ciências de Computação, Universidade Federal de Santa Catarina, Florianópolis, SC, 1996.

ANEXO I

Este anexo contém as implementações em VHDL dos elementos que compõe o crossbar descrito nesse trabalho.

Utilizando a simbologia VHDL, é apresentado as seguintes entidades:

1. broadc – máquina que gera os sinais para *broadcasting*
2. disc6 – máquina para desconexão de duas saídas
3. disc – processos que geram sinais de desconexão e reset
4. mesage2 – máquina que gera sinais para leitura do status de uma saída
5. conect – máquina que era os sinais para conexão entre dois canais
6. muxr – multiplexador auxiliar para selecionar latch a ser lido na mensagem 2
7. muxD – multiplexador auxiliar para selecionar tipo de desconexão
8. muxc – multiplexador para saída do ConfigLinkOut
9. muxBC – multiplexador para selecionar conexão *broadcasting*
10. reg_m2 – registrador de deslocamento para a esquerda
11. mux0 – multiplexador do canal de saída
12. latch0 – latch associado aos multiplexadores de saída
13. deco2 – decodificador de instruções
14. read_me2 – máquina que controla a leitura de toda a mensagem de configuração
15. read_by – máquina que controla a leitura e montagem do byte
16. d_ender1 – decodificador de endereços de saída
17. d_ender2 – decodificador de endereços de entrada
18. reg_des – registrador de deslocamento

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- ************************************************************
-- máquina de estados para conectar uma entrada em todas as saída
-- (broadcasting)
-- OBS. é uma mensagem de três bytes, onde o terceiro byte é um
-- byte nulo
-- ************************************************************

ENTITY broadc IS
    PORT(clk                :       in std_logic;
         MEN7               :       in std_logic;
         habilita1          :       in std_logic;
         AtivaLB0           :       out std_logic;
         AtivaLB1           :       out std_logic;
         AtivaLB2           :       out std_logic;
         AtivaLB3           :       out std_logic;
         conectB0           :       out std_logic;
         conectB1           :       out std_logic;
         conectB2           :       out std_logic;
         conectB3           :       out std_logic;
         Sel_conect         :       out std_logic);

END broadc;

ARCHITECTURE broadc OF broadc IS

    Type STATE_TYPE is (idle7,bitBA,bitBC,endM7);
                        -- estados para gerar os sinais de conexao -
                                                    --       broadcasting

    Signal state            :   STATE_TYPE; -- estados para mensagem 7

BEGIN

estado_men7     :   Process (clk)
                    begin
                        if (clk'event and clk = '1' )
                        then
                            CASE state IS
                            WHEN idle7 =>
                                IF MEN7 = '1'
                                THEN state <= bitBA;
                                ELSE state <=   idle7;
                                END IF;
                            WHEN bitBA =>
                                if habilita1 = '1'
                                then state <= bitBC;
                                else state <= bitBA;
                                end if;
                            WHEN bitBC =>
                                state <= endM7;
                            WHEN endM7 =>
                                if MEN7 = '0'
                                then state <= idle7;
                                else state <= endM7;
                                end if;
                            END CASE;
                        END IF;
                    END PROCESS estado_men7;
```

```
saidas_men7     :   Process(state)
                    BEGIN
                        CASE state IS
                        WHEN idle7 =>
                            conectB0   <= '0';
                            conectB1   <= '0';
                            conectB2   <= '0';
                            conectB3   <= '0';
                            AtivaLB0   <= '0';
                            AtivaLB1   <= '0';
                            AtivaLB2   <= '0';
                            AtivaLB3   <= '0';
                            Sel_conect <= '0';
                        WHEN bitBA =>
                            conectB0   <= '0';
                            conectB1   <= '0';
                            conectB2   <= '0';
                            conectB3   <= '0';
                            AtivaLB0   <= '1';
                            AtivaLB1   <= '1';
                            AtivaLB2   <= '1';
                            AtivaLB3   <= '1';
                            Sel_conect <= '1';
                        WHEN  bitBC =>
                            conectB0   <= '1';
                            conectB1   <= '1';
                            conectB2   <= '1';
                            conectB3   <= '1';
                            AtivaLB0   <= '0';
                            AtivaLB1   <= '0';
                            AtivaLB2   <= '0';
                            AtivaLB3   <= '0';
                            Sel_conect <= '1';
                        WHEN  endM7 =>
                            conectB0   <= '0';
                            conectB1   <= '0';
                            conectB2   <= '0';
                            conectB3   <= '0';
                            AtivaLB0   <= '0';
                            AtivaLB1   <= '0';
                            AtivaLB2   <= '0';
                            AtivaLB3   <= '0';
                            Sel_conect <= '1';
                        END CASE;
                    END Process saidas_men7;

END broadc;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- ************************************************************
-- máquina de estados para desconectar duas saídas - mensagem 6
-- ************************************************************


ENTITY disc6 IS
    PORT(clk                :       in std_logic;
         habilita1          :       in std_logic;
         Ender1             :       in std_logic_vector (1 downto 0);
         habilita2          :       in std_logic;
         Ender2             :       in std_logic_vector (1 downto 0);
         MEN6               :       in std_logic;
         disconect6A        :       out std_logic;
         disconect6B        :       out std_logic;
         disconect6C        :       out std_logic;
         disconect6D        :       out std_logic;
         sel6               :       out std_logic);

END disc6;

ARCHITECTURE disc6 OF disc6 IS

    Type State_Type is (idle6,decodeD1,bitD10,bitD11,bitD12, bitD13,
                        decodeD2,bitD20, bitD21,bitD22, bitD23,endD6);
                         -- estados para gerar os sinais de desconexão


    Signal state            :   STATE_TYPE; -- estados para mensagem 6

    Signal Sdisconect1A     :       std_logic;
    Signal Sdisconect1B     :       std_logic;
    Signal Sdisconect1C     :       std_logic;
    Signal Sdisconect1D     :       std_logic;
    Signal Sdisconect2A     :       std_logic;
    Signal Sdisconect2B     :       std_logic;
    Signal Sdisconect2C     :       std_logic;
    Signal Sdisconect2D     :       std_logic;

begin

estado_men6     :   Process (clk)
                    variable aux1 : std_logic_vector (1 downto 0);
                    variable aux2 : std_logic_vector (1 downto 0);
                    begin
                        aux1 := ender1;
                        aux2 := ender2;
                        if (clk'event and clk = '1' ) then
                                CASE state IS
                                WHEN idle6 =>
                                    IF MEN6 = '1'
                                    THEN state <= decodeD2;
                                    ELSE state <= idle6;
                                    END IF;
                                WHEN decodeD2 =>
                                    IF habilita2 = '1'
                                    THEN
                                        CASE aux2 IS
                                        WHEN "00" =>
                                            state <= bitD20;
                                        WHEN "01" =>
                                            state <= bitD21;
                                        WHEN "10" =>
                                            state <= bitD22;
                                        WHEN "11" =>
                                            state <= bitD23;
```

```
                                    WHEN others =>
                                        state <= endD6;
                                    end case;
                                ELSE state <= decodeD2;
                                END IF;
                            WHEN bitD20 =>
                                state <= decodeD1;
                            WHEN bitD21 =>
                                state <= decodeD1;
                            WHEN bitD22 =>
                                state <= decodeD1;
                            WHEN bitD23 =>
                                state <= decodeD1;
                            WHEN decodeD1 =>
                                IF habilita1 = '1'
                                THEN
                                    CASE aux1 IS
                                    WHEN "00" =>
                                        state <= bitD10;
                                    WHEN "01" =>
                                        state <= bitD11;
                                    WHEN "10" =>
                                        state <= bitD12;
                                    WHEN "11" =>
                                        state <= bitD13;
                                    WHEN others =>
                                        state <= endD6;
                                    end case;
                                ELSE state <= decodeD1;
                                END IF;
                            WHEN bitD10 =>
                                state <= endD6 ;
                            WHEN bitD11 =>
                                state <= endD6 ;
                            WHEN bitD12 =>
                                state <= endD6 ;
                            WHEN bitD13 =>
                                state <= endD6 ;
                            WHEN endD6 =>
                                IF MEN6 = '1'
                                THEN state <= endD6;
                                ELSE state <= idle6;
                                END IF;
                            end case;
                    end if;
                end process estado_men6;

saidas_men6     :   Process(state)
                    BEGIN
                        CASE state IS
                        WHEN idle6 =>
                            Sdisconect1A <= '0';
                            Sdisconect1B <= '0';
                            Sdisconect1C <= '0';
                            Sdisconect1D <= '0';
                            Sdisconect2A <= '0';
                            Sdisconect2B <= '0';
                            Sdisconect2C <= '0';
                            Sdisconect2D <= '0';
                            Sel6 <= '0';
                        WHEN decodeD2 =>
                            Sdisconect1A <= '0';
                            Sdisconect1B <= '0';
                            Sdisconect1C <= '0';
                            Sdisconect1D <= '0';
                            Sdisconect2A <= '0';
                            Sdisconect2B <= '0';
                            Sdisconect2C <= '0';
```

```
            Sdisconect2D <= '0';
            Sel6 <= '0';
        WHEN bitD20 =>
            Sdisconect1A <= '0';
            Sdisconect1B <= '0';
            Sdisconect1C <= '0';
            Sdisconect1D <= '0';
            Sdisconect2A <= '1';
            Sdisconect2B <= '0';
            Sdisconect2C <= '0';
            Sdisconect2D <= '0';
            Sel6 <= '1';
    WHEN bitD21 =>
            Sdisconect1A <= '0';
            Sdisconect1B <= '0';
            Sdisconect1C <= '0';
            Sdisconect1D <= '0';
            Sdisconect2A <= '0';
            Sdisconect2B <= '1';
            Sdisconect2C <= '0';
            Sdisconect2D <= '0';
            Sel6 <= '1';
        WHEN bitD22 =>
            Sdisconect1A <= '0';
            Sdisconect1B <= '0';
            Sdisconect1C <= '0';
            Sdisconect1D <= '0';
            Sdisconect2A <= '0';
            Sdisconect2B <= '0';
            Sdisconect2C <= '1';
            Sdisconect2D <= '0';
            Sel6 <= '1';
        WHEN bitD23 =>
            Sdisconect1A <= '0';
            Sdisconect1B <= '0';
            Sdisconect1C <= '0';
            Sdisconect1D <= '0';
            Sdisconect2A <= '0';
            Sdisconect2B <= '0';
            Sdisconect2C <= '0';
            Sdisconect2D <= '1';
            Sel6 <= '1';
        WHEN decodeD1 =>
            Sdisconect1A <= '0';
            Sdisconect1B <= '0';
            Sdisconect1C <= '0';
            Sdisconect1D <= '0';
            Sdisconect2A <= '0';
            Sdisconect2B <= '0';
            Sdisconect2C <= '0';
            Sdisconect2D <= '0';
            Sel6 <= '1';
        WHEN bitD10 =>
            Sdisconect1A <= '1';
            Sdisconect1B <= '0';
            Sdisconect1C <= '0';
            Sdisconect1D <= '0';
            Sdisconect2A <= '0';
            Sdisconect2B <= '0';
            Sdisconect2C <= '0';
            Sdisconect2D <= '0';
            Sel6 <= '1';
        WHEN bitD11 =>
            Sdisconect1A <= '0';
            Sdisconect1B <= '1';
            Sdisconect1C <= '0';
            Sdisconect1D <= '0';
            Sdisconect2A <= '0';
```

```
                Sdisconect2B <= '0';
                Sdisconect2C <= '0';
                Sdisconect2D <= '0';
                Sel6 <= '1';
            WHEN bitD12 =>
                Sdisconect1A <= '0';
                Sdisconect1B <= '0';
                Sdisconect1C <= '1';
                Sdisconect1D <= '0';
                Sdisconect2A <= '0';
                Sdisconect2B <= '0';
                Sdisconect2C <= '0';
                Sdisconect2D <= '0';
                Sel6 <= '1';
            WHEN bitD13 =>
                Sdisconect1A <= '0';
                Sdisconect1B <= '0';
                Sdisconect1C <= '0';
                Sdisconect1D <= '1';
                Sdisconect2A <= '0';
                Sdisconect2B <= '0';
                Sdisconect2C <= '0';
                Sdisconect2D <= '0';
                Sel6 <= '1';
            WHEN endD6 =>
                Sdisconect1A <= '0';
                Sdisconect1B <= '0';
                Sdisconect1C <= '0';
                Sdisconect1D <= '0';
                Sdisconect2A <= '0';
                Sdisconect2B <= '0';
                Sdisconect2C <= '0';
                Sdisconect2D <= '0';
                Sel6 <= '0';
            END CASE;
        END Process saidas_men6;


        disconect6A <= Sdisconect1A or Sdisconect2A;
        disconect6B <= Sdisconect1B or Sdisconect2B;
        disconect6C <= Sdisconect1C or Sdisconect2C;
        disconect6D <= Sdisconect1D or Sdisconect2D;

end disc6;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- ************************************************************
-- máqina de estados para desconexão ... mnesagem 5 e 4 (resete)
-- ************************************************************


ENTITY disc IS
    PORT(clk                :       in std_logic;
         habilita1          :       in std_logic;
         Ender1             :       in std_logic_vector (1 downto 0);
         MEN4               :       in std_logic;
         MEN5               :       in std_logic;
         disconect5A        :       out std_logic;
         disconect5B        :       out std_logic;
         disconect5C        :       out std_logic;
         disconect5D        :       out std_logic);

END disc;

ARCHITECTURE disc OF disc IS

    Type State_Type is (idle5,decodeD1,bitD0,bitD1,bitD2, bitD3,endD);
                        -- estados para gerar os sinais de desconexao


    Signal state            :   STATE_TYPE; -- estados para mensagem 5

    Signal Sdisconect0      :       std_logic;
    Signal Sdisconect1      :       std_logic;
    Signal Sdisconect2      :       std_logic;
    Signal Sdisconect3      :       std_logic;
    Signal A                :       std_logic;
    Signal B                :       std_logic;
    Signal C                :       std_logic;
    Signal D                :       std_logic;


begin

estado_men5 :   Process (clk)
                    variable aux : std_logic_vector (1 downto 0);
                    begin
                        aux := ender1;
                        if (clk'event and clk = '1' ) then
                                CASE state IS
                                WHEN idle5 =>
                                    IF MEN5 = '1'
                                    THEN state <= decodeD1;
                                    ELSE state <= idle5;
                                    END IF;
                                WHEN decodeD1 =>
                                    IF habilita1 = '1'
                                    THEN
                                        CASE aux IS
                                        WHEN "00" =>
                                            state <= bitD0;
                                        WHEN "01" =>
                                            state <= bitD1;
                                        WHEN "10" =>
                                            state <= bitD2;
                                        WHEN "11" =>
                                            state <= bitD3;
                                        WHEN others =>
                                            state <= idle5;
                                        end case;
                                    ELSE state <= decodeD1;
```

```
                                END IF;
                            WHEN bitD0 =>
                                state <= endD;
                            WHEN bitD1 =>
                                state <= endD;
                            WHEN bitD2 =>
                                state <= endD;
                            WHEN bitD3 =>
                                state <= endD;
                            WHEN endD=>
                                IF MEN5 = '1'
                                THEN state <= endD;
                                ELSE state <= idle5;
                                END IF;
                            end case;
                    end if;
                end process estado_men5;

saidas_men5     :   Process(state)
                    BEGIN
                        CASE state IS
                        WHEN idle5 =>
                            Sdisconect0 <= '0';
                            Sdisconect1 <= '0';
                            Sdisconect2 <= '0';
                            Sdisconect3 <= '0';
                        WHEN decodeD1 =>
                            Sdisconect0 <= '0';
                            Sdisconect1 <= '0';
                            Sdisconect2 <= '0';
                            Sdisconect3 <= '0';
                        WHEN bitD0 =>
                            Sdisconect0 <= '1';
                            Sdisconect1 <= '0';
                            Sdisconect2 <= '0';
                            Sdisconect3 <= '0';
                        WHEN bitD1 =>
                            Sdisconect0 <= '0';
                            Sdisconect1 <= '1';
                            Sdisconect2 <= '0';
                            Sdisconect3 <= '0';
                        WHEN bitD2 =>
                            Sdisconect0 <= '0';
                            Sdisconect1 <= '0';
                            Sdisconect2 <= '1';
                            Sdisconect3 <= '0';
                        WHEN bitD3 =>
                            Sdisconect0 <= '0';
                            Sdisconect1 <= '0';
                            Sdisconect2 <= '0';
                            Sdisconect3 <= '1';
                        WHEN endD =>
                            Sdisconect0 <= '0';
                            Sdisconect1 <= '0';
                            Sdisconect2 <= '0';
                            Sdisconect3 <= '0';
                        END CASE;
                    END Process saidas_men5;

men_4       :       Process (MEN4)
                    variable aux : std_logic;
                    begin
                        aux := MEN4;
                        if aux = '1'
                        then
                            A <= '1';
                            B <= '1';
                            C <= '1';
```

```
                D <= '1';
            else
                A <= '0';
                B <= '0';
                C <= '0';
                D <= '0';
            end if;
        end process men_4;

        disconect5A <= Sdisconect0 or A;
        disconect5B <= Sdisconect1 or B;
        disconect5C <= Sdisconect2 or C;
        disconect5D <= Sdisconect3 or D;

end disc;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- ************************************************************
-- máquina de estados para exibir status de uma saída
-- ************************************************************


ENTITY mesage2 IS
    PORT(clk                :       in std_logic;
         MEN2               :       in std_logic;
         habilita1          :       in std_logic;
         loadr              :       out std_logic;
         discharge          :       out std_logic;
         sel                :       out std_logic);

END mesage2;

ARCHITECTURE mesage2 OF mesage2 IS

    Type STATE_TYPE is (idler,wait_ender1,load_reg,shiftst1,shiftst2,
                        shift0,shift1,shift2,shift3,shift4,shift5,
                                shift6,shift7,shiftstop,endm2);
                    -- estados para saída no configlinkout

    Signal state            :   STATE_TYPE; -- estados para mensagem 2

BEGIN

estado_men2     :   Process (clk)
                    begin
                        if (clk'event and clk = '1' )
                        then
                            CASE state IS
                            WHEN idler =>
                                IF MEN2 = '1'
                                THEN state <= wait_ender1;
                                ELSE state <= idler;
                                END IF;
                            WHEN wait_ender1 =>
                                if habilita1 = '1'
                                then state <= load_reg;
                                else state <= wait_ender1;
                                end if;
                            WHEN load_reg =>
                                state <= shiftst1;
                            WHEN shiftst1 =>
                                state <= shiftst2;
                            WHEN shiftst2 =>
                                state <= shift0;
                            WHEN shift0 =>
                                state <= shift1;
                            WHEN shift1 =>
                                state <= shift2;
                            WHEN shift2 =>
                                state <= shift3;
                            WHEN shift3 =>
                                state <= shift4;
                            WHEN shift4 =>
                                state <= shift5;
                            WHEN shift5 =>
                                state <= shift6;
                            WHEN shift6 =>
                                state <= shift7;
                            WHEN shift7 =>
                                state <= shiftstop;
                            WHEN shiftstop =>
                                state <= endm2;
```

```
                    WHEN endm2=>
                        if MEN2 = '1'
                        then state <= endm2;
                        else state <= idler;
                        end if;
                    END CASE;
                END IF;
            END PROCESS estado_men2;

saidas_men2     :   Process(state)
                    BEGIN
                        CASE state IS
                        WHEN idler =>
                            loadr <= '0';
                            discharge <= '0';
                            sel <= '0';
                        WHEN wait_ender1 =>
                            loadr <= '0';
                            discharge <= '0';
                            sel <= '0';
                        WHEN load_reg =>
                            loadr <= '1';
                            discharge <= '0';
                            sel <= '0';
                        WHEN shiftst1 =>
                            loadr <= '0';
                            discharge <= '1';
                            sel <= '1';
                        WHEN shiftst2 =>
                            loadr <= '0';
                            discharge <= '1';
                            sel <= '1';
                        WHEN shift0 =>
                            loadr <= '0';
                            discharge <= '1';
                            sel <= '1';
                        WHEN shift1 =>
                            loadr <= '0';
                            discharge <= '1';
                            sel <= '1';
                        WHEN shift2 =>
                            loadr <= '0';
                            discharge <= '1';
                            sel <= '1';
                        WHEN shift3 =>
                            loadr <= '0';
                            discharge <= '1';
                            sel <= '1';
                        WHEN shift4 =>
                            loadr <= '0';
                            discharge <= '1';
                            sel <= '1';
                        WHEN shift5 =>
                            loadr <= '0';
                            discharge <= '1';
                            sel <= '1';
                        WHEN shift6 =>
                            loadr <= '0';
                            discharge <= '1';
                            sel <= '1';
                        WHEN shift7 =>
                            loadr <= '0';
                            discharge <= '1';
                            sel <= '1';
                        WHEN shiftstop =>
                            loadr <= '0';
                            discharge <= '1';
                            sel <= '1';
```

```
            WHEN endm2 =>
                loadr <= '0';
                discharge <= '0';
                sel <= '0';
            END CASE;
        END Process saidas_men2;

END mesage2;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- ************************************************************
-- máquina de estados para conectar uma entrada em uma saída
-- ************************************************************

ENTITY conect IS
    PORT(clk                :       in std_logic;
         MEN0               :       in std_logic;
         MEN3               :       in std_logic;
         habilita1          :       in std_logic;
         Ender1             :       in std_logic_vector (1 downto 0);
         AtivaLC0           :       out std_logic;
         AtivaLC1           :       out std_logic;
         AtivaLC2           :       out std_logic;
         AtivaLC3           :       out std_logic;
         conectC0           :       out std_logic;
         conectC1           :       out std_logic;
         conectC2           :       out std_logic;
         conectC3           :       out std_logic);

END conect;

ARCHITECTURE conect OF conect IS

    Type STATE_TYPE is (idle0,decodeC1,bitA0,bitC0,bitA1,bitC1,bitA2,
                                                bitC2,bitA3,bitC3);
                        -- estados para gerar os sinais de conexao

    Signal state            :   STATE_TYPE; -- estados para mensagem 0

BEGIN

estado_men0     :   Process (clk)
                    begin
                        if (clk'event and clk = '1' )
                        then
                            CASE state IS
                            WHEN idle0 =>
                                IF MEN0 = '1'
                                THEN state <= decodeC1;
                                ELSE state <=   idle0;
                                END IF;
                            WHEN decodeC1 =>
                                IF habilita1 = '1'
                                THEN
                                    CASE ender1 IS
                                    WHEN "00" =>
                                        state <= bitA0;
                                    WHEN "01" =>
                                        state <= bitA1;
                                    WHEN "10" =>
                                        state <= bitA2;
                                    WHEN "11" =>
                                        state <= bitA3;
                                    WHEN others =>
                                        state <= decodeC1;
                                    end case;
                                ELSE state <= decodeC1;
                                END IF;
                            WHEN bitA0 =>
                                IF MEN3 = '1'
                                THEN state <= bitC0;
                                ELSE state <= bitA0;
                                END IF;
                            WHEN bitC0 =>
                                state <= idle0;
```

```
                    WHEN bitA1 =>
                        IF MEN3 = '1'
                        THEN state <= bitC1;
                        ELSE state <= bitA1;
                        END IF;
                    WHEN bitC1 =>
                        state <= idle0;
                    WHEN bitA2 =>
                        IF MEN3 = '1'
                        THEN state <= bitC2;
                        ELSE state <= bitA2;
                        END IF;
                    WHEN bitC2 =>
                        state <= idle0;
                    WHEN bitA3 =>
                        IF MEN3 = '1'
                        THEN state <= bitC3;
                        ELSE state <= bitA3;
                        END IF;
                    WHEN bitC3 =>
                        state <= idle0;
                    END CASE;
                END IF;
            END PROCESS estado_men0;

saidas_men0     :   Process(state)
                    BEGIN
                        CASE state IS
                        WHEN idle0 =>
                            conectC0 <= '0';
                            conectC1 <= '0';
                            conectC2 <= '0';
                            conectC3 <= '0';
                            AtivaLC0    <= '0';
                            AtivaLC1    <= '0';
                            AtivaLC2    <= '0';
                            AtivaLC3    <= '0';
                        WHEN decodeC1 =>
                            conectC0 <= '0';
                            conectC1 <= '0';
                            conectC2 <= '0';
                            conectC3 <= '0';
                            AtivaLC0    <= '0';
                            AtivaLC1    <= '0';
                            AtivaLC2    <= '0';
                            AtivaLC3    <= '0';
                        WHEN bitA0 =>
                            conectC0 <= '0';
                            conectC1 <= '0';
                            conectC2 <= '0';
                            conectC3 <= '0';
                            AtivaLC0    <= '1';
                            AtivaLC1    <= '0';
                            AtivaLC2    <= '0';
                            AtivaLC3    <= '0';
                        WHEN bitC0 =>
                            conectC0 <= '1';
                            conectC1 <= '0';
                            conectC2 <= '0';
                            conectC3 <= '0';
                            AtivaLC0    <= '0';
                            AtivaLC1    <= '0';
                            AtivaLC2    <= '0';
                            AtivaLC3    <= '0';
                        WHEN bitA1 =>
                            conectC0 <= '0';
                            conectC1 <= '0';
                            conectC2 <= '0';
```

```
            conectC3 <= '0';
            AtivaLC0    <= '0';
            AtivaLC1    <= '1';
            AtivaLC2    <= '0';
            AtivaLC3    <= '0';
        WHEN bitC1 =>
            conectC0 <= '0';
            conectC1 <= '1';
            conectC2 <= '0';
            conectC3 <= '0';
            AtivaLC0    <= '0';
            AtivaLC1    <= '0';
            AtivaLC2    <= '0';
            AtivaLC3    <= '0';
        WHEN bitA2 =>
            conectC0 <= '0';
            conectC1 <= '0';
            conectC2 <= '0';
            conectC3 <= '0';
            AtivaLC0    <= '0';
            AtivaLC1    <= '0';
            AtivaLC2    <= '1';
            AtivaLC3    <= '0';
        WHEN bitC2 =>
            conectC0 <= '0';
            conectC1 <= '0';
            conectC2 <= '1';
            conectC3 <= '0';
            AtivaLC0    <= '0';
            AtivaLC1    <= '0';
            AtivaLC2    <= '0';
            AtivaLC3    <= '0';
        WHEN bitA3 =>
            conectC0 <= '0';
            conectC1 <= '0';
            conectC2 <= '0';
            conectC3 <= '0';
            AtivaLC0    <= '0';
            AtivaLC1    <= '0';
            AtivaLC2    <= '0';
            AtivaLC3    <= '1';
        WHEN bitC3 =>
            conectC0 <= '0';
            conectC1 <= '0';
            conectC2 <= '0';
            conectC3 <= '1';
            AtivaLC0    <= '0';
            AtivaLC1    <= '0';
            AtivaLC2    <= '0';
            AtivaLC3    <= '0';
        WHEN others =>
            conectC0 <= '0';
            conectC1 <= '0';
            conectC2 <= '0';
            conectC3 <= '0';
            AtivaLC0    <= '0';
            AtivaLC1    <= '0';
            AtivaLC2    <= '0';
            AtivaLC3    <= '0';
        END CASE;
    END Process saidas_men0;

END conect;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- ********************************************************
-- multiplexador auxiliar para a mensagem 2
-- ********************************************************


ENTITY MUXR IS
    PORT(MEN2               :       in std_logic;
         Ender1             :       in std_logic_vector (1 downto 0);
         MU0                :       in std_logic_vector (2 downto 0);
         MU1                :       in std_logic_vector (2 downto 0);
         MU2                :       in std_logic_vector (2 downto 0);
         MU3                :       in std_logic_vector (2 downto 0);
         dataL              :       out std_logic_vector (2 downto 0));
END MUXR;

ARCHITECTURE MUXR OF MUXR IS
signal A : std_logic_vector (2 downto 0);
begin

gera_sinal2     :   Process (Ender1,MEN2)
                    begin
                        if MEN2 = '1'
                        then
                            case Ender1 is
                                when "00" =>  -- 0
                                        A <= MU0;
                                when "01" =>  -- 1
                                        A <= MU1;
                                when "10" =>  -- 2
                                        A <= MU2;
                                when "11" =>  -- 3
                                        A <= MU3;
                                when OTHERS =>  -- outras entradas
                            end case;
                        end if;
                    end process gera_sinal2;
                    dataL <= A;

end MUXR;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

-- ************************************************************
-- multiplexador para tipo de desconexão
-- ************************************************************


ENTITY muxD IS
    PORT(disconect6A        :       in std_logic;
         disconect6B        :       in std_logic;
         disconect6C        :       in std_logic;
         disconect6D        :       in std_logic;
         sel6               :       in std_logic;
         disconect5A        :       in std_logic;
         disconect5B        :       in std_logic;
         disconect5C        :       in std_logic;
         disconect5D        :       in std_logic;
         disconect0         :       out std_logic;
         disconect1         :       out std_logic;
         disconect2         :       out std_logic;
         disconect3         :       out std_logic);

END muxD;

ARCHITECTURE muxD OF muxD IS

BEGIN
multiplexador   :   Process(sel6,disconect6A,disconect6B,disconect6C,
                            disconect6D,disconect5A,disconect5B,
                                        disconect5C,disconect5D)
                    begin
                        IF sel6 = '1'
                        THEN
                             disconect0 <= disconect6A;
                             disconect1 <= disconect6B;
                             disconect2 <= disconect6C;
                             disconect3 <= disconect6D;
                        ELSE
                             disconect0 <= disconect5A;
                             disconect1 <= disconect5B;
                             disconect2 <= disconect5C;
                             disconect3 <= disconect5D;
                        END IF;
                    End process multiplexador;
END muxD;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- ***********************************************************
-- multiplexador para saída ConfigLinkOut
-- ***********************************************************

ENTITY muxc IS
    PORT(CLinkOut1          :       in std_logic;
         CLinkOut2          :       in std_logic;
         sel                :       in std_logic;
         ConfigLinkOut      :       out std_logic);
END muxc;

ARCHITECTURE muxc OF muxc IS

BEGIN
multiplexador   :   Process(Clinkout1,ClinkOut2,sel)
                    begin
                        IF sel = '1'
                        THEN ConfigLinkOut <= CLinkOut2;
                        ELSE ConfigLinkOut <= CLinkOut1;
                        END IF;
                    End process multiplexador;
END muxc;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
-- ************************************************************
-- multiplexador para selecionar tipo de conexão
-- ************************************************************

ENTITY muxBC IS
    PORT(conectC0           :       in std_logic;
         conectC1           :       in std_logic;
         conectC2           :       in std_logic;
         conectC3           :       in std_logic;
         AtivaLB0           :       in std_logic;
         AtivaLB1           :       in std_logic;
         AtivaLB2           :       in std_logic;
         AtivaLB3           :       in std_logic;
         sel_conect         :       in std_logic;
         conectB0           :       in std_logic;
         conectB1           :       in std_logic;
         conectB2           :       in std_logic;
         conectB3           :       in std_logic;
         AtivaLC0           :       in std_logic;
         AtivaLC1           :       in std_logic;
         AtivaLC2           :       in std_logic;
         AtivaLC3           :       in std_logic;
         conect0            :       out std_logic;
         conect1            :       out std_logic;
         conect2            :       out std_logic;
         conect3            :       out std_logic;
         AtivaL0            :       out std_logic;
         AtivaL1            :       out std_logic;
         AtivaL2            :       out std_logic;
         AtivaL3            :       out std_logic);
END muxBC;

ARCHITECTURE muxBC OF muxBC IS
BEGIN
multiplexador   :   Process(sel_conect,conectB0,conectB1,conectB2,conectB3,
                            AtivaLB0,AtivaLB1,AtivaLB2,AtivaLB3,conectC0,
                            conectC1,conectC2,conectC3,AtivaLC0,AtivaLC1,
                                                    AtivaLC2,AtivaLC3)
                    begin
                        IF sel_conect = '1'
                        THEN
                            conect0 <= conectB0;
                            conect1 <= conectB1;
                            conect2 <= conectB2;
                            conect3 <= conectB3;
                            AtivaL0 <= AtivaLB0;
                            AtivaL1 <= AtivaLB1;
                            AtivaL2 <= AtivaLB2;
                            AtivaL3 <= AtivaLB3;
                        ELSE
                            conect0 <= conectC0;
                            conect1 <= conectC1;
                            conect2 <= conectC2;
                            conect3 <= conectC3;
                            AtivaL0 <= AtivaLC0;
                            AtivaL1 <= AtivaLC1;
                            AtivaL2 <= AtivaLC2;
                            AtivaL3 <= AtivaLC3;
                        END IF;
                    End process multiplexador;
END muxBC;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- ********************************************************
-- registrador de deslocamento da saída  para esquerda
-- ********************************************************

ENTITY reg_m2 IS
    PORT(clk                :       in std_logic;
         datal              :       in std_logic_vector (2 downto 0);
         loadr              :       in std_logic;
         discharge          :       in std_logic;
         CLinkOut2          :       out std_logic);
END reg_m2;

ARCHITECTURE reg_m2 OF reg_m2 IS

BEGIN
Reg_deslocamento:   Process(clk)
                    variable Sa,Sb,Sc,Sd,Se,Sf,Sg,Sh,Si,Sj,Sk  : std_logic;
                    begin
                        IF (clk'event and clk = '1')
                        THEN IF loadr = '1'
                             THEN
                                Sa := '1';
                                Sb := '1';
                                Sc := datal(2);
                                Sd := '0';
                                Se := '0';
                                Sf := '0';
                                Sg := '0';
                                Sh := '0';
                                Si := datal(1);
                                Sj := datal(0);
                                Sk := '0';
                             END IF; -- fim do loadr
                             IF discharge = '1'
                             THEN
                                CLinkOut2 <= Sa;
                                Sa := Sb;
                                Sb := Sc;
                                Sc := Sd;
                                Sd := Se;
                                Se := Sf;
                                Sf := Sg;
                                Sg := Sh;
                                Sh := Si;
                                Si := Sj;
                                Sj := Sk;
                              END IF; -- fim do discharge
                        END IF; -- fim do clk
                    End process reg_deslocamento;
END reg_m2;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- ************************************************************
-- multiplexador
-- ************************************************************


ENTITY mux0 IS
    PORT(LinkIn0            :       in std_logic; -- links de entrada
         LinkIn1            :       in std_logic;
         LinkIn2            :       in std_logic;
         LinkIn3            :       in std_logic;
         MU0                :       in std_logic_vector (2 downto 0);
         LinkOut0           :       out std_logic); -- link de saída

END mux0;

ARCHITECTURE mux0 OF mux0 IS
SIGNAL A                    :       std_logic ;

Begin


multiplexa01    :       Process (MU0,LinkIn0,LinkIn1,LinkIn2,LinkIn3)
                        variable aux : std_logic_vector (1 downto 0);
                        variable aux2 : std_logic;
                        begin
                            aux := MU0 (1 downto 0);
                            aux2:= MU0(2);
                            if aux2 = '1'
                            then
                                CASE aux IS
                                WHEN "00"=>
                                    A <= LinkIn0;
                                WHEN "01"=>
                                    A <= LinkIn1;
                                WHEN "10"=>
                                    A <= LinkIn2;
                                WHEN "11"=>
                                    A <= LinkIn3;
                                WHEN others =>
                                    A <= '0';
                                END CASE;
                            else A <= '0';
                            end if;
                        end Process multiplexa01;
                        LinkOut0 <= A;

end mux0;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- ************************************************************
-- Latchn
-- ************************************************************


ENTITY latch0 IS
    PORT(ativaL0            :       in std_logic; -- ativa latch 0
         Ender2             :       in std_logic_vector (1 downto 0);
         Conect0            :       in std_logic; -- conecta MUX 0
         DisConect0         :       in std_logic; -- conecta MUX 0
         MU0                :       out std_logic_vector (2 downto 0));

END latch0;

ARCHITECTURE latch0 OF latch0 IS
Signal      Q               :       std_logic_vector (2 downto 0);

Begin

p_latch01       :       Process (ativaL0,conect0,disconect0)
                        begin
                            if (ativaL0 = '1')
                            then
                                Q(1 downto 0) <= Ender2(1 downto 0);
                            end if;
                            if disconect0 = '1' then
                                Q(2) <= '0';
                            elsif conect0'EVENT and conect0 = '1' then
                                Q(2) <= '1';
                            end if;
                        end Process p_latch01;

                        MU0 <= Q ;

end latch0;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- ************************************************************
-- decodificador de instruções
-- obs: não implementado MEN1 (mensagem 1)
-- ************************************************************

ENTITY deco2 IS
    PORT(enable1            :       in std_logic;
         data               :       in std_logic_vector (7 downto 0);
         nbyte2             :       out std_logic;
         nbyte3             :       out std_logic;
         ignora             :       out std_logic;
         MEN0               :       out std_logic;
    --   MEN1               :       out std_logic; não implementado
         MEN2               :       out std_logic;
         MEN3               :       out std_logic;
         MEN4               :       out std_logic;
         MEN5               :       out std_logic;
         MEN6               :       out std_logic;
         MEN7               :       out std_logic);
END deco2 ;

ARCHITECTURE deco2 OF deco2 IS

BEGIN
decode_instr    :   Process (enable1)
                    begin
                        if enable1 = '1'
                        then
                            case data is
                                when "00000000" =>  -- mensagem 0 conectar

                                    nbyte2  <= '1'; -- entrada ender2 em
                                    nbyte3  <= '1'; -- saída ender1
                                    ignora  <= '0';
                                    MEN0    <= '1';
                                    MEN2    <= '0';
                                    MEN3    <= '0';
                                    MEN4    <= '0';
                                    MEN5    <= '0';
                                    MEN6    <= '0';
                                    MEN7    <= '0';
                                when "00000001" =>  -- mensagem 1 conectar

                                    nbyte2  <= '1'; -- entradas e saída
                                    nbyte3  <= '1'; -- OBS. não implementado

                                    ignora  <= '1';
                                    MEN0    <= '0';
                                    MEN2    <= '0';
                                    MEN3    <= '0';
                                    MEN4    <= '0';
                                    MEN5    <= '0';
                                    MEN6    <= '0';
                                    MEN7    <= '0';
                                when "00000010" =>  -- mensagem 2 status
                                    nbyte2  <= '1'; -- da saída
                                    nbyte3  <= '0';
                                    ignora  <= '0';
                                    MEN0    <= '0';
                                    MEN2    <= '1';
                                    MEN3    <= '0';
                                    MEN4    <= '0';
                                    MEN5    <= '0';
                                    MEN6    <= '0';
                                    MEN7    <= '0';
```

```
            when "00000011" =>  -- mensagem 3 final
                nbyte2  <= '0'; -- da configuração
                nbyte3  <= '0';
                ignora  <= '0';
                MEN0    <= '0';
                MEN2    <= '0';
                MEN3    <= '1';
                MEN4    <= '0';
                MEN5    <= '0';
                MEN6    <= '0';
                MEN7    <= '0';
        when "00000100" => -- mensagem 4 reinicialização

                nbyte2  <= '0';
                nbyte3  <= '0';
                ignora  <= '0';
                MEN0    <= '0';
                MEN2    <= '0';
                MEN3    <= '0';
                MEN4    <= '1';
                MEN5    <= '0';
                MEN6    <= '0';
                MEN7    <= '0';
            when "00000101" => -- mensagem 5 desconectar

                nbyte2  <= '1'; -- saída endereçada
                nbyte3  <= '0';
                ignora  <= '0';
                MEN0    <= '0';
                MEN2    <= '0';
                MEN3    <= '0';
                MEN4    <= '0';
                MEN5    <= '1';
                MEN6    <= '0';
                MEN7    <= '0';
            when "00000110" => -- mensagem 6 desconectar

                nbyte2  <= '1'; -- saídas endereçadas
                nbyte3  <= '1';
                ignora  <= '0';
                MEN0    <= '0';
                MEN2    <= '0';
                MEN3    <= '0';
                MEN4    <= '0';
                MEN5    <= '0';
                MEN6    <= '1';
                MEN7    <= '0';
            when "00000111" =>  -- mensagem 7
                nbyte2  <= '1'; -- broadcasting
                nbyte3  <= '1';
                ignora  <= '0';
                MEN0    <= '0';
                MEN2    <= '0';
                MEN3    <= '0';
                MEN4    <= '0';
                MEN5    <= '0';
                MEN6    <= '0';
                MEN7    <= '1';
            when OTHERS =>  -- decodificação de mensagem

                nbyte2  <= '0'; -- ignorada
                nbyte3  <= '0';
                ignora  <= '1';
                MEN0    <= '0';
                MEN2    <= '0';
                MEN3    <= '0';
                MEN4    <= '0';
                MEN5    <= '0';
```

```
                    MEN6    <= '0';
                    MEN7    <= '0';
                end case;
            end if; -- fim do if do enable1
        end process decode_instr;
END deco2;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- ************************************************************
-- máquina de estados para ler toda a mensagem de configuração
-- ************************************************************

ENTITY read_me2 IS
    PORT(ConfigLinkIn       :       in std_logic;
         clk                :       in std_logic;
         trigger            :       in std_logic;
         ignora             :       in std_logic;
         nbyte2             :       in std_logic;
         nbyte3             :       in std_logic;
         enable1            :       out std_logic;
         enab_ender2        :       out std_logic;
         enab_ender1        :       out std_logic);

END read_me2;

ARCHITECTURE read_me2 OF read_me2 IS

    Type STATE_TYPE is (idle2,rec_byte1,wait_1,confirm1,rec_byte2,wait_2,
                                                confirm2,rec_byte3,wait_3);
                -- estados para leitura de toda mensagem de configuração


    Signal state            :   STATE_TYPE;
                                -- estados para leitura da mensagem

BEGIN
leitura_mensagem:   Process (clk)
                    begin
                        if (clk'event and clk = '1' )
                        then
                            CASE state IS
                            WHEN idle2 =>
                                IF ConfigLinkIn = '0'
                                THEN state <= idle2;
                                ELSE state <= rec_byte1;
                                END IF;
                            WHEN rec_byte1 =>
                                IF trigger = '0'
                                THEN state <= rec_byte1;
                                ELSE state <= wait_1;
                                END IF;
                            WHEN wait_1 =>
                                state <= confirm1;
                            WHEN confirm1 =>
                                if ignora = '1'
                                then    state <= idle2;
                                else    if nbyte3 = '0' then
                                            state <= confirm2;
                                        elsif nbyte2 = '0' then
                                            state <= idle2;
                                        else
                                            state <= rec_byte2;
                                        end if;
                                end if;
                            WHEN rec_byte2 =>
                                IF trigger = '0'
                                THEN state <= rec_byte2;
                                ELSE state <= wait_2;
                                END IF;
                            WHEN wait_2 =>
                                state <= confirm2;
                            WHEN confirm2 =>
                                IF nbyte2 = '1'
```

```
                        THEN
                            state <= rec_byte3;
                        ELSE
                            state <= idle2;
                        END IF;
                    WHEN rec_byte3 =>
                        IF trigger = '0'
                        THEN state <= rec_byte3;
                        ELSE state <= wait_3;
                        END IF;
                    WHEN wait_3 =>
                        state <= idle2;
                    END CASE;
                END IF;
            END PROCESS leitura_mensagem;

saida_mensagem  :   Process(state)
                    BEGIN
                        CASE state IS
                        WHEN idle2 =>
                            enable1 <= '0';
                            enab_ender2 <= '0';
                            enab_ender1 <= '0';
                        WHEN rec_byte1 =>
                            enable1 <= '0';
                            enab_ender2 <= '0';
                            enab_ender1 <= '0';
                        WHEN wait_1 =>
                            enable1 <= '1';
                            enab_ender2 <= '0';
                            enab_ender1 <= '0';
                        WHEN confirm1 =>
                            enable1 <= '0';
                            enab_ender2 <= '0';
                            enab_ender1 <= '0';
                        WHEN rec_byte2 =>
                            enable1 <= '0';
                            enab_ender2 <= '0';
                            enab_ender1 <= '0';
                        WHEN wait_2 =>
                            enable1 <= '0';
                            enab_ender2 <= '1';
                            enab_ender1 <= '0';
                        WHEN confirm2 =>
                            enable1 <= '0';
                            enab_ender2 <= '0';
                            enab_ender1 <= '0';
                        WHEN rec_byte3 =>
                            enable1 <= '0';
                            enab_ender2 <= '0';
                            enab_ender1 <= '0';
                        WHEN wait_3 =>
                            enable1 <= '0';
                            enab_ender2 <= '0';
                            enab_ender1 <= '1';
                       END CASE;
                    END Process saida_mensagem;
END read_me2;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- ***********************************************************
-- máquina de estados para controle da leitura do byte
-- ***********************************************************

ENTITY read_by IS
    PORT(ConfigLinkIn       :       in std_logic;
         Clk                :       in std_logic;
         CLinkOut1          :       out std_logic;
         Load               :       out std_logic;
         Enable             :       out std_logic;
         Trigger            :       out std_logic);
END read_by;

ARCHITECTURE read_by OF read_by IS
    Type STATE_TYPE is (idle, rec_start, rec_data0, rec_data1,rec_data2,
                        rec_data3, rec_data4, rec_data5,rec_data6,
                        rec_data7,rec_stop_bit, data_ok, dispara);
                       -- estados para leitura dos bytes de configuração

    Signal state            :   STATE_TYPE; -- estados para leitura do byte
BEGIN

leitura_byte    :   Process (clk)
                    begin
                        if (clk'event and clk = '1' )
                        then
                            CASE state IS
                            WHEN idle =>                -- estado ocioso
                                IF ConfigLinkIn = '0'
                                THEN state <= idle;
                                ELSE state <= rec_start;
                                END IF;
                            WHEN rec_start =>     -- estado espera start bit

                                IF ConfigLinkIn = '0'
                                THEN state <= idle;
                                ELSE state <= rec_data0;
                                END IF;
                            WHEN rec_data0 =>       -- leitura dos dados
                                state <= rec_data1;
                            WHEN rec_data1 =>
                                state <= rec_data2;
                            WHEN rec_data2 =>
                                state <= rec_data3;
                            WHEN rec_data3 =>
                                state <= rec_data4;
                            WHEN rec_data4 =>
                                state <= rec_data5;
                            WHEN rec_data5 =>
                                state <= rec_data6;
                            WHEN rec_data6 =>
                                state <= rec_data7;
                            WHEN rec_data7 =>       -- fim leitura
                                state <= rec_stop_bit;
                            WHEN rec_stop_bit =>    -- espera stop bit
                                IF ConfigLinkIn = '1'
                                THEN state <= idle;
                                ELSE state <= data_ok;
                                END IF;
                            WHEN data_ok =>     -- pacote ok enviar ack
                                    state <= dispara;
                            WHEN dispara =>     -- dispara o próximo estado
                                    state <= idle;  -- shoot
                                END CASE;
                            END IF;
```

```
                    END PROCESS leitura_byte;

saida_leitura   :   Process(state)
                    BEGIN
                        CASE state IS
                        WHEN idle =>            -- saídas desabilitadas
                            CLinkOut1 <= '0';
                            load <= '0';
                            enable <= '0';
                            trigger <= '0';
                        WHEN rec_start =>       -- saídas desabilitadas
                            CLinkOut1 <= '0';
                            load <= '0';
                            enable <= '0';
                            trigger <= '0';
                        WHEN rec_data0 =>  -- carrega no reg de deslocamento

                            CLinkOut1 <= '0';
                            load <= '1';
                            enable <= '0';
                            trigger <= '0';
                        WHEN rec_data1 =>
                            CLinkOut1 <= '0';
                            load <= '1';
                            enable <= '0';
                            trigger <= '0';
                        WHEN rec_data2 =>
                            CLinkOut1 <= '0';
                            load <= '1';
                            enable <= '0';
                            trigger <= '0';
                        WHEN rec_data3 =>
                            CLinkOut1 <= '0';
                            load <= '1';
                            enable <= '0';
                            trigger <= '0';
                        WHEN rec_data4 =>
                            CLinkOut1 <= '0';
                            load <= '1';
                            enable <= '0';
                            trigger <= '0';
                        WHEN rec_data5 =>
                            CLinkOut1 <= '0';
                            load <= '1';
                            enable <= '0';
                            trigger <= '0';
                        WHEN rec_data6 =>
                            CLinkOut1 <= '0';
                            load <= '1';
                            enable <= '0';
                            trigger <= '0';
                        WHEN rec_data7 =>
                            CLinkOut1 <= '0';
                            load <= '1';
                            enable <= '0';
                            trigger <= '0';
                        WHEN rec_stop_bit =>
                            CLinkOut1 <= '0';
                            load <= '0';
                            enable <= '0';
                            trigger <= '0';
                        WHEN data_ok => -- habilta saída reg de deslocamento

                            CLinkOut1 <= '1';  -- envia sinal de recebimento

                            load <= '0';                        -- do byte
                            enable <= '1';
                            trigger <= '0';
```

```
                    WHEN dispara =>
                        CLinkOut1 <= '0';
                        load <= '0';
                        enable <= '0';
                        trigger <= '1';
                    END CASE;
                END Process saida_leitura;
END read_by;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- ********************************************************
-- decodificador de endereços de saída  ( LinkOutn )
-- ********************************************************


ENTITY d_ender1 IS
    PORT(clk                :       in std_logic;
         data               :       in std_logic_vector (7 downto 0);
         enab_ender1        :       in std_logic;
         habilita1          :       out std_logic;
         ender1             :       out std_logic_vector (1 downto 0));

END d_ender1;

ARCHITECTURE d_ender1 OF d_ender1 IS
BEGIN
decode_ender1   :   Process (enab_ender1,clk)
                    begin
                        if (clk'EVENT and clk = '1')
                        then    if enab_ender1 = '1'
                                then
                                    ender1  <= data (1 downto 0);
                                end if;
                                if enab_ender1 = '1'
                                then
                                    habilita1 <= '1';
                                else
                                    habilita1 <= '0';
                                end if;
                        end if;

                    end process decode_ender1;
END d_ender1;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- ************************************************************
-- decodificador de endereços de entrada    ( LinkInn )
-- ************************************************************

ENTITY d_ender2 IS
    PORT(clk                :       in std_logic;
         data               :       in std_logic_vector (7 downto 0);
         enab_ender2        :       in std_logic;
         ender2             :       out std_logic_vector (1 downto 0);
         habilita2          :       out std_logic);

END d_ender2;

ARCHITECTURE d_ender2 OF d_ender2 IS
BEGIN
decode_ender2   :   Process (enab_ender2,clk)
                    begin
                        if (clk'EVENT and clk = '1')
                        then    if enab_ender2 = '1'
                                then
                                    ender2  <= data (1 downto 0);
                                end if;
                                if enab_ender2 = '1'
                                then
                                    habilita2 <= '1';
                                else
                                    habilita2 <= '0';
                                end if;
                        end if;
                    end process decode_ender2;


END d_ender2;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- ************************************************************
-- registrador de deslocamento de entrada para a direita
-- ************************************************************


ENTITY reg_des IS
    PORT(ConfigLinkIn       :       in std_logic;
         clk                :       in std_logic;
         load               :       in std_logic;
         enable             :       in std_logic;
         data               :       out std_logic_vector (7 downto 0));
END reg_des;

ARCHITECTURE reg_des OF reg_des IS

BEGIN
Reg_deslocamento:   Process(clk)
                    variable Sa,Sb,Sc,Sd,Se,Sf,Sg,Sh  : std_logic;
                    begin
                        IF (clk'event and clk = '1')
                        THEN IF load = '1'
                             THEN
                                Sh := Sg  ;
                                Sg := Sf  ;
                                Sf := Se  ;
                                Se := Sd  ;
                                Sd := Sc  ;
                                Sc := Sb  ;
                                Sb := Sa  ;
                                Sa := ConfigLinkIn ;
                             END IF; -- fim do load
                             IF enable = '1'
                             THEN
                                data(0) <= Sh;
                                data(1) <= Sg;
                                data(2) <= Sf;
                                data(3) <= Se;
                                data(4) <= Sd;
                                data(5) <= Sc;
                                data(6) <= Sb;
                                data(7) <= Sa;
                              END IF; -- fim do enable
                        END IF; -- fim do clk
                    End process reg_deslocamento;
END reg_des;
```