UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Implementação de um sistema de Síntese de Alto Nível baseado em modelos Java

por

DÉBORA BERTASI

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

Prof. Dr. Luigi Carro
Orientador

Porto Alegre, fevereiro de 2002.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Bertasi, Débora
Implementação de um sistema de Síntese de Alto Nível baseado em modelos Java / por Débora Bertasi. – Porto Alegre: PPGC da UFRGS. 2002.
76p.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2002. Orientador: Carro, Luigi.

1. Síntese de Alto Nível. 2. ASICs. 3. VHDL. 4. Sistemas Embutidos. 5. Ambiente CAD. I. Carro, Luigi. II. Título.

Agradecimentos

Este trabalho foi desenvolvido graças à bolsa concedida pela CNPq, a quem registro aqui os meus mais sinceros agradecimentos.

Gostaria de dedicar especial agradecimento a meus pais, Nelsi e Darci, pelo contínuo incentivo ao estudo, e por saberem compreender os momentos de ausência durante o mestrado.

Ao meu orientador, Luigi Carro agradeço sua confiança inicial, sua disponibilidade para a orientação, suas críticas e sugestões e, principalmente, sua amizade e carinho. Muito mais que orientador, foi um amigo inestimável, proporcionando um excelente ambiente de trabalho, além do exemplo como docente, pela sua dedicação e empenho.

Aos meus queridos amigos e professores do Grupo da Microeletrônica, em especial a minha turma do mestrado – Tatiane, Luciano, Alex, Roberto e Flávio, agradeço pelas dicas, amizade, aprendizado e pelo apoio. Um agradecimento especial ao meu amigo Sérgio Akira Ito, que sempre me auxiliou com conselhos e opiniões técnicas, para a concretização do meu trabalho. A Rita, Ana, Daniela e Silvana pelo apoio dado na minha estada em Porto Alegre e nos momentos mais difíceis com a minha vinda a Foz do Iguaçu, bem como pelo continuado incentivo e papos descontraídos.

A todos os membros do Instituto de Informática pelas valiosas idéias, ao ambiente acadêmico onde “respiramos conhecimento”, ao excelente ambiente de trabalho e pela amizade, jamais esquecerei esse momento que passei com vocês, MUITO OBRIGADA!!

A todas as pessoas que de uma forma ou outra ajudaram na concretização do trabalho o meu sincero agradecimento.

Se não houver frutos,
valeu a beleza das flores.
Se não houver flores,
valeu a sombra das folhas.
Se não houver folhas,
valeu a intenção da
semente.

(Henfil)

Sumário

Lista de Abreviaturas

ALAP	*As Late As Possible*
ALU	*Arithmetic and Logic Unit*
ASAP	A*s Soon As Possible*
ASIC	*Application Specific Integrated Circuits*
ASIP	Application Specific Instruction set Processor
CAD	*Computer Aided Design*
CI	Circuito Integrado
CPI	Relógios Por Instrução
DSP	Processamento de Sinais Digitais
FIR	Filtros de Resposta à Impulsos Finitos
FSM	Máquina de Estados Finita
FSMD	Máquina de Estados Finita com Datapath
FPGAs	*Field Programmable Gate Arrays*
HDL	H*ardware Description Language*
HIS	High Level Synthesis IBM System
JDK	Java Development Kit
LSI	*Large Scale Integratio*n
MIMOLA	*Machine Independent MicroprOgramming Language*
MVJ	Máquina Virtual Java
RAM	*Random Access Memory*
ROM	*Read Only Memory*
RT	Transferência entre Registradores
RTL	Nível de Transferência entre Registradores
SASHIMI	*Systems As Software and Harwdware for Microcontrollers*
VHDL	*Very High speed integrated circuit*
VLSI	Very Large Scale Integration

Lista de Figuras

Lista de Tabelas

Resumo

Este trabalho apresenta uma metodologia para a geração automática de ASICs, em VHDL, a partir da linguagem de entrada Java. Como linguagem de especificação adotou-se a Linguagem Java por esta possuir características desejáveis para especificação a nível de sistema, como: orientação a objetos, portabilidade e segurança. O sistema é especificamente projetado para suportar síntese de ASICs a partir dos modelos de computação Máquina de Estados Finita e Pipeline. Neste trabalho, adotou-se estes modelos de computação por serem mais usados em sistemas embarcados

As principais características exploradas são a disponibilização da geração de ASICs para a ferramenta SASHIMI, o alto nível de abstração com que o projetista pode contar em seu projeto, as otimizações de escalonamento realizadas automaticamente, e o sistema ser capaz de abstrair diferentes modelos de computação para uma descrição em VHDL. Portanto, o ambiente permite a redução do tempo de projeto e, consequentemente, dos custos agregados, diminuindo a probabilidade de erros na elaboração do projeto, portabilidade e reuso de código – através da orientação a objetos de Java – podendo-se proteger os investimentos prévios em desenvolvimento de software.

A validação desses conceitos foi realizada mediante estudos de casos, utilizando-se algumas aplicações e analisando os resultados obtidos com a geração dos ASICs.

Palavras-Chave: Síntese de Alto Nível, ASICs, VHDL, Sistemas Embutidos, Ambiente de CAD.

TITLE: “A IMPLEMENTATION OF A HIGH LEVEL SYNTHESIS SYSTEM BASED ON JAVA MODEL.”

Abstract

This work presents a methodology for the automatic generation of ASICs, in VHDL, using Java as the input language. The Java language was adopted as the specification language because it has desirable features for specification at the system level. The system is specifically designed to support ASICs synthesis from Finite States Machine and Pipeline computation models. In this work, we adopted these computation models as they are oftenlyfor being more used in embedded systems.

The main goals of this work are to make ASICs generation available in the SASHIMI tool, allow a the high level of abstraction to the design, the scheduling optimizations automatically carried through and the system being automatically capable to abstract different computation models for a VHDL basead synthesis. Therefore, the environment allows the reduction of design time and the aggregated costs, diminishes the probability of errors in the design elaboration, enhancing portability and code reuse. Also, by means of Java´s object orientation one can protect the previous investments in software development. The validation of these concepts was carried through by means of case studies, using some applications and analyzing the results obtained with the generation of the ASICs.

Keywords: High Level Synthesis, ASICs, VHDL, Embedded System, CAD.

1 Introdução

Atualmente, circuitos digitais estão sendo usados em muitas aplicações, como nas indústrias de computadores, na indústria automobilística, em equipamentos telefônicos e em numerosas outras áreas de aplicações. Devido ao rápido progresso na fabricação de circuitos integrados digitais, a tarefa de projetar os sistemas digitais torna-se mais complexa. A complexidade dos circuitos que estão sendo projetados pode ser comparada ao número de transistores na implementação de circuitos integrados. O número de transistores de um único circuito integrado em 1975, era de centenas, já em 1980, o número cresceu para vários milhares de transistores. Muitos dos grandes projetos fabricados nos dias atuais possuem milhões de transistores [GAJ92], [AJL99], [LAV00].

O projeto de grandes circuitos com milhões de transistores é complexo, e o tempo de projeto é muito grande. Realmente, como o nível dos circuitos integrados aumentou, realizar projetos completos sem erros de projeto transforma-se em uma tarefa cada vez mais difícil de se realizar manualmente pelos projetistas. Otimizações manuais do circuito, para um alta qualidade de projeto, são muito difíceis devido ao número de opções crescendo rapidamente com o aumento da complexidade. A tarefa de verificar o comportamento desejado dos circuitos e testar para detectar defeitos também se tornou muito complexa.

A dificuldade nos projetos de circuitos VLSI tornou o Projeto Auxiliado por Computador (CAD – *Computer Aided Design*) de circuitos integrados uma tecnologia chave para a indústria de projeto de circuitos. Técnicas de projetos auxiliados por computador representam uma tarefa importante na redução de tempo de projeto. Uma ferramenta automatizada é o ideal para evolução rápida de diferentes escolhas para produção de circuitos otimizados. Ferramentas de CAD são indispensáveis para os projetistas, não somente para projetar circuitos eficientes sem erros, mas também para diminuir o tempo e consequentemente o custo de projeto, para ajudar a otimizar os parâmetros do projeto e reduzir o tempo de chegada ao mercado. A diminuição nos custos do projeto é muito importante para muitos circuitos integrados de aplicações específicas (ASICs – *Application Specific Integrated Circuits*), não somente pela grande quantidade de comercialização, mas pelo pequeno tempo de vida.

Um circuito pode ser especificado por vários níveis de abstração: em nível de transistor, em nível de portas lógicas ou em nível de algoritmo. Com o aumento da complexidade dos circuitos, a tendência é aumentar os níveis de abstração na especificação do circuito.

Com o grande crescimento em complexidade e tamanho dos sistemas de hardware, aumenta a procura por ferramentas de síntese para auxiliar o projetista no projeto de circuitos digitais. Os sistemas de síntese são apresentados por serem muito efetivos, suportando o projeto de circuitos digitais e, em particular, o projeto de ASICs.

Um circuito pode ser projetado a partir de uma descrição específica de alto nível, que pode ser muito independente da tecnologia alvo ou estilo de projeto. Descrito em alto nível, o projeto é mais portável; mudanças são incorporadas mais facilmente, e o ciclo de vida do projeto é provavelmente maior. Síntese automática requer o uso de ferramentas de computação, integradas para capturar e otimizar o processo do projeto e para obter e otimizar o projeto em diferentes níveis de representação. Existe a

necessidade de automatizar os processos dos projetos para se obter o projeto em pouco tempo.

Outra razão para adotar metodologias de projetos de alto nível é o nível de abstração necessário aos projetistas na realização de seus projetos. É difícil imaginar um projetista especificando e documentando o projeto de chip em termos de um esquemático de circuito com 100.000 portas lógicas, ou uma descrição com 100.000 expressões Booleanas.

Com o aumento da complexidade dos projetos, tornou-se impossível um projetista compreender a funcionalidade de um chip ou a especificação de um sistema completamente apenas com os circuitos ou o esquema lógico. Síntese de alto nível é uma forte tendência para projetos de metodologias de sistemas VLSI.

A síntese automática é a transformação de uma especificação realizada em um determinado nível de abstração, no nível mais próximo da realização física, com a aplicação de um conjunto de ferramentas que acrescentam detalhes estruturais e/ou geométricos à especificação inicial [WAG88], [GAJ92].

A síntese automática de sistemas digitais, a partir do nível de transferência entre registradores (RT) até o nível de *layout*, já está bastante consolidada, com ferramentas razoavelmente eficientes e comercialmente disponíveis [GAJ92]. Com isso, duas fortes tendências têm se destacado nos últimos anos: a migração de ferramentas de projeto para os níveis de abstração mais altos, notadamente para os níveis algorítmicos e de sistema, e a adoção de HDL (*Hardware Description Language* – Linguagem de Descrição de Hardware) , como padrão para a descrição de circuitos.

A adoção de uma e outra dessas tendências na síntese de circuitos é cercada de problemas e controvérsias. Entretanto, as vantagens superam os problemas. Algoritmos desenvolvidos para a síntese de alto nível, como alocação e escalonamento, vem migrando para ferramentas comerciais, baseando-se no grande número de publicações sobre o tema [GAJ92], [LIS88], [LIP90], [GLU90], [CAM91].

1.1 Características da Metodologia Proposta

O sucesso de sistemas de síntese de alto nível é altamente dependente da maneira de capturar efetivamente, na linguagem de entrada do sistema, as idéias do projetista de uma maneira simples e compreensível. Neste trabalho, um sistema de síntese de alto nível, baseado em modelos Java, é apresentado. O sistema é especificamente projetado para suportar síntese de ASICs a partir dos modelos de computação Máquina de Estados Finita (FSM) e Pipeline. Neste trabalho, adotou-se estes modelos de computação por serem mais usados em sistemas embarcados. Através de classes, há a possibilidade de verificar qual o melhor estilo de síntese que o projetista irá usar, isto é, definindo o corpo do modelo e definindo o nome da classe de FSM, o sistema irá sintetizar usando as características do modelo FSM. Já para o sistema sintetizar usando o modelo pipeline, o projetista somente mudará o nome da classe para pipeline e sintetizará segundo um novo modelo. Portanto, o estilo da sínteze é captura pelo modelo Java que o projetista escolher.

O sistema utilizará estes modelos computacionais devido à melhor adaptação de cada um a diferentes arquiteturas. Por exemplo, um modelo *DataFlow* é melhor utilizado para projetar circuitos com muito paralelismo, já o modelo Pipeline caracteriza-se por arquitetura que possui dados chegando ao sistema em uma taxa fixa, como em filtros digitais.

Para a implementação, utilizou-se a linguagem de programação Java [SUN95], [SUN95a], [SUN95b]. Essa linguagem tem ganho um destaque especial no cenário mundial dos sistemas de informação por apresentar propriedades como portabilidade, orientação a objetos e segurança. Possui ainda características interessantes do ponto de vista de aplicações embutidas, em função da independência do hardware hospedeiro, oferecida por essa tecnologia, conferindo alto grau de portabilidade às aplicações [ITO99]. Portanto, portabilidade e reuso de código – através da orientação a objetos de Java – podem proteger os investimentos prévios em desenvolvimento de software.

O presente trabalho teve como objetivo o estudo para complementar o trabalho SASHIMI (*Systems As Software and Harwdware for Microcontrollers*), ambiente para geração de aplicações específicas baseado em software e hardware para microcontroladores, que utiliza a linguagem Java como tecnologia fundamental para o projeto. Neste ambiente, o projetista fornece uma aplicação Java que será analisada e otimizada para execução em um microcontrolador batizado de FemtoJava, capaz de executar instruções Java nativamente, exibindo ainda características de um ASIP [ITO99a].

Este trabalho disponibiliza a ferramenta de geração de ASIC para o SASHIMI, fornecendo maior flexibilidade e inclusão de circuitos específicos, que permitem acelerar a execução da aplicação pelo projetista . O sistema de geração de ASICs interpreta o modelo computacional que o projetista forneceu, armazenando-o em estruturas de dados. Sobre a estrutura de dados, são feitas transformações parciais, onde se busca a obtenção da maior quantidade de paralelismo possível [BER00]. A partir daí, a descrição VHDL em nível RT, sintetizável por ferramentas comerciais como Altera [ALT96] ou Mentor [MEN93] é gerada.

A característica de obter precisamente qual o modelo que será abstraído permite capturar precisamente o comportamento do componente do sistema, governado por um certo modelo computacional particular. Tal precisão traduz-se em eficiência da ferramenta de síntese, por exemplo, podendo capturar um ótimo escalonamento com custo de memória minimizado, enquanto remove o obstáculo de performance.

Para o escalonamento, foram implementados os algoritmos de síntese de alto nível, ASAP (A*s Soon As Possible*), ALAP (*As Late As Possible*), *List-Scheduling* e *Schedule Pipeline*. Além dos vários modelos para a abstração do código VHDL, este trabalho deverá fornecer suporte à ferramenta de especificação e síntese de sistemas baseados em processadores Java, como descrito em [ITO99], fornecendo os modelos para os usuários e especializando as ferramentas de síntese. Quando, eventualmente, uma aplicação possuir requisitos de tempo muito exigentes para o processador, o código Java deverá ser traduzido em um código VHDL sintetizável, de modo que se consiga executar o algoritmo com maior velocidade.

A descrição em VHDL gerada pelo sistema poderá ser abstraída de várias maneiras, dependendo do modelo de computação especificado. Cada modelo é especificado permitindo capturar o comportamento do componente do sistema governado por um certo modelo computacional particular.

Embora já existam comercialmente sistemas de síntese de alto nível, este trabalho diferencia-se dos demais por permitir vários comportamentos do código VHDL gerado pelo sistema. Quando um modelo específico é disponível, consegue-se explorar as otimizações pertinentes a cada modelo.

Utilizando o sistema de síntese proposto, o projetista deverá primeiramente descrever o sistema usando modelos que herdam características particulares de síntese. Isso facilitará a descrição do projeto e possibilitará o reuso. Após, o projetista irá executar o projeto na ferramenta SASHIMI, seguindo o ciclo tradicional edição-compilação-execução, e verificar o desempenho do sistema projetado. O usuário do SASHIMI deverá verificar se alguma rotina executa num tempo maior que o esperado.

Quando uma estimativa de desempenho não satisfaz os requisitos da aplicação, partes críticas do código poderão ser identificadas pela ferramenta de análise, e o projetista poderá intervir na adaptação do código. Uma vez identificada uma ou mais rotinas cuja execução não atende aos requisitos de desempenho, os modelos indicarão qual o melhor estilo de síntese para geração automática de um ASIC.

Na medida em que o projetista utilizar os modelos, o sistema conseguirá capturar o estilo de síntese mais adequado para explorar o espaço de projeto, conferindo um melhor desempenho à aplicação.

1.2 Organização da Dissertação

O presente trabalho está organizado de acordo com a seguinte estrutura:

O Capítulo 2 discorre sobre as características desejáveis de um sistema de síntese de alto nível, algoritmos de otimização e o estudo de algumas ferramentas de síntese de alto nível.

O Capítulo 3 apresenta o modelo FSM em Java e em VHDL, apresentando os processos de captura, otimização e geração do código otimizado, apresentando alguns exemplos.

O Capítulo 4 apresenta os detalhes da implementação dos modelos pipeline e seu algoritmo de escalonamento, mostrando o modelo em Java e o código gerado em VHDL.

O Capítulo 5 apresenta o sistema proposto neste trabalho, (SiMo – Sistema de Síntese de Alto Nívrel Baseado em Modelos) e a sua ligação com o SASHIMI.

O Capítulo 6 é dedicado à conclusão, incluindo trabalhos futuros e contribuições da dissertação.

2 Introdução à Síntese de Alto Nível

Atualmente, a maioria dos sistemas ambarcados, sistemas de computação dedicados a aplicações específicas utilizam componentes de hardware e software, e não somente hardware (circuitos integrados). Em contraponto, o uso de plataformas padrão de hardware e programas de aplicação requerem o uso de um hardware específico mais eficiente e/ou programas de aplicação específica.

São exemplos desses equipamentos os sistemas de instrumentação médica, controle de processos, controle para automóveis, sistemas de comunicação e redes, impressoras, utilitários domésticos, telefonia, telecomunicações, entre outros. Devido à crescente necessidade de produzir estes sistemas com complexidade cada vez maior e em um tempo menor, houve a necessidade de desenvolver metodologias para o projeto conjunto de sistemas integrados de hardware e software.

O volume de mercado para sistemas embutidos é alto. Comparando-se o volume de vendas de processadores nos Estados Unidos em 1995 com o volume de venda de microcontroladores, nota-se que este último – utilizado para implementar sistemas embutidos- foi superior. Isto sem considerar os microprocessadores também utilizados em aplicações embutidas. Em 1991, o volume de vendas de equipamentos para instrumentação médica foi de US$ 31 bilhões, enquanto que o volume de vendas de sistemas de computação foi de US$ 46,5 bilhões [CAM96], [JUL93].

Uma das formas de reduzir custos e o tempo de desenvolvimento de sistemas embutidos de Circuitos Integrados de Aplicação Especifica é a utilização de ferramentas para a implementação automática. Estas ferramentas podem reduzir custos de manutenção e atualização e reduzir o tempo de projeto.

Neste capítulo será apresentada a evolução dos sistemas de hardware/software, e uma revisão bibliográfica da síntese de alto nível. Também serão apresentados os algoritmos, ASAP , ALAP, *List-Scheduling* e *Scheduling Pipeline*, utilizados para a otimização do código gerado pelo sistema apresentado.

2.1 Histórico

Tradicionalmente, o hardware era considerado uma tecnologia imutável, com desempenho e custo de desenvolvimento altos, e usado somente em grande produção. Ao mesmo tempo, o software era considerado uma tecnologia com alto grau de adaptabilidade, de desempenho e custo de desenvolvimento baixos. Assim, ao se utilizar componentes de hardware e software, um sistema poderia se beneficiar do aumento de desempenho trazido pelo hardware, e ao mesmo tempo da redução de custos com a introdução de componentes de software.

Até há poucos anos, subsistemas de hardware eram implementados em placas de circuitos impressos com circuitos LSI *(Large Scale Integratio*n), ou com ASICs, dependendo da escala de produção. Devido a isso, os subsistemas de hardware possuiam baixa tolerância a mudanças, por serem confeccionados para executar uma só tarefa, possuindo alto desempenho.

Subsistemas de software, por outro lado, eram projetados para serem executados em um microprocessador. Seu desempenho era naturalmente baixo, visto que um microprocessador, para garantir a generalidade, necessitava de instruções de ALU (*Arithmetic and Logic Unit*), controle de fluxo, manipulação de memória, entrada/saída baseadas no paradigma de busca, decodificação, execução e escrita dos resultados. Qualquer função mais complexa poderia ser decomposta nas instruções de máquina do microprocessador. Entretanto, para ser modificado, bastava trocar o código do microprocessador, o que facilitava bastante a atualização desse componente.

Projeto de sistemas, contendo subsistemas de hardware e software tem sido feito na indústria e nas universidades desde os primórdios da computação (os primeiros computadores eram máquinas com partes em hardware e software). Entretanto, esses projetos eram feitos de maneira *ad hoc*, e a escolha dos componentes utilizados na implementação era feita utilizando-se na maior parte a experiência anterior do projetista, o que nem sempre levava a um sistema de custo/benefício/desempenho ótimo. Se o protótipo não possuísse desempenho ou custo adequado, o projeto voltava à prancheta para o reparticionamento do sistema nos seus subsistemas de hardware e software.

Os avanços em engenharia e computação, assim como o aumento na complexidade dos sistemas projetados, contribuíram para a análise e projeto integrados de sistemas de hardware e software. Primeiramente, o aumento do nível de integração permitiu que sistemas de computação inteiros fossem encapsulados em um único *chi*p. Tais sistemas eram programados por software, apesar do *chip* não ser mais reprogramável. A invenção de dispositivos programáveis, FPGAs – *Field Programmable Gate Arrays* [BYE84], permitiu que um dispositivo de hardware trabalhasse com o princípio de programa armazenado, isto é, durante a inicialização do dispositivo, a configuração do hardware é carregada a partir de uma memória.

2.2 Sistemas Embutidos

Sistemas embutidos apresentam algumas características que os diferenciam de outros sistemas integrados:

· Alta integração de módulos de hardware e software;

· São projetados para aplicações específicas;

· Possuem interface com o mundo exterior bem definida;

· Possuem restrições e requisitos fortes e bem definidos.

Estas características impõem a necessidade de uma metodologia que não somente as atenda, como também, atenda as demandas de caráter tecnológico como, por exemplo, tempo de projeto e custos financeiros.

Basicamente, quatro tarefas são identificadas durante o projeto de um sistema integrado de hardware e software:

- **Especificação e análise**: o sistema é descrito utilizando-se linguagens gráficas ou textuais nos diversos níveis de abstração, que são analisadas para obter diretrizes sobre quais componentes devem ser implementados em software e quais componentes devem ser implementados em hardware;

- **Particionamento:** o sistema é particionado em seus dois componentes de implementação – hardware ou software. Cada vez que há fluxo de dados entre os diversos componentes, haverá a necessidade de gerar uma interface entre o hardware e o software;
- **Síntese:** os componentes de hardware e software são implementados em suas primitivas básicas de implementação – portas lógicas para o hardware, linguagem de máquina para o software e uma interface são gerada para as diferentes interconexões entre componentes de hardware e software;
- **Validação:** antes de um protótipo do sistema ser confeccionado, um modelo virtual do sistema é construído e simulado em computador.

Diversas técnicas podem ser utilizadas para validar um sistema, variando entre verificação formal, simulação ou emulação.

2.3 Síntese de Alto Nível

A síntese automática é a transformação de uma especificação realizada em um determinado nível de abstração, para um nível mais próximo da realização física, através da aplicação de um conjunto de ferramentas que acrescentam detalhes estruturais e/ou geométricos à especificação inicial [WAG88].

A síntese de alto nível consiste em, a partir de uma especificação do comportamento requerido e de um conjunto de restrições e objetivos de um sistema digital, encontrar uma estrutura que implemente o comportamento especificado e satisfaça as restrições e objetivos.

A figura 2.1 [LAU92], apresenta um exemplo de uma estrutura em nível de transferência entre registradores (RTL - *Register Transfer Level*) que gera uma descrição comportamental. Na figura 2.1, os componentes RTL são apresentados com seu desenvolvimento em nível de portas de 1 bit. Por exemplo, a unidade de comparação que implementa a operação de comparação foi apresentada com uma porta AND com a entrada invertida dentro na caixa. O objetivo de otimização da síntese de alto nível pode ser o número de unidades funcionais ou o número de períodos de *relógio* necessários para executar o projeto, ou ainda, o período de *relógio* da implementação ou a combinação dos objetivos citados.

O circuito RTL é um conjunto de unidades funcionais interconectadas e elementos armazenados. Exemplos de unidades funcionais são somadores, comparadores e multiplexadores. Elementos armazenadores tipicamente incluem registradores para variáveis, e um registro para estados, onde se controla a Máquina de Estados Finita. Os componentes RTL são trocados por expansões em nível de portas lógicas, produzindo um circuito em nível de portas que é subseqüentemente submetido para a fase de síntese lógica.

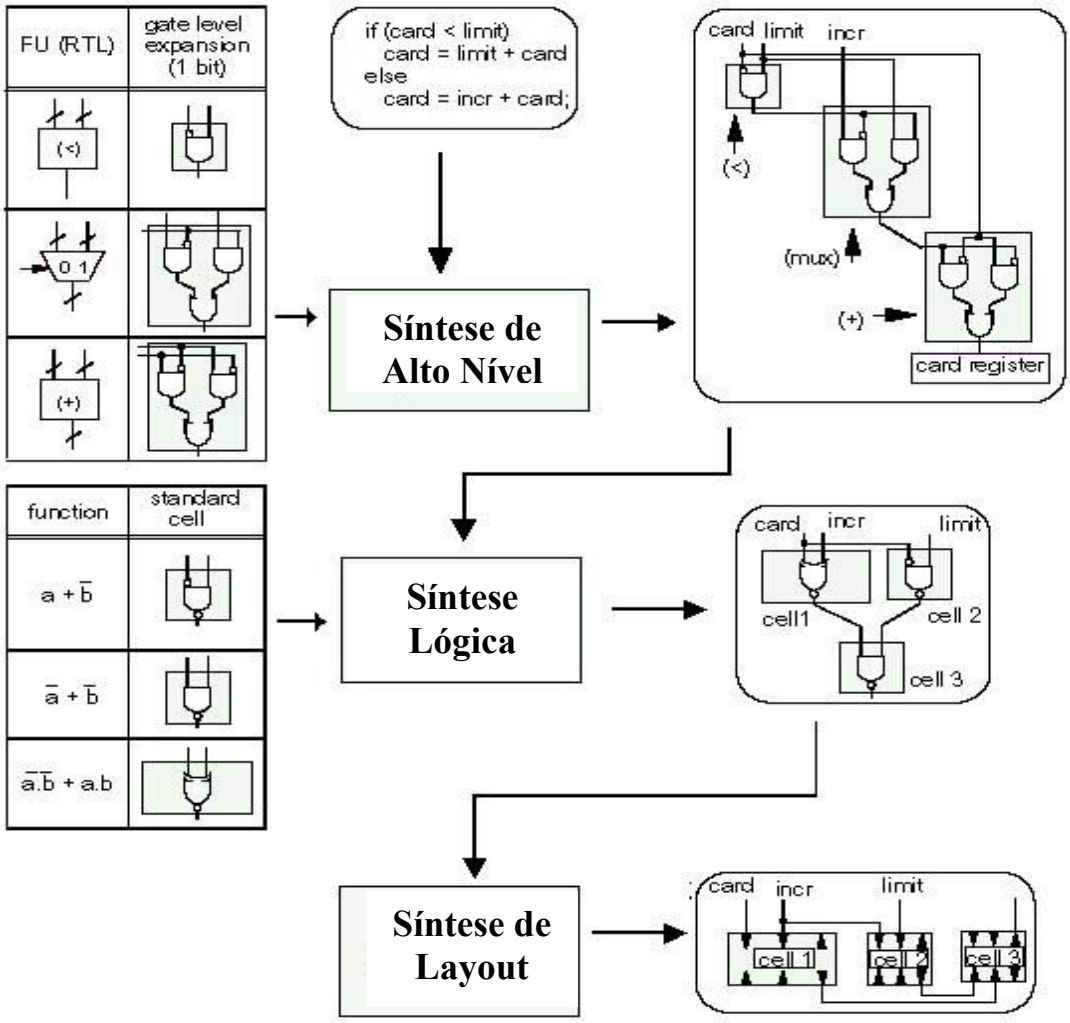


FIGURA 2.1 – Fluxo do Projeto de Síntese

2.4 Áreas de Aplicação da Síntese de Alto Nível

ASICs são utilizados em muitas aplicações nas mais diversas áreas, como computadores, sistemas eletrônicos, multimídia e telecomunicação. Essas áreas de aplicação possuem diferentes características, desde o projeto até a síntese. Uma aproximação pode ser possível para desenvolver um sistema de síntese de alto nível geral, podendo direcionar as características divergentes e realizar domínios de aplicação diferentes. Entretanto, é muito difícil e trabalhoso conseguir direcionar divergências num único sistema. Portanto, a tendência é desenvolver ferramentas de síntese de alto nível para domínios diferentes.

Um domínio em que as ferramentas de alto nível tem tido muito sucesso é na área de DSP (Processamento Digital de Sinais), [HAD91], [WAL91], [RAB91], [CLA88], [MAN87],[MIT93],[TSE86]. Os projetos em DSP podem ser caracterizados pelo domínio de operações aritméticas. Um exemplo típico de um projeto para DSP é mostrado na figura 2.2, que ilustra a descrição comportamental de um filtro ondular

elíptico de sexta ordem, possuindo muitas operações de adição, subtração e multiplicação. O gráfico de fluxo de dados é mostrado na figura 2.3, onde os arcos apresentam o fluxo dos dados.

```
Architecture behavior of digital_filter is
Begin
        Process
                type coef_arr is array (integer range 1 to 10) of integer;
        variable c: coef_ar := (0,1,2,3,4,5,6,7,8,9); -- coefficient array
        variable s1, s2, s3, s4,s5,s6: integer := 0;   --state variables
        variable x0, x1,x2,x3,x4,x5,x6 : integer;  -- temporary variables
                variable v1, v2: integer ;
                begin
                        done <='0';  -- ready to read input, wait until is ready
                        wait until go ='1';
                        x0 := input;   x1 := s1;    x2 := s2;   x3 := s3;
                        x4 := s4;        x5 := s5;    x6 := s6;
                        v1 := x1 *x2 –(x0-x1) *c(7);
                        v2 := x5 * c(5) + x6 – x4;
                        s1 := v1 * c(9)+v2*c(9)+x1;
                        s2 := x2 * x5 * c(4)-x5 * c(2);
                        s3 := x4 * c(8) + x3;
                        s4 := (x1-x3-x5) * c(6)*x4;
                        s5 := v1 * c(7)+v2 * c(10) + x5;
                        s6 := x1 * c(3)-(x5*c(1)+x6);
                        output <= s5;    done <= '1';
                        wait until go ='0';
        end process;
end behavior.
```

FIGURA 2.2 - Descrição Comportamental do Filtro Ondular Elíptico de Sexta Ordem

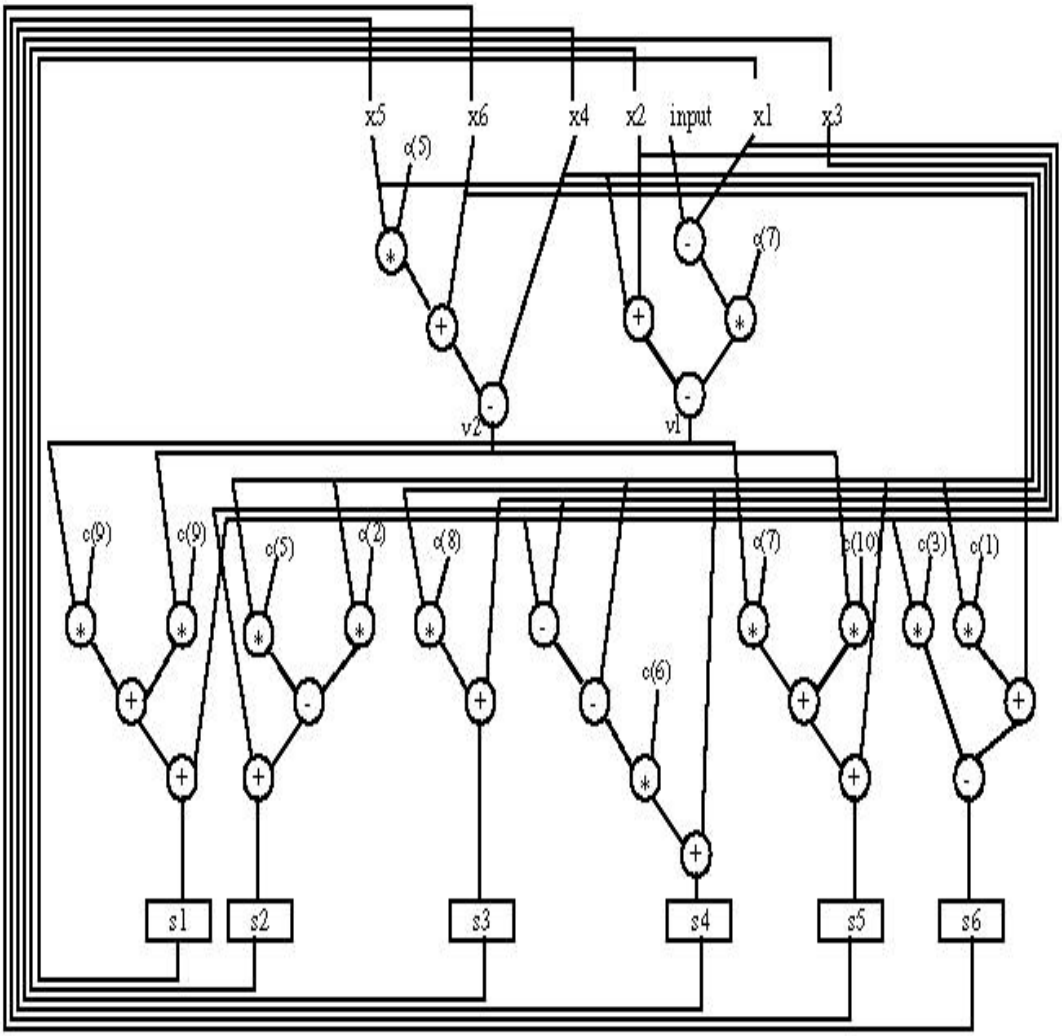


FIGURA 2.3 - Gráfico do Fluxo de Dados Filtro Ondular Elíptico de Sexta Ordem

Define-se projeto de síntese como um processo de tradução de uma descrição comportamental para uma descrição estrutural. Para definir e diferenciar tipos de síntese, pode-se usar o diagrama-Y, figura 2.4 [GAJ92]. Os eixos no diagrama-Y representam três diferentes domínios de descrição: comportamental, estrutural e físico. Ao longo de cada eixo estão diferentes níveis da descrição do domínio. Quanto mais longe do centro do Y, o nível de descrição é mais abstrato. Pode-se representar ferramentas de projetos como arcos que estão juntos com um domínio ou entre os eixos para ilustrar qual informação é usada por cada ferramenta e qual informação é gerada pelo uso das ferramentas de projeto.

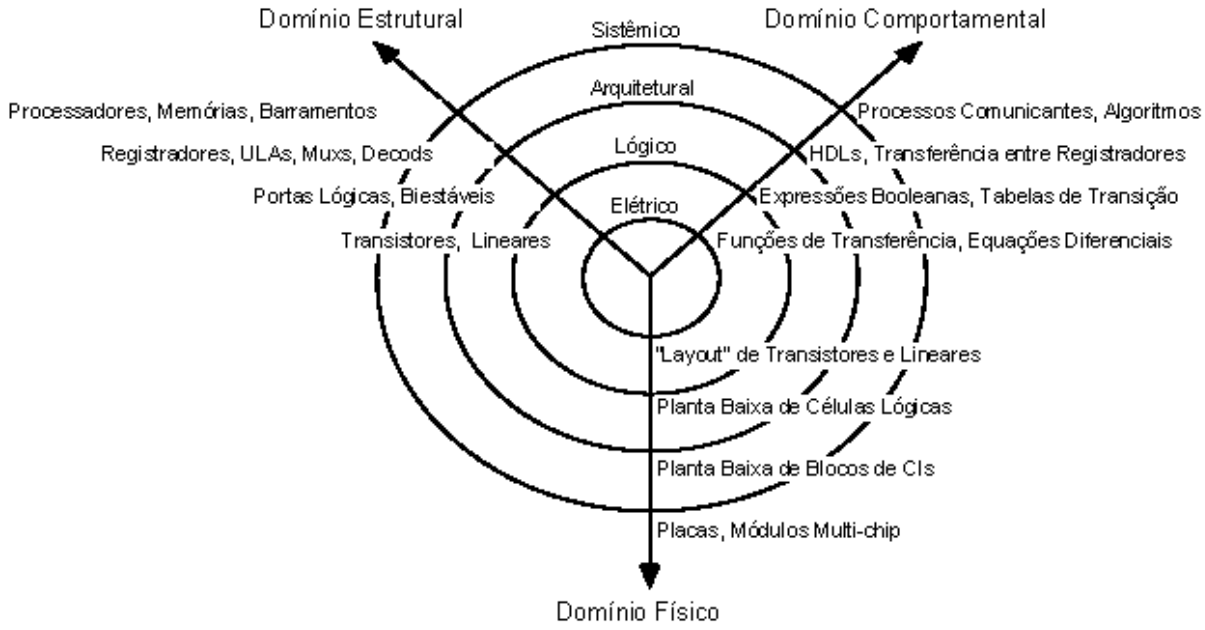


FIGURA 2.4 - Diagrama de Gajski-Kuhn ou Diagrama Y

Pode-se estender esse gráfico desenhando círculos concêntricos no diagrama Y. Cada círculo intersecta o eixo Y para um nível particular de representação com um domínio. O círculo representa toda a informação conhecida sobre o projeto para algum ponto no tempo. A parte externa do ciclo é o nível de sistema; o próximo, é a microarquitetura ou nível de transferência de registradores, formado pelos níveis lógicos e nível do circuito. A tabela 2.1 lista esses níveis [GAJ92].

TABELA 2.1 - Projeto de objetos em diferentes níveis de abstração

Nome do Nível	**Representação Comportamental**	**Representação Estrutural**	**Representação Física**
Nível Sistema	Especificação gráfico Fluxo Gráfico Algoritmos	Processador, Controlador Memórias, Barramentos	Placas *Chips*
Nível Microarquitetural	Transferência de registradores	ALUs , MUXs Multiplicadores Registradores, Memórias	*Chips*
Nível Lógico	Equações Booleana	Portas Flip-flops	Módulos Células
Nível do Circuito	Funções de transferência	Concepção de transistores	Layouts transistores Contatos

Fonte: Gajski – High- Level Synthesis pag. 5

No domínio comportamental, o interesse está em saber o que o projeto faz, e não como ele é feito. O projeto é considerado como uma ou mais caixas-pretas, com um conjunto específico de entradas e saídas e um conjunto de funções, descrevendo o comportamento de cada saída em tempo da entrada sobre tempo. Uma descrição comportamental inclui uma descrição da interface e das constantes impostas no projeto. A descrição da interface específica das portas I/O e relação de tempo ou protocolos entre sinais dessas portas.

Para descrever comportamento, usam-se funções de transferência e diagrama de tempo no nível do circuito, e expressões booleanas e diagrama estado no nível lógico. No nível RT, tempo é dividido para intervalos chamados de estados ou passos de controle. Usa-se uma descrição transferência-registradores, que especifica para cada estado de controle a condição para ser testada, toda transferência de registradores para ser executada, e o próximo estado de controle para ser listado. No nível do sistema, usa-se variáveis e operadores de linguagem para expressar funcionalidade dos componentes do sistema. [GAJ92].

Uma representação estrutural liga a representação comportamental e física. Ela é um mapeamento de um para muitos de uma representação comportamental, para um conjunto de componentes e conexões sob obstáculos, como: custo, área e atraso. Em tempos, uma representação estrutural, tais como um esquemático lógico ou de um circuito, pode servir como uma descrição funcional. Por outro lado, descrição comportamental, tais como, expressões Booleanas sugerem uma implementação trivial, como estrutura de soma de produtos constituídos de portas NOT, AND e OR. O nível mais comumente usado da representação estrutural é identificado em termos de elementos estruturais básicos. No nível do circuito, elementos básicos são transistores, resistores e capacitores, enquanto portas e flip-flops são elementos básicos no nível lógico. ALUs, multiplicadores, registradores, RAMs e ROMs são usados para identificar transferência de registrador. Processadores, memórias e barramento são usados no nível do sistema [GAJ92].

A representação física ignora, tanto quanto possível, o que o projeto está suposto para fazer e projetar sua estrutura no espaço ou para silício. O nível mais comumente usado na representação física são polígonos, células, módulos, módulos multi-chip e placas de circuito.

2.5 Definição de Síntese

Síntese é definida como uma tradução de uma descrição comportamental para uma descrição com mais informações, similar para a compilação de uma linguagem de programação tal como Java, C ou Pascal para uma linguagem Assembler, que não é estrutural, mas agrega informações. Cada componente na descrição estrutural é definido pela própria descrição comportamental. A estrutura de componentes pode ser obtida através da síntese num baixo nível de abstração. Síntese, algumas vezes chamada de refinamento de projeto, acrescenta um nível adicional de detalhamento para fornecer informações necessárias para o próximo nível de síntese ou para produção do projeto.

Cada passo de síntese pode diferenciar-se do outro, na quantidade de informações entre a descrição comportamental e a descrição estrutural sintetizada. Por exemplo, uma soma de produtos de expressões Booleanas pode ser trivialmente convertida para um segundo nível, usando na implementação portas AND e OR com um número ilimitado de entradas. Substancialmente mais trabalho é necessário para implementar a mesma expressão com portas NAND com somente duas entradas. Para cada descrição comportamental, podem ser sugeridas algumas implementações triviais. Geralmente, é impossível encontrar uma implementação ótima possuindo uma biblioteca com recursos mínimos.

2.6 Principais Tarefas da Síntese de Alto Nível

As etapas da Síntese de Alto Nível englobam uma série de tarefas conjuntas a fim de obter o melhor resultado.

2.6.1 Representação do sistema e compilação

O comportamento do sistema para ser sintetizado é geralmente especificado em nível de algoritmo, podendo-se usar linguagens de programação como C, Java e Pascal, ou uma linguagem de descrição de hardware, tais como VHDL e Verilog.

VHDL [ASH90], [NAV92] é uma linguagem padrão, desenvolvida pelo Departamento de Defesa Americano para descrever sistemas eletrônicos digitais. VHDL suporta o projeto, a descrição e a simulação de estruturas de hardware em diferentes níveis de abstração, do arquitetural ao lógico. Esta linguagem foi projetada para ser independente de qualquer tecnologia específica, ambiente de projeto, ou métodos de projeto, e, consequentemente, possibilitar a integração de qualquer combinação de ambiente, tecnologia, e metodologia.

Quando a especificação do comportamento do sistema estiver pronta, necessita-se compilar a mesma para uma representação interna, sendo geralmente grafos de fluxos de dados ou grafos de fluxo de controle. Compilação em síntese de alto nível tem muitas coisas em comum com uma compilação não otimizada de uma linguagem de programação. Cada especificação comportamental é transformada para uma representação de gramática única. O grafo de fluxo de dados é um grafo direcionado que representa como os dados estão se movendo, enquanto o grafo de fluxo de controle é um grafo direcionado que indica a seqüência das operações no tempo.

A representação de dados usada por muitas aproximações de síntese comportamental é completamente diferente em estilo e estrutura, mas em geral, informações de fluxo de dados e controle são armazenadas em uma ou duas partes ou em gráficos hierárquicos.

2.6.2 Transformação de Alto Nível

Desde a especificação são desejáveis algumas otimizações iniciais para a representação dos dados. A compilação da descrição do sistema digital para uma representação interna adequada ao processo de síntese. As principais atividades são:

- Otimização da representação interna, usando-se transformações comportamentais, como as comumentes empregadas na otimização de compilação das linguagens de programação convencionais, por exemplo, remoção de constantes de laços;
- Escalonamento das operações, ou seja, atribuir cada operação a um passo de controle;
- Alocação de componentes de hardware para a realização das operações e armazenamento de variáveis;

- Mapeamento de cada operação específica para os componentes alocados e mapeamento das variáveis aos elementos de armazenamento;
- Geração de um controlador, seja na forma de uma máquina de estados ou de um microprograma.

Resumindo, pode-se dividir a síntese de alto nível em duas etapas, a primeira etapa é responsável por definir a posição das operações do grafo de seqüenciamento em suas dimensões: tempo e espaço. Nesta etapa, é determinado o intervalo de tempo disponível para execução de cada operação, a relação entre estas e os recursos funcionais que realizarão as operações. O domínio temporal do posicionamento das operações é denominado escalonamento, e o domínio espacial, alocação.

A segunda etapa é a fase da definição das interconexões detalhadas da parte operativa e especificação da unidade de controle. Ou seja, uma vez conhecidas as relações entre as unidades funcionais e os instantes para execução, a síntese de alto nível organiza o grafo de seqüenciamento dentro da estrutura básica da arquitetura-alvo da síntese.

O escalonamento é a tarefa que trata o domínio temporal do posicionamento das operações. No escalonamento é determinado o tempo de início de cada operação, conforme a precedência e as restrições constantes do grafo. Basenado-se na estrutura do grafo, nas prioridades das operações e nas restrições impostas na especificação do projeto, pode-se definir o algoritmo de ordenação topológica.

As restrições influenciam consideravelmente a implementação do problema e o método empregado na sua solução. São definidas nos domínios temporal e espacial, sendo denominadas restrições temporais e de recursos. Os projetos também podem possuir restrições temporais e de recursos ao mesmo tempo.

A alocação é a tarefa que trata do domínio espacial da aplicação. O alocador estabelece a relação entre as operações e recursos, ou seja, dado o exemplo de uma operação de adição, ela pode ser alocada a um recurso como ALU, que inclua a função de somador, ou um somador que funcione isoladamente de outras unidades.

2.7 Algoritmos de Síntese de Alto Nível

Geralmente, tipos de algoritmos usados em síntese de alto nível são particionamento, escalonamento e distribuição. Algoritmos de particionamento são utilizados para o mapeamento de uma descrição comportamental para estruturas, reduzindo o tamanho do problema ou para satisfazer algum objetivo, tais como, tamanho do chip, número de pinos, dissipação de potência ou comprimento máximo da conexão.

Os algoritmos utilizados em síntese de alto nível partem do princípio da ordenação topológica. Tendo como estrutura básica esta ordenação, distingüem-se pelas características de restrição do problema a que são destinados. Os algoritmos podem aceitar ou não restrições de recursos e também permitir restrições temporais ou não.

O problema básico é, portanto, dado um certo algoritmo a ser implementado em hardware, qual o número mínimo de ciclos de relógio e qual o número mínimo de componentes necessários para executar o algoritmo na maior velocidade possível. Para

maior velocidade, precisa-se de um mínimo de passos, mas para executar um algoritmo num mínimo de passos, maior quantidade de hardware deve estar disponível, e portanto maior o custo envolvido.

2.7.1 List-Scheduling

Existem algoritmos já consagrados que permitem descobrir o menor número de passos necessários, dada uma restrição de recurso [GAJ92]. Na figura 2.5 apresenta-se um pseudo-código, o algoritmo ASAP (A*s Soon As Possible*), através do qual distribuem-se as operações em diversos passos de relógio. Cada operação é agendada para o primeiro passo de controle possível, em que existam recursos suficientes para tal.

```
// N é o conjunto de nós, E o conjunto de estados
// Predecessores_ni é o conjunto dos predecessores do nó ni
// Inicialização
Para cada nó ni ∈ N faça
     Se Predecessores_ni = ϕ então
          Ei = 1;
          N = N – {ni};
    Senão
          Ei = 0;
 Fim para;
//Operação
Enquanto N ≠ ϕ faça
     Para cada nó ni ∈ N faça
          Se todos_os_nós_agendados(Predecessores_ni, E) então
               Ei = MAX(Predecessores_ni, E)+1;
               N = N – {ni};
          Fim_se
     Fim_para
Fim_enquanto.
```

FIGURA 2.5 - Algoritmo ASAP

Enquanto o algoritmo ASAP tenta colocar cada operação no primeiro momento em que existam operadores disponíveis, o algoritmo ALAP (*As Late As Possible*) coloca cada operação o mais próximo possível do último passo de controle. A figura 2.6 apresenta o algoritmo ALAP.

```
// N é o conjunto de nós, E o conjunto de estados
// Sucessores_ni é o conjunto dos sucessores do nó ni
// Tem-se T passos
// Inicialização
Para cada nó ni ∈ N faça
     Se Sucessores_ni = ϕ então
          Ei = T;
          N = N – {ni};
    Senão
          Ei = 0;
 Fim para;
//Operação
Enquanto N ≠ ϕ faça
```

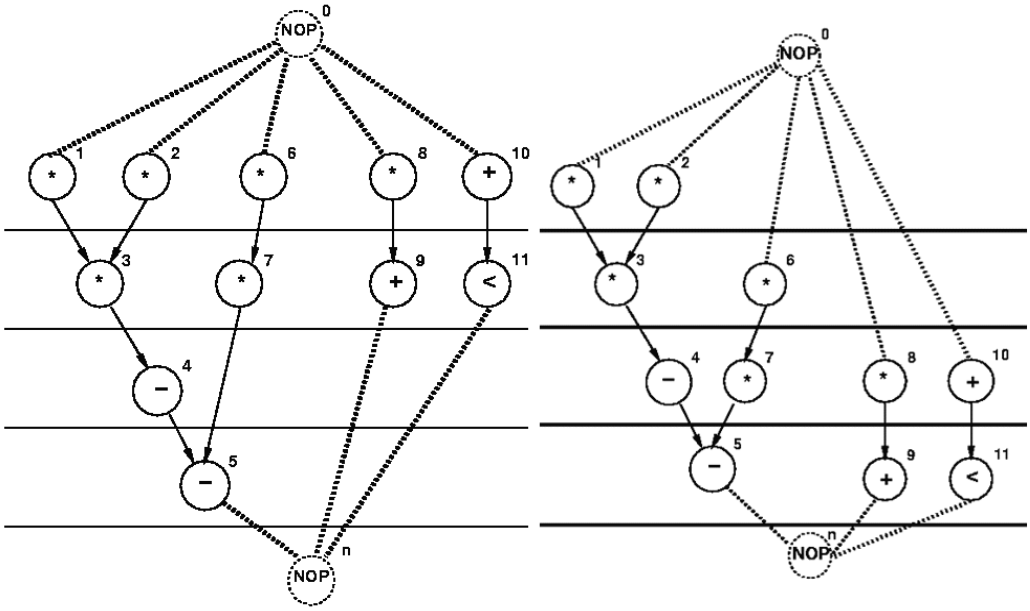
```
        Para cada nó ni ∈ N faça
            Se todos_os_nós_agendados(Sucessores_ni, E) então
                Ei = MIN(Sucessores_ni, E)-1;
                N = N – {ni};
            Fim_se
        Fim_para
    Fim_enquanto.
```

FIGURA 2.6 - Algoritmo ALAP

```
While (x<a) do
    x1 = x + dx;
    u1=u-(3*x*u*dx) – (3 * y *  dx);
    y1 = y + (u*dx);
    x = x1;
    u = u1;
    y = y1;
endwhile
```

FIGURA 2.7 - Seqüência de operações

A figura 2.8 exibe o escalonamento das operações apresentadas na figura 2.7 segundo os algoritmos ASAP e ALAP. No algoritmo ASAP, observa-se um grande desperdício de recursos, já que os quatro multiplicadores estarão ativos apenas uma pequena parte do tempo. No algoritmos ALAP, o resultado é significativamente melhor que o algoritmo ASAP, já que existe uma melhor distribuição de operações ao longo do tempo total de execução.



(a) **(b)**

FIGURA 2.8 - Exemplo dos Algoritmos ASAP (a) e ALAP (b)

Pode-se achar exemplos onde o algoritmo ASAP teria um melhor desempenho que o algoritmo ALAP [GAJ92]. De qualquer modo, ambos os algoritmos fornecem o número mínimo de ciclos, para uma dada restrição devido à dependência de dados.

Um resultado melhor pode ser obtido pelo uso do algoritmo de ordenação por listas, ou *List-Schedulig* ([GAJ 92],[MIC 92], [DEM 94]). Neste caso, mantém-se uma lista de prioridades dos nós já prontos, que são aqueles cujos predecessores já foram colocados em algum passo de controle. A idéia básica é ordenar todas as operações usando a ordem ALAP em ordem ascendente como chave primária, e a chave ASAP em ordem descendente como chave secundária. Quando ambas as chaves têm o mesmo valor, qualquer dos nós é escolhido. Por exemplo, a tabela 2.2 apresenta as chaves ALAP e ASAP, ordenadas conforme acima descrito. Os nós n11, n8 e n9 possuem ALAP 1, pois foram agendados para o último passo de controle quando o algoritmo ALAP foi executado. O nó n11 tem ASAP maior, e portanto este nó é escolhido para ter a menor prioridade. Os nós n7 e n9 têm o mesmo índice ASAP, e portanto qualquer prioridade é factível e, no caso, escolheu-se o nó n7. Prossegue-se assim até a montagem completa da tabela 2.3.

TABELA 2.2 - Projeto de objetos em diferentes níveis de abstração

Nó	*11*	*7*	*9*	*10*	*8*
ALAP	1	1	1	2	2
ASAP	4	2	2	3	2
Prioridade	1	2	3	4	5

Fonte: Gajski – High- Level Synthesis pag. 238

TABELA 2.3 - Montagem das prioridades para ordenação por listas

Nó	*11*	*7*	*9*	*10*	*8*	*3*	*5*	*6*	*4*	*1*	*2*
ALAP	1	1	1	2	2	2	2	3	3	4	4
ASAP	4	2	2	3	2	1	1	2	1	1	1
Prioridade	1	2	3	4	5	6	7	8	9	10	11

Uma vez definida a ordem de prioridades, basta especificar quantos operadores estão disponíveis para uso. Inicia-se pela maior prioridade, e cada vez que os recursos estão esgotados, desloca-se a operação de um certo nó para o passo seguinte. O resultado do algoritmo para o exemplo da figura 2.7 encontra-se na tabela 2.4, assumindo-se que dois multiplicadores estejam disponíveis, além de um subtrator, um comparador e um somador.

TABELA 2.4 - Ordenação por lista

Estado	*Operações dos nós*
e1	n1, n2, n5 (2 multiplicadores, 1 somador)
e2	n6, n4, n9 (2 multiplicadores, 1 comparador)
e3	n8, n3, n10 (2 multiplicadores, 1 subtrator)
e4	n11, n7 (1 subtrator, 1 somador)

2.7.2 Scheduling-Pipeline

Existem algoritmos já consagrados que permitem agendar tarefas em processamento paralelo Pipeline. *Maximal schedule* e *Feasible Schedule* [MIC92a] são algoritmos utilizados para melhorar a performance de execução das tarefas. O escalonamento é realizado a partir de um grafo que representa o sistema proposto.

A idéia básica é aplicar os algoritmos *Maximal Schedule* e *Feasible Schedule,* agendando os nós em seus devidos tempos. Este processo visa uma economia de processamento paralelo, pois implica em um maior aproveitamento do tempo na execução das tarefas.

O algoritmo Forward Urgency verifica o maior caminho de um nó até a sua saída. Para realizar o algoritmo Forward Urgency, segue-se os seguintes passos: antes de agendar os nós do grafo, para serem executados em um determinado passo, é necessário ordená-los de forma decrescente de urgência em uma lista. Ordenar os grafos em ordem decrescente de urgência consiste em percorrer o grafo, e para cada nó do grafo, verificar o maior caminho. O maior caminho vai ser definido pela distância que o nó se encontra do nó terminal do grafo (nó de saída, último nó), e também vai depender do tempo de execução (atraso) do recurso que o nó representa. A figura 2.9 apresenta o fluxograma do algoritmo.

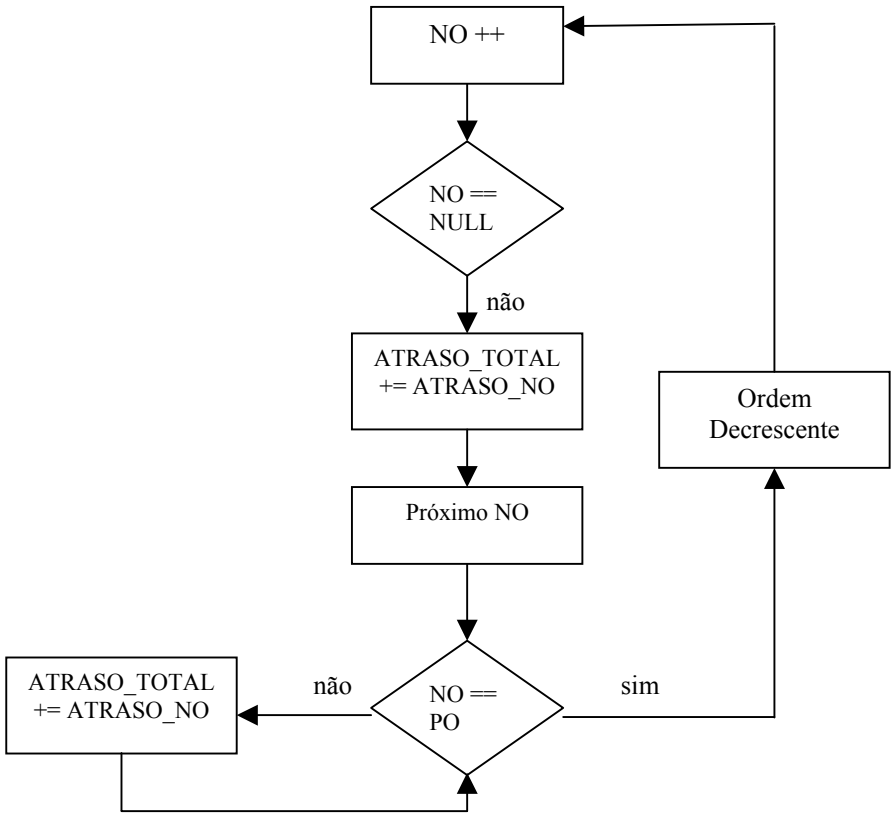


FIGURA 2.9 - Fluxograma do algoritmo Forward Urgency

O algoritmo *Maximal Schedule* segue a partir da operação mais urgente e mantém dependência de dados. Muitas operações são possíveis em cada passo, contanto que o caminho mais longo de atraso em um passo não exceda o máximo estágio por ciclo. A busca no grafo segue verificando se um nó está em série ou em

paralelo com seu nó vizinho, como resultado no *Maximal Scheduling,* cada nó é agendado para o menor passo de tempo que ele pode ser agendado. A figura 2.10 apresenta o fluxograma do método *Maximal Schedule.*

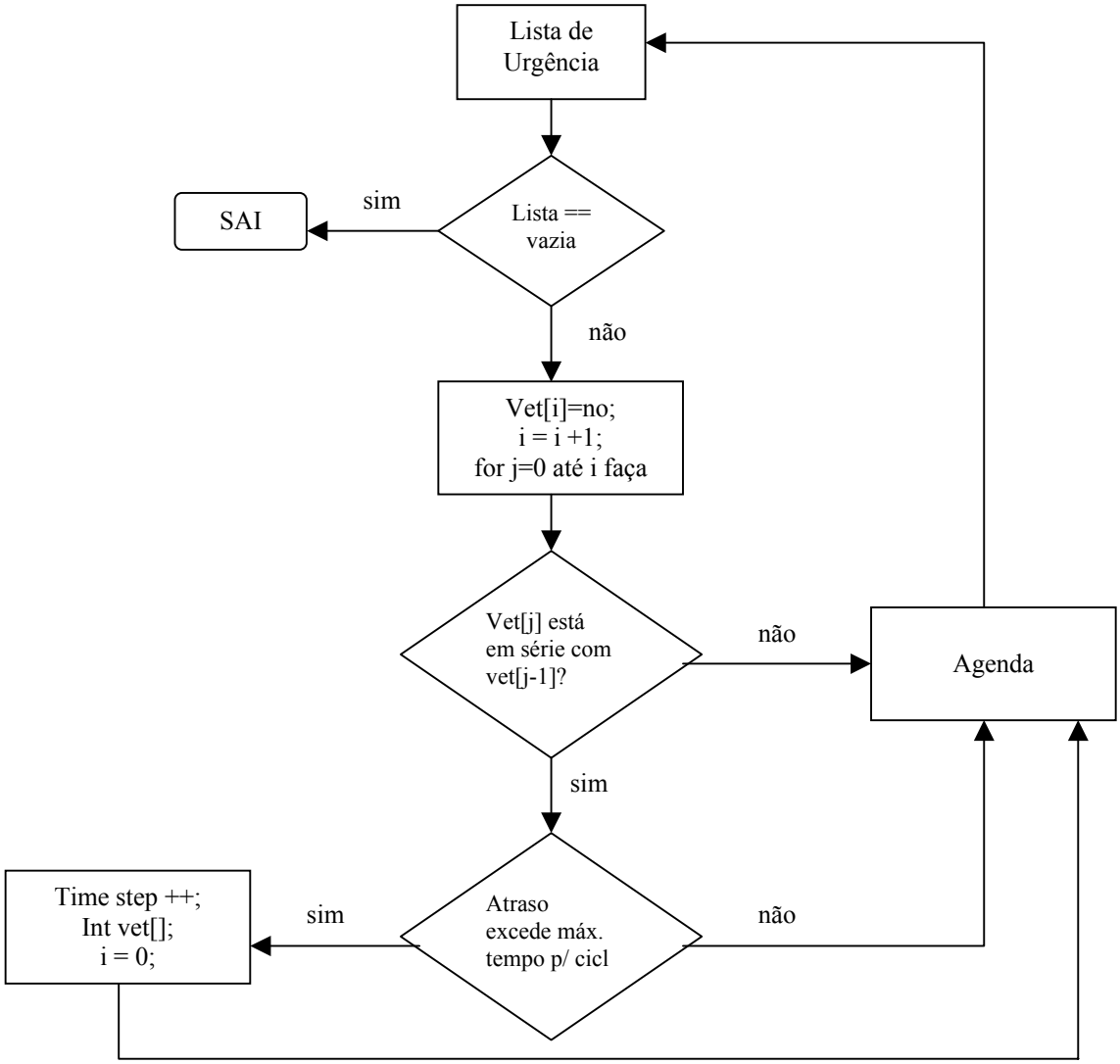


FIGURA 2.10 - Fluxograma do método Maximal Schedule

O algoritmo Feasible Schedule é o principal algoritmo de Scheduling Pipeline. Este algoritmo faz a alocação de recursos no mesmo tempo e em ordem para alcançar a máxima performance. Para este escalonamento é dada uma latência fixa e um tempo máximo de execução de um ciclo de relógio tendo que ser determinada também, a quantidade de recursos disponíveis por ciclo. Na Figura 2.11, pode ser visualizado o fluxograma do algoritmo *Feasible Schedule*.

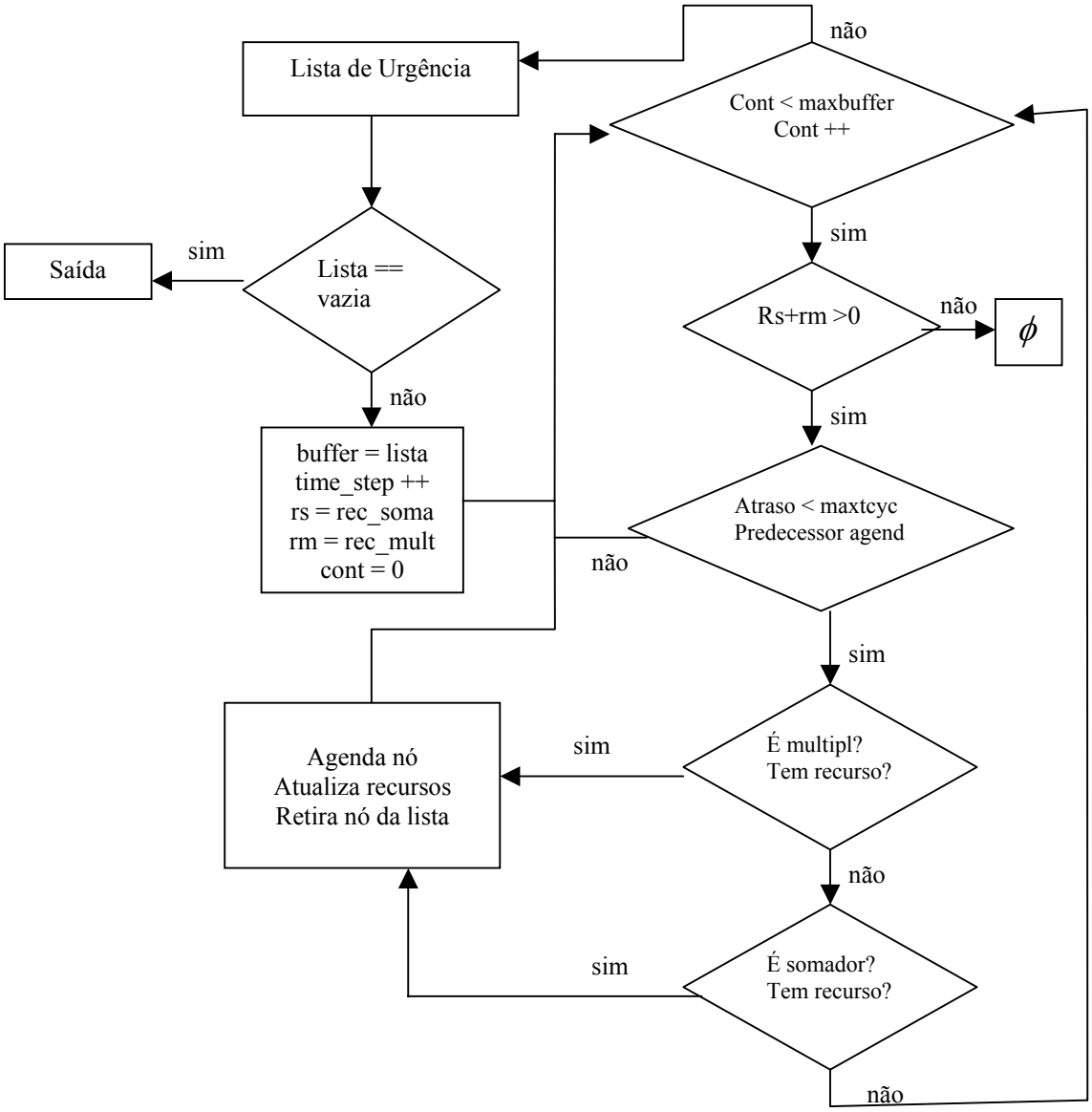


FIGURA 2.11 - Fluxograma do método Feasible schedule

2.8 Alguns sistemas de síntese

Determinando uma especificação funcional ou a performance de um projeto, o projetista poderia criar uma implementação que execute a função e satisfazer as características de performance. O projetista também pode refinar o projeto, e verificar se ele satisfaz as especificações funcionais e de performance.

Com o advento de tecnologias VLSI, tornou-se possível empacotar milhões de transistores em um único chip. Esse resultado trouxe um aumento na complexidade do projeto, e aumentou o tempo de elaboração do projeto. Com isso, houve a necessidade de melhorar as metodologias de projetos e sistemas automatizados, que são geralmente conhecidos como CAD.

O projetista possui um conjunto de ferramentas para capturar, verificar e avaliar o seu projeto. O projetista tende a realizar o projeto de forma bottom-up e muitos projetistas tendem a projetar primeiro a construção de blocos, e posteriormente

aproveitar como componentes para realizar as estruturas de alto nível. O resultado são projetos com alta qualidade. Por outro lado, projetistas humanos são lentos, e realizam projetos com muitos erros. Além disso, projetistas criativos otimizam projetos por quebrar regras antigas e introduzir novas regras; através disso, criando uma demanda para novas ferramentas de verificação e linguagens de descrição de projetos.

Devido à limitação do ambiente centralizado no projetista, pesquisadores procuram estabelecer um método de automação de projetos. Neste método, acredita-se que o processo de síntese pode ser automatizado em vários níveis, assegurando o comportamento apropriado para o projeto, e reduzindo o tempo de concepção do mesmo. A seguir algumas ferramentas de Síntese de Alto Nível analisadas são apresentadas.

2.8.1 MIMOLA Synthesis System

O sistema MIMOLA [MAR86] compreende uma série de ferramentas para descrever, simular e sintetizar circuitos digitais. A entrada para o sistema é uma especificação do circuito dada na linguagem MIMOLA *(Machine Independent MicroprOgramming Language*) [MAR 86]. A partir dessa especificação, o sistema pode gerar uma série de soluções alternativas, todas elas formadas por uma parte operativa com componentes RT, e uma parte de controle, representada por um microprograma. Cada alternativa explora em diferentes graus o paralelismo que pode ser extraído da descrição. Levando em consideração fatores como custo e desempenho, o usuário controla a quantidade de paralelismo desejada para o seu circuito. O sistema MIMOLA tem como pontos fracos tanto a sua linguagem de entrada, difícil de usar e fora de padrão, como a forma de interação com o usuário durante o processo de síntese.

2.8.2 System Architect's Workbench

Esta ferramenta é o resultado da evolução dos trabalhos clássicos da Universidade de Carnegie Mellon, em síntese automática e linguagens de descrição de hardware. Como um sistema típico de síntese de alto nível, *Workbench* converte uma descrição algorítmica de um sistema digital em um conjunto de componentes ao nível de transferência entre registradores, cujo controle é especificado por uma tabela de transição de estados [THO90].

As ferramentas de síntese automática, desenvolvidas ao longo do tempo em Carnegie Mellon, englobam uma quantidade razoável de algoritmos, estratégias e procedimentos, o que permite uma vasta experimentação e um grande acúmulo de experiência na área. Particularmente, este sistema incorpora duas estratégias: uma específica, considerando o estilo da descrição (basicamente um mapeamento para uma arquitetura alvo); e outra genérica, que inclui as tarefas convencionais de escalonamento e alocação. A estratégia convencional permite um projeto mais rápido. A estratégia permite uma melhor exploração do espaço de projeto, além de atender a uma gama mais variada de estilos e opções.

Uma descrição de entrada é compilada para uma estrutura de dados representada por um grafo acíclico. As *procedures* e blocos são mapeados em subgrafos; as operações em nodos, e as variáveis e sinais em arcos. Nodos especiais são usados para

representar operações de desvio condicional e decodificação. A partir dessa representação em grafos, inicia-se ou a estratégia específica, conduzida por um único programa, que combina algoritmos e regras e gera apenas microprocessadores, ou a estratégia genérica.

A estratégia genérica é conduzida por uma série de procedimentos seqüenciais. O primeiro, realiza transformações comportamentais sobre o grafo, na tentativa de tornar o projeto tanto quanto possível independente do estilo pessoal de descrição do projetista. Esse procedimento permite também interferência do projetista, durante as transformações, com o objetivo de explorar o espaço do projeto. O segundo procedimento, escalona as operações em passos de controle, obedecendo a restrições de interface fornecidas pelo projetista. A seguir é realizada a síntese do fluxo de dados, e é escolhida a estrutura de barramentos adequada ao projeto.

O particionamento de operações em grupos lógicos é realizado durante o escalonamento e a síntese do fluxo de dados com o auxílio de um procedimento de particionamento arquitetural. Programas especiais auxiliam na visualização do processo de síntese e no controle de correlação entre a estrutura gerada e a descrição inicial. Este sistema produziu experimentalmente resultados comparáveis aos melhores projetos manuais [THO90].

2.8.3 High Level Synthesis IBM System – HIS

HIS é um sistema de síntese de alto nível, que utiliza um subconjunto de VHDL como linguagem de entrada. [CAM91]. O objetivo de HIS é gerar automaticamente circuitos digitais síncronos de alta complexidade, otimizados e em curto espaço de tempo, a partir de descrições de alto nível em VHDL. Uma descrição inicial é compilada para uma estrutura de dados representada por um grafo cíclico. Esse grafo representa somente o comportamento do circuito descrito, e contém informações separadas sobre o fluxo de dados e de controle. A síntese de alto nível adiciona uma estrutura ao grafo: o escalonamento consiste basicamente de uma síntese de controle, e adiciona uma Máquina de Estados Finita ao grafo. A alocação consiste na síntese do fluxo de dados, e adiciona uma parte operativa ao grafo.

Escalonamento e alocação seguem aproximadamente os algoritmos clássicos na área com pequenas alterações. A análise do fluxo de dados para determinar o tempo de vida das variáveis é global. O escalonamento é realizado usando o algoritmo *As-Fast-As-Possible* [CAM 90b], que enfatiza redução do número de passos de controle, explorando desvios condicionais, e não incremento de paralelismo como no escalonamento *Force-Directed*.

2.8.4 Olympus

Olympus [MIC90] é um sistema de projeto automatizado que combina ferramentas de síntese de alto nível (comportamental e estrutural), síntese lógica e validação, perfeitamente integradas. Foi desenvolvido na Universidade de Stanford com o objetivo de facilitar o projeto de circuitos ASIC de alta complexidade em um ambiente que permite, tanto projetos auxiliados por computador como projetos

totalmente automáticos. A linguagem de entrada do sistema é HardwareC, que fornece como recursos a descrição de processos, e modela a comunicação entre processos, tanto por troca de mensagens como por passagem de parâmetros.

A ferramenta Hercules realiza a síntese comportamental, que em Olympus, agrupa todas as transformações similares às encontradas para otimização em compiladores convencionais (desenrolamento de loops, propagação de constantes e variáveis, eliminação de subexpressões comuns, eliminação do código morto, substituição de desvios condicionais por lógica combinacional quando possível). A ferramenta Hebe é responsável pela síntese estrutural, que em Olympus compreende alocação e escalonamento. Ao contrário da maioria dos outros sistemas, que para a alocação buscam componentes em uma biblioteca, em Olympus os componentes alocados são posteriormente sintetizados no nível lógico. Também diferente da maioria dos demais sistemas, em Olympus, primeiro é realizada a alocação, e só então o escalonamento, tentando dessa forma alcançar um escalonamento acurado, pois os retardos dos componentes alocados já são conhecidos.

Olympus se completa com as ferramentas Ariadne para simulação comportamental, Theseus para visualização de forma de onda, Mercury para realizar a interface com procedimentos de síntese e simulação lógica e Ceres, que realiza o mapeamento tecnológico. Olympus é completamente operacional e já foi usado para a síntese de circuitos complexos.

2.8.5 Outros Sistemas de Síntese de Alto Nível

Em [NET99] foi apresentado um sistema de síntese a partir de uma descrição Java e o comportamento do código gerado é sempre baseado em Máquina de Estados Finita. Já em [GIR99], é permitida a abstração em Máquina de Estados Finita e dataflow.

O sistema apresentado em [BER 93], usa VHDL como linguagem de entrada, incluindo especificações comportamentais. Suporta vários níveis de descrição, mas o projetista deve ter a flexibilidade para escolher o nível mais apropriado para cada parte do projeto. A saída da síntese de alto nível é uma descrição VHDL em nível RT ou em nível de portas, que podem ser sintetizadas por ferramentas existentes.

O sistema CALLAS [BIE93] usa VHDL para ambas as especificação de entrada e a descrição do hardware gerado. As principais características deste sistema são a equivalência funcional e temporização da especificação comportamental e a estrutura sintetizada. Inclui várias transformações de alto nível e a nível-RT, no fluxo de dados, e controle para otimizar área e o atraso do circuito.

Embora já existam comercialmente sistemas de síntese de alto nível, o sistema SinMo é proposto. O SinMo diferencia-se de outros sistemas de síntese de alto nível por suportar vários modelos de computação, e por disponibilizar, na ferramenta SASHIMI, os modelos para o projetista. Isto é, quando o projetista possuir um algoritmo que na síntese com a ferramenta SASHIMI, levará muito tempo para a execução, poderá usar os modelos para agilizar o processo.

2.9 Ambiente Sashimi

Durante um longo período, os microcontroladores vêm se impondo como uma importante tecnologia no mercado de sistemas embarcados, em função do seu baixo custo e a integração de funções necessárias às aplicações específicas.

Para o desenvolvimento de aplicações embarcadas simples, foi proposta uma metodologia baseada em um ambiente para geração de aplicações específicas, baseada em software e hardware para microcontroladores, SASHIMI [ITO00]. Utiliza a linguagem Java como tecnologia fundamental para o projeto. Neste ambiente, o projetista fornece uma aplicação Java que será analisada e otimizada para execução em um microcontrolador femtoJava, capaz de executar instruções Java nativamente, exibindo ainda características de um ASIP (*Application Specific Instruction set Processor*).

No ambiente SASHIMI, a implementação das aplicações é realizada através da linguagem Java, seguindo o ciclo tradicional edição-compilação-execução em qualquer plataforma que possua uma implementação do JDK disponível.

O fluxo de projeto desde o código fonte até o chip do microcontrolador sintetizado é ilustrado na Figura 2.12. O código Java da aplicação é fornecido pelo projetista, passando pela compilação e execução em um interpretador. Quando a implementação da aplicação é finalizada, os dados fornecidos pelo projetista e os dados de saída da aplicação são armazenados para posterior fase de validação de *bytecodes*. Uma ferramenta de análise de código carregará o código executável, demonstrando as estruturas internas dos arquivos de classe e fornecendo estimativas de desempenho, provavel área ocupada pelo hardware e dados úteis à fase de adaptação de código. Em geral, a aplicação é adaptada através da tradução de instruções complexas e raras, por seqüências equivalentes de instruções mais simples.

Após a fase de validação de *bytecodes*, o próximo passo compreende uma nova execução da aplicação, confrontando os dados obtidos previamente e que foram armazenados na fase de implementação, de forma que se espera que esta fase possa ser realizada de forma automatizada. Neste ponto, o conjunto de arquivos de classe adaptados são utilizados para resolver a relação entre as classes e converter o código da aplicação e das bibliotecas de sistema em um única imagem para ROM.

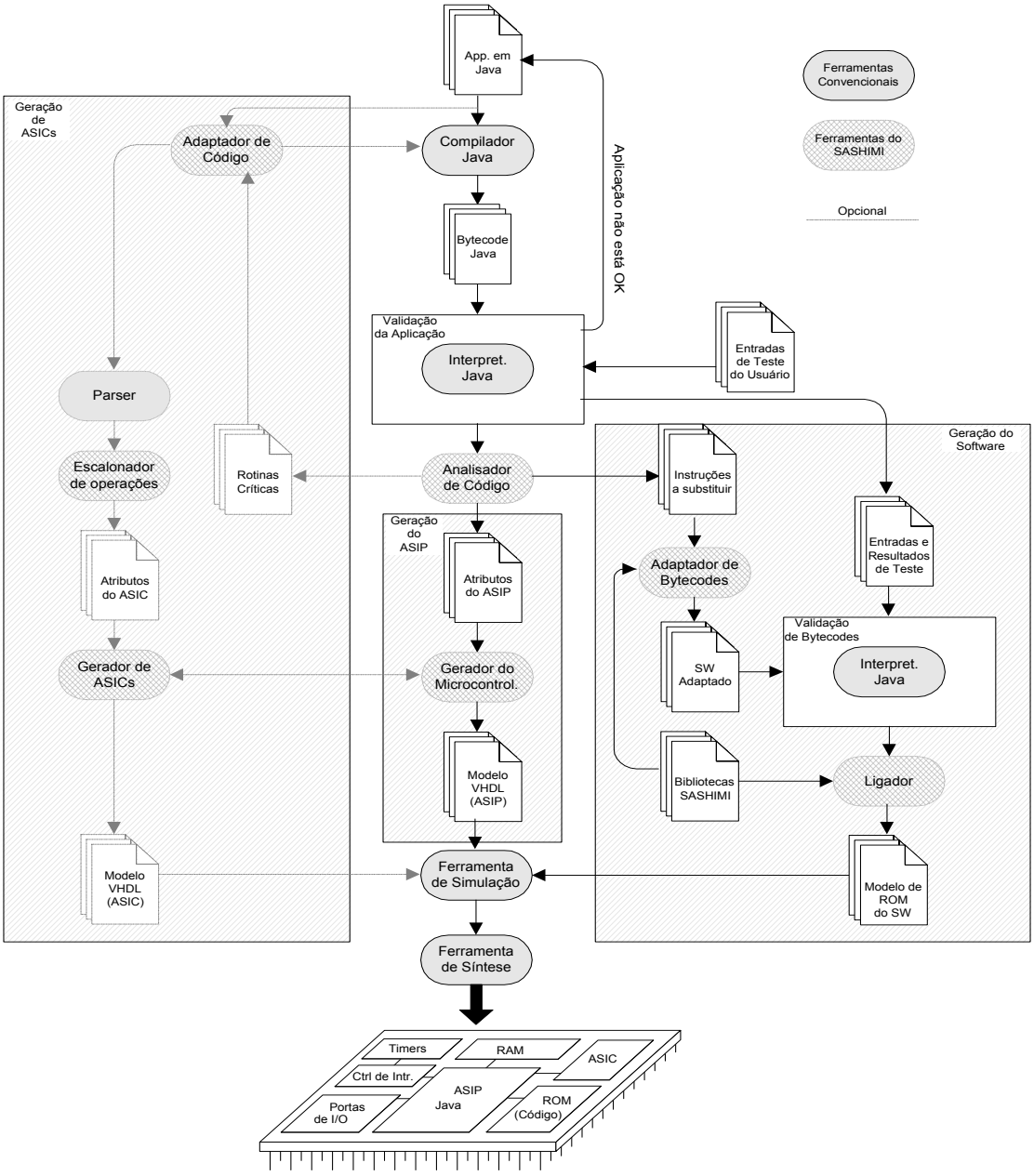


FIGURA 2.12 - Ferramenta SASHIMI
Fonte: ITO. Projetos de Aplicações Específicas com Microcontroladores Java Dedicados. P. 47

3 A Ferramenta de Geração de ASIC e o modelo FSM

Atualmente, é uma prática comum os projetistas descreverem modelos de circuitos e sistemas utilizando linguagens de alto nível, realizarem simulações funcionais e então traduzir o resultado para o modelo a ser implementado em hardware, num sub-sistema apropriado de uma linguagem de descrição de hardware, tais como Verilog e VHDL, que podem ser sintetizadas a nível de portas lógicas. Realizar essa tradução manualmente consome muito tempo, predispõe a erros e é uma tarefa maçante.

Na última década, grupos de pesquisas têm trabalhado para facilitar o mapeamento de modelos de hardware em linguagens de programação de alto nível (ex. C, C++ e Java), para modelos em Linguagem HDL correspondente. Muitas aproximações foram feitas, todas estendidas e limitadas a construções na linguagem de programação [MAR86], [THO90], [CAM91], [MIC90].

Uma vez que estender modelos de software com anotação para suportar síntese de hardware é uma tarefa geralmente aceitável pelo projetista, porque ele adiciona informações para direcionar uma implementação particular, a restrição na construção usável é problemática. Por exemplo, se um projetista que possui um modelo elegante, usando ponteiros na linguagem C, deve passar o código para uma linguagem que não suporta ponteiros, o projetista deve reescrever o modelo. Algumas vezes, reescrever o projeto consome mais tempo do que gerar um novo.

Este trabalho dirige-se nesta área para disponibilizar modelos em Java que possam ser sintetizados para hardware. A maior contribuição deste trabalho é a disponibilização dos modelos de computação para a síntese de hardware. Quando o sistema captura o modelo, saberá quais otimizações realizar, disponibilizando uma maior velocidade na execução do hardware.

Este capítulo tem por objetivo apresentar a ferramenta desenvolvida para a geração de ASIC e o modelo de computação FSM em Java, para a síntese em VHDL.

3.1 Ferramenta de Geração de ASIC

A evolução da Microeletrônica fez com que a complexidade dos circuitos fosse aumentando até chegar ao grau de sofisticação da CPU do computador. No princípio, os circuitos usavam "relés" do tamanho de um copo de 200 mililitros, que tinham um alto consumo. A microscópica dimensão dos transistores miniaturizou os circuitos. O resultado: mais transistores num espaço menor, ou seja, mais espaço para "0" e "1", o que significa mais poder de cálculo com menos consumo de energia.

Entre os diversos tipos de microchips, os ASIC (Circuito Integrado de Aplicação Específica) são especialmente projetados para se conseguir uma determinada função, são caros e a produção se justifica em casos de aplicação em massa, onde um chip de uso geral não garante a mesma eficácia.

Atualmente, os avanços tecnológicos têm permitido o rápido aumento da densidade dos circuitos integrados. A cada dois ou três anos, dobra o número de transistores que podem ser implementados na superfície de uma pastilha de silício. Consequentemente, a complexidade dos projetos tem aumentado proporcionalmente.

Infelizmente, a capacidade humana de compreender o funcionamento destes circuitos não pode aumentar na mesma proporção. Assim, torna-se necessário o uso de novas ferramentas, que permitam refletir os novos recursos em um correspondente aumento de produtividade. Alternativas utilizadas para solucionar os problemas recém surgidos consistem em novas técnicas de projeto. Uma alternativa seria aumentar o nível de abstração na descrição dos circuitos. Isto significa que os projetistas passariam a concentrar-se mais na função dos circuitos, e menos em sua implementação.

A automatização do tratamento de descrições de sistemas digitais desempenha papel primordial na redução do tempo de projeto, bem como na qualidade de ASICs, memórias ou microprocessadores. Entretanto, à medida que a escala de integração de CIs aumenta, cresce a dificuldade de se obter projetos sem erros. Assim, as ferramentas para automatizar o processo de projeto devem estar em constante evolução, para serem capazes de trabalhar com a crescente complexidade dos módulos usados na construção de sistemas digitais. A indústria de ferramentas de Projeto Auxiliado por Computador (*Computer-Aided Design*, ou CAD) tornou-se hoje, ela própria, um setor susbtancial da indústria eletrônica global [DeM 94].

Métodos de síntese automatizada, ótima para equações Booleanas, datam das décadas de 50 e 60, mas seu impacto na prática de projeto de sistemas digitais não foi tão significativo, tendo permanecido como trabalho de relevância puramente teórica até meados da década de 70. A necessidade de manipular quantidades crescentes de informação durante o projeto de sistemas digitais exigia a consideração constante de novas ferramentas. Estas, deveriam transcender as atividades de facilitar a captura e de exercitar descrições, passando a serem capazes de gerar novas descrições de forma automatizada e corretas por construção, baseadas nos mesmos requisitos manipulados pelos projetistas.

O estilo de projeto estabelecido por estas novas ferramentas é centrado não apenas no talento humano, mas também no fato deste talento poder ser capturado por ferramentas computacionais e usado para guiar o processo de projeto. A ferramenta central possui embutido um conjunto de modelos de síntese, e é capaz de gerar uma descrição correta por construção, bem como uma avaliação do desempenho desta descrição. O laço de realimentação é fechado pelo projetista, que julga os resultados da síntese e aceita descrição, ou a rejeita e escolhe um novo modelo de síntese para ser usado. Os modelos de síntese podem ser apenas ligeiramente diferentes entre si, provendo uma exploração do espaço de soluções .

Motivada pelos argumentos apresentados, o presente trabalho apresenta uma metodologia para o desenvolvimento de aplicações específicas, baseada em modelos Java, para síntese em VHDL. A utilização da tecnologia Java torna possível o desenvolvimento e a manutenção das aplicações em qualquer plataforma. O sistema implementado em Java trata as tarefas de compilação dos modelos Java, escalona as operações a serem feitas para uma representação interna para posterior escalonamento e alocação de recursos.

O sistema suporta modelos de Máquina de Estados Finita e Pipeline, adotou-se esses modelos de computação por serem mais usadas para sistemas embarcados. Os algoritmos *List-Scheduling* e *Scheduling-Pipeline* foram implementados para o escalonamento e alocação de recursos. Foram implementados esses algoritmos para conseguir a melhor otimização em cada modelo de hardware sintetizado.

Além dos modelos para a abstração do código VHDL, este trabalho fornecerá suporte à ferramenta SASHIMI [ITO00]. Quando, eventualmente, uma aplicação possuir um requisito de tempo muito exigente para o processador FemtoJava, o código Java deverá ser a um código VHDL sintetizável, de modo a que se consiga executar o algoritmo com maior velocidade. O protocolo de comunicação entre o co-processador VHDL e o processador Java estará também embutido na ferramenta de síntese, visto esta ser dedicada ao sistema SASHIMI.

3.2 Máquinas de Estados Finita

Máquinas de Estados Finita (FSM) e Máquinas de Estados Finita com *Data Path* (FSMD) são métodos de especificação popular em muitas aplicações, tais como, circuitos VLSI, analisadores léxicos, procedimentos de pesquisa em textos, protocolos de comunicação e sistemas de controle. São modelos de projetos também usados em sistemas de síntese de alto nível e projeto de sistemas de hardware. A definição formal de ambos os modelos é apresentada em [GAJ92]. Aqui, uma breve definição de ambos os modelos será apresentada.

Uma Máquina de Estados Finita (FSM) é o modelo de computação mais popular na ciência da computação e engenharia. O modelo consiste de um conjunto de estados, um conjunto de transições entre estados e um conjunto de ações associadas com estados, transições ou ambos (estados e transições). Mais formalmente, uma FSM é um quíntuplo:

$$\left\langle S, I, O, f : S \times I \longrightarrow S, h : SxI \longrightarrow O \right\rangle$$

Onde S = $\{Si\}$ é um conjunto de estados, I = $\{Ij\}$é um conjunto de valores de entrada, e O = $\{Ok\}$é um conjunto de valores de saídas, *f* e h são o próximo estado e os mapas de funções de saídas são obtidos através de S e I para S e O, respectivamente. O modelo básico da FSM pode ser limitado ou estendido para representar diferentes arquiteturas alvo.

A função *f:* S → S é mostrada na figura 3.1 como um diagrama em que cada nodo representa um estado; transições entre estados são representados por fios unidirecionais. Na tabela 3.1, o mesmo diagrama de estados é apresentado na forma tabular. Nesta representação, projetou-se uma codificação de dois bits para cada estado.

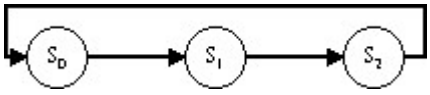


FIGURA 3.1- Diagrama de Estados

TABELA 3.1 - Tabela de próximo estado

Estados Atual	*Próximo Estado*
Q_1Q_0	Q_1Q_0
$S_0 = 0\ 0$	$S_1 = 0\ 1$
$S_1 = 0\ 1$	$S_2 = 1\ 0$
$S_2 = 1\ 0$	$S_0 = 0\ 0$

No exemplo apresentado na figura 3.1 e na tabela 3.1, foi mostrada uma FSM com entradas e saídas. Ambas FSMs baseadas em estados e transições possuem conjunto de entradas não vazias I. Elas diferem na especificação da função de saída h. Na FSM baseada em estado (Moore), o valor da saída depende somente do estado da FSM (h: S -> O), mas na baseada em transição (Mealy), o valor de saída depende dos valores dos estados e das entradas (h: S x I -> 0). Na FSM baseada em transição, a saída irá mudar quando o valor de entrada mudar, enquanto o estado irá mudar somente no próximo pulso do relógio. Na FSM baseada em estado, a saída irá persistir até o estado mudar, não importando quando o valor da entrada mudar. [GAJ92].

O processo de síntese para FSMs, baseada em estado e transição, consiste das seguintes tarefas:

- Compilação
- Minimização de estados;
- Codificação dos estados;
- Síntese da função próximo estado e funções de saídas.

A compilação traduz uma descrição de entrada de uma linguagem particular para uma descrição padrão FSM. A descrição inicial pode ter estados redundantes. Dois estados são redundantes quando muitas seqüências de entrada são aplicadas para FSM em cada estado, gerando a mesma seqüência de saída. Numa instância, um dos dois estados redundantes pode ser eliminado durante a minimização dos estados. Depois da minimização dos estados, a cada estado deve ser determinado num código binário.

Um dos objetivos da codificação dos estados é a minimização dos números de bits usados para codificar cada estado. O resultado é minimizar o número de flip-flops usados para armazenar a codificação dos estados.

O modelo FSM trabalha bem para menos de 100 estados. Além disso, com várias centenas de estados, o modelo fica incompreensível para projetistas humanos. Mesmo que componentes de baixa complexidade, tais como interfaces de I/O, e controles de barramentos, podem ter vários centenas de estados quando todos os elementos armazenados são contados. Para possibilitar o modelo FSM usado para projetos complexos, usou-se um conjunto de variáveis inteiras e de ponto flutuante armazenada em registradores, arquivos de registros e memórias. Cada variável repassa milhares de estados diferentes. Por exemplo, uma variável de 16 bits inteira representa 2^{16} ou 65536 estados diferentes, a introdução de uma variável de 16 bits reduz o número de estado no modelo FSM por 65536. O uso da variável com essa vantagem conceituou-se de uma FSM com datapath (FSMD).

Define-se um conjunto de variáveis VAR, um conjunto de expressões, EXP= $\{f(x,y,z,...) \mid x,y,z,... \in VAR\}$, e um conjunto de tarefas armazenadas, A = $\{X \Leftarrow e \mid X \in VAR, e \in EXP\}$. Futuramente, define-se um conjunto de sinais de "estados" como relação lógica entre duas expressões do conjunto EXP, STAT = $\{\mathrm{Re}\, l(a,b) \mid a,b \in EXP\}$. Dadas essas definições, uma FSMD pode ser definida como um quíntuplo:

$$\langle S, I \cup STAT, O \cup A, f, h \rangle$$

Onde S, f e h são definidas como em FSM, o conjunto de valores de entrada é estendido para incluir uma combinação de status de valores, e o conjunto de saídas é estendido para incluir variáveis armazenadas determinadas.

Como exemplo, pode-se modelar um divisor como uma FSMD baseada em estados, como mostra a tabela 3.2, onde um estado diferente é usado para cada combinação de valores de saída e armazenamento determinado.

TABELA 3.2 - Divisor modelado como uma FSMD baseada em transição

Estado atual	***Entrada***	***Próximo Estado***	***Saída***
S_0	(Count = 1) and (x ≠ 2)	S_0	x = x + 1, y = 0
	(Count = 1) and (x = 2)		x = 0, y = 1
	Count = 0		x = 0, y = 0

3.3 Modelo FSM e FSMD

A figura 3.2 apresenta o modelo de máquina de estados finito com datapath em Java. O sistema reconhece o modelos FSM e FSMD pelo nome da classe. Para escrever os modelos em Java, precisa-se de um ambiente de desenvolvimento para essa linguagem. Para a descrição do modelo Java pode-se usar um editor de texto comum, ou em um editor que possa salvar arquivos em ASCII simples, sem nenhum caracter de formatação.

```
1. import java.io.*;
2. import java.util.*;
3. public class fsm {
4.     public static int x,dx,u,y,a;
5.     public static int x1,u1,y1;
6.     public static void start() {
7.         x = 2; dx = 4;
8.         u = 56; y = -3;
9.         a = 6;
10.    }
11.    public static void varinput() {
12.      BitSet x, a,u,y, dx = new BitSet(7);
13.    }
14.    public static void varoutput() {
15.      BitSet x1, u1, y1 = new BitSet(8);
```

```
16.     }
17.     public static void main(String[] args) {
18.       start();
19.       while(x<a){
20.         x1 = x + dx;
21.         u1 = u -(3*x*u*dx)-(3*y*dx);
22.         y1 = y + (u*dx);
23.         x = x1; u=u1; y = y1;
24.       }
25.     }
26.  }
```

FIGURA 3.2 - Modelo em Java da FSMD

Todo o modelo Java está contido em uma definição de classe. Para o modelo FSM e FSMD, a classe deve possuir o nome respectivo ao modelo. Para o modelo FSM, a classe deve ser FSM.

No método **start()** são atribuídos os valores iniciais para as variáveis a serem inicializadas antes de começar a execução do programa. No exemplo apresentado na figura 3.2 nas linhas 6 à 9, foram atribuídos os valores inicias para as variáveis **x**, **dx**, **u**, **y** e **a**.

Nos métodos **varimput()** e **varoutput()** são definidas as variáveis de entrada e saída do sistema, respectivamente. No exemplo apresentado na figura 3.2, a definição das variáveis de entrada e saída do sistema são definidas em seus respectivos métodos, podendo serem visualizadas nas linhas 11 à 16. Foram definidas as variáveis como BitSet *variável* = new BitSet(4)", definida como BitSet, pois o sistema irá abstrair como um BitVector, o parâmetro 4 indica que a variável possui 4 bits. O número entre parênteses identifica o número de bits para a variável

A parte principal do programa está contida no método **Main()**. Nos aplicativos Java, é o primeiro método a ser executado, assim como a linguagem C. O desenvolvimento do sistema é definido dentro desse método, isto é, toda a execução necessária ao sistema é implementada. No exemplo, a descrição em Java para o cálculo de uma expressão aritmética é apresentada [GAJ92].

Para a compilação do modelo em Java, é necessário passar como parâmetro a biblioteca de operadores, isto é, um arquivo como apresentado na tabela 3.3, dizendo os operadores disponíveis para a abstração do modelo.

TABELA 3.3 - Biblioteca de operadores

Operadores	*Número*
*	2
+	1
-	1
<	1

Ao interpretar o modelo, o parser traduz a descrição em Java para uma estrutura interna, onde se verifica a prioridade na execução. Feito isso, sobre a estrutura o algoritmo *As Soon As Possible* é executado para a obtenção de um maior número de operações num mesmo passo de controle. A estrutura criada pelo parser, após a execução do ASAP, é apresentada na tabela 3.4, onde:

- *ID* – Identificador da expressão
- *Ordem* – Passo de controle que a expressão será executada
- *Operador* – Operador da Expressão
- *ID operando1* – identificação do operando da esquerda
- *ID operando2* – identificação do operando da direita
- *ID exp1* – identificação da expressão do operando da esquerda, isto é, quando o operando da esquerda é uma expressão já contida na estrutura
- *ID exp2* – identificação da expressão do operando da direita, isto é, quando o operando da esquerda é uma expressão já contida na estrutura

TABELA 3.4 - Estrutura armazenando o escalonamento das operações - ASAP

ID	*Ordem*	*Operador*	*ID operando1*	*ID Operando2*	*ID exp1*	*ID exp2*
1	0	+	X	dx		
2	0	*	3	x		
3	0	*	U	dx		
4	0	*	3	y		
5	1	*			2	3
6	1	*		dx	4	
7	1	+	Y			3
8	2	-	U		5	
9	3	-			8	6
10	1	<		a	1	

Após a execução do algoritmo ASAP, é executado o algoritmo ALAP, onde cada operação é colocada o mais próximo possível do último passo de controle. A tabela 3.5 apresenta a estrutura para armazenar o algoritmo ALAP, onde:

- *ID* – Identificador da expressão, referenciando a mesma expressão da estrutura do algoritmo ALAP.
- *Ordem* – Passo de controle que a expressão será executada

TABELA 3.5 - Estrutura armazenando o escalonamento das operações - ALAP

ID	*Ordem*
1	2
2	0
3	0
4	1
5	1
6	2
7	3
8	2
9	3
10	3

Após a execução dos Algoritmos ALAP e ASAP, é executado o algoritmo List-Scheduling, onde as operações são ordenadas em função do número de operadores disponíveis e usando a ordem ALAP, em ordem ascendente, como chave primária e a chave ASAP, em ordem descendente, como chave secundária. Quando ambas as chaves possuem o mesmo valor, qualquer um dos nós é escolhido. Para o exemplo apresentado na figura 3.2, e tendo a biblioteca de operadores apresentada na tabela 3.3, o resultado da execução do algoritmo List-Scheduling, onde as operações são escalonadas em determinado passo de controle, é apresentada na tabela 3.6.

TABELA 3.6 - List-Scheduling

Ordem de execução	*ID*
1	3; 2;1;
2	5;4;7;10
3	6;8
4	9

Com a execução do List-Scheduling, o número de estados para a execução do projeto é determinado. Depois da execução do List-Scheduling, cada estado deve ser determinado num código binário. Um dos objetivos da codificação dos estados é a minimização dos números de bits usados para codificar cada estado, o resultado é minimizar o número de flip-flops usados para armazenar a codificação dos estados.

Após a execução do List-Scheduling, é gerada a descrição de saída. Neste estágio do trabalho, a linguagem de saída em VHDL não está totalmente completa, gerando apenas a seqüência em que as operações devem ser executadas. A descrição de saída gerada atualmente pelo sistema é apresentada na figura 3.3.

```
ENTITY pc IS
PORT
(
        clk, reset, start  : IN bit;  //variáveis de controle
        dx,u,x,y,a         : IN bit_vector(7 downto 0);
        ready              : OUT bit; //indica qdo a FSMD terminou
        sel                : OUT bit_vector(3 downto 0)
       x1,u1,y1            : OUT bit_vector(7 downto 0)

);
END pc;

ARCHITECTURE comportamento OF pc IS
      Type t_state is (st_0,st_1, st_2, st_3,st_4, st_5, st_6, st_7, st_8);
      Signal state : t_state;

BEGIN
PROCESS (clk)
        BEGIN
           If reset='1'then
                ready <='0';
                state <=st_0;
           elsif clk'event and clk='1' then
                case state is
```

```
                        when st_0 =>
                                ready <= '0';
                                if start ='1'then
                                        state<=st_1;
                                end if;
                        when st_1 =>
                                 x <= 000001;
                                 a <= 111000;
                                 dx <= 000110;
                                 u <=001001;
                                 y <=011010;
                                state<=st_2;
                        when st_2 =>
                                if x < a then
                                        state <=st_3;
                                else
                                         x <= tmp3;
                                         u <= tmp9;
                                         y <= tmp7;
                                         ready <= '1';
                                         state <=st_0;
                                end if;
                       when st_3=>
                                tmp1 <= u * dx;
                                tmp2 <= 3 * x;
                                tmp3 <=x + dx;
                                state <=st_4;
                       when st_4 =>
                                tmp4 <= tmp1 * tmp2;
                                tmp5 <= 3 * y;
                                tmp6 <= y + tmp1;
                                 state<=st_5;
                        when st_5 =>
                                tmp7 <= tmp5 * dx;
                                tmp8 <= u – tmp4;
                                 state<=st_6;
                        when st_6 =>
                                tmp4 <= tmp8 * tmp7;
                                 state<=st_2;
                end case;
        end if;
 END PROCESS;
with state select
        sel  <=  B"000" when st_0,
                 B"001" when st_1,
                 B"010" when st_2,
                 B"011" when st_3,
                 B"100" when st_4,
                 B"101" when st_5,
                 B"110" when st_6,
                 B"111" when others;
END comportamento;
```

FIGURA 3.3 - Saída Gerada pelo Sistema

Foi realizado também o estudo de caso com o Podos, sendo um dispositivo capaz de medir a distância percorrida por uma pessoa durante uma corrida ou caminhada. A aplicação desse sistema insere-se na avaliação biomédica, assim como para controle de atividades físicas individuais [CAR 98] [ITO 00].

O Podos é um dispositivo portátil, possuindo fatores fundamentais no seu projeto, sendo eles: o consumo de potência, o tamanho e o peso total do dispositivo. Para tornar-se confiável, é preciso ter-se algumas considerações de variáveis, como por exemplo, peso, altura e padrão de caminha, característica de cada indivíduo e, que podem influenciar nos resultados obtidos.

Para se obter a distância percorrida é realizado o cálculo da integral dupla de aceleração imprimida pela pessoa durante o seu movimento. A aceleração é captada por sensores fixos na perna do atleta e os dados obtidos são então processados para acumular o valor da distância percorrida. O cálculo utilizado para obter a distância percorrida é dada pelas equações 2.1 e 2.2.

$$V = \frac{\sqrt{a^2_v + a^2_h}}{2} \qquad \text{Equação 2.1}$$

$$D = \frac{V_1 + V_{1-1}}{2} \qquad \text{Equação 2.2}$$

Onde a_v e a_h são os valores de aceleração vertical e horizontal, tomados para o cálculo da aceleração resultante do movimento, e V e D correspondem à velocidade e distância, respectivamente. O modelo em Java está apresentado na figura 3.4.

```
import java.io.*;
import java.util.*;

public class fsm {
    public static int value, result,s,d,vacc,hacc,curracc;
    public static int currspeed,prevacc, distance,prevspeed;
    public static void start() {
        s=1; d=1;
        result = 0, prevacc=0;
    }
    public static void varinput() {
      BitSet vacc,hacc= new BitSet(8);
    }
    public static void varoutput() {
      BitSet distance, currspeed = new BitSet(8);
    }
    public static void main(String[] args) {
      start();
      value = (vacc*vacc)+(hacc*hacc);
      while(value - s>=0){
        result = result + 1;
        d = d + 2;
        s = s + d;
      }
      if((s-d)+result < value)
          result = result + 1;
```

```
        curracc = result;
        currspeed = (curracc+prevacc)/2;
        prevacc = curracc;
        distance = distance + (currspeed+prevspeed)/2;
        prevspeed = currspeed;
      }
}
```

FIGURA 3.4 - Modelo em Java do Podos

TABELA 3.7 - Biblioteca de operadores

Operadores	*Número*
*	2
+	2
-	1
/	1
>=	1
<	1

Ao interpretar o modelo, o parser traduz a descrição em Java para uma estrutura interna, sobre a estrutura do algoritmo ASAP. A estrutura gerada é apresentada tabela 3.8.

TABELA 3.8 - Estrutura armazenando o escalonamento das operações - ASAP

ID	*Ordem*	*Operador*	*ID operando1*	*ID Operando2*	*ID exp1*	*ID exp2*
1	0	*	vacc	vacc		
2	0	*	hacc	hacc		
3	1	+			1	2
4	2	-		5	3	
5	0	>=		0	4	
6	1	+	result	1		
7	1	+	d	2		
8	2	+	s			7
9	0	-			8	7
10	1	+			9	6
11	0	<			10	3
12	0	+		1	6	
13	0	+		prevacc	12	
14	1	/		2	13	
15	2	+		prevspeed	14	
16	3	/		2	15	
17	4	+	distance		16	

Após a execução do algoritmo ASAP, é executado o algoritmo ALAP. A tabela 3.9 apresenta a estrutura para armazenar o algoritmo ALAP.

TABELA 3.9 - Estrutura armazendo o escalonamento das operações - ALAP

ID	*Ordem*
1	2
2	2
3	1
4	0
5	0
6	0
7	1
8	0
9	1
10	0
11	0
12	0
13	4
14	3
15	2
16	1
17	0

Após a execução dos Algoritmos ALAP e ASAP, é executado o algoritmo *List-Scheduling*, Para o exemplo apresentado na figura 3.4, e tendo a biblioteca de operadores apresentada na tabela 3.7, o resultado da execução do algoritmo *List-Scheduling*, onde as operações são escalonadas em determinado passo de controle, é apresentada na tabela 3.10.

TABELA 3.10 - List-Scheduling

Ordem de execução	*ID*
1	1;2
2	3
3	4
4	7;6;5
5	8
6	9
7	10
8	11;12
9	13
10	14
11	15
12	16
13	17

Com a execução do *List-Scheduling,* o número de estados para a execução do projeto é determinado. Depois da execução do *List-Scheduling*, cada estado deve ser determinado num código binário. Um dos objetivos da codificação dos estados é a minimização dos números de bits usados para codificar cada estado, o resultado é minimizar o número de *flip-flops* usados para armazenar a codificação dos estados.

Após a execução do *List-Scheduling,* é gerada a descrição de saída. Neste estágio do trabalho, gera-se apenas a seqüência em que as operações devem ser executadas. A descrição de saída, gerada atualmente pelo sistema, é apresentada no Anexo I.

A geração das aplicações tomadas como estudo de caso, através do sistema de síntese proposto, foi realizada assim que foi definido o modelo em Java. A análise das aplicações tornou possível verificar a otimização, levando-se em conta parâmetros como tempo de elaboração do projeto, número de linhas de um projeto descrito em Java, comparado com o descrito em VHDL, e escalonamento das operações.

Uma avaliação rápida dos resultados obtidos com a geração do código, a partir do modelo Java FSMD, permite verificar que o projetista, em nenhum momento, precisou preocupar-se com a otimização das operações, pois com a execução dos algoritmos de escalonamento, a otimização das operações e minimização dos estados foram geradas automaticamente pelo sistema.

A eliminação de estados desnecessários à aplicação representa a otimização com efeitos mais positivos, reduzindo a área e podendo aumentar a freqüência de operação. Comparando a diferença em número de linhas de código, necessárias para a implementação do projeto, em Java é muito menor, comparando com a descrição em VHDL. A tabela 3.11 apresenta a comparação do número de linhas do modelo Java e VHDL gerado pelo sistema.

TABELA 3.11 - Comparação número de linhas do código gerado

Sistemas	*Nº Linhas código Java*	*Nº Linhas código VHDL*
Podos	33	123
Expressão	26	76

No projeto desenvolvido diretamente em VHDL, o projetista terá que preocupar-se com a otimização necessária para reduzir a freqüência do seu projeto, isto é, o escalonamento ótimo para uma biblioteca de operadores disponíveis. Com essas otimizações, o tempo de desenvolvimento de projeto diminui e a possibilidade de erros também.

Portanto, cabe salientar que o projetista deve decidir qual o objetivo do projeto; se precisa de uma freqüência mínima, o número de operadores necessários será maior, consequentemente a área do projeto também será maior. Deve-se ressaltar ainda que o sistema está parcialmente implementado, e que neste estágio de trabalho, apenas a geração da parte de controle, com a indicação em que estados as operações devem ser executadas, está disponível. Espera-se que, com a disponibilidade do ambiente completo, os resultados sejam ainda mais expressivos.

4 Modelo Pipeline e a comparação com o modelo FSMD

Atualmente, processadores para sistemas embutidos necessitam muita performance. Telecomunicação, especialmente comunicação móvel e eletrônicos de consumo, são aplicações de tempo real precisando alta performance do processador para gerenciar as instruções complexas de computação. As características do mercado para DSP complicam o desenvolvimento de hardware e software otimizados. Baixo custo, baixo consumo de potência, algoritmos complexos e pouco tempo para comercialização são contrários à alta flexibilidade, alta velocidade de processamento e software otimizado.

O hardware de DSPs consiste de unidades funcionais para um tratamento efetivo dos algoritmos para processamento de sinais, assim como filtros de resposta finita ao impulso (FIR). Vários barramentos, registradores distribuídos e memórias de programa de dados separadas suportam a exploração do paralelismo.

Para implementar um sistema com *throughput* alto, é muito importante explorar a concorrência no algoritmo. Implementações com muita concorrência podem ser obtidas executando várias operações em uma iteração em paralelo e/ou por sobreposição de iterações consecutivas em um *pipeline*.

Durante muitas décadas, a concorrência dentro de uma única iteração tem sido extensivamente estudada [MCF90]. O objetivo é explorar a concorrência além do limite de uma única iteração. Muita atenção foi dada para a exploração de algoritmos de concorrência, através de pipeline [MCF 90], [PAR 88], [MAL 90], [PAU 89], [HWA 89], [GIR87], [POT90], [AIK88], [ELM89], [GOO90], [CYT 84], [LAM 89], [HWA90].

Este capítulo tem por objetivo apresentar o modelo pipeline em Java, para a abstração do sistema.

4.1 Pipeline

No modelo pipeline, uma tarefa é subdividida numa seqüência de subtarefas; cada uma devendo ser executada por um estágio de hardware específico, que trabalha concorrentemente com os outros estágios do pipeline, criando um paralelismo temporal na execução das subtarefas.

As subtarefas são executadas cada uma por um dos estágios, de forma a se ter na saída da cadeia do pipeline a tarefa executada. Numa estrutura básica de um pipeline linear, os estágios são separados por registradores, cuja função é armazenar o resultado do estágio anterior para execução no próximo estágio. Cria-se dessa forma um delimitador temporal, que será empregado para sincronizar os estágios.

Para o funcionamento do pipeline não ter grandes perdas, é importante que as subtarefas dos estágios tenham um tempo de processamento o mais semelhante possível. Dessa forma, pode-se definir o pipeline como uma cascata de estágios de processamento. Esses estágios são circuitos combinatórios que executarão operações lógicas e aritméticas sobre o fluxo de dados. Entre cada estágio, existem registradores que servem como barreira lógica e que sincronizam o pipeline. São esses que guardam os resultados intermediários de um estágio para outro.

Aplicações típicas que requerem soluções de pipeline são encontradas no domínio de processamento de sinais digitais de vídeo. O tempo de computação é proporcional ao número total de unidades de tempo para completar a computação da entrada. A latência é definida como o número de unidades de tempo entre duas inicializações consecutivas, onde inicialização é o começo de uma computação inicializada na entrada. Um estágio (pipe) é uma peça do hardware que é capaz de executar certas subtarefas de computação. Pode-se visualizar melhor os conceitos na figura 4.1.

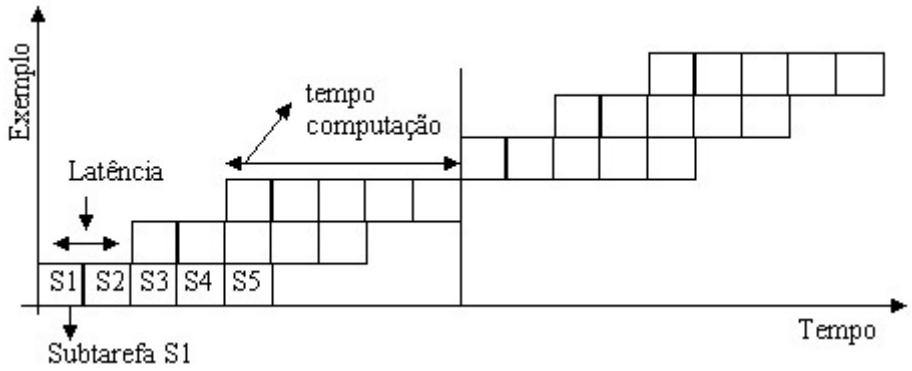


FIGURA 4.1 - Pipeline

A tabela de reserva é uma representação de duas dimensões do fluxo de dados durante uma computação. Uma dimensão corresponde a estágios, e a outra dimensão corresponde a unidades de tempo.

Um lugar na tabela reserva é marcado se um estágio da linha correspondente é usado ou ativado pela unidade de tempo da coluna correspondente. Para verificar quando uma certa latência L pode ser satisfeita por uma opção de implementação, uma simples troca da tabela reserva de acordo com a latência, pode ser feita, até todas as trocas do módulo S serem feitas. As mudanças da tabela reserva são então sobrepostas com a tabela reserva original.

Assim que o lugar marcado na sobreposição da tabela reserva coincidir, a implementação pipeline desejada não pode satisfazer uma latência paterna. Na figura 4.2, o conceito da tabela reserva é ilustrado. A implementação satisfaz uma latência de 2, mas não satisfaz uma latência de 3.

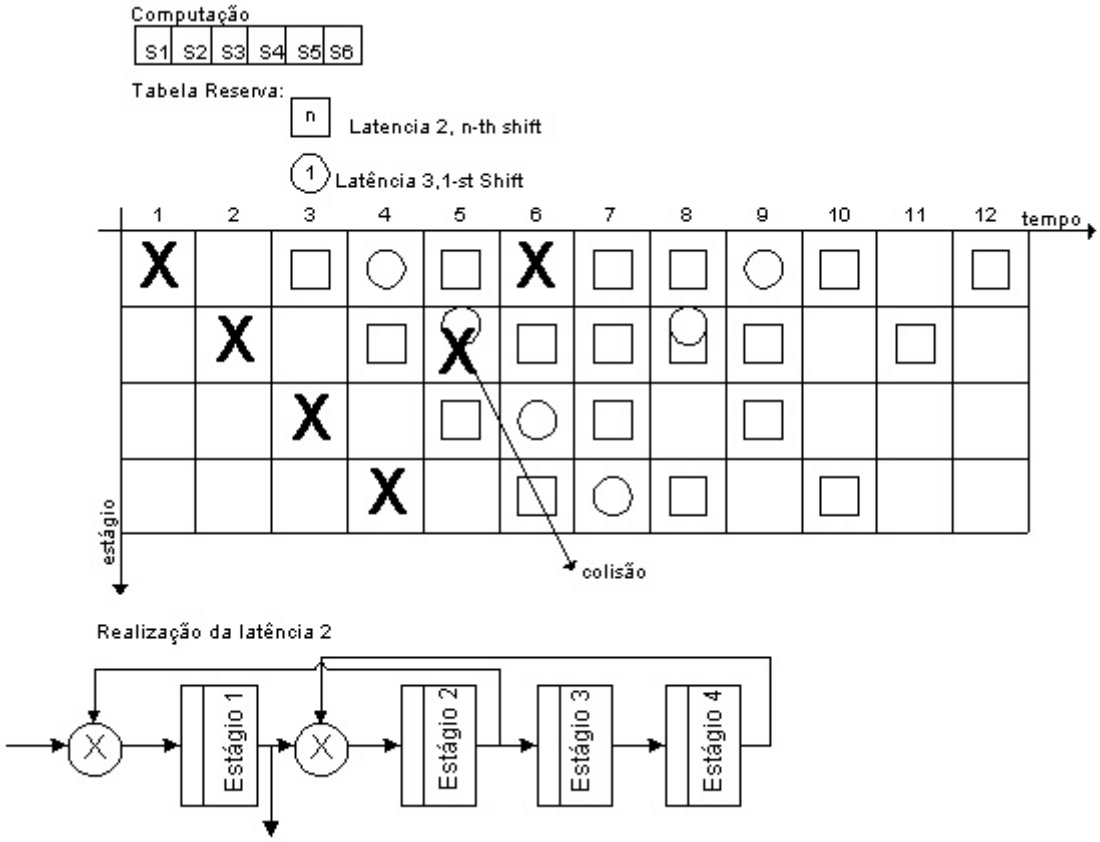


FIGURA 4.2 - O uso da Tabela Reserva

Portanto, usando pipeline pode-se ter a redução em tempo médio de execução de programas, redução média do CPI (Ciclos de Relógio Por Instrução), redução da duração do ciclo de relógio e aceleração do processamento, sem mudar a forma de programação. Soluções com pipeline possuem também algumas inconveniências como: estágios, em geral não podem ser totalmente balanceados; implementação complexa acrescentando custos (hardware, tempo de projeto), e para ser implementado, o conjunto de funções deve ser simples. Portanto, pipelines são difíceis de implementar, mas são fáceis de usar.

4.2 Modelo Pipeline Java

A figura 4.3 apresenta o modelo pipeline em Java. Assim, como no modelo FSM, o sistema reconhece o modelo pipeline pelo nome da classe; e para escrever o modelo em Java, precisa-se de um ambiente de desenvolvimento para essa linguagem.

Para verificar o resultado com o modelo pipeline, foi modelo em Java a descrição comportamental de um filtro ondular elíptico de sexta ordem, possuindo muitas operações de adição, subtração e multiplicação.

```
1. import java.io.*;
2. import java.util.*;
3. public class pipeline {
4.     public static int x,dx,u,y,a;
5.     public static int x1,u1,y1;
6.     public static void start() {
7.
10.     }
11.     public static void varinput() {
12.    BitSet x1, x2,x3,x4, x5, x6, x7,x8 = new BitSet(8);
13.    BitSet x9, x10,x11,x12, x13, x14,x15,x16 = new BitSet(8);
14.    BitSet h0, h1,h2,h3,h4,h5,h6,h7 = new BitSet(8);
15.     }
16.     public static void varoutput() {
17.       BitSet result = new BitSet(8);
18.     }
19.     public static void main(String[] args) {
20.       start();
21.      PO1  = (x1+x2) * h0;
22.      PO2  = (x3+x4) * h1;
23.      PO3  = (x5+x6) * h2;
24.      PO4  = (x7+x8) * h3;
25.      PO5  = (x9+x10) * h4;
26.      PO6  = (x11+x12) * h5;
27.      PO7  = (x13+x14) * h6;
28.      PO8  = (x15+x16) * h7;
29.      result = po1+po2+po3+po4+po5+po6+po7+po8;
25.     }
26. }
```

FIGURA 4.3 - Modelo em Java do filtro FIR

No método **start()** são atribuídos os valores iniciais para as variáveis a serem inicializadas antes de começar a execução do programa. O exemplo da figura 4.1 não possui nenhuma variável para inicialização.

Nos métodos **varimput()** e **varoutput(),** as variáveis de entrada e saída do sistema foram respectivamente determinadas. No exemplo apresentado na figura 4.3, é realizada a definição das variáveis de entrada e saída do sistema em seus respectivos métodos, podendo serem visualizadas nas linhas 11 à 18. Foram definidas as variáveis como "BitSet *variável* = new BitSet(8)", definida como BitSet, pois o sistema irá abstrair como um BitVector de 8 bits. O número entre parênteses identifica o número de bits para a variável

A parte principal do programa está contida no método **main()**. Nos aplicativos, Java este é o primeiro método a ser executado, assim como a linguagem C. O desenvolvimento do sistema é definido dentro desse método, isto é, toda a execução necessária ao sistema é implementada. No exemplo, a descrição em Java para o cálculo de um filtro de 16 taps é apresentada [OLI00].

Para a compilação do modelo em Java, é necessário passar como parâmetro a biblioteca de operadores, o atraso de cada operação, a latência e o tempo máximo por ciclo, isto é, um arquivo como apresentado na tabela 4.1, dizendo os parâmetros disponíveis para a abstração do modelo.

TABELA 4.1 - Biblioteca de Recursos

Operadores	*Número*	*Atraso*
*	3	80
+	5	40
Latência	3	
Máximo de tempo por ciclo	100 ns	

Ao interpretar o modelo, o parser traduz a descrição em Java para estruturas internas, como no modelo FSM. Após, o primeiro passo é identificar os operadores com seus respectivos nós. Como apresentado na tabela 4.2.

TABELA 4.2 - Lista dos operadores com seus respectivos nós

ID	Nó	Operador	Expressão que está representando
1	P1	+	x1+x2
2	P2	+	x3+x4
3	P3	+	x5+x6
4	P4	+	x7+x8
5	P5	+	x9+x10
6	P6	+	x11+x12
7	P7	+	x13+x14
8	P8	+	x15+x16
9	M1	*	P1 * h0
10	M2	*	P2 * h1
11	M3	*	P3 * h2
12	M4	*	P4 * h3
13	M5	*	P5 * h4
14	M6	*	P6 * h5
15	M7	*	P7 * h6
16	M8	*	P8 * h7
17	P9	+	M1+M2
18	P10	+	P9+M3
19	P11	+	P10+M4
20	P12	+	P11+M5
21	P13	+	P12+M6
22	P14	+	P13+M7
23	P15	+	P14+M8

Após, é criada a estrutura com os nós de origem e os de destinos. Realizado isso, o algoritmo Forward Urgency é executado, verificando o maior caminho de cada nó até a saída, levando em consideração o atraso que cada operador representa. A tabela 4.3 apresenta o resultado da execução do algoritmo Forward Urgency, utilizando a biblioteca de recursos apresentada na tabela 4.1.

TABELA 4.3 - Forward Urgency

Nó	*Tempo total*
P1	400
P2	400
P3	360
M1	360
M2	360
P4	320
M3	320
P5	280
M4	280
P9	280
P6	240
M5	240
P10	240
P7	200
M6	200
P11	200
P8	160
M7	160
P12	160
M8	120
P13	120
P14	80
P15	40

Após a execução do algoritmo Forward Urgency, é executado o algoritmo Maximal Scheduling, onde cada operação é alocada em função do máximo tempo por ciclo. Neste exemplo, é de 100ns. O resultado pode ser visualizado na tabela 4.4.

TABELA 4.4 - Maximal Scheduling (100ns)

Time Step	*Nós*
1	P1, P2, P3
2	M1, M2, P4, M3, P5
3	M4, P9, P6, M5, P10, P7
4	M6, P11, P8, M7, P12
5	M8, P13,P14
6	P15

Executado o algoritmo Maximal Scheduling, executa-se o Feasible Scheduling, levando em consideração todos os recursos apresentados na tabela 4.1. O resultado está sendo apresentado na tabela 4.5.

TABELA 4.5 - Feasible Scheduling

Time Step	*Nós*
1	P1, P2, P3, P4, P5
2	M1, M2, M3, P6, P7, P8
3	M4, P9, M5, P10, M6
4	M7, M8

5	P11, P12
6	P13, P14
7	Vazio
8	Vazio
9	P15

Após a execução do Feasible Scheduling, é gerada a descrição de saída. Neste estágio do trabalho, a linguagem de saída em VHDL não está totalmente completa, gerando apenas a seqüência em que as operações devem ser executadas. A descrição de saída gerada, atualmente pelo sistema é apresentada na figura 4.4. Nesta versão não foi tratado as condições quando um modelo possuir if-else-then.

```
ENTITY pc IS
PORT
(
        clk, reset, start  : IN bit;  //variáveis de controle
        x1, x2,x3,x4, x5, x6, x7,x8 = IN bit_vector (7 downto 0);
        x9, x10,x11,x12, x13, x14,x15,x16 = IN bit_vector (7 downto 0);
        h0, h1,h2,h3,h4,h5,h6,h7 = IN bit_vector (7 downto 0);
        ready              : OUT bit; //indica qdo a FSMD terminou
        sel                : OUT bit_vector(3 downto 0)
        result           : OUT bit_vector(7 downto 0)

);
END pc;

ARCHITECTURE comportamento OF pc IS
      Type t_stage is (stage_0,stage_1, stage_2, stage_3,stage_4, stage_5, stage_6, stage_7,
stage_8, stage_9);
      Signal stage : t_stage;

BEGIN
PROCESS (clk)
        BEGIN
           If reset='1'then
                ready <='0';
                state <=stage;
           elsif clk'event and clk='1' then
             p1'<= p1; p2'<=p2; p3'<= p3; p4'<=p4;
             p5'<= p5; p6'<=p6; p7'<= p7; p8'<=p8;
             p9'<= p9; p10'<=p10; p11'<= p11; p12'<=p12;
             p13'<= p13; p14'<=p14; p15'<= p15;
             m1'<= m1; m2'<=m2; m3'<= m3; m4'<=m4;
             m5'<= m5; m6'<=m6; m7'<= m7; m8'<=m8;

                          if stage_1 =>
                                  p1 <=x1+x2;
                                  p2 <= x3+x4;
                                  p3 <= x5 + x6;
                                 p4 <= x7+x8;
                                 p5 <= x9+x10;
```

```
                    m7 <= p7' * h6;
                    m8 <= p8' * h7;
                if stage_2 =>
                    m1 <= p1'* h0;
                    m2 <= p2'*h1;
                    m3 <= p3' * h2;
                    p6  <= x11 + x12;
                    p7  <= x13 + x14;
                    p8  <= x15 + x16;
                    p11 <= p10' + m4';
                    p12 <= p11' + m5';

                if stage_3=>
                    m4 <= p4' * h3;
                    p9  <= m1' + m2';
                    m5 <= p5' * h4;
                    p10 <= p9' + m3';
                    m6  <= p6' * h5;
                    p13 <= p12' + m6';
                    p14 <= p13' + m7';
                    p15 <= p14' + m8';

        end if;
 END PROCESS;
END comportamento;
```

FIGURA 4.4 – Saída Gerada pelo Sistema

A geração das aplicações tomadas como estudo de caso, através do sistema de síntese proposto, foi realizada assim que foi definido o modelo pipeline em Java. A análise das aplicações tornou possível verificar a otimização levando-se em conta parâmetros como tempo de elaboração do projeto, número de linhas de um projeto descrito em Java, comparado com o descrito em VHDL e escalonamento das operações.

Uma avaliação rápida dos resultados obtidos com a geração do código, a partir do modelo Java Pipeline, permite verificar que o projetista em nenhum momento precisou preocupar-se com a otimização das operações, pois com a execução dos algoritmos de escalonamento, a otimização das operações e minimização dos estados foram geradas automaticamente pelo sistema. A tabela 4.6 apresenta a comparação do número de linhas do modelo Java e VHDL gerado pelo sistema.

TABELA 4.6 - Comparação número de linhas

Sistemas	*Nº Linhas código Java*	*Nº Linhas código VHDL*
Filtro 16 taps	26	63

No projeto desenvolvido diretamente em VHDL, o projetista teria de preocupar-se com a otimização necessária para reduzir o custo do seu projeto, isto é, com a biblioteca de recursos disponível. Com essas otimizações, o tempo de desenvolvimento de projeto diminui e a possibilidade de erros também.

Portanto, cabe salientar que o projetista deve decidir qual o objetivo fim do projeto, se precisa de uma freqüência mínima, o número de operadores necessários será maior; consequentemente, a área do projeto também será maior. Deve-se ressaltar ainda que o sistema está parcialmente implementado, e que neste estágio de trabalho apenas a geração da parte de controle com a indicação em que estados as operações devem ser executadas está disponível. Espera-se que, com a disponibilidade do ambiente completo, os resultados sejam ainda mais expressivos.

5 Análise do sistema SinMo e sua ligação com o SASHIMI

Tradicionalmente, ambientes de projeto de hardware trabalham principalmente em função do projetista. Determinando uma especificação funcional ou a performance de um projeto, o projetista poderia criar uma implementação que executasse a função e satisfizesse as características de performance. O projetista também pode refinar o projeto, e verificar se ele satisfaz as especificações funcionais e de performance.

Com o advento de tecnologias VLSI, tornou-se possível empacotar milhões de transistores em um único chip. Esse resultado trouxe um aumento na complexidade do projeto, e aumentou o tempo de elaboração do projeto. Com isso, houve a necessidade de melhorar as metodologias de projetos e sistemas automatizados, que são geralmente conhecidos como CAD.

Há, basicamente, dois métodos de pensamento na área de CAD. O primeiro consiste na premissa proponente que o projetista deve ser o foco do trabalho, orientado ao ambiente de projeto. Acredita-se que todas as decisões do projeto devam ser somente realizadas pelo projetista, que possui muita experiência em desenvolvimento de projetos no passado, e que o projeto deve ser elaborado aumentando a produtividade dos recursos críticos – a experiência do projetista.

Assim, o projetista possui um completo conjunto de ferramentas para capturar, verificar e avaliar o seu projeto. A metodologia de projeto tende a ser bottom-up, e muitos projetistas tendem a projetar primeiro a construção de blocos, e posteriormente aproveitar como componentes para realizar as estruturas de alto nível. O resultado são projetos com alta qualidade. Por outro lado, projetistas humanos são lentos, e realizam projetos com muitos erros. Além disso, projetistas criativos otimizam projetos introduzindo novas regras; através disso, criando uma demanda para novas ferramentas de verificação e linguagens de descrição de projetos.

Devido à limitação do ambiente centralizado no projetista, pesquisadores procuram estabelecer um método de automação de projetos. Neste método, acredita-se que o processo de síntese pode ser automatizado em vários níveis, assegurando o comportamento apropriado para o projeto, e reduzindo tempo de concepção do mesmo

Este capítulo tem por objetivo apresentar as característica da ferramenta SinMo e descrever a proposta para uma futura ligação com o sistema SASHIMI.

5.1 O sistema SinMo

Este sistema disponibilizou a ferramenta de geração de ASIC para o SASHIMI. Utilizando o sistema de SinMo, o projetista deverá primeiramente descrever o sistema, usando modelos em Java, isso facilitará a descrição do projeto e possibilitará o reuso. A alta flexibilidade de poder descrever o modelo em Java e poder sintetizar um circuito sob diferentes modelos somente mudando o nome da classe, é uma característica que o SinMo disponibilizou para o projetista.

A geração de ASIC é uma alternativa que fornece mais flexibilidade ao SASHIMI, pois possibilita a inclusão de circuitos específicos que permitam acelerar a

execução da aplicação. Nesse processo, o próprio projetista pode fornecer o modelo VHDL do ASIC ou optar pela geração automática, a partir da identificação dos métodos cuja execução contribua significativamente para o tempo total de execução da aplicação [ITO00]. Após, o projetista irá executar o projeto na ferramenta SASHIMI e verificar o desempenho do sistema projetado. O usuário do SASHIMI deverá verificar se alguma rotina executa num tempo maior que o esperado.

Uma vez identificadas uma ou mais rotinas, cuja execução não atende aos requisitos de desempenho, os modelos permitirão o estudo de qual o melhor estilo de síntese para geração automática de um ASIC. Isto é representado no bloco Geração de ASICs da figura 5.1. Na medida em que o projetista utilizar os modelos, o sistema conseguirá capturar o estilo de síntese mais adequado para explorar o espaço de projeto com mais eficiência.

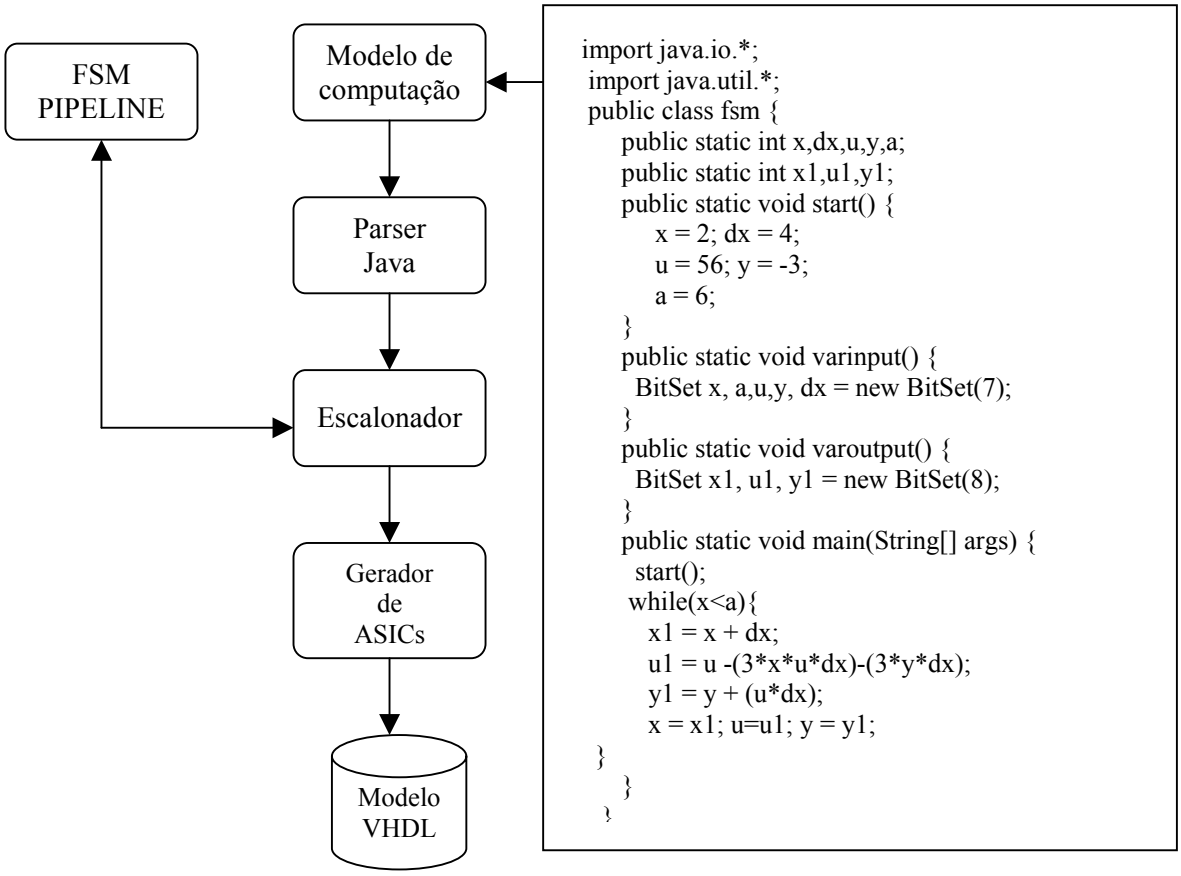


FIGURA 5.1 - Sistema SinMo

De posse do escalonamento adequado, a próxima etapa consiste em gerar o modelo VHDL do circuito que implemente a função desejada, com base em alguma biblioteca de operadores disponível.

Caso o projetista opte por fornecer a descrição VHDL do ASIC, este deve ainda informar ao sistema qual parte do modelo deve ser substituído pelo novo componente, bem como instruir o sistema como sintetizar a comunicação entre microcontrolador e o componente incluso. A comunicação em geral será realizada a partir de posições específicas de memória, e pela utilização de um método *stub* responsável pela leitura e

armazenamento dos resultados na pilha, de forma a manter a semântica e execução consistente à especificação da MVJ (Máquina Virtual Java).

Os modelos do microcontrolador, do código da aplicação e dos ASICs gerados podem ser simulados conjuntamente, depurados utilizando-se ferramentas de simulação comercialmente disponíveis, e posteriormente sintetizados. A síntese do sistema pode ser realizada por uma ferramenta de síntese VHDL convencional, como por exemplo Maxplus II ou Mentor.

Entretanto, é importante ressaltar ainda que os componentes utilizados durante a simulação e modelados como parte do sistema externo ao microcontrolador são removidos pelas ferramentas do ambiente SASHIMI. Contudo, são inclusas todas as rotinas que implementam a interface com tais componentes, baseados no comportamento modelado. Portanto, o projetista pode facilmente dispor de componentes reais e realizar a montagem final do sistema.

O exemplo apresentado na figura 3.1, modelo em Java para a resolução da expressão, foi também sintetizado no SASHIMI. Os resultado obtidos podem ser visualizados na tabela 5.1.

TABELA 5.1 - Comparação da síntese usando ao SASHIMI e o SinMo

Sistemas	*Nº de ciclos SinMo*	*Nº de Ciclos no SASHIMI*
Expressão	26	372

Como se pode observar, o ASIC gerado executa o exigido num número bem menor de ciclos de relógio que se traduz em menor tempo de execução.

O principal valor agregado neste projeto é a disponibilização de geração de ASIC para a ferramenta SASHIMI, o alto nível de abstração com que o projetista pode projetar seu projeto e o sistema ser capaz de abstrair diferentes modelos de computação para uma descrição em VHDL.

5.2 Ligação com o SASHIMI

No fluxo de projeto do SASHIMI está prevista a geração automática de ASICs a partir de métodos Java, permitindo acelerar a execução de uma aplicação. Atualmente, apenas aplicações de uma única *thread* pode ser sintetizada utilizando as ferramentas do SASHIMI, porém, quando a geração de ASICs estiver disponível, diversas *threads* poderão ser sintetizadas como hardware que irão se comunicar com a *theread* principal executando no microcontrolador FemtoJava.

O microcontrolador FemtoJava é composto por uma unidade de processamento baseada em arquitetura de pilha, memórias RAM e ROM integradas, portas de entrada e saída mapeadas em memória e um mecanismo de tratamento de interrupções com dois níveis de prioridade. Uma possível solução para a ligação com o SASHIMI é apresentada na figura 5.2.

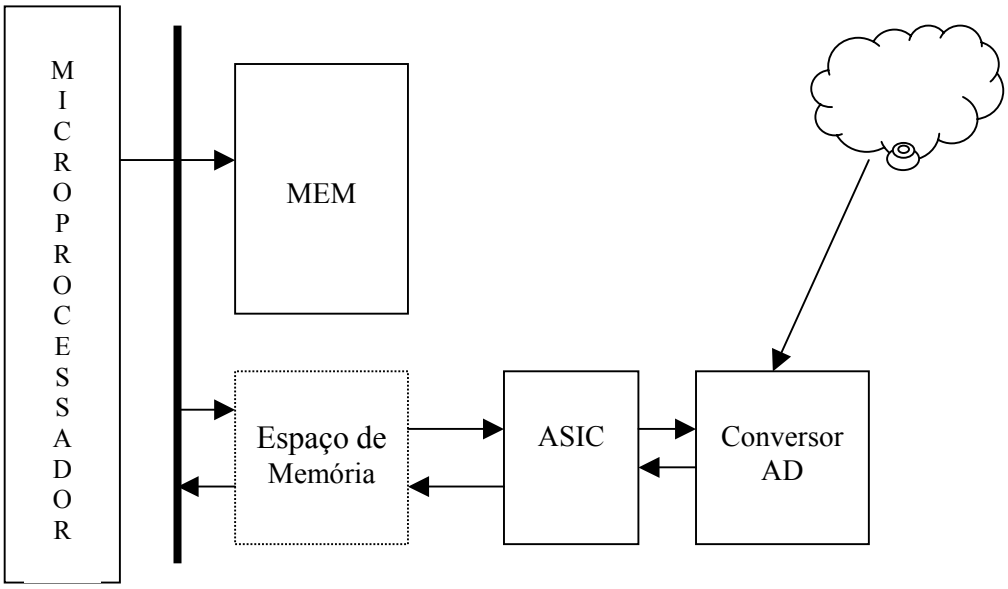


FIGURA 5.2 - Ligação da ferramenta SinMo ao SASHIMI

A ferramenta SinMo seria ligada diretamente com o processador, através de um conjunto de posições no espaço de endereçamento do processador. Um conversor analógico digital também poderia estar ligado ao circuito gerado pela ferramenta SinMo.

O ambiente de projeto SASHIMI automatiza o processo de projeto de aplicações dedicadas em Java, tendo o microcontorlador FemtoJava como plataforma de hardware alvo. Automatizar as atividades do projetista e ainda permitir alternativas de projetos, podem acelerar a concepção e a chegada de novos produtos ao mercado.

A geração de ASICs é uma alternativa que fornecerá ainda mais flexibilidade ao SASHIMI, pois possibilita a inclusão de circuitos específicos que permitem acelerar a execução da aplicação. A tabela 5.1 apresenta o resultado do exemplo apresentado na tabela

Durante este capítulo, foi apresentada uma abordagem para o projeto de descrições em Java para geração de ASIC. De fato, o presente trabalho está provando ser possível implementar aplicações em um alto nível de abstração em Java tendo um ASIC como dispositivo alvo.

6 Conclusões

Pela quantidade de sistemas em desenvolvimento e a qualidade de pesquisa na área, é fácil constatar a importância da síntese automática de alto nível para a geração eficiente de circuitos de alta complexidade.

No texto, foram apresentados os detalhes envolvidos na concepção do presente trabalho, contextualizando-o no cenário de sistemas de síntese de alto nível. As motivações do trabalho residiram basicamente na contribuição com a ferramenta SASHIMI, onde a geração de ASIC poderá ser acrescentada à ferramenta, através dos algoritmos de escalonamento implementados e da geração automática da descrição VHDL.

6.1 Contribuições do Trabalho

O presente trabalho procurou fornecer um estudo para apresentar alternativas e soluções de implementações para ASICs, onde se busca elevada performance e flexibilidade com baixo custo, e um rápido desenvolvimento do projeto. Apresentou-se uma metodologia capaz de obter uma implementação semi-automática de circuitos integrados, a partir de especificações realizadas na linguagem Java, para uma saída em VHDL. Primeiramente, as diferentes estruturas da linguagem Java foram analisadas e os modelos em Java foram propostos. Essa versão permite a obtenção de especificações eficientes em Java que facilitariam o mapeamento em VHDL. Em segundo lugar, foram estabelecidas normas para um bom acoplamento entre o modelo da linguagem a ser utilizada para a especificação dos modelos FSM e Pipeline e a arquitetura gerada após a síntese. Nesta etapa ainda foram implementados alguns dos algoritmos básicos usados na tarefa de escalonamento, durante a Síntese de Alto Nível.

Em seguida, foram estabelecidas as regras para obter um sistema descrito em VHDL, a partir de sua especificação Java. Finalmente, foram obtidos os modelos propostos neste trabalho. A metodologia proposta neste trabalho integra o CAD SASHIMI.

A linguagem Java torna possível o desenvolvimento e manutenção das aplicações em qualquer plataforma. VHDL é uma linguagem de descrição de hardware, usada para a simulação e síntese de hardware. Para um projetista de CIs, Java é mais fácil de ser utilizada para especificar e analisar o projeto do que o VHDL, já que Java é uma linguagem de alto nível e dispõem de uma bibliografia abundante.

A estratégia proposta para o mapeamento de especificações Java, em descrições VHDL, é adequada à simulação. Os procedimentos que tornam este mapeamento adequado à síntese, quando se usa o ambiente SASHIMI, são também indicados.

Este trabalho apresentou os diferentes estilos de projetos de ASICs que podem ser descritos em VHDL, e as restrições que determinam o mais adequado modelo para os objetivos do projeto à serem realizados. As principais contribuições deste trabalho são:

- O uso de uma linguagem de alto nível, Java, para especificação do projeto, permitindo alta produtividade, redução dos erros e tempo de projeto;

- A abstração com que o projetista pode trabalhar em seu projeto com a disponibilização em Java dos modelos FSM e Pipeline;
- O escalonamento automático das operações realizadas, reduzindo o número de ciclo de relógio para a execução do projeto;
- Geração automática dos projetos, a partir da linguagem Java para VHDL;
- A inclusão de soluções que permitam o co-design de hardware/software, utilizando o mapeamento proposto de Java para VHDL e a futura ligação com o CAD SASHIMI.

6.2 Pesquisas Futuras

Esta dissertação foi direcionada para o escalonamento e a alocação de recursos para na síntese de alto nível. Como citado anteriormente, o sistema fornecerá suporte à ferramenta SASHIMI [ITO99], sendo observado que neste trabalho, a Geração de ASIC o ramo à esquerda da figura 2.12, foi explorada. O sistema de síntese proposto diferencia-se de outros sistemas de síntese de alto nível por suportar vários modelos de computação, e por disponibilizar, na ferramenta SASHIMI, os templates para o projetista. Isto é, quando o projetista possuir um algoritmo que na síntese com a ferramenta SASHIMI, levará muito tempo para a execução, poderá usar os templates para agilizar o processo de síntese do circuito dedicado.

Utilizando o sistema de síntese proposto, o projetista deverá primeiramente descrever o sistema usando templates, isso facilitará a descrição do projeto e possibilitará o reuso. Após, o projetista irá executar o projeto na ferramenta SASHIMI e verificar o desempenho do sistema projetado. O usuário do SASHIMI deverá verificar se alguma rotina executa num tempo maior que o esperado.

Uma vez identificadas uma ou mais rotinas cuja execução não atende aos requisitos de desempenho, o projetista poderá verificar, através dos modelos gerados, qual modelo indica o melhor estilo de síntese para geração automática de um ASIC. Isto é representado no bloco Geração de ASICs da figura 2.12.

Na medida em que o projetista utilizar os templates, o sistema conseguirá capturar o estilo de síntese mais adequado para explorar o espaço de projeto com mais eficiência.

Futuramente, esperam-se melhorias sensíveis quanto a geração do código VHDL, fornecendo os operadores na descrição gerada, e a ligação com o fentoJava.

Anexo 1 Código VHDL gerado pelo sistema

```
ENTITY pc IS
PORT
(
        clk, reset, start       : IN bit;  //variáveis de controle
        vacc, hacc              : IN bit_vector(7 downto 0);
        ready                   : OUT bit; //indica qdo a FSMD terminou
        sel                     : OUT bit_vector(5 downto 0)
        distance, currspeed : OUT bit_vector(7 downto 0)

);
END pc;

ARCHITECTURE comportamento OF pc IS
      Type t_state is (st_0,st_1, st_2, st_3,st_4, st_5, st_6, st_7, st_8, st_9, st_10, st_11,
st_12, st_13, st_14, st_15, st_16);
      Signal state : t_state;
      Signal s, d, result : bit_vector(7 downto 0);
      Signal value,curracc, prevacc, prevspeed : bit_vector(7 downto 0);
      Signal sinal      : bit;


BEGIN
PROCESS (clk)
        BEGIN
          If reset='1'then
                Ready <='0';
                State <=st_0;
          Elsif clk'event and clk='1' then
                case state is
                        when st_0 =>
                                ready <= '0';
                                if start ='1'then
                                        state<=st_1;
                                end if;
                        when st_1 =>
                                 s <= 00000001;
                                 d <= 00000001;
                                 result <= 00000000;
                                 prevacc <= 00000000;
                                 prevspeed <= 00000000;
                                 distance <= 00000000;
                                state<=st_2;
                        when st_2 =>
                                if  sinal='0' then
                                        state <=st_3;
                                else
                                        ready <= '1';
                                        distance <= tmp14;
                                        currspeed <= tmp12;
                                        state <=st_0;
```

```
                    end if;
                when st_3=>
                    tmp1 <= vacc * vacc;
                    tmp2 <= hacc * hacc;
                    state <=st_4;
                when st_4 =>
                    tmp3 <= tmp1 + tmp2;
                    state<=st_5;
                when st_5 =>
                    tmp4 <= tmp3 – s;
                    state<=st_6;
                when st_6 =>
                    if  tmp4 >= '00000000' then
                        state<=st_7;
                    else
                        state <= st_9;
                    end if;
                when st_7 =>
                    tmp5 <= result + 00000001;
                    tmp6 <= d + 00000010;
                    state <= st_8;
                when st_8 =>
                    tmp7 <= s + tmp6;
                    state <= st_6;
                when st_9 =>
                    tmp8 <= tmp7 - tmp6;
                    state <= st_10;
                when st_10 =>
                    tmp9 <= tmp8 + tmp5;
                    state <= st_11;
                when st_11 =>
                    if tmp9 < tmp3 then
                        tmp10 <= tmp5 + 00000001;
                        state <= st_12;
                    else
                        state <= st_12;
                when st_12 =>
                    tmp11 <= tmp10 + prevacc;
                    state <= st_13;
                when st_13 =>
                    tmp12 <= tmp11/00000010;
                    state <= st_14;
                when st_14 =>
                    tmp13 <= tmp12 + prevspeed;
                    state <= st_15;
                when st_15 =>
                    tmp14 <= tmp13 / 00000010;
                    state <= st_16;
                when st_16 =>
                    tmp14 <= distance+tmp14;
                    state <= st_12;

            end case;
        end if;
```

```
 END PROCESS;
with state select
        sel  <=  B"00000" when st_0,
                 B"00001" when st_1,
                 B"00010" when st_2,
                 B"00011" when st_3,
                 B"00100" when st_4,
                 B"00101" when st_5,
                 B"00111" when st_6,
                B"01001" when st_7,
        B"01010" when st_8,
                 B"01011" when st_9,
                 B"01110" when st_10,
                 B"01111" when st_11,
                 B"10000" when st_12,
                 B"10001" when st_13,
                 B"10010" when st_14,
                 B"10011" when st_15,
                 B"10111" when st_16,
                 B"11111" when others;
END comportamento;
```

Bibliografia

[AIK 88] AIKEN, A.; NICOLAU, A. Optimal loop parallelization. Ithaca: Cornell University, 1988. 12 p. : il. (Technical report 88-905 cornell university)

[AJL 99] AJLUNI, Cheryl. System-Level Design Language Promises a Unified, Integred Design Flow. **Eletronic Design**, Cleveland OH, v. 77, n. 15, p.35-36, July 1999.

[ALT 96] ALTERA CORPORATION. **Data Book.** San Jose, California, 1996.

[ASH 90] ASHENDEN, Peter J. **The VHDL Cookbook.** South Australia: Dept. Computer Science of University of Adelaide, 1990.

[BER 93] BERGAMASCHI, R.A; KUEHLMANN, A. A System for Production Use of High-Level Synthesis. **IEEE Transactions on Very Large Scale Integration (VLSI)**, New York, v.1, n.3, p.223-243, Sept. 1993.

[BER 2000] BERTASI, Débora. **Implementação Parcial de um Sistema de Síntese de Alto Nível**. 2000. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[BIE 93] BIESENACK, J.; KOSTER, M. The Siemens High-Level Synthesis System Callas. **IEEE Transactions on Very Large Scale Integration (VLSI)**, New York, v. 1, n.3, p.244-253, Sept 1993.

[BYE 84] BYE, C.T.; LIGHTNER, M.R.; RAVENSCROFT, D.L. A Functional Modeling and Simulation Environment based on ESIM and C. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1984, Santa Clara. **Proceedings...** Califórnia: ICCAD, 1984. p. 51-53.

[CAM 90] CAMPOSANO, R.; BERGAMASCHI, R. A. Synthesis using path-based scheduling: algorithms and exercises. In: DESIGN AUTOMATION CONFERENCE, 27., 1990, Orlando. **Proceedings...** NewYork: ACM/IEEE, 1990. p. 450-455.

[CAM 91] CAMPOSANO, R. From Behavior to Structure: High-Level Synthesis. **IEEE Design & Test of Computers**, New York, v.8, n.1, p.43-49, Mar.1991.

[CAM 96] CAMPOSANO, R.; WILBERG, J. Embedded System Design. **Jornal On Design Automation For Embedded Systems**, [S.l.], v. 1, p. 5-50, 1996.

[CAR 98] CARRO, L.; NALE, L. B. de. Podos: Distance Measurement Device. In: UFRGS MICROELETRONICS SEMINAR, 13., 1998, Porto Alegre. **Proceeding...** Porto Alegre: PGCC da UFRGS, 1998. p. 165 - 168.

[CLA 88] CLAESEN, L. et al. Automatic Synthesis of Signal Processing Benchmark using the CATHEDRAL Silicon Compilers. In: IEEE CUSTOM INTERGRATED CIRCUITS CONFERENCE, 1989, San Diego. **Proceeding...** California: IEEE/CICC, 1988. p. 14.17.1-14.17.4.

[CYT 84] CYTRON, R. Compiler-time Scheduling and Optimization for Asynchronous Machines. 1984. Tese (Doutorado em Ciência da Computação) - Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana.

[DeM 94] MICHELI, G. De. **Synthesis and optimization of digital circuits**. Stanford: Mc Graw-Hill, Inc., 1994.

[ELM 89] HAROUN, B.S.; ELMASRY, M.I. Architectural Synthesis for DSP Silicon Compiler. **IEEE Transactions on Computer-Aided Design**, New York, v.8, p. 431-447, April 1989.

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

[GIR 87] GIRCZYC; E.M. Loop Winging - a Data Flow Approach to Functional Pipelining**.** In: IEEE ISCAS, 1987, Philadelphia. **Proceedings...** [S.l.] IEEE,1987. p. 382-385.

[GIR 99] GIRAULT, A.; LEE, B.; LEE, E. Hierarchical Finite State Machines with Multiple Concurrency Models. **IEEE Transactions on Computer-Aided Design**, New York, v.18, n.6, p. 742-760, June 1999.

[GLU 90] GLUNZ, W.; UMBREIT, G. VHDL for High-Level Synthesis of Digital Systems. In: EUROPEAN CONFERENCE ON VHDL, 1990, Marseilles. **Proceedings...** [S.l.: s.n.], 1990. p.1-11.

[GOO 90] GOOSSENS, G. et al. An efficient Micro-code Compiler for Application Specific DSP Processors. **IEEE Transactions on Computer-Aided Design**, New York, v.9, n.9, p. 925-937, Sept. 1990.

[HWA 89] HWANG, Ki Soo. et al. Scheduling and Hardware sharing in pipelined data paths. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, ICCAD, 1989, Santa Clara. **Digest of Technical papers**. Washington: IEEE Computer Society, 1989. p. 24-27.

[HWA 90] HWANG, C. T. et al**.** Optimum and Heuristic Data Path Scheduling under Resource Constraints. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, DAC, 27., 1990, Orlando. **Proceedings...** New York: IEEE, 1990. p. 65-70.

[ITO 99] ITO, Sérgio Akira. **Estudo de Requisitos de Hardware para a Plataforma Java**. 1999. Trabalho Individual(Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[ITO 99a] ITO, S. A.; SUSIM, A.; CARRO, L.; JACOBI, R. Implementação de uma Máquina Java. In: WORKSHOP IBERCHIP, 5., 1999, Lima, Peru. **Memorias...** [Lima:Hozlo S.R.L.], 1999. p.252-259.

[ITO 2000] ITO, Sérgio Akira. **Projeto de Aplicações Específicas com Microcontroladores Java Dedicados.** 2000. Dissertação de Mestrado (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brasil.

[JUL 93] JULIUSSEN, K. P.; JULIUSSEN, E. **The 6th. Annual Computer Industry Almanac**. Austin, TX: The Reference Press, 1993.

[LAM 89] LAM, M. S. **A Systolic Array Optimizing Compiler.** 1987 Tese de Doutorado(Doutorado em Ciência da Computação) – Departamento de Ciência da Computação, Carnegie Mellon University, Carnegie Mellon.

[LAU 92] MICHEL, P.; LAUTHER, U.; DUZY, P. **The Synthesis Approach to Digital System Design.** Norwell, Massachusetts: Kluwer Academic Publishers, 1992.

[LAV 2000] LAVALOGIC COMPANY. FPGA Enabled Home Networking Technology Bridges. Disponível em: <http://lavalogic.com>. Acesso em: Jan. 2000.

[LIP 90] LIPSET, R. et al. VHDL: Hardware Description and Design. Boston: Kluwer Academic Publishers, 1990. 300p.

[LIS 88] LIS, J.; GAJSKI, D. D. **Synthesis from VHDL**. In: INTERNACIONAL CONFERENCE ON COMPUTER DESIGN, 1988. **Proceeding...** New York: CM/IEEE, 1988. p.606-609.

[MAN 87] MAN, H. D. **Synthesis of DSP Systems at Leuven**. In: IEEE CONFERENCE COMPUTER DESIGN: VLSI IN COMPUTERS AND PROCESSORS, 1987. **Proceedings...** New York: ICCD, 1987. p. 133-145.

[MAL 90] MALLON, D. J.; DENYER, P. B. **A New Approach To Pipeline Optimisation**. In: EUROPEAN DESIGN AUTOMATION CONFERENCE, 1990. **Proceedings...** Scotland:EDAC, 1990. p 83-88.

[MAR 86] MARWEDEL, P. **A new synthesis algorithm for the MIMOLA software system**. In: DESIGN AUTOMATION CONFERENCE, 1986. **Proceedings...** New York:ACM/IEEE, 1986. p.271-277

[MCF 90] MCFARLAND, M.C.; PARKER, A.C.; CAMPOSANO, R. The High-Level Synthesis of Digital System. **Proceedings of the IEEE**, New York, v.78, n.2, p. 301-318, Feb.1990.

[MEN 93] MENTOR GRAPHICS CORPORATION. **Mentor Graphics System Overview Manual.** Wilsonville, Oregon, 1993.

[MIC 90] MICHELI, G. De et al. The Olympys synthesis system. **IEEE Design & Test of Computers**, New York, v.7, n.5, p.37-53, Oct.1990

[KU 92] KU, David; MICHELI, Giovanni De. **High-Level Synthesis of ASICs under Timing and Synchronization Constraints.** Boston, MA: Kluwer Academic Publishers, 1992.

[MIC 92] MICHEL, Petra. **The synthesis approach to digital system design**. Boston: Kluwer, c1992. 415 p.

[MIT 93] MITRA, S.K.; KAISER, J. F. **Handbook for Digital Signal Processing.** Somersrt, NJ: John Wiley & Sons, 1993.

[NAV 92] NAVABI; Z. A high-level language for Design and Modeling of Hardware. **Journal of Systems and Software,** Amsterdam, p. 5-18, Dec. 1992.

[NET 99] NETO, H.; CARDOSO, J. M. P**.** Macro-Based Hardware Compilation of Java Bytecodes into a Dynamic Reconfigurable Computing System. In: IEEE SYMPOSIUM ON FIELD PROGRAMMABLE CUSTOM COMPUTING MACHINES,1999. **Proceeding...** Napa Valley, California: FCCM/IEEE, 1999.

[PAR 88] PARK, N.; PARKER, A.C. Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications. In: IEEE TRANS. ON COMPUTER-AIDED DESIGN, 1988. **Proceeding...** Anaheim, CA: DAC'88, 1988. p 356-370.

[PAU 89] PAULIN, P.G.; KNIGHT, J.P**.** Force-Directed Scheduling for the Behavioral Synthesis of ASIC's. **IEEE Transaction on Computer-Aided Design**, Canada, v.8, n.6, p.661-679, June 1989.

[POT 90] POTASMAN, R. et al. Percolation Based Synthesis. In: DESIGN AUTOMATION CONFERENCE, 27., 1990, Orlando. **Proceedings...** New York: ACM/IEEE, 1990. p. 444-449.

[RAB 91] RABAEY,J. et al. Fast Prototyping of Data Path intensive Architectures. **IEEE Design & Test of Computers**, New York, v.8, n. 2, June 1991.

[SUN 95] SUN MICROSYSTEM. **Writing Java Programs.** Disponível em: <http://java.sun.com/doc/programer.html>. Acesso em: dez. 2002.

[SUN 95a] SUN MICROSYSTEM. **The Java Language Tutorial.** Disponível em: <http://java.sun.com/tutorial/index.html>. Acesso em: fev. 2002.

[SUN 95b] SUN MICROSYSTEM. **The Java Language Environment.** Disponível em: <http://java.sun.com>. Acesso em: fev. 2002.

[THO 90] THOMAS, D. E. **Algoritmic and Register-Transfer Level Synthesis: the System Architect's Workbench**. Boston: Kluwer Academic Publishers, 1990. 300p.

[TSE 86] TSENG, C.; SIEWIOREK, D. Automated Synthesis of Data Paths on Digital Systems. **IEEE Transactions on Computer-Aided Design**, New York, v. 5, n.3, p.379-395, July 1986.

[WAG 88] WAGNER, F.R. et al. **Métodos de Validação de Sistemas Digitais.** Campinas: UNICAMP, 1988. 300p.

[WAL 91] WALKER, R.; CAMPOSANO,R. **Survey of High-Level Synthesis Systems**. Boston: Kluwer Academic Publishers, 1991.