UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Geração de Processador para Aplicação Específica

por

MÁRCIO EDUARDO KREUTZ

Dissertação submetida à avaliação como requisito parcial para
a obtenção do grau de Mestre em
Ciência da Computação

Prof. Altamiro Amadeu Susin
Orientador

Porto Alegre, agosto de 1997

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Kreutz, Márcio Eduardo

Geração de Processador para Aplicação Específica / por Márcio Eduardo Kreutz. - Porto Alegre: CPGCC da UFRGS, 1997.

111f.: il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul. Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, BR - RS, 1997. Orientador: Suzim, Altamiro A.

1. Microcontrolador para aplicação específica. 2. Pipeline 3. Sistema Integrado para Aplicação Específica. 4. Otimização do conjunto de instruções do processador 5. Microcontrolador MCS8051. I. Suzim, Altamiro Amadeu. II. Título.

Agradecimentos

Gostaria de agradecer primeiramente a duas pessoas que tornaram possível a realização deste trabalho, seja pelo constante apoio e compreensão, seja pelas oportunidades oferecidas: Altamiro Suzim e Luigi Carro.

Em segundo lugar, gostaria de agradecer a todos os colegas de trabalho no grupo de microeletrônica, pela amizade e por proporcionarem um ambiente de trabalho agradável e construtivo; principalmente a Carlos Antonio Alba Pinto que gentilmente cedeu horas de trabalho em todos os momentos em que sua ajuda tornou-se imprescindível.

Agradeço também os professores da graduação que me incentivaram a seguir o caminho do mestrado, inclusive ajudando com as recomendações: Raul Ceretta (UFSM) e André Lemos (Unijuí).

Finalmente, agradeço ao Brasil pelo Instituto de Informática da UFRGS, com todos os seus recursos humanos e materiais, indispensáveis para a realização deste trabalho.

À todos, *VIELEN DANK !!!*

Sumário

Lista de Abreviaturas

ACC	Acumulador.
AHDL	Altera Hardware Description Language.
ASIC	Application Specific Integrated Circuit.
ASIP	Application Specific Instruction-Set Processor.
ASIS	Application Specific Integrated System.
C	Linguagem de programação C.
CCC51	C Cross Compiler to MCS8051.
CCC51 - I	CCC51 primeira otimização.
CCC51 - II	CCC51 segunda otimização.
CCC51 - III	CCC51 terceira otimização.
CPGCC	Curso de Pós-Graduação em Ciência da Computação.
FPGA	Field-Programmable Gate Arrays.
HDL	Hardware Description Language.
HW	Hardware.
MUX	Multiplexador.
PC	Program Counter.
PO	Parte Operativa.
PSW	Processor Status Word.
SIAE	Sistema Integrado para Aplicação Específica.
SP	Stack Pointer.
SW	Software.
RISC	Reduced Instruction-Set Computer.
UFRGS	Universidade Federal do Rio Grande do Sul.
ULA	Unidade Lógico-Aritmética.

VHDL	Very high speed integrated circuits Hardware Description Language.

Lista de Figuras

Lista de Tabelas

Resumo

Este trabalho propõe a geração de uma arquitetura dedicada a aplicações específicas, baseadas no microcontrolador MCS8051. Por ser utilizado na solução de problemas em indústrias locais, este processador foi escolhido para servir como base em um sistema dedicado. O 8051 dedicado gerado deverá permitir a integração completa do sistema, proporcionando um aumento do valor agregado e, conseqüentemente, a diminuição do custo.

Busca-se com a otimização da arquitetura obter um conjunto de instruções reduzido, construído com as instruções mais utilizadas em cada aplicação. O objetivo principal da otimização do conjunto de instruções está relacionado ao fato de que os circuitos decodificadores e geradores de microcódigo da parte de controle ocupam uma área significativa do processador. Uma otimização no sentido de reduzir-se o conjunto de instruções, portanto, resulta numa economia de área, o que vem de encontro com a idéia da integração completa do sistema com o processador.

Um processador dedicado a aplicações específicas (ASIP) irá possuir um custo maior do que a sua versão original, devido as otimizações realizadas. Para compensar este custo, uma alternativa a seguir é a integração completa do sistema. Um Sistema Integrado para Aplicações Específicas (SIAE) torna-se desejável, pois aumentando o valor agregado do circuito possibilita-se a redução do custo pela eliminação de conexões da placa, do encapsulamento de outros circuitos, entre outros motivos. Todavia, para que um SIAE possa ser construído com um custo aceitável, é necessário que seja construído em uma área que não exceda muito a área original do processador. Tenta-se fazer isto neste trabalho, através da implementação de aplicações com poucas instruções diferentes.

Por ser uma arquitetura comercial, o 8051 possui um grande parque de *software* desenvolvido e resolvendo problemas. Isto pode ser considerado uma vantagem pois, *software* básicos como por exemplo, compiladores, já estão desenvolvidos. Outra vantagem é o grande número de engenheiros treinados na sua utilização. Desse modo, torna-se necessária a criação de uma compatibilidade de *software*, para preservar o que já está desenvolvido. Uma vez que a programação em nível de linguagem montadora tende a constituir-se em uma tarefa cansativa e sujeita a erros, é desejável que se tenha uma compatibilidade em alto nível, ou seja, através de um compilador.

Para criar a compatibilidade de SW necessária é realizada a otimização de um compilador C desenvolvido para o 8051. A escolha pela linguagem C deve-se ao fato de sua grande utilização. O compilador C otimizado procura utilizar um conjunto de instruções reduzido para obter a economia de área. Quando uma instrução necessita ser utilizada e não está presente no conjunto de instruções desejado, o compilador tenta substituí-la por outra(s). Um conjunto de instruções é utilizado para cada aplicação, sendo constituído pelas instruções mais utilizadas por esta. Para determinar as instruções mais utilizadas de cada aplicação é realizada uma análise estática sobre um código em linguagem montadora previamente compilado. As instruções implementadas serão sempre parte do conjunto de instruções original do 8051, de modo que novas instruções não serão criadas.

Um programa em linguagem montadora gerado com um conjunto de instruções reduzido (RISC) normalmente terá um número maior de instruções do que o seu

equivalente com o conjunto de instruções completo (CISC). Isto ocorre porque possivelmente algumas substituições de uma instrução por outras, terão que ser realizadas. Como as instruções que serão utilizadas nas substituições pertencem ao conjunto de instruções original, o programa gerado com o compilador otimizado poderá executar em um tempo maior do que se fosse compilado com o código CISC. Para compensar esse atraso foi implementado um *pipeline* de instruções para o 8051.

Este trabalho apresenta resultados da Síntese Lógica em *Standard Cell* e FPGA da arquitetura otimizada. Além disso, resultados de programas em linguagem montadora gerados com o compilador otimizado, são também apresentados.

Palavras-chave: Microcontrolador para aplicação específica, Otimização do conjunto de instruções do processador, Pipeline, Sistema Integrado para Aplicação Específica, Microcontrolador MCS8051.

Title: "**Application Specific Processor Generation**"

Abstract

This work discusses a processor for specific applications architecture, based on the MCS8051 microcontroller. This processor is used as a solution for many local industry applications, being the base of dedicated systems. The dedicated 8051 generated should allow complete integration of the system, and with the added value to the chip, reduced costs.

The architecture optimization will produce as result a reduced instruction set, made by the often used instructions for each application. The main instruction set optimization goal refers to the instrucions decoders and microcode generators in the control part, because a large area in the processor is needed to implement them. Thus, a reduced instruction set will allow area savings, making possible the complete system integration in a chip.

An ASIP architecture will have a higher cost than the original one. An alternative to solve this problem is add value to the chip, creating an Application Specific Integrated System (ASIS). An ASIS can be made with a acceptable cost, if it's possible to integrate other circuits to the chip without area increase. This can be done in the area saved by using fewer implemented instructions.

Because the 8051 is a commercial architecture, there is a large amount of software developed for it. This can be considered an advantage because basic softwares like compilers are available, being not necessary to create them. Another advantage refers to the large number of engineers trained to use the 8051. To preserve the already developed applications it's necessary to mantain software compatibility. Assembler level programming is very boring an error prone task, being desirable to have software compatibility at higher levels through the use of high level languages.

To create the necessary SW compatibility, a C compiler developed for 8051 was optimized. The chose for C language refers to its large utilization. The optimized C compiler tries to use a reduced instruction set, formed with the most important instructions for each application, in order ro save area. When an instruction needs to be used in an application, and it's not present in the instruction set, the compiler tries to replace it with other instructions. The compiler will not use instructions not present in the original 8051 instruction set. So, new instrucions will be not created. To create an instruction set formed with the most important instructions for each application, a static analysis is made on a precompiled assembler source.

An assembler source generated with a reduced instruction set (RISC) will probably have more instructions than the same assembler generated with a full instruction set (CISC). This can be explained because of the replacements instruction. If one instruction is replaced by other two, and these are from the original instruction set, probably the time needed to execute them would be higher. In order to deal with this problem, an instruction pipeline was implemented to the 8051.

This work presents Standard Cells and FPGA results of Logic Synthesis of the optimized architecture. Also, assembly programs generated by the optimized compiler are presented.

Keywords: Application specific microcontrollers, Core Processor Instruction set optimization, Pipeline, Application Specific Integrated System, MCS8051 Microcontroller.

1 Introdução

Microcontroladores são Circuitos Integrados que implementam em uma pastílha de Silício um Sistema Computacional completo. Integram um microcontrolador: Unidade Central de Processamento, memórias RAM e ROM, portas de entrada e saída, temporizadores, etc. As características deste tipo de circuito, conferem-lhe funcionalidades largamente utilizadas na indústria para diversas aplicações, dentre as quais: impressora de cheques, controle de motores, controle de barramento de chão de fábrica, etc. Como um destaque na Indústria brasileira, pode-se citar o Microcontrolador Intel MCS-8051. O projeto deste chip data dos anos 80. Face o vertiginoso avanço tecnológico das áreas de informática e microeletrônica pode ser considerado um projeto desatualizado. Mesmo assim, este componente ainda continua sendo largamente utilizado na indústria nacional, pois atende muito bem a uma grande variedade de aplicações, além de possuir um custo muito baixo, devido ao projeto ter sido amortizado. Isto garante uma relação custo/benefício satisfatória.

O baixo custo do componente, aliado ao grande investimento em bases de desenvolvimento viabiliza pensar em transformações a serem feitas em sua arquitetura original, visando adaptá-lo melhor ainda às aplicações a que se destina. Deve-se ressaltar que mudanças feitas na arquitetura não devem implicar em um custo muito elevado, pois nesse caso estaría-se elevando a uma condição pior, a relação custo/benefício da arquitetura sendo utilizada no momento. Assim, um objetivo a ser conquistado pode ser a adaptação do 8051 para trazer mais vantagens na execução de certas aplicações, mantendo-se um baixo custo. Modificações que se podem fazer em uma arquitetura podem ser mudanças no conjunto de instruções, na Parte Operativa, na temporização, etc.

Para fazer uma modificação é necessitário antes verificar as características arquiteturais do Processador. Uma característica importante deste circuito, é que se constitue numa arquitetura CISC (*Complex Instruction Set Computer*), ou seja, possui um conjunto de instruções grande, constituído por instruções complexas. Quando forem feitas adaptações na arquitetura de um processador em relação ao seu conjunto de instruções, é bastante interessante que também se façam alterações nos compiladores desenvolvidos para esta arquitetura. Isto é importante para que se mantenha a compatibilidade com o *software* já desenvolvido. Para processadores CISC como o 8051, uma mudança no Compilador será muito onerosa, tendo em vista o grande número de instruções a serem gerenciadas por este compilador. Esta característica dificulta as modificações necessárias quanto a compatibilidade de *software* nas arquiteturas CISC.

Outra característica que deve ser levada em consideração é que em arquiteturas CISC, a Parte de Controle ocupa a maior parte da área do chip [CAR 96a]. Isto ocorre porque, com um conjunto muito grande de instruções complexas, tem-se também como consequência um grande número de decodificadores ou um decodificador de grande complexidade, além de um grande número de estados para o circuito de geração das microoperações e sequencialização destas. Para arquiteturas CISC dessa natureza pode-se utilizar uma memória ROM para o armazenamento do microcódigo. Esta técnica facilita a tarefa de sequencialização das operações, mas impõe como penalidade uma grande área e uma velocidade menor de operação.

Trabalhos realizados no IEE-UFRGS [CAR 96a] aplicaram programas de análise de execução de instruções para certos programas de aplicações específicas que executam sobre o 8051. Com base em uma análise estática e dinâmica da execução destes programas, verificou-se que um número pequeno de instruções era suficiente para executar quase que a totalidade daquelas aplicações, tendo por base o conjunto de instruções completo do 8051. Como resultado estatístico dessas análises, tem-se que 17 instruções eram suficientes para executar aproximadamente 82% do código das aplicações. Esses resultados vieram a afirmar a possibilidade de se adaptar o 8051 para uma arquitetura RISC. Dentro da filosofia RISC temos apenas um conjunto pequeno de instruções implementadas. Neste caso, a decodificação torna-se mais simples, podendo ser realizada diretamente em *hardware* através de uma Parte de Validação.

Entretando ressalta-se que, dentro do contexto aqui abordado, o processador a ser gerado não possui todas as características de uma arquitetura RISC pura, como por exemplo, instruções que executam em apenas um ciclo de relógio, orientação a registrador, acesso à memória através de instruções *load* e store, etc. Para esta abordagem estão incluídas instruções que acessam a memória (não sendo orientadas a registrador) e executam em mais de um ciclo. Portanto, dentro do contexto deste trabalho, a classificação RISC, refere-se a uma arquitetura composta de um conjunto de instruções reduzido e com *pipeline*. Além disto, pode-se citar que a maioria das instruções originais do 8051 são relativamente simples e portanto não demandam por circuitos decodificadores e geradores de microcódigo complexos, como por exemplo, os encontrados nas máquinas CISC atuais. Sendo assim, as instruções implementadas para a arquitetura a ser gerada podem ser decodificadas por *hardware* (*hardwired*), o que vem a constituir-se de uma importante característica das máquinas RISC.

A Parte de Validação de uma máquina RISC constitue-se em um pequeno decodificador para cada instrução que se deseja implementar na arquitetura, trabalhando em conjunto com a máquina de estados para gerar a sequência de microoperações necessárias à execução de cada instrução. Estas microoperações são enviadas diretamente para a Parte Operativa. A Parte de Validação implementada como descrito acima permite que a inserção/retirada de uma instrução seja feita de maneira muito simples. Uma descrição VHDL pode ser utilizada para gerar diferentes conjuntos de instruções.

Com as modificações explicadas nos parágrafos acima pode-se chegar a duas conclusões, verificadas em [CAR 96a]:

> com a arquitetura RISC (um conjunto pequeno de instruções implementadas) tem-se uma grande diminuição na área e no consumo do chip, através da diminuição da Parte de Controle; e
>
> existe a possibilidade de modificar o conjunto de instruções de uma aplicação para outra.

Tem-se assim, um processador adaptado para uma aplicação específica, ou seja, um ASIP (*Application Specific Instruction-Set Processor*). Estas características permitem uma grande flexibilidade em termos de adaptação do MCS8051 para aplicações específicas. Dessa forma modifica-se o conjunto de instruções do 8051, a

fim de que este seja o menor possível, para economizar área, e adaptado à uma aplicação específica.

Como se pode observar em processadores RISC, o tamanho do código em linguagem montadora tende a ser maior do que o mesmo código escrito com um conjunto complexo de instruções (CISC). Esta característica provem do fato de que cada uma das instruções complexas presentes no código CISC, deverá ser representada ou emulada por 2 ou mais instruções simples, características do código RISC. Com isso, um programa RISC normalmente ocupa mais espaço em memória. Porém, em uma máquina RISC típica, as instruções são simples o suficiente para serem decodificadas por *hardware* (*hardwired*) e o seu tempo de execução é pequeno, de forma que o tempo total de execução de um programa equivalente pode até ser menor. Para o caso de processadores para aplicações específicas em que implementam-se subconjuntos (específicos à cada aplicação) do conjunto de instruções total igualmente pode haver necessidade de uma emulação. Neste caso, as instruções implementadas e específicas da aplicação deverão emular as outras instruções. A emulação normalmente necessita de 2 ou mais instruções para substituir a instrução que se deseja emular. Neste caso, as instruções que emularão outras instruções, não serão necessariamente mais simples do que a instrução emulada. Desta maneira, o código não apenas ficará maior, como também terá um tempo de execução maior. Para o caso do 8051 CISC, o tempo de execução das instruções é semelhante, pois poucas instruções são muito complexas. Assim, por exemplo, no caso em que tem-se uma substituição de uma instrução por duas, o tempo de execução destas será em torno do dobro do tempo de execução da instrução que foi substituída.

Para compensar este atraso, pensou-se em implementar para o 8051 alguma técnica de paralelismo para acelerar o tempo de execução das instruções. Dentre as técnicas de paralelismo existente, como por exemplo sistemas multiprocessados ou matriciais, optou-se pelo *pipeline*, devido ao fato de este apresentar uma melhor adequação em relação a um dos objetivos, que é a manutenção do baixo custo.

Um dos objetivos a serem alcançados é que as modificações que serão efetuadas sobre o 8051 não aumentem o custo do processador de maneira significativa. Se isto ocorrer, vê-se que provavelmente a relação custo/benefício que tem-se hoje, não será mais tão favorável. Em outras palavras, o custo do processador poderá não justificar a sua substituição do original para o modelo modificado. Técnicas de paralelismo como por exemplo, multiprocessadores, não poderiam ser implementados sem um aumento significativo de custo, devido à replicação de *hardware* e dos barramentos. Além do mais, estas instruções deveriam executar em paralelo, para ocuparem o mesmo tempo que levariam para executar a instrução que estas estão substituindo (emulando). Em muitos casos não é possível por causa da dependência entre os seus operandos. É claro que dependendo da aplicação poderia-se ter partes de um programa executando em processadores replicados, o que pode resultar uma aceleração considerável. Em segundo lugar, é necessário que se respeite as características originais da arquitetura para manter-se uma compatibilidade com o processador original e manter o baixo custo. Isto significa que não podemos por exemplo, implementar mais de uma ULA no 8051. Embora o custo de um ASIP sempre seja maior do que o processador original, as características desse novo processador devem compensar esse custo adicional, aumentando o valor agregado chip pela integração do sistema em um único chip.

A técnica de *pipeline* foi a técnica de paralelismo que mais adequou-se aos objetivos propostos, proporcionando uma aceleração de execução das instruções sem a necessidade de adição de *hardware* significativo. A Parte Operativa permanece praticamente a mesma. As modificações que são necessárias para implementar o pipeline dizem respeito a inclusão de registradores e alterações da Parte de Controle, pois tem-se uma máquina de estados modificada e microoperações executando em paralelo. As modificações na Parte de Controle implicam em uma pequena adição de *hardware*. Assim, com o pipeline, tem-se uma aceleração na execução das instruções, a um custo aceitável. Com o pipeline implementado no 8051 será possível compensar o aumento do tempo de execução dos programas otimizados para aplicações específicas.

No momento em que se tem um microcontrolador MCS8051 RISC com *pipeline* adaptado a aplicações específicas, é necessário que se proporcione também uma compatibilidade de SW, principalmente para que se possa aproveitar todo o parque de SW já instalado. Uma compatibilidade de SW deve ser feita no Compilador do processador, já que foi modificado o seu conjunto de instruções. Se não for adaptado um Compilador para esse novo conjunto de instrucões definido, seria necessário que se programasse o processador através de linguagem Montadora, ou *Assembly*. A própria linguagem montadora deveria ser modificada para não aceitar as instruções não implementadas. Assim, é conveniente que se mantenha a compatibilidade de *software* através da adaptação do Compilador para cada diferente conjunto de instruções do processador. Isto implicará em dois benefícios básicos:

facilidade para o desenvolvimento de novas aplicações; e

compatibilidade com os programas já desenvolvidos para o processador original.

Devido a sua grande popularidade, foi escolhido o Compilador da linguagem C para proporcionar uma compatibilidade de SW com o MCS8051 RISC pipeline. Primeiramente foi desenvolvido no CPGCC-UFRGS um Compilador C para o 8051 [MED 91]. Este compilador foi desenvolvido para gerar um código Assembly para o conjunto completo de instruções do 8051. Durante a compilação do código fonte em C, após as análises léxica, sintática e semântica é gerado um código intermediário. A partir desse código intermediário podem ser gerados diferentes códigos em linguagem *Assembly*, ou até em outras linguagens. Dentro do contexto deste trabalho está-se tratando o 8051 como um ASIP, com diferentes conjuntos de instruções, um para cada aplicação. O conjunto de instruções de cada aplicação deverá constituir-se de poucas instruções para salvar área na Parte de Validação. Tendo por base este fato, fica claro neste contexto a importância do Compilador. Este terá que, para cada aplicação em questão, gerar um código *assembly* do 8051 contendo apenas as instruções definidas para cada aplicação. Na verdade será papel do Compilador definir o conjunto de instruções que será implementado. As instruções em que não se conseguiu emular, também farão parte do conjunto de instruções desta aplicação. Poderão existir instruções que não executam em um ciclo normal do *pipeline*, quebrando-o. Estas instruções deverão ser evitadas ao máximo, principalmente se tiverem um percentual grande de utilização. Os objetivos do Compilador são:

gerar um código em linguagem montadora que utilize um conjunto de instruções reduzido;

emular as instruções pouco utilizadas. Se isto não for possível, incluí-las no conjunto de instruções; e

evitar o uso das instruções que quebram o *pipeline*.

É importante ressaltar que quanto menor for o conjunto de instruções implementado, menor será a Parte de Validação, e por conseguinte, menor a área e o consumo do chip.

Uma vez gerado o código em linguagem montadora e definido o conjunto de instruções da aplicação, este poderá ser implementado em HW, colocando-se estas instruções na Parte da Validação da descrição do 8051 em VHDL. Finalmente, através de ferramentas de síntese automática, pode-se sintetizar a descrição e prototipar o circuito em FPGA. As vantagens de se prototipar em FPGA dizem respeito ao baixo custo e velocidade. Desse modo, pode-se ter rapidamente um protótipo de um MCS8051 adaptado à uma aplicação específica.

O fato de ter-se poucas instruções implementadas resulta em um grande ganho de área em relação ao processador original. Esta área poderá ser utilizada para implementar outros circuitos, como por exemplo protocolos de comunicação, memória RAM, etc., aumentando o valor agregado do chip [CAR 96]. Desse modo, numa mesma área é possível ter-se um circuito que realize mais tarefas, trazendo um maior benefício ao usuário. Em última análise, estaria-se desenvolvendo o conceito de um ASIS (*Application Specific Integrated System* - Sistema Integrado para Aplicação Específica) ou seja, um sistema computacional completo *on chip*.

1.1 Contribuições

O presente trabalho diferencia-se da literatura atual em termos de pesquisa sobre ASIPS, pela geração de um sistema microcontrolado adaptado a aplicações específicas baseado no 8051 e compatível com as aplicações existentes. O escopo deste trabalho situa-se na otimização do 8051 para operar com um conjunto de instruções reduzido e específico a cada aplicação, onde existe a compatibilidade com os programas já escritos em C, obtendo-se assim, a compatibilidade de *software* em alto nível. Esta compatibilidade é requerida por tratar-se o 8051 de uma arquitetura de uso comercial e portanto largamente utilizada na indústria local e nacional. Modificações já foram realizadas sobre o 8051 em relação à redução do seu conjunto de instruções e programas de análise estática das instruções executadas em cada aplicação. A novidade deste trabalho é a otimização da velocidade de execução das instruções através da técnica de *pipeline*, e a geração de um conjunto de instruções reduzido e compatível com o *pipeline*, através de um compilador C.

Desta forma, não é necessário que se desenvolvam novas aplicações, compiladores etc., pois todo o parque de *software* desenvolvido será compatível, bastanto recompilar o programa. Com isso será possível gerar-se um novo 8051 com um conjunto de instruções menor, adaptado a uma aplicação específica. Como consequência tem-se uma economia de área, por ter-se implementadas poucas instruções. Essa área pode ser utilizada para a integração do sistema. O 8051 otimizado pode ser prototipado em FPGA.

Implementando-se as aplicações com um conjunto de instruções reduzido (RISC), a tendência é que os programas em linguagem montadora venham a tornar-se maior. Mesmo tendo programas RISC um maior número de instruções, possivelmente, devido ao *pipeline*, o tempo de execução poderá ser menor.

Através de uma prototipação rápida e a um baixo custo (FPGA), as aplicações atuais poderão passar a executar sobre esse novo processador. Isto permite uma migração rápida das aplicações para essa nova abordagem do 8051. O Compilador C garante a compatibilidade de *software* com o que já foi desenvolvido permitindo uma migração rápida para circuitos integrados dedicados.

Após a prototipação tem-se um processador com área e consumo de potência menor. Além disso, com a área excedente em relação a arquitetura original, pode-se implementar outros circuitos (como funções implementadas em *hardware*) aumentando o valor agregado do microcontrolador. A arquitetura permanece praticamente a mesma, pois a inclusão do pipeline não implica em muitas modificações quanto ao *hardware*, além de não serem criadas novas instruções. As instruções utilizadas são as mesmas da arquitetura original, onde dentre estas são escolhidos subconjuntos do conjunto total de instruções. Cada subconjunto irá corresponder a uma aplicação. Isto mantém a compatibilidade com Montadores já desenvolvidos para o MCS8051. Finalmente, pode-se observar que, para aplicações desenvolvidas em C, o número de instruções utilizadas é pequeno pois os compiladores em geral não utilizam todo o repertório de instruções disponível no processador.

1.2 Organização da Dissertação

A dissertação está organizada da seguinte forma:

O capítulo 2 refere-se a sistemas para a geração de ASIPs encontrados na literatura, realizando uma comparação com a abordagem para aplicações específicas proposta para o 8051.

O capítulo 3 apresenta alguns conceitos de arquitetura de computadores no que diz respeito a processadores para aplicações específicas. Ainda neste capítulo é mostrada e analisada a arquitetura do 8051, bem como a sua descrição em VHDL.

O capítulo 4 realiza uma discussão sobre alternativas de implementação do pipeline para esta arquitetura, explicitando e justificando a maneira como esta foi realizada. Segue-se a análise do desempenho e as implicações da implementação do pipeline para 8051.

No capítulo 5 é realizado processo de Síntese Lógica para a descrição VHDL do 8051 em duas versões: com e sem pipeline. Comparações são feitas entre estas duas descrições em termos dos resultados obtidos com a síntese. Esses resultados são úteis para a validação do pipeline.

O capítulo 6 trata do Compilador C desenvolvido para o 8051. Neste capítulo são mostradas as características do Compilador e as modificações necessárias para adaptá-lo à geração de instruções específicas a cada aplicação. Neste capítulo também

é tratada a geração e prototipação do circuito tendo como entrada o programa em C, e como saída a prototipação em FPGA. Alguns exemplos são analisados e testados.

O capítulo 7 apresenta conclusões e as propostas de trabalhos futuros.

2 A abordagem ASIP para o MCS8051

Os microprocessadores oferecem uma solução flexível e de baixo custo para sistemas que trabalham com algoritmos complexos, como no processamento digital de sinais (DSP) e em controle digital ou as que trabalham com tempo real. Sistemas baseados em microprocessadores são de natureza preponderantemente eletrônica e digital. Esses sistemas aplicam-se a uma larga gama de problemas desde que sejam programados para tal. Estruturalmente estes sistemas são organizados através de um ou mais processadores (capazes de executar instruções), memória e periféricos (ASICs, ...). Todos esses componentes atuam em conjunto para a execução de programas que implementam uma ampla variedade de aplicações.

Em relação a aplicações específicas os sistemas microprocessados apresentam algumas desvantagens que dizem respeito principalmente a área, consumo e desempenho se comparado aos sistemas de *hardware* específico. Isto porque os sistemas adaptados para aplicações específicas possuem apenas o HW e o conjunto de instruções necessários à execução daquela aplicação. Portanto, Sistemas Microprocessados para Aplicações Específicas são a solução quando os sistemas centrados em microprocessador de propósito geral não podem cumprir algum dos requisitos de desempenho, de portabilidade, de menor área ou de consumo [ALB 96].

Microcontroladores são sistemas microprocessados que integram, em apenas uma pastilha de silício, processador, memórias, portas de entrada/saída, temporizadores, etc, sendo utilizados para aplicações gerais. A adaptação de microcontroladores para aplicações específicas, com o objetivo de conseguir um melhor desempenho, consiste basicamente na adequção do processador às características da aplicação. Uma maneira de conseguir isto é através da modificação do conjunto de instruções, tornando-o um processador com conjunto de instruções adaptado a uma aplicação específica. Dessa maneira, o conjunto de instruções desse processador será formado pelas instruções mais importantes, ou aquelas mais utilizadas pela aplicação.

Como pode ser visto em [MIC 94], dependendo do estilo de projeto adotado, tem-se diferentes soluções para a execução das aplicações. Uma abordagem seria a concepção de um ASIC (Application Specific Integrated Circuit), como sendo uma solução orientada ao hardware. Para essa abordagem é construído um circuito que implementa um algoritmo. Este procedimento constitui-se na solução mais eficiente, devido à execução do algoritmo em *hardware*, porém demanda um longo tempo de projeto e um custo considerável, por tratarem-se normalmente de circuitos complexos.

Como um outro estilo de projeto, tem-se a abordagem ASIP (Application Specific Instruction-set Processor), onde se tem um processador capaz de ajustar-se à execução de aplicações específicas, através da modificação de seu conjunto de instruções e de sua arquitetura. Assim, para cada aplicação serão definidas as instruções que melhor se adaptam a sua realização, sendo estas implementadas no processador. A arquitetura da Parte Operativa será escolhida de modo a implementar da melhor maneira possível o conjunto de instruções, normalmente com base em certas restrições como por exemplo, a área ocupada pelo circuito, o consumo de potência, etc. Um ASIP possui um alto desempenho, pois a sua arquitetura é projetada e ajustada de acordo com as características da aplicação. O tempo de projeto de um

ASIP tende a ser maior do que tempo necessário para o projeto do mesmo sistema utilizando o processador padrão. Isto tende a diminuir, entretanto, com a crescente automatização. Um ASIP necessita de um CAD de suporte bem elaborado, para permitir a migração de um aplicação para outra. A compatibilidade de software, especialmente do Compilador do ASIP é importante pois permite a programação da nova arquitetura em alto nível sem a necessidade de alteração na plataforma de desenvolvimento.

Como uma terceira alternativa de projeto, tem-se um processador central com HW adicional, resultante do processo *Hardware/Software Codesign*. Essa abordagem resulta num desempenho relativamente inferior aos ASIPs, mas mantém uma programabilidade muito boa, bem como um custo de projeto médio. Pelo fato de o HW adicional constituir normalmente uma função da aplicação, no caso de esta ser uma função complexa, praticamente estar-se-á implementando um ASIC com um tempo de projeto proporcional.

Finalmente, tem-se a abordagem orientada ao SW. Nessa abordagem, a execução da aplicação se realizará através da execução de um programa num processador de propósito geral. Assim, obtem-se como vantagens principais a versatilidade e a programabilidade, pois esta máquina pode resolver um grande número de aplicações. O tempo de projeto para essa abordagem é menor em relação às outras, porém com um desempenho inferior e possivelmente maior consumo.

Dentro do contexto deste trabalho, está-se utilizando a abordagem ASIP, porém com uma arquitetura já existente. Desse modo, para manter um baixo custo, as alterações efetuadas para a migração entre as aplicações restringir-se-ão a alteração do conjunto de instruções, sem modificações na Parte Operativa. Assim, a vantagem de se optar por essa abordagem confirma fato constatado em [CAR 96b], onde a implementação de apenas algumas instruções do conjunto total das instruções especificadas originalmente resulta em uma grande economia de área, através da diminuição da Parte de Validação. Essa área excedente pode ser utilizada para aumentar o valor agregado do processador [CAR 96], possibilitando a realização do "Sistema Integrado para Aplicações Específicas" (SIAE) ou *"Application Specific Integrated System"* (ASIS).

2.1 O Projeto de ASIPs

Para o projeto de um ASIP, existem duas abordagens possíveis:

a criação de uma arquitetura nova; e

adaptação de um processador já existente.

A figura 2.1 apresenta essas duas abordagens.

Na primeira abordagem, uma arquitetura para uma determinada aplicação é criada a partir da análise de processos escritos em linguagem de alto nível. Esses processos são traduzidos para estruturas do tipo grafos de fluxo de controle, para a verificação de como são realizadas as instruções em nível de microoperações. Em seguida são realizadas análises sobre essas estruturas para saber, por exemplo quais são as instruções mais utilizadas pela aplicação, o custo em termos de HW para sua

execução, as restrições (área, consumo) a serem obedecidas, etc. A partir dos resultados desta análise, é escolhido o conjunto de instruções que melhor atende a aplicação, composto pelas instruções mais utilizadas e uma arquitetura adequada, bem como a lógica de controle para a execução dessas instruções. A especificação da arquitetura consiste normalmente de uma seleção de elementos de HW pré-definidos. Para a programação com o conjunto de instruções definida, é criada uma linguagem montadora, e em alguns casos, um compilador, permitindo a programação em alto nível. A nova arquitetura ASIP é descrita através de uma HDL (*Hardware Description Language*), para posterior síntese.

Em relação ao que foi encontrado na literatura, verifica-se que a maioria desses trabalhos adotam este tipo de abordagem para a especificação da arquitetura do ASIP.

Isto pode ser verificado por exemplo, em [GOV 96], que permite o projeto simultâneo de um HW com pipeline e do SW para executar sobre este pipeline. Em [NGU 95], propõe-se um algoritmo para particionamento de HW/SW codesign para selecionar uma boa arquitetura para o pipeline. Procura-se achar uma série de operações implementadas em HW para obter um bom desempenho para o ASIP pipeline, levando-se em consideração o número de portas (*gate count*) e consumo de potência como limitações. Nota-se que se tratam de arquiteturas genéricas criadas automaticamente. Assim, são arquiteturas novas, tornando-se necessário que se desenvolvam novos compiladores e SW para torná-la compativel comercialmente. Isto pode ter um grande custo e também um grande tempo de chegada ao mercado. Em [VAN 94] é apresentada uma abordagem interativa para definição de microinstruções otimizadas dos ASIPs. Também apresenta um método para definição das instruções quando for gerado o código para o ASIP. Então é gerado o modelo da parte operativa com o seu conjunto de instruções onde a aplicação é mapeada. Neste caso, temos um *data-path* específico e otimizado e um modelo da nova arquitetura. Uma abordagem semelhante é encontrada em [IMA 92], onde a definição de microinstruções otimizadas de ASIPs é feita interativamente.

Em [HUA 94a] verifica-se que o projeto de um sistema de um ASIP inclui pelo menos três tarefas independentes:

1. projeto da microarquitetura;

1. projeto do conjunto de instruções; e

1. mapeamento do conjunto de instruções da aplicação.

Em [HUA 94a] é apresentado um método que unifica estes três problemas de projeto com uma única formulação: um problema de escalonamento/alocação agregado a um processo de formação de instruções integrado. As microoperações que representam a aplicação são escalonadas em passos de tempo. As instruções são formadas e os recursos de HW são alocados durante o processo de escalonamento. Ao fim do processo de escalonamento é obtido automaticamento o código em linguagem montadora da aplicação. Esta abordagem considera o paralelismo entre as microoperações, codificação das instruções, o atraso de *load/store/branch*, execução condicional das microoperações e vários modelos de arquiteturas. Nota-se que ao final do processo é gerado um código em linguagem montadora e não um compilador.

Ainda sob o enfoque da criação de arquiteturas novas dedicadas, em [ALO 93] verifica-se a implementação e experimentos com o PEAS - I, um sistema de HW/SW co-design para geração de ASIPs. Este sistema possue como entrada um conjunto de programas (aplicações) escritos em C associado um conjunto de dados e as limitações do projeto, como área e consumo. A partir dessas variáveis, o sistema gera um modelo de CPU descrito numa HDL, um compilador C, um assembler e um simulador. Umas das contribuições desse artigo é a criação de um novo modelo para o projeto do conjunto de instruções através de uma abordagem que utiliza programação inteira. Além disso, verifica-se a preocupação com a compatibilidade de SW, através da geração do compilador e do assembler para a arquitetura gerada. Esta pode ser considerada uma semelhança com a abordagem aqui proposta.

A segunda abordagem para o projeto de um ASIP diz respeito a adaptação de um processador já existente. Neste enfoque realiza-se uma modificação sobre um processador, de modo a que se obtenha um melhor desempenho para a execução de determinada aplicação. Assim como na abordagem explicitada no parágrafo anterior, nesta abordagem igualmente são realizados testes sobre a aplicação de modo a determinar quais as instruções mais utilizadas. O passo seguinte consiste em adaptar-se o conjunto de instruções para as instruções que serão implementadas. Isto pode ser feito simplesmente pela retirada de algumas instruções, e/ou pela criação de outras. De qualquer maneira, as alterações no conjunto de instruções implicarão mudanças na Parte de Controle e na Parte de Validação. A otimização da Parte Operativa não se torna necessária para o caso e em que não haverá a criação de novas instruções, ou até mesmo, se as novas instruções são adaptadas ao *data-path* e aos elementos de HW da Parte Operativa. Porém, qualquer otimização que venha a ser feita, resultará numa melhor adequação do ASIP às restrições impostas quando da idéia de sua concepção. Ainda, uma modificação na Parte Operativa pode ser a inclusão de funções à serem executadas em HW. Caso o processador que será transformado em ASIP já possua um Compilador desenvolvido, para manter a compatibilidade de SW, torna-se necessário adaptar o Compilador para que este reconheça o novo conjunto de instruções, também para que se possa continuar utilizando o SW já desenvolvido para esta arquitetura.

Sistemas que trabalham sob essa abordagem são encontrados em [ATH 93] e [LIE 94]. Athanas [ATH 93] usa a informação extraída durante a compilação do programa de aplicação para a criação e seleção das instruções. Liem [LIE 94] usa uma representação de padrões de instrução que descrevem fluxos de dados, fluxos de controle e fluxos combinados de dados e controle do programa de aplicação e obtém as instruções pelo agrupamento desses padrões.

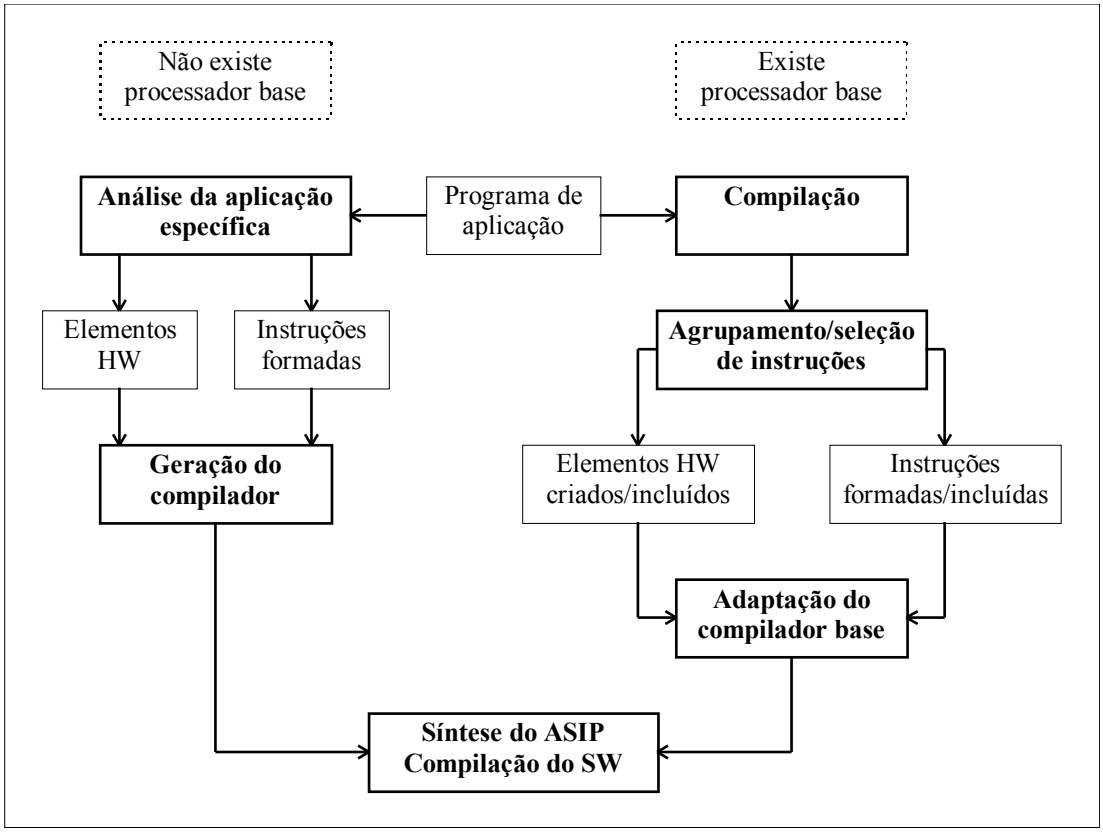


FIGURA 2.1 - O projeto de ASIPs

O projeto de transformação do 8051 em ASIP, pode ser enquadrado sob a segunda abordagem apresentada, pois está-se trabalhando com uma arquitetura existente. A maior diferença encontrada em relação a literatura, diz respeito ao fato de esta ser uma arquitetura de uso comercial. De certa forma isto constitui-se numa vantagem, pois para cada aplicação, basta apenas que se gere uma nova versão do processador, adaptada àquela aplicação. A vantagem verifica-se principalmente no número de engenheiros treinados, pois sempre que uma arquitetura nova é lançada torna-se necessário a construção (projeto e programação) de todo o software básico necessário ao suporte da nova arquitetura. A construção desses softwares básicos, como por exemplo compiladores, sistemas operacionais, por possuírem algoritmos não triviais, envolve uma tarefa relativamente complicada e principalmente demorada. Este problema não se verifica numa arquitetura já existente, pois para esse processador não apenas estão implementados todos esses *softwares*, como já foram largamente testados, sendo utilizados comercialmente [MED 91]. Além disso, existe um grande número de engenheiros e programadores treinados na sua utilização.

Outra característica importante diz respeito ao custo de produção da nova arquitetura. É interessante observar que, para uma arquitetura nova tornar-se barata, é necessário que se seja produzida em larga escala. O baixo custo é uma exigência, sendo que a escolha do 8051 para que este seja adaptado à aplicações específicas, deve-se também ao fato de esse processador ser produzido comercialmente há um longo tempo, e em larga escala. Como consequência, tem-se um baixo custo final para os consumidores. Dentro do contexto do mercado brasileiro, produzir uma arquitetura nova em larga escala pode ser uma tarefa complicada, principalmente pela carência de estrutura industrial em termos de *foundries*. Assim, uma maneira encontrada para trabalhar-se com uma arquitetura reconfigurável a um baixo custo, é a utilização de

um processador comercial, sob o ponto de vista de que esta arquitetura já possui um grande parque de SW instalado, além de engenheiros treinados na sua utilização. Em relação ao custo propriamente dito, ressalta-se que a arquitetura ASIP terá um custo mais elevado em relação ao seu equivalente comercial. Porém o que se busca para compensar este custo a mais é o aumento do valor agregado da arquitetura, pela inclusão de outras partes do sistema no mesmo chip do 8051 ASIP, ou seja, a integração do sistema para aplicação específica (ASIS).

Outras características deste trabalho dizem respeito a prototipação rápida e economia de área para aumentar o valor agregado do processador. A prototipação rápida verifica-se pelo fato de ser realizada em FPGA. A implementação utilizará a abordagem da Altera, como pode ser verificado no capítulo 5. Além disso, um FPGA permite a integração completa do sistema a um baixo custo.

As idéias que deram origem a este trabalho foram discutidas em trabalhos anteriores realizados no GME-UFRGS e podem ser encontradas em [CAR 96] e [ALB 96].

Em [CAR 96] encontram-se as idéias que deram origem a este trabalho, no sentido de se trabalhar com arquiteturas reconfiguráveis e dedicadas a aplicações específicas. Em [CAR 96] são mostradas diferentes formas de se adaptar uma arquitetura à melhor execução de determinados tipos de aplicações como por exemplo, aplicações que realizam diversas chamadas a sub-rotinas ou possuem um grande número de saltos. O presente trabalho constitui-se numa extensão destas propostas no sentido da realização de modificações na temporização e implementação de pipeline para arquiteturas comerciais, como é o caso do 8051, a fim de tornar estas arquiteturas dedicadas.

Em [ALB 96], trabalha-se com um gerador de microcontroladores para aplicações específcas MbSG (*Microcontroller based System Generator*). Este sistema está projetado para ser automático desde a análise de um programa na linguagem C, junto com requisitos do usuário, até a geração da descrição em VHDL da arquitetura do microcontrolador resultante. As análises realizadas no MbSG têm como objetivo otimizar a arquitetura de um processador chamado Risco [JUN 93] (que será a base do microcontrolador) adaptando-a à aplicação. Assim, são determinados os registradores usados pelo programa de entrada, as unidades funcionais e instruções necessárias para o programa de aplicação ser executado e serem considerados posteriormente na geração da arquitetura. O processo central do MbSG é a análise das rotinas do programa de aplicação. Essa análise consiste na execução simulada do programa, junto com os vetores de entrada fornecidos pelo usuário, para encontrar a rotina ou conjunto de rotinas críticas (com o maior número de ciclos de máquina necessários à sua execução) a serem otimizadas com uma arquitetura particular de acordo a sua característica básica (rotinas podem ser computacionalmente intensivas, intensivas em acessos à memória e intensivas em desvios do fluxo de controle do programa). A arquitetura gerada consiste em um processador Risco base, memórias de instruções (ROM e/ou WCS) memória de dados (RAM), interfaces com o exterior e *hardware* de suporte (rotinas críticas sintetizadas). Essa arquitetura é descrita em VHDL para logo ser sintetizada com as ferramentas do Sistema Alliance [BAZ 94].

Verifica-se que a intenção deste trabalho igualmente é a geração de uma arquitetura microcontroladora tendo como entrada um compilador C para o 8051,

adaptado a geração de instruções específicas de cada aplicação.. A diferença está no fato de que aqui não se está tratando da criação de uma arquitetura nova baseada na variação de um modelo, mas apenas na modificação, através da análise realizada em tempo de compilação, do subconjunto de instruções implementadas. Outra diferença é a própria prototipação em FPGA. Assim, neste trabalho, o objetivo maior é a economia de área pela implementação de poucas instruções e a rápida prototipação para facilitar a integração de sistemas [KRE 97].

2.2 O Processo de criação do MCS8051 ASIP

A figura 2.2 mostra as etapas necessárias para a adaptação do 8051 a uma determinada aplicação.

O processo inicia-se com a aplicação escrita em linguagem C. Num primeiro momento essa aplicação é compilada, utilizando-se qualquer compilador C definido para gerar instruções em linguagem montadora do 8051, inclusive o compilador C desenvolvido no GME-UFRGS para o 8051 [MED 91], sobre o qual este trabalho se baseia. O próprio compilador modificado terá uma opção para gerar todas as instruções. Em seguida, o programa em linguagem montadora gerado pelo compilador é submetido a uma ferramenta que realiza a análise de quais são as instruções mais utilizadas pela aplicação. Tanto o compilador C, quanto as ferramentas de análise são ferramentas comerciais. Neste sentido procura-se simular o ambiente encontrado pelos projetistas e engenheiros que dedicam-se a trabalhar com o 8051, como demonstrado em [CAR 96].

Seguindo o processo, tem-se então a atuação do compilador C modificado para gerar um conjunto de instruções reduzido e adaptado à aplicação. Uma possível abordagem a ser seguida, é a de que o compilador receba como entrada um arquivo contendo as instruções que são passíveis de implementação, ou seja, as mais utilizadas pela aplicação. Nota-se que esse arquivo contém somente instruções do 8051, sendo que nenhuma instrução nova será criada. O compilador, a partir do código intermediário, tenta implementar somente as instruções especificadas no arquivo de entrada. Uma regra a ser seguida pelo compilador diz respeito a que, quanto maior for a porcentagem de utilização da instrução, maior a necessidade de que esta não quebre o pipeline para sua execução. Se este for o caso, tenta-se substituí-la por outra(s) que possua(m) a mesma função sem quebrar o pipe. Por exemplo, a instrução INC A pode ser susbtituída por ADD A,#01. No caso de uma substituição não ser possível, a instrução será implementada.

Após a definição das instruções que serão implementadas, é gerada a Parte de Validação sob a forma de um arquivo VHDL comportamental. Às outras partes da descrição VHDL do 8051 (ver capítulo 3) é incorporada a Parte de Validação, tornando a descrição completa e pronta para ser sintetizada. Pode-se verificar aqui que não são realizadas modificações sobre a Parte Operativa, pois as mudanças que ocorrem dizem respeito somente ao conjunto de instruções, sendo que este será constituído de um subconjunto do conjunto de instruções completo do 8051. Nota-se ainda que esta descrição já está implementada com o pipeline. Detalhes sobre a implementação do pipeline para o 8051 são discutidos no capítulo 4.

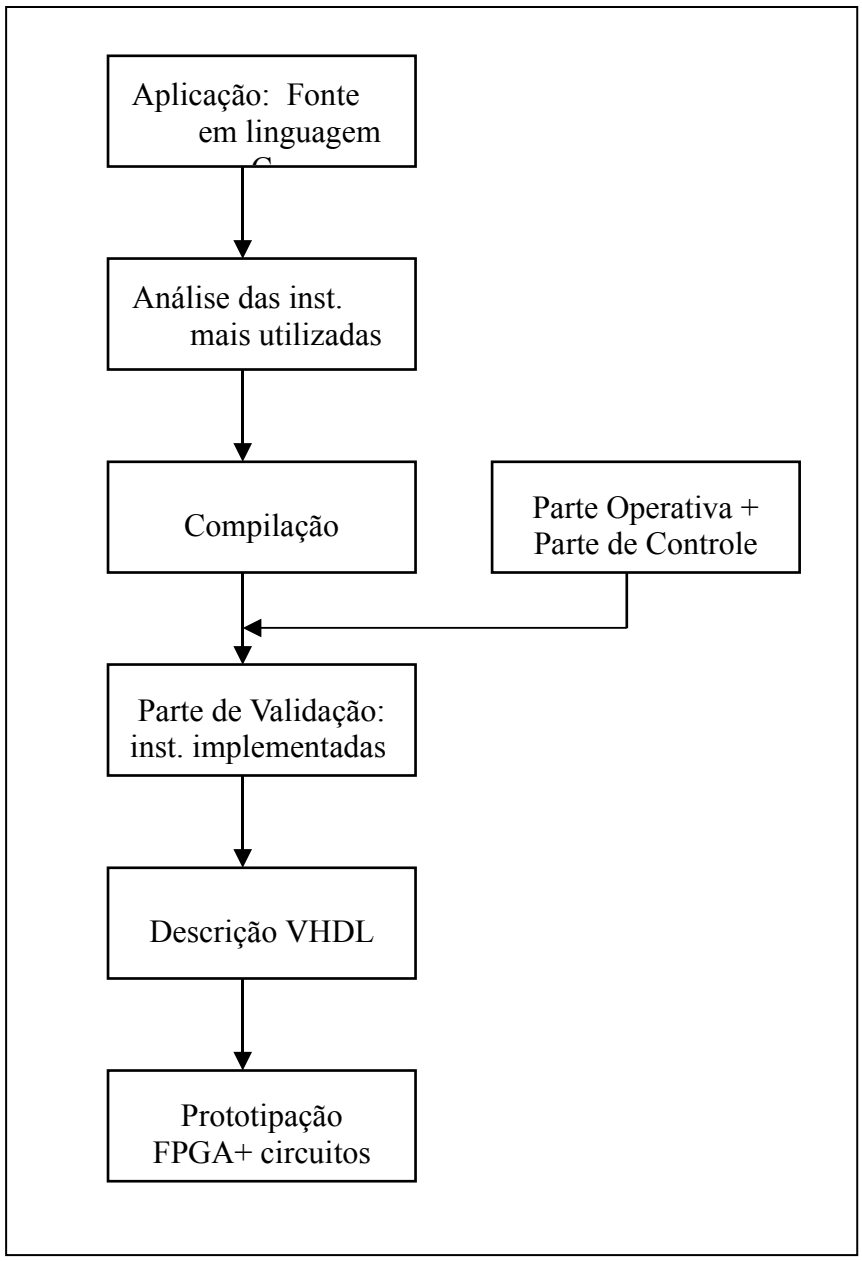


FIGURA 2.2 - Etapas na criação de um 8051 adaptado a uma aplicação.

Finalmente, após a compilação da descrição VHDL, tem-se a prototipação em FPGA. De acordo com a economia de área obtida, através da implementação de poucas instruções, torna-se possível então a colocação de outros circuitos, como por exemplo *hardware functions*, para serem prototipados em conjunto com o 8051. Dessa forma, obtem-se a integração do sistema completo em apenas um chip (ASIS). Observa-se entretanto, que não constitui-se em tarefa do compilador a determinação de como a área excedente será utilizada.

3 Arquitetura do MCS8051

Por ser um microcontrolador, o MCS8051 possui em sua pastilha não somente a Unidade Lógica e aritmética e Registradores, mas também temporizadores, memórias RAM e ROM internas e portas de Entrada e Saída. Para este trabalho, foi realizada uma descrição VHDL do 8051, com a finalidade de automatizar a síntese através de ferramentas para síntese automática de circuitos, como por exemplo, Alliance [BAZ 94] e Altera[ALT 92]. Uma discussão sobre a descrição do 8051 é realizada na seção 3.1, sendo que esta descrição contém todos os elementos microarquiteturais necessários à execução das instruções as quais desejam-se implementar. Estas instruções serão necessárias à execução das aplicações para as quais o 8051 será dedicado dentro do conceito de ASIP. Algumas elementos da Parte Operativa não foram descritos ainda, pois para a simulação da execução das instruções, esses elementos não são essenciais. Portanto, foram descritos somente os elementos microarquiteturais necessários à execução das instruções as quais está-se simulando. Num momento de implementação de um protótipo ou da utilização efetiva do 8051 em uma aplicação específica, possivelmente algums elementos a mais deverão ser descritos, como por exemplo, as portas de comunicação e a memória RAM externa.

Tipicamente, uma instrução do 8051 é composta pelo código da operação (*opcode*) e pelo(s) operando(s). O código da operação é representado pelo respectivo mnemônico em *assembler*, como por exemplo, ADD para soma, DIV para divisão. Os operandos fonte são representados pelo modo de endereçamento que especifica como estes são acessados, e o operando destino, pelo endereço onde o resultado da instrução deve ser armazenado. Desse modo, o que diferencia uma instrução de outra é o seu tipo e o modo como acessa e armazena os operandos. Cabe relembrar aqui, que o conjunto de instruções a ser implementado constitui-se, na verdade, de um subconjunto do conjunto total de instruções original do 8051. Dessa maneira, a substituição ou emulação de uma instrução por outra(s), se fará através da modificação do tipo e modos de endereçamento dos operandos da instrução por um tipo e/ou modo de endereçamento já existente no conjutno de instruções original. Isto ocorre porque não serão criadas instruções novas. Conclui-se, então, que se houver uma substituição de um tipo de instrução por outro, a instrução substituta deve ser de um tipo contido no conjunto de instruções original do 8051.

O 8051 possui definido, no seu conjunto de instruções, 5 diferentes modos para endereçar os operandos de uma instrução:

1. Registrador: o operando está em um registrador do banco de registradores. O registrador é endereçado através dos três bits menos significativos do primeiro byte da instrução;

1. Direto: o segundo byte da instrução especifica o endereço do operando na RAM interna, sendo um endereço de 8 bits;

1. Indireto: o bit menos significativo do primeiro byte da instrução especifica entre o registrador R0 ou R1 do banco de registradores, o qual possuirá como conteúdo, o endereço de 8 bits do operando. O endereço é de 8 bits pelo fato de os registrdores possuírem o tamanho de 8 bits;

1. Constante: pode ser uma constante no segundo byte da instrução, um endereço de 11 ou 16 bits (instrução de três bytes), ou ainda um endereço para uma instrução de salto. Dependendo da instrução, este endereço pode estar no segundo byte da instrução (8 bits) ou no segundo e no terceiro byte da instrução (16 bits). Quando a constante constitui-se num endereço para uma instrução de salto é chamada de endereço relativo;

1. Bit: endereço direto de um bit na RAM interna. Este modo de endereçamento é utilizado nas instruções de manipulação de bit.

O banco de registradores do 8051 é mapeado nas posições inferiores da memória RAM interna, sendo composto por 4 bancos com 8 registradores cada. A memória RAM externa pode ser endereçada por 8 ou 16 bits.

As instruções do 8051 podem possuir um, dois ou três bytes. O código da operação sempre é especificado no primeiro byte, não importando o número de bytes da instrução. Em uma instrução de um byte, além do código de operação pode estar especificado o endereço de um registrador, implementando assim os modos de endereçamento direto e indireto. Nas instruções de dois bytes, o segundo byte será um endereço de memória (modo de endereçamento direto), uma constante ou um endereço de salto.

3.1 Descrição VHDL do MCS8051

Para a descrição em VHDL do MCS-8051 procurou-se tornar prática a tarefa de retirar/colocar instruções, devido ao fato de este trabalho estar inserido dentro do contexto de ASIPs, ou seja, arquiteturas com conjunto de instruções reconfigurável para aplicações específicas, ou em outras palavras, um conjunto de instruções para cada aplicação. Torna-se importante destacar que, por estar-se trabalhando sobre uma arquitetura já implementada, inclusive de uso comercial, e também para manter um baixo custo, não serão realizadas grandes modificações sobre esta arquitetura. Desta maneira, uma modificação no conjunto de instruções não implicará em uma modificação em todos os elementos microarquiteturais, no sentido de retirar alguns e colocar outros para melhor implementar as instruções. A opção, para manter o baixo custo, se realizou no sentido de que qualquer modificação no conjunto de instruções implicará basicamente apenas na mudança da parte de decodificação e geração do microcódigo, que é parte integrante da Parte de Controle. Em outras palavras, o que se está dizendo é que quando houver modificações no conjunto de instruções, isto não implicará em mudanças arquiteturais na Parte Operativa. Este conceito se reforça quando se pensa que as instruções implementadas serão um subconjunto do conjunto total de instruções do 8051. Dessa maneira, para a decodificação e geração do microcódigo das instruções, foi reservada uma entidade VHDL separada. Esta entidade é chamada *Parte de Validação*.

Para a descrição de todo o microcontrolador, foram utilizadas as seguintes entidades VHDL:

Parte de Controle;
Parte Operativa;
Parte de Validação;
Máquina de Estados;
Memória ROM interna; e
Memória RAM interna.

A figura 3.1.1 mostra a relação em termos do fluxo dos sinais entre as entidades.

Para esta descrição, a Parte de Controle é responsável pelo controle de alguns sinais durante a execução de cada instrução. Por exemplo, é esta entidade que gera o INC-PC no tempo apropriado, ordenando à Parte Operativa que realize um incremento no Contador de Programa (*Program Counter*). Também esta entidade é responsável pela geração dos sinais de leitura da próxima instrução (LD-IR), igualmente endereçado à Parte Operativa, bem como os sinais de leitura e habilitação da ROM interna (LD-MAR). Um endereço para a ROM é o conteúdo do PC, pois é nesta memória que estão armazenadas as instruções do programa a ser executado. Como o objetivo aqui é trabalhar-se com arquiteturas para aplicações específicas, é natural que o programa seja armazenado em uma ROM. A instrução lida da ROM passa pelo barramento MEMO e é escrita no Registrador de Instruções (*Instruction Register* - IR) na Parte Operativa. Para a descrição com *pipeline,* a entidade Parte de Controle também envia um sinal à Parte Operativa para que esta carregue em um registrador auxiliar a instrução que está sendo executada. Este sinal chama-se *save_context* e é enviado um estado antes da próxima instrução ser carregada no IR. Este sinal torna-se necessário para que se mantenha a instrução sendo executada, uma vez que a próxima instrução será carregada no IR. Nota-se que apenas o primeiro byte da instrução é carregado no registrador auxiliar, pois para manter-se o contexto basta que o código da instrução (*opcode*) seja preservado.

Também a entidade Parte de Controle determina o número de ciclos e de bytes da instrução a ser executada. O número de ciclos é enviado à Máquina de Estados através do sinal *ciclos*, para que esta possa determinar o número de estados necessários à execução de cada instrução. O número de bytes é utilizado pela própria Parte de Controle para determinar o número de vezes que o Contador de Programas é incrementado em cada ciclo de execução de uma instrução. Se por exemplo, a instrução a ser executada possuir 3 bytes, o PC terá que ser incrementado 3 vezes durante a sua execução; uma vez para endereçar a busca do primeiro operando da instrução, outra vez para o segundo e finalmente a terceira vez que serve para endereçar a próxima instrução a ser executada. Ainda, a Parte de Controle é responsável pelo envio do código da instrução sendo executada para a Parte de Validação. Na descrição com *pipeline* é realizado um teste ao enviar-se o código da instrução para a Parte de Validação, para verificar se este código pertence a instrução sendo executada e não a instrução seguindo, já carregada no IR.

A entidade Parte Operativa contém todos os elementos microarquiteturais necessários à execução das instruções definidas para o 8051. Sendo assim, nesta entidade está definida a Unidade Lógico-Aritmética (ULA), o registrador PC, o incrementador do PC, os registradores Acumulador, o Apontador de Pilha, o Registrador de Instruções, além de alguns registradores auxiliares. A conexão entre os elementos microarquiteturais está realizada através de multiplexadores ao invés de

barramentos, como especifica a arquitetura original. A escolha por multiplexadores deu-se pelo fato de que o objetivo final será a prototipação em FPGA, que é constituído por multiplexadores.

A ULA realiza as operações de soma e subtração e deslocamentos para a direita e esquerda.

As figuras nos parágrafos abaixo ilustram os multiplexadores utilizados nas conexões entre os elementos microarquiteturais da Parte Operativa.

Na figura 3.1.2 estão expostas as entradas para o MUX que serve para o endereçamento da memória RAM interna. Este MUX permite que a RAM seja endereçada de diversas maneiras. Por exemplo, quando o operando de uma instrução é um registrador do banco de registradores (ver início do capítulo), o endereçamento da RAM é formado através dos 3 bits menos significativos do primeiro byte da palavra de instrução. Isto ocorre por que o banco de registradores está mapeado na RAM interna. Assim, uma entrada para o multiplexador constitui-se nos 3 bits de endereçamento da palavra de instrução.

O registrador *RamAd* constitui-se no barramento de endereços da RAM. Nota-se que tanto para *reg_ir* quanto para *inst_at* estão definidas entradas para o bit menos siginificativo e para os 3 bits menos significativos. Estas entradas correspondem respectivamente ao modo de endereçamento indireto e a registrador, como visto anteriormente.

O registrador *inst_at* faz-se necessário para a descrição com pipeline. Mais detelhes sobre o *pipeline* para o 8051 são mostrados no capítulo 4. Ainda podem endereçar a RAM interna, o Acumulador, o Ponteiro de Pilha, SP, a saída da própria RAM e a saída da ROM. A saída da ROM está representada pelo registrador memow2 e serve para implementar o modo de endereçamento direto, pois o conteúdo de memow2 será o segundo byte de uma instrução. A saída da RAM, DRAM_OUT, é realimentada para o seu barramento de endereços para a implementação do modo de endereçamento indireto, pois o byte (R0 ou R1) que o bit menos significativo irá endereçar possui como conteúdo, um endereço. A saída da RAM, DRAM_OUT, pode ser realimentada para a entrada do MUX de endereçamento pois o barramento de endereços *Ram_Ad*, está implementado com sendo um registrador.

A figura 3.1.3 mostra o MUX que conecta todos os elementos microarquiteturais que podem realizar a operação de escrita no Acumulador, como por exemplo, a saída da RAM, DRAM-OUT, da ULA, OUT_ALU, da ROM, MEMO e o Contador de Programas, PC. A entrada X”00”corresponde ao *reset*.

O registrador InBus atua como um barramento da Parte Operativa, pois realiza a conexão entre algums de seus componentes. A figura 3.1.4 mostra os componentes que podem escrever no registrador InBus através do MUX apropriado.

A figura 3.1.5 mostra o MUX definido na Parte Operativa para a escrita de alguns componentes no Contador de Programas, PC, bem como o Incrementador do PC. Pode-se verificar na figura 3.1.5 que o resultado de um incremento no PC (saída do Incrementador INC-PC) é armazenado num registrador auxiliar chamado *res-pc*. O conteúdo desse registrador é posteriormente armazenado no PC. Dessa forma o

registrador res-pc constitui-se também numa das entradas para o MUX de entrada do Contador de Programa. O fato de existir o registrador res-pc permite ao contador de programas endereçar uma instrução ao mesmo tempo em que é incrementado. Esta característica é utilizada para instruções de dois bytes que executam sobre o *pipeline*. O capítulo 4, que discute o *pipeline*, exemplifica como o PC é incrementado através do registrador res_pc.

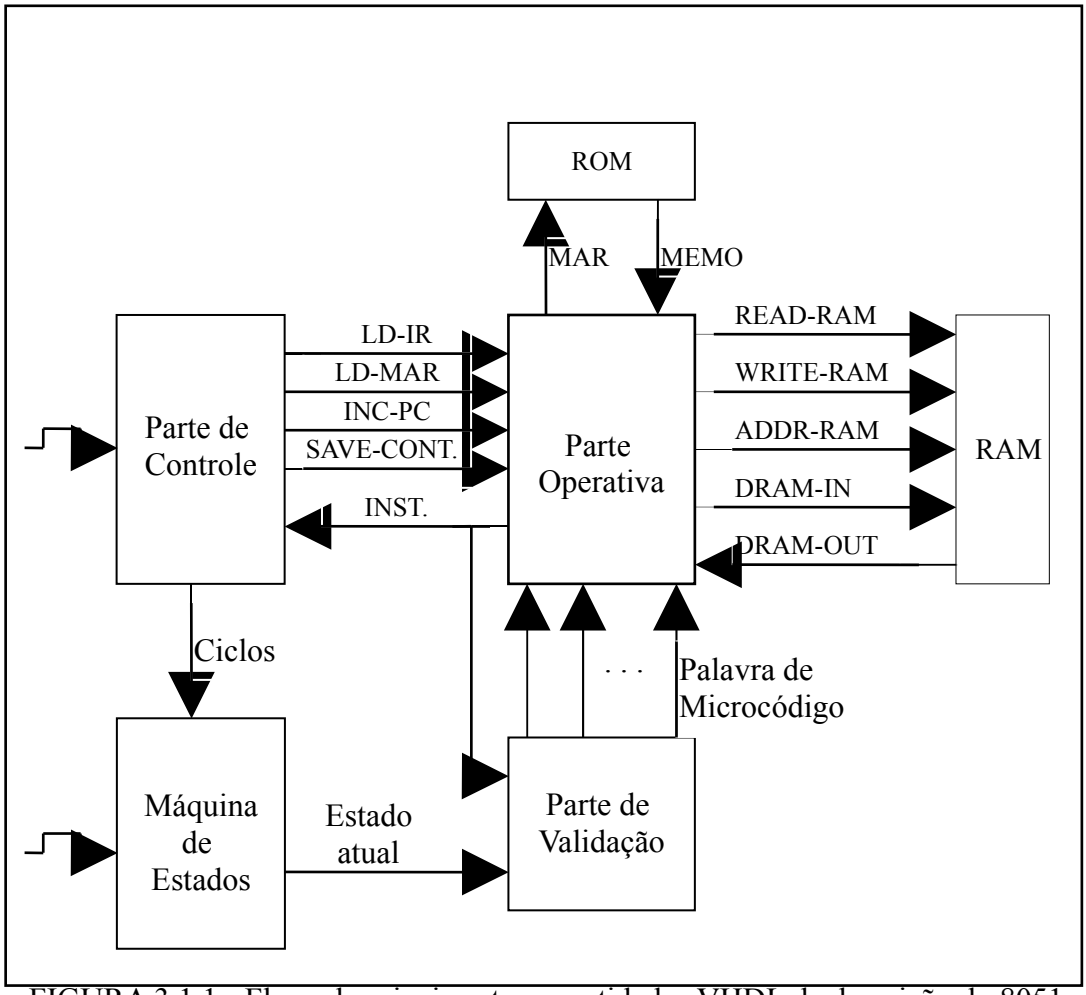


FIGURA 3.1.1 - Fluxo dos sinais entre as entidades VHDL da descrição do 8051.

A saída da RAM, DRAM_OUT, atualiza o PC quando é executada uma instrução de retorno de sub-rotina, RET. Neste caso, a saída da RAM é o conteúdo da pilha que contém o valor do PC armazenado antes da chamada da sub-rotina. Por outro lado, a saída da ROM, MEMO, escreve no PC quando da execução de uma instrução de chamada a sub-rotina, LCALL. Neste caso, MEMO, possui o endereço de chamada da sub-rotina. Tanto para MEMO quanto para DRAM_OUT, são necessárias duas entradas no MUX de entrada do PC porque o PC possui 16 bits. Desse modo, para cada entrada é realizada a concatenação (sinal & na descrição) com os outros 8 bits do PC, sendo necessárias duas leituras para atualizar o Contador de Programa.

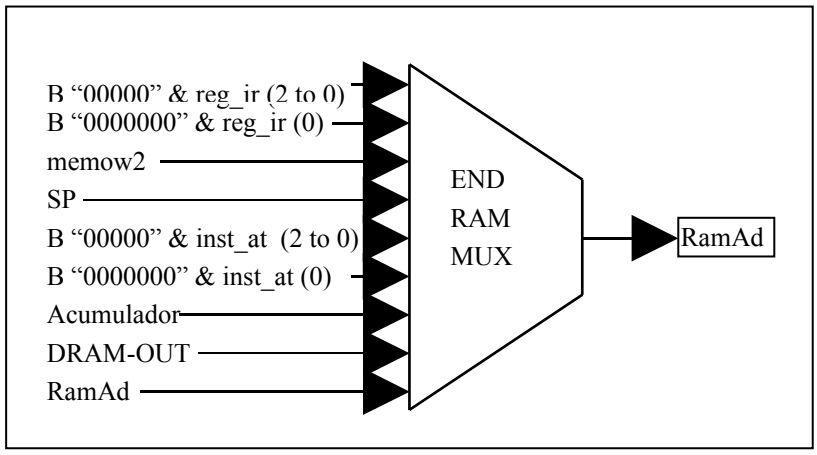


FIGURA 3.1.2 - MUX para endereçamento da RAM

A figura 3.1.6 mostra as entradas (registradores Reg-A e Reg-B) e a saída (Out-ALU) da Unidade Lógico-Aritmética.

O registrador de entrada da ULA chamado Reg-A, pode ter o seu conteúdo alterado através de mais de um elemento microarquitetural. Dessa forma foi especificado um MUX para realizar a conexão entre todos os componentes da Parte Operativa que realizam a operação de escrita neste registrador. A figura 3.1.7 mostra esse MUX, onde pode-se verificar que o Acumulador, o Apontador de Pilha e a saída da ULA podem alterar o conteúdo de Reg-A.

A figura 3.1.8 ilustra um esboço da Parte Operativa. Por questões de praticidade do desenho, os elementos microarquiteturais estão conectados através de um barramento. Na figura 3.1.8 aparecem destacados os registradores que foram acrescentados à PO para a impementação do *pipeline*. O capítulo 4 exemplifica cada um desses registradores.

A entidade Parte de Validação serve para decodificação das instruções e geração do microcódigo. Para conseguir isso, esta entidade recebe como entradas, da Parte Operativa a instrução sendo executada, e da Máquina de Estados, o Estado atual do sistema. Desse modo, tendo por base a instrução decodificada e o estado, gera-se o microcódigo apropriado, pois cada palavra do microcódigo está em função dessas duas variáveis. Devido ao comportamento da Parte de Validação, para colocar-se e retirar-se instruções do conjunto de instruções do 8051, basta modificá-la em relação aos decodificadores de cada instrução.

Assim, a entidade Parte de Validação oferece a facilidade de modificação do conjunto de instruções, sendo indicada para arquiteturas reconfiguráveis. Cabe ressaltar que, como a PO não sofre alterações, o conceito de arquiteturas reconfiguráveis dentro dessa abordagem está significando uma alteração do conjunto de instruções, utilizando-se apenas as instruções definidas originalmente para o 8051. Isso significa que não irão ser incorporadas instruções novas no conjunto de instruções.

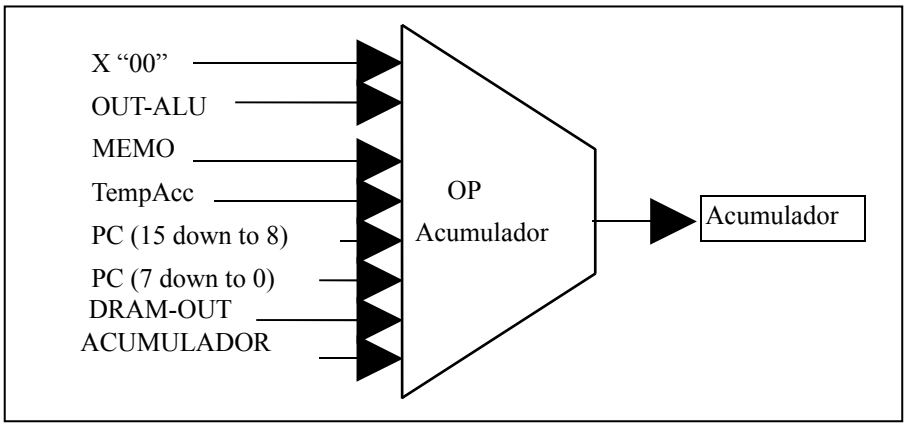


FIGURA 3.1.3 - MUX para escrita no Acumulador

A entidade ROM constitui-se na memória ROM interna, onde estão gravadas as instruções de qualquer programa específico que se queira realizar sobre o 8051. Do mesmo modo, a entidade RAM é a memória RAM interna onde se tem as variáveis ou os operandos.

Finalmente, a entidade Máquina de Estados é a geradora dos estados do microcontrolador para a execução do microcódigo. Tendo como entrada o número de bytes da instrução recebida da Parte de Controle, esta entidade gera o número de estados apropriados à execução da instrução. Como saída, envia o *estado atual* do microcontrolador para a Parte de Controle e Parte de Validação. A Parte de Controle utiliza-o para enviar os sinais de controle no tempo correto, como por exemplo, uma ordem para a Parte Operativa realizar uma leitura na ROM. A Parte de Validação utiliza o estado atual para gerar cada palavra do microcódigo. Para a descrição com Pipeline o número de estados gerados pela Máquina de Estados, alternam-se entre três (T0, T1 e T2) para as instruções que executam em um ciclo normal do *pipeline*, e doze (T0,...T11) para as instruções que causam uma “quebra” no *pipeline*.

A descrição do microcontrolador foi realizada dividindo o processador em módulos. Isso facilitou a simulação e depuração de cada módulo embora tenha criado uma tarefa suplementar: a definição dos sinais de *interface* entre os módulos.

A decisão de restringir as alterações unicamente ao módulo correspondente a *Parte de Validação*, ao ser gerado um novo processador, facilita a geração da descrição para a síntese em alto nível, porque o processo de decodificação e geração do microcódigo para todas instruções, é realizado neste módulo. Isto torna a Parte de Validação autônoma no sentido da realização de alterações no conjunto de instruções.

Como a Parte Operativa não sofrerá alterações, o que irá diferenciar um processador otimizado para uma determinada aplicação, de um processador otimizado para outra aplicação, será o conjunto de instruções. Assim, sempre que for necessária a definição de uma nova descrição, basta que sejam efetuadas alterações na Parte de Validação.

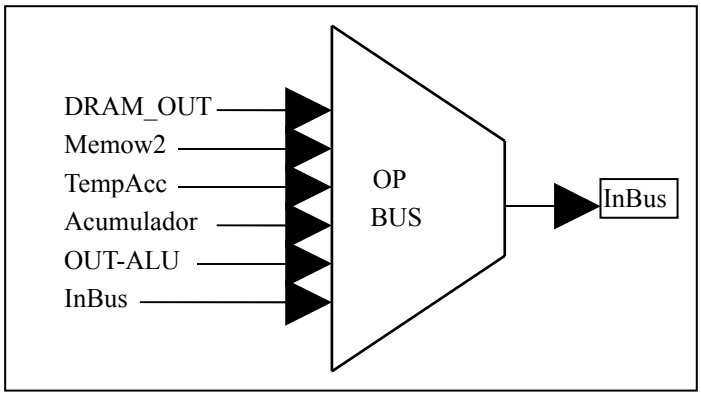


FIGURA 3.1.4 - MUX para ecrita no Registrador InBus

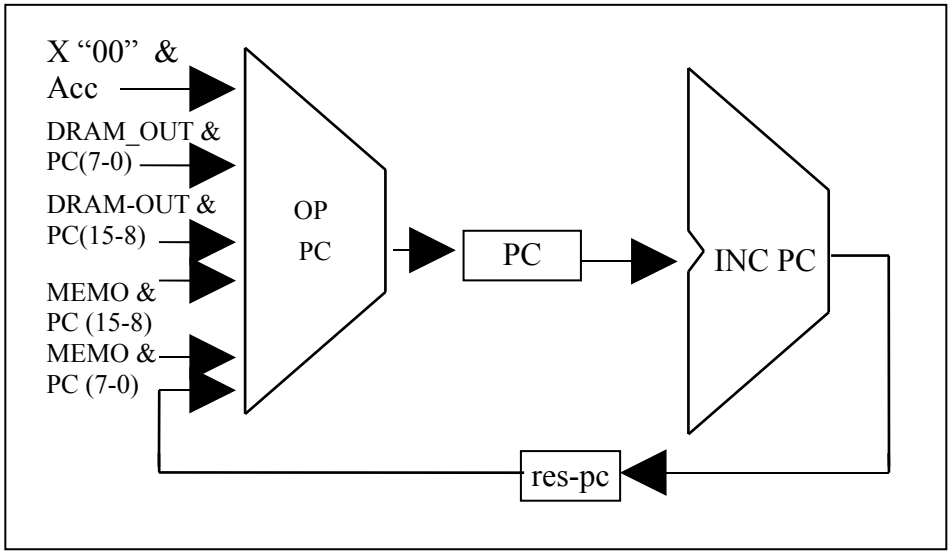


FIGURA 3.1.5 - MUX para escrita no PC e Incrementador do PC

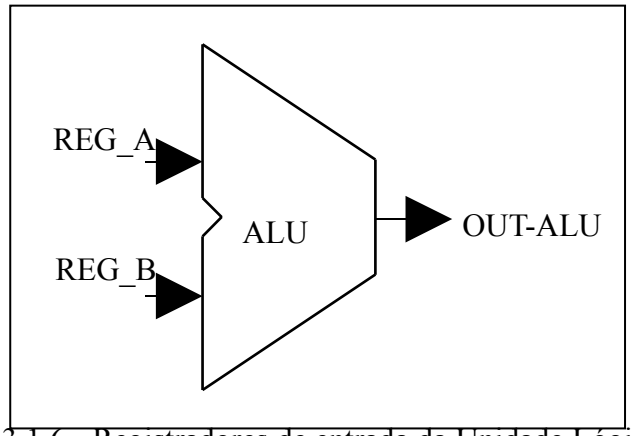


FIGURA 3.1.6 - Registradores de entrada da Unidade Lógica-Aritmética

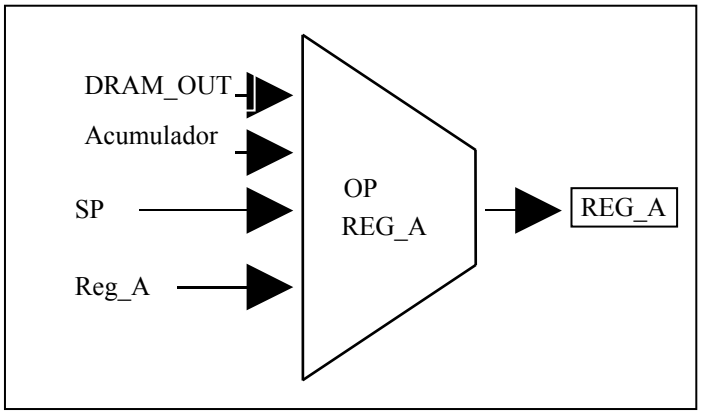


FIGURA 3.1.7 - MUX para escrita no Registrador *Reg_a* da ULA.

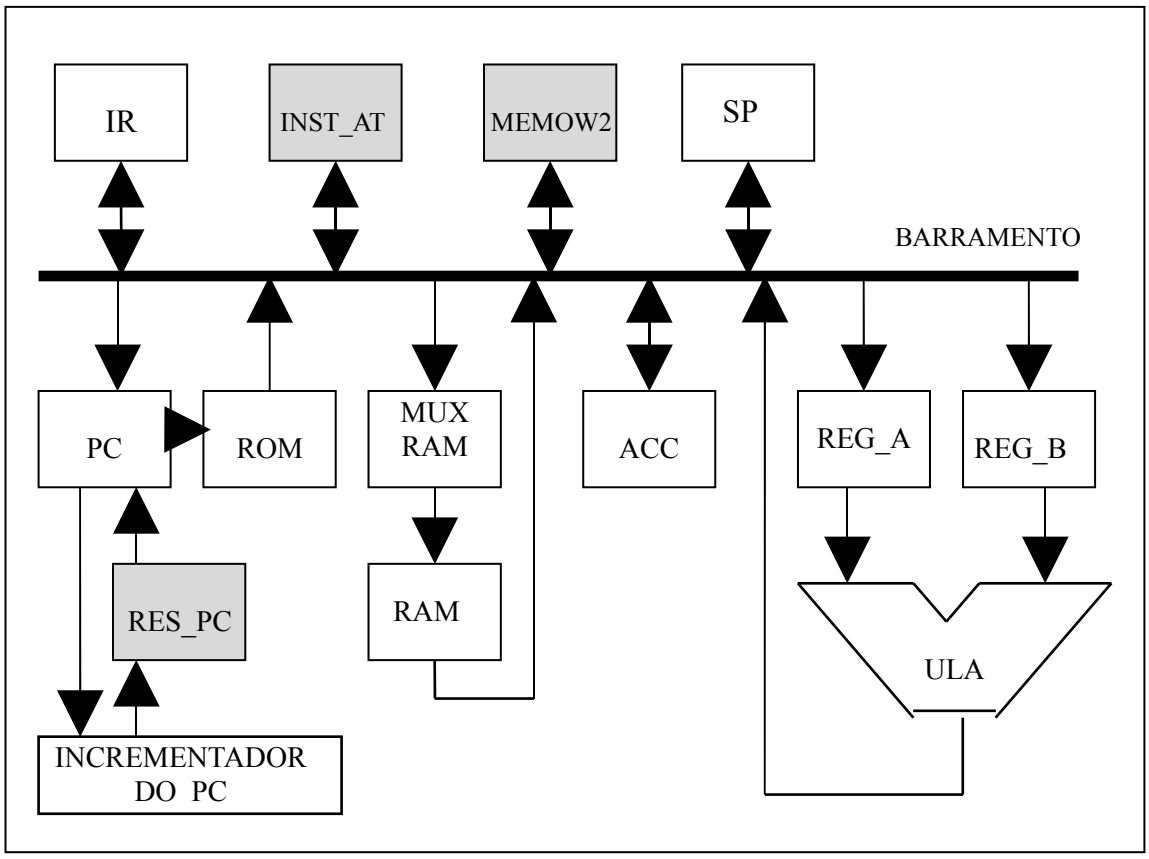


FIGURA 3.1.8 - Parte Operativa do 8051 com *pipeline*

4 A abordagem Pipeline para o MCS8051

Microprocessadores e Microcontroladores cujo projeto datam da década de 70, como 8080, 8086, M6800, Z80, MCS8051, não exploram exaustivamente em sua arquitetura, a implementação de técnicas de paralelismo, como por exemplo, o *Pipeline*. Naquela época, praticamente ainda no princípio do desenvolvimento dos grandes microprocessadores, provavelmente não se imaginava que a demanda pelo aumento do poder de computação desses circuitos fosse crescer tanto e tão rapidamente, quanto verificou-se nos anos posteriores.

Além da evolução da microeletrônica, com chips contendo cada vez um número maior de transistores e com velocidades de operação crescendo surpreendentemente, algumas técnicas de paralelismo foram introduzidas nos processadores, a fim de que estes tivessem aumentado ainda mais o seu poder de processamento. O paralelismo é a forma encontrada pelos projetistas de HW para aumentar a velocidade dos processadores através da execução de instruções (ou microoperações) ou operações de cálculo em paralelo.

Hwang em [HWA 84], comenta que existem basicamente 3 técnicas de paralelismo que são mais empregadas nos processadores atuais:

Sistemas Multiprocessados;

Sistemas Matriciais; e

Pipeline.

Os Sistemas Multiprocessados atentam para a execução de instruções, sub-rotinas ou até programas inteiros em paralelo, através da replicação do número de processadores do sistema. Dessa maneira, tem-se diversas Partes de Controle e Partes Operativas trabalhando independentemente [NAV 87]. A idéia em termos de HW é bastante simples, bastando que se conecte o número de processadores desejados de maneira apropriada. Porém, a implementação do HW normalmente é complexa pondendo possuir um custo elevado, dependendo do número de processadores utilizados. Cada processador acrescentado ao sistema aumenta a complexidade e o custo do sistema. Em termos de complexidade pode-se citar a arquitetura dos barramentos de conexão entre os processadores, e destes com a memória. A arquitetura das conexões deve oferecer recursos de comunicação e sincronização eficientes. Para o custo verifica-se, o custo de cada processador, as conexões na placa, o controle, muitas vezes complexo, do fluxo de sinais entre as conexões, etc. Dessa forma, este tipo de arquitetura para os dias atuais, está direcionada para aplicações que priorizam a velocidade em detrimento do custo. A programação de uma arquitetura multiprocessada tende muitas vezes a constituir-se de uma tarefa complicada, devido a dependência entre os dados dos programas. A programação deve ser feita levando-se em consideração a arquitetura da máquina, sendo em muitos casos realizada em baixo nível, como é o caso da linguagem Occam para Transputers. Entretanto, para os

programas que ajustam-se bem à arquitetura, o ganho em termos de desempenho é considerável.

Máquina matriciais adequam-se bem a resolução de problemas onde tem-se instruções que manipulam uma grande quantidade de dados que podem ser executados em paralelo. O exemplo clássico são as operações com matrizes. Essa máquinas possuem uma Parte de Controle e várias Unidades Funcionais. Esta técnica, a exemplo dos sistemas multiprocessados, exige também uma replicação de HW, que aumenta de acordo com o número de unidades funcionais.

O *pipeline* implementa uma idéia semelhante ao processo de fabricação industrial em série, onde cada funcionário realiza uma tarefa específica na fabricação de um produto. A soma das tarefas de todos resultará no produto completo [KOG 81] [RAM 77]. Essa mesma idéia pode ser aplicada para um processador na execução de instruções e, em operações aritméticas. A execução de cada instrução requer uma sequência de microinstruções, sendo que cada microinstrução opera sobre um circuito. Assim, aplicar a técnica de *pipeline* à execução de instruções consiste na execução de microinstruções em paralelo de modo a manter todos os circuitos ocupados. Como consequência, tem-se a execução das instruções em paralelo através de um paralelismo temporal. A mesma idéia aplica-se a circuitos envolvidos com operações aritméticas.

A implementação de um *pipeline* requer uma mudança na maneira como são seqüencializadas as microinstruções. Isto implica em alterações nos circuitos seqüencializadores da Parte de Controle. A Parte Operativa poderá sofrer alterações no sentido de evitar conflitos nas fases do *pipeline*, como por exemplo, a utilização do mesmo circuito por duas microinstruções simultaneamente. Normalmente são acrescentados alguns registradores na Parte Operativa pois cada fase de execução do *pipeline* possui tempo diferente de execução. Esses registradores servem então para sincronizar as fases, evitando conflito entre as microinstruções.

Dessa maneira, o *pipeline* não requer uma grande replicação de HW, como pode ocorrer em Sistemas Multiprocessados ou Matriciais. O custo maior do *pipeline* está relacionado a gerência das fases de execução. Esta gerência pode ser toda realizada em HW na Parte de Controle, ou pode possuir partes implementadas em SW. Um exemplo seria deixar por conta do Compilador a tarefa de administrar os conflitos que podem ocorrer entre os estágios do *pipeline,* como é realizado em [JUN 93]. Observa-se ainda que aqui está-se referindo a um *pipeline* de instruções típico de uma máquina de *von Neumann*, com poucos estágios de execução. Para *pipelines* com vários estágios, *superpipelines* ou *pipelines superescalares* ocorrem maiores replicações de HW na Parte Operativa. Além disso, o gerenciamento dos estágios de execução na Parte de Controle torna-se complexo, pois exige um número maior de circuitos para controlar e sincronizar cada fase do *pipeline*. Isto envolve um custo maior em termos de HW. Em relação ao SW, o *pipeline* possui a vantagem de não exigir alterações na programação, ou seja, o mesmo programa executa em uma máquina com ou sem *pipeline*.

Conclui-se então que, um *pipeline* de instruções com um número não muito elevado de estágios, possui um custo em termos de HW e SW inferior a sistemas que usam uma maior replicação de HW e exigem uma programação específica, como Sistemas Multiprocessados e Matriciais.

Como uma das imposições deste trabalho diz respeito a não aumentar significativamente o custo do 8051, optou-se por aplicar sobre esse processador como técnica de paralelismo, o *pipeline*. Além disso, o SW já desenvolvido para esse processador não terá que ser adaptado, pois o *pipeline* é transparente para o programador.

4.1 Motivações para o uso do Pipeline

A implementação de alguma técnica de paralelismo para o 8051, dentro do escopo deste trabalho, advém da necessidade de um aumento na velocidade de execução das instruções, o que pode ser obtido através de um *pipeline* de instruções. O 8051 em que está-se trabalhando será uma máquina RISC, possuindo um conjunto de instruções reduzido. O conceito de RISC neste caso, aplica-se pelo fato do conjunto de instruções ser reduzido. As instruções utilizadas serão as mesmas do conjunto de instruções original.

Com um conjunto de instruções reduzido, a tendência é que os programas em linguagem montadora assumam um tamanho maior do que os seus correspondentes que utilizam o conjunto completo de instruções. Esse fato ocorre porque, em muitas situações, uma instrução que não está implementada por ser pouco utilizada pela aplicação, deverá ser susbtituída por duas ou mais que realizem a mesma função. Como se tem um conjunto pequeno de instruções disponíveis, é provável que num programa sejam realizadas algumas substituições. Como as substituições serão realizadas utilizando-se as instruções CISC originais, toda vez que uma instrução for substituída por duas ou mais, haverá um aumento do tempo de execução. Para que o código RISC resultante não execute em um tempo maior do que o seu equivalente CISC, optou-se pela implementação de um *pipeline* de instruções, que proporcionará um aumento na velocidade de execução das instruções.

As instruções do 8051, levam 12 ou 24 ciclos para executarem. Para o 8051 com *pipeline* a maioria das instruções levam 3 ciclos, sendo que algumas podem levar 11 por serem mais complexas. O *pipeline* deverá então, compensar o atraso que se tiver a mais, resultante do processo de substituição de instruções. Quanto a memória ocupada pelo programa, estima-se que seja maior. O tamanho da memória é um preço que deve-se pagar, pois programas RISC possuem a tendência de ocuparem mais memória do que programas CISC por possuirem um número maior de instruções. Se o programa RISC executar num tempo menor ou igual ao seu equivalente CISC, o custo maior em termos de memória é compensado pelo número pequeno de instruções implementadas. Um conjunto de instruções reduzido resulta numa economia de área, pois são necessários poucos circutos decodificadores e geradores de microcódigo.

Uma parte da área economizada pode inclusive ser utilizada para aumentar o tamanho da memória RAM interna do processador.

4.2 A escolha do modelo do Pipeline

Para a escolha sobre qual a configuração do *pipeline* a ser utilizado, leva-se em consideração, principalmente, as limitações impostas pela arquitetura em termos de fluxos dos sinais pelo *data-path* e os objetivos propostos. Em termos dos fluxos de sinais, deve-se observar quais os circuitos que podem realizar operações em paralelo para cada microoperação. Os objetivos neste caso, estão relacionados ao custo, sendo que deve-se procurar não aumentá-lo significativamente. Uma maneira de se conseguir isto é através da realização de poucas alterações na arquitetura original, no que diz respeito a inclusão de HW.

Em relação às limitações arquiteturais, verifica-se que esta possui apenas um barramento para as operações de leitura e escrita na memória RAM que contém os dados. Desse modo, não são permitidas as operações simultâneas de leitura e escrita para esta memória. Praticamente todas as instruções do 8051 possuem como um dos operandos fonte ou como operando destino uma posição da RAM. O banco de registradores está mapeado nos endereços inferiores da memória RAM interna, ou seja, constituem-se de posições de memória. Algumas poucas instruções que trabalham apenas com o Acumulador e com o registrador de *status* não necessitam acessar a RAM interna, como por exemplo, *INC A*. Por outro lado, nenhuma instrução tem os dois operandos fonte como posições da RAM. Para todas as instruções que possuem dois operandos fonte, como por exemplo *ADD*, um deles será o Acumulador, um bit da palavra de *status* do processador, uma constante ou um endereço de salto. Pode ocorrer ainda de o primeiro operando ser o Acumulador e o segundo operando o registrador B. Isto significa que nunca haverá a necessidade de se realizarem duas leituras na RAM para a execução de uma instrução. Por outro lado, normalmente ter-se-á uma leitura e/ou uma escrita a cada instrução executada.

As operações de escrita e leitura nos registradores (Acumulador, Apontador de Pilha, Registrador B, Registrador de Estado e Contador de Programa) podem ser realizadas em paralelo, pois, devido ao fato da descrição VHDL do 8051 ter sido projetada pensando-se na implementação em FPGA's, os fluxos dos sinais entre os barramentos e os registradores foram implementados através de multiplexadores. Desse modo, não existirão conflitos por exemplo, para a escrita simultânea de dois dados diferentes em dois registradores, os quais, na especificação original da arquitetura, estariam conectados ao mesmo barramento.

Tendo em vista as limitações da arquitetura e o custo envolvido, pesquisou-se entre diversas configurações de *pipeline*, aquela que melhor refletisse as condições especificadas.

Para a execução de uma instrução em um processador típico de *Von Neumann*, devem existir as seguinte etapas ou microoperações[JUN 93]:

1. **E** ⇨ Endereçamento da próxima instrução a ser executada;

1. **I** ⇨ Leitura da instrução;

1. **D** ⇨ Decodificação;

1. **R** ⇨ Endereçamento e leitura dos operandos na RAM ou no banco de registradores;

1. **O** ⇨ Operação ou execução da instrução; e

1. **W** ⇨ Escrita se necessário do resultado da operação na RAM ou no banco de registradores.

1. Através da combinação entre essas seis fases criam-se diversas configurações para a realização de um *pipeline* de instruções. As combinações vão desde a execução das seis fases em sequência, portanto sem nenhum paralelismo; até a execução das seis fases em paralelo, onde se obtém o máximo de paralelismo. As outras combinações são formadas com algumas fases executando em sequência, paralelamente às fases restantes. Para o caso das seis fases estarem executando em paralelo e, assumindo-se que cada fase execute em um ciclo de relógio, obtém-se a execução de uma instrução por ciclo de relógio, como mostra a figura 4.2.1.

E	I	D	R	O	**W**						
	E	I	D	R	**O**	W					
		E	I	D	**R**	O	W				
			E	I	**D**	R	O	W			
				E	**I**	D	R	O	W		
					E	I	D	R	O	W	
						E	I	D	R	O	W

1. FIGURA 4.2.1 - Pipeline de instruções com uma instrução por ciclo de relógio.

1. Observa-se na figura 4.2.1, destacado em negrito, a fase a partir da qual tem-se uma instrução a cada ciclo de relógio, através da execução em paralelo de todas as fases de execução de uma instrução. Porém, para que se consiga implementar essa configuração de *Pipeline*, a arquitetura deve possuir algumas características, entre as quais:

   Devem existir dois barramentos de acesso a memória que contém as instruções: um para o endereçamento e outro para a leitura da instrução, visto que as fases E e I executam em paralelo;

   Para o acesso aos operandos, necessitam-se três barramentos, pelo fato das fases R e W executarem simultaneamente. Isto ocorre porque muitas instruções utilizam dois operandos. No caso dos dois operandos estarem presentes na memória, são necessários dois barramentos para a leitura e uma organização de memória que permita duas leituras simultâneas. Ainda, se o resultado da operação deve ser escrito na memória, surge a necessidade de um terceiro barramento para a escrita, em paralelo com as leituras. No caso dos operandos estarem presentes no banco de

registradores, serão necessários três barramentos internos à Parte Operativa;

No momento em que o resultado da execução de uma instrução é armazenado no registrador destino ou na memória (fase W), a instrução seguinte no *Pipeline* realizou a leitura dos operandos (R) na fase anterior. Isto significa que o resultado da instrução que acabou de executar não irá estar disponível para a instrução seguinte. Desse modo, diz-se que este *Pipeline* possui um atraso de uma instrução para o uso dos dados resultantes de uma instrução. Além do mais, no ciclo seguinte a terceira instrução executando no *Pipeline* estará realizando a leitura dos seus operandos, de modo que as saídas das unidades funcionais devem ser realimentadas para as suas entradas, a fim de que se tenha o atraso de apenas uma instrução;

Finalmente, uma instrução de salto terá o endereço do salto calculado apenas na fase W, sendo que nessa fase as quatro instruções seguintes no *pipeline* já estão em execução. Assim, no caso da condição de salto ser verdadeira, há um atraso de quatro instruções. Conclui-se então, que este modelo de *pipeline* possui um atraso de quatro instruções para instruções de salto.

Apesar do modelo de *pipeline* apresentado na figura 4.2.1 ser considerado o modelo mais rápido, verifica-se pelas condições expostas acima que a sua implementação demanda um custo significativo em termos de HW (mais de um barramento interno a PO e para acesso a memória, realimentação das unidades funcionais), e de controle (atrasos das instruções). Portanto, este modelo não pode ser implementado para o 8051 sem que sejam realizadas modificações significativas na arquitetura, principalmente no que diz respeito a duplicação do barramento de acesso à RAM interna. O mesmo raciocínio pode ser aplicado em relação a um *pipeline* superescalar, devido a replicação de unidades funcionais, por exemplo. Sendo que o objetivo é evitar ao máximo alterações na arquitetura, foram procuradas aquelas configurações de *pipeline* que atendessem as limitações. Como exposto no início desta seção, as limitações encontradas na arquitetura, implicam em que a execução das fases R e W não possam ocorrer em paralelo. Dentre as configurações que atendem a essa restrição, optou-se pelo modelo mostrado na figura 4.2.2:

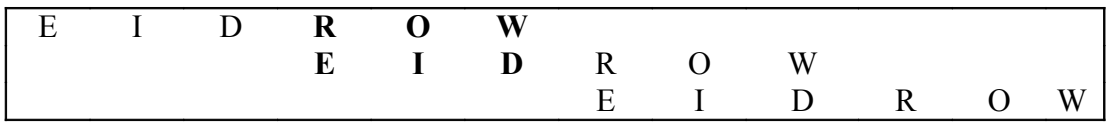


FIGURA 4.2.2 - Pipeline implementado para o 8051

Observa-se destacado em negrito na figura 4.2.2 que, nesta configuração de *pipeline*, existem duas máquinas executando em paralelo. Uma realiza as operações de endereçamento da próxima instrução a ser executada, a leitura dessa instrução e a decodificação. A segunda máquina realiza a leitura dos operandos, e execução da operação e a escrita do resultado.

Assumindo-se que cada fase do *pipeline* corresponde a um período de relógio, tem-se uma instrução executada a cada três períodos de relógio.

O *pipeline* escolhido para ser implementado no 8051 possui um atraso de uma instrução para uma instrução de salto.

3 4.3 A implementação do Pipeline

3 As instruções do 8051 podem ser constituídas por um, dois ou três bytes. As instruções de um byte constituem-se pelo código da operação (*opcode*) e possivelmente alguns bits para endereçamento dos registradores. Para as instruções de dois bytes, o segundo byte constitui um endereço da memória RAM interna ou uma constante. Se o segundo byte for um endereço, com esse endereço é realizado o modo de endereçamento *direto* de um dos operandos. Nas instruções de três bytes utiliza-se um endereçamento de 16 bits. Este endereçamento serve para, por exemplo, realizar uma chamada a sub-rotina através da instrução *LCALL* (*Long Call*).

3 Como visto no capítulo 3, para a execução do *pipeline* a Máquina de Estados alterna entre os estados T0, T1 e T2. Dessa forma, cada estado da Máquina de Estados corresponde a uma fase do *pipeline*. Assim, o estado T0 corresponde à fase do *pipeline* onde executam em paralelo, os estágios E e R; o estado T1 corresponde à fase do *pipeline* onde executam em paralelo, os estágios I e O; e finalmente, o esado T2 corresponde à fase do *pipeline* onde executam em paralelo, os estágios D e W.

3 A figura 4.3.1 exemplifica as fases correspondentes aos estágios E, I e D de execução do *pipeline* explicadas no próximo parágrafo. A figura 4.5 ilustra a temporização do *pipeline*. Os estados S0, S1 e S2 da figura 4.5 correspondem respectivamente, às fases E, I e D do *pipeline*.

3 Nos dois parágrafos que seguem, os números entre parênteses correspondem às indicações da figura 4.3.1.

3 No Estado T0 o valor atual do Contador de Programa é enviado para a ROM de instruções (1), realizando assim o endereçamento da próxima instrução a ser executada, o que correspondente à fase E do *pipeline*. Durante o estado T1, a Parte de Controle envia um sinal à ROM para que esta coloque no barramento de saída chamado MEMO (2a), a instrução endereçada no estado anterior (fase I do *pipeline*). Como visto no capítulo 3, MEMO é um registrador atuando como o barramento da ROM. Ainda neste Estado, o Registrador de Instruções carrega a instrução presente no registrador MEMO (2b). Ao mesmo tempo, no estado T1 o Contador de Programa é incrementado. O processo para incrementar o Contador de Programa ocorre com o auxílio do registrador *res_pc*. Como pode-se verificar nas figuras 4.3.1 e 3.1.5, a saída do Incrementador do PC está conectada ao registrador res_pc que, por sua vez, possui a saída conectada ao MUX de entrada do PC. Durante o bordo crescente do relógio a saída do Incrementador do PC é armazenada no registrador res_pc (3a) e, durante o bordo decrescente do relógio o conteúdo de res_pc é armazenado no Contador de Programa (3b).

3 No estado T2, é realizada a Decodificação da instrução pela Parte de Validação (4). Durante este mesmo estado, se a instrução atualmente sendo executada possuir dois bytes, novamente o conteúdo do Contador de Programa é enviado à ROM para endereçar o segundo byte (5), ao mesmo tempo em que a Parte de Controle envia um sinal para ROM colocar em MEMO o valor endereçado (2a). O segundo byte é então armazenado no registrador *memow2* (6). Ainda, se a instrução for de dois bytes, no bordo crescente do relógio no Estado T2 o registrador res_pc é incrementado (7a). No bordo descrescente deste mesmo Estado, o Contador de Programa é atualizado (7b) para endereçar a instrução seguinte no próximo Estado (T0 do ciclo seguinte). Uma instrução de três bytes causa uma quebra no *pipeline*, de modo que são necessários mais que três ciclos para sua execução. Nesse caso, a Máquina de Estados assumirá mais que três estados. Quando uma instrução de três bytes está sendo executada, a ROM é novamente endereçada no estado T3 para a leitura do terceiro byte e o Contador de Programa incrementado no estado T4. Esse é o funcionamento da Máquina correspondente às fases E, I e D do *pipeline*.

3 A outra máquina que complementa o *pipeline* corresponde às fases R, O e W. Esta máquina está implementada em parte na Parte de Validação através da geração das microoperações, e em parte na Parte Operativa, pela realização da leitura dos operandos, execução da operação da instrução e escrita do resultado.

O 8051 sofreu alterações na temporização, o que veio a reduzir o número de estados para a execução das instruções. O objetivo dessa alteração na temporização é a redução do número de estados de execução de cada instrução para a adequação com o modelo do *pipeline* proposto. Como as fases R, O e W executam em sequência, torna-se necessário que estas fases possam ser executadas em 3 estados da Máquina de Estados, caso contrário, não poderão ser executados em paralelo com a máquina correspondente às fases E, I e D do *pipeline*. As figuras 4.3.2 e 4.3.3 ilustram respectivamente, o diagrama de temporização do 8051 sem e com *pipeline*.

Como pode-se verificar nas figuras 4.3.2 e 4.3.3, tanto o 8051 original quanto a versão adaptada ao *pipeline*, possuem um ciclo de máquina de 6 estados. A diferença porém, está no número de períodos de relógio de cada estado para cada versão do 8051. Para o 8051 sem *pipeline,* cada estado equivale a 2 períodos de relógio, enquanto que no 8051 adaptado ao *pipeline,* cada estado equivale a 1 período de relógio. O 8051 original necessitava de 2 ciclos de relógios para cada estado de execução das instruções pelo fato de que a máquina de estados estava implementada com *latches* sincronizados por um relógio de dois níveis, atuando como mestre-escravo. Assim, cada estado consumia dois ciclos de relógio para ser gerado. Para esta implementação utilizam-se *Flip-Flops* sensíveis ao *nível* (acoplamento direto), o que demanda apenas um ciclo de relógio para cada estágio de execução das instruções. Assim, um ciclo de máquina do 8051 original corresponde a 12 períodos de relógio, enquanto que um ciclo de máquina no 8051 adaptado ao *pipeline* corresponde a 6 períodos de relógio. Dessa forma,

cada estado da máquina de estados equivale à uma fase de execução do *pipeline*.

3 As figuras 4.3.4 e 4.3.5 mostram o exemplo da alteração da temporização realizada na descrição VHDL da Parte de Validação para a instrução de Incremento do Registrador **n**, *INC Rn*.

Observe-se a diferença entre as gerações de microoperações nas Partes de Validação com e sem *pipeline*. Na figura 4.3.4, de acordo com a temporização especificada originalmente para o 8051, são necessários nove ciclos para a leitura dos operandos, execução da operação e escrita do resultado (T3 até T12). Os primeiros ciclos (T0 até T2) correspondem ao endereçamento, leitura e decodificação da instrução, como pode ser verificado na figura 4.3.5. Já na Parte de Validação adequada ao *pipeline*, as operações de leitura dos operandos, execução da operação da instrução e escrita do resultado, são realizadas em três ciclos (T0, T1 e T2) como mostra a figura 4.3.3. As operações de endereçamento, leitura e decodificação da instrução são realizadas em paralelo nos mesmos ciclos.

Com a alteração da temporização uma instrução que antes necessitava doze ciclos para executar, agora necessita de apenas seis, correspondendo a execução das fases E, I, D, R, O e W em sequência. Como no *pipeline* a máquina E, I e D (estados T0, T1 e T2 da figura 4.3.3) executa em paralelo com a máquina R, O e W (estados T3, T4 e T5 da figura 4.3.3), tem-se um ganho de quatro vezes, ou seja, uma redução de doze para três ciclos necessários à execução de uma instrução.

As instruções que quebram o *pipeline* são aquelas compostas por três bytes, e são minoria no conjunto de instruções do 8051. Estas instruções necessitam de 2 ciclos de máquina para executarem, como mostra a figura 4.3.3.

Com relação ao custo em termos de HW necessário para a implementação do *pipeline*, verifica-se que foram necessários três registradores a mais para a Parte Operativa, um aumento do MUX de endereçamento da RAM e um sinal de controle a mais na Parte de Controle.

Os registradores que foram acrescentados à Parte Operativa são:

inst_at: serve para armazenar a instrução sendo executada após a leitura da instrução seguinte na fase I;

memow2: serve para armazenar o segundo byte de uma instrução. O registrador memow2 serve para manter a consistência do registrador MEMO. Isso porque o segundo byte de uma instrução será utilizado apenas na fase de escrita (W) do *pipeline*, e nesta fase a instrução seguinte já foi carregada no Registrador de Instruções através do registrador MEMO;

res-pc: serve para permitir o incremento do PC ao mesmo tempo em que este registrador está endereçando a próxima instrução. A figura 3.1.5

mostra as conexões entre os registradores PC e res-pc. O registrador res-pc torna-se necessário quando uma instrução de dois bytes executa sobre o *pipeline*, pois no terceiro ciclo, o PC endereça o segundo byte da instrução, ao mesmo tempo em que é incrementado para endereçar a instrução seguinte.

O aumento do tamanho do MUX de endereçamento da RAM deve-se a inclusão das entradas para que o registrador *inst_at* (ver figura 3.1.2) consiga endereçar a memória RAM com a finalidade de escrever o resultado de uma operação.

O sinal de controle a mais na Parte de Controle serve para carregar o registrador *inst_at* com o valor do registrador de instruções no estado T0.

Verifica-se então que o modelo de *pipeline* proposto, proporciona uma aceleração na execução das instruções da ordem de 4 vezes, respeita as imposições da arquitetura e pode ser implementado com um custo aceitável.

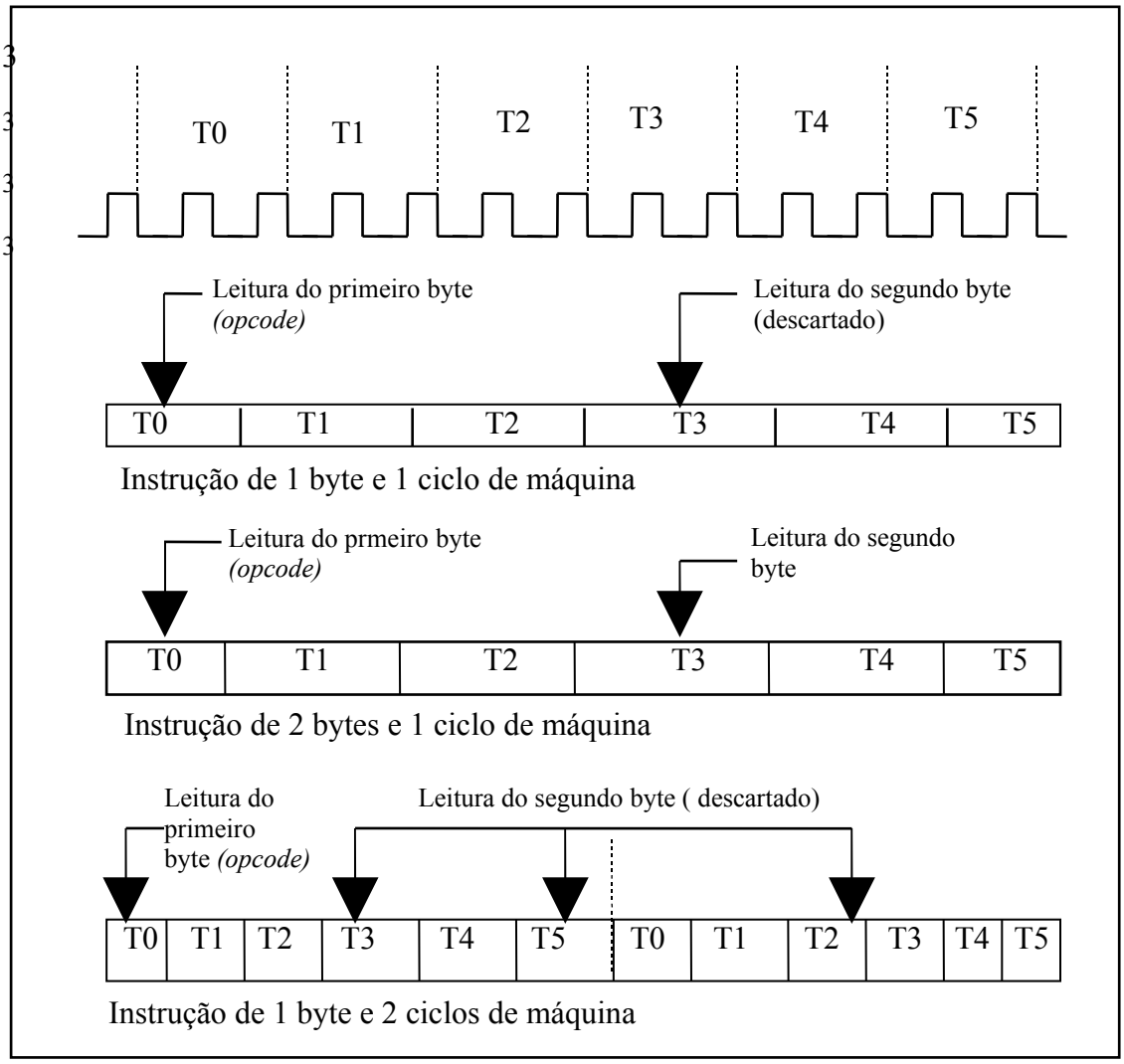


FIGURA 4.3.2 - Temporização do 8051 original

3
3

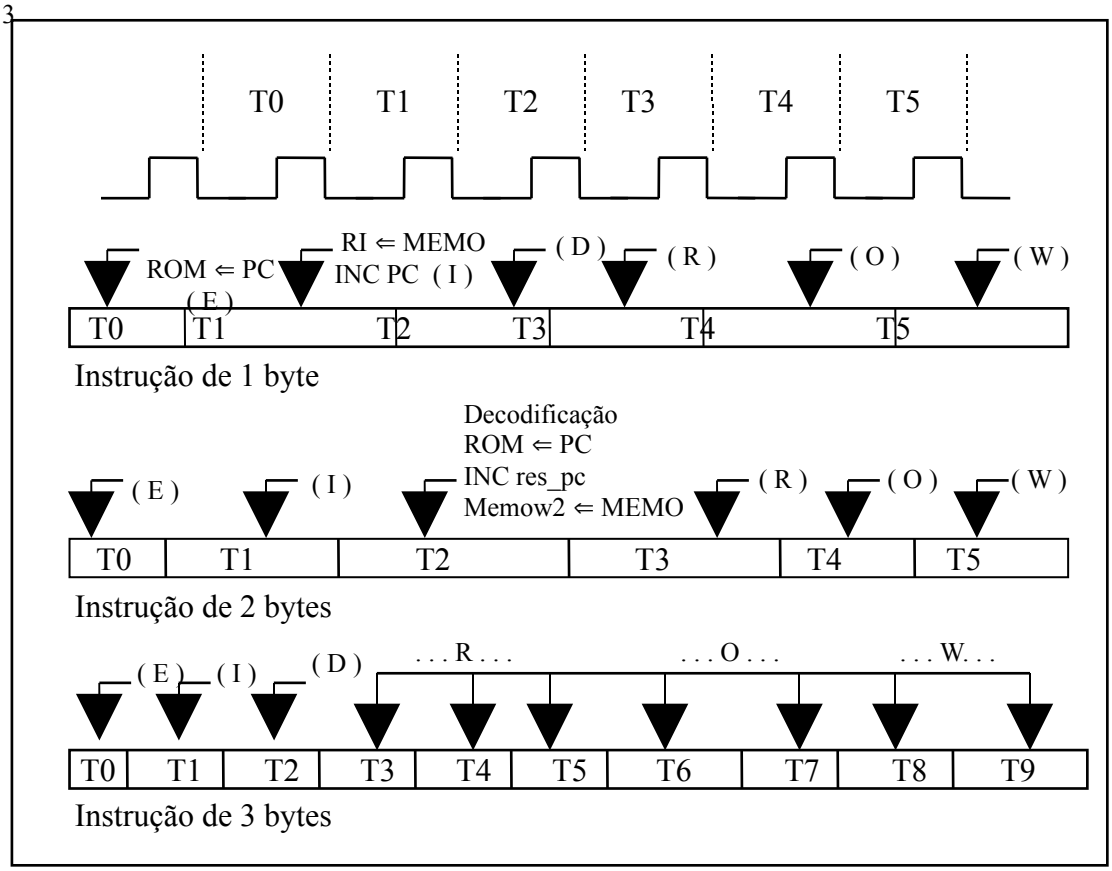
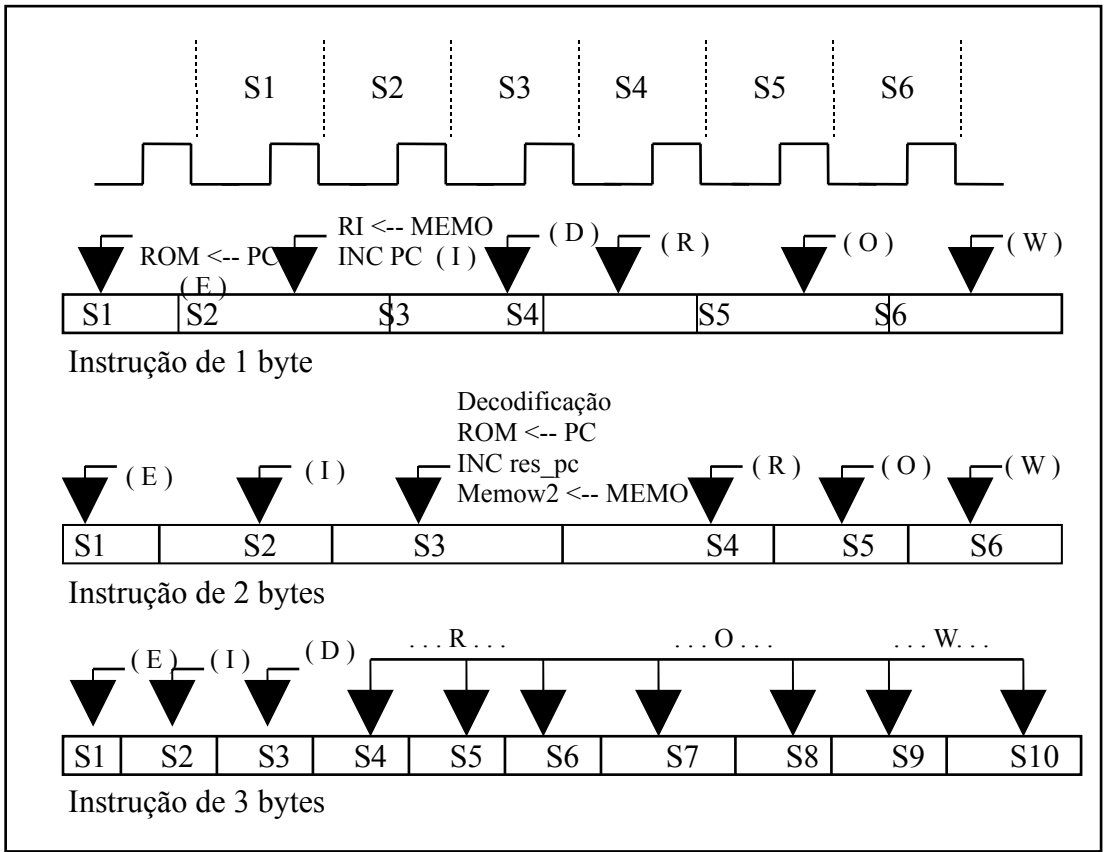


FIGURA 4.3.3 - Temporização do 8051 com *pipeline*



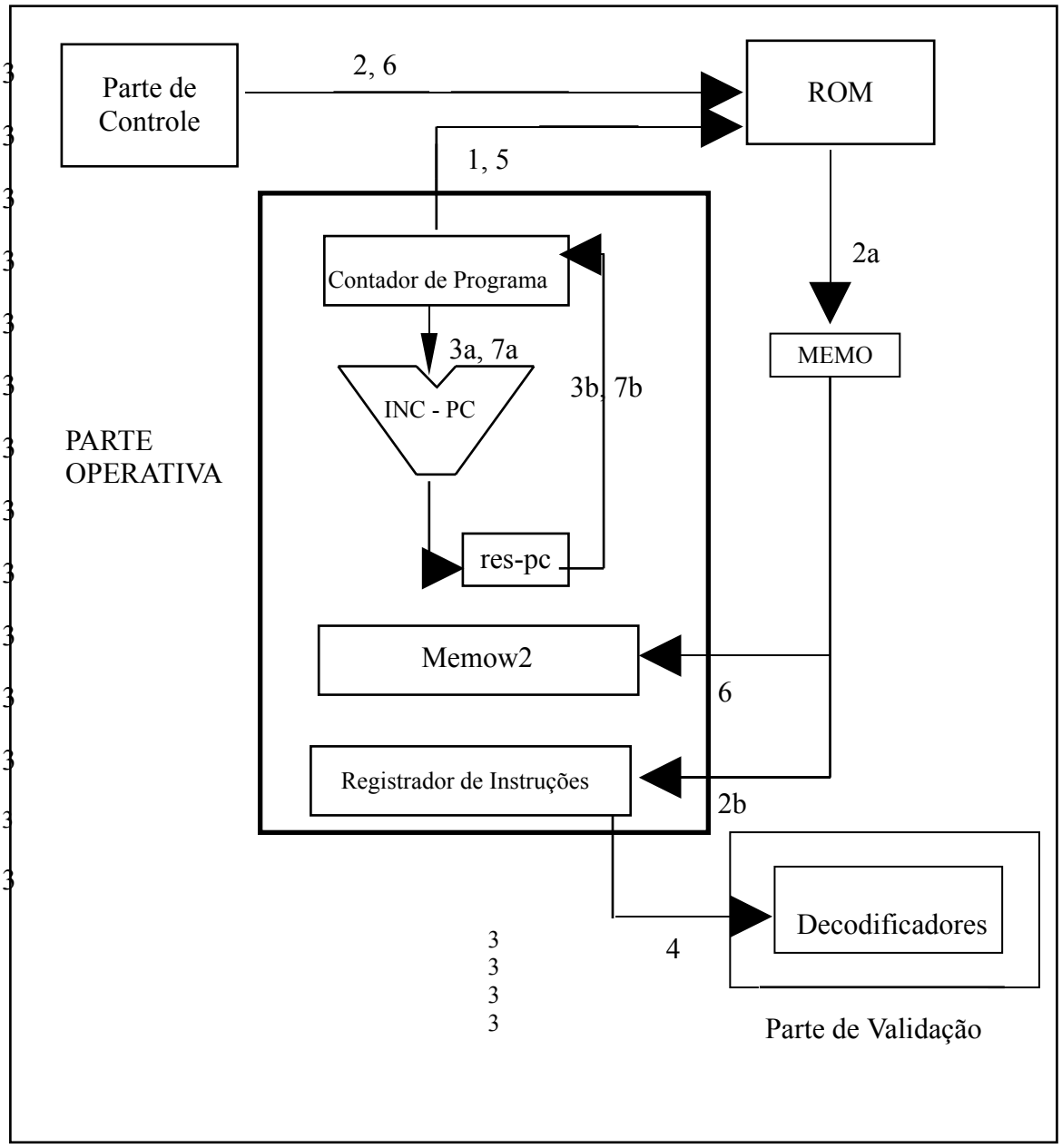


FIGURA 4.3.1 - Fases E, I e D de execução do *pipeline*

```
inc_r_block: block (INSTRUCTION = inc_r)
begin
        with ST select
                output_control <= guarded
                        X"03400000" when B"00011", -- T3 = READ_RAM
                                                   -- endereça a RAM
                        X”000000100" when B"00100", -- T4 = LD_RA_RAM
                             -- carrega no registrador RA o conteúdo endereçado da RAM,
                                  -- que neste caso corresponde ao registrador n.
                        X"04000000" when B"00101", -- T5  = Rn=Rn+ 1
                                -- Incrementa o conteúdo de RA (Rn)
                        X"00000060" when B"01010", -- T10 = LD_BUS_ALU
                                -- carrega no regsitrador BUS a saída da ULA
                        X"01600000" when B"01100", -- T12 = WRITE_RAM
                                -- escreve o conteúdo do registrador BUS (resultado) na
                                        -- memória
                        X"00000000" when others;
end block;
```

FIGURA 4.3.4 - Microinstruções na Parte de Validação para instrução com temporização original

```
inc_r_block: block (INSTRUCTION = inc_r)
begin
        with ST select
                output_control <= guarded
                X"008000201" when B"00000", -- T0 = RD_RAM, LD_RA_BUS
                X"020000000" when B"00001", -- T1 = ALU_CTRL => Rn = Rn + 1
                X"800001000" when B"00010", -- T2 = WR_RAM, LD_BUS_ALU
                X"000000000" when others;

end block;
```

FIGURA 4.3.5 - Microinstruções na Parte de Validação para instrução com temporização adequado ao Pipeline

5 Síntese Lógica do MCS8051

Para a realização da Síntese Lógica para o 8051 foram utilizadas as ferramentas Alliance [BAZ 94] e Altera [ALT 92], o que permite uma comparação de resultados entre duas abordagens distintas: *Standard Cells* e FPGA. A realização da Síntese Lógica visa determinar o número de portas lógicas necessárias para a implementação do 8051 com e sem *pipeline*. De acordo com as resultados obtidos torna-se possível avaliar a descrição com *pipeline* para posterior validação, como discutido em [KRE 97a].

5.1 Síntese do MCS8051 no Alliance

Como foi visto no capítulo 3, foram utilizados vários arquivos e entidades para a descrição completa do 8051. Algumas dessas entidades como a Parte Operativa e a Parte de Validação extrapolam a capacidade da ferramenta *Logic* do ambiente Alliance para realizar a síntese lógica.

Como o Logic trabalha com BDDs para representar as expressões booleanas, possivelmente alguns desse BDDs nas partes Operativa e de Validação são demasiado complexos para serem tratados com o algoritmo da ferramenta Alliance - Logic. Isto pode ter acontecido na Parte de Validação devido a esta ser composta por vários decodificadores e circuitos combinacionais complexos, e na Parte Operativa, por esta ser composta por diversos elementos microarquiteturais, como ULA (que por si só já é complexa), circuito de deslocamento (*shifter),* registradores, etc. Esta diversidade de elementos, muitos deles complexos, impossibilitam o tratamento pelo Logic, ou seja, a ferramenta não consegue realizar as etapas de síntese lógica sobre esse conjunto de elementos microarquiteturais.

Tendo por base este cenário, foi necessário realizar uma divisão dessas entidades, de forma que cada uma das partes (Operativa e Validação) foram divididas, com o objetivo de ter-se entidades mais simples. Pode-se dizer que foi adotado aqui a estratégia "dividir para conquistar", de modo a tornar mais simples a tarefa para o Logic. A estratégia para se obter isso foi subdividir as entidades em entidades menores, de modo que a união dessas entidades menores possuíssem a mesma função da entidade geradora.

Essa divisão das entidades tornou-se uma tarefa cansativa e demorada, pois a cada nova subdivisão, foi necessária a realização de diversos passos, tais como: tentar não repetir os mesmos registradores em ambas entidades por questões de economia e consistência; reajustar as entradas e saídas das entidades; conectar as novas entidades no arquivo estrutural (.vst); compilar novamente e finalmente realizar uma nova simulação. Após uma série de tentativas, acabou-se optando por devidir a PO em 7 partes. Em cada uma dessas tentativas foi observado a colocação dos registradores e quais elementos microarquiteturais que ficavam em cada entidade. Foi necesssário também que se criassem algumas conexões a mais para levar o conteúdo dos registradores de uma entidade para outras. Por exemplo, na entidade oper_ula, tem-se a saída da ULA. Esse sinal deve ser levado para diversas outras partes ou entidades componentes do 8051. Para fazer isso, foi necessário que se criasse uma conexão para cada entidade que necessitasse ler esse dado. Desse modo, foi criada uma saída na entidade oper_ula, e para cada entidade que necessitasse ler a saída da ULA, foi

criado uma porta de entrada com a conexão desse sinal saindo da entidade oper_ula para as entidades que irão ler a saída da ULA.

As entidades, nas quais foi dividida a Parte Operativa são:

oper_pc;

oper_ula;

oper_sp;

oper_ram;

oper_rom;

oper_bus; e

oper_acu.

A entidade *oper_pc* compreende o registrador PC, a unidade de incremento do PC, e um outro registrador chamado PC2, além dos multiplexadores que possuem como entrada os diversos elementos que podem escrever dados no PC. O registrador PC2 é utilizado como auxiliar em algumas operações com o PC, como por exemplo alterações que são realizadas sobre o PC durante uma instrução de chamada de sub-rotina. Neste caso, PC2 é utilizado para realizar as operações e o PC é atualizado apenas no final da execução da instrução.

Esta entidade recebe como entradas o sinal de incremento do PC, vindo da Parte de Controle, os sinais de controle para as entradas dos Multiplexadores de PC e PC2 e tem como saída o conteúdo do PC para ser enviado a outras entidades. Isso significa que, quando alguma outra entidade precisar ler o conteúdo do PC, deverá realizar uma leitura sobre esse sinal. Como conseqüência, essa porta de saída chamada OUTPC, deverá ser uma entrada nessa entidade. Conexões como essa não existiam na descrição original, pois como a Parte Operativa constituía-se em apenas uma única entidade, o conteúdo do PC era direcionado diretamente para todas as outras partes componentes da PO.

A entidade *oper_ula* compreende a Unidade Lógico-Aritmética, com operações de soma, subtração, operações lógicas (and, or, xor ...) e de deslocamentos (*shifters*). Além disso, tem-se nesta entidade os registradores **A** e **B** que são as entradas da ULA, a palavra de *status* do processador (com os bits de *carry*, *overflow*, etc.), e alguns registradores internos à ULA que servem como auxiliares nas operações. Esta entidade possui como saída o resultado de qualquer operação realizada na ULA, através de um multiplexador. Esse MUX tem como entradas a saída do somador/subtrator; a saída do circuito de deslocamento; e a saída do circuito que realiza as operações lógicas, sendo que desta forma a sua saída constitui-se na saída da Unidade Lógico-Aritmética do 8051.

A entidade *oper_sp* possui o registrador *Stack_Pointer* cuja função é apontar para o topo da Pilha do sistema.

Cabe aqui ressaltar a maneira como foram realizadas as subdivisões da entidade-VHDL Parte Operativa em entidades menores para conseguir-se a síntese com o Logic. Aqui, o exemplo da entidade oper_sp deixa muito claro a estratégia que se adotou para realizar essas subdivisões da PO. Como se pode verificar, criou-se uma entidade contendo apenas o registrador SP, e as operações realizadas sobre ele. A opção por se dividir a PO desta maneira se deve ao fato de que houve a preocupação em não repetir os registradores da PO em mais de uma entidade, por questões de custo e consistência, no caso de termos mais de um SP, declarados em diferentes entidades. Neste caso, uma atualização no SP deveria ser feita em mais de um lugar. A entidade oper_sp deixa bem claro que o SP esta descrito apenas nela, e por conseqüência qualquer operação de escrita nesse registrador, deve ser feita através de sua porta de entrada, que irá atualizar o conteúdo do SP. Dessa maneira, o custo a mais que apresenta-se por realizar a subdivisão da PO reduziu-se ao máximo, ou seja, ao custo das conexões de entrada e saída das entidades menores. Ainda mais, o problema de consistência fica automaticamente resolvido, pois o SP fica encapsulado dentro da entidade oper_sp.

As entradas de oper_sp são a saída da ULA (sinal que vem da entidade oper_ula), e um sinal de controle da Parte de Validação que indica uma atualização do conteúdo do SP. Na verdade, a única operação que se realiza sobre o SP é a atualização de seu conteúdo, no momento em que ocorre uma operação que envolve a Pilha. Como saída, oper_sp tem o conteúdo do SP através da porta de saída *outsp*. Essa porta de saída conecta-se com a ULA, que irá incrementá-lo ou decrementá-lo de acordo com a operação de pilha que foi realizada. Oper_sp possui ainda uma outra entrada que constitui-se no sinal de *Reset.* Este é um sinal de controle, que atribui ao SP o valor X”07”. Isto significa que a base na Pilha está na posição 7 da memória RAM interna.

A entidade *oper_ram* realiza as operações de leitura e escrita na memória RAM interna do 8051. No momento em que se escreve este texto, não está implementada na descrição, a memória externa do sistema. No 8051 existem basicamente 3 modos de endereçamento da memória: direto, indireto e registrador. No modo direto, o endereço de memória está diretamente presente na palavra de instrução; no modo indireto, o conteúdo do registrador que vem especificado na palavra de instrução constitui-se no endereço de memória; no modo registrador, o registrador que vem especificado na palavra de memória, é o endereço de memória. Isto ocorre por que no 8051, cada um dos 4 bancos de registradores está mapeado nas posições inferiores da RAM interna. Os únicos registradores físicos são o Acumulador, o registrador B e o *Stack Pointer,* além obviamente, daqueles que não são visíveis ao programador, como os registradores de entrada da ULA, os reg. de operações internas do somador/subtrator, etc.

Tendo em vista os modos de endereçamento possíveis para a RAM, como explicitado no parágrafo acima, criou-se na entidade oper_ram um MUX como mostra a figura abaixo, com o código VHDL.

Como se pode ver na figura 5.1.1, foi definido um MUX, chamado *END_RAM_MUX*, que possui como entrada os diferentes modos de endereçamento possíveis para a RAM. A saída deste MUX está conectada a palavra de endereçamento na RAM. Ressalta-se que a figura 5.1.1, está com o MUX para a descrição com *pipeline*, pois pode-se perceber que nas entradas temos o registrador

de instruções (*reg_ir*) e o registrador de instruções auxiliar (*inst_at*) utilizado no *pipeline* para manter a instrução sendo executada quando a instrução seguinte já foi carregada na Parte de Controle.

```
with END_RAM_MUX select
    RamAd <= guarded B"00000" & reg_ir (2 downto 0) when B"000",
                     B"0000000" & reg_ir(0)         when B"001",
                     MEMO                           when B"010",
                     SP                             when B"011",
                     B"00000" & inst_at (2 downto 0) when B"100",
                     B"0000000" & inst_at(0)        when B"101",
                     Acumulator                     when B"110",
                     DRAM_OUT                       when B"111",
                     RamAd                          when others;
```

FIGURA 5.1.1 - Código VHDL contendo os modos de endereçamento da RAM

Sendo assim, quando se tem na entrada do MUX *B"00000" & reg_ir (2 downto 0)* ou *B"00000" & inst_at (2 downto 0)*, está-se concatenando alguns bits que são iguais a zero, com os 3 bits menos significativos da palavra de instrução, presente em reg_ir ou inst_at, dependendo da situação. Isto é feito dessa maneira, por que assim encontra-se especificado o *modo de endereçamento a registrador* na arquitetura original do 8051. Utiliza-se 3 bits, pois com 3 bits pode-se contar até 7, e este é exatamente o número de registradores que o 8051 possui em cada um de seus 4 bancos de registradores. Como o endereçamento dá-se com 8 bits, basta concatenar os 3 bits de endereçamento, com os bits mais significativos valendo zero, como está especificado na entrada do multiplexador.

Para o *modo de endereçamento indireto*, tem-se na entrada do MUX *B"0000000" & reg_ir(0)* ou *B"0000000" & inst_at(0).* O processo aqui é exatamente igual ao descrito no parágrafo acima para o modo de endereçamento a registrador, com a única diferença que para o modo indireto tem-se apenas o bit menos significativo da palavra de instrução para o endereçar a RAM. Isto ocorre por que os únicos registradores que podem endereçar a RAM indiretamente são o R0 e R1, e então basta um bit para especificar entre um e outro. Para os outros bits, concatena-se com bits valendo zero. Como o conteúdo de R0 ou R1 são um endereço da memória, tem-se também como entrada no MUX a linha *DRAM_OUT.* DRAM_OUT é a saída da RAM, logo quando esta entrada do MUX for habilitada, o próximo endereço da RAM será o dado que acabou de ser lido. Isto significa que para o modo de endereçamento indireto, tem-se duas leituras a RAM: a primeira carrega para a entrada do MUX de endereçamento da RAM, o conteúdo de R0 ou R1, sendo este o endereço da segunda leitura. Dessa maneira, consegue-se carregar um operando indiretamente.

Quando se tem o modo de endereçamento *direto,* o endereço é o segundo byte na palavra de instrução. Como as instruções estão na ROM interna, para endereçar a RAM diretamente, basta conectar a saída da ROM a entrada do MUX que endereça a RAM. Isto está especificado na linha de entrada *MEMO* de END_RAM_MUX, pois

MEMO é a palavra de saída da ROM. A RAM ainda pode ser endereçada pelo Acumulador e pelo SP.

A entidade oper_ram ainda recebe da Parte de Validação os sinais de controle para realizar uma leitura ou escrita na RAM e executa estas ações. Assim, essa entidade possui como saída a palavra de endereçamento da RAM (que é a saída do MUX de endereços como visto acima), o dado a ser escrito, e o bit de habilitação de leitura/escrita na RAM. Todos esses sinais estão conectados a entidade RAM. Como entradas, tem-se todas as entradas do MUX de endereçamento e a saída da RAM.

A entidade *oper_rom* trata da operação de leitura de uma instrução na memória ROM interna. Desse modo, estão declarados nesta entidade, o registrador de instrução *reg_ir,* o registrador de endereço de memória *reg_mar* (Memory Address Register), e no caso da descrição com *pipeline* o registrador de instrução auxiliar *inst_at*.

Esta entidade recebe, como entrada, os sinais de controle para endereçamento e leitura da ROM, vindos da entidade Parte de Controle, o PC que endereça as instruções, e a saída da ULA, que eventualmente pode também endereçar uma instrução. Como saída, esta entidade possui a palavra de endereçamento da ROM, conectada a esta, além de mais duas portas de saída para a instrução (em reg_ir e inst_at). Estas saídas conectam-se às entidades Parte de Controle e Parte de Validação. As operações desta entidade resumem-se em endereçamento e leitura da ROM e a escrita do conteúdo de reg_ir em inst_at antes da leitura da instrução seguinte, para a descrição com *pipeline*. Esta operação é realizada através de um sinal de controle, que vem da entidade Parte de Controle, sendo também um bit de entrada de oper_rom.

A entidade *oper_bus* possui como único componente microarquitetural, o registrador *InBus*, e realiza a operação de escrita neste registrador. Para esta descrição, InBus atua como se fosse o barramento do sistema. Isto significa que muitos sinais ao serem transmitidos de um componente microarquitetural para outro, primeiramente são gravados em InBus, e o componente que deverá receber este dado realiza uma operação de leitura no InBus. Esta descrição do 8051 não foi implementada diretamente com barramento, pelo fato de que a prototipação será feita em FPGA, onde são apenas suportado multiplexadores.

Assim, como na parte de endereçamento da RAM, para esta entidade igualmente implementou-se um MUX para permitir a escrita de diversos sinais em InBus. A figura 5.1.2 mostra o código VHDL contendo esse multiplexador.

Desse modo, as entradas para a entidade oper_bus são as entradas do MUX *OP_BUS* cuja saída é escrita em InBus. A saída desta entidade é o conteúdo de InBus. Esta porta de saída está conectada como porta de entrada a todas outras entidades que necessitam realizar uma leitura em InBus. Essa entidade recebe ainda como entrada, os bits de controle do multiplexador OP_BUS para realizar a escrita do componente adequado. Esse sinal vem da Parte de Validação.

Finalmente, a sétima e última parte em que foi dividida a entidade Parte Operativa, chama-se *oper_acu*. Esta entidade possui como elementos microarquiteturais, os registradores Acumulador e TempAcc. Tal como a entidade oper_bus, multiplexa os diversos componentes microarquiteturais que realizam uma escrita no Acumulador. A figura 5.1.3 mostra o código VHDL para este MUX.

Como se vê na figura 5.1.3, tem-se o multiplexador *OP_ACU* que irá selecionar qual o componente que irá realizar uma atualização no Acumulador. As entradas para este MUX são as entradas para a entidade oper_acu que possui como saídas o conteúdo do Acumulador e do registrador auxiliar chamado TempAcc. Este registrador é utilizado para salvar o conteúdo do Acumulador em algumas operações onde o Acumulador é utilizado como registrador auxiliar e, no final, deve ter seu conteúdo restaurado. Novamente, todas as entidades que necessitarem realizar uma leitura no Acumulador deverão ter como porta de entrada a porta de saída desta entidade, chamada *outacu.* A figura 5.1.4 ilustra as entidades que escrevem ou modificam o conteúdo do Acumulador através de sua porta de entrada, e quais as entidades que recebem o seu valor através da porta de saída *outacu.*

A outra entidade que teve que ser dividida em entidades menores para ser sintetizada com o Alliance - Logic foi a *Parte de Validação.* Esta entidade foi dividida em 3 partes, chamadas de *valid1*, *valid2* e *valid3.* Em cada uma das partes ficou a decodificação e a geração de microcódigo de algumas instruções. Dessa forma, criaram-se 3 palavras de microcódigo diferentes. Para enviar a palavra de microcódigo à Parte Operativa torna-se necessário que se una novamente essas palavras. Isto foi realizado através de um MUX. Na verdade foi necessário que se criasse uma quarta entidade chamada *exit_valid* que contivesse esse MUX. O multiplexador da entidade exit_valid recebe como entrada as 3 palavras de microcódigo as quais constituem-se nas saídas das entidades valid1, valid2 e valid3. A saída desse MUX é a palavra de microcódigo a ser enviada a Parte Operativa, ou melhor, para as entidades que compõem a Parte Operativa. Observe-se que neste caso houve um custo considerável em termos de fios pela implementação desse MUX.

O próprio código da instrução, constitui-se nos bits de controle para o MUX da entidade exit-valid selecionar qual palavra de microcódigo deve ser enviada à Parte Operativa. Isto ocorre por que cada uma das entidades, valid1, valid2 e valid3 geram o microcódigo para determinadas instruções. Assim, para o MUX, basta ser informado em qual entidade está a instrução atual: se estiver em valid1 por exemplo, a entrada correspondente à saída desta entidade é transferida para a saída, e assim sucessivamente.

Para as demais entidades não foi necessário efetuar-se nenhuma divisão para realizar a síntese lógica.

5.1.1 Resultados da Síntese Lógica no Alliance

Os resultados obtidos da síntese lógica estão ilustrados nas Tabelas 5.1.1 e 5.1.2 para a descrição do 8051 com e sem *pipeline.*

Como se pode observar na tabela 5.1.1, foram necessárias 1851 portas lógicas da biblioteca *Standard Cells* do Alliance para implementar todo o 8051, descrito através do Alliance-VHDL com *pipeline*. Observa-se que a entidade VHDL *Exit_valid* não estava presente na descrição original sendo inserida para a realização da Síntese, como explicado no início do deste capítulo. Deste modo, no momento em que se realizar a Síntese de *layout,* o número de portas lógicas necessárias para a implementação do circuito com *pipeline* será igual a 1851, pois sem a entidade *Exit_valid* não há como implementar-se o circuito eficientemente, visto que não se teria a Síntese Lógica. Devido a isso, o custo aumentou em 256 portas lógicas.

Quanto a divisão da Parte Operativa em 7 entidades, acredita-se que isto não aumentou o custo no sentido de acrescentar mais portas lógicas para a implementação física. A maneira como se procurou dividir a Parte Operativa (ver seção 3.1), acrescentou apenas entradas e saídas a mais às entidades componentes da PO, e não elementos microarquiteturais pois estes não foram replicados. Dessa maneira evitou-se um aumento do custo para a síntese. Por exemplo, a todos os elementos que realizam uma leitura do conteúdo do Acumulador, estava conectado um fio vindo da sua saída. Isso ocorria dentro da entidade Parte Operativa. Na descrição modificada para a síntese lógica com o *Logic*, esses fios foram transformados na porta de saída da entidade *oper_acu* e em portas de entrada para cada entidade que lê o conteúdo do Acumulador. O mesmo vale para todos as outras entidades em que foi dividida a entidade Parte Operativa, inclusive para a descrição sem *pipeline*.

Em relação ao número de *latches* utilizados, pode-se verificar que estes estão em número razoável, e aparecem mais na partes *oper_pc* e *oper_ula.* Este resultado se verifica pelo fato de ULA possuir registradores auxiliares para realização das operações.

Em relação a implementação da Parte Operativa, pode-se verificar nas tabelas 5.1.1 e 5.1.2 que na descrição com *pipeline* tem-se um total de 1124 portas lógicas. Já para a descrição sem *pipeline*, necessita-se 1107 porta lógicas. Esta diferença está presente na entidade *oper_rom*, pois, como se pode observar, *oper_rom* com *pipeline* necessita de 41 portas lógicas, contra 24 de *oper_rom* sem *pipeline*. Isto ocorre por que, nesta entidade tem-se dois registradores a mais, necessários para a implementação do *pipeline* para o 8051.

```
with OP_BUS select
    InBus <= guarded DRAM_OUT  when B"000",
                memow2         when B"001",
                TempAcc        when B"010",
                Acumulator     when B"011",
                out_alu        when B"100",
                InBus          when others;
```

FIGURA 5.1.2 - Entradas para o Registrador InBus

```
with OP_ACU select
        Acumulator  <= guarded X"00"          when B"000",
                       out_alu                when B"001",
                       MEMO                   when B"010",
                       TempAcc                when B"011",
                       PC (15 downto 8)       when B"100";
                       PC (7  downto 0)       when B"101",
                       DRAM_OUT               when B"110",
                       Acumulator             when others;
```

FIGURA 5.1.3 - Entradas para o Registrador Acumulador.

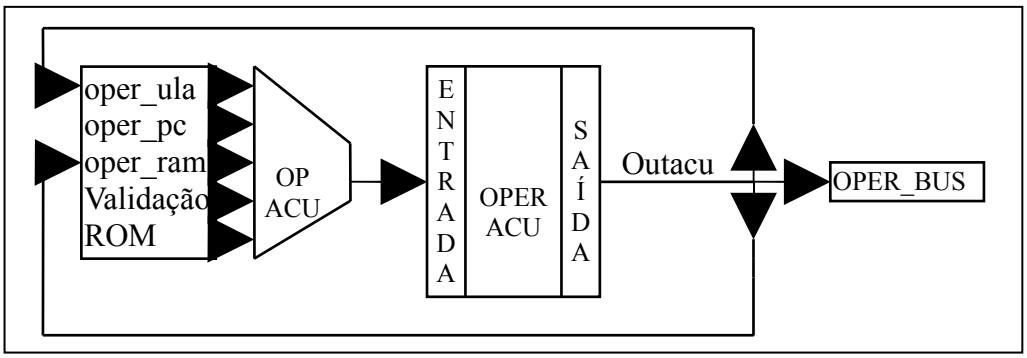


FIGURA 5.1.4 - Entradas e saídas da entidade *oper_acu*

Para a implementação do 8051 completo, verifica-se que para a descrição com *pipeline* necessita-se de um número menor de portas lógicas do que para a descrição sem *pipeline*. Isto ocorre devido as entidades Parte de Validação e Máquina de Estados serem menores na descrição com *pipeline.* A Máquina de Estados é menor pois possui menos estados do que a Máquina de Estados da descrição sem *pipeline*. Isto ocorre por que sem *pipeline* tem-se 24 estados, e com *pipeline* 11. Como conseqüência da Máquina de Estados com *pipeline* possuir menos estados, a entidade Parte de Validação possui um circuito combinacional menor para a geração de microcódigo, o que ocasiona, um número menor de portas lógicas para essa entidade na implementação com *pipeline*.

Devido as entidades Máquina de Estados e Parte de Validação serem menores na descrição com *pipeline,* mesmo com a introdução de registadores auxiliares na

Parte Operativa, verifica-se que houve uma economia em termos de portas lógicas na descrição com *pipeline*.

5.2 Síntese do MCS8051 no Altera

Para realizar a síntese lógica no Sistema Altera, o trabalho necessário para a realização desta tarefa foi muito menor, em comparação com a ferramenta Logic do Alliance. Isto se deve ao fato de que a ferramenta para síntese lógica do Altera suportou o tamanho das entidades como especificado originalmente. Desta maneira, não foi necessário que se efetuasse a divisão das entidades maiores da descrição como ocorreu com a PO e a Parte de Validação no Alliance.

Para se realizar a síntese com o Altera foi necessário somente que se convertesse a descrição Alliance - VHDL em uma descrição compatível com o Compilador VHDL da Altera. A única restrição encontrada neste sentido diz respeito ao fato de que na descrição Alliance muitas vezes encontram-se sinais sendo atribuídos a um registrador em diferentes blocos através do comando *guarded*. Isto não é aceito pelo Altera de modo que foi necessário que, quando ocorriam estas situações, o registrador fosse substituído por outros auxiliares e em seguida multiplexado em um único para posterior utilização. Por exemplo, na Parte de Validação, tem-se como saída uma palavra que corresponde aos sinais de controle (microcódigo) enviados à Parte Operativa. Como visto no capítulo 3, a Parte de Validação foi implementada utilizando-se um bloco VHDL para cada instrução. Desta maneira na decodificação e geração do controle de cada instrução tem-se a atribuição da palavra de controle para o sinal *output_ control* que será enviado à Parte Operativa. Isto pode ser visto na figura 5.2.1, como exemplo para as instruções de incremento e decremento.

Como pode ser visualizado na figura 5.2.1 acima, a palavra *output_control* recebe duas atribuições em dois blocos diferentes através da palavra chave *guarded* que não é aceita pelo Altera. Esta situação não é permitida no ambiente Altera. Para solucionar este problema, optou-se pela susbtituição desta palavra, por tantas quantas fossem necessárias, ou seja, o número de instruções implementadas, e em seguida reuni-las novamente em uma só através de um multiplexador. Deste modo, este arquivo contendo a entidade Parte de Validação da descrição do 8051, ficou como mostra a figura 5.2.2, para as mesmas instruções de incremento e decremento.

Sendo esta a única restrição, o processo de síntese tornou-se bastante simples, pois não se teve para o Altera o trabalho de dividir-se as entidades e reconectá-las novamente através das entradas e saídas. Com isso, foi necessário apenas compilar a descrição VHDL modificada e investigar os resultados da Síntese Lógica no relatório fornecido pela ferramenta. A figura 5.2.3 mostra como as várias palavras *out_control_**n*** foram novamente reunidas através de um MUX.

5.2.1 Resultados da Síntese Lógica no Altera

Os resultados obtidos com a ferramenta de Síntese Lógica do Altera, podem ser visualizados nas Tabelas 5.2.1 e 5.2.2, para as versões da descrição do 8051 com e sem *pipeline.*

Em relação às tabelas 5.2.1 e 5.2.2, ressalta-se que o resultado fornecido pelo Altera diz respeito ao número de *logic elements* utilizados em cada chip. Para a realização de uma melhor comparação com o Alliance, foi realizada a conversão do número de *logic elements* para o seu equivalente em portas lógicas.

Pelos mesmos motivos expostos no capítulo 3, seção 3.2, para a Síntese Lógica com o Altera, igualmente verifica-se que foi necessário um número menor de células lógicas para a implementação da descrição com *pipeline.*

Pode-se verificar nas tabelas 5.2.1 e 5.2.2, que a Parte Operativa é maior na descrição com *pipeline* do que a mesma na versão sem *pipeline.* As Partes de Controle, de Validação e a Máquina de Estados são menores na descrição com *pipeline,* em relação a versão sem *pipeline.* Dessa maneira, também com o Altera obteve-se uma implementação mais econômica em termos de número de células lógicas para a descrição com *pipeline.*

Em termos de portas lógicas (gates) necessárias para a implementação do 8051, em relação as duas ferramentas, verifica-se o seguinte: na descrição com *pipeline,* a Máquina de Estados necessita de 336 portas no Altera contra 93 portas no Alliance. Para implementar a Parte de Controle, o Altera necessita 1105 portas lógicas, contra 135 do Alliance. Para a Parte de Validação, o Altera necessita 2821 contra 476 do Alliance, mesmo com a inclusão da entidade *exit_valid* a mais que não está presente na Síntese com o Altera. Para a descrição sem *pipeline* verifica-se a mesma proporção, de modo que torna-se possível concluir que a implementação no Altera irá consumir um número maior de portas lógicas do que a implementação no Alliance. Porém, cabe ressaltar que esses resultados não são definitivos, tendo em vista a descrição utilizada para a síntese no Altera foi inicialmente projetada para ser executada no Alliance.

Desse modo, para ter-se um resultado mais adequado, seria necessário que se reescrevesse a descrição para a linguagem AHDL do Altera, que por ser mais adequada a este ambiente, possivelmente traria resultados mais eficientes. Além do mais, cabe ressaltar que essa diferença do número de portas lógicas entre as duas ferramentas, também se reflete no fato de que o Altera utiliza muitas portas para a sua programabilidade.

Outro fato interessante a ser observado é a ausência de Flip_Flops na síntese com o Altera. Este fato pode ser creditado a não perfeita adaptabilidade da linguagem Alliance-VHDL ao Altera. Conclui-se então que o *software* de síntese da Altera interpretou a descrição como sendo uma descrição que utiliza latches ao invés de Flip-Flops.

A figura 5.1.5 ilustra essa situação através do código VHDL de exit_valid.

```
output_control <= guarded    (palavra de microcódigo)
        valid1
                when   mov_rA  | mov_ri   | dec_r    | inc_r    | mov_inA,
                  (Código das instruções)
        valid2
                when   mov_Ain | mov_Ar   | xch_Ar   | mov_d_in | clr_A,
        valid3
                when   lcall   | mov_in_d | mov_in_i | orl_Ar   | ret,
        output_control
                when others;
```

FIGURA 5.1.5 - Multiplexador para o envio do microcódigo à Parte Operativa.

TABELA 5.1.1 - Resultados da Síntese Lógica no Alliance para a descrição com *pipeline* do 8051

	latches	**número de células**	**número de portas**
Parte de Controle	8	20	135
Máquina de Estados	4	19	93
Oper_acu	16	8	136
Oper_pc	64	14	262
Oper_sp	8	2	9
Oper_ram	18	12	87
Oper_rom	40	3	41
Oper_ula	65	23	491
Oper_bus	8	10	98
Valid1	19	11	53
Valid2	14	9	72
Valid3	30	17	118
Exit_valid	30	20	256
Total	324	168	1851

TABELA 5.1.2 - Resultados da Síntese Lógica no Alliance para a descrição sem *pipeline* do 8051

	latches	**número de células**	**número de portas**
Parte de Controle	0	17	136
Máquina de Estados	5	16	168
Oper_acu	16	8	136
Oper_pc	64	14	262
Oper_sp	8	2	9
Oper_rom	24	1	24
Oper_ram	18	12	87
Oper_ula	64	23	491
Oper_bus	8	10	98
Valid1	19	12	69
Valid2	14	13	84
Valid3	30	16	129
Exit_valid	30	20	256
Total	300	164	1949

```
 inc_r_block: block (INSTRUCTION = inc_r)
begin
 with ST select
 output_control <= guarded
            B"0000001010000000000" when B"00000", -- T0 = READ_RAM
            B"0000010000001000000" when B"00001", -- T1 = INC
            B"0000000111000001100" when B"00010", -- T2 = WRITE_RAM,
            B"0000000000000000000" when others
 end block;


dec_r_block: block (INSTRUCTION = dec_r)
begin
  with ST select
  output_control <= guarded
            B"0000001010000000000" when B"00000", -- T0 = READ_RAM
            B"0010110000001000000" when B"00001", -- T1 = DEC
            B"0000000111000001100" when B"00010", -- T2 = WRITE_RAM
            B"0000000000000000000" when others;
end block;
```

FIGURA 5.2.1 - Código VHDL de duas instruções na Parte de Validação

```
dec_r_block: block (INSTRUCTION = dec_r)
 begin
   with ST select
   out_control_2 <=
            B"00000010100000000000" when B"00011", -- T0 = READ_RAM
            B"00000000000000000010" when B"00100", -- T1 = DEC
            B"00101100000000000000" when B"00101", -- T2 = WRITE_RAM
            B"00000000000000000000" when others;
 end block;

 inc_r_block: block (INSTRUCTION = inc_r)
 begin
   with ST select
   out_control_3 <=
            B"00000101000000000000" when B"00011", -- T0 = INC_RAM
            B"00000000000000100000" when B"00100", -- T1 = INC
            B"00001000000000000000" when B"00101", -- T2 = WRITE_RAM
            end block;
```

FIGURA 5.2.2 - Código VHDL da Parte de Validação adaptado ao Altera

```
output_control <=   out_control_0 when INSTRUCTION = mov_rA    else
                    out_control_1 when INSTRUCTION = mov_ri     else
                    out_control_2 when INSTRUCTION = dec_r      else
                    out_control_3 when INSTRUCTION = inc_r      else
                    B"000000000000000";
```

FIGURA 5.2.3 - Multiplexador com a palavra de microcódigo enviada à Parte Operativa.

TABELA 5.2.1 - Resultados do processo de Síntese Lógica no Altera para a descrição do 8051 sem *pipeline*

	chip	**E/S**	**Células**	**Portas**	**% de utilização**	**FFs**
Parte de Validação	8452	13/30	251	2988	76% das 336	0
Parte de Controle	8282	13/14	117	1406	56% das 208	0
Parte Operativa	8636	39/42	477	5678	95% das 504	0
Máquina de Estados	8282	13/5	73	877	35% das 208	0
	Total	78/91	918	10949		

TABELA 5.2.2 - Resultados do processo de Síntese Lógica no Altera para a descrição do 8051 com *pipeline*

	chip	**E/S**	**Células**	**Portas**	**% de utilização**	**FFs**
Parte de Validação	8452	13/30	237	2821	70% das 336	0
Parte de Controle	8282	14/15	92	1105	44% das 208	0
Parte Operativa	8636	39/42	485	5773	96% das 504	0
Máquina de Estados	8282	3/5	25	336	11% das 208	4
	Total	68/92	842	10035		

6 O Compilador C para o MCS8051 ASIP

A idéia de criar um Compilador para um Processador de Aplicações Específicas surge da necessidade de tornar essa arquitetura programável em alto nível, característica essa muito desejável nos dias atuais. Para o caso de uma arquitetura comercial, e com um parque de *software* instalado, torna-se evidente a necessidade da adaptação de um Compilador existente para essa arquitetura, ou a criação de um novo, para manter uma certa compatibilidade de *software*, como foi constatado em [CAR 96]. Essa compatibilidade permite que as aplicações desenvolvidas para um certo processador sejam executadas na nova arquitetura, mesmo que sejam realizadas modificações no seu conjunto de instruções para, por exemplo, adaptá-la a uma aplicação específica.

Um compilador pode servir como entrada de um sistema de suporte ao projeto de uma arquitetura dedicada, uma vez que, durante o processo de compilação torna-se possível a escolha das instruções mais utilizadas por uma determinada aplicação, ou ainda, a escolha das rotinas críticas da aplicação [ALB 96]. A partir dessa informação, uma arquitetura dedicada à essa aplicação pode ser implementada tendo o seu conjunto de instruções composto por somente um número pré-determinado de instruções que são as mais necessárias à sua execução, ou seja, as mais utilizadas. A missão do compilador neste caso é, a partir da análise das instruções mais utilizadas, gerar um código em linguagem montadora, composto somente pelas instruções mais necessárias à execução da aplicação.

A vantagem de implementar-se apenas um conjunto reduzido de instruções pode ser encontrada dentro da filosofia de projeto de arquiteturas RISC. Máquinas RISC, por possuírem um conjunto pequeno de instruções simples, permitem a decodificação diretamente em hardware (*hardwired*), sem a necessidade da utilização de uma memória microprogramada [HEN 90] ou de estruturas complexa de hardware. Esta estratégia garante um grande ganho em termos de velocidade na decodificação e geração do microcódigo. As máquinas RISC conseguem decodificar as suas instruções com um baixo custo de HW, devido ao fato de que poucas instruções demandam poucos circuitos de decodificação. Além disto, estes circuitos não são muito complexos, pelo fato das instruções serem simples e, por consequência, necessitarem de poucas palavras de microcódigo para completarem a sua execução. Dizendo de outra forma, poucos circuitos traduzem-se em área reduzida de silício e maior velocidade.

Através de estudo realizado por [CAR 96], verificou-se que a maior área ocupada num processador diz respeito exatamente a Parte de Controle, responsável, entre outras coisas, pela decodificação e geração de microcódigo para a execução das instruções. Esta área está relacionada diretamente ao número de instruções implementadas devido ao número de circuitos decodificadores e geradores de microcódigo necessários. Desse modo, conclui-se que um bom caminho a seguir ao desejar-se reduzir a área de um processador é a redução de seu conjunto de instruções. Essa idéia foi aplicada a uma máquina CISC, o MCS8051, de modo a adaptá-la para a execução de aplicações específicas, justamente pela implementação de um conjunto reduzido de instruções, o qual é dedicado à aplicação. Desse modo tem-se um 8051 ASIP, como explicado no capítulo 2. Igualmente foi implementado um *pipeline* de

instruções para essa arquitetura com a finalidade de compensar um possível tempo maior gasto para a execução do código RISC (ver capítulo 4).

Como o MCS8051 é um processador comercial, torna-se necessária uma compatibilidade de SW com as aplicações já desenvolvidas para essa arquitetura. Dessa maneira, pensou-se em adaptar um Compilador C desenvolvido para o 8051 no GME-UFRGS, denominado CCC51 (*C Cross Compiler to MCS8051*) [MED 91]. Este Compilador modificado tem como finalidade servir como entrada para a criação de uma arquitetura ASIP, mantendo a compatibilidade de SW. O Compilador então, deve ser capaz de, a partir de análises realizadas sobre uma determinada aplicação, gerar um código em linguagem montadora composto somente pelas instruções mais utilizadas pela aplicação. Estas instruções constituirão um subconjunto do conjunto de instruções original do 8051, e serão obtidas a partir de uma análise estática realizada por uma analisador de *Assembly* desenvolvido para esta finalidade no GME-UFRGS [CAR 96b]. Além disso, o Compilador deve procurar evitar a implementação das instruções que não executam sob um ciclo normal do *pipeline*, a fim de evitar quebras [HUA 94]. Como complemento, o Compilador poderá gerar a Parte de Validação da descrição VHDL do 8051, contendo as instruções especificadas para a aplicação em questão e que serão sintetizadas e prototipadas em FPGA, juntamente com as partes restantes da descrição (ver capítulo 3). Com um conjunto de instruções reduzido, tem-se uma área excedente em relação a implementação do 8051 com o conjunto de instruções completo [CAR 96b], sendo que esta área será destinada à integração do sistema específico.

6.1 Alterações realizadas sobre o Compilador CCC51

A idéia inicial concebida para adaptar o CCC51 às aplicações específicas dizia respeito a que o Compilador receberia como entrada, além do programa fonte em C, um arquivo contendo as instruções a serem implementadas, ou seja, o conjunto de instruções desejável para cada aplicação. Esse arquivo é criado com base nos resultados obtidos pelas ferramentas de análise estática da aplicação (ver capítulo 3). Dessa forma, o compilador poderia ler este arquivo e gerar um código em linguagem montadora contendo apenas as instruções especificadas neste arquivo. Além disso, caso existissem neste arquivo instruções que provocassem a quebra do Pipeline, o Compilador deveria substituí-las por outras que mantivessem o *pipeline* cheio, sempre que possível, sem, entretanto, aumentar o número de ciclos.

Para concretizar a idéia apresentada no parágrafo acima, tornou-se necessário a criação de algumas rotinas em C, além de dois tipos de dado, os quais foram chamados de *instrução* e *operando*. Estes tipos de dados foram criados para permitir a troca das instruções. Como uma instrução deveria ser alterada, pensou-se em criar um tipo de dado que refletisse as partes de uma instrução: o tipo e os modos como os operandos são endereçados.Pensou-se então em separar as rotinas em dois grupos, de acordo com as alterações possíveis de serem realizadas sobre uma instrução, ou seja, alteração do *tipo* da instrução ou do *modo de endereçamento* dos operandos. Além destes dois grupos, há ainda um terceiro grupo, composto por rotinas que possuem a função de auxiliarem as rotinas que realizam as alterações nas instruções. A Tabela 6.1.1 mostra os nomes das funções implementadas, uma breve descrição de sua operação e o grupo ao qual pertence: tipo, operando ou auxiliar.

TABELA 6.1.1 - Rotinas C para a adaptar o CCC51 às aplicações específicas

Rotina	Função	Grupo
create_inst	cria instrução	auxiliar
change_oper	altera modo de endereçamento do operando	operando
write_inst	escreve a inst. no arquivo *assembly* se a inst. for válida	operando
inclui	inclui instrução na lista	auxiliar
init_inst	inicializa instrução	auxiliar
exists	verfica se instrução está na lista	operando
create_list	cria a lista com as inst. do arquivo de entrada	auxiliar

A necessidade da criação de um novo tipo de dado refere-se ao fato de que a maioria das rotinas citadas acima necessita modificar uma instrução. Dessa maneira, é mais conveniente criar um novo tipo de dado, de modo que esse tipo de dado reflita os campos de uma instrução, ou seja: o *tipo* da instrução e cada um de seus *operandos*, se houver. Para uma melhor manipulação dos operandos, foi criado um segundo tipo de dado, chamado *operando*, que contém o modo de endereçamento do operando, e o registrador ou a posição de memória utilizada. Caso o operando fosse uma constante ou um endereço de salto, o tipo operando conteria apenas esse valor.

Para reunir todas as instruções a serem implementadas no conjunto de instruções em uma única estrutura de dados, foi criada uma lista simplesmente encadeada formada por nodos do tipo instrução. Dessa forma, a tarefa da rotina *create_list* citada na tabela 6.1.1, consistia em ler as instruções do arquivo de entrada contendo as instruções da aplicação, e criar uma lista encadeada de modo a que cada nodo desta lista correspondesse a uma instrução do arquivo. Esta lista poderia ser utilizada pelas outras funções para, por exemplo, comparar se a instrução que estava sendo analisada poderia ser implementada, bastando para isso procurar por esta instrução na lista das instruções gerada em *create_list*.

Pode-se verificar na tabela 6.1.1 que não está especificada nenhuma rotina para alteração do tipo das instruções, pois optou-se primeiramente pela criação das rotinas referentes ao grupo *operando*. Justifica-se esta opção pelo fato de que muitas instruções do conjunto de instruções do 8051 diferenciam-se entre si, por possuírem diferentes modos de endereçamento dos operandos. Desse modo, a estratégia de reduzir o número de modos de endereçamento dos operandos, causaria uma diminuição do número de instruções diferentes. Por exemplo, existem especificadas 20 instruções diferentes para movimentação de dados (MOV), cuja única diferença entre uma e outra é a maneira pela qual cada uma endereça os seus operandos. Através da substituição de alguns modos de endereçamento por outros, torna-se possível a eliminação de muitas instruções do tipo MOV. O mesmo ocorre com a maioria das outras instruções.

Assim, o objetivo das rotinas do grupo *operando* é reduzir ao máximo o número de diferentes modos de endereçamento dos operandos. Se um operando for uma constante ou um endereço de salto, a substituição não é possível, porém, é possível que se substituam os modos *direto*, *indireto* e a *registrador.*

Após implementadas e testadas as rotinas mostradas na tabela 6.1.1, verificou-se que os resultados obtidos não estavam acontecendo dentro do previsto. Com os exemplos sobre os quais executaram estas rotinas, o número de diferentes modos de endereçamento era praticamente igual ao obtido sem nenhuma alteração no compilador. Um motivo que pode ter levado a esses resultados pode ter sido o fato de que as rotinas do grupo *operando* operavam com um operando por vez, ou ainda,

talvez essas rotinas deveriam operar em conjunto com as rotinas para substituição do tipo das instruções. Porém, o principal motivo que, acredita-se, tenha levado aos resultados, é a dificuldade de se implementar um algoritmo realmente eficiente para a substituição dos modos de endereçamento dos operandos em tempo de compilação. Desse modo esta idéia foi momentaneamente abandonada, de forma que as rotinas para alteração do tipo das instruções no momento ainda não foram implementadas.

Uma alternativa encontrada para obter um conjunto de instruções pequeno para as aplicações foi adotar a estratégia da geração de um código fixo para o mesmo conjunto de instruções relativo ao código intermediário, não importando a aplicação. Em outras palavras, para cada instrução do código intermediário, sempre serão geradas as mesmas instruções em linguagem montadora. Esta nova estratégia pode ser justificada por três motivos:

1. as rotinas mostradas nos parágrafos acima, realizam as suas funções a partir do código intermediário, ou seja, após as análises léxica e sintática. Dessa forma, a geração do código fixo também se fará neste nível;

1. se as análises léxica e sintática garantem sempre a geração do código intermediário, não importanto o código fonte, então um código *assembly* fixo a partir deste código intermediário também pode ser garantido;

1. o objetivo do compilador é a geração de um código *assembly* contendo um número pequeno de instruções diferentes, não importando quais sejam essas instruções.

Assim, é importante que seja gerado um código em linguagem montadora contendo poucas instruções diferentes para diminuir o tamanho da Parte de Validação. Além disso, a maioria das instruções implementadas não devem quebrar o *pipeline*. Isto quer dizer que, um conjunto de instruções pequeno e que aproveite as características do *pipeline* é realmente o que se deseja. A compatibilidade se dá a nível do código intermediário e não a nível da análise estática. Isto é possível porque o código em linguagem montadora utilizado para realizar a análise estática foi gerado a partir do mesmo código intermediário.

6.2 A implementação do CCC51

O CCC51 é composto de 6 módulos para a tradução de um código fonte em C para *Assembly* do 8051:

Análise Léxica;

Análise Sintática;

Manutenção das Tabelas de Constantes, Símbolos e variáveis Intermediárias;

Conversão de Tipos;

Geração de Código Intermediário; e

Análise Semântica e Geração de código em linguagem montadora.

Como os objetivos do CCC51 são a geração de poucas instruções diferentes no código em linguagem montadora e de instruções que não quebrem o *pipeline*, as alterações realizadas estão situadas no módulo de geração do código em linguagem montadora, chamado ccc51cod.c. Este módulo é responsável pela geração de código em linguagem montadora a partir de um código intermediário de três endereços [AHO 88].

1. Independentemente das instruções implementadas, a Parte Operativa não sofrerá modificações. Isto quer dizer que o *hardware* da Parte Operativa será capaz de executar todos os modos de endereçamento dos operandos, não importanto quais as instruções realmente implementadas. Dessa maneira, em certas aplicações poderá haver uma subutilização do MUX que implementa os diferentes modos de endereçamento da RAM, pela não utilização de algumas de suas entradas. Por exemplo, no caso do modo de endereçamento *a registrador* não ser utilizado, torna-se desnecessária a entrada do MUX correspondente ao endereçamento com os 3 bits menos significativos do primeiro byte da instrução. Porém, os objetivos aqui propostos dizem respeito à economia de área, através da redução do tamanho da Parte de Validação, ou seja, a otimização a ser realizada implica em reduzir o número de instruções implementadas. Um caminho para se obter este resultado é modificação dos modos de endereçamento dos operandos. Como consequência, alguns modos de endereçamento poderão não ser utilizados com a única finalidade de reduzir o número de instruções implementadas, e não como forma de otimização da área da Parte Operativa. Entretanto, esta otimização poderá ser realizada no futuro.

   O CCC51 possui definidas, no módulo ccc51cod.c*,* várias funções para a tradução do código intermediário para o código *Assembly*. Cada uma destas funções trata de uma instrução de código intermediário, traduzindo-a para um código em linguagem montadora equivalente. Visto que o código intermediário é composto por 36 instruções, estão definidas 36 funções chamadas de *funções de tratamento de mnemônicos*. O processo de tradução do código intermediário para o código em linguagem montadora realiza-se através de função que lê o arquivo contendo o código intermediário linha por linha e, de acordo com a instrução encontrada, realiza a chamada a função de tratamento de mnemônico correspondente àquela instrução. Cada função para tratamento de mnemônicos recebe como parâmetro um ponteiro para uma estrutura chamada *t_param*, cujas variáveis são ponteiros para as tabelas de Símbolos, Constantes e Temporárias.

   Na especificação original do CCC51, as funções de tratamento de mnemônicos geram o código *Assembly* através da escrita direta das instruções no arquivo *assembler* de saída. Porém, antes dessa escrita poder ser realizada é necessário que se determinem os endereços dos operandos. Com a finalidade de realizar esta tarefa, foi criada uma função chamada *TraduzOpUn* (Traduz Operação Unária). O protótipo C para esta função é:

*char *TraduzOpUn (char *oper, t_var **tabsimb, t_tabconst *tabconst, t_tabtemp**tabtemp);*

Essa função recebe como entrada um operando do código intermediário (*oper*) e aloca um endereço para este operando. Os outros parâmetros para essa função são ponteiros para as tabelas de Símbolos, Temporárias e Constantes, criadas durante as análises Léxica e Sintática. Essas tabelas servem para a localização do operando do código intermediário. O operando pode ser um símbolo, uma variável temporária ou uma constante e estará definido na tabela respectiva (de Símbolos, de Temporárias ou de Constantes). A função *TraduzOpUn* retorna o espaço alocado para o operando: registrador, posição de memória ou o valor da constante. Para as instruções de dois operandos, está especificada outra função chamada *TraduzOpBin,* que realiza a tradução de dois operandos através de duas chamadas a *TraduzOpUn*.

Seguindo-se a primeira estratégia adotada para alterar a geração das instruções, as rotinas para alteração dos modos de endereçamento dos operandos citadas na tabela 6.1.1 foram implementadas no módulo ccc51cod.c e interagiam com a função TraduzOpUn da seguinte forma: uma vez que TraduzOpUn detectasse o tipo do operando (Símbolo, Temporária ou Constante) e alocasse um modo para endereçá-lo, as rotinas criadas procuravam torná-lo compatível com o conjunto de instruções desejados. Primeiramente, era verificado se a instrução com o operando estava presente na lista de instruções criadas pela rotina *create_list*. Em caso afirmativo, não havia mais nada a fazer, pois essa era uma instrução compatível com o conjunto de instruções desejado. Caso contrário, era realizada uma chamada a rotina *change_oper* que procurava modificar a maneira pela qual o operando em questão estava sendo endereçado. Esse processo repetia-se até se achar uma instrução compatível ou não haver mais instruções do tipo em questão na lista com o conjunto de instruções desejado. Como visto na seção 6.1, as rotinas para alteração do tipo da instrução, no momento não estão implementadas.

Para a estratégia da geração do código fixo em linguagem montadora foram realizadas todas as alterações possíveis para uma instrução como mencionado na seção 6.1, ou seja, procurou-se alterar o *tipo* das instruções e os *modos de endereçamento* dos operandos dessas instruções. Para estas alterações foi desnecessária a criação de rotinas específicas, pois as alterações foram efetuadas diretamente nos códigos das funções *TraduzOpUn* e de *tratamento de mnemônicos*.

Em relação ao tipo das instruções, procurou-se implementar o menor número de instruções diferentes e instruções que não provocassem a quebra do *pipeline*. Para tanto, foram efetuadas as substituições mostradas na tabela 6.2.1.

Pode-se verificar na tabela 6.2.1 que ao lado de cada instrução está indicado se a instrução provoca uma quebra no *pipeline* ou não, e na última coluna é mostrado o custo em termos do número de instruções a mais que foram utilizadas para realizar a substituição. Por exemplo, o custo para a susbtituição da instrução de salto DJNZ foi de 2 instruções na substituição pela instrução JNZ.

TABELA 6.2.1 - Instruções substituídas no CCC51 - I

Instrução original	quebra o pipeline	Instrução(ões) substitutas	quebra(m) o pipeline	custo
INC A	não	ADD A,#01	não	0
DEC A	não	SUBB A,#01	não	0

DJNZ rel	não	MOV A, Rn
		SUBB A, #01
		JNZ rel
			não	2

Em termos do custo de memória, verifica-se que a instrução DJNZ possui o tamanho de 2 bytes, enquanto que o número de bytes das instruções que a susbtituíram, é igual a 4. A instrução DJNZ decrementa um Registrador e salta se o registrador permanecer diferente de zero. Portanto, na substituição realizada o conteúdo desse registrador é movido para o Acumulador, sendo este decrementado. Em seguida é executada a instrução JNZ que realiza o salto se o Acumulador for diferente de zero. Em relação a instruções de salto, o CCC51 - I implementa apenas as instruções JZ, JNZ e JMP. Nenhuma destas instruções quebram o *pipeline.*

Além destas substituições, a tabela 6.2.1 nos mostra que foram eliminidas do conjunto de instruções as instruções de incremento e decremento do Acumulador, INC A e DEC A, mesmo que estas instruções não quebram o *pipeline*. As trocas mostradas na tabela 6.2.1 foram realizadas apenas com a finalidade de reduzir o tamanho do conjunto de instruções.

Das instruções que permaneceram no conjunto de instruções, seis quebram o *pipeline*. São elas: LCALL (chamada a sub-rotina), RET (retorna de sub-rotina), DIV (divisão), MUL (Multiplicação), MOV *direct, direct* e MOV *direct, const*. As instruções de divisão e multiplicação serão substituídas por sub-rotinas com funções equivalentes. Esta operação terá um custo em termos de número de instruções igual ao tamanho das sub-rotinas, que estima-se seja em torno de 15 a 20 instruções. Desse modo, cada instrução MUL ou DIV do código em linguagem montadora será substituída por uma chamada a sub-rotina equivalente. Isto previne que o tamanho do código em linguagem montadora venha a tornar-se muito grande. Por outro lado, esta medida não tem efeito se as instruções de chamada e retorno a sub-rotina LCALL e RET, também quebrarem o *pipeline*. Para estas instruções existem duas soluções. A primeira diz respeito a criação de instruções de movimentação (MOV) que possam manipular diretamente o Contador de Programas (PC) e o Apontador de Pilha (SP). Estas instruções de movimentação não iriam quebrar o *pipeline* e realizariam as mesmas funções que as instruções LCALL e RET. O custo neste caso, seria a inclusão de algumas instruções a mais no conjunto de instruções o que não é desejável, mesmo que estas estariam substituindo LCALL e RET. Uma segunda solução seria a realização de uma mudança no modo como estas instruções são decodificadas, da seguinte maneira: substituir o decodificador das instruções LCALL e RET por decodificadores mais simples, sendo cada um responsável pela realização de alguma tarefa na execução da instrução. O resultado seria a geração de um microcódigo que não quebraria o *pipeline*. Mesmo que o número total de ciclos para a execução das tarefas destes decodificadores menores seja semelhante ao decodificar atual para estas instruções, ainda assim teria-se a vantagem de não precisar recarregar novamente o *pipeline* uma vez que este não será quebrado. As otimizações para as instruções MOV

que quebram o *pipeline* são discutidas na próxima seção. As outras instruções que posuem três bytes e quebram o *pipeline* não estão implementadas.

O modo como os operandos de uma instrução são endereçados determinam o tamanho da instrução. Desse modo, para as alterações sobre os modos de endereçamento dos operandos, optou-se por deixar por conta do usuário a decisão entre os modos *direto*, que implica em uma instrução de dois bytes, e o endereçamento a *registrador*, que pode demandar uma instrução de apenas um byte. Assim, de acordo com a opção do usuário, o compilador procurará priorizar um determinado modo de endereçamento. Como o 8051 possui apenas 8 registradores, é pouco provável que se consiga executar um programa inteiro contendo apenas o modo de endereçamento a *registrador*. A troca entre os modos de endereçamento dos operandos *direto* e a *registrador* pode ser feita de maneira direta porque os registradores são na verdade posições de memória. Em relação ao modo de endereçamento *indireto*, optou-se por deixá-lo apenas para a tradução de ponteiros. Uma constante e um endereço de salto não podem ser substituídos e permanecem inalterados.

Às otimizações realizadas sobre os tipos das instruções e sobre os modos de endereçamentos dos operandos como explicado nos parágrafos acima, somou-se uma terceira otimização. Esta otimização foi realizada sobre a maneira como o CCC51 original realizava a manipulação da pilha, com a finalidade de diminuir o tamanho do código em linguagem montadora gerado. Na implementação original do CCC51 a pilha era alocada para os símbolos automáticos e para as temporárias, uma vez que não houvessem mais registradores livres. Dessa maneira, o tamanho do código aumentava consideravelmente, pois para cada alocação eram executadas algumas instruções para manipulação de pilha. O problema maior com essa abordagem era o fato de que estava-se utilizando instruções inválidas, ou seja, não presentes no conjunto de instruções do 8051, como por exemplo INC SP. A operação de incremento do Apontador de Pilha é implicitamente realizada pela instrução PUSH. Tendo por base estes problemas, resolveu-se eliminar o uso da pilha para as temporárias e os símbolos. Para tanto, estas variáveis foram mapeadas diretamente na memória principal ou em registradores, sendo a pilha destinada apenas para chamadas a sub-rotinas.

A primeira versão do CCC51 otimizado à aplicações específicas, foi chamado de CCC51 - I. Os resultados obtidos com o CCC51 - I podem ser conferidos na seção seguinte, bem como outras otimizações realizadas sobre o conjunto de instruções do CCC51 - I.

6.3 Resultados obtidos

Para o CCC51 procurou-se verificar o tamanho do código gerado, o número de instruções diferentes utilizadas em cada aplicação testada e a porcentagem de quebra do *pipeline*.

Como base de comparação para os testes foi adotado um compilador comercial para o 8051, o Keil C Compiler [BIT 95]. Optou-se por um compilador comercial principalmente por este não possuir qualquer vínculo com esta abordagem e também para poder-se realizar os testes com um ambiente real.

Foram escolhidos programas simples e programas que refletem aplicações industriais, desenvolvidos no Grupo de Microeletrônica e no Departamento de Engenharia Elétrica da UFRGS. São aplicações encontradas na indústria local e que poderão vir a serem utilizadas, caso o CCC51 apresente resultados aceitáveis.

As aplicações testadas são as seguintes:

Blackjack: jogo do “resta-um”;

Elevador: controla um elevador;

Encoder: codifica sinais para transmissão;

Decoder: decodifica sinais para transmissão;

Elipse: implementa um filtro elíptico;

Motor: implementa o controle de um motor de indução; e

Equações Diferenciais: cálculo de uma equação diferencial.

Antes de analisarmos os resultados obtidos, cabe ressaltar que foram realizadas mais duas otimizações no CCC51 - I. Estas otimizações dizem respeito as instruções de movimentação de dados (MOV). Este tipo de instrução é a que mais aparece em um programa em linguagem montadora, como foi constatado em [CAR 96]. A primeira otimização foi a eliminação das instruções MOV *direct*, *direct* e MOV *direct, const* onde tem-se os dois operandos endereçados como uma posição da memória ou como uma posição de memória e uma constante respectivamente. Estas instruções por possuírem 3 bytes quebram o *pipeline*. Assim, estas instruções foram susbtituídas, como mostra a tabela 6.3.1.

TABELA 6.3.1 - Substituição das instruções MOV que quebram o *pipeline*

Inst. Que quebra o *pipe*	Instruções substitutas	custo
MOV *direct1, direct2*	MOV A, *direct2* MOV *direct1*, A	1
MOV *direct, const*	MOV A, *const* MOV *direct*, A	1

As substituições realizadas com as instruções MOV mostradas na tabela 6.3.1, resultaram num CCC51 com duas instruções a menos no conjunto de instruções, além do que estas instruções provocavam uma quebra no *pipeline*, e de fato necessitavam ser eliminadas. O custo em termos de memória é de 1 byte, pois tem-se uma instrução de 3 bytes sendo substituída por duas de 2 bytes. O resultado é o CCC51 - II. Observa-se ainda na tabela 6.3.1, que o custo em que resultaram as substituições, é de uma instrução a mais presente no código gerado em linguagem montadora para cada instrução MOV de três bytes presente originalmente neste código.

Ainda no CCC51 - II constatou-se um número considerável de diferentes instruções do tipo MOV. Assim, resolveu-se eliminar todas as instruções do tipo MOV que não possuíssem o Acumulador como um de seus operandos. Como

resultado, criou-se o CCC51 - III que gera um conjunto de instruções contendo apenas as instruções do tipo MOV mostradas na tabela 6.3.2.

TABELA 6.3.2 - Instruções de movimentação implementadas no CCC51 - III

Instrução	operando 1	operando 2
MOV	A	*direct*
MOV	A	*indirect*
MOV	A	*register*
MOV	*indirect*	A
MOV	*register*	A
MOV	*direct*	A

A tabela 6.3.3 mostra os resultados obtidos em termos do número de instruções que cada compilador necessitou para implementar as aplicações em linguagem montadora.

Pode-se verificar na tabela 6.3.3 que, devido as otimizações realizadas no sentido de retirar instruções para gerar um código com um conjunto de instruções reduzido, tem-se um aumento gradativo no tamanho do código gerado em linguagem montadora. Isto ocorre pelas emulações que foram feitas e comprova o custo verificado quando destas emulações, como pode ser observado nas tabelas 6.2.1 e 6.3.1.

Observa-se ainda que o custo em termos do tamanho do programa gerado, está dentro de um limite aceitável em torno de 30% para o compilador mais otimizado, CCC51 - III. Diz-se aceitável porque com a implementação do *pipeline*, tem-se um aumento de velocidade na execução das instruções da ordem de quatro vezes. Caso um programa possua um código 30% maior do que o original, algumas quebras de *pipeline* poderão ser suportadas. As excessões dizem respeito aos programas *Elipse* e *Motor*. No primeiro caso, houve um aumento demasiado do tamanho do código, da ordem de 3 vezes para o CCC51 - I e 5 vezes para o CCC51 - III. Isto ocorreu porque o código fonte em C do programa *elipse* é composto por diversas operações com vetores, sendo que este exemplo nos serviu para comprovar que a geração de código em linguagem montadora para vetores, necessariamente precisa passar por um processo de otimização. Por outro lado, o programa do *motor* mostrou-se muito menor do que o seu equivalente compilado no Keil C compiler. Mesmo na versão mais otimizada, o CCC51 - III, tem-se um código gerado em torno de 13% menor.

O número de instruções diferentes utilizadas por cada compilador pode ser visualizado na tabela 6.3.4. Nesta tabela é realizada uma comparação entre as três versões otimizadas do CCC51 com a sua versão original.

TABELA 6.3.3 - Número de instruções em linguagem montadora necessárias para implementar as aplicações

Aplicação	Keil	%	CCC51 - I	%	CCC51 - II	%	CCC51 - III	%
Blackjack	91	100	90	98.9	102	112.08	103	113.18
Elevador	114	100	133	116.6	141	123.68	149	130.7
Encoder	608	100	719	118.25	752	123.68	826	135.85
Decoder	666	100	902	135.43	937	140.69	1009	151.5
Motor	845	100	565	66.86	740	87.57	743	87.92
Elipse	309	100	1141	369.25	1650	533.98	1649	533.65
Diffeq	153	100	152	99.34	198	129.41	198	129.41

Observando-se a tabela 6.3.4 percebe-se que a primeira otimização realizada, CCC51 - I, em alguns caso teve desempenho igual ou até inferior a versão original. Isto pode ter ocorrido devido ao uso de muitas instruções diferentes do tipo MOV, além da utilização da substituição de instrução DJNZ, que como visto na tabela 6.2.1 possui um custo de duas instruções. Este pode ser o caso do programa *elevador*. Para as demais aplicações o CCC51 - I apresentou um desempenho superior em relação a versão original em torno de 10%. Mesmo assim, o número de instruções diferentes geradas em alguns casos encontra-se num nível relativamente elevado, como por exempo os programas *motor* e *decoder* que possuem 31 instruções diferentes.

Devido a esse número elevado de instruções diferentes, foram realizadas mais duas otimizações, como citado no início desta seção. As versões II e III do CCC51 apresentaram uma redução significativa do número de instruções diferentes utilizadas por cada aplicação, ficando em torno de 20. Por exemplo, o caso do programa *encoder* que na versão original do CCC51 necessitava em torno de 34 instruções diferentes para ser implementado, já no CCC51 - III necessita apenas 21 instruções diferentes. O custo da redução do número de instruções diferentes mostrado na tabela 6.3.4, pode ser visualizado na tabela 6.3.3, onde verifica-se um aumento gradativo do número de instruções total do programa gerado em linguagem montadora, em função da diminuição do número de instruções diferentes necessárias para implementar cada aplicação.

Verifica-se então que, como discutido anteriormente, esse custo gira em torno de 30% do tamanho do programa em linguagem montadora e que, devido ao *pipeline* de instruções implementado, esse tamanho a mais é compensado em termos de velocidade de execução da aplicação.

TABELA 6.3.4 - Número de instruçõs diferentes em cada aplicação

Aplicação	CCC51 original	CCC51 - I	CCC51 - II	CCC51 - III
Blackjack	24	24	17	17
Elevador	25	27	20	15
Encoder	34	30	27	21
Decoder	38	31	29	23
Motor	36	31	29	24
Diffeq	29	23	19	19
Elipse	22	20	15	14

A implementação do *pipeline* em conjunto com a mudança realizada na temporização original, garantem que o programa executará em um número menor de ciclos do que o seu equivalente executando sem o *pipeline*. Tomemos como exemplo o pior caso, ou seja, uma aplicação em que todas as instruções quebram o *pipeline*. Ressalta-se que este caso nunca ocorerrá no CCC51 - III, pois esta versão do compilador gera as apenas as instruções LCALL, RET, DIV e MUL que quebram o *pipeline*. Mesmo nesse caso, a aplicação executaria num tempo menor devido as mudanças realizadas sobre a temporização. As instruções LCALL e RET executam em 24 ciclos na temporização original, enquanto que na temporização modificada executam em 11 ciclos. As instruções MUL e DIV não estão implementadas até o momento, pois espera-se susbtituí-las por sub-rotinas equivalentes. As demais instruções executam em 12 ciclos na temporização original, enquanto na temporização modificada executam em 6 ciclos.

Assim, mesmo que todas as instruções de uma aplicação quebrassem o *pipeline*, ainda assim a aplicação executaria aproximadamente na metade do tempo comparando com uma execução sem *pipeline*. Desse modo, a aplicação poderia ter o dobro do tamanho e ainda assim executaria no mesmo tempo que levaria para executar sem *pipeline*. Dentre as aplicações testadas, com exceção do programa *elipse*, o pior caso teve um aumento do tamanho do código gerado em linguagem montadora de 51%. Conclui-se então que essa aplicação irá executar em um tempo menor do que se fosse executada sem *pipeline*.

A utilização de um compilador adaptado para geração de código para um processador específico permitiu a redução de número de instruções utilizadas e a consequënte otimização (redução de área de silício) do processador. A otimização visou ainda aproveitar ao máximo o *pipeline* do processador utilizado escolhendo instruções que evitem a quebra do *pipeline*. É necessário continuar o trabalho para otimização da geração de código, pois algumas instruções ainda provocam uma quebra no *pipeline*. Além disso pode-se tentar resultados melhores com a implementação de menos modos de endereçamento dos operandos e a geração de instruções específicas a cada aplicação como proposto originalmente.

7 Conclusões e trabalhos futuros

Os objetivos deste trabalho podem ser resumidos em três diferentes metas. A primeira é a implementação de um *pipeline* de instruções para o 8051, seguido da síntese lógica do processador com e sem *pipeline,* para verificar-se o número de portas lógicas ocupadas por cada descrição, com a finalidade de validar o *pipeline*. Finalmente tem-se a otimização do compilador C para o 8051 chamado CCC51, de modo a ter-se um *software* básico como entrada para um sistema de geração de um 8051 dedicado a aplicações específicas (ASIP), mantendo-se a compatibilidade de SW necessária.

Acredita-se que as três metas citadas no parágrafo acima tenham sido efetivadas. Em primeiro lugar foi implementado um *pipeline* de instruções para o 8051. Este *pipeline* proporcionou um acréscimo da ordem de quatro vezes na velocidade de execução das instruções, juntamente com uma alteração realizada na temporização deste processador.

Após realização da Síntese Lógica, esta comprovou a viabilidade de síntese da descrição VHDL com *pipeline*, de modo que este pôde ser validado. A fim de se ter uma avaliação maior em termos de síntese, o processo de síntese lógica foi aplicado a duas ferramentas com escopos diferentes: o ambiente Alliance, que sintetiza para para *standard cells*, e o conjunto de ferramentas *Altera*, que sintetiza para FPGA's. Os resultados obtidos através das duas abordagens mostraram-nos que a descrição com *pipeline* não apresenta um aumento significativo em termos de portas lógicas necessárias para a sua implementação, em relação a versão sem *pipeline*. Este resultado vem de encontro com os objetivos iniciais, que dizem respeito à otimização do 8051 para aplicações específicas sem um aumento considerável do custo em área. Esta a razão também, da escolha do *pipeline* como técnica de paralelismo para o 8051.

Finalmente, após se ter uma descrição com *pipeline* dentro dos propósitos de custo estabelecidos, passa-se à implementação da otimização do compilador. A necessidade de se otimizar um compilador para a abordagem ASIP do 8051 resulta do fato de que cada aplicação poderá executar sob um conjunto de instruções reduzido, ou seja, as instruções mais importantes para aquela aplicação. Dessa maneira, torna-se viável a redução do conjunto de instruções em relação ao original. Uma vez que se tenha um 8051 otimizado e com um conjunto de instruções reduzido, não é mais permitido a um programador utilizar o conjunto de instruções original para a sua programação. Uma solução seria a reconstrução de um programa montador (*assembler*) adaptado ao novo conjunto de instruções, ou de um compilador. Pelo fato da programação em nível de linguagem montadora constituir uma tarefa cansativa e com fortes tendências ao erro, optou-se pela otimização do compilador. Outro motivo pelo qual se procurou otimizar o compilador refere-se a compatibilidade com as aplicações já escritas em C atualmente em funcionamento.

Os resultados obtidos com o compilador mostraram-se satisfatórios, uma vez que para as aplicações testadas foram necessárias em torno de 20 instruções diferentes para executá-las. Esse número reduzido de instruções vem de encontro com o objetivo maior deste trabalho, que é a integração de sistemas para aplicações específicas (ASIS). Com a implementação de 20 instruções na Parte de Controle, obtém-se uma

área excedente em relação ao processador original. Esta área poderá ser utilizada para a integração de outras partes do sistema.

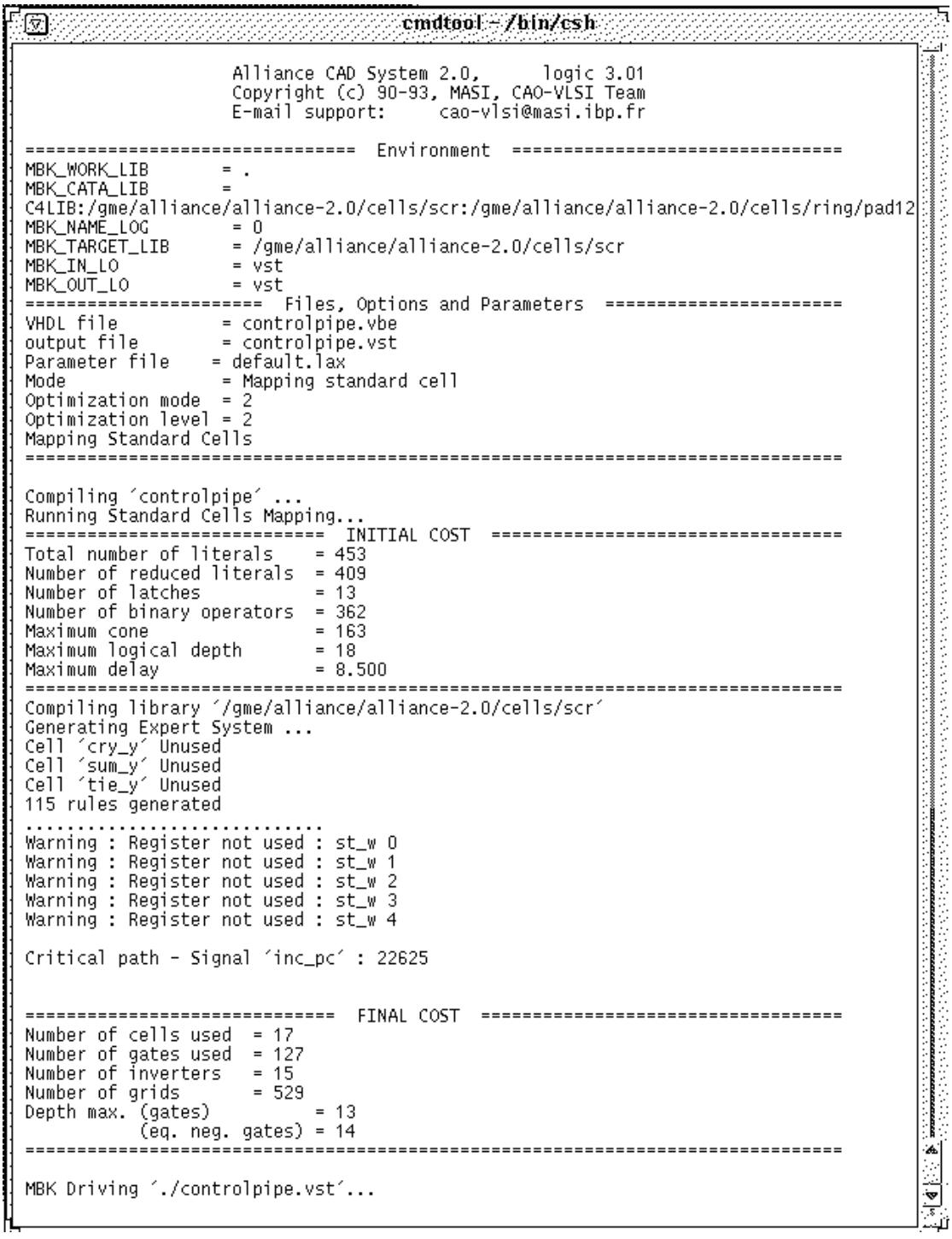
Ainda em relação ao compilador, verifica-se que o custo envolvido na otimização do conjunto de instruções reflete um aumento de aproximadamente 30% do tamanho do código gerado em linguagem montadora. Além disso, a otimização proporcionada pelo compilador não pemitiu que se eliminassem todas as instruções que provocam uma quebra do *pipeline*. Mesmo assim, devido as alterações realizadas sobre a temporização original, garante-se que as aplicações testadas executarão em um tempo menor do que se fossem executadas em um 8051 sem *pipeline*.

Em relação a trabalhos futuros, pode-se citar a integração do compilador com uma ferramenta de síntese que trabalhe com VHDL, como por exemplo *Altera*, para que o usuário possa entrar com a sua aplicação em C, e obter como resultado um protótipo do 8051 otimizado para sua aplicação. A necessidade da ferramenta de síntese ser compatível com VHDL diz respeito ao fato de que a descrição do 8051 está realizada nesta linguagem. Uma outra alternativa seria geração de um código em linguagem montadora contendo somente as instruções mais utilizadas pela aplicação. Neste caso teriam que ser aprimoradas as rotinas para alteração das instruções, mostradas no capítulo 6, além de integrar o compilador com a ferramenta de análise estática.

Seguindo-se um outro caminho, pode-se tentar otimizar a Parte Operativa, o que iria permitir a criação de novas instruções. Estas instruções possivelmente seriam mais adaptadas a cada aplicação, pois estariam sendo criadas com tal propósito e poderiam ser tão simples quanto uma máquina RISC exige. Outra abordagem ainda poderia ser a implementação de um sistema multiprocessado baseado no 8051 otimizado. Finalmente, poder-se-ia unir a abordagem para a criação de sistemas microcontrolados voltados a aplicações específicas aqui apresentada, com a abordagem baseada no Risco [ALB 96], para a criação de um ambiente que ressalte as vantagens e desvantagens de cada uma, auxiliando o usuário na escolha daquela que melhor lhe convir.

Anexo 1 Síntese Lógica no Alliance

A figura abaixo representa uma janela do ambiente SunOS contendo um exemplo de Síntese Lógica com a ferramenta *Logic* do ambiente Alliance. No exemplo mostrado abaixo está-se realizando a síntese da Parte de Controle com *pipeline*, chamada *Controlpipe*. A janela apresenta no final os resultados da Síntese Lógica, e a geração do arquivo estrutural *controlpipe.vst*.



Anexo 2 Simulação no Alliance

A listagem abaixo apresenta um exemplo de simulação de algumas instruções do 8051 com e sem *pipeline* no ambiente Alliance, utilizando-se a ferramenta *asimut*. No ínicio da execução de cada instrução está especificado o nome da instrução e o número de bytes. Observa-se que as instruções LCALL e RET provocam a quebra do *pipeline*, necessitando de 11 ciclos para completarem a sua execução. Pela listagem também é possível perceber que são necessários 6 ciclos para encher o *pipeline*. No ciclo 1 o registrador *inst_at* é atualizado pois a instrução seguinte é carregada no registrador de instruções (IR).

Simulação com *pipeline*:

```
-- description generated by Pat driver v104
--                       date : Tue Jun 10 15:53:29 1997


--               sequence : teste_pipe

-- input / output list :
in       clock B;
in       reset B;
register us1.st_w (4 downto 0) B;
register us2.pc (15 downto 0) X;
register us2.accu (7 downto 0) X;
register us2.reg_ir (7 downto 0) X;
register us2.inst_at (7 downto 0) X;
register us2.sp (7 downto 0) X;
register us5.ram_01 (7 downto 0) X;
register us5.ram_07 (7 downto 0) X;
register us5.ram_10 (7 downto 0) X;
register us2.inbus (7 downto 0) X;
register us2.memow2 (7 downto 0) X;

begin

-- Pattern description :

--                 cr u     u    u  u  u  u  u  u  u  u  u
--                 le s     s    s  s  s  s  s  s  s  s  s
--                 os 1     2    2  2  2  2  5  5  5  2  2
--                 ce .     .    .  .  .  .  .  .  .  .  .
--                 kt s     p    a  r  i  s  r  r  r  i  m
--                    t     c    c  e  n  p  a  a  a  n  e
--                          _    c  g  s     m  m  m  b  m
--                    w          u  _  t        _  _  _  u  o
--                               i  _     0  0  1  s  w
--                               r  a     1  7  0     2
--                                  t

(ciclos de reset)
pat_1           : 00?00000?0000?00?00?00?00?00?00?00?00?00;
pat_2           : 11?00000?0000?00?00?00?07?00?00?00?00?00;
pat_3           : 01?00000?0000?00?00?00?07?00?00?00?00?00;
pat_4           : 11?00000?0000?00?00?00?07?00?00?00?00?00;

-- MOV R1, #10 (2 bytes, 6 ciclos por ser a primeira inst.)
pat_5           : 00?00000?0000?00?00?00?07?00?00?00?00?00;
pat_6           : 10?00000?0000?00?00?00?07?00?00?00?00?00;
pat_7           : 00?00001?0000?00?79?00?07?00?00?00?00?00;
pat_8           : 10?00001?0000?00?79?00?07?00?00?00?00?00;
pat_9           : 00?00010?0001?00?79?00?07?00?00?00?00?00;
pat_10          : 10?00010?0001?00?79?00?07?00?00?00?00?00;

pat_11          : 00?00000?0002?00?79?79?07?00?00?00?00?0a;
pat_12          : 10?00000?0002?00?79?79?07?00?00?00?00?0a;
pat_13          : 00?00001?0002?00?12?79?07?00?00?00?0a?0a;
pat_14          : 10?00001?0002?00?12?79?07?00?00?00?0a?0a;
pat_15          : 00?00010?0003?00?12?79?07?0a?00?00?0a?0a;
pat_16          : 10?00010?0003?00?12?79?07?0a?00?00?0a?0a;

--LCALL 0x32 (3 bytes, quebra o pipeline)
pat_17          : 00?00011?0004?00?12?79?07?0a?00?00?0a?0a;
pat_18          : 10?00011?0004?00?12?79?07?0a?00?00?0a?0a;
pat_19          : 00?00100?0004?00?12?79?07?0a?00?00?0a?0a;
pat_20          : 10?00100?0004?00?12?79?07?0a?00?00?0a?0a;
pat_21          : 00?00101?0005?05?12?79?07?0a?00?00?0a?0a;
pat_22          : 10?00101?0005?05?12?79?07?0a?00?00?0a?0a;
pat_23          : 00?00110?0005?05?12?79?08?0a?00?00?05?0a;
pat_24          : 10?00110?0005?05?12?79?08?0a?00?00?05?0a;
pat_25          : 00?00111?0005?05?12?79?08?0a?00?00?05?0a;
pat_26          : 10?00111?0005?05?12?79?08?0a?00?00?05?0a;
pat_27          : 00?01000?0005?00?12?79?08?0a?00?00?05?0a;
pat_28          : 10?01000?0005?00?12?79?08?0a?00?00?05?0a;
pat_29          : 00?01001?0005?00?12?79?08?0a?00?00?00?0a;
```

```
pat_30          : 10?01001?0005?00?12?79?08?0a?00?00?00?0a;
pat_31          : 00?01010?0005?00?12?79?09?0a?00?00?00?0a;
pat_32          : 10?01010?0005?00?12?79?09?0a?00?00?00?0a;
pat_33          : 00?01011?0032?00?12?79?09?0a?00?00?00?0a;
pat_34          : 10?01011?0032?00?12?79?09?0a?00?00?00?0a;

--RET (1 byte, quebra o pipeline)
pat_35          : 00?00000?0032?00?12?12?09?0a?00?00?00?32;
pat_36          : 10?00000?0032?00?12?12?09?0a?00?00?00?32;
pat_37          : 00?00001?0032?00?22?12?09?0a?00?00?00?32;
pat_38          : 10?00001?0032?00?22?12?09?0a?00?00?00?32;
pat_39          : 00?00010?0033?00?22?12?09?0a?00?00?00?32;
pat_40          : 10?00010?0033?00?22?12?09?0a?00?00?00?32;
pat_41          : 00?00011?0033?00?22?12?09?0a?00?00?00?32;
pat_42          : 10?00011?0033?00?22?12?09?0a?00?00?00?32;
pat_43          : 00?00100?0033?00?22?12?09?0a?00?00?00?32;
pat_44          : 10?00100?0033?00?22?12?09?0a?00?00?00?32;
pat_45          : 00?00101?0033?00?22?12?09?0a?00?00?00?32;
pat_46          : 10?00101?0033?00?22?12?09?0a?00?00?00?32;
pat_47          : 00?00110?0033?00?22?12?08?0a?00?00?00?32;
pat_48          : 10?00110?0033?00?22?12?08?0a?00?00?00?32;
pat_49          : 00?00111?0033?00?22?12?08?0a?00?00?00?32;
pat_50          : 10?00111?0033?00?22?12?08?0a?00?00?00?32;
pat_51          : 00?01000?0005?00?22?12?08?0a?00?00?00?32;
pat_52          : 10?01000?0005?00?22?12?08?0a?00?00?00?32;
pat_53          : 00?01001?0005?00?22?12?08?0a?00?00?00?32;
pat_54          : 10?01001?0005?00?22?12?08?0a?00?00?00?32;
pat_55          : 00?01010?0005?00?22?12?07?0a?00?00?00?32;
pat_56          : 10?01010?0005?00?22?12?07?0a?00?00?00?32;
pat_57          : 00?01011?0005?00?22?12?07?0a?00?00?00?32;
pat_58          : 10?01011?0005?00?22?12?07?0a?00?00?00?32;

--INC R7 (1 byte, 6 ciclos para encher novamente o pipe)
pat_59          : 00?00000?0005?00?22?22?07?0a?00?00?00?00;
pat_60          : 10?00000?0005?00?22?22?07?0a?00?00?00?00;
pat_61          : 00?00001?0005?00?0f?22?07?0a?00?00?00?00;
pat_62          : 10?00001?0005?00?0f?22?07?0a?00?00?00?00;
pat_63          : 00?00010?0006?00?0f?22?07?0a?00?00?00?00;
pat_64          : 10?00010?0006?00?0f?22?07?0a?00?00?00?00;

pat_65          : 00?00000?0006?00?0f?0f?07?0a?00?00?00?1f;
pat_66          : 10?00000?0006?00?0f?0f?07?0a?00?00?00?1f;
pat_67          : 00?00001?0006?00?1f?0f?07?0a?00?00?00?1f;
pat_68          : 10?00001?0006?00?1f?0f?07?0a?00?00?00?1f;
pat_69          : 00?00010?0007?00?1f?0f?07?0a?01?00?01?1f;
pat_70          : 10?00010?0007?00?1f?0f?07?0a?01?00?01?1f;

--DEC R7 (1 byte)
pat_71          : 00?00000?0007?00?1f?1f?07?0a?01?00?01?0f;
pat_72          : 10?00000?0007?00?1f?1f?07?0a?01?00?01?0f;
pat_73          : 00?00001?0007?00?0f?1f?07?0a?01?00?01?0f;
pat_74          : 10?00001?0007?00?0f?1f?07?0a?01?00?01?0f;
pat_75          : 00?00010?0008?00?0f?1f?07?0a?00?00?00?0f;
pat_76          : 10?00010?0008?00?0f?1f?07?0a?00?00?00?0f;

--INC R7 (1byte)
pat_77          : 00?00000?0008?00?0f?0f?07?0a?00?00?00?a7;
pat_78          : 10?00000?0008?00?0f?0f?07?0a?00?00?00?a7;
pat_79          : 00?00001?0008?00?a7?0f?07?0a?00?00?00?a7;
pat_80          : 10?00001?0008?00?a7?0f?07?0a?00?00?00?a7;
pat_81          : 00?00010?0009?00?a7?0f?07?0a?01?00?01?a7;
pat_82          : 10?00010?0009?00?a7?0f?07?0a?01?00?01?a7;

--MOV @R1, R7 (2 bytes)
pat_83          : 00?00000?000a?00?a7?a7?07?0a?01?00?01?07;
pat_84          : 10?00000?000a?00?a7?a7?07?0a?01?00?01?07;
pat_85          : 00?00001?000a?00?00?a7?07?0a?01?00?01?07;
pat_86          : 10?00001?000a?00?00?a7?07?0a?01?00?01?07;
pat_87          : 00?00010?000b?00?00?a7?07?0a?01?01?01?07;
pat_88          : 10?00010?000b?00?00?a7?07?0a?01?01?01?07;

-- término de mov@R1, R7

end;
```

Simulação sem *pipeline*:

```
-- description generated by Pat driver v104
--                       date : Tue Jun 17 15:25:13 1997


--              sequence : teste_sem_pipe

-- input / output list :
in       clock B;
in       reset B;
register us1.st_w (4 downto 0) B;
register us2.pc (15 downto 0) X;
register us2.accu (7 downto 0) X;
register us2.reg_ir (7 downto 0) X;
register us2.sp (7 downto 0) X;
register us5.ram_01 (7 downto 0) X;
register us5.ram_07 (7 downto 0) X;
```

```
register us5.ram_10 (7 downto 0) X;
register us5.ram_17 (7 downto 0) X;
register us2.inbus (7 downto 0) X;

begin

-- Pattern description :

--                cr u     u    u  u  u  u  u  u  u  u
--                le s     s    s  s  s  s  s  s  s  s
--                os 1     2    2  2  2  5  5  5  5  2
--                ce .     .    .  .  .  .  .  .  .  .
--                kt s     p    a  r  s  r  r  r  r  i
--                   t     c    c  e  p  a  a  a  a  n
--                      _       c  g     m  m  m  m  b
--                   w          u  _     _  _  _  _  u
--                                 i     0  0  1  1  s
--                                 r     1  7  0  7

-- Estados de reset
pat_1           : 00?00000?0000?00?00?00?00?00?00?00?00;
pat_2           : 11?00000?0000?00?00?07?00?00?00?00?00;
pat_3           : 01?00000?0000?00?00?07?00?00?00?00?00;
pat_4           : 11?00000?0000?00?00?07?00?00?00?00?00;

-- MOV R1, 10 (12 ciclos, 2 bytes)
pat_5           : 00?00000?0000?00?00?07?00?00?00?00?00;
pat_6           : 10?00000?0000?00?00?07?00?00?00?00?00;
pat_7           : 00?00001?0000?00?79?07?00?00?00?00?00;
pat_8           : 10?00001?0000?00?79?07?00?00?00?00?00;
pat_9           : 00?00010?0001?00?79?07?00?00?00?00?00;
pat_10          : 10?00010?0001?00?79?07?00?00?00?00?00;
pat_11          : 00?00011?0001?00?79?07?00?00?00?00?00;
pat_12          : 10?00011?0001?00?79?07?00?00?00?00?00;
pat_13          : 00?00100?0001?00?79?07?00?00?00?00?0a;
pat_14          : 10?00100?0001?00?79?07?00?00?00?00?0a;
pat_15          : 00?00101?0001?00?79?07?00?00?00?00?0a;
pat_16          : 10?00101?0001?00?79?07?00?00?00?00?0a;
pat_17          : 00?00110?0001?00?79?07?00?00?00?00?0a;
pat_18          : 10?00110?0001?00?79?07?00?00?00?00?0a;
pat_19          : 00?00111?0001?00?79?07?00?00?00?00?0a;
pat_20          : 10?00111?0001?00?79?07?00?00?00?00?0a;
pat_21          : 00?01000?0002?00?79?07?0a?00?00?00?0a;
pat_22          : 10?01000?0002?00?79?07?0a?00?00?00?0a;
pat_23          : 00?01001?0002?00?79?07?0a?00?00?00?0a;
pat_24          : 10?01001?0002?00?79?07?0a?00?00?00?0a;
pat_25          : 00?01010?0002?00?79?07?0a?00?00?00?0a;
pat_26          : 10?01010?0002?00?79?07?0a?00?00?00?0a;
pat_27          : 00?01011?0002?00?79?07?0a?00?00?00?0a;
pat_28          : 10?01011?0002?00?79?07?0a?00?00?00?0a;
pat_29          : 00?01100?0002?00?79?07?0a?00?00?00?0a;
pat_30          : 10?01100?0002?00?79?07?0a?00?00?00?0a;

-- LCALL 0x32 (24 ciclos, 3 bytes)
pat_31          : 00?00000?0002?00?79?07?0a?00?00?00?0a;
pat_32          : 10?00000?0002?00?79?07?0a?00?00?00?0a;
pat_33          : 00?00001?0002?00?12?07?0a?00?00?00?0a;
pat_34          : 10?00001?0002?00?12?07?0a?00?00?00?0a;
pat_35          : 00?00010?0003?00?12?07?0a?00?00?00?0a;
pat_36          : 10?00010?0003?00?12?07?0a?00?00?00?0a;
pat_37          : 00?00011?0003?00?12?07?0a?00?00?00?0a;
pat_38          : 10?00011?0003?00?12?07?0a?00?00?00?0a;
pat_39          : 00?00100?0003?00?12?07?0a?00?00?00?0a;
pat_40          : 10?00100?0003?00?12?07?0a?00?00?00?0a;
pat_41          : 00?00101?0004?00?12?07?0a?00?00?00?0a;
pat_42          : 10?00101?0004?00?12?07?0a?00?00?00?0a;
pat_43          : 00?00110?0004?00?12?07?0a?00?00?00?0a;
pat_44          : 10?00110?0004?00?12?07?0a?00?00?00?0a;
pat_45          : 00?00111?0004?00?12?07?0a?00?00?00?0a;
pat_46          : 10?00111?0004?00?12?07?0a?00?00?00?0a;
pat_47          : 00?01000?0005?00?12?07?0a?00?00?00?0a;
pat_48          : 10?01000?0005?00?12?07?0a?00?00?00?0a;
pat_49          : 00?01001?0005?05?12?07?0a?00?00?00?0a;
pat_50          : 10?01001?0005?05?12?07?0a?00?00?00?0a;
pat_51          : 00?01010?0005?05?12?07?0a?00?00?00?05;
pat_52          : 10?01010?0005?05?12?07?0a?00?00?00?05;
pat_53          : 00?01011?0005?05?12?07?0a?00?00?00?05;
pat_54          : 10?01011?0005?05?12?07?0a?00?00?00?05;
pat_55          : 00?01100?0005?05?12?08?0a?00?00?00?05;
pat_56          : 10?01100?0005?05?12?08?0a?00?00?00?05;
pat_57          : 00?01101?0005?05?12?08?0a?00?00?00?05;
pat_58          : 10?01101?0005?05?12?08?0a?00?00?00?05;
pat_59          : 00?01110?0005?05?12?08?0a?00?00?00?05;
pat_60          : 10?01110?0005?05?12?08?0a?00?00?00?05;
pat_61          : 00?01111?0005?05?12?08?0a?00?00?00?05;
pat_62          : 10?01111?0005?05?12?08?0a?00?00?00?05;
pat_63          : 00?10000?0005?00?12?08?0a?00?00?00?05;
pat_64          : 10?10000?0005?00?12?08?0a?00?00?00?05;
pat_65          : 00?10001?0005?00?12?08?0a?00?00?00?05;
pat_66          : 10?10001?0005?00?12?08?0a?00?00?00?05;
pat_67          : 00?10010?0005?00?12?08?0a?00?00?00?05;
pat_68          : 10?10010?0005?00?12?08?0a?00?00?00?05;
pat_69          : 00?10011?0005?00?12?08?0a?00?00?00?00;
pat_70          : 10?10011?0005?00?12?08?0a?00?00?00?00;
pat_71          : 00?10100?0005?00?12?09?0a?00?00?00?00;
pat_72          : 10?10100?0005?00?12?09?0a?00?00?00?00;
pat_73          : 00?10101?0005?00?12?09?0a?00?00?00?00;
pat_74          : 10?10101?0005?00?12?09?0a?00?00?00?00;
pat_75          : 00?10110?0032?00?12?09?0a?00?00?00?00;
```

```
pat_76          : 10?10110?0032?00?12?09?0a?00?00?00?00;
pat_77          : 00?10111?0032?00?12?09?0a?00?00?00?00;
pat_78          : 10?10111?0032?00?12?09?0a?00?00?00?00;

-- RET (12 ciclos, 1 byte)
pat_79          : 00?00000?0032?00?12?09?0a?00?00?00?00;
pat_80          : 10?00000?0032?00?12?09?0a?00?00?00?00;
pat_81          : 00?00001?0032?00?22?09?0a?00?00?00?00;
pat_82          : 10?00001?0032?00?22?09?0a?00?00?00?00;
pat_83          : 00?00010?0033?00?22?09?0a?00?00?00?00;
pat_84          : 10?00010?0033?00?22?09?0a?00?00?00?00;
pat_85          : 00?00011?0033?00?22?09?0a?00?00?00?00;
pat_86          : 10?00011?0033?00?22?09?0a?00?00?00?00;
pat_87          : 00?00100?0033?00?22?09?0a?00?00?00?00;
pat_88          : 10?00100?0033?00?22?09?0a?00?00?00?00;
pat_89          : 00?00101?0033?00?22?09?0a?00?00?00?00;
pat_90          : 10?00101?0033?00?22?09?0a?00?00?00?00;
pat_91          : 00?00110?0033?00?22?08?0a?00?00?00?00;
pat_92          : 10?00110?0033?00?22?08?0a?00?00?00?00;
pat_93          : 00?00111?0033?00?22?08?0a?00?00?00?00;
pat_94          : 10?00111?0033?00?22?08?0a?00?00?00?00;
pat_95          : 00?01000?0005?00?22?08?0a?00?00?00?00;
pat_96          : 10?01000?0005?00?22?08?0a?00?00?00?00;
pat_97          : 00?01001?0005?00?22?08?0a?00?00?00?00;
pat_98          : 10?01001?0005?00?22?08?0a?00?00?00?00;
pat_99          : 00?01010?0005?00?22?07?0a?00?00?00?00;
pat_100         : 10?01010?0005?00?22?07?0a?00?00?00?00;
pat_101         : 00?01011?0005?00?22?07?0a?00?00?00?00;
pat_102         : 10?01011?0005?00?22?07?0a?00?00?00?00;
pat_103         : 00?01100?0005?00?22?07?0a?00?00?00?00;
pat_104         : 10?01100?0005?00?22?07?0a?00?00?00?00;

-- INC R7 (12 ciclos, 1 byte)
pat_105         : 00?00000?0005?00?22?07?0a?00?00?00?00;
pat_106         : 10?00000?0005?00?22?07?0a?00?00?00?00;
pat_107         : 00?00001?0005?00?0f?07?0a?00?00?00?00;
pat_108         : 10?00001?0005?00?0f?07?0a?00?00?00?00;
pat_109         : 00?00010?0006?00?0f?07?0a?00?00?00?00;
pat_110         : 10?00010?0006?00?0f?07?0a?00?00?00?00;
pat_111         : 00?00011?0006?00?0f?07?0a?00?00?00?00;
pat_112         : 10?00011?0006?00?0f?07?0a?00?00?00?00;
pat_113         : 00?00100?0006?00?0f?07?0a?00?00?00?00;
pat_114         : 10?00100?0006?00?0f?07?0a?00?00?00?00;
pat_115         : 00?00101?0006?00?0f?07?0a?00?00?00?00;
pat_116         : 10?00101?0006?00?0f?07?0a?00?00?00?00;
pat_117         : 00?00110?0006?00?0f?07?0a?00?00?00?00;
pat_118         : 10?00110?0006?00?0f?07?0a?00?00?00?00;
pat_119         : 00?00111?0006?00?0f?07?0a?00?00?00?00;
pat_120         : 10?00111?0006?00?0f?07?0a?00?00?00?00;
pat_121         : 00?01000?0006?00?0f?07?0a?00?00?00?00;
pat_122         : 10?01000?0006?00?0f?07?0a?00?00?00?00;
pat_123         : 00?01001?0006?00?0f?07?0a?00?00?00?00;
pat_124         : 10?01001?0006?00?0f?07?0a?00?00?00?00;
pat_125         : 00?01010?0006?00?0f?07?0a?00?00?00?01;
pat_126         : 10?01010?0006?00?0f?07?0a?00?00?00?01;
pat_127         : 00?01011?0006?00?0f?07?0a?00?00?00?01;
pat_128         : 10?01011?0006?00?0f?07?0a?00?00?00?01;
pat_129         : 00?01100?0006?00?0f?07?0a?01?00?00?01;
pat_130         : 10?01100?0006?00?0f?07?0a?01?00?00?01;

-- DEC R7 (12 ciclos, 1 byte)
pat_131         : 00?00000?0006?00?0f?07?0a?01?00?00?01;
pat_132         : 10?00000?0006?00?0f?07?0a?01?00?00?01;
pat_133         : 00?00001?0006?00?1f?07?0a?01?00?00?01;
pat_134         : 10?00001?0006?00?1f?07?0a?01?00?00?01;
pat_135         : 00?00010?0007?00?1f?07?0a?01?00?00?01;
pat_136         : 10?00010?0007?00?1f?07?0a?01?00?00?01;
pat_137         : 00?00011?0007?00?1f?07?0a?01?00?00?01;
pat_138         : 10?00011?0007?00?1f?07?0a?01?00?00?01;
pat_139         : 00?00100?0007?00?1f?07?0a?01?00?00?01;
pat_140         : 10?00100?0007?00?1f?07?0a?01?00?00?01;
pat_141         : 00?00101?0007?00?1f?07?0a?01?00?00?01;
pat_142         : 10?00101?0007?00?1f?07?0a?01?00?00?01;
pat_143         : 00?00110?0007?00?1f?07?0a?01?00?00?01;
pat_144         : 10?00110?0007?00?1f?07?0a?01?00?00?01;
pat_145         : 00?00111?0007?00?1f?07?0a?01?00?00?01;
pat_146         : 10?00111?0007?00?1f?07?0a?01?00?00?01;
pat_147         : 00?01000?0007?00?1f?07?0a?01?00?00?01;
pat_148         : 10?01000?0007?00?1f?07?0a?01?00?00?01;
pat_149         : 00?01001?0007?00?1f?07?0a?01?00?00?01;
pat_150         : 10?01001?0007?00?1f?07?0a?01?00?00?01;
pat_151         : 00?01010?0007?00?1f?07?0a?01?00?00?00;
pat_152         : 10?01010?0007?00?1f?07?0a?01?00?00?00;
pat_153         : 00?01011?0007?00?1f?07?0a?00?00?00?00;
pat_154         : 10?01011?0007?00?1f?07?0a?00?00?00?00;
pat_155         : 00?01100?0007?00?1f?07?0a?00?00?00?00;
pat_156         : 10?01100?0007?00?1f?07?0a?00?00?00?00;

-- INC R7 (12 ciclos, 1 byte)
pat_157         : 00?00000?0007?00?1f?07?0a?00?00?00?00;
pat_158         : 10?00000?0007?00?1f?07?0a?00?00?00?00;
pat_159         : 00?00001?0007?00?0f?07?0a?00?00?00?00;
pat_160         : 10?00001?0007?00?0f?07?0a?00?00?00?00;
pat_161         : 00?00010?0008?00?0f?07?0a?00?00?00?00;
pat_162         : 10?00010?0008?00?0f?07?0a?00?00?00?00;
pat_163         : 00?00011?0008?00?0f?07?0a?00?00?00?00;
pat_164         : 10?00011?0008?00?0f?07?0a?00?00?00?00;
pat_165         : 00?00100?0008?00?0f?07?0a?00?00?00?00;
pat_166         : 10?00100?0008?00?0f?07?0a?00?00?00?00;
```

```
pat_167         : 00?00101?0008?00?0f?07?0a?00?00?00?00;
pat_168         : 10?00101?0008?00?0f?07?0a?00?00?00?00;
pat_169         : 00?00110?0008?00?0f?07?0a?00?00?00?00;
pat_170         : 10?00110?0008?00?0f?07?0a?00?00?00?00;
pat_171         : 00?00111?0008?00?0f?07?0a?00?00?00?00;
pat_172         : 10?00111?0008?00?0f?07?0a?00?00?00?00;
pat_173         : 00?01000?0008?00?0f?07?0a?00?00?00?00;
pat_174         : 10?01000?0008?00?0f?07?0a?00?00?00?00;
pat_175         : 00?01001?0008?00?0f?07?0a?00?00?00?00;
pat_176         : 10?01001?0008?00?0f?07?0a?00?00?00?00;
pat_177         : 00?01010?0008?00?0f?07?0a?00?00?00?01;
pat_178         : 10?01010?0008?00?0f?07?0a?00?00?00?01;
pat_179         : 00?01011?0008?00?0f?07?0a?00?00?00?01;
pat_180         : 10?01011?0008?00?0f?07?0a?00?00?00?01;
pat_181         : 00?01100?0008?00?0f?07?0a?01?00?00?01;
pat_182         : 10?01100?0008?00?0f?07?0a?01?00?00?01;

-- MOV @R1, R7 (12 ciclos, 2 bytes)
pat_183         : 00?00000?0008?00?0f?07?0a?01?00?00?01;
pat_184         : 10?00000?0008?00?0f?07?0a?01?00?00?01;
pat_185         : 00?00001?0008?00?a7?07?0a?01?00?00?01;
pat_186         : 10?00001?0008?00?a7?07?0a?01?00?00?01;
pat_187         : 00?00010?0009?00?a7?07?0a?01?00?00?01;
pat_188         : 10?00010?0009?00?a7?07?0a?01?00?00?01;
pat_189         : 00?00011?0009?00?a7?07?0a?01?00?00?01;
pat_190         : 10?00011?0009?00?a7?07?0a?01?00?00?01;
pat_190         : 10?00011?0009?00?a7?07?0a?01?00?00?01;
pat_190         : 10?00011?0009?00?a7?07?0a?01?00?00?01;
pat_191         : 00?00100?0009?00?a7?07?0a?01?00?00?01;
pat_192         : 10?00100?0009?00?a7?07?0a?01?00?00?01;
pat_193         : 00?00101?0009?00?a7?07?0a?01?00?00?01;
pat_194         : 10?00101?0009?00?a7?07?0a?01?00?00?01;
pat_195         : 00?00110?0009?00?a7?07?0a?01?00?00?01;
pat_196         : 10?00110?0009?00?a7?07?0a?01?00?00?01;
pat_197         : 00?00111?0009?00?a7?07?0a?01?00?00?01;
pat_198         : 10?00111?0009?00?a7?07?0a?01?00?00?01;
pat_199         : 00?01000?000a?00?a7?07?0a?01?00?00?01;
pat_200         : 10?01000?000a?00?a7?07?0a?01?00?00?01;
pat_201         : 00?01001?000a?00?a7?07?0a?01?00?00?01;
pat_202         : 10?01001?000a?00?a7?07?0a?01?00?00?01;
pat_203         : 00?01010?000a?00?a7?07?0a?01?01?00?01;
pat_204         : 10?01010?000a?00?a7?07?0a?01?01?00?01;
pat_205         : 00?01011?000a?00?a7?07?0a?01?01?00?01;
pat_206         : 10?01011?000a?00?a7?07?0a?01?01?00?01;
pat_207         : 00?01100?000a?00?a7?07?0a?01?01?00?01;
pat_208         : 10?01100?000a?00?a7?07?0a?01?01?00?01;

end;
```

Anexo 3 Código em linguagem montadora da aplicação *decoder* gerado pelos compiladores CCC51 - III e Keil C Compiler

Código fonte em C da aplicação *decoder*.

```
/* Decoder:
  transforms serial stream of data bits, and perform error
  correction to retrieve original 8 bits data */

unsigned char P0   = 0x80;
unsigned char P1   = 0x90;
unsigned char P2   = 0xA0;
unsigned char P3   = 0xB0;

int inport(unsigned char porta)
{
  unsigned char *in, dado;
  in = &porta;
  dado = *in; /* coloca o conteudo da porta em dado */
  return (dado);
}

void outport(int dado, unsigned char porta)
{
  unsigned char *out;
  out = &porta;
      *out = dado; /* escreve um dado na porta de saida */
}

void xor4 (int a, int b, int c, int d, int result)
{
  int xor4_i = 0;
  xor4_i = xor4_i ^ a;
  xor4_i = xor4_i ^ b;
  xor4_i = xor4_i ^ c;
  xor4_i = xor4_i ^ d;
  result = xor4_i;
}

void main(void)
{
  int i, input_data, row_parity, column_parity, global_parity, error, top_bit, aux;
  int decoder_in, data_ready, strobe, data_out, err, out_ready;

  /* wait for incoming data */
  while (data_ready == 0) data_ready = inport(P1);
  out_ready = 0;
  outport (out_ready,P2); /* signal start of process */

  /* sample input stream */
  i = 0;
  while (i < 16)
  { /* wait on strobe until = 1 */
    while (strobe == 0) strobe = inport(P3);
    decoder_in = inport(P3);
    input_data += decoder_in;
    input_data = input_data << 1;
    i++;
  }

  /* compute parity check on input data */
  top_bit = (input_data & 0x8000) >> 15;
  global_parity = top_bit;

  /* i = 0 */
    xor4 ((input_data & 0x1),((input_data & 0x2) >> 1),((input_data & 0x4) >> 2),
  ((input_data & 0x200) >> 9),aux);
  row_parity = (row_parity & 0xfffe) | aux;

  xor4 ((input_data & 0x1),((input_data & 0x8) >> 3),((input_data & 0x40) >> 6),((input_data
  & 0x1000) >> 12),aux);
  column_parity = (column_parity & 0xfffe) | aux;
  xor4 (global_parity,(input_data & 0x1),((input_data & 0x2) >> 1),((input_data        &
0x4) >> 2),global_parity);
  /* i = 1 */
  xor4 (((input_data & 0x8) >> 3),((input_data & 0x10) >> 4),((input_data & 0x20) >> 5),
((input_data & 0x400) >> 10),aux);
  row_parity = (row_parity & 0xfffd) | (aux << 1);

 xor4 (((input_data & 0x2) >> 1),((input_data & 0x10) >> 4),((input_data & 0x80) >> 7),
  ((input_data & 0x2000) >> 13),aux);
  column_parity = (column_parity & 0xfffd) | (aux << 1);
  xor4 (global_parity,((input_data & 0x8) >> 3),((input_data & 0x10) >> 4),((input_data &
0x20) >> 5),global_parity);
  /* i = 2 */
  xor4 (((input_data & 0x40) >> 6),((input_data & 0x80) >> 7),((input_data & 0x100) >> 8),
((input_data & 0x800) >> 11),aux);
  row_parity = (row_parity & 0xfffb) | (aux << 2);
  xor4 (((input_data & 0x4) >> 2),((input_data & 0x20) >> 5),((input_data & 0x100) >> 8),
((input_data & 0x4000) >> 14),aux);
  column_parity = (column_parity & 0xfffb) | (aux << 2);
```

```
  xor4 (global_parity,((input_data & 0x40) >> 6),((input_data & 0x80) >> 7),((input_data &
0x100) >> 8),global_parity);

  /* Error correction */
  if (global_parity == 0)
  { if ((row_parity & 1) == 0 && (row_parity & 2) == 0 &&
    (column_parity & 1) == 0 && (column_parity & 2) == 0)
  error = 0;    /* no error */
    else error = 3; /* multiple errors, no correction possible */
  }
  else
  { /* single error */
    error = 2;
    if ((row_parity & 1) != 0)
    { if ((column_parity & 1) != 0) input_data ^= 0x1;
  else
    if ((column_parity & 2) != 0) input_data ^= 0x2;
    else
      if ((column_parity & 4) != 0) input_data ^= 0x4;
    }
    else
  if ((row_parity & 2) != 0)
  { if ((column_parity & 1) != 0) input_data ^= 0x8;
    else
      if ((column_parity & 2) != 0) input_data ^= 0x10;
      else
           if ((column_parity & 4) != 0) input_data ^= 0x20;
  }
  else
    if ((row_parity & 4) != 0)
    { if ((column_parity & 1) != 0) input_data ^= 0x40;
      else
           if ((column_parity & 2) != 0) input_data ^= 0x80;
    }
  }

  /* write outputs in parallel */
  /* then pulse out_ready */
  data_out = input_data;
  err = error;
  out_ready = 1;

  outport (data_out,P3);
  outport (err,P2);
  outport (out_ready,P1);

  out_ready = 0;
  outport (out_ready,P0);

}
```

Arquivo em linguagem montadora gerado pelo compilador CCC51 - III:

```
;**************************************
;* Arquivo DECODER.ASM
;*
;* CCC51: 17:38  14/05/1997
;**************************************


; Declaracao de constantes default

_R0_     EQU    00h   ; ender. do reg. R0
_R1_     EQU    01h   ; ender. do reg. R1
_R2_     EQU    02h   ; ender. do reg. R2
_R3_     EQU    03h   ; ender. do reg. R3
_R4_     EQU    04h   ; ender. do reg. R4
_R5_     EQU    05h   ; ender. do reg. R5
_R6_     EQU    06h   ; ender. do reg. R6
_R7_     EQU    07h   ; ender. do reg. R7
_R17_    EQU    11h   ; ender. de mem. 17
_R18_    EQU    12h   ; ender. de mem. 18
_R19_    EQU    13h   ; ender. de mem. 19
_R20_    EQU    14h   ; ender. de mem. 20


; Declaracao das funcoes publicas

PUBLIC  _inport
PUBLIC  _outport
PUBLIC  _xor4
PUBLIC  _main


; Declaracao das variaveis estaticas

              RSEG DADOS

_P0             DB  1
_P1             DB  1
_P2             DB  1
_P3             DB  1


; Inicio da area de codigo

              RSEG CODIGO

_inport:
           MOV A,#_porta
           MOV _R19_,A
           MOV A,_R19_
           MOV _in,A
           MOV A,_in
           MOV _R19_,A
           MOV A, _R20_
           MOV _R0_, A
       MOV A,@R0
       MOV _dado,A
       RET
       RET
_outport:
       MOV A,#_porta
       MOV _R19_,A
       MOV A,_R19_
       MOV _out,A
       MOV A,_out
       MOV _R19_,A
       MOV A,_dado
           MOV A, _R20_
           MOV _R0_, A
           MOV @R0,A
           RET
_xor4:
           MOV A,_xor4_i
           XRL A,_a
           MOV _R19_,A
       MOV A,_R19_
       MOV _xor4_i,A
       MOV A,_xor4_i
       XRL A,_b
       MOV _R19_,A
       MOV A,_R19_
       MOV _xor4_i,A
       MOV A,_xor4_i
       XRL A,_c
       MOV _R19_,A
       MOV A,_R19_
       MOV _xor4_i,A
       MOV A,_xor4_i
           XRL A,_d
       MOV _R19_,A
       MOV A,_R19_
       MOV _xor4_i,A
       MOV A,_xor4_i
       MOV _result,A
       RET
_main:
L0001:
```

```
       MOV A,_data_ready
       CLR C
       SUBB A,#00
       JZ G0000
       ORL A,#0FFh
G0000: CPL A
       MOV _R19_,A
       MOV A,_R19_
       JNZ L0002
       JMP L0003
L0002:
           PUSH _P1
       LCALL _inport
           MOV A,#1
G0025: SUBB A,#01
       POP _R50_
       JNZ G0001
       MOV A,_P1
       MOV _data_ready,A
       JMP L0001
L0003:
L0000:
       MOV A,#00
       MOV _out_ready,A
       PUSH _out_ready
       PUSH _P2
       LCALL _outport
           MOV A,#2
G0025: SUBB A,#01
       POP _R50_
       JNZ G0002
           MOV A,#00
       MOV _i,A
L0005:
       MOV A,_i
       CLR C
       SUBB A,#010
       CLR A
       MOV ACC.0,C
       MOV _R19_,A
       MOV A,_R19_
       JNZ L0006
       JMP L0007
L0006:
L0009:
       MOV A,_strobe
       CLR C
       SUBB A,#00
       JZ G0003
       ORL A,#0FFh
G0003: CPL A
           MOV _R19_,A
       MOV A,_R19_
       JNZ L0010
       JMP L0011
L0010:
       PUSH _P3
       LCALL _inport
           MOV A,#4
G0025: SUBB A,#01
       POP _R50_
       JNZ G0004
       MOV A,_P3
       MOV _strobe,A
       JMP L0009
L0011:
L0008:
       PUSH _P3
       LCALL _inport
           MOV A,#5
G0025: SUBB A,#01
            POP _R50_
       JNZ G0005
       MOV A,_P3
       MOV _decoder_in,A
       MOV A,_input_data
       ADD A,_decoder_in
       MOV _input_data,A
       MOV A,#01
       MOV _R19_,A
       MOV A,_input_data
G0006: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0006
       MOV A,_R19_
       MOV _R18_,A
       MOV A,_R18_
       MOV _input_data,A
           ADD _i,#01
       JMP L0005
L0007:
L0004:
       MOV A,_input_data
       ANL A,#08000
       MOV _R19_,A
       MOV A,#0f
```

```
       MOV _R19_,A
       MOV A,_R19_
G0007: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0007
       MOV A,_R19_
       MOV _R18_,A
       MOV A,_R18_
       MOV _top_bit,A
           MOV A,_top_bit
       MOV _global_parity,A
       MOV A,_input_data
       ANL A,#01
       MOV _R19_,A
       PUSH _R19_
       MOV A,_input_data
       ANL A,#02
       MOV _R19_,A
       MOV A,#01
       MOV _R19_,A
       MOV A,_R19_
G0008: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0008
       MOV A,_R19_
       MOV _R18_,A
           PUSH _R18_
       MOV A,_input_data
       ANL A,#04
       MOV _R19_,A
       MOV A,#02
       MOV _R19_,A
       MOV A,_R19_
G0009: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0009
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       MOV A,_input_data
       ANL A,#0200
       MOV _R19_,A
       MOV A,#09
           MOV _R19_,A
       MOV A,_R19_
G0010: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0010
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       PUSH _aux
       LCALL _xor4
           MOV A,#11
G0025: SUBB A,#01
       POP _R50_
       JNZ G0011
       MOV A,_row_parity
       ANL A,#0fffe
       MOV _R19_,A
           MOV A,_R19_
       ORL A,_aux
       MOV _R19_,A
       MOV A,_R19_
       MOV _row_parity,A
       MOV A,_input_data
       ANL A,#01
       MOV _R19_,A
       PUSH _R19_
       MOV A,_input_data
       ANL A,#08
       MOV _R19_,A
       MOV A,#03
       MOV _R19_,A
       MOV A,_R19_
G0012: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
   JNZ G0012
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       MOV A,_input_data
       ANL A,#040
       MOV _R19_,A
```

```
       MOV A,#06
       MOV _R19_,A
       MOV A,_R19_
G0013: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0013
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       MOV A,_input_data
           ANL A,#01000
       MOV _R19_,A
       MOV A,#0c
       MOV _R19_,A
       MOV A,_R19_
G0014: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0014
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       PUSH _aux
       LCALL _xor4
           MOV A,#15
G0025: SUBB A,#01
       POP _R50_
       JNZ G0015
           MOV A,_column_parity
       ANL A,#0fffe
       MOV _R19_,A
       MOV A,_R19_
       ORL A,_aux
       MOV _R19_,A
       MOV A,_R19_
       MOV _column_parity,A
       PUSH _global_parity
       MOV A,_input_data
       ANL A,#01
       MOV _R19_,A
       PUSH _R19_
       MOV A,_input_data
       ANL A,#02
       MOV _R19_,A
       MOV A,#01
       MOV _R19_,A
       MOV A,_R19_
G0016: CLR C
           RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0016
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       MOV A,_input_data
       ANL A,#04
       MOV _R19_,A
       MOV A,#02
       MOV _R19_,A
       MOV A,_R19_
G0017: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0017
           MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       PUSH _global_parity
       LCALL _xor4
           MOV A,#18
G0025: SUBB A,#01
       POP _R50_
       JNZ G0018
       MOV A,_input_data
       ANL A,#08
       MOV _R19_,A
       MOV A,#03
       MOV _R19_,A
       MOV A,_R19_
G0019: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
           JNZ G0019
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       MOV A,_input_data
       ANL A,#010
```

```
       MOV _R19_,A
       MOV A,#04
       MOV _R19_,A
       MOV A,_R19_
G0020: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0020
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       MOV A,_input_data
           ANL A,#020
       MOV _R19_,A
       MOV A,#05
       MOV _R19_,A
       MOV A,_R19_
G0021: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0021
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       MOV A,_input_data
       ANL A,#0400
       MOV _R19_,A
       MOV A,#0a
       MOV _R19_,A
       MOV A,_R19_
G0022: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0022
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       PUSH _aux
       LCALL _xor4
           MOV A,#23
G0025: SUBB A,#01
       POP _R50_
       JNZ G0023
       MOV A,_row_parity
       ANL A,#0fffd
       MOV _R19_,A
       MOV A,#01
       MOV _R18_,A
           MOV A,_aux
G0024: CLR C
       RLC A
       MOV _R18_,A
       MOV A,_R18_
       SUBB A,#01
       JNZ G0024
       MOV A,_R18_
       MOV _R17_,A
       MOV A,_R19_
       ORL A,_R17_
       MOV _R19_,A
       MOV A,_R19_
       MOV _row_parity,A
       MOV A,_input_data
       ANL A,#02
       MOV _R19_,A
       MOV A,#01
       MOV _R19_,A
       MOV A,_R19_
G0025: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0025
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       MOV A,_input_data
       ANL A,#010
       MOV _R19_,A
       MOV A,#04
       MOV _R19_,A
       MOV A,_R19_
G0026: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
           JNZ G0026
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       MOV A,_input_data
```

```
       ANL A,#080
       MOV _R19_,A
       MOV A,#07
       MOV _R19_,A
       MOV A,_R19_
G0027: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0027
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       MOV A,_input_data
           ANL A,#02000
       MOV _R19_,A
       MOV A,#0d
       MOV _R19_,A
       MOV A,_R19_
G0028: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0028
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       PUSH _aux
       LCALL _xor4
           MOV A,#29
G0025: SUBB A,#01
       POP _R50_
       JNZ G0029
           MOV A,_column_parity
       ANL A,#0fffd
       MOV _R19_,A
       MOV A,#01
       MOV _R18_,A
       MOV A,_aux
G0030: CLR C
       RLC A
       MOV _R18_,A
       MOV A,_R18_
       SUBB A,#01
       JNZ G0030
       MOV A,_R18_
       MOV _R17_,A
       MOV A,_R19_
       ORL A,_R17_
       MOV _R19_,A
       MOV A,_R19_
       MOV _column_parity,A
       PUSH _global_parity
           MOV A,_input_data
       ANL A,#08
       MOV _R19_,A
       MOV A,#03
       MOV _R19_,A
       MOV A,_R19_
G0031: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0031
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       MOV A,_input_data
       ANL A,#010
       MOV _R19_,A
       MOV A,#04
       MOV _R19_,A
           MOV A,_R19_
G0032: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0032
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       MOV A,_input_data
       ANL A,#020
       MOV _R19_,A
       MOV A,#05
       MOV _R19_,A
       MOV A,_R19_
G0033: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
           SUBB A,#01
       JNZ G0033
       MOV A,_R19_
       MOV _R18_,A
```

```
       PUSH _R18_
       PUSH _global_parity
       LCALL _xor4
           MOV A,#34
G0025: SUBB A,#01
       POP _R50_
       JNZ G0034
       MOV A,_input_data
       ANL A,#040
       MOV _R19_,A
       MOV A,#06
       MOV _R19_,A
       MOV A,_R19_
G0035: CLR C
       RLC A
       MOV _R19_,A
           MOV A,_R19_
       SUBB A,#01
       JNZ G0035
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       MOV A,_input_data
       ANL A,#080
       MOV _R19_,A
       MOV A,#07
       MOV _R19_,A
       MOV A,_R19_
G0036: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0036
       MOV A,_R19_
       MOV _R18_,A
           PUSH _R18_
       MOV A,_input_data
       ANL A,#0100
       MOV _R19_,A
       MOV A,#08
       MOV _R19_,A
       MOV A,_R19_
G0037: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0037
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       MOV A,_input_data
       ANL A,#0800
       MOV _R19_,A
       MOV A,#0b
           MOV _R19_,A
       MOV A,_R19_
G0038: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0038
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       PUSH _aux
       LCALL _xor4
           MOV A,#39
G0025: SUBB A,#01
       POP _R50_
       JNZ G0039
       MOV A,_row_parity
       ANL A,#0fffb
       MOV _R19_,A
           MOV A,#02
       MOV _R18_,A
       MOV A,_aux
G0040: CLR C
       RLC A
       MOV _R18_,A
       MOV A,_R18_
       SUBB A,#01
       JNZ G0040
       MOV A,_R18_
       MOV _R17_,A
       MOV A,_R19_
       ORL A,_R17_
       MOV _R19_,A
       MOV A,_R19_
       MOV _row_parity,A
       MOV A,_input_data
       ANL A,#04
       MOV _R19_,A
       MOV A,#02
           MOV _R19_,A
       MOV A,_R19_
G0041: CLR C
```

```
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0041
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       MOV A,_input_data
       ANL A,#020
       MOV _R19_,A
       MOV A,#05
       MOV _R19_,A
       MOV A,_R19_
G0042: CLR C
       RLC A
       MOV _R19_,A
           MOV A,_R19_
       SUBB A,#01
       JNZ G0042
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       MOV A,_input_data
       ANL A,#0100
       MOV _R19_,A
       MOV A,#08
       MOV _R19_,A
       MOV A,_R19_
G0043: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0043
       MOV A,_R19_
       MOV _R18_,A
           PUSH _R18_
       MOV A,_input_data
       ANL A,#04000
       MOV _R19_,A
       MOV A,#0e
       MOV _R19_,A
       MOV A,_R19_
G0044: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0044
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       PUSH _aux
       LCALL _xor4
           MOV A,#45
G0025: SUBB A,#01
           POP _R50_
       JNZ G0045
       MOV A,_column_parity
       ANL A,#0fffb
       MOV _R19_,A
       MOV A,#02
       MOV _R18_,A
       MOV A,_aux
G0046: CLR C
       RLC A
       MOV _R18_,A
       MOV A,_R18_
       SUBB A,#01
       JNZ G0046
       MOV A,_R18_
       MOV _R17_,A
       MOV A,_R19_
       ORL A,_R17_
       MOV _R19_,A
       MOV A,_R19_
   MOV _column_parity,A
       PUSH _global_parity
       MOV A,_input_data
       ANL A,#040
       MOV _R19_,A
       MOV A,#06
       MOV _R19_,A
       MOV A,_R19_
G0047: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0047
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       MOV A,_input_data
       ANL A,#080
       MOV _R19_,A
           MOV A,#07
       MOV _R19_,A
```

```
       MOV A,_R19_
G0048: CLR C
       RLC A
       MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0048
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       MOV A,_input_data
       ANL A,#0100
       MOV _R19_,A
       MOV A,#08
       MOV _R19_,A
       MOV A,_R19_
G0049: CLR C
       RLC A
           MOV _R19_,A
       MOV A,_R19_
       SUBB A,#01
       JNZ G0049
       MOV A,_R19_
       MOV _R18_,A
       PUSH _R18_
       PUSH _global_parity
       LCALL _xor4
           MOV A,#50
G0025: SUBB A,#01
       POP _R50_
       JNZ G0050
       MOV A,_global_parity
       CLR C
       SUBB A,#00
       JZ G0051
       ORL A,#0FFh
G0051: CPL A
       MOV _R19_,A
           MOV A,_R19_
       JNZ L0012
       JMP L0013
L0012:
       MOV A,_row_parity
       ANL A,#01
       MOV _R19_,A
       MOV A,_R19_
       CLR C
       SUBB A,#00
       JZ G0052
       ORL A,#0FFh
G0052: CPL A
       MOV _R19_,A
       MOV A,_row_parity
       ANL A,#02
       MOV _R18_,A
       MOV A,_R18_
       CLR C
       SUBB A,#00
           JZ G0053
       ORL A,#0FFh
G0053: CPL A
       MOV _R18_,A
       MOV A,_R19_
       ANL A,_R18_
       MOV _R19_,A
       MOV A,_column_parity
       ANL A,#01
       MOV _R18_,A
       MOV A,_R18_
       CLR C
       SUBB A,#00
       JZ G0054
       ORL A,#0FFh
G0054: CPL A
       MOV _R18_,A
       MOV A,_R19_
       ANL A,_R18_
       MOV _R19_,A
   MOV A,_column_parity
       ANL A,#02
       MOV _R18_,A
       MOV A,_R18_
       CLR C
       SUBB A,#00
       JZ G0055
       ORL A,#0FFh
G0055: CPL A
       MOV _R18_,A
       MOV A,_R19_
       ANL A,_R18_
       MOV _R19_,A
       MOV A,_R19_
       JNZ L0014
       JMP L0015
L0014:
       MOV A,#00
       MOV _error,A
       JMP L0016
L0015:
```

```
       MOV A,#03
       MOV _error,A
L0016:
       JMP L0017
L0013:
       MOV A,#02
       MOV _error,A
       MOV A,_row_parity
       ANL A,#01
       MOV _R19_,A
       MOV A,_R19_
       CLR C
       SUBB A,#00
       MOV _R19_,A
       MOV A,_R19_
       JNZ L0018
       JMP L0019
L0018:
       MOV A,_column_parity
           ANL A,#01
       MOV _R19_,A
       MOV A,_R19_
       CLR C
       SUBB A,#00
       MOV _R19_,A
       MOV A,_R19_
       JNZ L0020
       JMP L0021
L0020:
       MOV A,_input_data
       XRL A,#01
       MOV _input_data,A
       JMP L0022
L0021:
       MOV A,_column_parity
       ANL A,#02
       MOV _R19_,A
       MOV A,_R19_
       CLR C
   SUBB A,#00
       MOV _R19_,A
       MOV A,_R19_
       JNZ L0023
       JMP L0024
L0023:
       MOV A,_input_data
       XRL A,#02
       MOV _input_data,A
       JMP L0025
L0024:
       MOV A,_column_parity
       ANL A,#04
       MOV _R19_,A
       MOV A,_R19_
       CLR C
       SUBB A,#00
       MOV _R19_,A
       MOV A,_R19_
       JNZ L0026
           JMP L0027
L0026:
       MOV A,_input_data
       XRL A,#04
       MOV _input_data,A
L0027:
L0025:
L0022:
       JMP L0028
L0019:
       MOV A,_row_parity
       ANL A,#02
       MOV _R19_,A
       MOV A,_R19_
       CLR C
       SUBB A,#00
       MOV _R19_,A
       MOV A,_R19_
       JNZ L0029
       JMP L0030
L0029:
       MOV A,_column_parity
       ANL A,#01
       MOV _R19_,A
       MOV A,_R19_
       CLR C
       SUBB A,#00
       MOV _R19_,A
       MOV A,_R19_
       JNZ L0031
       JMP L0032
L0031:
       MOV A,_input_data
       XRL A,#08
       MOV _input_data,A
       JMP L0033
L0032:
       MOV A,_column_parity
       ANL A,#02
       MOV _R19_,A
```

```
            MOV A,_R19_
       CLR C
       SUBB A,#00
       MOV _R19_,A
       MOV A,_R19_
       JNZ L0034
       JMP L0035
L0034:
       MOV A,_input_data
       XRL A,#010
       MOV _input_data,A
       JMP L0036
L0035:
       MOV A,_column_parity
       ANL A,#04
       MOV _R19_,A
       MOV A,_R19_
       CLR C
       SUBB A,#00
       MOV _R19_,A
   MOV A,_R19_
       JNZ L0037
       JMP L0038
L0037:
       MOV A,_input_data
       XRL A,#020
       MOV _input_data,A
L0038:
L0036:
L0033:
       JMP L0039
L0030:
       MOV A,_row_parity
       ANL A,#04
       MOV _R19_,A
       MOV A,_R19_
       CLR C
       SUBB A,#00
       MOV _R19_,A
       MOV A,_R19_
            JNZ L0040
       JMP L0041
L0040:
       MOV A,_column_parity
       ANL A,#01
       MOV _R19_,A
       MOV A,_R19_
       CLR C
       SUBB A,#00
       MOV _R19_,A
       MOV A,_R19_
       JNZ L0042
       JMP L0043
L0042:
       MOV A,_input_data
       XRL A,#040
       MOV _input_data,A
       JMP L0044
L0043:
       MOV A,_column_parity
            ANL A,#02
       MOV _R19_,A
       MOV A,_R19_
       CLR C
       SUBB A,#00
       MOV _R19_,A
       MOV A,_R19_
       JNZ L0045
       JMP L0046
L0045:
       MOV A,_input_data
       XRL A,#080
       MOV _input_data,A
L0046:
L0044:
L0041:
L0039:
L0028:
L0017:
       MOV A,_input_data
   MOV _data_out,A
       MOV A,_error
       MOV _err,A
       MOV A,#01
       MOV _out_ready,A
       PUSH _data_out
       PUSH _P3
       LCALL _outport
            MOV A,#56
G0025: SUBB A,#01
       POP _R50_
       JNZ G0056
       PUSH _err
       PUSH _P2
       LCALL _outport
            MOV A,#57
G0025: SUBB A,#01
            POP _R50_
            JNZ G0057
```

```
            PUSH _out_ready
            PUSH _P1
            LCALL _outport
            MOV A,#58
G0025: SUBB A,#01
            POP _R50_
            JNZ G0058
            MOV A,#00
            MOV _out_ready,A
            PUSH _out_ready
            PUSH _P0
            LCALL _outport
            MOV A,#59
G0025: SUBB A,#01
            POP _R50_
            JNZ G0059
            RET
            END
```

Arquivo em linguagem montadora gerado pelo compilador Keil C Compiler.

```
C51 COMPILER V5.02,  SN-  DECOD3
05/13/97  17:22:02  PAGE 1


DOS C51 COMPILER V5.02, COMPILATION OF MODULE DECODER
OBJECT MODULE PLACED IN DECOD3.OBJ
COMPILER INVOKED BY: C:\C51\BIN\C51.EXE DECODER.C CD LC SB FF(3) PL(69) PW(132) ROM(LARGE) SMALL
OT(6,SPEED)

C51 COMPILER V5.02,  SN-  DECOD3
05/13/97  17:22:02  PAGE 4

ASSEMBLY LISTING OF GENERATED OBJECT CODE


            ; FUNCTION _inport (BEGIN)
0000 8F00    R     MOV     porta,R7
                                          ; SOURCE LINE # 17
                                          ; SOURCE LINE # 18
                                          ; SOURCE LINE # 20
0002 7B00          MOV     R3,#00H
0004 7A00    R     MOV     R2,#HIGH porta
0006 7900    R     MOV     R1,#LOW porta
;---- Variable 'in' assigned to Register 'R1/R2/R3' ----
                                          ; SOURCE LINE # 21
0008 120000  E     LCALL   ?C?CLDPTR
000B FF            MOV     R7,A
;---- Variable 'dado' assigned to Register 'R7' ----
                                          ; SOURCE LINE # 22
000C 7E00          MOV     R6,#00H
                                          ; SOURCE LINE # 23
000E         ?C0001:
000E 22            RET
            ; FUNCTION _inport (END)

            ; FUNCTION _outport (BEGIN)
0000 8E00    R     MOV     dado,R6
0002 8F00    R     MOV     dado+01H,R7
0004 8D00    R     MOV     porta,R5
                                          ; SOURCE LINE # 25
                                          ; SOURCE LINE # 26
                                          ; SOURCE LINE # 28
0006 7B00          MOV     R3,#00H
0008 7A00    R     MOV     R2,#HIGH porta
000A 7900    R     MOV     R1,#LOW porta
;---- Variable 'out' assigned to Register 'R1/R2/R3' ----
                                          ; SOURCE LINE # 29
000C EF            MOV     A,R7
000D 120000  E     LCALL   ?C?CSTPTR
                                          ; SOURCE LINE # 30
0010 22            RET
            ; FUNCTION _outport (END)
            ; FUNCTION _xor4 (BEGIN)
0000 8E00    R     MOV     a,R6
0002 8F00    R     MOV     a+01H,R7
0004 8C00    R     MOV     b,R4
0006 8D00    R     MOV     b+01H,R5
;---- Variable 'c' assigned to Register 'R2/R3' ----
                                          ; SOURCE LINE # 32
                                          ; SOURCE LINE # 33
                                          ; SOURCE LINE # 34
;---- Variable 'xor4_i' assigned to Register 'R6/R7' ----
0008 E4            CLR     A
0009 FF            MOV     R7,A
000A FE            MOV     R6,A
                                          ; SOURCE LINE # 35
000B E500    R     MOV     A,a
000D FE            MOV     R6,A
000E E500    R     MOV     A,a+01H
0010 FF            MOV     R7,A
                                          ; SOURCE LINE # 36
0011 E500    R     MOV     A,b
0013 6206          XRL     AR6,A
0015 E500    R     MOV     A,b+01H
```

```
0017 6207          XRL     AR7,A
                                          ; SOURCE LINE # 37
0019 EB            MOV     A,R3
001A 6207          XRL     AR7,A

C51 COMPILER V5.02,  SN-  DECOD3
05/13/97  17:22:02  PAGE 5

001C EA            MOV     A,R2
001D 6206          XRL     AR6,A
                                          ; SOURCE LINE # 38
001F E500    R     MOV     A,d
0021 6206          XRL     AR6,A
0023 E500    R     MOV     A,d+01H
0025 6207          XRL     AR7,A
                                          ; SOURCE LINE # 39
0027 8E00    R     MOV     result,R6
0029 8F00    R     MOV     result+01H,R7
                                          ; SOURCE LINE # 40
002B 22            RET
             ; FUNCTION _xor4 (END)

             ; FUNCTION main (BEGIN)
                                          ; SOURCE LINE # 42
                                          ; SOURCE LINE # 43
0000         ?C0004:
                                          ; SOURCE LINE # 48
0000 E500    R     MOV     A,data_ready+01H
0002 4500    R     ORL     A,data_ready
0004 700B          JNZ     ?C0005
0006 AF00    R     MOV     R7,P1
0008 120000  R     LCALL   _inport
000B 8E00    R     MOV     data_ready,R6
000D 8F00    R     MOV     data_ready+01H,R7
000F 80EF          SJMP    ?C0004
0011         ?C0005:
                                          ; SOURCE LINE # 49
0011 E4            CLR     A
0012 F500    R     MOV     out_ready,A
0014 F500    R     MOV     out_ready+01H,A
                                          ; SOURCE LINE # 50
0016 AF00    R     MOV     R7,out_ready+01H
0018 AE00    R     MOV     R6,out_ready
001A AD00    R     MOV     R5,P2
001C 120000  R     LCALL   _outport
                                          ; SOURCE LINE # 53
;---- Variable 'i' assigned to Register 'R4/R5' ----
001F E4            CLR     A
0020 FD            MOV     R5,A
0021 FC            MOV     R4,A
0022         ?C0006:
                                          ; SOURCE LINE # 54
                                          ; SOURCE LINE # 55
0022         ?C0008:
                                          ; SOURCE LINE # 56
0022 E500    R     MOV     A,strobe+01H
0024 4500    R     ORL     A,strobe
0026 700B          JNZ     ?C0009
0028 AF00    R     MOV     R7,P3
002A 120000  R     LCALL   _inport
002D 8E00    R     MOV     strobe,R6
002F 8F00    R     MOV     strobe+01H,R7
0031 80EF          SJMP    ?C0008
0033         ?C0009:
                                          ; SOURCE LINE # 57
0033 AF00    R     MOV     R7,P3
0035 120000  R     LCALL   _inport
;---- Variable 'decoder_in' assigned to Register 'R2/R3' ----
0038 AB07          MOV     R3,AR7
003A AA06          MOV     R2,AR6
                                          ; SOURCE LINE # 58
003C EB            MOV     A,R3
003D 2500    R     ADD     A,input_data+01H
003F F500    R     MOV     input_data+01H,A
0041 EA            MOV     A,R2

C51 COMPILER V5.02,  SN-  DECOD3
05/13/97  17:22:02  PAGE 6

0042 3500    R     ADDC    A,input_data
0044 F500    R     MOV     input_data,A
                                          ; SOURCE LINE # 59
0046 E500    R     MOV     A,input_data+01H
0048 25E0          ADD     A,ACC
004A F500    R     MOV     input_data+01H,A
004C E500    R     MOV     A,input_data
004E 33            RLC     A
004F F500    R     MOV     input_data,A
                                          ; SOURCE LINE # 60
0051 0D            INC     R5
0052 BD0001        CJNE    R5,#00H,?C0033
0055 0C            INC     R4
0056         ?C0033:
                                          ; SOURCE LINE # 61
0056 ED            MOV     A,R5
0057 6410          XRL     A,#010H
0059 4C            ORL     A,R4
005A 70C6          JNZ     ?C0006
005C         ?C0007:
```

```
                                                ; SOURCE LINE # 64
005C E500    R     MOV     A,input_data
005E 5480          ANL     A,#080H
0060 C4            SWAP    A
0061 13            RRC     A
0062 13            RRC     A
0063 13            RRC     A
0064 5401          ANL     A,#01H
;---- Variable 'top_bit' assigned to Register 'R6/R7' ----
                                                ; SOURCE LINE # 65
0066 750000  R     MOV     global_parity,#00H
0069 F500    R     MOV     global_parity+01H,A
                                                ; SOURCE LINE # 68
006B 7E00          MOV     R6,#00H
006D E500    R     MOV     A,input_data+01H
006F 5401          ANL     A,#01H
0071 FF            MOV     R7,A
0072 C006          PUSH    AR6
0074 C007          PUSH    AR7
0076 E500    R     MOV     A,input_data+01H
0078 5402          ANL     A,#02H
007A FF            MOV     R7,A
007B EE            MOV     A,R6
007C A2E7          MOV     C,ACC.7
007E 13            RRC     A
007F FC            MOV     R4,A
0080 EF            MOV     A,R7
0081 13            RRC     A
0082 FD            MOV     R5,A
0083 E500    R     MOV     A,input_data+01H
0085 5404          ANL     A,#04H
0087 7802          MOV     R0,#02H
0089         ?C0034:
0089 CE            XCH     A,R6
008A A2E7          MOV     C,ACC.7
008C 13            RRC     A
008D CE            XCH     A,R6
008E 13            RRC     A
008F D8F8          DJNZ    R0,?C0034
0091 FB            MOV     R3,A
0092 AA06          MOV     R2,AR6
0094 E500    R     MOV     A,input_data
0096 5402          ANL     A,#02H
0098 FE            MOV     R6,A
0099 E4            CLR     A
009A 7809          MOV     R0,#09H
009C         ?C0035:

C51 COMPILER V5.02,  SN-  DECOD3
05/13/97  17:22:02  PAGE 7

009C CE            XCH     A,R6
009D A2E7          MOV     C,ACC.7
009F 13            RRC     A
00A0 CE            XCH     A,R6
00A1 13            RRC     A
00A2 D8F8          DJNZ    R0,?C0035
00A4 F500    R     MOV     ?_xor4?BYTE+07H,A
00A6 8E00    R     MOV     ?_xor4?BYTE+06H,R6
00A8 850000  R     MOV     ?_xor4?BYTE+08H,aux
00AB 850000  R     MOV     ?_xor4?BYTE+09H,aux+01H
00AE D007          POP     AR7
00B0 D006          POP     AR6
00B2 120000  R     LCALL   _xor4
                                                ; SOURCE LINE # 69
00B5 E500    R     MOV     A,row_parity+01H
00B7 54FE          ANL     A,#0FEH
00B9 FF            MOV     R7,A
00BA E500    R     MOV     A,aux
00BC 4500    R     ORL     A,row_parity
00BE F500    R     MOV     row_parity,A
00C0 E500    R     MOV     A,aux+01H
00C2 4F            ORL     A,R7
00C3 F500    R     MOV     row_parity+01H,A
                                                ; SOURCE LINE # 71
00C5 7E00          MOV     R6,#00H
00C7 E500    R     MOV     A,input_data+01H
00C9 5401          ANL     A,#01H
00CB FF            MOV     R7,A
00CC C006          PUSH    AR6
00CE C007          PUSH    AR7
00D0 E500    R     MOV     A,input_data+01H
00D2 5408          ANL     A,#08H
00D4 7803          MOV     R0,#03H
00D6         ?C0036:
00D6 CE            XCH     A,R6
00D7 A2E7          MOV     C,ACC.7
00D9 13            RRC     A
00DA CE            XCH     A,R6
00DB 13            RRC     A
00DC D8F8          DJNZ    R0,?C0036
00DE FD            MOV     R5,A
00DF AC06          MOV     R4,AR6
00E1 7E00          MOV     R6,#00H
00E3 E500    R     MOV     A,input_data+01H
00E5 5440          ANL     A,#040H
00E7 7806          MOV     R0,#06H
00E9         ?C0037:
00E9 CE            XCH     A,R6
```

```
00EA A2E7          MOV     C,ACC.7
00EC 13            RRC     A
00ED CE            XCH     A,R6
00EE 13            RRC     A
00EF D8F8          DJNZ    R0,?C0037
00F1 FB            MOV     R3,A
00F2 AA06          MOV     R2,AR6
00F4 E500    R     MOV     A,input_data
00F6 5410          ANL     A,#010H
00F8 FE            MOV     R6,A
00F9 E4            CLR     A
00FA 780C          MOV     R0,#0CH
00FC         ?C0038:
00FC CE            XCH     A,R6
00FD A2E7          MOV     C,ACC.7
00FF 13            RRC     A
0100 CE            XCH     A,R6
0101 13            RRC     A
0102 D8F8          DJNZ    R0,?C0038

C51 COMPILER V5.02,  SN-  DECOD3
05/13/97  17:22:02  PAGE 8

0104 F500    R     MOV     ?_xor4?BYTE+07H,A
0106 8E00    R     MOV     ?_xor4?BYTE+06H,R6
0108 850000  R     MOV     ?_xor4?BYTE+08H,aux
010B 850000  R     MOV     ?_xor4?BYTE+09H,aux+01H
010E D007          POP     AR7
0110 D006          POP     AR6
0112 120000  R     LCALL   _xor4
                                           ; SOURCE LINE # 72
0115 E500    R     MOV     A,column_parity+01H
0117 54FE          ANL     A,#0FEH
0119 FF            MOV     R7,A
011A E500    R     MOV     A,aux
011C 4500    R     ORL     A,column_parity
011E F500    R     MOV     column_parity,A
0120 E500    R     MOV     A,aux+01H
0122 4F            ORL     A,R7
0123 F500    R     MOV     column_parity+01H,A
                                           ; SOURCE LINE # 73
0125 7C00          MOV     R4,#00H
0127 E500    R     MOV     A,input_data+01H
0129 5401          ANL     A,#01H
012B FD            MOV     R5,A
012C E500    R     MOV     A,input_data+01H
012E 5402          ANL     A,#02H
0130 FF            MOV     R7,A
0131 E4            CLR     A
0132 A2E7          MOV     C,ACC.7
0134 13            RRC     A
0135 FA            MOV     R2,A
0136 EF            MOV     A,R7
0137 13            RRC     A
0138 FB            MOV     R3,A
0139 7E00          MOV     R6,#00H
013B E500    R     MOV     A,input_data+01H
013D 5404          ANL     A,#04H
013F 7802          MOV     R0,#02H
0141         ?C0039:
0141 CE            XCH     A,R6
0142 A2E7          MOV     C,ACC.7
0144 13            RRC     A
0145 CE            XCH     A,R6
0146 13            RRC     A
0147 D8F8          DJNZ    R0,?C0039
0149 F500    R     MOV     ?_xor4?BYTE+07H,A
014B 8E00    R     MOV     ?_xor4?BYTE+06H,R6
014D 850000  R     MOV     ?_xor4?BYTE+08H,global_parity
0150 850000  R     MOV     ?_xor4?BYTE+09H,global_parity+01H
0153 AF00    R     MOV     R7,global_parity+01H
0155 AE00    R     MOV     R6,global_parity
0157 120000  R     LCALL   _xor4
                                           ; SOURCE LINE # 75
015A 7E00          MOV     R6,#00H
015C E500    R     MOV     A,input_data+01H
015E 5408          ANL     A,#08H
0160 7803          MOV     R0,#03H
0162         ?C0040:
0162 CE            XCH     A,R6
0163 A2E7          MOV     C,ACC.7
0165 13            RRC     A
0166 CE            XCH     A,R6
0167 13            RRC     A
0168 D8F8          DJNZ    R0,?C0040
016A FF            MOV     R7,A
016B C006          PUSH    AR6
016D C007          PUSH    AR7
016F 7E00          MOV     R6,#00H
0171 E500    R     MOV     A,input_data+01H

C51 COMPILER V5.02,  SN-  DECOD3
05/13/97  17:22:02  PAGE 9

0173 5410          ANL     A,#010H
0175 7804          MOV     R0,#04H
0177         ?C0041:
0177 CE            XCH     A,R6
0178 A2E7          MOV     C,ACC.7
```

```
017A 13            RRC     A
017B CE            XCH     A,R6
017C 13            RRC     A
017D D8F8          DJNZ    R0,?C0041
017F FD            MOV     R5,A
0180 AC06          MOV     R4,AR6
0182 7E00          MOV     R6,#00H
0184 E500    R     MOV     A,input_data+01H
0186 5420          ANL     A,#020H
0188 7805          MOV     R0,#05H
018A         ?C0042:
018A CE            XCH     A,R6
018B A2E7          MOV     C,ACC.7
018D 13            RRC     A
018E CE            XCH     A,R6
018F 13            RRC     A
0190 D8F8          DJNZ    R0,?C0042
0192 FB            MOV     R3,A
0193 AA06          MOV     R2,AR6
0195 E500    R     MOV     A,input_data
0197 5404          ANL     A,#04H
0199 FE            MOV     R6,A
019A E4            CLR     A
019B 780A          MOV     R0,#0AH
019D         ?C0043:
019D CE            XCH     A,R6
019E A2E7          MOV     C,ACC.7
01A0 13            RRC     A
01A1 CE            XCH     A,R6
01A2 13            RRC     A
01A3 D8F8          DJNZ    R0,?C0043
01A5 F500    R     MOV     ?_xor4?BYTE+07H,A
01A7 8E00    R     MOV     ?_xor4?BYTE+06H,R6
01A9 850000  R     MOV     ?_xor4?BYTE+08H,aux
01AC 850000  R     MOV     ?_xor4?BYTE+09H,aux+01H
01AF D007          POP     AR7
01B1 D006          POP     AR6
01B3 120000  R     LCALL   _xor4
                                           ; SOURCE LINE # 76
01B6 E500    R     MOV     A,aux+01H
01B8 25E0          ADD     A,ACC
01BA FF            MOV     R7,A
01BB E500    R     MOV     A,aux
01BD 33            RLC     A
01BE FE            MOV     R6,A
01BF E500    R     MOV     A,row_parity+01H
01C1 54FD          ANL     A,#0FDH
01C3 FD            MOV     R5,A
01C4 EE            MOV     A,R6
01C5 4500    R     ORL     A,row_parity
01C7 F500    R     MOV     row_parity,A
01C9 EF            MOV     A,R7
01CA 4D            ORL     A,R5
01CB F500    R     MOV     row_parity+01H,A
                                           ; SOURCE LINE # 78
01CD E500    R     MOV     A,input_data+01H
01CF 5402          ANL     A,#02H
01D1 FF            MOV     R7,A
01D2 E4            CLR     A
01D3 A2E7          MOV     C,ACC.7
01D5 13            RRC     A
01D6 FE            MOV     R6,A

C51 COMPILER V5.02,  SN-  DECOD3
05/13/97  17:22:02  PAGE 10

01D7 EF            MOV     A,R7
01D8 13            RRC     A
01D9 FF            MOV     R7,A
01DA C006          PUSH    AR6
01DC C007          PUSH    AR7
01DE 7E00          MOV     R6,#00H
01E0 E500    R     MOV     A,input_data+01H
01E2 5410          ANL     A,#010H
01E4 7804          MOV     R0,#04H
01E6         ?C0044:
01E6 CE            XCH     A,R6
01E7 A2E7          MOV     C,ACC.7
01E9 13            RRC     A
01EA CE            XCH     A,R6
01EB 13            RRC     A
01EC D8F8          DJNZ    R0,?C0044
01EE FD            MOV     R5,A
01EF AC06          MOV     R4,AR6
01F1 7E00          MOV     R6,#00H
01F3 E500    R     MOV     A,input_data+01H
01F5 5480          ANL     A,#080H
01F7 7807          MOV     R0,#07H
01F9         ?C0045:
01F9 CE            XCH     A,R6
01FA A2E7          MOV     C,ACC.7
01FC 13            RRC     A
01FD CE            XCH     A,R6
01FE 13            RRC     A
01FF D8F8          DJNZ    R0,?C0045
0201 FB            MOV     R3,A
0202 AA06          MOV     R2,AR6
0204 E500    R     MOV     A,input_data
0206 5420          ANL     A,#020H
```

```
0208 FE            MOV     R6,A
0209 E4            CLR     A
020A 780D          MOV     R0,#0DH
020C         ?C0046:
020C CE            XCH     A,R6
020D A2E7          MOV     C,ACC.7
020F 13            RRC     A
0210 CE            XCH     A,R6
0211 13            RRC     A
0212 D8F8          DJNZ    R0,?C0046
0214 F500    R     MOV     ?_xor4?BYTE+07H,A
0216 8E00    R     MOV     ?_xor4?BYTE+06H,R6
0218 850000  R     MOV     ?_xor4?BYTE+08H,aux
021B 850000  R     MOV     ?_xor4?BYTE+09H,aux+01H
021E D007          POP     AR7
0220 D006          POP     AR6
0222 120000  R     LCALL   _xor4
                                           ; SOURCE LINE # 79
0225 E500    R     MOV     A,aux+01H
0227 25E0          ADD     A,ACC
0229 FF            MOV     R7,A
022A E500    R     MOV     A,aux
022C 33            RLC     A
022D FE            MOV     R6,A
022E E500    R     MOV     A,column_parity+01H
0230 54FD          ANL     A,#0FDH
0232 FD            MOV     R5,A
0233 EE            MOV     A,R6
0234 4500    R     ORL     A,column_parity
0236 F500    R     MOV     column_parity,A
0238 EF            MOV     A,R7
0239 4D            ORL     A,R5
023A F500    R     MOV     column_parity+01H,A
                                           ; SOURCE LINE # 80

C51 COMPILER V5.02,  SN-  DECOD3
05/13/97  17:22:02  PAGE 11

023C 7E00          MOV     R6,#00H
023E E500    R     MOV     A,input_data+01H
0240 5408          ANL     A,#08H
0242 7803          MOV     R0,#03H
0244         ?C0047:
0244 CE            XCH     A,R6
0245 A2E7          MOV     C,ACC.7
0247 13            RRC     A
0248 CE            XCH     A,R6
0249 13            RRC     A
024A D8F8          DJNZ    R0,?C0047
024C FD            MOV     R5,A
024D AC06          MOV     R4,AR6
024F 7E00          MOV     R6,#00H
0251 E500    R     MOV     A,input_data+01H
0253 5410          ANL     A,#010H
0255 7804          MOV     R0,#04H
0257         ?C0048:
0257 CE            XCH     A,R6
0258 A2E7          MOV     C,ACC.7
025A 13            RRC     A
025B CE            XCH     A,R6
025C 13            RRC     A
025D D8F8          DJNZ    R0,?C0048
025F FB            MOV     R3,A
0260 AA06          MOV     R2,AR6
0262 7E00          MOV     R6,#00H
0264 E500    R     MOV     A,input_data+01H
0266 5420          ANL     A,#020H
0268 7805          MOV     R0,#05H
026A         ?C0049:
026A CE            XCH     A,R6
026B A2E7          MOV     C,ACC.7
026D 13            RRC     A
026E CE            XCH     A,R6
026F 13            RRC     A
0270 D8F8          DJNZ    R0,?C0049
0272 F500    R     MOV     ?_xor4?BYTE+07H,A
0274 8E00    R     MOV     ?_xor4?BYTE+06H,R6
0276 850000  R     MOV     ?_xor4?BYTE+08H,global_parity
0279 850000  R     MOV     ?_xor4?BYTE+09H,global_parity+01H
027C AF00    R     MOV     R7,global_parity+01H
027E AE00    R     MOV     R6,global_parity
0280 120000  R     LCALL   _xor4
                                           ; SOURCE LINE # 82
0283 7E00          MOV     R6,#00H
0285 E500    R     MOV     A,input_data+01H
0287 5440          ANL     A,#040H
0289 7806          MOV     R0,#06H
028B         ?C0050:
028B CE            XCH     A,R6
028C A2E7          MOV     C,ACC.7
028E 13            RRC     A
028F CE            XCH     A,R6
0290 13            RRC     A
0291 D8F8          DJNZ    R0,?C0050
0293 FF            MOV     R7,A
0294 C006          PUSH    AR6
0296 C007          PUSH    AR7
0298 7E00          MOV     R6,#00H
029A E500    R     MOV     A,input_data+01H
```

```
029C 5480          ANL     A,#080H
029E 7807          MOV     R0,#07H
02A0         ?C0051:
02A0 CE            XCH     A,R6
02A1 A2E7          MOV     C,ACC.7
02A3 13            RRC     A

C51 COMPILER V5.02,  SN-  DECOD3
05/13/97  17:22:02  PAGE 12

02A4 CE            XCH     A,R6
02A5 13            RRC     A
02A6 D8F8          DJNZ    R0,?C0051
02A8 FD            MOV     R5,A
02A9 AC06          MOV     R4,AR6
02AB E500    R     MOV     A,input_data
02AD 5401          ANL     A,#01H
02AF FB            MOV     R3,A
02B0 33            RLC     A
02B1 95E0          SUBB    A,ACC
02B3 FA            MOV     R2,A
02B4 E500    R     MOV     A,input_data
02B6 5408          ANL     A,#08H
02B8 FE            MOV     R6,A
02B9 E4            CLR     A
02BA 780B          MOV     R0,#0BH
02BC         ?C0052:
02BC CE            XCH     A,R6
02BD A2E7          MOV     C,ACC.7
02BF 13            RRC     A
02C0 CE            XCH     A,R6
02C1 13            RRC     A
02C2 D8F8          DJNZ    R0,?C0052
02C4 F500    R     MOV     ?_xor4?BYTE+07H,A
02C6 8E00    R     MOV     ?_xor4?BYTE+06H,R6
02C8 850000  R     MOV     ?_xor4?BYTE+08H,aux
02CB 850000  R     MOV     ?_xor4?BYTE+09H,aux+01H
02CE D007          POP     AR7
02D0 D006          POP     AR6
02D2 120000  R     LCALL   _xor4
                                           ; SOURCE LINE # 83
02D5 E500    R     MOV     A,aux+01H
02D7 AE00    R     MOV     R6,aux
02D9 7802          MOV     R0,#02H
02DB         ?C0053:
02DB C3            CLR     C
02DC 33            RLC     A
02DD CE            XCH     A,R6
02DE 33            RLC     A
02DF CE            XCH     A,R6
02E0 D8F9          DJNZ    R0,?C0053
02E2 FF            MOV     R7,A
02E3 E500    R     MOV     A,row_parity+01H
02E5 54FB          ANL     A,#0FBH
02E7 FD            MOV     R5,A
02E8 EE            MOV     A,R6
02E9 4500    R     ORL     A,row_parity
02EB F500    R     MOV     row_parity,A
02ED EF            MOV     A,R7
02EE 4D            ORL     A,R5
02EF F500    R     MOV     row_parity+01H,A
                                           ; SOURCE LINE # 84
02F1 7E00          MOV     R6,#00H
02F3 E500    R     MOV     A,input_data+01H
02F5 5404          ANL     A,#04H
02F7 7802          MOV     R0,#02H
02F9         ?C0054:
02F9 CE            XCH     A,R6
02FA A2E7          MOV     C,ACC.7
02FC 13            RRC     A
02FD CE            XCH     A,R6
02FE 13            RRC     A
02FF D8F8          DJNZ    R0,?C0054
0301 FF            MOV     R7,A
0302 C006          PUSH    AR6
0304 C007          PUSH    AR7
0306 7E00          MOV     R6,#00H

C51 COMPILER V5.02,  SN-  DECOD3
05/13/97  17:22:02  PAGE 13

0308 E500    R     MOV     A,input_data+01H
030A 5420          ANL     A,#020H
030C 7805          MOV     R0,#05H
030E         ?C0055:
030E CE            XCH     A,R6
030F A2E7          MOV     C,ACC.7
0311 13            RRC     A
0312 CE            XCH     A,R6
0313 13            RRC     A
0314 D8F8          DJNZ    R0,?C0055
0316 FD            MOV     R5,A
0317 AC06          MOV     R4,AR6
0319 E500    R     MOV     A,input_data
031B 5401          ANL     A,#01H
031D FB            MOV     R3,A
031E 33            RLC     A
031F 95E0          SUBB    A,ACC
0321 FA            MOV     R2,A
```

```
0322 E500    R     MOV     A,input_data
0324 5440          ANL     A,#040H
0326 FE            MOV     R6,A
0327 E4            CLR     A
0328 780E          MOV     R0,#0EH
032A         ?C0056:
032A CE            XCH     A,R6
032B A2E7          MOV     C,ACC.7
032D 13            RRC     A
032E CE            XCH     A,R6
032F 13            RRC     A
0330 D8F8          DJNZ    R0,?C0056
0332 F500    R     MOV     ?_xor4?BYTE+07H,A
0334 8E00    R     MOV     ?_xor4?BYTE+06H,R6
0336 850000  R     MOV     ?_xor4?BYTE+08H,aux
0339 850000  R     MOV     ?_xor4?BYTE+09H,aux+01H
033C D007          POP     AR7
033E D006          POP     AR6
0340 120000  R     LCALL   _xor4
                                           ; SOURCE LINE # 85
0343 E500    R     MOV     A,aux+01H
0345 AE00    R     MOV     R6,aux
0347 7802          MOV     R0,#02H
0349         ?C0057:
0349 C3            CLR     C
034A 33            RLC     A
034B CE            XCH     A,R6
034C 33            RLC     A
034D CE            XCH     A,R6
034E D8F9          DJNZ    R0,?C0057
0350 FF            MOV     R7,A
0351 E500    R     MOV     A,column_parity+01H
0353 54FB          ANL     A,#0FBH
0355 FD            MOV     R5,A
0356 EE            MOV     A,R6
0357 4500    R     ORL     A,column_parity
0359 F500    R     MOV     column_parity,A
035B EF            MOV     A,R7
035C 4D            ORL     A,R5
035D F500    R     MOV     column_parity+01H,A
                                           ; SOURCE LINE # 86
035F 7E00          MOV     R6,#00H
0361 E500    R     MOV     A,input_data+01H
0363 5440          ANL     A,#040H
0365 7806          MOV     R0,#06H
0367         ?C0058:
0367 CE            XCH     A,R6
0368 A2E7          MOV     C,ACC.7
036A 13            RRC     A

C51 COMPILER V5.02,  SN-  DECOD3
05/13/97  17:22:02  PAGE 14

036B CE            XCH     A,R6
036C 13            RRC     A
036D D8F8          DJNZ    R0,?C0058
036F FD            MOV     R5,A
0370 AC06          MOV     R4,AR6
0372 7E00          MOV     R6,#00H
0374 E500    R     MOV     A,input_data+01H
0376 5480          ANL     A,#080H
0378 7807          MOV     R0,#07H
037A         ?C0059:
037A CE            XCH     A,R6
037B A2E7          MOV     C,ACC.7
037D 13            RRC     A
037E CE            XCH     A,R6
037F 13            RRC     A
0380 D8F8          DJNZ    R0,?C0059
0382 FB            MOV     R3,A
0383 AA06          MOV     R2,AR6
0385 E500    R     MOV     A,input_data
0387 5401          ANL     A,#01H
0389 F500    R     MOV     ?_xor4?BYTE+07H,A
038B 33            RLC     A
038C 95E0          SUBB    A,ACC
038E F500    R     MOV     ?_xor4?BYTE+06H,A
0390 850000  R     MOV     ?_xor4?BYTE+08H,global_parity
0393 850000  R     MOV     ?_xor4?BYTE+09H,global_parity+01H
0396 AF00    R     MOV     R7,global_parity+01H
0398 AE00    R     MOV     R6,global_parity
039A 120000  R     LCALL   _xor4
                                           ; SOURCE LINE # 89
039D E500    R     MOV     A,global_parity+01H
039F 4500    R     ORL     A,global_parity
03A1 701F          JNZ     ?C0010
                                           ; SOURCE LINE # 90
03A3 E500    R     MOV     A,row_parity+01H
03A5 20E012        JB      ACC.0,?C0011
03A8 20E10F        JB      ACC.1,?C0011
03AB E500    R     MOV     A,column_parity+01H
03AD 20E00A        JB      ACC.0,?C0011
03B0 20E107        JB      ACC.1,?C0011
                                           ; SOURCE LINE # 92
03B3 E4            CLR     A
03B4 F500    R     MOV     error,A
03B6 F500    R     MOV     error+01H,A
03B8 806B          SJMP    ?C0013
03BA         ?C0011:
```

```
                                        ; SOURCE LINE # 93
03BA 750000  R     MOV     error,#00H
03BD 750003  R     MOV     error+01H,#03H
                                        ; SOURCE LINE # 94
03C0 8063          SJMP    ?C0013
03C2         ?C0010:
                                        ; SOURCE LINE # 96
                                        ; SOURCE LINE # 97
03C2 750000  R     MOV     error,#00H
03C5 750002  R     MOV     error+01H,#02H
                                        ; SOURCE LINE # 98
03C8 E500    R     MOV     A,row_parity+01H
03CA 30E01E        JNB     ACC.0,?C0014
                                        ; SOURCE LINE # 99
03CD E500    R     MOV     A,column_parity+01H
03CF 30E005        JNB     ACC.0,?C0015
03D2 630001  R     XRL     input_data+01H,#01H
03D5 804E          SJMP    ?C0013
03D7         ?C0015:
                                        ; SOURCE LINE # 101
03D7 E500    R     MOV     A,column_parity+01H

C51 COMPILER V5.02,  SN-  DECOD3
05/13/97  17:22:02  PAGE 15

03D9 30E105        JNB     ACC.1,?C0017
03DC 630002  R     XRL     input_data+01H,#02H
03DF 8044          SJMP    ?C0013
03E1         ?C0017:
                                        ; SOURCE LINE # 103
03E1 E500    R     MOV     A,column_parity+01H
03E3 30E23F        JNB     ACC.2,?C0013
03E6 630004  R     XRL     input_data+01H,#04H
                                        ; SOURCE LINE # 104
03E9 803A          SJMP    ?C0013
03EB         ?C0014:
                                        ; SOURCE LINE # 106
03EB E500    R     MOV     A,row_parity+01H
03ED 30E11E        JNB     ACC.1,?C0021
                                        ; SOURCE LINE # 107
03F0 E500    R     MOV     A,column_parity+01H
03F2 30E005        JNB     ACC.0,?C0022
03F5 630008  R     XRL     input_data+01H,#08H
03F8 802B          SJMP    ?C0013
03FA         ?C0022:
                                        ; SOURCE LINE # 109
03FA E500    R     MOV     A,column_parity+01H
03FC 30E105        JNB     ACC.1,?C0024
03FF 630010  R     XRL     input_data+01H,#010H
0402 8021          SJMP    ?C0013
0404         ?C0024:
                                        ; SOURCE LINE # 111
0404 E500    R     MOV     A,column_parity+01H
0406 30E21C        JNB     ACC.2,?C0013
0409 630020  R     XRL     input_data+01H,#020H
                                        ; SOURCE LINE # 112
040C 8017          SJMP    ?C0013
040E         ?C0021:
                                        ; SOURCE LINE # 114
040E E500    R     MOV     A,row_parity+01H
0410 30E212        JNB     ACC.2,?C0013
                                        ; SOURCE LINE # 115
0413 E500    R     MOV     A,column_parity+01H
0415 30E005        JNB     ACC.0,?C0029
0418 630040  R     XRL     input_data+01H,#040H
041B 8008          SJMP    ?C0013
041D         ?C0029:
                                        ; SOURCE LINE # 117
041D E500    R     MOV     A,column_parity+01H
041F 30E103        JNB     ACC.1,?C0013
0422 630080  R     XRL     input_data+01H,#080H
                                        ; SOURCE LINE # 118
                                        ; SOURCE LINE # 119
0425         ?C0013:
                                        ; SOURCE LINE # 123
;---- Variable 'data_out' assigned to Register 'R6/R7' ----
0425 AF00    R     MOV     R7,input_data+01H
0427 AE00    R     MOV     R6,input_data
                                        ; SOURCE LINE # 124
0429 850000  R     MOV     err,error
042C 850000  R     MOV     err+01H,error+01H
                                        ; SOURCE LINE # 125
042F 750000  R     MOV     out_ready,#00H
0432 750001  R     MOV     out_ready+01H,#01H
                                        ; SOURCE LINE # 127
0435 AD00    R     MOV     R5,P3
0437 120000  R     LCALL   _outport
                                        ; SOURCE LINE # 128
043A AF00    R     MOV     R7,err+01H
043C AE00    R     MOV     R6,err
043E AD00    R     MOV     R5,P2
0440 120000  R     LCALL   _outport

C51 COMPILER V5.02,  SN-  DECOD3
05/13/97  17:22:02  PAGE 16

                                        ; SOURCE LINE # 129
0443 AF00    R     MOV     R7,out_ready+01H
0445 AE00    R     MOV     R6,out_ready
```

```
0447 AD00    R     MOV     R5,P1
0449 120000  R     LCALL   _outport
                                          ; SOURCE LINE # 131
044C E4            CLR     A
044D F500    R     MOV     out_ready,A
044F F500    R     MOV     out_ready+01H,A
                                          ; SOURCE LINE # 132
0451 AF00    R     MOV     R7,out_ready+01H
0453 AE00    R     MOV     R6,out_ready
0455 AD00    R     MOV     R5,P0
0457 120000  R     LCALL   _outport
                                          ; SOURCE LINE # 134
045A 22            RET
             ; FUNCTION main (END)

C51 COMPILER V5.02,  SN-  DECOD3
05/13/97  17:22:02  PAGE 17

NAME                                  CLASS   MSPACE  TYPE    OFFSET  SIZE
====                                  =====   ======  ====    ======  ====


_inport. . . . . .  PUBLIC   CODE   PROC     -----  -----
  porta. . . . . .  AUTO     DATA   U_CHAR   0000H  1
  in . . . . . .    * REG *  DATA   PTR      0001H  3
  dado . . . . . .  * REG *  DATA   U_CHAR   0007H  1
P0 . . . . . .      PUBLIC   DATA   U_CHAR   0000H  1
P1 . . . . . .      PUBLIC   DATA   U_CHAR   0001H  1
_outport . . . . .  PUBLIC   CODE   PROC     -----  -----
  dado . . . . . .  AUTO     DATA   INT      0000H  2
  porta. . . . . .  AUTO     DATA   U_CHAR   0002H  1
  out. . . . . .    * REG *  DATA   PTR      0001H  3
P2 . . . . . .      PUBLIC   DATA   U_CHAR   0002H  1
P3 . . . . . .      PUBLIC   DATA   U_CHAR   0003H  1
main . . . . . .    PUBLIC   CODE   PROC     -----  -----
  i. . . . . .      * REG *  DATA   INT      0004H  2
  input_data . . .  AUTO     DATA   INT      0000H  2
  row_parity . . .  AUTO     DATA   INT      0002H  2
  column_parity. .  AUTO     DATA   INT      0004H  2
  global_parity. .  AUTO     DATA   INT      0006H  2
  error. . . . . .  AUTO     DATA   INT      0008H  2
  top_bit. . . . .  * REG *  DATA   INT      0006H  2
  aux. . . . . .    AUTO     DATA   INT      000AH  2
  decoder_in . . .  * REG *  DATA   INT      0002H  2
  data_ready . . .  AUTO     DATA   INT      000CH  2
  strobe . . . . .  AUTO     DATA   INT      000EH  2
  data_out . . . .  * REG *  DATA   INT      0006H  2
  err. . . . . .    AUTO     DATA   INT      0010H  2
  out_ready. . . .  AUTO     DATA   INT      0012H  2
_xor4. . . . . .    PUBLIC   CODE   PROC     -----  -----
  a. . . . . .      AUTO     DATA   INT      0000H  2
  b. . . . . .      AUTO     DATA   INT      0002H  2
  c. . . . . .      AUTO     DATA   INT      0004H  2
  d. . . . . .      AUTO     DATA   INT      0006H  2
  result . . . . .  AUTO     DATA   INT      0008H  2
  xor4_i . . . . .  * REG *  DATA   INT      0006H  2
MODULE INFORMATION:   STATIC OVERLAYABLE
   CODE SIZE        =   1191    ----
   CONSTANT SIZE    =   ----    ----
   XDATA SIZE       =   ----    ----
   PDATA SIZE       =   ----    ----
   DATA SIZE        =      4      34
   IDATA SIZE       =   ----    ----
   BIT SIZE         =   ----    ----
END OF MODULE INFORMATION.
C51 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

Bibliografia

[AHO 88] AHO, A.; SETHI, R.; ULLMAN, J. **Compilers, Principles, Techniques and Tools**. Massachusetts: Addison-Wesley, Reading, 1986. 796p.

[ALB 96] ALBA, C. **Sistema de geração de microcontroladores para aplicações específicas**. Porto Alegre: CPGCC da UFRGS, 1996. 109p. Dissertação de Mestrado.

[ALO 93] ALOMARY, A. et al. PEAS-I: A hardware/software co-design system for ASIPs. In: EUROPEAN DESIGN AUTOMATION CONFERENCE, 1993, Hamburg, Germany. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society Press, 1993. p. 2-7.

[ALT 92] ALTERA CORPORATION. **MAX + PLUS II:** AHDL. San Jose: The Altera Corporation, 1992. 314p.

[ATH 93] ATHANAS, P.; SILVERMAN, H. Processor Reconfiguration Through Instruction-Set Metamorphosis. **Computer**, New York, v.26, n.3, p.11-18, Mar. 1993.

[AUG 90] AUGUIN, M. et al. From program to hardware: A parallel architecture compiler. **Microprocessing and Microprogramming**, Amsterdam, v.30, n.1-5, p.467-474, Aug. 1990.

[BAZ 94] BAZARGAN-SABET, P. et al. Methodology of Development of a Complete CAD System for Digital VLSI Design - ALLIANCE System Tools. In: SIMPÓSIO BRASILEIRO DE CONCEPÇÃO DE CIRCUITOS INTEGRADOS, 8., 1994, Gramado. **Anais...** Porto Alegre: SBC e CPGCC da UFRGS, 1994. p.11-15.

[BIN 95] BINH, N. et al. A hardware/software partitioning algorithm for pipelined instruction set processor. In: EUROPEAN DESIGN AUTOMATION CONFERENCE, 1995, Brighton, UK. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society Press, 1995. p. 176-181.

[BIT 95] BITWARE/RAINBOLT & ASSOCIATES. **Keil 8051 C Compiler, version 5.0.** [S.l.]: Bitware/Rainbolt & Associates, 1995.

[BOT 94] BOTH, A. et al. Hardware-Software-Codesign of Application Specific Microcontrollers with the ASM Environment. In: THE EUROPEAN DESIGN AUTOMATION CONFERENCE WITH EURO-VHDL AND EXHIBITION, 1994, Grenoble. **Proceedings...** New York: Sigda Publications, ACM, 1994. p.91-95. (CD-ROM).

[CAR 96] CARRO, L. **Algoritmos e Arquiteturas para o desenvolvimento de Sistemas Computacionais**. Porto Alegre: CPGCC da UFRGS, 1996. Tese de Doutorado.

[CAR 96a] CARRO, L.; ALBA, C.; SUZIM, A. Dedicated Microcontroller as ASIPS. In: WORKSHOP IBERCHIP, 2., São Paulo. **Anais...** São Paulo: LSI/USP, 1996. p. 260-266.

[CAR 96b] CARRO, L. et al. System Design using Asips. In: INTERNATIONAL IEEE SYMPOSIUM ON ENGENEERING OF COMPUTER-BASED SYSTEMS, 1996, Friedrichshafen, Germany. **Proceedings...** [S.l. : s.n.], 1996.

[GAJ 92] GAJSKI, D. et. al. **High-Level Synthesis. Introduction to Chip and System Design.** [S.l.]: Kluwer Academic Publishers, 1992.

[GAJ 93] GAJSKI, D. Design process beyond ASICs. In: THE EUROPEAN CONFERENCE ON DESIGN AUTOMATION WITH THE EUROPEAN EVENT IN ASIC DESIGN, 1993, Paris. **Proceedings...** Los Alamitos, CA: IEEE Computer Society Press, 1993. p.3-4.

[GOV 96] GOVINDARAJAM, R. et al. Co-scheduling hardware and software pipelines. In: INTERNATIONAL SYMPOSIUM ON HIGH-PERFORMANCE COMPUTER ARCHITECTURE, 2., 1996, San Jose, CA, USA. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society Press, 1996. p.52-61.

[HEN 90] HENNESSY, J.; PATTERSON, D. **Computer Architecture A quantitative approach.** San Mateo: Morgan Kaufmann, 1990. 755p.

[HUA 94] HUANG, I.; DESPAIN, A. Synthesis of Instruction Sets for Pipelined Microprocessors. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 1994, San Jose, CA, USA. **Proceedings...** New York, NY, USA: ACM, 1994. P. 5-11.

[HUA 94a] HUANG, I.; DESPAIN, A. M. Generating instrucion sets and microarchitectures from applications. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1994, San Jose, CA, USA. **Proceedings...** New York, NY, USA: ACM, 1994. p. 391-396.

[HWA 84] HWANG, K.; BRIGGS, F. A. **Computer Architecture and Parallel Processing.** [S.l.]: McGraw Hill, 1984.

[IMA 92] IMAI, M. et al. An integer programming approach to instruction implementation method selection problem. In: EUROPEAN DESIGN AUTOMATION CONFERENCE, 1992, Hamburg, Germany. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society Press, 1992. p. 106-111.

[JUN 93] JUNQUEIRA, A. **Risco**: Microprocessador RISC CMOS de 32 bits. Porto Alegre: CPGCC da UFRGS, 1993. 256p. Dissertação de Mestrado.

[KOG 81] KOGGE, P. M. **The Architecture of Pipelined Computers.** [S.l.]: McGraw Hill, 1981.

[KRE 97] KREUTZ, M.; CARRO, L.; SUZIM, A. **System integration with dedicated processor for industrial applications.** Paper aceito para publicação no SICICA 97, Annecy, France, junho 1997.

[KRE 97a] KREUTZ, M.; CARRO, L.; SUZIM, A. Síntese do processador 8051 dedicado com arquitetura pipeline. In: WORKSHOP IBERCHIP, 3, 1997. **Proceedings...** Mexico: [s.n.], 1997. p.218-226.

[LEV 89] LEVEUGLE, R.; SOUEIDAN, M. Design of an Application Specific Microprocessor. In: INTERNATIONAL WORKSHOP ON LOGIC AND ARCHITECTURE SYNTHESIS FOR SILICON COMPILERS, 1988. **Proceedings ...** [S.l. : s.n.], 1988. P. 255-268.

[LIE 94] LIEM, C.; MAY, T.; PAULIN, P. Instruction-Set Matching and Selection for DSP and ASIP Code Generation. In: THE EUROPEAN DESIGN AND TEST CONFERENCE, 1994, Paris. **Proceedings...** Los Alamitos, CA: IEEE Computer Society Press, 1994. p.31-37.

[MED 91] MEDEIROS, A. F. **CCC51, um Compilador Cruzado "C" para a famíia de Microcontroladores Intel MCS-51.** Trabalho de Diplomação. Porto Alegre: [s.n.], 1991.

[MIC 94] MICHELI, G. de. Computer-Aided Hardware-Software Codesign. **IEEE Micro**, New York, v.14, n.4, p.10-16, Aug. 1994.

[NAV 87] NAVAUX, P. O. A. Introdução às Arquiteturas de Computadores para Processamento Paralelo. In: JORNADA DE ATUALIZAÇÃO EM INFORMÁTICA, 6., 1987, Salvador, BA. **Anais...** [S.l. : s.n.], 1987.

[NGU 95] NGUYEN, N. et al. An instruction set optimization algorithm for pipelined ASIPs. **IEICE Transactions on Fundamentals of Eletronics, Communications and Computer Sciences**, Japan, v. E78A, n. 12, p. 1707-1714, Dec. 1995.

[RAM 77] RAMAMOORTHY, C. V. Pipeline Architecture. **Computing Surveys**, [S.l.], v. 9, n. 1, Mar. 1977.

[VAN 94] VAN PRAET, J. et al. Instruction set definition and instruction selection for ASIPs. In: INTERNATIONAL SYMPOSIUM ON HIGH-LEVEL SYNTHESIS, 7., 1994, Ontario, Canada. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society Press, 1994. p. 11-16.