UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
PROGRAMA DE PÓS-GRADUAÇAO EM ENGENHARIA ELÉTRICA E INFORMÁTICA INDUSTRIAL

PERCI AYRES ANTIQUEIRA

IMPLEMENTAÇÃO DE MODELOS DE REDES DE PETRI EM HARDWARE DE LÓGICA RECONFIGURÁVEL

DISSERTAÇÃO

CURITIBA

2011

PERCI AYRES ANTIQUEIRA

IMPLEMENTAÇÃO DE MODELOS DE REDES DE PETRI EM HARDWARE DE LÓGICA RECONFIGURÁVEL

Dissertação apresentada ao Programa de Pós-graduaçao em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do grau de "Mestre em Ciências" – Área de Concentração: Informática Industrial.

Orientador: Prof. Dr. Carlos Raimundo Erig Lima

Co-orientador: Prof. Dr. Jean Marcelo Simão

CURITIBA

2011

Dados Internacionais de Catalogação na Publicação

A633 Antiqueira, Perci Ayres
Implementação de modelos de redes de Petri em hardware de lógica reconfigurável / Perci Ayres Antiqueira .— 2011
125 p.. : il. ; 30 cm

Orientador: Carlos Raimundo Erig Lima
Coorientador: Jean Marcelo Simão.
Dissertação (Mestrado) – Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial. Curitiba, 2011.
Bibliografia: p. 123-125.

1. Redes de Petri. 2. VHDL (Linguagem descritiva de hardware). 3. Arranjos de lógica programável em campo. 4. Dispositivos lógicos programáveis. 5. Engenharia elétrica – Dissertações. I. Lima, Carlos Raimundo Erig, orient. II. Simão, Jean Marcelo, coorient. III. Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial. IV. Título.

CDD (22. ed.) 621.3

Biblioteca Central da UTFPR, Campus Curitiba

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
Câmpus Curitiba

Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial

Título da Dissertação Nº 585

"Implementação de Modelos de Rede de Petri em *Hardware* de Lógica Reconfigurável"

por

Perci Ayres Antiqueira

Esta dissertação foi apresentada como requisito parcial à obtenção do grau de MESTRE EM CIÊNCIAS – Área de Concentração: Engenharia de Automação e Sistemas, pelo Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial – CPGEI – da Universidade Tecnológica Federal do Paraná – UTFPR – Câmpus Curitiba, às 9h do dia 15 de dezembro de 2011. O trabalho foi aprovado pela Banca Examinadora, composta pelos professores:

Prof. Carlos Raimundo Erig Lima, Dr.
(Presidente – UTFPR)

Prof. Luis Allan Kunzle, Dr.
(UFPR)

Prof. Paulo Cézar Stadzisz, Dr.
(UTFPR)

Visto da coordenação:

Prof. Fábio Kurt Schneider, Dr.
(Coordenador do CPGEI)

Dedico este trabalho aos meus pais Patricio e Vidalvina "*in memoriam*", que sempre deram suporte e estímulo para minha formação intelectual e à minha esposa Silvana pelo incentivo, carinho e apoio em minha vida.

AGRADECIMENTOS

Agradeço aos professores Carlos Raimundo Erig Lima e Jean Marcelo Simão pela orientação e apoio prestados durante o desenvolvimento deste trabalho. Agradeço também aos colegas de laboratório que em muito contribuíram para a concretização deste trabalho, em especial ao colega Robson Ribeiro Linhares pela colaboração técnica nos experimentos de análise de desempenho e aos colegas Alisson Antônio de Oliveira e Faimara do Rocio Strauhs pelas sugestões e incentivos prestados. Agradeço aos professores membros da banca examinadora pela revisão deste trabalho. Agradeço à CAPES pelo apoio financeiro através de bolsa de estudos. Agradeço à minha família e em especial à minha esposa Silvana pela compreensão, apoio e incentivo em todos os momentos do meu mestrado. Acima de tudo, agradeço a Deus por ter me dado forças para esta empreitada.

RESUMO

ANTIQUEIRA, Perci Ayres. Implementação de Modelos de Redes de Petri em Hardware de Lógica Reconfigurável. 125 f. Dissertação – Programa de Pós-graduaçao em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná. Curitiba, 2011.

Neste trabalho de pesquisa, foi realizado um estudo dos principais tipos de ferramentas para modelagem de *hardware* buscando-se verificar as vantagens da utilização de Redes de Petri para a modelagem de sistemas dinâmicos e concorrentes e de sua implementação em *hardware*. Observou-se que, apesar de existirem ferramentas para esta finalidade, existem pontos que podem ser trabalhados para facilitar o acesso a esta tecnologia. Assim, foi desenvolvido um método para facilitar a implementação de sistemas modelados em Redes de Petri, em *hardware* de lógica reconfigurável. Para isto, utilizou-se um *software* de captura no qual, a partir do gráfico do modelo em Rede de Petri, é gerado um arquivo de descrição no formato PNML (*Petri Net Markup Language*). A partir desta descrição, é gerado um arquivo de descrição de *hardware* no formato VHDL (VHSIC *Hardware Description Language*), que pode ser gravado em um circuito de lógica reconfigurável. Para possibilitar esta etapa, foi realizado o desenvolvimento de uma ferramenta que gera um arquivo em linguagem VHDL a partir da descrição no formato PNML. A ferramenta desenvolvida é descrita em detalhes, mostrando todas as etapas e critérios utilizados na conversão. Para validar o método, é mostrado um exemplo de aplicação com a implementação em FPGA (*Field Programmable Gate Arrow*), de uma Rede de Petri modelando uma planta industrial hipotética. Finalmente é feita uma comparação de desempenho entre o modelo executado em *hardware* e o modelo executado em *software*.

Palavras-chave: Redes de Petri, VHDL, Lógica Reconfigurável, FPGA.

ABSTRACT

ANTIQUEIRA, Perci Ayres. Implementation of Petri Nets Models in Reconfigurable Logic Hardware. 125 f. Dissertação – Programa de Pós-graduaçao em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná. Curitiba, 2011.

In this research work, a study of main types of hardware modeling tools was performed searching to verify the advantages of utilizing for modeling dynamic and concurrent systems and for its hardware implementation. It was observed that even though there are tools for this purpose, exists some points that may be worked out to facilitate access to this technology. So, was developed a method for facilitate implementation of systems modeled in Petri nets, in reconfigurable logic hardware. For that, was utilized a capture software where, from the graphic of the Petri net model, is generated a description in PNML (Petri Net Markup Language) format. From this description, is generated a hardware description file in VHDL (VHSIC Hardware Description Language) format, that may be loaded in a reconfigurable logic circuit. To make possible this stage, was performed the development of tool that generate a file in VHDL language from the description in PNML format. The developed tool is described in details, showing all stages and criteria utilized in the conversion. To validate the method, is showed an application example for this toll with the implementation in FPGA (Field Programmable Gate Arrow), of a Petri net modeling a hypothetic industrial plant. Finally, a performance comparison is made between the model executed in hardware and the model executed in software.

Keywords: Petri Nets, VHDL, Reconfigurable Logic, FPGA.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

PNML	*Petri Net Markup Language* - Linguagem de Marcação para Rede de Petri
VHDL	VHSIC *Hardware Description Language* - Linguagem de Descrição de *Hardware* VHSIC
FPGA	*Field Programmable Gate Arrow* - Matriz de Portas Programável em Campo
HDL	*Hardware Description Language* - Linguagem de Descrição de *Hardware*
ROM	*Read Only Memory* - Memória Somente de Leitura
PROM	*Programmable Read Only Memory* - Memória Somente de Leitura Programável
PLA	*Programmable Logic Array* - Matriz Lógica Programável
PAL	*Programmable Array Logic* - Lógica de Matriz Programável
ASIC	*Application-Specific Integrated Circuits* - Circuitos Integrados de Aplicação Específica
MPGA	*Mask-Programmable Gate Array* - Matriz de Portas Programável por Máscara
CPLD	*Complex Programmable Logic Device* - Dispositivo Lógico Programável Complexo
DSP	*Digital Signal Processor* - Processador de Sinais Digitais
LUT	*Look Up Table* - Tabela de Busca
RTL	*Register Transfer Level* - Nível de Transferência entre Registros
VHSIC	*Very High Speed Integrated Circuits* - Circuitos Integrados de Velocidade Muito Alta
FSM	*Finite State Machine* - Máquina de Estados Finitos

SUMÁRIO

1 INTRODUÇÃO

Este capítulo apresenta as motivações, objetivos, estruturação e método do trabalho de pesquisa.

1.1 MOTIVAÇÕES

Nesta seção são descritas as motivações para este trabalho e as vantagens da implementação de modelos de sistemas em hardware de lógica reconfigurável.

A utilização de sistemas de controle utilizando processos reativos e concorrentes é cada vez mais intensa na área industrial. Isto exige um método eficaz para a modelagem de tais processos, auxiliando no projeto, simulação, execução e otimização dos mesmos. Uma importante ferramenta para modelagem deste tipo de sistemas é a Rede de Petri, que permite a representação gráfica de sistemas reativos com processos paralelos e a simulação dinâmica dos processos modelados. Os modelos gerados podem ser simulados por meio de um *software* apropriado ou implementados em *hardware* (SILVA et al., 2007).

A implementação dos modelos em Redes de Petri em software apresenta contudo, uma certa limitação, quando se necessita representar vários processos ocorrendo com paralelismo real, sendo necessária a utilização de *treads* para simular este paralelismo. Esta limitação não ocorre quando o modelo é executado em hardware, onde cada processo é simulado de forma independente em uma parte do circuito e com paralelismo real. Neste contexto, a utilização de hardware de lógica reconfigurável como FPGAs, proporciona vantagens em relação a sua implementação em *software*, tais como:

- Uma vez que os processos podem operar de forma paralela, os modelos apresentam um comportamento mais próximo daquele dos sistemas reais.
- Menor tempo de latência, uma vez que os cálculos e decisões lógicas são executados diretamente pelo *hardware* utilizando lógica combinacional.
- Obtenção de alto desempenho com frequências de *clock* relativamente baixas.

- Circuitos compactos e de baixo consumo, adequados para sistemas embarcados.

Para realizar a implementação em *hardware*, existem atualmente algumas ferramentas para conversão do modelo gráfico da Rede de Petri para a linguagem de descrição de hardware ou HDL (*Hardware Description Language*). Neste ponto, porém, observa-se que existe ainda um trabalho a ser feito no sentido de facilitar o acesso a estas ferramentas, proporcionando um melhor entendimento dos métodos de conversão e dos códigos fontes, possibilitando futuras modificações e adaptações.

Assim, visando facilitar o processo de implementação física de modelos em Redes de Petri, bem como realizar um estudo do processo de conversão, possibilitando sua modificação ou adaptação, foi desenvolvida uma versão de ferramenta capaz de converter sistemas modelados em Rede de Petri para VHDL, permitindo assim a sua implementação em *hardware* de lógica reconfigurável como FPGAs.

1.2 OBJETIVOS

Esta seção descreve os objetivos gerais e específicos deste trabalho.

1.2.1 OBJETIVO GERAL

O objetivo deste trabalho de pesquisa é desenvolver um método para implementar modelos de sistemas em Redes de Petri, em *hardware* de lógica reconfigurável, comprovar sua correta operação tanto em simulação quanto na implementação física e avaliar as vantagens desta implementação quando comparada à implementação em *software*.

1.2.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos deste trabalho são descritos a seguir:

- Desenvolver um método para implementar modelos em Rede de Petri em *hardware* de lógica reconfigurável.
- Desenvolver uma ferramenta que permita a conversão do modelo gráfico para VHDL.
- Realizar um teste da ferramenta gerada, aplicada a um modelo de planta industrial hipotética.
- Realizar a simulação do hardware antes de sua implementação.

- Implementar o modelo em um *hardware* de lógica reconfigurável (FPGA), verificando sua correta operação.

- Realizar um comparação de desenpenho entre o modelo em *hardware* um modelo equivalente em *software*.

1.3 ESTRUTURA DA DISSERTAÇÃO

Esta dissertação está organizada em seis capítulos. O capítulo 1 apresenta uma introdução sobre motivações, objetivos, estruturação e método do trabalho de pesquisa. O capítulo 2 trata das FPGAs e das ferramentas de especificação de *hardware*, destacando-se as Redes de Petri e sua implementação em *hardware* de Lógica Reconfigurável, permitindo concluir sobre as dificuldades e possíveis contribuições para facilitar a utilização desta tecnologia. O capítulo 3 descreve em detalhes o desenvolvimento do método proposto. O capítulo 4 relata os experimentos relativos à utilização da ferramenta desenvolvida para obtenção do código VHDL e os resultados obtidos. O capítulo 5 relata os experimentos relativos à implementação em *hardware* do código gerado comparando também seu desempenho com um modelo em *software* . E, finalmente, o capítulo 6 apresenta a discussão dos resultados, as conclusões do trabalho e as propostas de trabalhos futuros.

1.4 MÉTODO DE TRABALHO

A método seguido durante desenvolvimento desta dissertação, constitui-se das seguintes etapas:

- Etapa 1: Realizar a revisão bibliográfica sobre ferramentas de especificação de *hardware*.

- Etapa 2: Identificar as vantagens da utilização de Redes de Petri para a modelagem de sistemas reativos e concorrentes.

- Etapa 3: Realizar a revisão bibliográfica sobre Redes de Petri e de meios utilizados para a implementação de modelos em *hardware*.

- Etapa 4: Identificar as dificuldades de acesso à tecnologia de implementação de modelos em Redes de Petri em *hardware*.

- Etapa 5: Desenvolver um método e uma ferramenta para facilitar a implementação de modelos de sistemas em *hardware* de lógica reconfigurável.

- Etapa 6: Realizar ensaios de validação do método e da ferramenta desenvolvida.
- Etapa 7: Comparar o desempenho do modelo em *hardware* com um modelo em *software*.
- Etapa 8: Elaborar a dissertação de mestrado.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo trata das FPGAs e das ferramentas de especificação de *hardware*, destacando-se as Redes de Petri e sua implementação em *hardware* de Lógica Reconfigurável, permitindo concluir sobre as dificuldades e possíveis contribuições para facilitar a utilização desta tecnologia.

2.1 DISPOSITIVOS LÓGICOS PROGRAMÁVEIS

Os dispositivos lógicos programáveis evoluíram a partir de circuitos utilizando memórias ROM (*Read Only Memory*) até o estado da arte representado atualmente pelas FPGAs.

Os primeiros dispositivos lógicos programáveis utilizavam as memórias PROM (*Programmable Read Only Memory*) e foram uma alternativa ao uso de memórias ROM que exigiam a criação de máscaras de gravação durante a fabricação. Para implementar lógica utilizando PROMs, os terminais de endereçamento são usados com entradas e os terminais de dados como saídas, compondo assim várias tabelas verdade. Os principais problemas de se utilizar PROMs são a limitação do número de entradas e sua lentidão, com tempos de acesso em torno de 40ns.

A PLA (*Programmable Logic Array*) foi uma solução para as limitações de velocidade e de número de entradas da PROM. A PLA consiste de um grande número de entradas conectadas a um plano que permite diferentes conexões de lógica AND. As saídas do plano de lógica AND são ligadas a um plano de lógica OR, possibilitando realizar a descrição lógica pela soma de produtos. Tanto nas entradas como nas saídas são disponibilizados inversores para obtenção do NOT lógico, possibilitando o mapeamento de todas as lógicas possíveis como na PROM, mas com muito mais entradas e respostas muito mais rápidas (ZEIDMAN, 2002).

A PAL (*Programmable Array Logic*) é uma variação da PLA, na qual o plano de lógica OR tem um tamanho reduzido baseando-se na lei de DeMorgan que estabelece que A OR B = NOT(NOT A AND NOT B). Isto significa que se pode criar toda lógica necessária quando se dispõe de inversores na entrada e na saída, com um amplo plano AND ou um amplo plano OR, mas não necessariamente com ambos. A redução do plano OR possibilitou a inclusão de

outros elementos básicos como multiplexadores, OU-exclusivos e *latches*, que facilitaram sua utilização em máquinas de estado. Isto, aliado a uma alta velocidade, possibilitou que estes dispositivos substituíssem a maioria dos circuitos lógicos em muitos projetos e a criação de controladores de alta velocidade em lógica programável. O aumento da complexidade incentivou o nascimento de linguagens de descrição de *hardware* como ABEL, CUPL e PALASM, precursoras do Verilog e do VHDL (ZEIDMAN, 2002).

O ASIC (*Application-Specific Integrated Circuits*) e a MPGA (*Mask-Programmable Gate Array*) não é um dispositivo programável mas é precursor no desenvolvimento que levou às FPGAs. Estes dispositivos são formados por uma estrutura de milhões de transistores que são conectados pela aplicação de máscaras durante o processo de fabricação, de acordo com o circuito que se deseja obter. Devido à proximidade dos componentes que são agrupados em células, obtêm-se dispositivos de alta densidade e alta velocidade trabalhando com frequências de centenas de megahertz. Estes dispositivos porém apresentam um maior custo e tempo de projeto, sendo mais adequados para produção em larga escala (ZEIDMAN, 2002).

O CPLD (*Complex Programmable Logic Devices*) e a FPGA foram criados para suprir o *gap* existente entre os dispositivos PAL e os dispositivos ASIC, permitindo o desenvolvimento de circuitos mais densos e mais rápidos, sem a necessidade de programação por máscaras durante a fabricação. Os CPLDs têm velocidade semelhante às PALs, mas com circuitos mais complexos. A FPGA apresenta a complexidade das MPGAs, porém são programáveis (ZEIDMAN, 2002).

2.2 FIELD PROGRAMMABLE GATE ARROW - FPGA

As FPGAs, introduzidas em 1985, a princípio para atuar como solução mais densa (maior número de portas lógicas por área) para lógica associada em sistemas, tornou-se uma alternativa para implementação de sistemas digitais, sendo hoje utilizada como dispositivo principal, e muitas vezes único, de alguns sistemas. As FPGAs apresentam vantagens em relação aos microprocessadores e DSPs, principalmente no que diz respeito à velocidade de execução de operações lógicas e aritméticas. Isto ocorre em função da possibilidade de configuração do *hardware* especificamente para uma determinada função, utilizando a quantidade exata de bits necessários e processos paralelos os quais se conectam muitas vezes de forma combinacional. Por outro lado, as FPGAs apresentam, como desvantagem, menor flexibilidade quando comparadas ao microprocessador e ao DSP (*Digital Signal Processor*).

As FPGAs também apresentam vantagens em relação aos circuitos do tipo ASIC e MPGA, como menor tempo para mercado, sem custo de engenharia recorrente para a fabricação e pro-

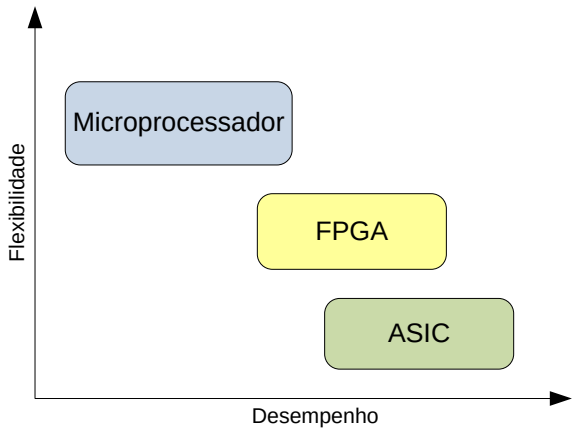


Figura 1: Comparação Desempenho X Flexibilidade da FPGA.
Fonte: Adaptado de Andrade (2006, p. 20).

gramabilidade, permitindo atualização ou modificações através de programação no sistema já implantado. Através da reconfiguração do dispositivo pode-se corrigir erros de projeto e novas funções podem ser adicionadas ou o circuito pode ser redirecionado para outra aplicação. Como desvantagens, quando comparadas com ASICs e MPGAs, as FPGAs custam mais por chip para realizar certa função, assim não são adequadas para volume de produção extremamente alto, e apesar de apresentarem alto desempenho, sua velocidade é ligeiramente inferior. Assim, pode-se dizer que a FPGA se enquadra como solução em situações onde se necessita de alto desempenho e paralelismo, juntamente com a possibilidade de reconfiguração, estando em uma posição intermediária entre os microprocessadores e os ASICs, como pode ser observado na figura 1 (ANDRADE, 2006).

AS FPGAs são compostas por três tipos de elementos: Blocos lógicos, Roteamento e Blocos de I/O. Assim, uma FPGA consiste em uma matriz de milhares de Blocos Lógicos reconfiguráveis que podem ser ligados com outros Blocos Lógicos ou com Blocos de I/O por meio das entidades de roteamento. Devido à arquitetura de granularidade fina, as operações lógicas podem ser realizadas utilizando-se apenas a quantidade de bits necessária para uma determinada aplicação. Assim, uma aplicação que exija um somador de 6 bits não terá que utilizar, por exemplo, um somador de 32 bits. A figura 2 mostra um diagrama simplificado da arquitetura de uma FPGA (GOKHALE; GRAHAM, 2005).

2.2.1 BLOCOS LÓGICOS

Conforme descrito em Gokhale e Graham (2005), os blocos lógicos são geralmente formados por uma lógica combinacional programável, por um *flip-flop* e por uma lógica de *carry* como mostrado na figura 3. Como pode-se observar, existe também a possibilidade de seleção

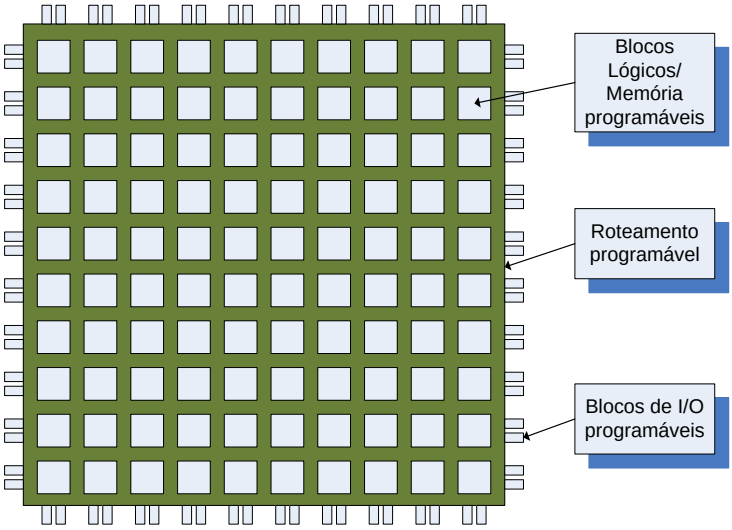


Figura 2: Arquitetura de uma FPGA genérica.
Fonte: Adaptado de Gokhale e Graham (2005, p. 13).

configurável entre a saída diretamente da parte combinacional ou através do *flip-flop*.

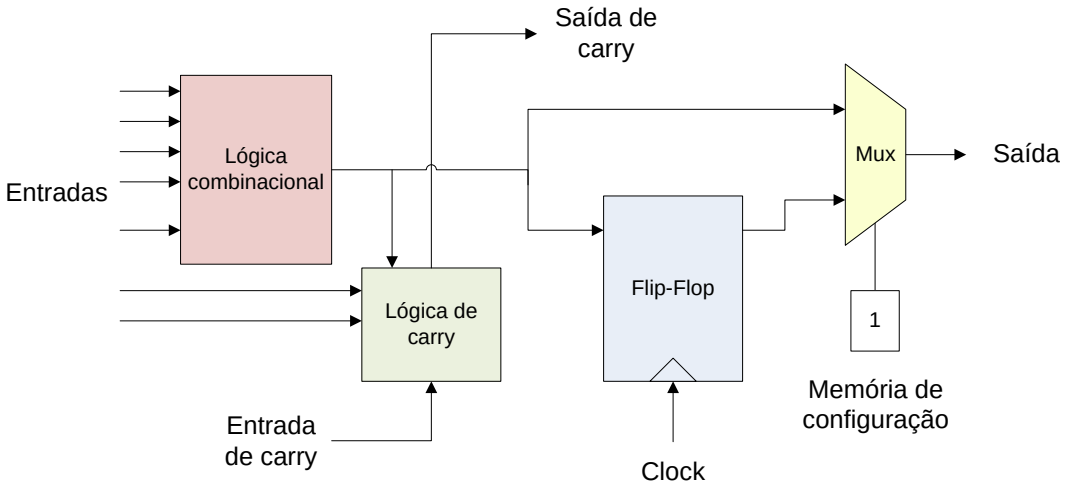


Figura 3: Um bloco lógico reconfigurável.
Fonte: Adaptado de Gokhale e Graham (2005, p. 14).

Os blocos lógicos configuráveis das FPGAs comerciais apresentam uma grande flexibilidade. Os flip-flop's, por exemplo, podem ser configurados com *set* e *reset* síncrono ou assíncrono e podem ser sensíveis à borda de subida ou de descida. A lógica de carry pode ser aumentada para facilitar multiplicação ou ser composta por somadores completos. A parte de lógica combinacional é normalmente formada por uma LUT (*Look Up Table*). A função lógica obtida depende dos valores da tabela verdade que são carregados na memória. Por exemplo, na figura 4 vemos uma LUT configurada para a função AND das três entradas A, B e C ou seja a saída só terá o valor 1 quando todas as entradas tiverem o valor 1 (GOKHALE; GRAHAM, 2005).

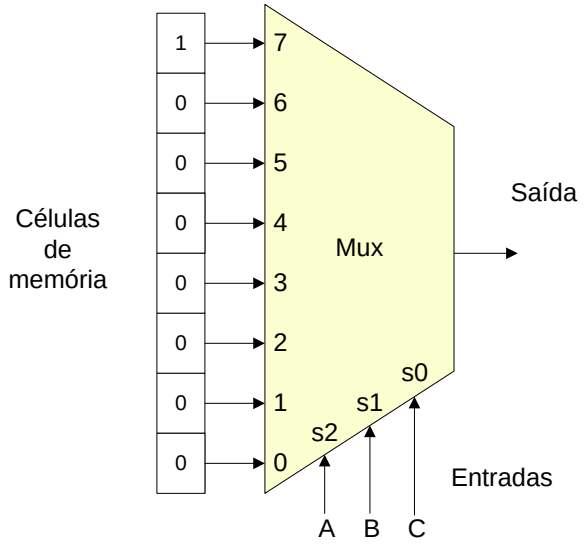


Figura 4: Uma LUT configurada para a função E.
Fonte: Adaptado de Gokhale e Graham (2005, p. 15).

Para obter um melhor aproveitamento de área, as FPGAs utilizam LUTs com 4 entradas permitindo que muitas funções lógicas sejam executadas em uma única LUT, diminuindo os atrasos devido a roteamento. Por sua vez, são utilizados *clusters* de blocos lógicos, possibilitando a implementação de funções mais complexas com ligações curtas. Na figura 5 é mostrado um *cluster* com 2 blocos lógicos. A tabela 1 mostra algumas configurações disponíveis comercialmente.

2.2.2 ROTEAMENTO

Roteamento é o processo de interligação dos vários elementos de uma FPGA, que é feito através de três subtipos do elemento básico: o multiplexador, o transistor e o buffer tri-state. Internamente aos *clusters* de blocos lógicos, já existe um roteamento que tem a função de interligar as LUTs com as entradas e saídas ou com outras LUTs, passando ou não através de flip-flop's, de acordo com a função lógica desejada. Porém, é a ligação entre os diversos *clusters* que caracteriza o tipo de roteamento global da FPGA, o qual é usualmente dos tipos ilha, linha longa ou celular.

No roteamento tipo ilha os *clusters* são conectados a blocos de conexão através de linhas horizontais e verticais, compondo, assim, uma matriz de conexão. Neste caso os *clusters* lógicos são interligados por segmentos de conexão como pode ser visto na figura 6 (GOKHALE; GRAHAM, 2005).

No roteamento tipo linha longa a interligação dos *clusters* lógicos é feita por canais horizontais e verticais com comprimento igual ao comprimento do chip. Alguns canais adicionais

Tabela 1: Algumas FPGAs comerciais

Empresa	Familia	Ano	Tecnologia	Blocos Lógicos
Altera	Cyclone	2002	130 nm	2.910 a 20.060
	Stratix	2002	130 nm	10.570 a 79.040
	Stratix GX	2003	130 nm	10.570 a 41.250
	Cyclone II	2004	90 nm	4.608 a 68.416
	Stratix II	2004	90 nm	15.600 a 179.400
	Stratix II GX	2005	90 nm	33.88 a 132.540
	Stratix III	2006	65 nm	47.500 a 337.500
	Arria GX	2007	90 nm	21.580 a 90.220
	Cyclone III	2007	65 nm	5.136 a 198.464
	Stratix IV	2008	40 nm	72.600 a 531.200
	Arria II GX	2009	40 nm	42.959 a 244.188
	Cyclone IV	2009	60 nm	6.272 a 149.760
	Arria II GZ	2010	40 nm	224.000 a 348.500
	Stratix V	2010	28 nm	236.000 a 695.000
	Arria V	2011	28 nm	75.000 a 503.500
	Cyclone V	2011	28 nm	25.000 a 300.000
Xilinks	Virtex-4	2004	90 nm	13.824 a 200.448
	Virtex-5	2006	65 nm	30.720 a 331.776
	Spartan-6	2009	45 nm	3.840 a 147.443
	Virtex-6	2009	40 nm	74.496 a 566.784
	Artix-7	2010	28 nm	8.000 a 360.000
	Kintex-7	2010	28 nm	65.600 a 477.760
	Virtex-7	2010	28 nm	284.000 a 1.954.560

Fonte: Autoria própria

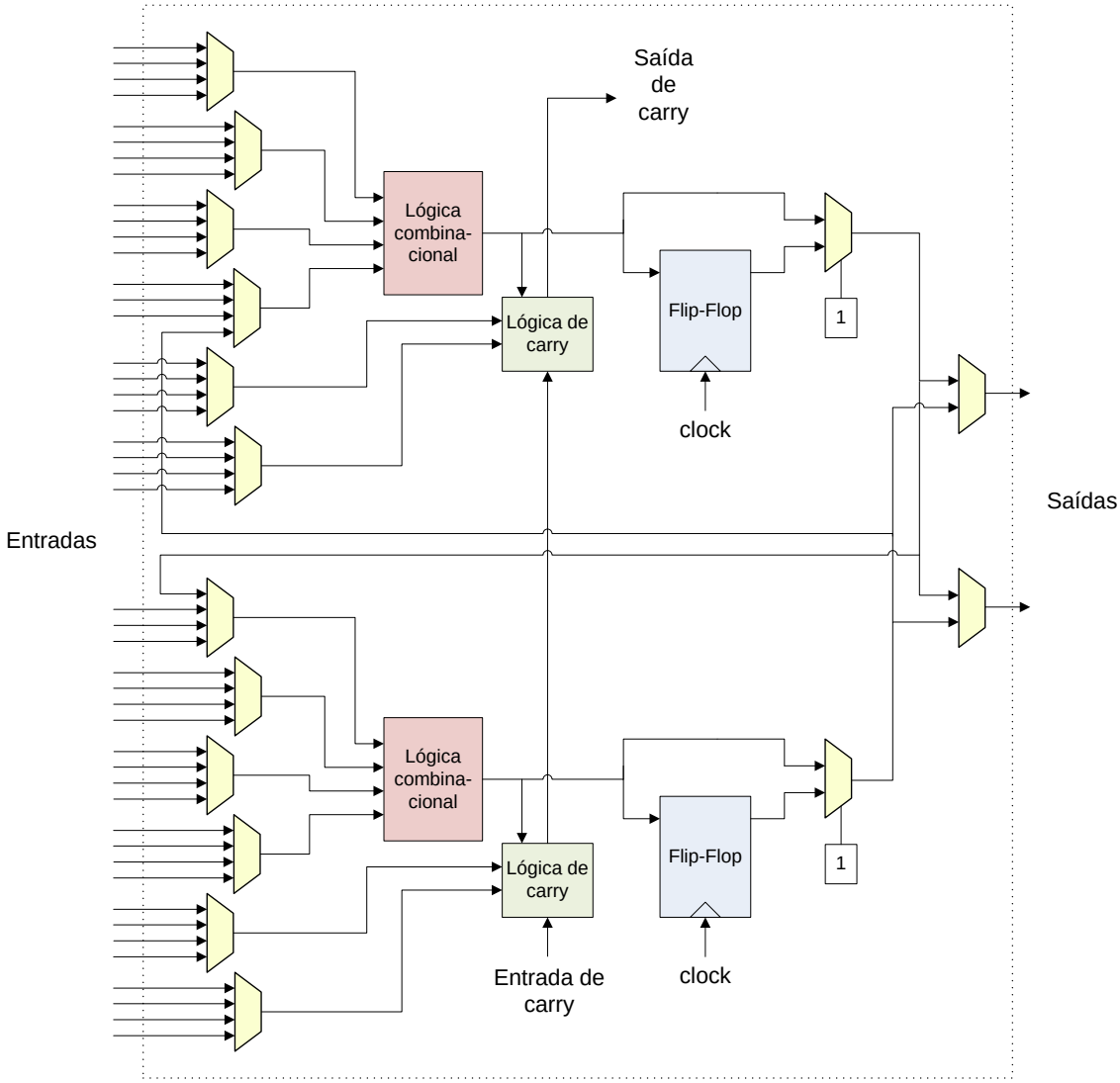


Figura 5: *Cluster* com 2 blocos lógicos.
Fonte: Adaptado de Gokhale e Graham (2005, p. 18).

são fornecidos para interligar *clusters* próximos de forma direta. A figura 7 mostra este tipo de roteamento.

No roteamento tipo celular a maioria das ligações entre os *clusters* lógicos é realizada de forma local. Para isto, os blocos lógicos trabalham em uma granularidade fina e utilizam parte de sua estrutura para a função de roteamento. A figura 8 mostra este tipo de ligação.

2.2.3 BLOCOS DE I/O

Os blocos de I/O fazem a ligação do circuito interno com o pino da FPGA, possibilitando configurá-lo como entrada, saída ou em aberto. Para isto, utilizam buffers tri-state para a saída e buffer simples para a entrada. Possui ainda *flip-flops* que permitem a ação de registro dos sinais. A figura 9 mostra a arquitetura básica deste tipo de bloco (GOKHALE; GRAHAM, 2005).

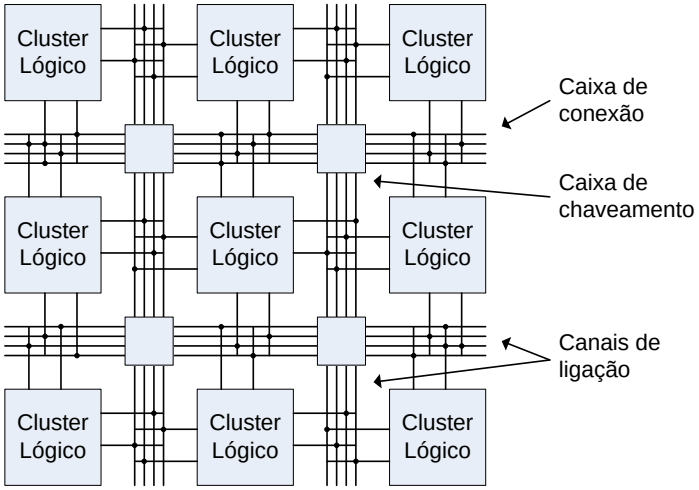


Figura 6: Roteamento tipo ilha.
Fonte: Adaptado de Gokhale e Graham (2005, p. 18).

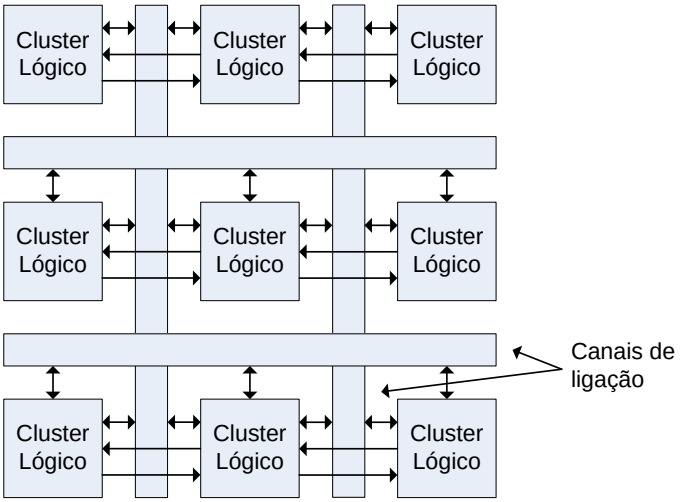


Figura 7: Roteamento tipo linha longa.
Fonte: Adaptado de Gokhale e Graham (2005, p. 19).

Dois artigos descrevendo as vantagens de FPGAs com arquitetura 3D e com arquitetura amórfica podem ser vistos em (LIN et al., 2006) e (LIN, 2008).

2.3 FERRAMENTAS DE ESPECIFICAÇÃO DE HARDWARE

O desenvolvimento de sistemas de software e hardware contempla as etapas de Levantamento de Requisitos, Análise/Modelagem, Desenvolvimento/Programação e Testes. Nesta seção são descritas ferramentas textuais utilizadas nas etapas de Desenvolvimento e de Teste e ferramentas gráficas utilizadas nas etapas de Análise e Modelagem dos projetos de *hardware*.

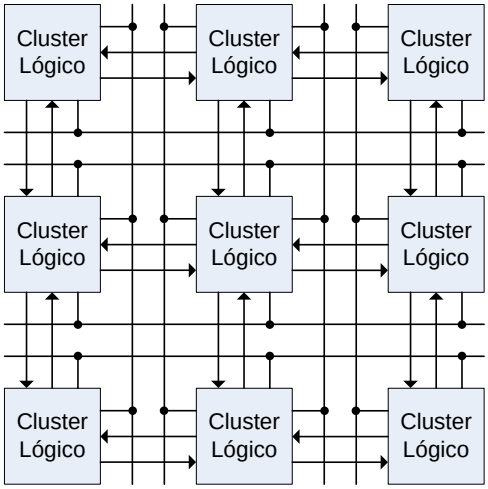


Figura 8: Roteamento tipo celular.
Fonte: Adaptado de Gokhale e Graham (2005, p. 20).

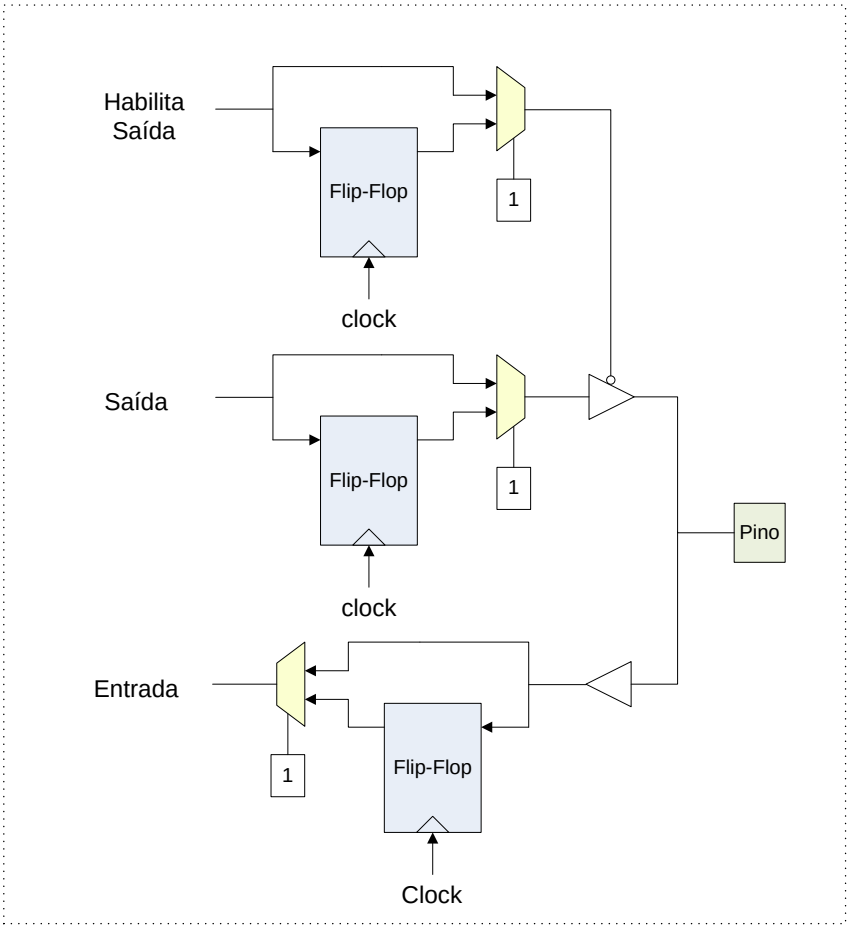


Figura 9: Arquitetura de um bloco de I/O.
Fonte: Adaptado de Gokhale e Graham (2005, p. 22).

2.3.1 LINGUAGENS TEXTUAIS DE ESPECIFICAÇÃO DE HARDWARE

As linguagens textuais de especificação de *hardware* podem ser agrupadas em duas categorias: Aquelas que descrevem diretamente os dispositivos e processos de *hardware*, chamadas

de HDLs (*Hardware Description Languages*) tais como Verilog e VHDL e aquelas baseadas nas linguagens de programação, tais como V++, SpecC, JavaTime, SystemC++ que descrevem comportamentos em um nível maior de abstração, podendo, em um passo posterior, produzir um texto em HDL correspondente ao código digitado. Nesta seção mostra-se uma visão geral das principais linguagens utilizadas para especificação de *hardware* (BUNKER; GOPALAKRISHNAN; MCKEE, 2004).

2.3.1.1 Verilog

A linguagem Verilog (IEEE, 2006) surgiu em 1984 e em 1995 tornou-se um padrão IEEE-1364-1995, sendo posteriormente atualizada em 2001 pelo padrão IEEE1364-2001 e em 2005 pelo padrão IEEE1364-2005. Verilog permite o projeto de *hardware* digital, analógico ou híbrido, em vários níveis de abstração, como comportamental, RTL (*Register Transfer Level*) ou lógico. Sua sintaxe é similar a linguagens como C e PASCAL, utilizando mecanismos de controle de fluxo como *if* e *while* e com os mesmos operadores, com exceção daqueles para incremento e decremento (++ e - -), que não estão disponíveis em Verilog. O tipo de dados *wire* conecta dois pontos do circuito e o tipo *reg* é utilizado para armazenar um valor (SOARES, 2005).

Neste âmbito, a descrição do *hardware* é feita em blocos chamados *Modules* que possuem um nome, uma lista de portas de entrada e saída e um processo delimitado pelas palavras *begin/end* ao invés de chaves. Os blocos podem ser do tipo *initial bloc* que executam apenas uma vez ou um *always block* que estão sempre em execução e possuem uma lista de sensibilidade, na qual aparecem sinais ou variáveis que ao serem alterados, determinam que o processo seja executado. Dentro de um bloco, o processo ocorre de forma sequencial, sendo que vários blocos podem ser criados trabalhando de forma paralela, aumentando a velocidade do processamento. Para reutilização de código, são utilizadas *tasks* e *functions*, sendo que as primeiras permitem atraso e as últimas não, conforme descrito em Soares (2005).

A figura 10 mostra um exemplo do diagrama esquemático de um flip-flop D com os pinos de entrada D e CLK e os pinos de saída Q e Q'. Na mesma figura aprece o código em Verilog correspondente. O sinal de *clock* (CLK) é colocado na lista de sensibilidade e quando ocorre uma borda de subida, o processo é executado fazendo com que a saída (Q) receba o valor da entrada (D) e com que a saída (Q') receba o complemento deste valor.

Um exemplo de aplicação da linguagem Verilog utilizada para descrição de *hardware* sintetizável pode ser encontrado em (GILLENWATER et al., 2008).

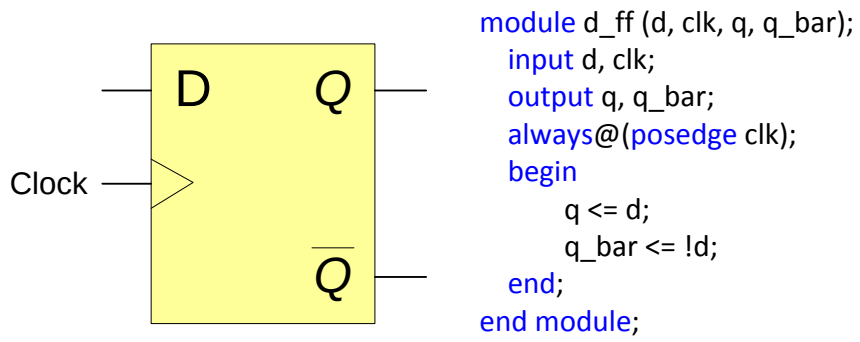


Figura 10: Diagrama do Flip-Flop D e código em Verilog.
Fonte: Autoria própria

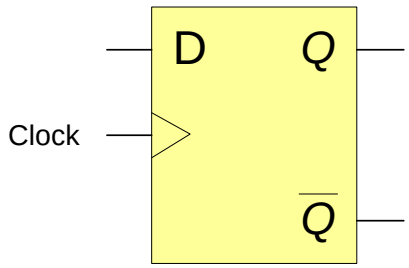
2.3.1.2 VHDL

VHDL (VHSIC *Hardware Description Language*) (IEEE, 2009b), onde VHSIC é a sigla para *Very High Speed Integrated Circuits*, foi criada no Departamento de Defesa dos Estados Unidos em meados de 1980. Foi padronizada pelo IEEE1076-1987 teve uma versão mais moderna lançada em 1993, sendo atualizada em 2000 e 2002 e em 2008 de acordo com o padrão IEEE1076-2008 (SOARES, 2005).

A linguagem VHDL é baseada em Fluxo de Dados e permite a descrição de sistemas contendo processos paralelos, sendo fortemente baseada na linguagem ADA de programação, tanto no conceito como na sintaxe, sendo como ADA, fortemente tipado e não *case sensitive* e tendo estruturas para manusear o paralelismo inerente aos projetos de *hardware*. Contudo, estas estruturas (*process*) diferem das estruturas concorrentes de ADA (*tasks*). Há também aspectos em VHDL que não são encontrados em ADA, tal como o conjunto estendido de operadores Booleanos, incluindo *nand* e *nor* para representar diretamente operações comuns em *hardware* e também possibilidade de indexar matrizes (de forma ascendente ou descendente), pois ambas as convenções são utilizadas em *hardware*.

Um projeto em VHDL é constituído de uma *Entity* que descreve as portas de entrada e saída de um componente e uma *Architecture* que descreve seu comportamento. As portas ou ligações entre componentes são disponíveis como Sinais, que podem ser utilizados como operando de atribuição na codificação da *Architecture*. Os comandos dentro de uma *Architecture* são executados concorrentemente. Quando se deseja que alguns comandos sejam executados sequencialmente, deve-se codificá-los dentro de estruturas *Process*, as quais ficam dentro de uma *Architecture* e possuem uma lista de sensibilidade para sua execução. A figura 11 mostra o texto em VHDL correspondente a um Flip-Flop tipo D.

Uma vantagem de se utilizar a linguagem VHDL é que ela permite que o comportamento




```
library ieee;
use ieee.std_logic_1164.all;
-----------------------------------------------
entity dff is
port(D, clock: in std_logic;
     Q, Q_bar: out std_logic);
end dff;
-----------------------------------------------
architecture behv of dff is
begin
  process(data_in,clock)
  begin
    if(clock'event and clock='1')then
      Q <= D;
      Q_bar <= not D;
    end if;
  end process;
end behv;
```

Figura 11: Diagrama do Flip-Flop D e código em VHDL.
Fonte: Autoria própria

do sistema desenvolvido seja descrito (modelado) e verificado (simulado) antes das ferramentas de síntese traduzirem o projeto para *hardware* real. (SOARES, 2005).

2.3.1.3 Objective VHDL

Objective VHDL (OVHDL) eleva o nível de abstração no qual as atividades de desenvolvimento ocorrem. Esta versão orientada a objetos do VHDL permite a criação de classes, objetos, métodos, herança, encapsulamento, abstração, polimorfismo e implementa mecanismos para lidar com estados, transição de estados, tempo de vida e comunicação entre objetos e arbitragem de requisição. O *Objective VHDL* suporta concorrência, utilizando expressões de guarda e políticas de escalonamento, que podem ser por prioridade estática, *round robin* e *round robin melhorado*.Ainda, pode-se criar uma política de escalonamento própria (RADETZKI, 2000).

2.3.1.4 SystemVerilog

A Linguagem *SystemVerilog* (IEEE, 2009a) é uma extensão da linguagem Verilog, que surgiu em 2002 e tornou-se um padrão IEEE1800-2005, sendo posteriormente atualizada em 2009 pelo padrão IEEE1800-2009. Esta linguagem fornece uma convergência entre as tarefas de projeto e verificação e foi adotada pela empresa ACCELLERA e difere de Verilog pela definição e verificação de blocos em alto nível de abstração com escrita de assertivas e criação de bancos de testes. Além disto, proporciona características de linguagens de programação como

classes orientadas a objetos, passagem de variáveis por referência, tipos de dados abstratos e definidos pelo usuário, semáforos, caixas de correio e eventos para comunicação entre processos (SUTHERLAND, 2002).

2.3.1.5 V++

V++ é uma linguagem de descrição de *hardware* síncrona, baseada em classes, métodos e elementos de dados como C++, tendo suas declarações uma sintaxe semelhante ao Verilog. Os objetos criados se comunicam através de canais de mensagens. Apesar de ser uma linguagem supostamente orientada a objetos, não provê Herança nem Polimorfismo. A semântica de *hardware* é um mapeamento computacional formal, onde cada objeto corresponde a uma de máquina de estado finita ou FSM (*Finite State Machine*), funcionando em paralelo e se comunicando de forma concorrente através de mensagens em um modelo orientado a objeto (RADETZKI, 2000).

2.3.1.6 SpecC

A linguagem SpecC foi criada para o desevolvimento de sistemas, incluindo *hardware* e *software*. SpecC é baseada no ANSI-C e apresenta um conjunto de ferramentas adicionais para modelagem de *hardware* tais como hierarquia estrutural, concorrência, sincronização, tratamento de exceções e temporização. Um programa em SpecC consiste de um conjunto de *behaviors* (comportamentos), *channels* (canais) e *interfaces* (interfaces) com *ports* (portas). Os *behaviors* são blocos contendo computação ativa, enquanto *channels* e *interfaces* são blocos passivos encapsulando comunicação. A especificação segue o estilo de diagramas de blocos, formando uma rede hierárquica de canais interligados e conectados a portas (PADERBORN; MUELLER, 2002).

2.3.1.7 JavaTime

JavaTime é um conjunto de ferramentas que transformam JAVA em um modelo computacional chamado *Abstractabel Synchronous Reactive* (ASR) model, adequado para especificação e síntese de sistemas reativos embarcados. Uma política de uso que proíbe, por exemplo, a instanciação dinâmica de objetos deve ser utilizada, já que a capacidade de JAVA vai além do que pode ser expresso como ASR. Assim, os objetos devem ser instanciados durante uma fase inicial. Como a natureza de *hardware* é estática e JAVA é baseado em mecanismos dinâmicos, sugere-se que o comportamento de um objeto seja sintetizado em um método separado. Assim,

o objeto propriamente dito não é sintetizado, mas fornece uma plataforma para especificação, definindo uma política de uso ao invés de uma biblioteca de programação (RADETZKI, 2000).

2.3.1.8 SystemC++

SystemC++ é uma extensão do C++ para objetos concorrentes e sua comunicação. A criação de objetos dinâmicos é suportada em *hardware* pela ligação controlada por *hardware* de objetos a um conjunto estático de recursos. O código do *hardware* gerado é similar a um código de *software*, onde um sistema em tempo de execução é gerado em um modelo de orientação a objeto de forma que muitos conceitos de *software* podem ser implementados em *hardware*. Porém a arquitetura não é baseada em memória global, mas em pequenas unidades concorrentes, onde os objetos são mapeados de forma semelhante a um processador de aplicação específica com múltiplas unidades de execução, rodando um programa em *firmware*. Isto pode gerar um cabeçalho exagerado para gerenciamento da alocação dinâmica de recursos, desencorajando seu uso em *hardware* embarcado (RADETZKI, 2000).

Um novo paradigma de programação com orientação a aspectos utilizando System C++, pode ser visto em (DéHARBE; MEDEIROS, 2006).

2.3.1.9 AutoPilot

AutoPilot é uma ferramenta de alto nível, baseada em técnicas desenvolvidas na UCLA e licenciadas para a empresa AutoESL. *AutoPilot* transforma uma descrição de funcionalidade em C, C++ ou SystemC com alto nível de abstração em uma descrição de *hardware* em Verilog ou VHDL em nível RTL para um determinado dispositivo FPGA ou ASIC. Isto elimina o passo demorado e sujeito a erros de criação da implementação RTL. Também cria um acurado modelo de simulação em *SystemC*, para o resultado sintetizado. O uso de ferramentas de alto nível permite um aumento de produtividade para desenvolvedores que trabalham com FPGA e permite que outros desenvolvedores familiarizados com DSP, possam utilizar FPGAs sem precisar conhecer sobre especificação RTL.

Foram realizados estudo comparativos sobre as vantagens da utilização de Autopilot (BDTI, 2009). No artigo de Cong et al. (2011) pode ser visto um exemplo de utilização do AutoPilot que demonstra a efetividade da conversão C para FPGA em multiplos domínios de aplicação.

2.3.2 LINGUAGENS VISUAIS DE ESPECIFICAÇÃO DE HARDWARE

As notações visuais compreendem um pequeno número de linguagens de especificação, e são tipicamente fáceis de aprender. Pode-se dividir esta abordagem naquelas oriundas de práticas de Engenharia de *Software*, da descrição de *hardware* tradicional e da Engenharia de Sistemas (BUNKER; GOPALAKRISHNAN; MCKEE, 2004).

2.3.2.1 Statecharts

A *Statechart* estende o formalismo de máquinas de estado e o convencional diagrama de transição de estados, essencialmente em três elementos que lidam com noções de hierarquia, concorrência e comunicação. Estes elementos são importantes para a especificação e desenvolvimento de sistemas reativos complexos, tais como, sistemas multicomputadores em tempo real, protocolos de comunicação e unidades de controle digital. *Statechart* é uma linguagem de descrição altamente estruturada, onde os estados podem ser compostos sequencialmente ou concorrentemente para formar estados abstratos. Uma ferramenta de suporte à hierarquia facilita a navegação mesmo em grandes modelos. Como os usuários usualmente utilizam máquinas de estado em seus modelos mentais de desenvolvimento, as *Statecharts* são facilmente aprendidas e integradas na prática industrial. A *Statechart* pode ser utilizada sozinha ou como parte de uma metodologia mais geral de projeto, que lide com outros aspectos do sistema, como decomposição funcional e especificação de fluxo de dados (HAREL, 1987).

Como ferramenta de desenvolvimento utilizando Statecharts, foi desenvolvido o sistema *STATEMATE*, que permite a especificação, análise, desenvolvimento e documentação de complexos sistemas reativos, com geração automática de código em C ou VHDL (HAREL et al., 1990). A Figura 12 mostra um exemplo de modelagem de processo de chamada telefônica, utilizando *Statechart*. No artigo de Mura et al. (2007), é apresentada uma ferramenta para conversão de especificações escritas em *Statechart* para modelos comportamentais em SystemC, permitindo a modelagem de sistemas em vários níveis de abstração e precisão, além de proporcionar rapidez de desenvolvimento, simplicidade e reusabilidade.

2.3.2.2 Message Sequence Charts - MSC

A MSC é um mecanismo de especificação de cenários que descreve padrões de interação entre processos e objetos, sendo útil nos estágios iniciais do desenvolvimento de sistemas. Esta linguagem encontrou utilidade em muitas metodologias de projeto, como na UML, onde tem uma variante chamada de diagrama de sequências. A *International Telecommunication Union*

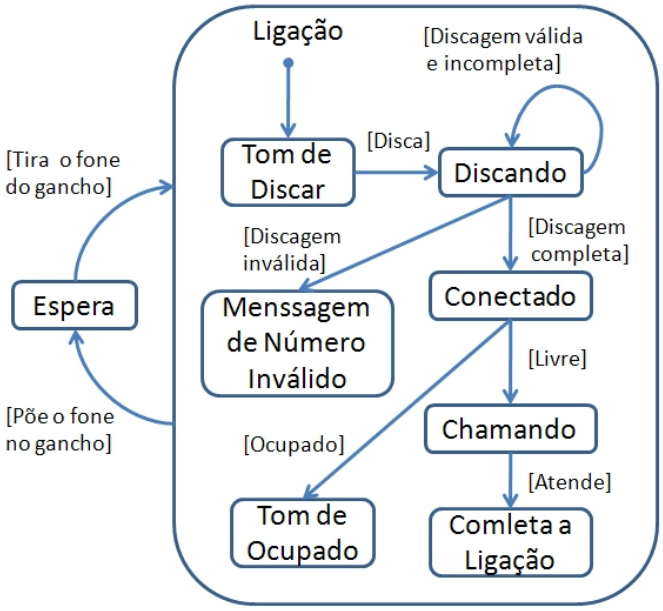


Figura 12: Modelo de chamada telefônica em Statechart.
Fonte: Autoria própria

(ITU) mantém uma definição formal e semântica para MSC (ITU-T. . . , 1999).

Em desenvolvimento orientado a objeto, o usuário especifica os casos de uso do sistema usando diagramas de sequências (MSCs). Em um passo de modelagem posterior o comportamento de uma classe é descrito por um diagrama de estados (*Statechart*). Então os objetos são implementados como código, utilizando-se ferramentas como ObjectTime e Rhapsody.

Porém, o MSC possui uma ordem parcial semântica fraca que torna impossível capturar certos comportamentos. Em função disto, foi criada uma extensão do MSC, chamada de *live sequency charts* (LSCs), que pode ser vista como uma versão multimodal do MSC, com várias formas de distinguir entre comportamentos possível, necessário ou proibido. O poder da LSCs é comparável à lógica temporal e *Statecharts* exceto pela profundidade de alternâncias de modalidades aninhadas. Utilizando-se LSCs pode ser possível uma transição automática entre os requerimentos capturados pelos casos de uso e as especificações executáveis. Ferramentas como Play-Engine (HAREL; MARELLY, 2003), tornam possível ver as LSCs como especificações executáveis (HAREL; THIAGARAJAN, 2003). Na Figura 13 é mostrado um exemplo de modelagem de processo de chamada telefônica, utilizando um diagrama de sequências.

2.3.2.3 Specification and Description Language - SDL

A *Specification and Description Language* (SDL) é um padrão ITU promovido pela SDL *Forum Society*, amplamente utilizada na área de telecomunicações para o desenvolvimento de

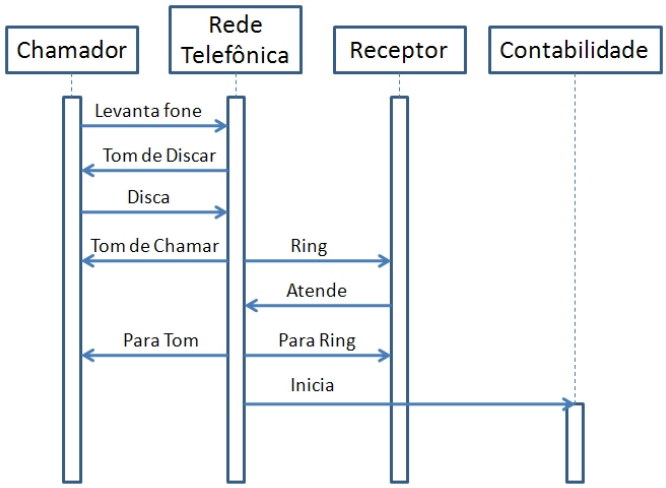


Figura 13: Modelo de chamada telefônica em Diagama de Sequências.
Fonte: Adaptado de Harel e Thiagarajan (2003, p. 13).

sistemas complexos. A SDL é projetada para o desenvolvimento de sistemas reativos, concorrentes, de tempo-real, distribuídos e heterogêneos (BRÆK, 1996). Com SDL, um desenvolvedor pode especificar um sistema completo em sua estrutura e comportamento, com seus diversos componentes e a comunicação entre eles (ITU-T..., 1993). Os elementos básicos de uma descrição SDL são:

- Estrutura: Decomposição hierárquica com sistema, blocos e processo, como os blocos principais de construção e caminhos de sinais e canais para conectá-los.
- Comportamento: Definição de um comportamento pelos processos, empregando máquinas de estado finito estendidas que se comunicam.
- Dados: Tipos de dados abstratos como a linguagem de descrição de dados usada para a especificação de mensagem na área de telecomunicações.
- Comunicação: Assíncrona com parâmetros opcionais e chamadas a procedimentos remotos.

A figura 14 mostra um exemplo de um bloco de um sistema de abertura de porta especificado em SDL. Este bloco é composto de três processos que se comunicam entre sí e trabalham de forma concorrente. O processo Leitura do Cartão recebe os pulsos de um leitor de cartão magnético e os transforma em um código. O bloco Controle de Acesso recebe o código e verifica se o mesmo está cadastrado. Em caso afirmativo, um sinal de admissão e enviado. O processo Porta recebe o sinal Admite e envia um comando para a abertura da porta. Após 5 segundos a porta é fechada. A figura 15 mostra a estrutura do processo Porta.

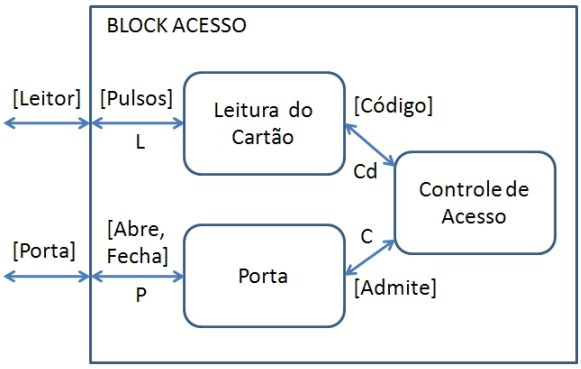


Figura 14: Bloco de um sistema SDL
Fonte: Adaptado de Bræk (1996, p. 1590).

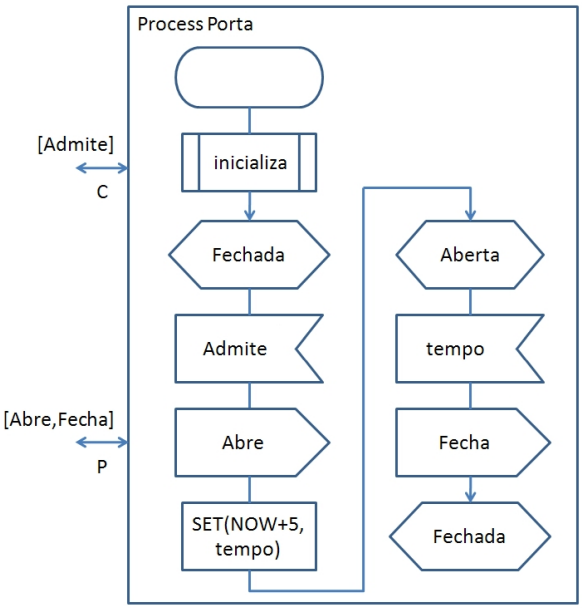


Figura 15: Processo PORTA
Fonte: Adaptado de Bræk (1996, p. 1592).

2.3.2.4 UML

A *Unified Modeling Language* (UML) vem se tornando um padrão industrial de especificação, visualização e documentação de sistemas com *software* orientado a objetos, permitindo o acompanhamento de todos os estágios do projeto e com vários tipos de diagramas para a modelagem do *software* em diferentes níveis de abstração. A modelagem em UML é baseada em casos de uso que dão origem a cenários de comportamentos. Estes cenários são representados por uma variante do MSC *Message Sequence Charts* chamada de diagramas de sequências. Quando existe um conjunto suficiente de cenários, são criadas *stacharts* para os objetos participantes. A geração das *Statecharts* a partir dos cenários, descritos em *Message Sequence Charts* - MSC, pode ser feita de forma automática através de várias ferramentas (MäKINEN; SYSTä, 2000). As *Statecharts* podem então ser convertidas em VHDL para a implementação do *hardware* em FPGA ou ASIC.

2.3.2.5 Redes de Petri

Devido à relevância das Redes de Petri para esta dissertação, as mesmas são tratadas de forma mais aprofundada na seção 2.4.

2.3.3 FERRAMENTA UTILIZADA NESTE TRABALHO

Após analisadas as diversas ferramentas para especificação de *hardware*, decidiu-se optar pela modelagem utilizando Redes de Petri em razão de sua forma de representação ser bastante intuitiva e adequada à representação de processos industriais, permitindo ainda um tratamento matemático facilitado em função de sua representação por matrizes. Quanto à linguagem de descrição de *hardware* optou-se por se utilizar o VHDL em função de sua grande utilização atual e também por ser de maior domínio por parte da equipe de pesquisa. A tabela 2 mostra um comparativo entre as diversas ferramentas de especificação de *software* pesquisadas neste trabalho.

2.4 REDES DE PETRI

A teoria das Redes de Petri foi apresentada na tese de doutoramento de C. A. Petri em 1962 (PETRI, 1962), tendo sido posteriormente desenvolvida por A. W. Holt e outros pesquisadores, como descrito em (MURATA, 1989). Trata-se de uma ferramenta gráfica de modelagem e simulação adequada para sistemas paralelos, assíncronos e não-determinísticos, orientados a eventos. Assim, permite a representação dos diversos subprocessos, que podem ocorrer de

Tabela 2: Comparativo entre ferramentas de especificação de *hardware*

Ferramenta	Nível de abstração	Representação gráfica	Simulação gráfica	Formalismo matemático	Domínio do desenvolvedor
Verilog	Baixo	-	-	-	-
VHDL	Baixo	-	-	-	Sim
Objective VHDL	Alto	-	-	-	-
System Verilog	Alto	-	-	-	-
V++	Alto	-	-	-	-
SpecC	Alto	-	-	-	-
JavaTime	Alto	-	-	-	-
SystemC++	Alto	-	-	-	-
AutoPilot	Alto	-	-	-	-
Statecharts	Alto	Sim	-	-	-
MSC	Alto	Sim	-	-	-
SDL	Alto	Sim	-	-	-
UML	Alto	Sim	-	-	-
Redes de Petri	Alto	Sim	Sim	Sim	Sim

Fonte: Autoria própria

forma paralela e da interação entre eles. Além disto, possui um formalismo matemático que possibilita diversos tipos de análise como, de alcançabilidade e de existência de *deadlocks*. A representação gráfica das Redes de Petri permite a visualização do processo e da comunicação entre seus elementos.

Na Figura 16 é mostrado um exemplo de modelagem de um processo de produção de canetas utilizando Redes de Petri. Os Lugares, representados por círculos, correspondem às variáveis de estado e as Transições, representadas por traços ou barras, correspondem às ações ou eventos do sistema. Estes dois elementos são interligados por Arcos Dirigidos que representam a relação entre as ações e as variáveis de estado. Os arcos dirigidos podem possuir um peso (inteiro positivo). Um arco com peso k equivale a um conjunto de k arcos em paralelo. Quando o arco não possui indicação subentende-se peso 1. As marcas (bolinhas pretas) representam estados dos lugares. Quando existem marcas suficientes nos lugares de entrada de uma transição, esta é disparada, retirando marcas destes lugares de entrada e enviando marcas para os lugares de saída, segundo o peso dos arcos relacionados. O número de marcas em cada lugar da Rede de Petri em um determinado instante é chamado de Marcação.

No exemplo, a produção da carga e a produção do invólucro são atividades que ocorrem em paralelo. Na figura, foram produzidas 3 canetas (marcas no lugar P5) e uma marca no lugar Pi indica que a máquina está disponível e assim que houver matéria prima (marcas no lugar P0) uma nova caneta será produzida (MACIEL; LINS; CUNHA, 1996).

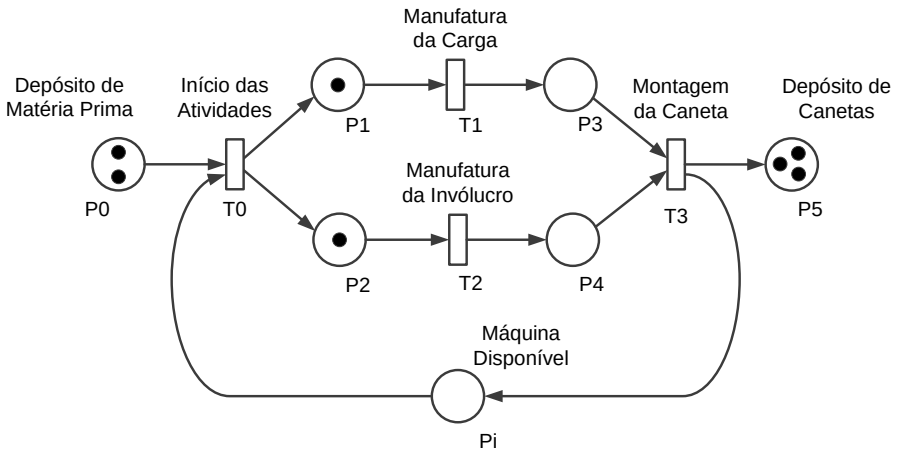


Figura 16: Fábrica de Canetas em Rede de Petri.
Fonte: Adaptado de Maciel, Lins e Cunha (1996, p. 40).

2.4.1 DEFINIÇÃO FORMAL DAS REDES DE PETRI

Uma definição formal das Redes de Petri é apresentada em Murata (1989), conforme descrito a seguir:
As Redes de Petri são uma 5-tupla, $PN = (P,T,F,W,M_0)$ onde: $P = \{p_1, p_2, ..., p_m\}$ é um conjunto finito de lugares, $T = \{t_1, t_2, ..., t_n\}$ é um conjunto finito de transições, $F \subseteq (PxT) \cup (TxP)$ é um conjunto de arcos (relação de fluxo), $W : F \rightarrow \{1,2,3,...\}$ é uma função de peso, $M_0 : P \rightarrow \{0,1,2,3,...\}$ é a marcação inicial, $P \cap T = \emptyset$ e $P \cup T \neq \emptyset$.

A rede de petri sem marcação inicial é representada por $N = (P,T,F,W)$. Uma Rede de Petri marcada é denota (N,M_0).

2.4.2 REGRAS DE DISPARO DAS TRANSIÇÕES

A seguir, são comentadas as regras de disparo das transições, juntamente com alguns tipos especiais de transição, conforme descrito em Murata (1989).

Uma transição t está habilitada para disparo quando cada lugar de entrada p de t contém pelo menos w(p,t) marcas, sendo w(p,t) o peso do arco ligando p a t, o que é descrito formalmente como $\forall p \in P : M(p) \geq W(p,t) \rightarrow M[t \in T >$. Uma transição habilitada para disparo pode ou não disparar, dependendo se o evento associado a ela realmente vier a ocorrer. O disparo de uma transição t remove w(p,t) marcas dos lugares de entrada e adiciona w(t,p) marcas nos lugares de saída, onde w(t,p) é o peso do arco ligando t a p. A figura 17 mostra um exemplo do disparo de uma transição t.

Uma transição que não apresenta nenhum lugar de entrada é chamada de transição *source*

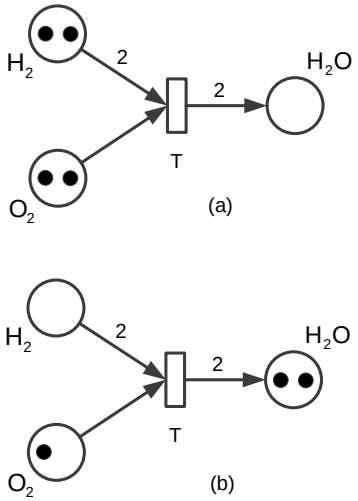


Figura 17: Disparo de uma transição. Em (a) antes e em (b) após o disparo.
Fonte: Adaptado de Murata (1989, p. 543).

(fonte) , pois está sempre habilitada e fornece marcas livremente para os lugares de saída. Uma transição que não está ligada a nenhum lugar de saída, é chamada de transição *sink* (dreno) e pode consumir marcas de forma ilimitada, bastando, para isto, estar habilitada. Quando o lugar de entrada de uma transição t é também o seu lugar de saída, têm-se o que é chamado de *self-loop*. Uma Rede de Petri é dita uma rede pura quando não possui nenhum *self-loop* e é dita rede ordinária quando o peso de todos os arcos é 1.

Uma Rede de Petri é dita de capacidade infinita quando não há um limite máximo para o número de marcas que cada lugar p desta rede pode receber. Caso contrário, ou seja, quando há um limite máximo k de marcas para determinados lugares p, diz-se que a rede é de capacidade finita ou ainda que a rede é limitada.

As Redes de Petri limitadas são formalmente representadas por uma 6-tupla, $PN = (P,T,F,K,W,M_0)$ onde: $K : F \rightarrow \{1,2,3,...\}$ é uma função de capacidade dos lugares.

Para as Redes de Petri limitadas, a regra para habilitação do disparo da transição tem uma condição adicional, que é a de que nenhum de seus lugares de saída tenha a capacidade $K(p)$ excedida após o disparo sendo então chamada de regra de transição estrita e definida formalmente como $\forall p \in P : W(p,t) \leq M(p) \leq K(p) - W(t,p) \rightarrow M[t \in T >$. A regra de transição para redes de capacidade infinita é também chamada de regra de transição fraca.

O disparo das transições habilitadas pode ocorrer de forma simultânea gerando uma novo estado de marcação para a Rede de Petri. Temos então, o que é chamado de passo, sendo definido como um subconjunto de ações, distribuídos pela rede e que podem ocorrer simultaneamente.

2.4.3 MODELOS COM REDES DE PETRI

A seguir, são apresentados alguns modelos básicos de Redes de Petri, os quais podem fazer parte de redes maiores e mais complexas, conforme descrito em Murata (1989).

2.4.3.1 Máquina de Estados Finitos

Uma máquina de estados finitos é uma subclasse de Rede de Petri onde os lugares representam os estados e todas as transições possuem apenas um arco de entrada e apenas um arco de saída. A cada transição está associada uma condição externa que determina seu disparo caso a mesma esteja habilitada. Um lugar pode habilitar simultaneamente várias transições, o que é chamado de conflito ou escolha. Na figura 18 têm-se um exemplo de máquina de estados modelando uma máquina de venda de balas, onde o lugares com dois ou mais arcos de saída representam escolha entre vários caminhos de acordo com o valor da moeda depositada. Uma máquina de estados não possibilita a modelagem de sincronização de eventos paralelos que necessita de uma transição com dois ou mais arcos de entrada (MURATA, 1989).

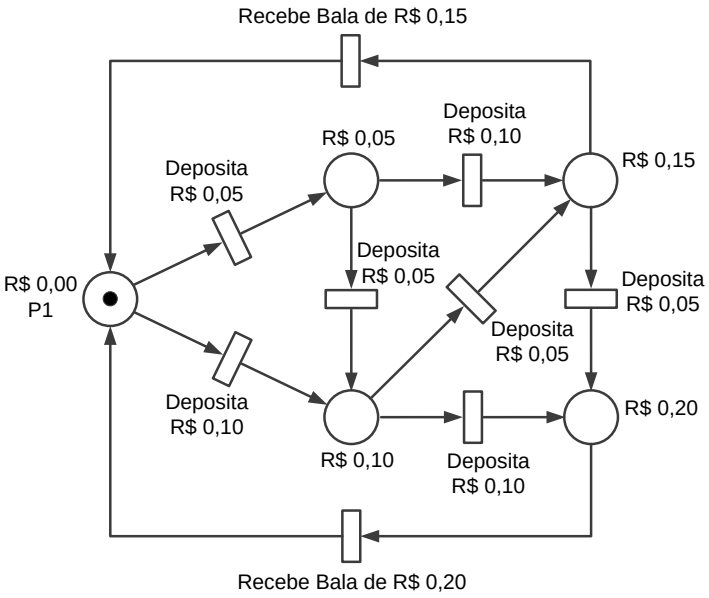


Figura 18: Máquina de estados com Rede de Petri.
Fonte: Adaptado de Murata (1989, p. 544).

2.4.3.2 Atividades Paralelas

As atividades paralelas ou concorrência podem ser representadas pelas Redes de Petri. Para isto, estas atividades devem ser causalmente independentes, ou seja, o disparo da transição que inicia uma determinada atividade não impede o disparo das transições que iniciam as outras

atividades concorrentes e podem ocorrer inclusive de forma simultânea. Um exemplo de concorrência é mostrado na figura 19, onde o disparo das transições t1 e t4 marcam o início e o final das atividades paralelas. Observa-se que todos os lugares desta rede possuem um único arco de entrada e um único arco de saída. Por isto, este tipo de rede é uma subclasse das Redes de Petri, chamada de Grafo Marcado (MURATA, 1989).

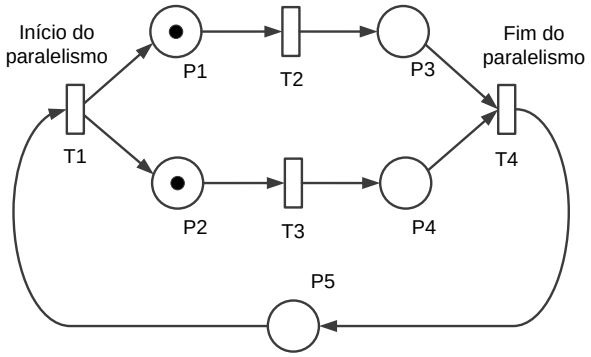


Figura 19: Atividades paralelas em uma rede tipo Grafo Marcado.
Fonte: Adaptado de Murata (1989, p. 545).

2.4.3.3 Conflito

Dois ou mais eventos estão em conflito quando apenas um dentre eles pode ocorrer. Isto pode ser representado por um lugar com uma única marca habilitando o disparo de várias transições. Aquela que ocorrer primeiro, retira a marca do lugar e desabilita as demais. Uma rede Grafo Marcado não é capaz de representar conflito uma vez que cada lugar possui apenas um arco de saída (MURATA, 1989).

2.4.3.4 Distribuição

O início de várias atividades paralelas pode ser representado por um a transição ligada a vários lugares. Quando a transição é disparada ela envia marcas para os diversos lugares habilitando diversas atividades de forma simultânea. Este tipo de representação é chamado de distribuição (MURATA, 1989).

2.4.3.5 Sincronismo

Quando várias atividades paralelas devem todas ser concluídas para que uma outra atividade possa ocorrer, temos o que é chamado de sincronismo. O sincronismo pode ser representado por uma transição com vários arcos de entrada ligados aos lugares finais dos diversos processos pa-

ralelos. Quando todos tiverem uma marca a transição é habilitada e o processo pode prosseguir (MURATA, 1989).

2.4.3.6 Confusão

Quando as situações de concorrência e conflito ocorrem juntas em Rede de Petri, têm-se o que é chamado de confusão. Na figura 20 temos dois tipos de confusão. Em (a) temos a confusão simétrica onde t1 e t3 são concorrentes e t2 está em conflito com ambas. Em (b) temos a confusão assimétrica onde t1 e t2 são a princípio concorrentes, mas, se t1 disparar primeiro, ocorre um conflito entre t1 e t3 (MURATA, 1989).

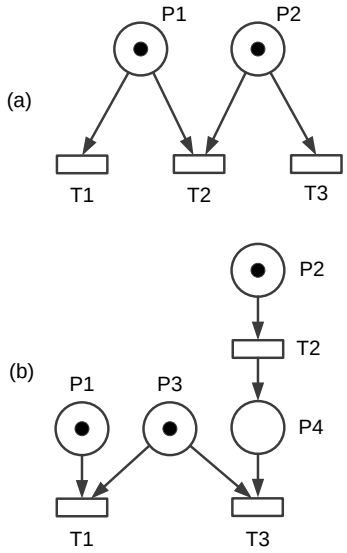


Figura 20: Dois tipos de confusão:(a) Simétrica. (b) Assimétrica.
Fonte: Adaptado de Murata (1989, p. 545).

2.4.3.7 Computação de fluxo de dados

Uma Rede de Petri pode representar a computação de fluxo de dados. Neste caso, as fichas ou marcas representam dados disponíveis para serem processados ou o resultado de um processo. As transições representam o processo ou operação realizada com os dados. Vários processos podem ser realizados em paralelo. Na figura 21 mostra-se um exemplo deste tipo de modelagem (MURATA, 1989).

Convem observar, que inclusão de controles (como if) e atribuição (como a + b), provocam um a redução na capacidade de análise da rede utilizando os métodos matemáticos convencionais.

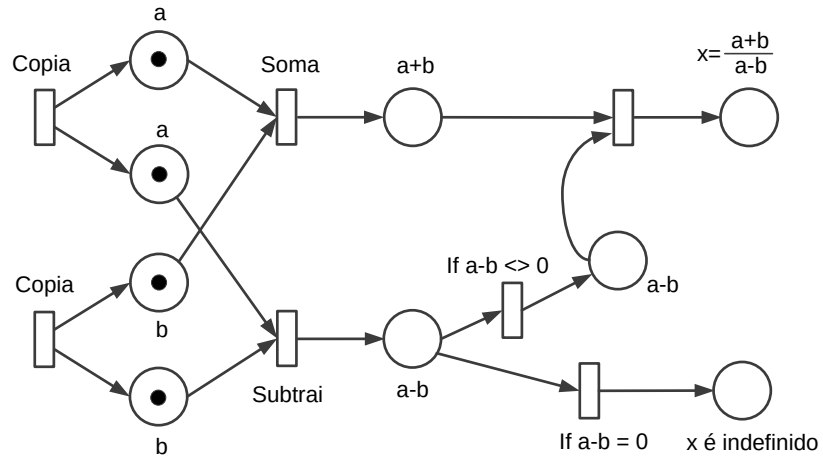


Figura 21: Computação de fluxo de dados.
Fonte: Adaptado de Murata (1989, p. 545).

2.4.3.8 Protocolos de comunicação

As Redes de Petri são bastante apropriadas para a modelagem de sistemas de comunicação onde propriedades tais como vitalidade e segurança são importantes para garantir o correto intercâmbio de informação. A figura 22 mostra a modelagem de um sistema de comunicação entre dois processos. O processo P1 gera uma informação que é enviada para o processo P2 e aguarda um sinal de reconhecimento. O processo P2 recebe a informação e envia o sinal de reconhecimento e utiliza a informação recebida (MURATA, 1989).

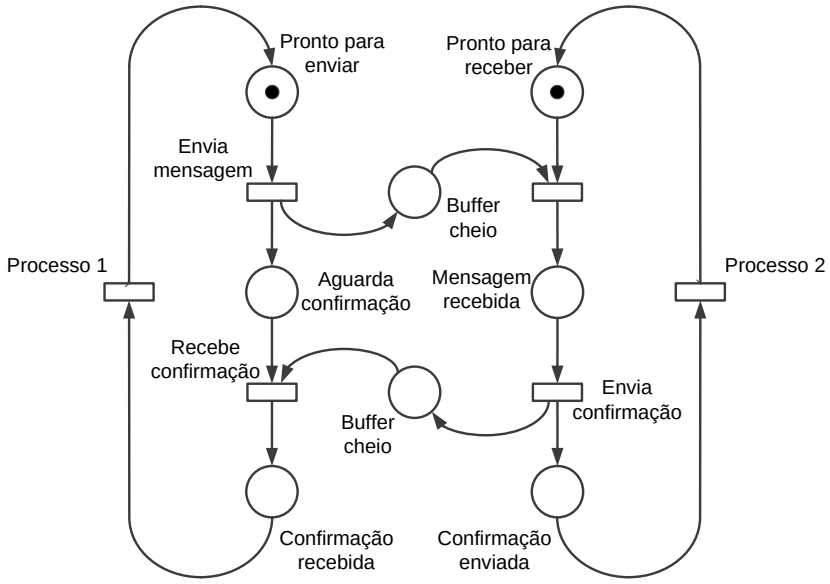


Figura 22: Um protocolo de comunicação.
Fonte: Adaptado de Murata (1989, p. 546).

2.4.3.9 Controle de Sincronização

As redes de petri possibilitam a modelagem de uma série de mecanismos de controle de sincronização utilizados em sistemas multiprocessador ou processamento distribuído onde alguns recursos como, por exemplo, memória, devam ser compartilhados. Alguns mecanismos utilizados são: Exclusão Mútua, Leitura e Escrita e Produtor Consumidor. A figura 23 mostra uma exemplo de um controle de Leitura e Escrita onde até k processos podem ler uma memória compartilhada. Porém quando um dos processos estiver escrevendo na memória, os demais não podem acessá-la. Isto ocorre devido ao peso k do arco entre p3 e t2, que retira a possibilidade de novos disparos das transições t1 e t2 até que ocorra a transição t4, indicando o final da operação de escrita (MURATA, 1989).

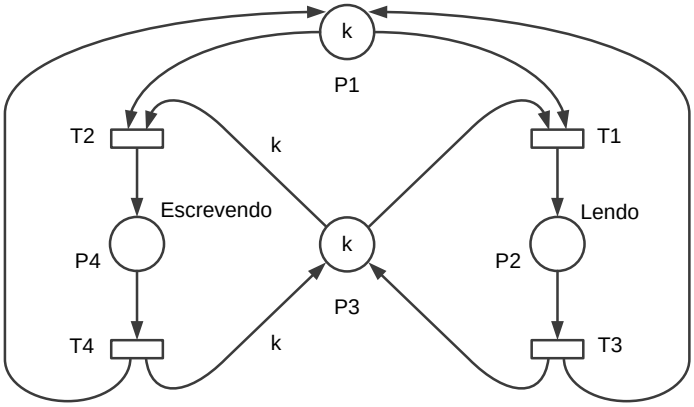


Figura 23: Sincronização de Leitura e Escrita em memória.
Fonte: Adaptado de Murata (1989, p. 546).

2.4.3.10 Produtor Consumidor com Prioridade

As Redes de Petri representam com facilidade um sistema produtor consumidor. No exemplo da figura 24 têm-se dois processos produtores e dois processos consumidores. Os produtores A e B estão constantemente produzindo itens que são armazenados nos Buffers A e B. O consumidor A tem prioridade e pode consumir um item sempre que haja uma marca no Buffer A. O consumidor B só poderá consumir um item se existir uma marca no Buffer b e se o Buffer A estiver vazio. A verificação de um lugar vazio é possível através da utilização de um arco inibidor entre p3 e t7. Este arco, tracejado e terminando em um círculo, habilita a transição t7 apenas se p3 estiver vazio. O arco inibidor é uma extensão das Redes de Petri e não transfere nenhuma marca por ocasião do disparo da transição habilitada (MURATA, 1989).

Existem atualmente várias ferramentas para a modelagem de sistemas utilizando Rede de Petri, que podem ser consultadas em:

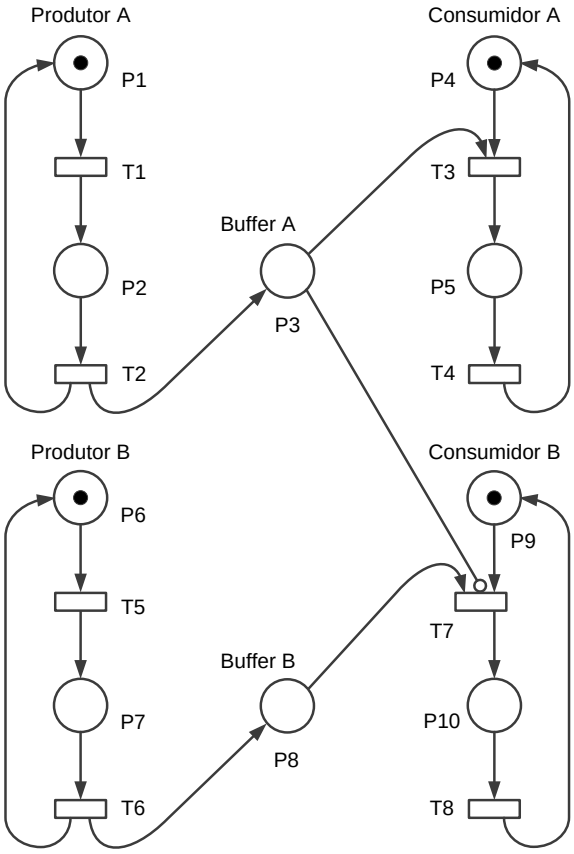


Figura 24: Sistema Produtor Consumidor.
Fonte: Adaptado de Murata (1989, p. 546).

Petri Net World Website (Hamburgo).
Site: http://www.informatik.uni-hamburg.de/TGI/PetriNets.

2.5 REDES DE PETRI EM FPGA

Uma Rede de Petri pode representar um espaço de estados bastante grande, mesmo para sistemas simples. Assim, os simuladores de Redes de Petri em *software*, apresentam geralmente um tempo de execução relativamente longo, especialmente para modelos de sistemas mais complexos. Neste contexto, uma abordagem interessante é sua implementação em FPGA, que pode ser configurada para executar vários subprocessos em paralelo, o que permite a execução em tempo real do processo modelado, mesmo para um grande número de variáveis de estado. Assim, um dos objetivos deste trabalho de pesquisa é a obtenção de um método para facilitar esta implementação.

2.6 TRABALHOS RELACIONADOS

Existem atualmente alguns métodos de modelagem de Redes de Petri em circuitos com FPGA, dos quais sao descritos alguns detalhes a seguir.

Em *An FPGA Implementation of the Petri net Firing Algorithm* (BUNDELL, 1997) foi utilizada uma técnica de modelagem onde apenas o processo de transição é executado em uma FPGA, através de módulos básicos que são interfaceados com um PC, o qual por sua vez, simula a Rede de Petri. O *software* executado no PC permite que a entrada gráfica dos componentes da Rede de Petri simulada, bem como de seus parâmetros. Os módulos são então carregados na FPGA e inicia-se a simulação, que também é exibida de forma gráfica.

Neste trabalho o modelo é simulado parcialmente na FPGA, sendo necessário um controle por parte de um PC, o que demanda um tempo maior de execução, apesar de ocorrer um paralelismo no disparo das transições. Também não são exibidos maiores detalhes sobre a implementação e sobre o *software* de interfaceamento, dificultando sua reprodução.

No trabalho *Asynchronous microprocessors: From high level model to FPGA implementation* (LLOYD et al., 1999) foi implementado um microprocessador assíncrono utilizando FPGA e modelado em Rede de Petri. O microprocessador, chamado ADLX, tem uma estrutura de micropipeline e foi baseado em um microprocessador síncrono chamado DLX devido à sua simplicidade. A modelagem em Rede de Petri foi feita primeiramente em um alto nível de abstração sendo posteriormente feito um refinamento para modelagem de cada estágio do processo. Em seguida, o modelo foi convertido para VHDL utilizando-se a ferramenta Petrify. O código em VHDL foi então transferido para uma FPGA, permitindo uma rápida prototipagem e teste do dispositivo desenvolvido. No artigo são analisadas questões sobre projetos síncronos e assíncronos utilizando-se FPGAs e são realizados testes e análise de desempenho do microprocessador criado.

Neste trabalho foi utilizada a ferramenta Petrify para implementação de *hardware* em fpga. Porém, o processo utiliza muitas etapas e também não esclarece o funcionamento da ferramenta.

Em *Implementing a Petri Net Specification In a FPGA Using VHDL* (SOTO; PEREIRA, 2001) foi utilizado um processo de modelagem onde a Rede de Petri é constituída pela união de diversos blocos padrão, contendo cada bloco um lugar conectado a uma transição com entradas e saídas alternativas de forma que, dependendo do tipo de ligação, pode-se utilizar apenas o lugar, apenas a transição ou ambos. Este tipo de configuração tem como objetivo uma redução no número de ligações necessárias e um melhor aproveitamento dos blocos lógicos reconfiguráveis integrando o máximo número de elementos numa FPGA.

Apesar desta implementação contemplar de forma fidedigna o paralelismo das Redes de Petri, uma questão que não está muito clara nesta técnica é o tratamento de conflitos. Além disto, a ligação entre blocos que compõem a Rede de Petri deve ser feita de forma manual, o que pode ser difícil para modelos mais complexos.

Em *Implementation of safety critical logic controller by means of FPGA* (WEGRZYN, 2003) apresenta dois métodos de síntese em FPGA de controladores lógicos discretos para sistemas críticos. Em um deles o sistema é modelado em Redes de Petri e, através de um processo automático, é gerado o código em VHDL para ser carregado na FPGA utilizada. No outro, o sistema é descrito em termos de uma tabela de estados possíveis que são convertidos manualmente para VHDL utilizando-se comandos *if...then*. Neste artigo, comenta-se sobre a vantagem da implementação de controladores lógicos em FPGA, onde o circuito é ajustado exatamente aos requisitos, apresentando um desempenho quase ótimo, onde as saídas respondem às entradas com alta velocidade, de uma forma previsível, repetitiva e sem *glitches*. Além disto, os projetos normalmente utilizam um único *chip*, permitindo verificação formal e também que atualizações sejam realizadas com facilidade.

Apesar de demonstrar a aplicabilidade das Redes de Petri para modelar controladores lógicos, este trabalho não descreve com clareza o *software* e o processo de conversão entre Redes de Petri VHDL fazendo-se apenas referência a algumas regras utilizadas.

Em *A Hardware Implementation of Petri Nets Models* (SUDACEVSCHI; GUTULEAC; ABABII, 2004) é mostrada uma metodologia de implementação de Redes de Petri em FPGAs, através da utilização de módulos de lugares e transições descritos em Verilog. Estes módulos podem ser interligados em um grande número, possibilitando a modelagem de redes maiores. Além destes módulos existe também um módulo de memória onde são armazenadas as marcações que representam os estados do sistema e um módulo de sincronismo para controle da sequência de disparos e de operações de escrita e leitura em memória. Neste artigo também é comentada a vantagem de utilização de FPGAs para a implementação em *hardware*, das Redes de Petri, em função do paralelismo e da possibilidade de reconfiguração.

Trata-se de um trabalho interessante que demonstra uma maneira de interfacear o modelo de Rede de Petri em *hardware* com um PC. Porém, exige que a Rede de Petri seja configurada manualmente, utilizando-se uma combinação de blocos básicos.

Em *Modeling by Petri Nets* (KUBáTOVá, 2005) é realizada uma análise de três tipos de Rede de Petri na modelagem da comunicação entre duas impressoras que trabalham paralelamente e um sistema de controle de envio de dados a serem impressos. O sistema é modelado primeiramente utilizando-se a Rede de Petri elementar. Uma segunda modelagem utiliza a Rede

de Petri colorida com arcos constantes e uma terceira forma mostra a modelagem com rede de petri colorida. Além desta análise, são mostrados dois exemplos de aplicação: O clássico Jantar dos Filósofos e um controle de tráfego em uma estrada de ferro. Estes modelos foram implementados em *hardware* utilizando-se FPGA. Para isto, foram utilizadas ferramentas de modelagem de Redes de Petri(*Design/CPN* ou *JARP editor*), que geram um arquivo correspondente em PNML. Este arquivo foi convertido de forma automática para um bitstream de gravação da FPGA. Porém, a forma como é realizada esta conversão não fica clara.

Em *A Hardware Implementation of Safe Petri Net Models* (SUDACEVSCHI; ABABII; NEGURA, 2006) é feita a modelagem de um sistema com Rede de Petri segura. Inicialmente, utiliza-se uma ferramenta para modelagem da Rede de Petri (*VPNP Tools*) onde a mesma é desenhada e convertida em objetos. Em seguida, outra ferramenta de codificação (*HDLCS Tool*) transforma os modelos em código HDL. Numa terceira etapa o código HDL é compilado na ferramenta (*MAX+plus II Tools*) e carregado em uma FPGA através de uma interface JTAG. Na sequência, é feita a simulação do sistema e os resultados são armazenados na memória do PC. Finalmente, os resultados são exibidos na tela do PC.

Este trabalho é limitado a Redes de Petri Seguras, ou seja, aquelas onde cada lugar comporta no máximo uma marca e também não esclarece como são feitas as conversões dos arquivos.

Em *Ferramentas para a Integração de Redes de Petri e VHDL na Síntese de Sistemas Digitais* (DIAS, 2007) são descritas duas metodologias para a modelagem de máquinas de estados finitos FMS dos tipos Moore e Mealy em Redes de Petri e posterior obtenção do código em VHDL. Para isso, são utilizadas várias ferramentas, como o programa PIPE para a modelagem em Rede de Petri que gera como saída um arquivo tipo PNML. Este arquivo é analisado por outra ferramenta que gera uma tabela de transição de estados. Outra ferramenta utiliza algoritmo genético para alocar os estados em código binário de forma otimizada, gerando uma nova tabela. Outro programa utiliza esta tabela para gerar as funções booleanas do sistema de forma minimizada. Finalmente outro programa realiza a conversão destas funções para VHDL em descrição do tipo RTL. Existe ainda a opção de utilização de uma ferramenta para a conversão direta da descrição em PNML para VHDL comportamental.

Este trabalho é focado na modelagem de máquinas de estado, apresentando ferramentas de conversão PNML para VHDL apenas neste contexto.

Em *Geração de código VHDL a partir de especificações IOPT - PNML2VHDL* Lima (2009) é descrita uma ferramenta para a geração automática de código VHDL partindo-se do arquivo de descrição de Redes de Petri no formato PNML. Este trabalho se apresenta bem detalhado, com exemplo de aplicação e principalmente, expõe regras de conversão, que fo-

ram utilizadas parcialmente no desenvolvimento da ferramenta de conversão da atual pesquisa. Contudo, não apresenta o código da ferramenta de conversão.

2.7 CONSIDERAÇÕES

Existem atualmente várias ferramentas para a modelagem de sistemas utilizando Rede de Petri sendo algumas para a sua implementação em *hardware* de lógica reconfigurável. Observa-se, porém, que não existe uma abertura total para acesso ao código destas ferramentas, facilitando sua análise e possíveis alterações ou adaptações, sendo esta também uma das motivações para este trabalho de pesquisa.

3 METODOLOGIA

Nesta seção descreve-se o processo de desenvolvimento deste trabalho de pesquisa, bem como os detalhes da ferramenta de conversão obtida como produto final.

3.1 PESQUISA

Inicialmente foram realizadas pesquisas em livros e artigos sobre linguagens de especificação de *hardware*, buscando-se verificar qual seria a mais adequada para a modelagem de circuitos digitais reativos. Considerando-se os aspectos de representação gráfica, fácil aprendizado, possibilidade de visualização dos aspectos estáticos e dinâmicos do processo facilitando a modelagem de sistemas reativos, concorrentes e distribuídos, aliado a um formalismo matemático que possibilita uma análise precisa do comportamento do sistema, optou-se pela utilização das Redes de Petri.

Verificou-se porém, que a simulação de modelos de Redes de Petri em *software* não ocorre de forma fidedigna, uma vez que não contempla um paralelismo real. Assim, passou-se a pesquisar sobre a modelagem de sistemas em Redes de Petri implementadas em *hardware* de lógica reconfigurável ou mais especificamente, em FPGAs, onde foram analisadas diversos métodos e ferramentas existentes.

A partir desta análise, constatou-se que existe ainda um trabalho a ser feito no sentido de facilitar o acesso a esta tecnologia. Com este objetivo, foi desenvolvido um método para implementar modelos em Redes de Petri em *hardware* e uma versão de ferramenta para a conversão de arquivos em formato PNML para o formato VHDL. Tanto o método quanto a ferramenta são descritos detalhadamente, de forma a permitir um perfeito entendimento e até mesmo possíveis adaptações dos mesmos.

3.2 MÉTODO DESENVOLVIDO

Visando facilitar a implementação em dispositivos de lógica reconfigurável, de sistemas modelados em redes de Petri, foi utilizado um método no qual a captura do modelo é feita por um *software* já existente, o qual gera um arquivo de descrição no formato PNML (*Petri Net Markup Language*), cujo padrão pode ser estudado em (WEBER et al., 2000), (BILLINGTON et al., 2003) e (KINDLER, 2011). Este arquivo é então processado por uma ferramenta, desenvolvida neste trabalho, que gera um arquivo no formato VHDL. Este arquivo é utilizado em seguida por uma ferramenta de síntese que gera os arquivos *Register Transfer Level* (RTL) para simulação e os códigos binários para gravação em um dispositivo de lógica reconfigurável, obtendo-se assim, o circuito desejado.

Um fluxograma descrevendo a sequência de operações realizadas neste método pode ser visto na figura 25, onde são destacadas as etapas de geração do arquivo em linguagem VHDL.

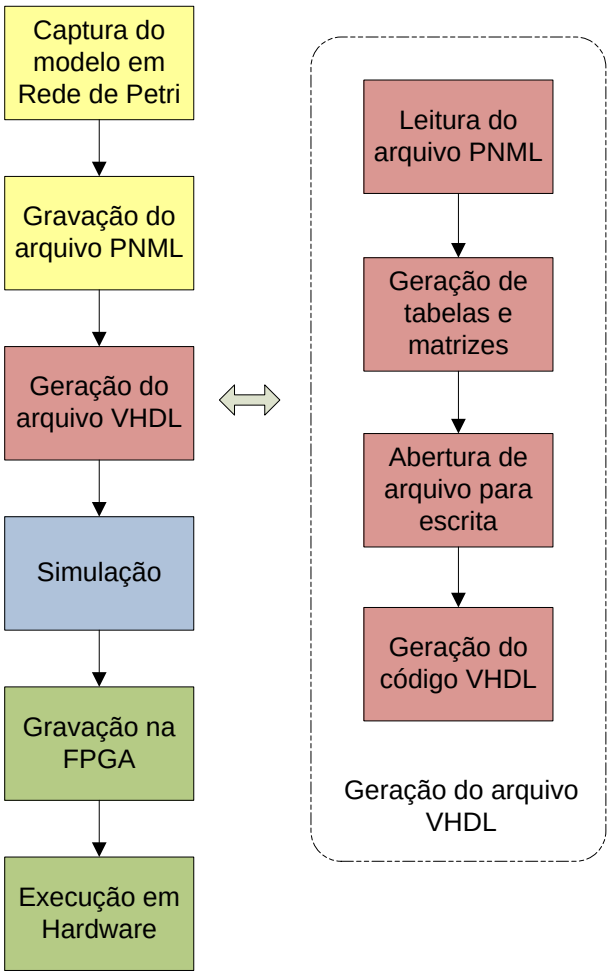


Figura 25: Sequência de operações realizadas.
Fonte: Autoria própria

3.3 PLATAFORMAS DE DESENVOLVIMENTO

Foram utilizadas três plataformas de desenvolvimento, sendo a primeira relativa à captura gráfica do modelo em Rede de Petri, a segunda relativa à criação da ferramenta que converte o arquivo PNML com a descrição da Rede de Petri para um arquivo de descrição de *hardware* na linguagem VHDL e a terceira relativa ao processo de compilação, simulação, gravação e execução em *hardware*.

Para a captura do modelo gráfico da Rede de Petri, utilizou-se o *software* Pipe-3, que é um *software* livre, fácil de operar e que gera um arquivo de descrição da Rede de Petri no formato PNML, quando o gráfico é salvo (LIU; ZENG; HE, 2011).

A linguagem escolhida para desenvolvimento da ferramenta de conversão de modelos de sistemas em Rede de Petri no formato PNML para o formato VHDL foi o C, por se tratar de uma linguagem poderosa, portável e bastante difundida nos meios científicos. Como plataforma de desenvolvimento desta ferramenta de conversão, foi utilizado o ambiente integrado de desenvolvimento para C/C++, DEV-C++, que é um *software* livre sob licença pública da GNU. Esta plataforma facilita o desenvolvimento da aplicação, pois engloba em um único ambiente o editor de código fonte, o compilador-ligador e o depurador. Além disso, possui interface gráfica com janela de gerenciamento dos arquivos do projeto.

Após a geração do arquivo em formato VHDL, utilizou-se a plataforma de desenvolvimento de *hardware* reconfigurável, Quartus II, da Altera, para compilação do arquivo VHDL, simulação do arquivo RTL e gravação do código binário relativo ao *hardware* gerado, na placa de desenvolvimento DE1, da Altera-Terasic, onde o modelo pode ser executado em *hardware*.

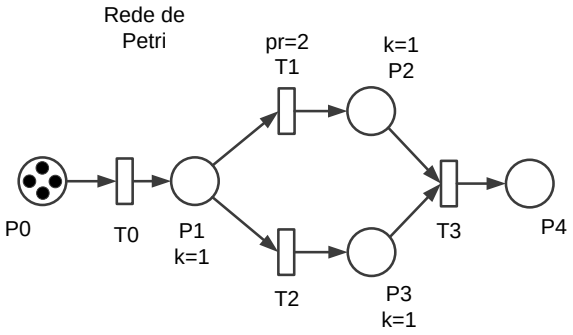
3.4 A FERRAMENTA DESENVOLVIDA

A ferramenta desenvolvida, chamada doravante de PN2VHDL, lê um arquivo de descrição da Rede de Petri no formato PNML e gera um arquivo com o código correspondente no formato VHDL. Este arquivo pode ser utilizado para simulação da execução em *hardware* em um *software* apropriado ou para implementação física da Rede de Petri em um circuito de lógica reconfigurável.

3.4.1 CAPTURA DO DIAGRAMA DA REDE DE PETRI

Para a captura do diagrama da Rede de Petri, foi utilizado o *software* livre Pipe-3, que gera um arquivo em formato PNML, contendo as informações relevantes para a geração do arquivo

em formato VHDL. Na figura 26 têm-se um exemplo de uma Rede de Petri simples e parte do código em formato PNML, gerado pelo *software* Pipe-3.




```
<?xml version="1.0" encoding="iso-8859-1" ?>
- <pnml>
 - <net id="Net-One" type="P/T net">
   - <place id="P0">
     - <graphics>
         <position x="30.0" y="90.0" />
       </graphics>
     - <name>
         <value>P0</value>
       - <graphics>
           <offset x="-5.0" y="35.0" />
         </graphics>
       </name>
     - <initialMarking>
         <value>Default,4</value>
       - <graphics>
           <offset x="0.0" y="0.0" />
         </graphics>
       </initialMarking>
      - <capacity>
         <value>0</value>
       </capacity>
   </place>
   - <place id="P1">
```

Figura 26: Rede de Petri e Trecho do arquivo PNML.
Fonte: Autoria própria

Como pode ser observado, os diversos elementos da Rede de Petri, bem como suas propriedades, são descritos utilizando-se *tags*. Assim, o *tag* <place id = ”P0”> é o início da descrição do lugar P0, que termina com o *tag* </place>.

3.4.2 LEITURA DO ARQUIVO PNML

A primeira etapa a ser realizada pelo *software* consiste em ler o arquivo PNML que contém a representação da Rede de Petri a ser codificada em VHDL. Assim, o programa solicita ao usuário que digite o nome do arquivo a ser aberto, o qual deve possuir uma extensão “xml” e estar no mesmo diretório do programa executável. O arquivo é então aberto para leitura, permitindo que seja manipulado pelas funções de busca de parâmetros, que varrem o mesmo em busca dos *tags* contendo todas as informações correspondentes aos diversos elementos da Rede de Petri. As informações obtidas são armazenadas em estruturas de dados, correspondentes aos lugares, transições ou arcos da Rede de Petri. Cada estrutura é composta por uma identificação do elemento e por suas propriedades, permitindo que sejam criados vetores de elementos, facilitando o acesso aos dados de forma indexada. A figura 27 mostra as estruturas de dados utilizadas no *software* desenvolvido. O código C, referente a esta etapa encontra-se no Apêndice, nas páginas 110 e 111 .

```
struct place
{
  char id[10];
  int marking;
  int capacity;
};
struct transition
{
  char id[10];
  float rate;
  int timed;
  int infiniteServer;
  int priority;
};
struct arc
{
  char id[20];
  char source[10];
  char target[10];
  int weight;
  int tagged;
  char type[10];
};
```

Figura 27: Código da estrutura de dados.
Fonte: Autoria própria

3.4.3 GERAÇÃO DE TABELAS DE ELEMENTOS DA REDE

Com os dados obtidos da leitura do arquivo PNML, são geradas três tabelas contendo todos os elementos da rede, juntamente com suas propriedades. A figura 28 mostra os elementos da Rede de Petri representada na figura 26. O código C, referente a esta etapa encontra-se no Apêndice, nas páginas 112 e 113.

Observa-se que os lugares P0 e P5 apresentam valor 0 na propriedade capacidade. Isto indica que não foi especificado um valor de capacidade limite para estes dois lugares e, portanto, os mesmos teriam capacidade infinita. Na realidade, porém, durante a criação do código em VHDL, deve-se estabelecer um limite para que seja possível a implementação destes lugares em *hardware*. Este limite deve ser um valor relativamente elevado e pode ser ajustado de acordo com a aplicação à qual o modelo se destina. No presente estudo foi utilizado o valor limite de 255, ocupando assim, 8 bits de *hardware* para cada lugar.

```
-------------------------------------------------------
                Lugares da Rdp

Lugar = P0
Marcas = 4
Capacidade = 0

Lugar = P1
Marcas = 0
Capacidade = 1

Lugar = P2
Marcas = 0
Capacidade = 1

Lugar = P3
Marcas = 0
Capacidade = 1

Lugar = P4
Marcas = 0
Capacidade = 0

A Rdp contem 5 lugares.
-------------------------------------------------------
                Transicoes da Rdp

Transicao = T0
Taxa = 1.000000
Temporizada = 0
infiniteServer = 0
Prioridade = 1

Transicao = T1
Taxa = 1.000000
Temporizada = 0
infiniteServer = 0
Prioridade = 2
```

Figura 28: Elementos da Rede de Petri
Fonte: Autoria própria

3.4.4 GERAÇÃO DE MATRIZES DA REDE DE PETRI

Através da análise da interação entre os elementos da rede obtidos na etapa anterior, são geradas as matizes de entrada, saída e de incidência da Rede de Petri. Estas matrizes permitem que sejam realizadas análises matemáticas das características da rede. Além disto, elas também são utilizadas no processo de geração do código VHDL, uma vez que permitem uma análise global da interação entre os diversos elementos. O código C, referente a esta etapa encontra-se no Apêndice, na página 115.

As matrizes geradas para a Rede de Petri da figura 26 são mostradas na figura 29. A primeira é a matriz de entrada, que indica a quantidade de marcas retiradas dos lugares de entrada com o disparo de cada transição. A segunda é a matriz de saída que indica a quantidade de marcas adicionadas aos lugares de saída com o disparo de cada transição. A terceira e última matriz gerada é a matriz de incidência, que computa as marcas adicionadas menos as marcas retiradas de cada lugar da rede com o disparo de cada transição.

Por exemplo, considerando-se o disparo da transição T0, pode-se verificar pela matriz de incidência, que uma marca é retirada do lugar P0 e uma marca é acrescentada aos lugares P1 e P2. Considerando-se o disparo da transição T1, pode-se verificar que uma marca é retirada do lugar P1 e uma marca é acrescentada ao lugar P3. A mesma análise pode ser feita para o disparo das demais transições.

```
------------------------------------------------------
                Matrizes da Rdp

Matriz de entrada.

      T0  T1  T2  T3
  P0  -1   0   0   0
  P1   0  -1  -1   0
  P2   0   0   0  -1
  P3   0   0   0  -1
  P4   0   0   0   0

Matriz de saida.

      T0  T1  T2  T3
  P0   0   0   0   0
  P1   1   0   0   0
  P2   0   1   0   0
  P3   0   0   1   0
  P4   0   0   0   1

Matriz de incidencia.

      T0  T1  T2  T3
  P0  -1   0   0   0
  P1   1  -1  -1   0
  P2   0   1   0  -1
  P3   0   0   1  -1
  P4   0   0   0   1

------------------------------------------------------
```

Figura 29: Matrizes da Rede de Petri.
Fonte: Autoria própria

3.4.5 GERAÇÃO DO CÓDIGO EM VHDL

A última etapa do processo é a geração do código em VHDL que poderá então ser simulado e gravado em um dispositivo de lógica reconfigurável. Este arquivo consiste em um texto descrevendo o *hardware* no formato VHDL e com extensão vhd. Para isto, foram seguidas algumas orientações obtidas em (LIMA, 2009), onde foram aplicadas modificações no sentido de atender à certos requisitos referentes à solução de conflitos e a simplificação do código gerado e que serão descritos a seguir. A figura 30 mostra um diagrama com as etapas de geração do código VHDL.

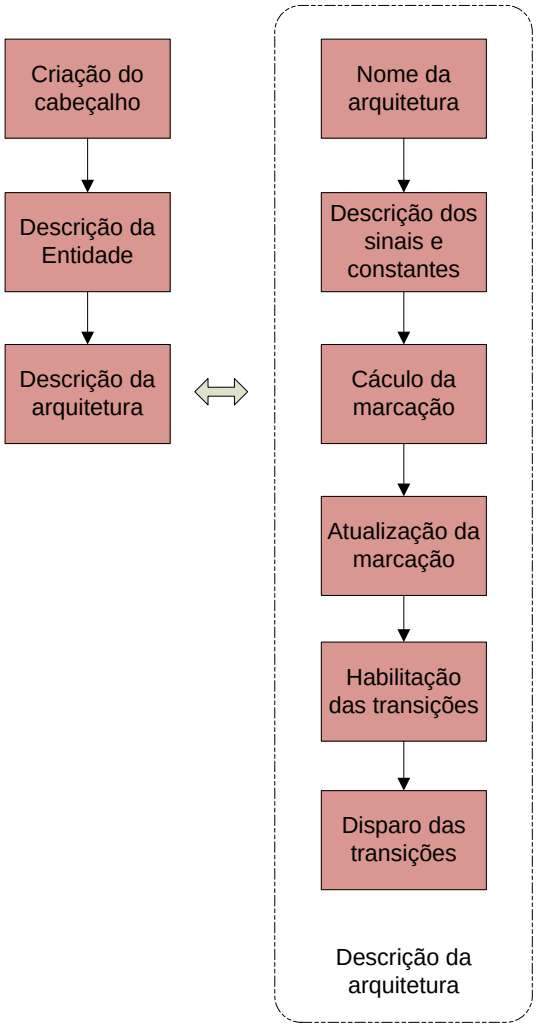


Figura 30: Etapas da geração do código em VHDL
Fonte: Autoria própria

3.4.5.1 Descrição comportamental do simulador de Redes de Petri

O código VHDL gerado, deve produzir um simulador em hardware, que tem a seguinte descrição comportamental:

1. Quando o sinal RESET estiver em nível alto, todos os registros que representam os lugares da Rede de Petri recebem a marcação inicial. Esta inicialização é feita de forma assíncrona e paralela, ou seja, todos os lugares são atualizados simultaneamente.

2. Quando o sinal de RESET está em nível baixo, a marcação evolui normalmente, respondendo a cada borda de subida do sinal de *CLOCK*.

3. Considerando o sinal de RESET em nível baixo e estando a Rede de Petri em uma determinada marcação, são obtidos os sinais de habilitação e disparo das transições. Estes

sinais podem assumir os valores 1 ou 0 e também são gerados de forma assíncrona, combinacional e paralela, a partir dos valores de marcação, dos pesos dos arcos das capacidades dos lugares de saída e das prioridades de disparo em caso de conflitos.

4. A próxima marcação é calculada considerando-se a marcação atual, as transições ajustadas para disparo, e o peso dos arcos de saída e de entrada das mesmas. Este cálculo é feito de forma assíncrona, combinacional e paralela, estando seu resultado disponível e estável nos sinais temporários (extensão temp), cerca de 10ns após a atualização de uma marcação.

5. Com a chegada de uma nova borda de subida do sinal de *CLOCK*, os valores de marcação contidos nos sinais temporários são lidos pelos registradores que representam os lugares da Rede de Petri. Esta atualização da marcação ocorre de forma síncrona e paralela.

3.4.5.2 Abertura do arquivo e criação do cabeçalho

A primeira etapa no processo de composição do arquivo em VHDL é a abertura de um arquivo texto para escrita, o qual recebe o nome do projeto e tem uma extensão vhd. Em seguida, utilizando-se a função fprintf da linguagem C, escreve-se o cabeçalho do código, que contém as bibliotecas utilizadas. A figura 31 mostra um trecho do *software* responsável pela escrita do cabeçalho.

```
printf("\t\tCodigo em VHDL\n\n");
printf("library ieee;\n");
printf("use ieee.std_logic_1164.all;\n");
printf("use ieee.std_logic_arith.all;\n");
printf("use ieee.std_logic_unsigned.all;\n\n");
```

Figura 31: Código da declaração de bibliotecas.
Fonte: Autoria própria

O código C, referente a esta etapa encontra-se no Apêndice, na página 118.

3.4.5.3 Criação da descrição da entidade

Na etapa de descrição da entidade são especificados os terminais de entrada e saída do *hardware* a ser gerado pelo código em VHDL. No presente trabalho de pesquisa, que tem como objetivo a criação de um *hardware* que simule uma Rede de Petri, utilizou-se o tipo de terminal buffer para cada um dos lugares da rede. Isto permite que mesmo atuando como saída, seus valores possam ser lidos internamente. Além disto, criou-se uma função que determina a quantidade de bits a ser utilizada de acordo com a capacidade de cada lugar, sendo utilizado o tipo

de dado *std_logic_vector* (IEEE, 2009b). Na figura 32, têm-se o fluxograma para o trecho do programa que escreve a descrição da entidade. Como pode ser observado, além dos terminais para os sinais de *CLOCK* e *RESET* foi criado um loop for para gerar os terminais para todos os lugares da Rede de Petri.

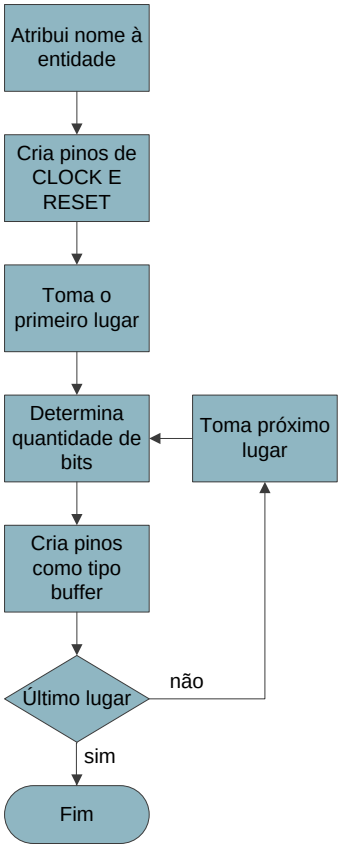


Figura 32: Codificação da descrição da entidade.
Fonte: Autoria própria

O código C, referente a esta etapa encontra-se no Apêndice, na página 118. O código VHDL para o cabeçalho e entidade da Rede de Petri da figura 26 é mostrado na figura 33.

3.4.5.4 Criação da descrição da arquitetura

Na etapa de descrição da arquitetura são especificados os processos que compõem os diversos elementos da Rede de Petri.

3.4.5.5 Descrição dos sinais e constantes

Na etapa inicial de descrição da arquitetura são especificados os sinais e constantes utilizados pelo *hardware*. No presente trabalho foram criados sinais temporários para cada lugar da

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity exemplo IS
        PORT
        (
        CLK : in std_logic;
        RESET : in std_logic;
        P0 : buffer std_logic_vector(7 downto 0);
        P1 : buffer std_logic_vector(0 downto 0);
        P2 : buffer std_logic_vector(0 downto 0);
        P3 : buffer std_logic_vector(0 downto 0);
        P4 : buffer std_logic_vector(7 downto 0)
        );
end exemplo;
```

Figura 33: Código VHDL da descrição da entidade.
Fonte: Autoria própria

Rede de Petri, sinais de 1 bit, representando o disparo e a habilitação de cada transição. Os sinais temporários são necessários para evitar o problema da metaestabilidade, que ocorre quando uma determinada entrada digital tem seu valor alterado durante uma mudança de estado do sistema. Foram criadas também as constantes de tipo inteiro, que representam os pesos de cada arco, as capacidades de cada lugar e as prioridades de disparo de cada transição. O fluxograma para o trecho de programa que escreve esta parte do código VHDL é mostrado na figura 34.

O código C, referente a esta etapa encontra-se no Apêndice, na página 119. O código VHDL gerado para descrição dos sinais e constantes referentes à Rede de Petri da figura 26 é mostrado na figura 35.

Como pode ser observado, foram criados os sinais temporários para armazenamento do cálculo das marcações para cada lugar da Rede de Petri, com um tamanho de palavra adequado à capacidade do mesmo. Também foram criados sinais de 1 bit para habilitação e disparo das transições. Em seguida foram criadas as constantes de peso dos arcos, constantes de prioridade das transições e constantes de capacidade de cada lugar. Observa-se que todos os arcos possuem peso 1 e que a transição T1 tem prioridade maior que as demais. Isto foi feito para solucionar o conflito entre as transições T1 e T2. Os lugares P1, P2 e P3, possuem capacidade limitada em 1 unidade. Já os lugares P0 e P4 não foram limitados e recebem um limite automático de 255 unidades.

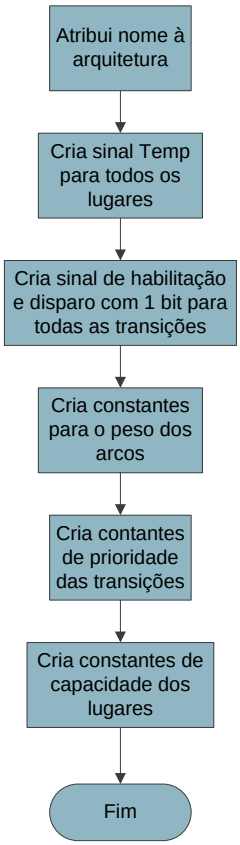


Figura 34: Codificação dos sinais e constantes.
Fonte: Autoria própria

3.4.5.6 Cálculo da marcação

Nesta etapa é produzida a codificação do *hardware* responsável pelo cálculo do número de marcas em cada lugar da Rede de Petri de acordo com o disparo das transições. Para isto, é computada a quantidade atual de marcas em cada lugar mais o peso dos arcos de entrada menos o peso dos arcos de saída de cada lugar multiplicados pelo sinal de disparo das transições associadas a estes arcos. Os sinais de disparo das transições utilizam um único bit, podendo assim, assumir os valores 0 ou 1, que fazendo parte de um produto, determinam quais valores de peso de arco serão somados ou subtraídos em cada lugar.

Para determinar quais são os arcos de entrada e saída de cada lugar, o *software* utiliza as matrizes de entrada e saída, que expressam os valores de pesos de arcos associados a cada par lugar, transição.

Esta codificação é produzida logo após a palavra *begin* no início da *architecture* não estando dentro de nenhum processo e produzindo assim, um circuito combinacional que mantém os valores temporários constantemente atualizados. O fluxograma relativo ao trecho do *software*

```
architecture behavioral OF exemplo IS

        signal P0Temp : std_logic_vector(7 downto 0);
        signal P1Temp : std_logic_vector(0 downto 0);
        signal P2Temp : std_logic_vector(0 downto 0);
        signal P3Temp : std_logic_vector(0 downto 0);
        signal P4Temp : std_logic_vector(7 downto 0);

        signal T0, T0_ENABLE : std_logic;
        signal T1, T1_ENABLE : std_logic;
        signal T2, T2_ENABLE : std_logic;
        signal T3, T3_ENABLE : std_logic;

        constant WP0T0   : integer := 1;
        constant WP1T1   : integer := 1;
        constant WP1T2   : integer := 1;
        constant WP2T3   : integer := 1;
        constant WP3T3   : integer := 1;
        constant WT0P1   : integer := 1;
        constant WT1P2   : integer := 1;
        constant WT2P3   : integer := 1;
        constant WT3P4   : integer := 1;

        constant PRIOR_T0        : integer := 1;
        constant PRIOR_T1        : integer := 2;
        constant PRIOR_T2        : integer := 1;
        constant PRIOR_T3        : integer := 1;

        constant K_P0    : integer := 255;
        constant K_P1    : integer := 1;
        constant K_P2    : integer := 1;
        constant K_P3    : integer := 1;
        constant K_P4    : integer := 255;
```

Figura 35: Código VHDL da descrição dos sinais e constantes.
Fonte: Autoria própria

que produz esta codificação é mostrado na figura 36.

O código C, referente a esta etapa encontra-se no Apêndice, na página 119. O código VHDL gerado para o cálculo da marcação da Rede de Petri da figura 26 é mostrado na figura 37.

3.4.5.7 Atualização do número de marcas

Nesta etapa é produzida a codificação do *hardware* responsável pela atualização da marcação da Rede de Petri, que responde a dois sinais. O sinal de *RESET* provoca o carregamento do número de marcas inicial em cada lugar da rede. A borda de subida do sinal de *CLOCK* faz com que o número de marcas calculado, presente nos sinais temporários, seja carregado em cada lugar da rede.

Para que esta atualização ocorra de forma paralela, é realizada a codificação de um processo para cada lugar da rede, contendo cada um, uma lista de sensibilidade com os sinais *clock* e *reset*. O trecho do *software* que produz esta codificação é mostrado na figura 38.

O código C, referente a esta etapa, encontra-se no Apêndice, na página 120. O código

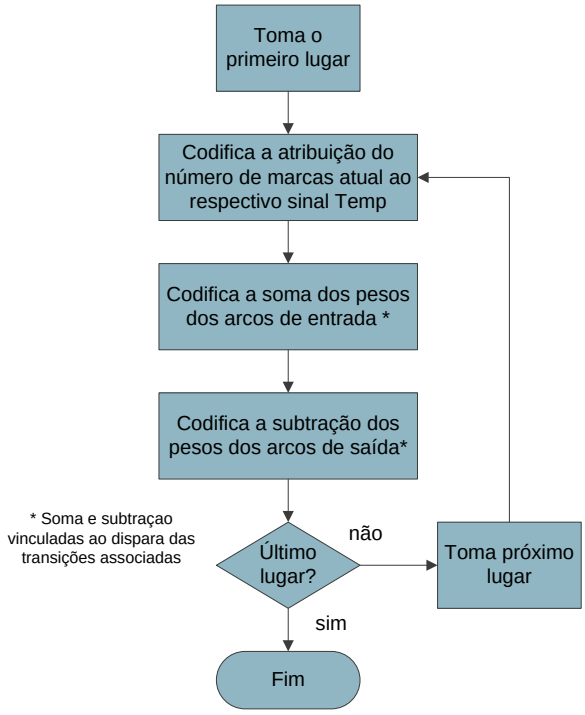


Figura 36: Codificação do cálculo da marcação.
Fonte: Autoria própria

```
begin

        P0Temp <= conv_std_logic_vector((conv_integer(P0)
                - WP0T0*conv_integer(T0)),8);

        P1Temp <= conv_std_logic_vector((conv_integer(P1)
                + WT0P1*conv_integer(T0)
                - WP1T1*conv_integer(T1)
                - WP1T2*conv_integer(T2)),1);

        P2Temp <= conv_std_logic_vector((conv_integer(P2)
                + WT1P2*conv_integer(T1)
                - WP2T3*conv_integer(T3)),1);

        P3Temp <= conv_std_logic_vector((conv_integer(P3)
                + WT2P3*conv_integer(T2)
                - WP3T3*conv_integer(T3)),1);

        P4Temp <= conv_std_logic_vector((conv_integer(P4)
                + WT3P4*conv_integer(T3)),8);
```

Figura 37: Código VHDL do cáculo da marcação.
Fonte: Autoria própria

VHDL gerado para a atualização da marcação da Rede de Petri da figura 26 é mostrado na figura 39.

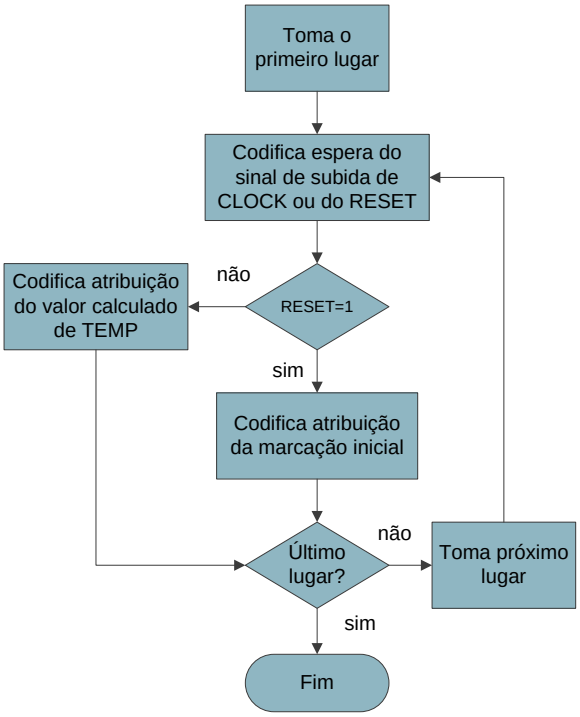


Figura 38: Codificação da atualização da marcação.
Fonte: Autoria própria

3.4.5.8 Habilitação das transições

Nesta etapa é produzida a codificação do *hardware* responsável pela habilitação para disparo das transições da Rede de Petri. Para isto, o *hardware* deve analisar duas condições necessárias para que uma determinada transição esteja habilitada.

A primeira condição é que a quantidade de marcas nos lugares de entrada de cada transição seja maior ou igual aos pesos dos respectivos arcos de entrada.

A segunda condição a ser verificada é que a capacidade dos lugares de saída de cada transição não seja excedida com o disparo da mesma. Ou seja, a capacidade dos lugares de saída menos o número de marcas já existentes nestes lugares deve ser maior ou igual ao peso dos arcos que interligam a transição aos mesmos.

Quando uma transição é habilitada seu sinal de habilitação recebe o valor 1, em caso contrário, o sinal de habilitação recebe o valor 0.

Como pode ser observado no fluxograma da figura 40, para cada transição, são verificados os lugares de entrada e saída em conjunto com os respectivos arcos, para a composição do código VHDL com as condições de habilitação. Para esta verificação são utilizadas as matrizes de entrada e saída, que já possuem os dados necessários para esta codificação.

```
process (CLK, RESET) begin
        if (RESET = '1') then P0 <= conv_std_logic_vector(4,8);
        elsif (CLK'event and CLK='1') then
                P0 <= P0Temp;
        end if;
end process;

process (CLK, RESET) begin
        if (RESET = '1') then P1 <= conv_std_logic_vector(0,1);
        elsif (CLK'event and CLK='1') then
                P1 <= P1Temp;
        end if;
end process;

process (CLK, RESET) begin
        if (RESET = '1') then P2 <= conv_std_logic_vector(0,1);
        elsif (CLK'event and CLK='1') then
                P2 <= P2Temp;
        end if;
end process;

process (CLK, RESET) begin
        if (RESET = '1') then P3 <= conv_std_logic_vector(0,1);
        elsif (CLK'event and CLK='1') then
                P3 <= P3Temp;
        end if;
end process;

process (CLK, RESET) begin
        if (RESET = '1') then P4 <= conv_std_logic_vector(0,8);
        elsif (CLK'event and CLK='1') then
                P4 <= P4Temp;
        end if;
end process;
```

Figura 39: Código VHDL da atualização da marcação.
Fonte: Autoria própria

O código C, referente a esta etapa, encontra-se no Apêndice, na página 120. O código VHDL gerado para a habilitação das transições da Rede de Petri da figura 26 é mostrado na figura 41.

3.4.5.9 Disparo das transições

Nesta etapa é produzida a codificação do *hardware* responsável pelo disparo das transições habilitadas. Para isto, o *hardware* gerado deve considerar também as possíveis condições de conflito entre várias transições que estejam habilitadas, os quais podem ser basicamente de dois tipos:

O primeiro tipo ocorre quando um determinado lugar possui marcas que estão contribuindo para a habilitação de duas ou mais transições. Caso a quantidade de marcas seja inferior àquela necessária para suprir o disparo de todas elas, tem-se uma situação de conflito de entrada.

O segundo tipo de conflito ocorre quando duas ou mais transições, ao serem disparadas fornecem simultaneamente marcas para um mesmo lugar da Rede de Petri. Caso a capacidade disponível deste lugar seja inferior àquela necessária para receber as marcas advindas destas

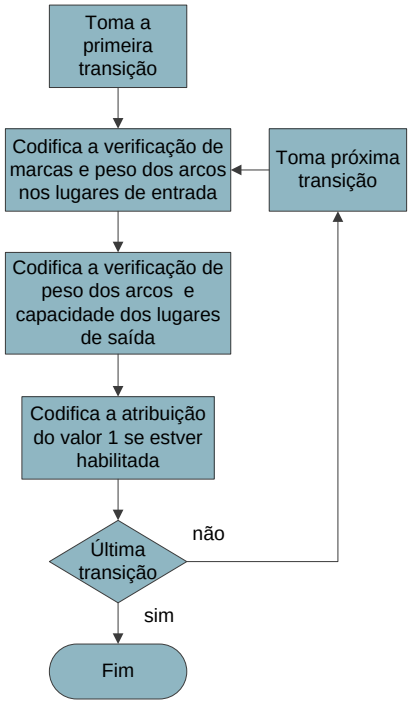


Figura 40: Codificação da habilitação das transições.
Fonte: Autoria própria

```
T0_ENABLE <= '1' when
    (P0 >= WP0T0 and
    K_P1-P1 >= WT0P1)
    else '0';

T1_ENABLE <= '1' when
    (P1 >= WP1T1 and
    K_P2-P2 >= WT1P2)
    else '0';

T2_ENABLE <= '1' when
    (P1 >= WP1T2 and
    K_P3-P3 >= WT2P3)
    else '0';

T3_ENABLE <= '1' when
    (P2 >= WP2T3 and P3 >= WP3T3 and
    K_P4-P4 >= WT3P4)
    else '0';
```

Figura 41: Código VHDL para habilitação das transições.
Fonte: Autoria própria

transições, tem-se uma situação de conflito de saída, também conhecido como contato.

Neste caso algumas das transições serão disparadas, enquanto outras não, segundo algum critério de escolha. No presente estudo foi utilizado o critério de prioridades diferentes para a solução dos conflitos, onde serão disparadas preferencialmente as transições de maior prioridade.

Em entras abordagens, pode-se utilizar outros critérios de solução de conflitos, como dis-

paro aleatório, disparo probabilístico, ou disparo por tempo determinístico.

Quando uma transição é disparada seu sinal de disparo recebe o valor 1, em caso contrário, o sinal de disparo recebe o valor 0. O fluxograma do processo de codificação em VHDL da etapa do *hardware* que controla o disparo das transições pode ser visto na figura 42 e utiliza novamente as matrizes de entrada e saída, que permitem inclusive uma análise dos possíveis conflitos entre as transições. As constantes de prioridade são utilizadas na lógica de solução dos conflitos.

O código C, referente a esta etapa encontra-se no Apêndice, na página 121. O código VHDL gerado para o disparo das transições da Rede de Petri da figura 26 é mostrado na figura 43, onde pode ser observada a solução do conflito entre as transições T1 e T2.

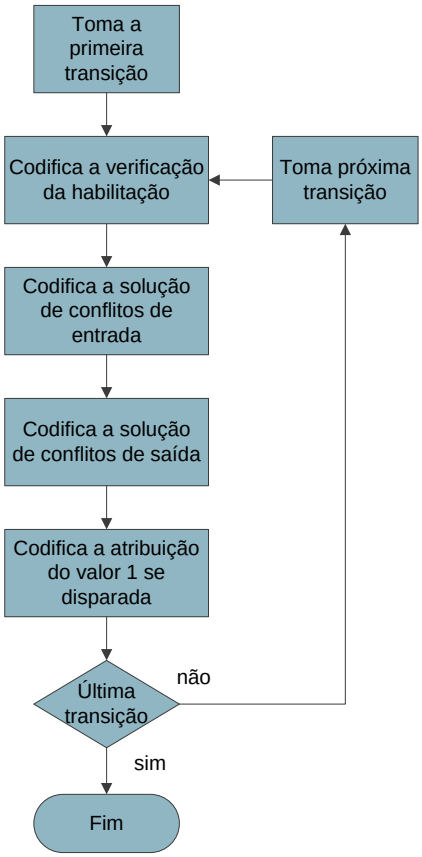


Figura 42: Codificação do disparo e análise de conflitos.
Fonte: Autoria própria

3.4.5.10 Divisor de Clock

Conforme descrito anteriormente as transições da Rede de Petri são disparadas em sincronismo com a borda de subida de um sinal de *clock*, que possui normalmente uma frequência relativamente elevada. Isto faz com que a sequência de disparos da Rede de Petri se processe

```
    T0 <= '1' when (T0_ENABLE = '1')
          else '0';

    T1 <= '1' when (T1_ENABLE = '1' and
          ((conv_integer(P1) >= WP1T1 + conv_integer(T2_ENABLE)*WP1T2) or
          (PRIOR_T1 > conv_integer(T2_ENABLE)*PRIOR_T2)))
          else '0';

    T2 <= '1' when (T2_ENABLE = '1' and
          ((conv_integer(P1) >= WP1T2 + conv_integer(T1_ENABLE)*WP1T1) or
          (PRIOR_T2 > conv_integer(T1_ENABLE)*PRIOR_T1)))
          else '0';

    T3 <= '1' when (T3_ENABLE = '1')
          else '0';


end behavioral;
```

Figura 43: Código VHDL para disparo das transições.
Fonte: Autoria própria

de forma muito rápida, com um tempo de poucos nanosegundos entre uma transição e outra, representando para muitas aplicações práticas um disparo quase imediato, o que é desejável.

Contudo, na fase de testes do modelo implementado em *hardware* de lógica reconfigurável, deseja-se algumas vezes acompanhar a sequencia de transições ocorrendo em uma velocidade menor, que permita monitorar o correto funcionamento do circuito. Para isto, a ferramenta desenvolvida neste estudo gera também o código VHDL para um circuito divisor de *clock*, permitindo assim a escolha de um período tempo maior entre o acionamento das transições. O valor deste período de tempo, expresso em milissegundos, é solicitado por ocasião da geração do arquivo VHDL, juntamente com o valor da frequência de *clock* original utilizada.

O circuito divisor de *clock* é gerado através de dois contadores, sendo que o primeiro gera uma borda de subida a cada 1ms e o segundo conta estes milissegundos até atingir o valor de tempo desejado, gerando então um sinal de *clock* com um período adequado, conforme mostrado no diagrama da figura 44.

O código C, referente a esta etapa encontra-se no Apêndice, na página 122. O código VHDL gerado para o divisor de *clock* é mostrado na figura 45, sendo gerado quando o período selecionado for diferente de zero.

3.5 CONSIDERAÇÕES

Para a solução de conflitos foi utilizada a atribuição de prioridades diferentes para as transições envolvidas. Foi considerada a situação onde uma transição de prioridade menor em um conflito possa ser disparada caso a transição nos caso de haver marcas excedentes ou

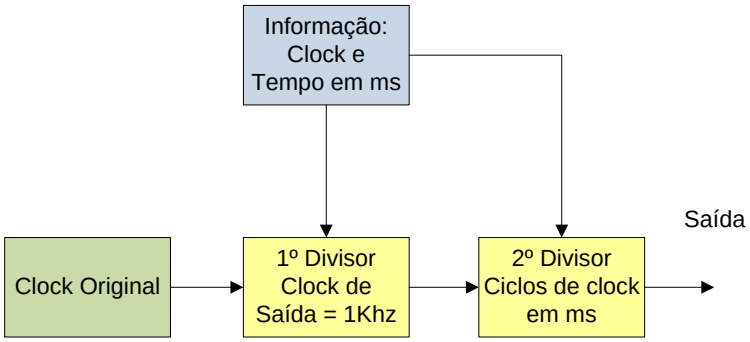


Figura 44: Diagrama de blocos do divisor de *clock*.

```
process (CLK1)
VARIABLE CONTADOR : INTEGER RANGE 0 TO 27000;
VARIABLE CONTA_MS : INTEGER RANGE 0 TO PERIODO;
begin
if (PERIODO = 0) then
CLK <= CLK1;
elsif (CLK1'event and CLK1='1') then CONTADOR :=
CONTADOR + 1;
if (CONTADOR = 27000/2) then CONTA_MS := CONTA_MS + 1;
CONTADOR :=0;
if (CONTA_MS = PERIODO) then
CONTA_MS :=0;
CLK <= NOT CLK;
end if;
end if;
end if;
end process;
```

Figura 45: Código VHDL do divisor de *clock*.
Fonte: Autoria própria

quando uma transição de maior prioridade estiver desabilitada.

Cabe também lembrar que outros processos de solução de conflito, menos determinísticos, também podem ser elaborados através de modificações na codificação da lógica de disparo.

4 VALIDAÇÃO DA FERRAMENTA

Para avaliação dos resultados utilizou-se um exemplo de aplicação da ferramenta desenvolvida, a qual deve ler um arquivo no formato PNML representando o modelo de uma planta industrial hipotética e gerar um código VHDL correspondente, conforme descrito nas seções seguintes.

4.1 EXEMPLO DE APLICAÇÃO

Como exemplo da utilização da ferramenta desenvolvida foi implementado um modelo em Rede de Petri representando uma planta industrial hipotética, destinada à produção de lápis e canetas.

Nesta fábrica, as canetas e os lápis são produzidos de forma independente em setores diferentes da planta industrial e enviados para o setor de embalagem. Neste setor são produzidos conjuntos contendo três lápis e duas canetas. Os conjuntos são então embalados em caixas de 10 unidades, as quais são enviadas ao depósito final. O depósito final tem um limite de capacidade, que ao ser atingido, determina a paralisação da produção. Este modelo de planta industrial hipotética foi inspirado no exemplo da fábrica de canetas de (MACIEL; LINS; CUNHA, 1996).

4.1.1 DIAGRAMA DA REDE DE PETRI

O diagrama da Rede de Petri modelando a planta industrial hipotética é mostrado na figura 46 e foi capturado utilizando-se o *software* livre Pipe 3.0, que gera um arquivo do tipo PNML, contendo a descrição do modelo.

Como pode ser observado, a planta industrial possui um setor de produção de canetas, onde os lugares P0 e P1 representam os depósitos de matéria prima, que poderiam ser por exemplo, plástico e tinta. Estes depósitos são alimentados através das transições T3 e T2, que são do tipo fonte. O fornecimento de matéria prima da transição T3 ocorre de uma em uma unidade e o da transição T2 ocorre em lotes de 6 unidades, como pode ser observado pelo peso dos

respectivos arcos. A transição T0 representa o início da produção de uma caneta, a qual ocorrerá no lugar P2 e consome a marca do lugar P3 que representa a disponibilidade da máquina. Desta forma, o início da produção de uma nova caneta só ocorrerá após o disparo da transição T1, que representa o termino da produção da caneta atual, enviando a mesma para um depósito representado por P4 e devolvendo uma marca ao lugar P3.

Um processo semelhante ocorre no setor de produção de lápis, onde as transições T6 e T7 fornecem matéria prima em lotes de 4 unidades para os depósitos representados por P6 e P5 respectivamente. Esta matéria prima poderia ser, por exemplo, madeira e grafite. As transições T4 e T5 indicam o início e o fim da produção de um lápis. O lugar P7 representa um lápis sendo produzido, o qual é posteriormente enviado para o depósito representado por P9 ao mesmo tempo em que uma marca é enviada ao lugar P8 que representa máquina disponível.

No setor de montagem de conjuntos, 3 canetas e 2 lápis são retirados dos depósitos P4 e P9 pelo disparo da transição T8 que indica o início da produção de um conjunto e consome a marca do lugar P12 que representa a disponibilidade da máquina. O término da produção do conjunto é indicado pelo disparo da transição T9, que devolve a marca para o lugar P12 e envia uma marca para o lugar P11, o qual representa o depósito de conjuntos. Quando a quantidade de conjuntos neste depósito for de 10 unidades, ocorre o disparo da transição T10, que indica um encaixotamento e envio para o depósito final, representado pelo lugar P13.

4.2 CÓDIGO EM PNML

Ao se executar o comando salvar arquivo na plataforma Pipe-3, é gerado um arquivo no formato PNML, que neste caso recebeu o nome de fabrica.xml e é mostrado parcialmente na figura 47.

4.3 CÓDIGO EM VHDL

Em seguida o arquivo no formato PNML foi convertido para código VHDL utilizando-se a ferramenta desenvolvida neste estudo. O código VHDL gerado é mostrado e descrito a seguir, em suas diversas partes.

4.3.1 CABEÇALHO E ENTIDADE

Na parte inicial do código, mostrado na figura 48, pode-se observar a cabeçalho contendo as bibliotecas utilizadas e logo em seguida a descrição da entidade, onde são descritos os terminais

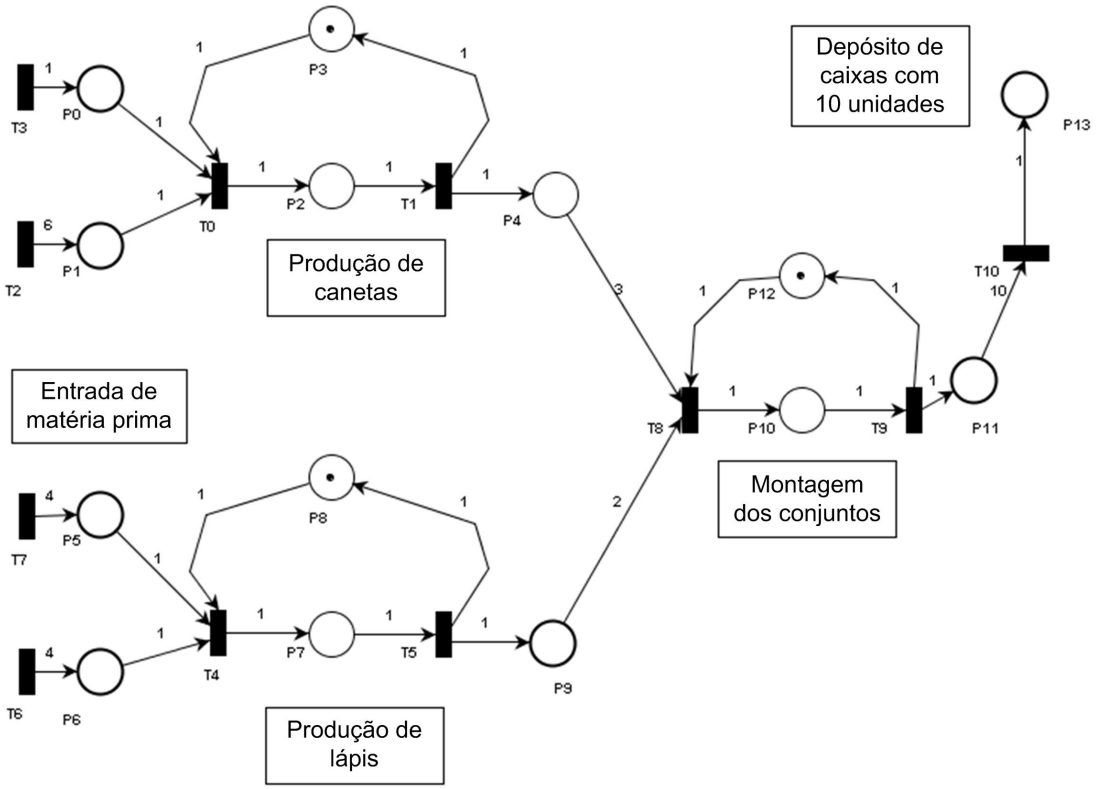


Figura 46: Planta industrial hipotética modelada em Rede de Petri
Fonte: Autoria própria

de entrada e saída do *hardware* a ser gerado. Inicialmente aparecem os terminais de *CLOCK* e *RESET*, que são entradas. Em seguida, são declarados os terminais referentes a cada lugar da rede, sendo do tipo buffer, o que permite que os mesmos sejam utilizados como entrada ou saída.

Como pode ser observado, os terminais possuem o tamanho de palavra digital necessário para comportar um valor correspondente à capacidade de marcas de cada lugar. Por exemplo, os lugares P0, P1, P5, P6, P9 e P13 possuem capacidade inferior a 8, utilizando desta forma, apenas 3 bits (2 DOWNTO 0). Como foi exposto anteriormente na descrição da ferramenta, os lugares que não tiveram sua capacidade limitada de forma explícita, receberam um limite máximo de 255, utilizando, assim, 8 bits (7 DOWNTO 0).

4.3.2 ARQUITETURA - SINAIS

Após a descrição da entidade, é gerada a parte do código referente à arquitetura, que pode ser vista parcialmente na figura 49 e que recebe o nome “behavioral”. Nesta arquitetura, logo

```
<?xml version="1.0" encoding="iso-8859-1" ?>
- <pnml>
  - <net id="Net-One" type="P/T net">
      <tokenclass id="Default" enabled="true" red="0" green="0" blue="0" />
    - <labels x="547" y="57" width="107" height="55" border="true">
        <text>Depósito de Caixas com 10 Unidades</text>
      </labels>
    - <labels x="25" y="287" width="105" height="37" border="true">
        <text>Entrada de matéria prima</text>
      </labels>
    - <labels x="498" y="348" width="107" height="39" border="true">
        <text>Montagem de Conjunto</text>
      </labels>
    - <labels x="196" y="200" width="107" height="39" border="true">
        <text>Produção de Canetas</text>
      </labels>
    - <labels x="195" y="513" width="107" height="39" border="true">
        <text>Produção de Lápis</text>
      </labels>
    - <place id="P0">
      - <graphics>
          <position x="55.0" y="70.0" />
        </graphics>
      - <name>
          <value>P0</value>
        - <graphics>
            <offset x="10.0" y="40.0" />
          </graphics>
        </name>
      - <initialMarking>
          <value>Default,0</value>
        - <graphics>
            <offset x="0.0" y="0.0" />
          </graphics>
        </initialMarking>
      - <capacity>
          <value>6</value>
        </capacity>
      </place>
    - <place id="P1">
      - <graphics>
          <position x="55.0" y="175.0" />
        </graphics>
      - <name>
          <value>P1</value>
        - <graphics>
            <offset x="10.0" y="42.0" />
          </graphics>
        </name>
      - <initialMarking>
          <value>Default,0</value>
        - <graphics>
```

Figura 47: Código PNML da Planta industrial hipotética
Fonte: Autoria própria

no início, são mostrados os sinais utilizados para registrar os valores temporários da quantidade de marcas nos lugares e em seguida os sinais utilizados para registrar a habilitação e o disparo das transições. Da mesma forma que os terminais da entidade, os sinais são do tipo std_logic e utilizam a quantidade necessária de bits para comportar os respectivos valores. Assim, enquanto o sinal P11Temp utiliza um vetor de 4 bits (3 DOWNTO 0), o sinal T0 e todos os sinais relativos às transições, utilizam apenas 1 bit.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_arith.all;
4  use ieee.std_logic_unsigned.all;
5  ---------------------------------------------
6  entity fabrica IS
7    PORT
8    (
9    CLK : buffer std_logic;
10   CLK1 : in std_logic;
11   RESET : in std_logic;
12   P0 : buffer std_logic_vector(2 downto 0);
13   P1 : buffer std_logic_vector(2 downto 0);
14   P10 : buffer std_logic_vector(7 downto 0);
15   P11 : buffer std_logic_vector(3 downto 0);
16   P12 : buffer std_logic_vector(7 downto 0);
17   P13 : buffer std_logic_vector(2 downto 0);
18   P2 : buffer std_logic_vector(7 downto 0);
19   P3 : buffer std_logic_vector(7 downto 0);
20   P4 : buffer std_logic_vector(7 downto 0);
21   P5 : buffer std_logic_vector(2 downto 0);
22   P6 : buffer std_logic_vector(2 downto 0);
23   P7 : buffer std_logic_vector(7 downto 0);
24   P8 : buffer std_logic_vector(7 downto 0);
25   P9 : buffer std_logic_vector(2 downto 0)
26   );
27 end fabrica;
```

Figura 48: Código em VHDL do Cabeçalho e da Entidade.
Fonte: Autoria própria

4.3.3 ARQUITETURA - CONSTANTES

Após a descrição dos sinais, aparece a parte do código que descreve às constantes utilizadas pelo *hardware*. Primeiramente, têm-se as constantes referentes ao peso dos arcos da Rede de Petri, que podem ser vistas na figura 50. Estas constantes são do tipo inteiro e têm o nome de acordo com arco a que pertencem, indicando a origem e o destino do mesmo. Por exemplo, a constante WP5T4 possui o valor 1 e tem o nome formado pela letra W indicando o peso (*weight*), do arco com origem no lugar P5 e destino na transição T4.

Em seguida, como podem ser vistas na figura 51, aparecem as constantes referentes às prioridades de disparo das transições da Rede de Petri utilizadas para solucionar possíveis conflitos. Estas constantes também são do tipo inteiro e apresentam o nome de acordo com a transição a que pertencem. Como se pode observar neste caso, a prioridade de disparo de todas as transição recebe o valor 1, indicando que não existe uma prioridade efetiva entre elas. Isto ocorre, devido ao fato de não haver situações de conflito nesta rede. Caso ocorram conflitos em um determinado modelo, deve-se atribuir prioridades diferentes para as transições envolvidas para que

```
28  ----------------------------------------------------
29  architecture behavioral OF fabrica IS
30
31    signal P0Temp : std_logic_vector(2 downto 0);
32    signal P1Temp : std_logic_vector(2 downto 0);
33    signal P10Temp : std_logic_vector(7 downto 0);
34    signal P11Temp : std_logic_vector(3 downto 0);
35    signal P12Temp : std_logic_vector(7 downto 0);
36    signal P13Temp : std_logic_vector(2 downto 0);
37    signal P2Temp : std_logic_vector(7 downto 0);
38    signal P3Temp : std_logic_vector(7 downto 0);
39    signal P4Temp : std_logic_vector(7 downto 0);
40    signal P5Temp : std_logic_vector(2 downto 0);
41    signal P6Temp : std_logic_vector(2 downto 0);
42    signal P7Temp : std_logic_vector(7 downto 0);
43    signal P8Temp : std_logic_vector(7 downto 0);
44    signal P9Temp : std_logic_vector(2 downto 0);
45
46    signal T0, T0_ENABLE : std_logic;
47    signal T1, T1_ENABLE : std_logic;
48    signal T10, T10_ENABLE : std_logic;
49    signal T2, T2_ENABLE : std_logic;
50    signal T3, T3_ENABLE : std_logic;
51    signal T4, T4_ENABLE : std_logic;
52    signal T5, T5_ENABLE : std_logic;
53    signal T6, T6_ENABLE : std_logic;
54    signal T7, T7_ENABLE : std_logic;
55    signal T8, T8_ENABLE : std_logic;
56    signal T9, T9_ENABLE : std_logic;
57
```

Figura 49: Código em VHDL da Arquitetura descrevendo os Sinais.
Fonte: Autoria própria

ocorra a solução do conflito e o disparo das transições prioritárias. Caso isto não seja feito, o *software* fará uma atribuição automática de prioridades diferentes para que seja possível ocorrer o disparo. Assim, é conveniente examinar as constantes de prioridades das transições no código VHDL gerado, para verificar se as mesmas estão de acordo com o que se espera do modelo do sistema. Caso contrário, as prioridades das transições devem ser alteradas manualmente.

Finalmente, como mostrado na figura 52, aparecem as constantes referentes às capacidades dos lugares da Rede de Petri, que determinam o limite máximo de marcas que os mesmos podem conter. Estas constantes também são do tipo inteiro e apresentam o nome de acordo com o lugar a que pertencem. Como pode ser observado, existem valores de capacidade que foram estipulados e outros onde a própria ferramenta atribuiu um limite máximo de 255 marcas, representando os lugares não limitados. Este valor foi utilizado neste trabalho, por ser uma quantidade suficiente para os testes executados, podendo ser facilmente alterado, caso seja necessário.

```
57
58    constant WP5T4 : integer := 1;
59    constant WP0T0 : integer := 1;
60    constant WP6T4 : integer := 1;
61    constant WP10T9 : integer := 1;
62    constant WP11T10 : integer := 10;
63    constant WP12T8 : integer := 1;
64    constant WP1T0 : integer := 1;
65    constant WP7T5 : integer := 1;
66    constant WP2T1 : integer := 1;
67    constant WP8T4 : integer := 1;
68    constant WP3T0 : integer := 1;
69    constant WP4T8 : integer := 3;
70    constant WP9T8 : integer := 2;
71    constant WT4P7 : integer := 1;
72    constant WT0P2 : integer := 1;
73    constant WT5P8 : integer := 1;
74    constant WT5P9 : integer := 1;
75    constant WT10P13 : integer := 1;
76    constant WT1P3 : integer := 1;
77    constant WT1P4 : integer := 1;
78    constant WT6P6 : integer := 4;
79    constant WT2P1 : integer := 6;
80    constant WT7P5 : integer := 4;
81    constant WT3P0 : integer := 1;
82    constant WT8P10 : integer := 1;
83    constant WT9P11 : integer := 1;
84    constant WT9P12 : integer := 1;
85
```

Figura 50: Código em VHDL das Contantes de pesos dos arcos.
Fonte: Autoria própria

4.3.4 ARQUITETURA - CÁLCULO DAS MARCAÇÕES

Criados os sinais e constantes é iniciada, após a palavra *begin*, a parte lógica do circuito a ser gerado. Na figura 53 é mostrado o código responsável pelo cálculo das marcações da rede. Para isto são considerados os pesos dos arcos de entrada e saída de cada lugar e os sinais de disparo das transições associadas a este. O cálculo é realizado por lógica combinacional e ocorre de forma paralela e assíncrona, ou seja, não depende de nenhum sinal de *CLOCK*.

O cálculo é realizado tomando-se o número de marcas atual de cada lugar, somando-se a ele o peso de seus arcos de entrada e subtraindo-se o peso de seus arcos de saída, caso a transição associada àqueles arcos esteja disparada, ou seja, com valor 1. Para isto é realizado um produto dos pesos dos arcos pelo sinal de disparo da transição associada. Caso a transição não esteja disparada o termo será zerado, fazendo com que aqueles pesos não participem do cálculo. O resultado do cálculo é registrado nos sinais temporários referentes à quantidade de marcas dos lugares da Rede de Petri e na próxima borda de subida do sinal de *CLOCK* é atribuído ao sinal

```
86    constant PRIOR_T0 : integer := 1;
87    constant PRIOR_T1 : integer := 1;
88    constant PRIOR_T10 : integer := 1;
89    constant PRIOR_T2 : integer := 1;
90    constant PRIOR_T3 : integer := 1;
91    constant PRIOR_T4 : integer := 1;
92    constant PRIOR_T5 : integer := 1;
93    constant PRIOR_T6 : integer := 1;
94    constant PRIOR_T7 : integer := 1;
95    constant PRIOR_T8 : integer := 1;
96    constant PRIOR_T9 : integer := 1;
```

Figura 51: Código em VHDL das Constantes de prioridade das transições.
Fonte: Autoria própria

```
97
98    constant K_P0 : integer := 6;
99    constant K_P1 : integer := 6;
100   constant K_P10 : integer := 255;
101   constant K_P11 : integer := 10;
102   constant K_P12 : integer := 255;
103   constant K_P13 : integer := 5;
104   constant K_P2 : integer := 255;
105   constant K_P3 : integer := 255;
106   constant K_P4 : integer := 255;
107   constant K_P5 : integer := 4;
108   constant K_P6 : integer := 4;
109   constant K_P7 : integer := 255;
110   constant K_P8 : integer := 255;
111   constant K_P9 : integer := 5;
```

Figura 52: Código em VHDL das Constantes de capacidade dos lugares.
Fonte: Autoria própria

referente ao número de marcas atual de cada lugar.

4.3.5 ARQUITETURA - ATRIBUIÇÃO DAS MARCAÇÕES

A próxima parte do código gerado é responsável pela atribuição efetiva das marcações aos lugares. Esta atribuição ocorre de forma síncrona dentro de um processo, respondendo aos sinais de *CLOCK* e *RESET*, conforme mostrado na figura 54. Assim, pode ser observado que quando o sinal de *RESET* é levado a nível alto, o valor inicial de marcação é atribuído a cada lugar. Em caso contrário, a cada borda de subida do sinal de *CLOCK*, o valor de marcação é atualizado de acordo com o valor calculado anteriormente, presente nos sinais temporários. Convém ressaltar que todos os processos são atualizados de forma concorrente, independentemente da ordem em que aparecem codificados. Aqui fica clara a vantagem da representação paralela, inerente aos projetos em lógica reconfigurável.

```
116  begin
117
118     P0Temp <= conv_std_logic_vector((conv_integer(P0)
119           + WT3P0*conv_integer(T3)
120           - WP0T0*conv_integer(T0)),3);
121
122     P1Temp <= conv_std_logic_vector((conv_integer(P1)
123           + WT2P1*conv_integer(T2)
124           - WP1T0*conv_integer(T0)),3);
125
126     P10Temp <= conv_std_logic_vector((conv_integer(P10)
127           + WT8P10*conv_integer(T8)
128           - WP10T9*conv_integer(T9)),8);
129
130     P11Temp <= conv_std_logic_vector((conv_integer(P11)
131           + WT9P11*conv_integer(T9)
132           - WP11T10*conv_integer(T10)),4);
133
134     P12Temp <= conv_std_logic_vector((conv_integer(P12)
135           + WT9P12*conv_integer(T9)
136           - WP12T8*conv_integer(T8)),8);
137
138     P13Temp <= conv_std_logic_vector((conv_integer(P13)
139           + WT10P13*conv_integer(T10)),3);
140
141     P2Temp <= conv_std_logic_vector((conv_integer(P2)
142           + WT0P2*conv_integer(T0)
143           - WP2T1*conv_integer(T1)),8);
144
```

Figura 53: Código em VHDL do cálculo das marcações da rede.
Fonte: Autoria própria

4.3.6 ARQUITETURA - HABILITAÇÃO DAS TRANSIÇÕES

A próxima parte do código gerado é responsável pela habilitação das transições. Esta habilitação ocorre de forma assíncrona e concorrente, dependendo dos valores atuais de marcação dos lugares de entrada e da capacidade disponível dos lugares de saída, conforme mostrado na figura 55. Assim, uma transição estará habilitada se seus lugares de entrada possuem a quantidade de marcas maior ou igual ao peso dos respectivos arcos de entrada e seus lugares de saída possam comportar a quantidade de marcas referentes ao peso dos arcos de saída. Nesta etapa, não são considerados possíveis conflitos entre transições, o que é feito na etapa do código que trata do disparo das mesmas.

```
193 process (CLK, RESET) begin
194    if (RESET = '1') then P0 <= conv_std_logic_vector(0,3);
195    elsif (CLK'event and CLK='1') then
196        P0 <= P0Temp;
197    end if;
198 end process;
199
200 process (CLK, RESET) begin
201    if (RESET = '1') then P1 <= conv_std_logic_vector(0,3);
202    elsif (CLK'event and CLK='1') then
203        P1 <= P1Temp;
204    end if;
205 end process;
206
207 process (CLK, RESET) begin
208    if (RESET = '1') then P10 <= conv_std_logic_vector(0,8);
209    elsif (CLK'event and CLK='1') then
210        P10 <= P10Temp;
211    end if;
212 end process;
213
214 process (CLK, RESET) begin
215    if (RESET = '1') then P11 <= conv_std_logic_vector(0,4);
216    elsif (CLK'event and CLK='1') then
217        P11 <= P11Temp;
218    end if;
219 end process;
```

Figura 54: Código em VHDL da atribuição das marcações da rede.
Fonte: Autoria própria

4.3.7 ARQUITETURA - DISPARO DAS TRANSIÇÕES

A última parte do código gerado, mostrada na figura 56, é responsável pelo disparo das transições, que também ocorre de forma assíncrona e concorrente. O disparo de uma transição depende basicamente de sua habilitação e da solução de possíveis conflitos com outras transições. Assim, se uma transição está habilitada e não está em conflito com outras transições, ela é automaticamente disparada. Em caso contrário, o conflito é analisado e a transição que possuir maior prioridade será disparada.

Nesta etapa ainda poderiam ser analisadas outras condições de disparo, caso fosse necessário. Estas condições dependeriam do tipo de rede que se deseja modelar. Por exemplo, para redes temporizadas podem ser utilizados controles de tempo, que podem ser determinísticos ou estatísticos para condicionar o disparo. Outra situação pode ser o uso de parâmetros externos para condicionar o disparo. No presente estudo, foi utilizada apenas a atribuição de prioridades para a solução dos possíveis conflitos.

```
292     T0_ENABLE <= '1' when
293         (P0 >= WP0T0 and P1 >= WP1T0 and P3 >= WP3T0 and
294         K_P2-P2 >= WT0P2)
295         else '0';
296
297     T1_ENABLE <= '1' when
298         (P2 >= WP2T1 and
299         K_P3-P3 >= WT1P3 and K_P4-P4 >= WT1P4)
300         else '0';
301
302     T10_ENABLE <= '1' when
303         (P11 >= WP11T10 and
304         K_P13-P13 >= WT10P13)
305         else '0';
306
307     T2_ENABLE <= '1' when
308         (K_P1-P1 >= WT2P1)
309         else '0';
310
311     T3_ENABLE <= '1' when
312         (K_P0-P0 >= WT3P0)
313         else '0';
314
315     T4_ENABLE <= '1' when
316         (P5 >= WP5T4 and P6 >= WP6T4 and P8 >= WP8T4 and
317         K_P7-P7 >= WT4P7)
318         else '0';
```

Figura 55: Código em VHDL da habilitação das transições da rede.
Fonte: Autoria própria

4.3.8 ARQUITETURA - DIVISOR DE CLOCK

A próxima etapa de geração do código VHDL, é codificação do circuito divisor de sinal de *clock* que, conforme descrito anteriormente, possibilita que o tempo entre o disparo de uma transição e outra seja especificado em milissegundos. Isto possibilita acompanhamento visual da evolução da marcação da Rede de Petri, sendo muito útil na fase de testes do projeto. Para isto, são utilizados dois contadores, sendo que o primeiro divide o *clock* original gerando em sua saída um *clock* com período de 1 ms. O segundo contador recebe o *clock* do primeiro e gera em sua saída um *clock* final com o período especificado pelo usuário, que é representado pela constante chamada PERIODO. Caso o valor do período especificado seja zero, o *clock* original acionará as transições, não havendo nenhuma divisão. O código VHDL, do divisor de *clock* é mostrado na figura 57.

```
345     T0 <= '1' when (T0_ENABLE = '1')
346          else '0';
347
348     T1 <= '1' when (T1_ENABLE = '1')
349          else '0';
350
351     T10 <= '1' when (T10_ENABLE = '1')
352          else '0';
353
354     T2 <= '1' when (T2_ENABLE = '1')
355          else '0';
356
357     T3 <= '1' when (T3_ENABLE = '1')
358          else '0';
359
360     T4 <= '1' when (T4_ENABLE = '1')
361          else '0';
362
363     T5 <= '1' when (T5_ENABLE = '1')
364          else '0';
365
366     T6 <= '1' when (T6_ENABLE = '1')
367          else '0';
368
369     T7 <= '1' when (T7_ENABLE = '1')
370          else '0';
```

Figura 56: Código em VHDL do disparo das transições da rede.
Fonte: Autoria própria

4.4 CONSIDERAÇÕES

O arquivo contendo código VHDL completo é gerado no mesmo diretório onde está a ferramenta e apresenta extensão vhd. Este arquivo pode ser utilizado diretamente para simulação ou implementação em *hardware* de lógica reconfigurável.

Convém observar que a parte do código referente ao divisor de *clock* só será gerada no caso do usuário escolher um período diferente de zero, por ocasião da solicitação de parâmetros de geração, pela ferramenta.

```
175  process (CLK1)
176     VARIABLE CONTADOR : INTEGER RANGE 0 TO 27000;
177     VARIABLE CONTA_MS : INTEGER RANGE 0 TO PERIODO;
178
179  begin
180     if (PERIODO = 0) then
181         CLK <= CLK1;
182     elsif (CLK1'event and CLK1='1') then CONTADOR := CONTADOR + 1;
183         if (CONTADOR = 27000/2) then CONTA_MS := CONTA_MS + 1;
184             CONTADOR :=0;
185             if (CONTA_MS = PERIODO) then
186                 CONTA_MS :=0;
187                 CLK <= NOT CLK;
188             end if;
189         end if;
190     end if;
191 end process;
```

Figura 57: Código em VHDL do Divisor de Clock

Fonte: Autoria própria

5 ENSAIOS COM O MODELO DE VALIDAÇÃO PROPOSTO

Para validação do método utilizado, o modelo em Rede de Petri da planta industrial hipotética foi implementado em *hardware* de lógica reconfigurável. Para isto, o código VHDL gerado pela ferramenta desenvolvida foi compilado e simulado no *software* Quartus II da Altera e posteriormente gravado em uma FPGA utilizando-se o kit de desenvolvimento Cyclone II também da Altera. Também foi realizada uma comparação de desempenho entre a implementação do modelo em *hardware* com a implemntação em *software*, conforme descrito nas seções a seguir.

5.1 COMPILAÇÃO E SIMULAÇÃO

O código em VHDL gerado pela ferramenta desenvolvida neste estudo foi utilizado na criação de um novo projeto dentro do *software* Quartus II. Em seguida o mesmo foi compilado para a geração dos arquivos necessários para a simulação e posterior gravação em um dispositivo de lógica reconfigurável. O processo de compilação ocorreu normalmente, sem a observância de erros.

Em seguida foi realizada a simulação do *hardware* gerado, utilizando-se a ferramenta de simulação disponível no software Quartus II e cujos resultados podem ser vistos na figura 58. Observa-se que com a ativação do sinal de *RESET*, os lugares da Rede de Petri recebem sua marcação inicial, quando os lugares P3, P8 e P12 possuem uma marca e os demais possuem zero marca. A partir daí, pode-se observar a evolução da marcação da Rede de Petri respondendo às bordas de subida do sinal de *CLOCK*.

5.2 IMPLEMENTAÇÃO EM HARDWARE

A implementação da Rede de Petri em *hardware* foi feita em um dispositivo de lógica reconfigurável. Para isto, o código binário gerado pelo *software* Quartus II a partir do código em VHDL foi carregado em uma FPGA utilizando-se o kit de desenvolvimento Cyclone II

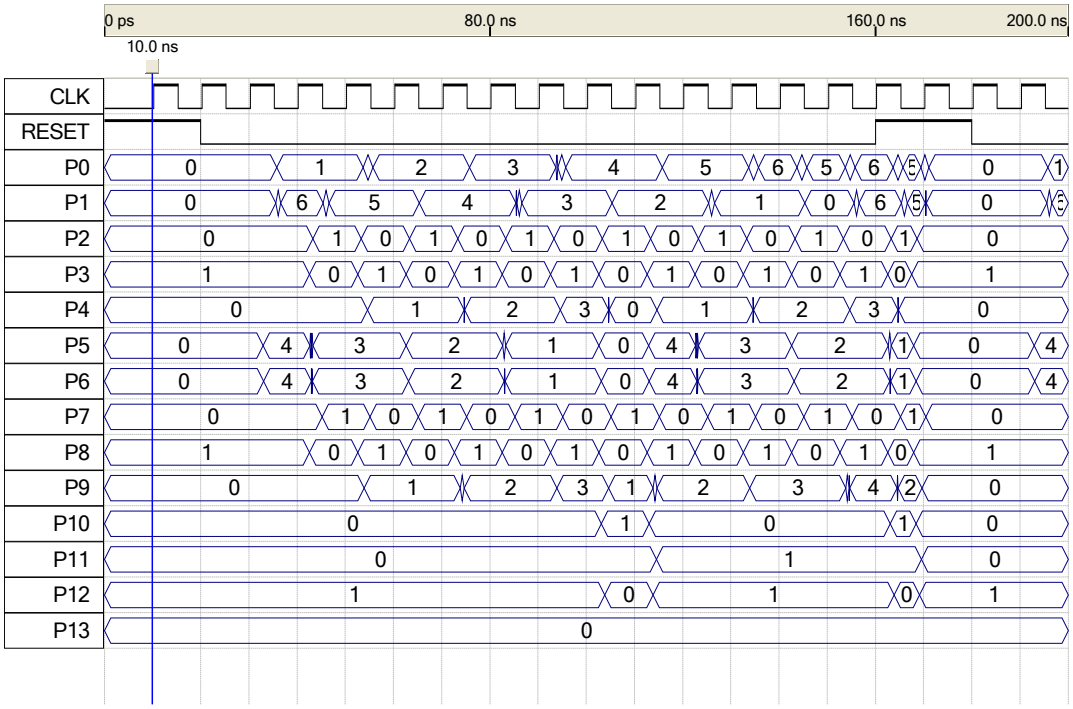


Figura 58: Simulação da Rede de Petri.
Fonte: Autoria própria

da Altera. Além disto, alguns dos terminais referentes aos sinais de quantidades de marcas e de habilitação e disparo de transições foram ligados à led's disponíveis no kit, permitindo sua monitoração, sendo que as entradas de *CLOCK* e *RESET* foram ligadas a botões, permitindo assim um acompanhamento passo a passo da evolução da marcação da Rede de Petri.

Tendo o circuito mostrado o comportamento esperado, conforme a simulação, realizou-se testes com sinais de *CLOCK* com frequência de 27Mhz e de 50Mhz. Em ambos os casos, obteve-se os valores corretos de evolução da marcação, os quais foram monitorados utilizando-se o analisador lógico integrado SignalTap II, disponível na placa de desenvolvimento DE1. A figura 59 mostra o modelo da Rede de Petri em estudo sendo executado em *hardware* .

5.3 ANÁLISE LÓGICA

Nesta seção são mostrados os procedimentos realizados para a análise dos sinais lógicos do modelo da planta industrial hipotética, utilizando o analisador lógico integrado SignalTap II, incluindo as tabelas e os gráficos de evolução da marcação obtidos.

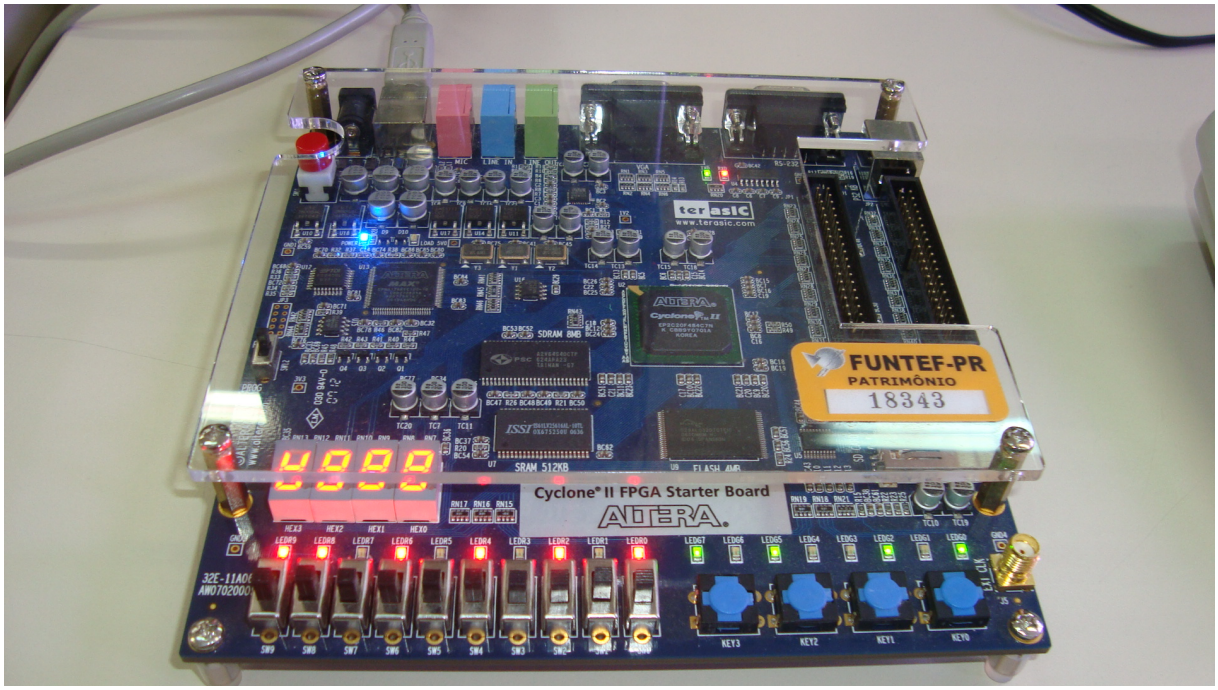


Figura 59: Moodelo sendo executado na placa DE1 do kit Ciclone II.
Fonte: Autoria própria

5.3.1 ANÁLISE LÓGICA DOS SINAIS

Uma etapa importante de avaliação de resultados foi a análise lógica dos sinais do modelo da Rede de Petri em *hardware*. Para isto, foi utilizado o módulo analisador lógico embarcado, SignalTap II, incluído no *software* Quartus II em conjunto com o kit de desenvolvimento Cyclone II. Este módulo permite a análise e captura em tempo real dos sinais lógicos em uma grande quantidade de pontos do circuito gerado. Assim, selecionando-se os sinais referentes aos lugares, foi possível verificar a evolução da marcação da Rede de Petri, mesmo com o modelo sendo executado em uma frequência de *clock* de 27Mhz.

A figura 60 mostra a tela do analisador com os valores capturados a partir de um sinal de disparo estabelecido como a borda de descida do sinal de *RESET*. A frequência de amostragem escolhida foi a própria frequência de *CLOCK*. Assim, uma amostra é obtida a cada ciclo de *CLOCK*, sendo que o número de amostras em cada instante é mostrado na parte superior da figura.

Os demais sinais mostrados são referentes à quantidade de marcas nos lugares da rede. Assim, observa-se que enquanto o sinal de *reset* esta ativado, ou seja com nível alto, os lugares estão carregados com o número de marcas inicial. Quando o sinal de reset é desativado inicia-se a a evolução da marcação. Por exemplo, o lugar P0 recebe uma marca a cada ciclo de *clock* e é subtraído de uma marca a cada dois ciclos de *clock*, tendo como resultado o acréscimo de

uma marca a cada dois ciclos de *clock*. Isto ocorre até que o lugar P0 atinja sua capacidade máxima de 6 unidades, quando então o número de marcas passa a oscilar entre 6 e 5 unidades. O lugar P1 recebe 6 marcas que vão sendo subtraídas a cada dois ciclos de *clock*. Isto prossegue até que o numero de marcas em P1 seja zero, quando então uma nova carga de 6 unidades é recebida. Da mesma forma os demais lugares vão tendo suas marcações alteradas de acordo com a dinâmica da Rede de Petri.

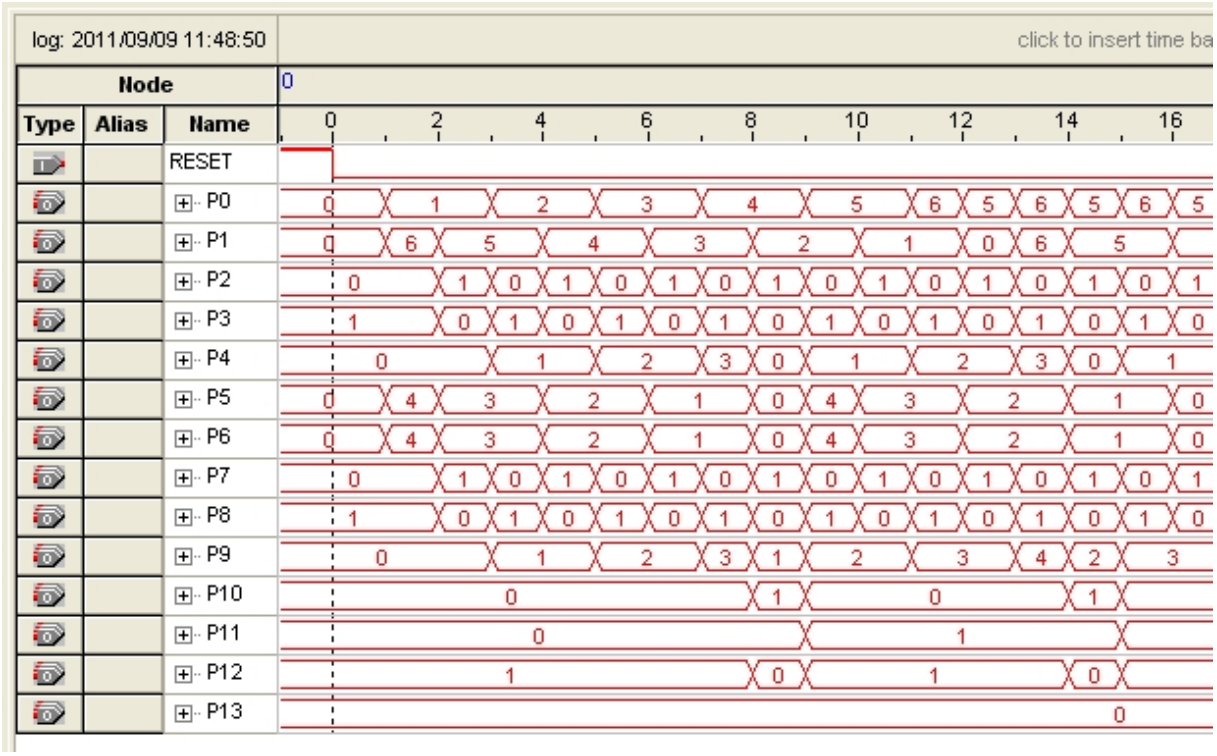


Figura 60: Sinais Lógicos a partir do *RESET*
Fonte: Autoria própria

Como pode ser observado, a evolução da marcação da Rede de Petri ocorre como esperado, atingindo uma estabilização a partir da amostra de número 879, como pode ser visto na figura 61.

5.3.2 TABELA DE MARCAÇÕES

O analisador lógico embarcado SignalTap II permite a exportação dos valores amostrados para um arquivo de valores separados por vírgula, permitindo que fosse criada uma tabela de marcação, da qual a tabela 3 é uma visão parcial com 70 amostras.

A partir da tabela de marcações, foram gerados gráficos que facilitam a visualização do comportamento dinâmico da Rede de Petri, permitindo avaliar com segurança o espaço de marcas ao longo de 400 transições, incluindo a transição de número 364, onde a planta industrial

Tabela 3: Marcações da Rede de Petri

Ciclos	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
1	0	0	0	1	0	0	0	0	1	0	0	0	1	0
2	1	6	0	1	0	4	4	0	1	0	0	0	1	0
3	1	5	1	0	0	3	3	1	0	0	0	0	1	0
4	2	5	0	1	1	3	3	0	1	1	0	0	1	0
5	2	4	1	0	1	2	2	1	0	1	0	0	1	0
6	3	4	0	1	2	2	2	0	1	2	0	0	1	0
7	3	3	1	0	2	1	1	1	0	2	0	0	1	0
8	4	3	0	1	3	1	1	0	1	3	0	0	1	0
9	4	2	1	0	0	0	0	1	0	1	1	0	0	0
10	5	2	0	1	1	4	4	0	1	2	0	1	1	0
11	5	1	1	0	1	3	3	1	0	2	0	1	1	0
12	6	1	0	1	2	3	3	0	1	3	0	1	1	0
13	5	0	1	0	2	2	2	1	0	3	0	1	1	0
14	6	6	0	1	3	2	2	0	1	4	0	1	1	0
15	5	5	1	0	0	1	1	1	0	2	1	1	0	0
16	6	5	0	1	1	1	1	0	1	3	0	2	1	0
17	5	4	1	0	1	0	0	1	0	3	0	2	1	0
18	6	4	0	1	2	4	4	0	1	4	0	2	1	0
19	5	3	1	0	2	3	3	1	0	4	0	2	1	0
20	6	3	0	1	3	3	3	0	1	5	0	2	1	0
21	5	2	1	0	0	2	2	1	0	3	1	2	0	0
22	6	2	0	1	1	2	2	0	1	4	0	3	1	0
23	5	1	1	0	1	1	1	1	0	4	0	3	1	0
24	6	1	0	1	2	1	1	0	1	5	0	3	1	0
25	5	0	1	0	2	0	0	1	0	5	0	3	1	0
26	6	6	0	1	3	4	4	1	0	5	0	3	1	0
27	5	5	1	0	0	4	4	1	0	3	1	3	0	0
28	6	5	0	1	1	4	4	0	1	4	0	4	1	0
29	5	4	1	0	1	3	3	1	0	4	0	4	1	0
30	6	4	0	1	2	3	3	0	1	5	0	4	1	0
31	5	3	1	0	2	2	2	1	0	5	0	4	1	0
32	6	3	0	1	3	2	2	1	0	5	0	4	1	0
33	5	2	1	0	0	2	2	1	0	3	1	4	0	0
34	6	2	0	1	1	2	2	0	1	4	0	5	1	0
35	5	1	1	0	1	1	1	1	0	4	0	5	1	0
36	6	1	0	1	2	1	1	0	1	5	0	5	1	0
37	5	0	1	0	2	0	0	1	0	5	0	5	1	0
38	6	6	0	1	3	4	4	1	0	5	0	5	1	0
39	5	5	1	0	0	4	4	1	0	3	1	5	0	0
40	6	5	0	1	1	4	4	0	1	4	0	6	1	0
41	5	4	1	0	1	3	3	1	0	4	0	6	1	0
42	6	4	0	1	2	3	3	0	1	5	0	6	1	0
43	5	3	1	0	2	2	2	1	0	5	0	6	1	0
44	6	3	0	1	3	2	2	1	0	5	0	6	1	0
45	5	2	1	0	0	2	2	1	0	3	1	6	0	0
46	6	2	0	1	1	2	2	0	1	4	0	7	1	0
47	5	1	1	0	1	1	1	1	0	4	0	7	1	0
48	6	1	0	1	2	1	1	0	1	5	0	7	1	0
49	5	0	1	0	2	0	0	1	0	5	0	7	1	0
50	6	6	0	1	3	4	4	1	0	5	0	7	1	0
51	5	5	1	0	0	4	4	1	0	3	1	7	0	0
52	6	5	0	1	1	4	4	0	1	4	0	8	1	0
53	5	4	1	0	1	3	3	1	0	4	0	8	1	0
54	6	4	0	1	2	3	3	0	1	5	0	8	1	0
55	5	3	1	0	2	2	2	1	0	5	0	8	1	0
56	6	3	0	1	3	2	2	1	0	5	0	8	1	0
57	5	2	1	0	0	2	2	1	0	3	1	8	0	0
58	6	2	0	1	1	2	2	0	1	4	0	9	1	0
59	5	1	1	0	1	1	1	1	0	4	0	9	1	0
60	6	1	0	1	2	1	1	0	1	5	0	9	1	0
61	5	0	1	0	2	0	0	1	0	5	0	9	1	0
62	6	6	0	1	3	4	4	1	0	5	0	9	1	0
63	5	5	1	0	0	4	4	1	0	3	1	9	0	0
64	6	5	0	1	1	4	4	0	1	4	0	10	1	0
65	5	4	1	0	1	3	3	1	0	4	0	0	1	1
66	6	4	0	1	2	3	3	0	1	5	0	0	1	1
67	5	3	1	0	2	2	2	1	0	5	0	0	1	1
68	6	3	0	1	3	2	2	1	0	5	0	0	1	1
69	5	2	1	0	0	2	2	1	0	3	1	0	0	1
70	6	2	0	1	1	2	2	0	1	4	0	1	1	1

Fonte: Autoria própria

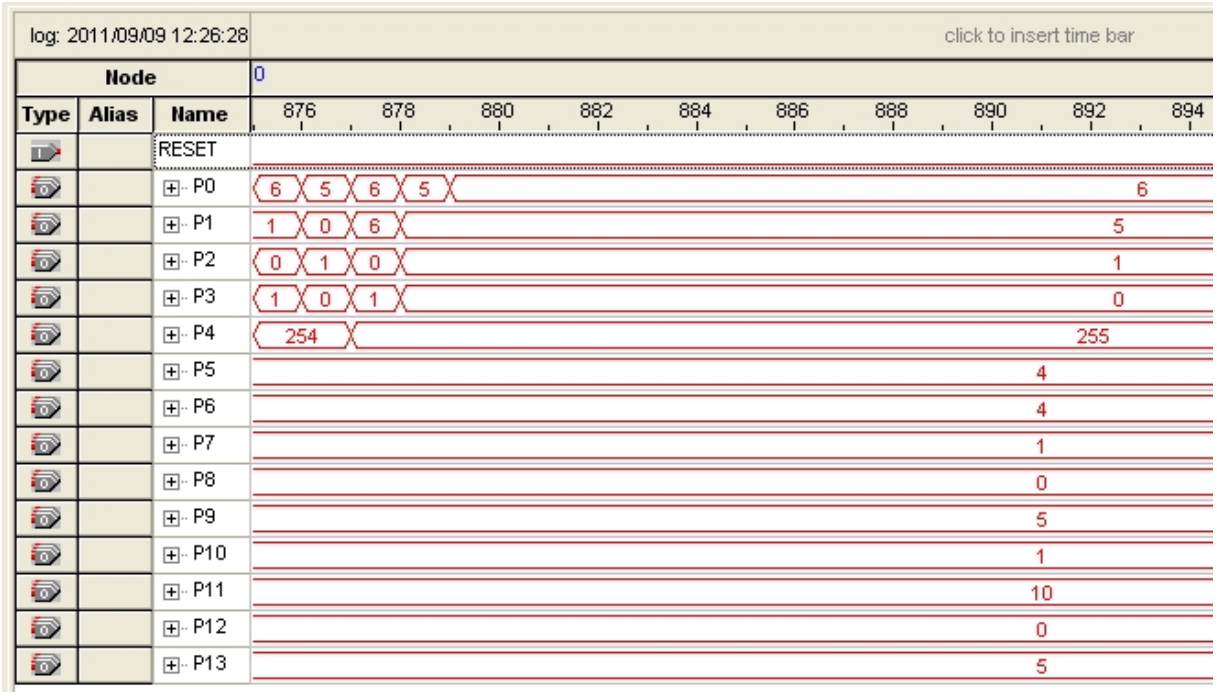


Figura 61: Sinais lógicos mostrando a estabilização da marcação
Fonte: Autoria própria

hipotética atinge sua máxima produção.

O gráfico da figura 62 mostra a evolução da quantidade de marcas nos lugares P0, P1, P2, P3 e P4, relacionados ao setor de produção de canetas, ao longo de 70 ciclos do sinal de *CLOCK*. Neste gráfico pode ser observado que a matéria prima para a produção das canetas é alimentada de duas formas diferentes. No depósito representado por P0, o material entra na taxa de uma unidade por vez até atingir o valor limite de 6 unidades. Durante a entrada deste material ocorre também consumo do mesmo para produção de canetas, o que faz com que sejam necessários dois passos ou o tempo de duas transições para a elevação em uma unidade no depósito. Quando o valor máximo é alcançado a transição fonte não irá mais disparar até que uma marca seja consumida, liberando capacidade para mais marcas. Na próxima transição o número de marcas volta para o valor máximo. Isto faz com que o número de marcas em P0 fique oscilando entre 5 e 6 marcas.

No depósito representado pelo lugar P1 a entrada de material ocorre em lotes de 6 unidades. À medida que as canetas vão sendo produzidas, este material vai sendo consumido até que a quantidade de marcas seja zerada. Na próxima transição uma nova batelada com 6 unidades é depositada pela transição fonte T2. O lugar P2 indica que uma caneta está sendo produzida e o lugar P3 indica que a máquina está disponível para produção. Assim, a marcação em P2 é sempre complementar à marcação em P3, ou seja, quando um destes lugares possui uma marca a outro está zerado e vice-versa. A Marcação em P4 indica a quantidade de canetas produzidas e

ainda não consumidas pelo setor de embalagens. Quando são produzidas pelo menos 3 canetas e caso existam pelo menos 2 lápis no depósito representado pelo lugar P9, a transição T8 será disparada representando o consumo de 2 lápis e 3 canetas pelo setor de embalagens, fazendo com que as marcas nos lugares P9 e P4 sejam subtraídas nesta quantidade.

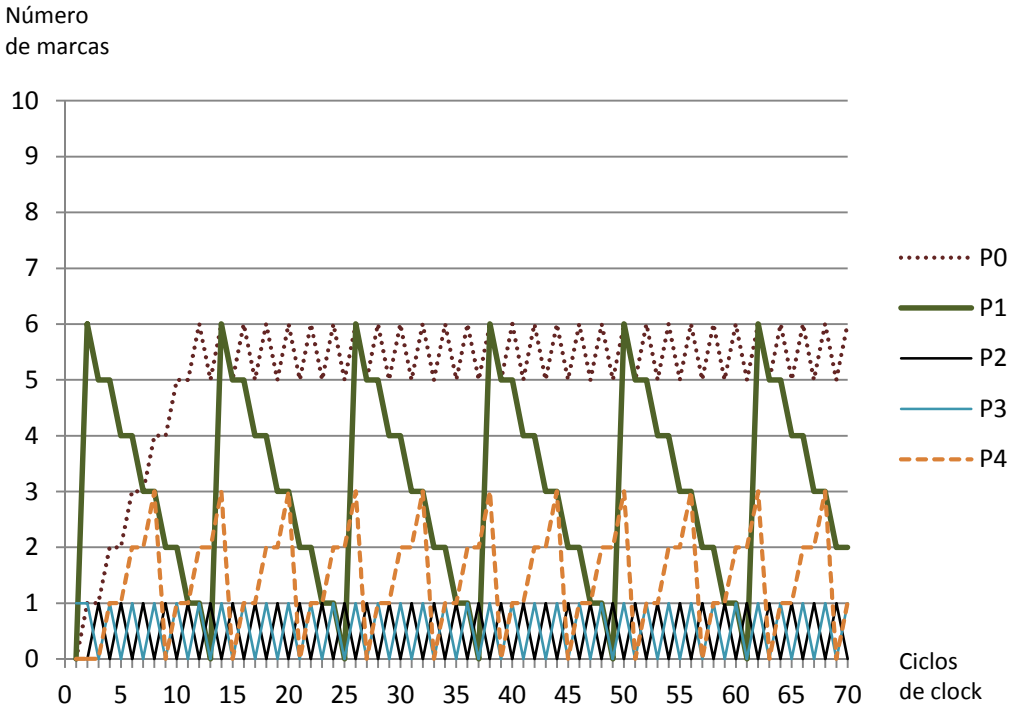


Figura 62: Marcações da Produção de Canetas
Fonte: Autoria própria

O gráfico da figura 63 mostra a evolução da quantidade de marcas nos lugares P5, P6, P7, P8 e P9, relacionados ao setor de produção de lápis, ao longo de 70 ciclos do sinal de *CLOCK*. Neste gráfico pode ser observado que a matéria prima para a produção dos lápis, ocorre em batelada de 4 unidades para os dois depósitos representados pelos lugares P5 e P6. À medida que os lápis vão sendo produzidos, este material vai sendo consumido até que a quantidade de marcas seja zerada. Na próxima transição uma nova batelada com 4 unidades é depositada pelas transições fonte T6 e T7. Como a evolução da marcação para estes dois lugares é a mesma os seus gráficos se sobrepõem. Assim, para permitir uma melhor visualização, o gráfico da marcação de P6 foi representado em pontilhado. O lugar P7 indica que um lápis está sendo produzido e o lugar P8 indica que a máquina está disponível para produção. Assim, a marcação em P7 é sempre complementar à marcação em P8, ou seja, quando um destes lugares possui uma marca a outro está zerado e vice-versa.

A Marcação em P9 indica a quantidade de lápis produzidos e ainda não consumidos pelo

setor de embalagens. Quando são produzidos pelo menos 2 lápis e caso existam pelo menos 3 canetas no depósito representado pelo lugar P4, a transição T8 será disparada representando o consumo de 2 lápis e 3 canetas pelo setor de embalagens, fazendo com que as marcas nos lugares P9 e P4 sejam subtraídas nesta quantidade. É interessante observar que o tempo gasto para a produção de lápis é o mesmo que para a produção de canetas, mas o consumo pelo setor de embalagens é diferente. Assim, enquanto são produzidas 3 canetas, também são produzidos 3 lápis, mas enquanto as 3 canetas são consumidas, apenas 2 lápis serão consumidos. Isto faz com que a quantidade de marcas no depósito representado pelo lugar P9 aumente até atingir sua capacidade máxima de 5 lápis, como pode ser visto no gráfico.

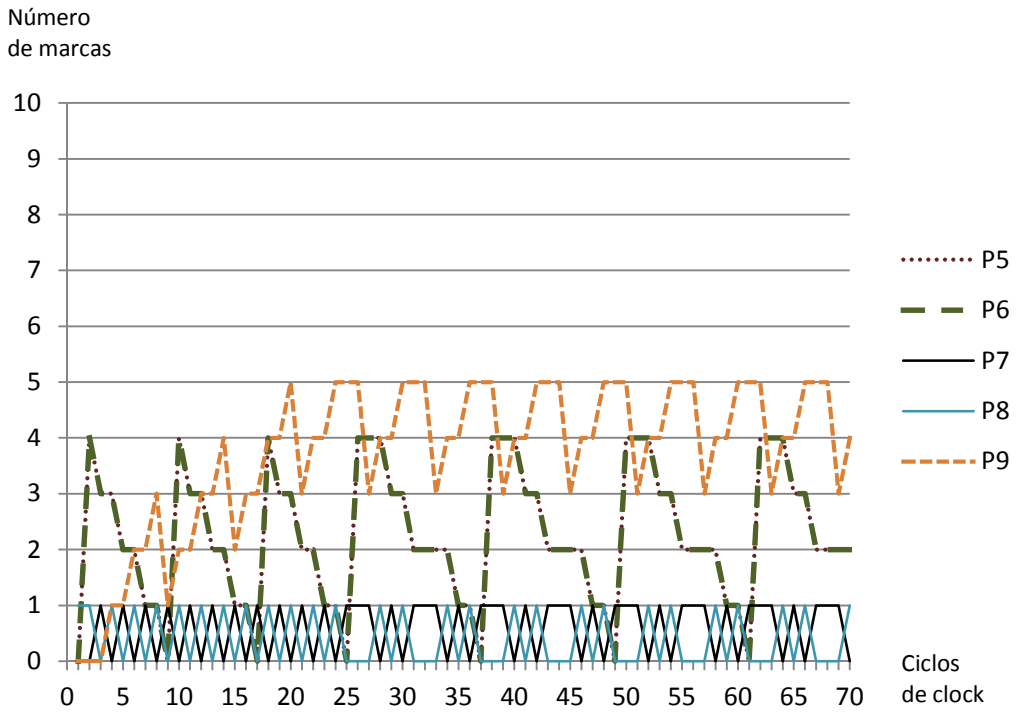


Figura 63: Marcações da Produção de Lápis
Fonte: Autoria própria

O gráfico da figura 64 mostra a evolução da quantidade de marcas nos lugares P10, P11, P12, e P13, relacionados ao setor de produção dos conjuntos contendo 2 lápis e 3 canetas, ao longo de 70 ciclos do sinal de *CLOCK*. Neste gráfico o lugar P10 indica que um conjunto contendo 3 canetas e 2 lápis está sendo embalado e aparece de forma completar ao lugar P12 que sinaliza a disponibilidade da máquina. A marcação no lugar P11 indica a quantidade de conjuntos produzidos e ainda não consumidos pela etapa de embalagem em caixas. Como pode ser visto, quando o número de conjuntos neste depósito atinge 10 unidades, a transição T10 é disparada, indicando o consumo dos mesmos e fazendo com que a quantidade de marcas seja

zerada. Os conjuntos são então embalados em caixas contendo 10 unidades, cuja quantidade é indicada pelo número de marcas no lugar P13.

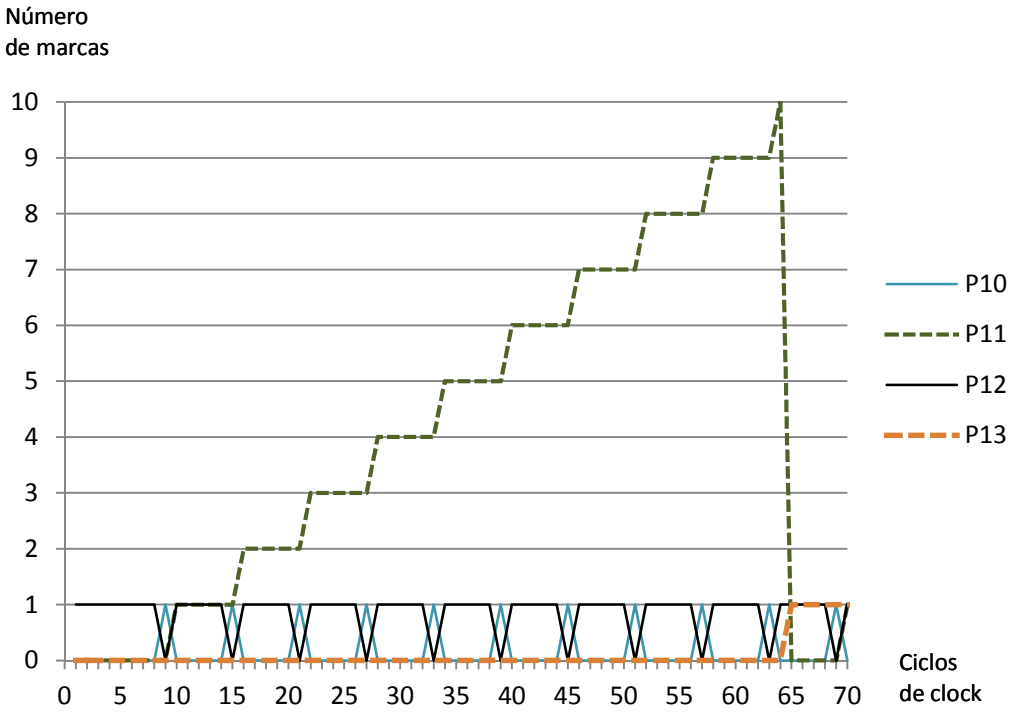


Figura 64: Marcações da Produção de Conjuntos
Fonte: Autoria própria

O gráfico da figura 65 mostra a evolução da quantidade de marcas nos lugares P10, P11, P12, e P13, ao longo de 400 ciclos do sinal de *CLOCK*, permitindo a visualização da evolução da quantidade de caixas contendo 10 conjuntos de 2 lápis e 3 canetas. Como pode ser observado, o número de caixas atinge um valor máximo de 5 unidades, que é a capacidade do depósito representado pelo lugar P13.

5.4 COMPARAÇÃO DE DESEMPENHO ENTRE HARDWARE E SOFTWARE

Com o objetivo de avaliar as vantagens de implementação do modelo de Redes de Petri em *hardware*, foi desenvolvida uma ferramenta que gera um *software* simulador deste modelo, em linguagem C, a partir do arquivo PNML, possibilitando assim, uma comparação de desempenho com a versão em *hardware*. Desta forma, foi gerado um *software* simulador correspondente à planta industrial hipotética, destinada à produção de lápis e canetas, cuja estrutura é similar àquela do código em linguagem VHDL, possuindo porém, duas diferenças fundamentais.

A primeira diferença refere-se ao sinal de sincronismo ou *clock* utilizado para comandar

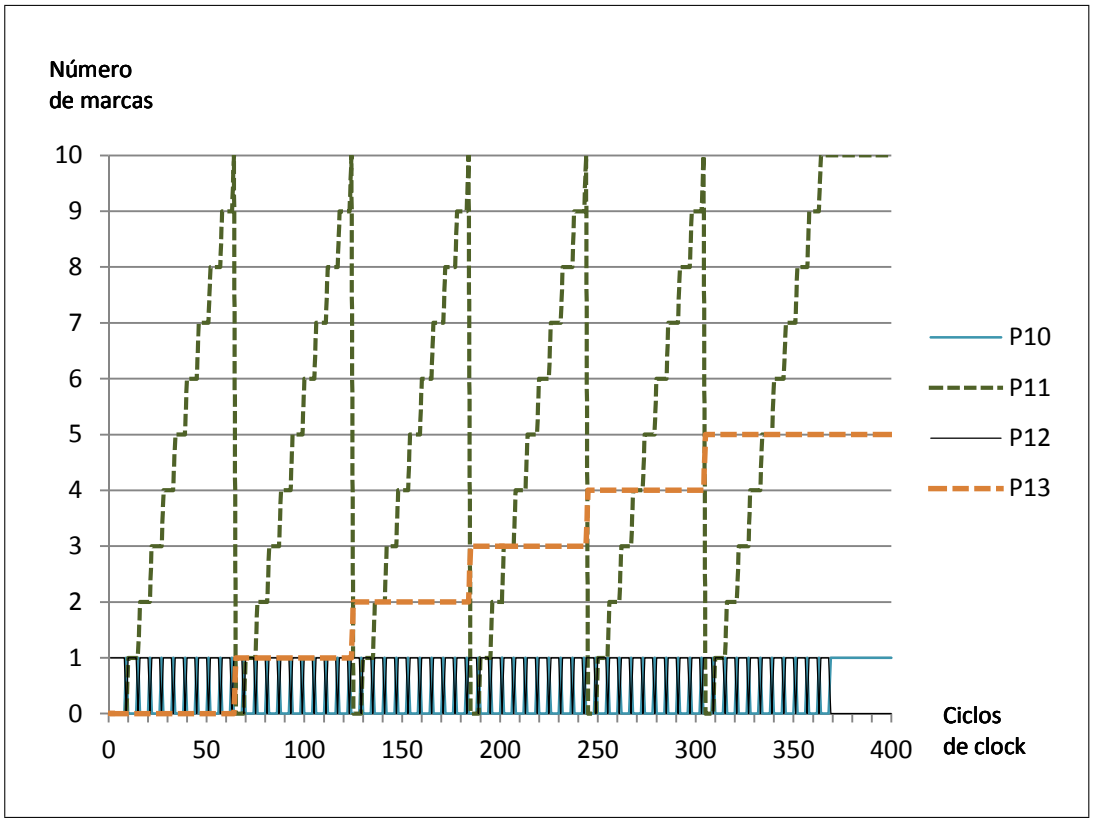


Figura 65: Marcações da Embalagem em Caixas

Fonte: Autoria própria

as mudanças de estado e disparo das transições. Enquanto no modelo em *hardware*, o sinal de *clock* é proveniente de um circuto na placa de desenvolvimento DE1, no modelo em *software*, este sinal é produzido artificialmente, sendo alterado a cada ciclo de varredura de atualização dos sinais envolvidos no processo. A segunda diferença refere-se ao processo de atualização dos sinais envolvidos no processo, representados por variáveis inteiras. Enquanto no modelo em *hardware*, a atualização ocorre de forma paralela, no modelo em *software*, esta atualização ocorre de forma concorrente, simulando um paralelismo, utilizando *threads*.

Este *software* foi executado em uma plataforma RISC ARM7 com frequência de *clock* de 32Mhz e em um computador pessoal com *clock* de 3Ghz. O tempo necessário para que a planta atingisse sua produção máxima, aproximado para 400 transições, foi de 5259μs para a plataforma ARM7 e de 18,4μ para o PC. Para a execução em *hardware*, com um *clock* de 27Mhz o tempo para que a planta atingisse sua máxima produção foi de 14,81μs. Foi realizada também a execução em *hardware* utilizando-se o sinal de *clock* máximo da placa de desenvolvimento, cuja frequência é de 50Mhz. Neste caso, o tempo para que o simulador atinja a produção máxima é de 8μs. Para tornar mais clara a comparação, foi obtido um valor normalizado de velocidade, dividindo-se o tempo gasto na implementação mais lenta, pelas demais implementações. A tabela 4 mostra a comparação de desempenho normalizado, que visa apresentar quantas vezes o

modelo executado em *hardware* é mais rápido que e o modelo executado em *software*.

Tabela 4: Tabela comparativa de desempenho

Tecnologia	Software		Hardware	
Plataforma	ARM7	PC	DE1	DE1
Frequência de *clock* (Mhz)	32	3000	27	50
Tempo gasto (μs)	5259,00	18,40	14,81	8,00
Desempenho normalizado(x)	1,00	285,82	355,10	657,38

Fonte: Autoria própria

6 CONCLUSÃO

Neste capítulo são expostas as conclusões gerais sobre este trabalho de pesquisa e são comentadas algumas possibilidades de complementação em trabalhos futuros.

6.1 CONCLUSÕES GERAIS

Analisando-se os gráficos exibidos na seção 4.5, que descrevem a evolução da marcação no modelo em Rede de Petri da planta industrial hipotética, pode-se constatar que não ocorrem desvios em relação ao comportamento esperado. Este resultado também pode ser verificado por uma análise cuidadosa dos valores de marcação que aparecem na tela de sinais exposta nas figuras 48 e 49, que foram obtidos através do circuito analisador lógico embarcado SignalTap II, disponível na placa DE1 e que deram origem à tabela 2 e aos gráficos de evolução da marcação.

Outra característica, que pôde ser visualizada na tela de sinais do analisador SignalTap II, foi o perfeito sincronismo e o paralelismo das mudanças de marcação. Além disso, observou-se que, mesmo com execução do modelo em *hardware* com valores de frequência de clock de 50Mhz, não houve qualquer tipo de desvio quanto à sequência de marcação a ser seguida. Assim, em todos os casos, a marcação do modelo da planta industrial hipotética evoluiu, representando adequadamente a produção da quantidade final de caixas contendo conjuntos de lápis e canetas. Esta quantidade é limitada pela capacidade do lugar que representa o depósito final, estipulada em 5 unidades.

Para os lugares que não tiveram o valor de capacidade deliberadamente especificado, a evolução da marcação também ocorreu de forma adequada, de modo que a quantidade máxima de marcas não ultrapassou 255 unidades, que é o valor de capacidade atribuído automaticamente pela ferramenta utilizada. Convém lembrar que este valor pode ser alterado para se adequar às diferentes aplicações de modelagem sem que isto tenha influência no desempenho do processo, pois a comparação para verificação de atingimento do limite é realizada por lógica combinacional cujo tempo de resposta não é alterado pelo tamanho em bits, da palavra utilizada.

Na seção 4.6 foi realizada uma comparação de desempenho entre o modelo executando em

hardware com frequência de clock de 27Mhz e um modelo executando em *software* em uma plataforma ARM7 com frequência de clock de 33Mhz e em um PC com frequência de clock de 3Ghz. O resultado da comparação mostrou que para realizar a evolução de marcação até o atingimento da produção máxima do modelo hipotético da planta industrial, o tempo gasto pelo *hardware* foi de 14,81μs enquanto que o tempo gasto pelo *software* foi de 5259,00μs na plataforma ARM7 e de 18,40μs. Assim, em relação à plataforma ARM7, o desempenho do modelo em *hardware* foi de 355,10 vezes superior com um clock de 27Mhz e de 657,38 vezes superior para um clock de 50Mhz.

Em relação ao modelo executado no PC, observou-se que mesmo com uma frequência de clock muito inferior àquela utilizada pelo *software*, o desempenho do *hardware* foi cerca de 1,24 vezes superior para clock de 27Mhz e de 2,30 vezes superior para clock de 50Mhz. Assim, pode-se concluir que devido a fatores como paralelismo real, cálculos e comparações realizados de forma combinacional, e a ausência de ciclos de leitura e execução de instruções, conduzem à esta superioridade no desempenho do modelo em *hardware*. Além disto, pode-se considerar também a redução em custo, tamanho e consumo de energia da implementação do modelo em *hardware*, como fatores para sua escolha.

Conforme descrito na seção 3.8, a parte de geração do código VHDL da ferramenta desenvolvida, apresenta modificações em relação ao trabalho desenvolvido por Lima (2009), o qual foi utilizado como base para o desenvolvimento desta etapa.

A primeira modificação refere-se à descrição de todos os lugares da Rede de Petri, como terminais do tipo (BUFFER) no *hardware* a ser gerado. Isto permite que os dados sejam escritos ou lidos nos mesmos, simplificando o processo de geração do código VHDL e a visualização da marcação. No entanto, é importante ressaltar que esta simplificação também gera uma limitação, pois, como os terminais gerados também são lugares, que precisam ter seus valores atualizados após o disparo de certas transições, eles não podem ser declarados como do tipo entrada ou (IN), os quais não aceitam operações de escrita. Assim, caso seja necessária a interação do processo modelado, com um sinal de controle externo, deve-se criar manualmente um terminal de entrada, que esteja associado diretamente à lógica de habilitação de uma transição. Por outro lado, para comandar o acionamento de algum dispositivo ou circuito externo, os lugares com tipo (BUFFER) não necessitam ser transformados para tipo (OUT), pois já têm esta função.

Outra modificação se refere à forma como são solucionados possíveis conflitos entre transições, onde não apenas a prioridade é analisada, mas também a possibilidade de disparo em função da capacidade dos lugares que receberão as marcas. Para isto, são consideradas todas as transições

habilitadas, mesmo que algumas não participem diretamente do conflito. Isto possibilita o disparo adequado de um número máximo de transições habilitadas em um determinado momento. Por exemplo, caso três transições estejam habilitadas e em conflito, será disparada a de maior prioridade e se possível a de prioridade intermediária. Supondo-se porém, que o disparo desta última transição intermediária acarrete um número de marcas acima da capacidade de algum de seus lugares de saída, ela não será disparada e será considerada a possibilidade do disparo da transição de menor prioridade. Toda esta análise é realizada por lógica combinacional, a qual será tanto mais complexa quanto maior for a possibilidade de conflitos no modelo representado. Isto acarreta por vezes o encadeamento de portas lógicas, gerando um atraso no sinal de decisão, o que pode limitar a frequência de operação do modelo. Ou seja, para que não ocorram falhas na evolução da marcação, o tempo do sinal de decisão deve ser inferior ao período do sinal de clock.

Finalmente, com este trabalho de pesquisa, foi possível perceber a importância da utilização de ferramentas adequadas para o desenvolvimento de *hardware* de lógica reconfigurável. Também foi possível verificar algumas vantagens da utilização de modelagem em Redes de Petri para sistemas assíncronos e concorrentes. Além disto, a implementação de modelos de Rede de Petri em dispositivos de lógica reconfigurável permite uma representação fidedigna de seu comportamento, devido a dois fatores. O primeiro fator é o paralelismo real, onde todas as transições e atualizações de marcação ocorrem de forma simultânea. O segundo fator é a independência de ciclos de busca e execução de instruções, possibilitando tingir índices de desempenho bastante superiores àqueles obtidos quando o sistema é executado em *software*. Isto abre um grande leque de possíveis aplicações para esta tecnologia em sistemas reativos de controle de processos, em casos onde desempenho, economia e tamanho reduzido sejam fatores relevantes.

Acredita-se que o uso da ferramenta desenvolvida neste trabalho possa contribuir para facilitar a implementação de sistemas modelados em Redes de Petri, em dispositivos de lógica reconfigurável. Para isto, procurou-se expor de forma detalhada todo o método e a ferramenta desenvolvida, incluindo o seu código. Fato este, que não ocorre em alguns dos trabalhos semelhantes pesquisados. Considera-se ainda, que esta ferramenta pode ser aprimorada em muitos aspectos, servindo inclusive como base para trabalhos futuros.

6.2 TRABALHOS FUTUROS

Existem diversas possibilidades de trabalhos futuros partindo da ferramenta desenvolvida neste estudo, sendo que na relação abaixo, são apresentadas algumas possibilidades.

- Integração do *software* de captura da Rede de Petri com a ferramenta para geração de código VHDL.
- Tornar a ferramenta mais universal, fazendo com que ela seja capaz de traduzir diferentes formatos de arquivo PNML referentes redes temporizadas e redes de alto nível.
- Adicionar à ferramenta uma etapa de pré-processamento para análise formal das Redes de Petri modeladas.
- Adaptação desta ferramenta para ambiente gráfico, facilitando assim, sua operação.
- Integrar a ferramenta em um *software* de geração de hardware de controle em sistemas digitais.

APÊNDICE A – CÓDIGO FONTE DA FERRAMENTA DESENVOLVIDA

```
//--------------------------------------------------------------------------
// PN2VHDL.c
// Lê arquivo PNML nome.xml e gera um arquivo VHDL nome.vhd correspondente
//--------------------------------------------------------------------------

//-------------------------Cabeçalho das Funções-----------------------------

#include <stdio.h>
#include <stdlib.h>
void le_file(char *str);        // Lê o arquivo PNML
int get_places(char *str);      // Busca os lugares da Rede de Petri
int get_transitions(char *str);// Busca as transições da Rede de Petri
int get_arcs(char *str);        // Busca os arcos da Rede de Petri
void ler(char *atual,char *p); // Lê um texto
int ler_val(char *atual);       // Lê um valor inteiro
float ler_float(char *atual);  // Lê um valor decimal
void make_matriz(int matrix[][20], int num_places, int num_transitions,
 int num_arcs, int tipo);       // Cria matrizes da Rede de Petri
void mostra_matriz(int matrix[][20], int num_places, int num_transitions,
 int tipo);    // Mostra matrizes da Rede de Petri
int index_p(char *id, int max); // Indexa os lugares da Rede de Petri
int index_t(char *id, int max); // Indexa os Transições da Rede de Petri
void ajusta_prioridade(int num_places, int num_transitions); /* Ajusta as
              prioridades das transições para solução de conflitos */
void le_file_name(); // Lê o nome do arquivo PNML
int le_periodo();    // Lê o valor do período entre transições
void escreve_file(); // Escreve o arquivo VHDL
int num_bits(int a); // Calcula número de bits para conter um inteiro

//--------------------------------------------------------------------------
```

```
//-------------------------Descrição das Estruturas----------------------------

struct place        // Estrutura para Lugares
{
 char id[10];
 int marking;
 int capacity;
};

struct transition   // Estrutura para Transições
{
 char id[10];
 float rate;
 int timed;
 int infiniteServer;
 int priority;
};

struct arc          // Estrutura para Arcos
{
 char id[20];
 char source[10];
 char target[10];
 int weight;
 int tagged;
 char type[10];
};

//-------------------------------------------------------------------------------


//-------------------------Variáveis Globais-----------------------------------
// Obs, as strings e matrizes foram arbitrariamente dimensionads para leitura
// de arquivos PNML de até 64000 caracteres, com no máximo 20 lugares,
// 20 transições e 50 arcos.
// Caso seja necessário, estes valores podem ser alterados.
//-------------------------------------------------------------------------------

char nome[25];
char file_name[25];
char name[25];
char str[64000]; // String para arquivo PNML de até 64000 caracteres
struct place lugar[20];  // Vetor para até 20 lugares
struct transition trans[20];// Vetor para até 20 transições
struct arc arco[50];     // Vetor para até 50 arcos
int matriz_i[20][20];    // Matriz de entrada 20 x 20
int matriz_o[20][20];    // Matriz de saída 20 x 20
int matriz[20][20];      // Matriz de incidência 20 x 20
int num_places;
int num_transitions;
int num_arcs;
int fclock;
int periodo;

//-------------------------------------------------------------------------------
```

```
//-------------------------Programa Principal--------------------------------

main()
{
 le_file(str); // Lê arquivo PNML.
 num_places=get_places(str);          // Busca os lugares da Rede de Petri.
 num_transitions=get_transitions(str);// Busca as transições da Rede de Petri.
 num_arcs=get_arcs(str);              // Busca os arcos da Rede de Petri.
 make_matriz(matriz_i,num_places,num_transitions,num_arcs,1);// Cria matrizes
 make_matriz(matriz_o,num_places,num_transitions,num_arcs,2);// da Rede de
 make_matriz(matriz,num_places,num_transitions,num_arcs,3);  // Petri.
 mostra_matriz(matriz_i,num_places,num_transitions,1);// Mostra matrizes
 mostra_matriz(matriz_o,num_places,num_transitions,2);// da Rede de
 mostra_matriz(matriz,num_places,num_transitions,3);  // Petri.
 ajusta_prioridade(num_places, num_transitions); // Ajusta prioridades.
 le_periodo(); // Lê o valor do período entre transições.
 escreve_file(num_places,num_transitions,num_arcs); // Escreve o arquivo VHDL.
}

//-----------------------------------------------------------------------------


//-------------------------------Funções---------------------------------------

//-----------Solicita nome e abre arquivo PNML para leitura--------------------

void le_file(char *str)
{
 printf("Digite o nome do arquivo a ser lido: ");// Solicita o nome do arquivo
 scanf("%s",nome);                               // PNML a ser lido.
 FILE *fp;
 if((fp=fopen(nome,"r"))==NULL){        // Abre o arquivo para leitura.
  printf("Read error ocurred");
  getchar();
  getchar();
  exit(1);
 }
 printf("O arquivo foi aberto com sucesso\n\n");
 int i=0;
 while((str[i]=fgetc(fp))!=EOF)i++;     // Lê o arquivo para uma string.
 str[i]='\0';                           // Encerra string.
 fclose(fp);                            // Fecha o arquivo.
 printf("-------------------------------------------------------\n");
}

//-----------------------------------------------------------------------------
```

```
//-----------------Obtêm os Lugares no arquivo PNML----------------------------

int get_places(char *str)
{
 printf("\t\tLugares da Rdp\n\n"); // Mostra título da tabela.
 char *onde,*atual;
 atual=str;              // Aponta para início da string com o arquivo lido.
 int i=0;
 while((onde=strstr(atual,"<place id="))!=NULL){       // Localiza tag,
  atual=onde+11;                                      // aponta dado,
  ler(atual,lugar[i].id);                             // lê e
  printf("Lugar = %s\n",lugar[i].id);                 // mostra id do Lugar.
  onde=strstr(atual,"<initialMarking>\n<value>Default,");// Localiza tag,
  atual=onde+32;                                      // aponta dado,
  lugar[i].marking=ler_val(atual);                    // lê e
  printf("Marcas = %d\n",lugar[i].marking);       // mostra a Marcação Inicial.
  onde=strstr(atual,"<capacity>\n<value>");           // Localiza tag,
  atual=onde+18;                                      // aponta dado,
  lugar[i].capacity=ler_val(atual);                   // lê e
  printf("Capacidade = %d\n\n",lugar[i].capacity);    // mostra a Capacidade.
  i++;                                              //Conta o número de Lugares.
 }
 printf("\nA Rdp contem %d lugares.\n",i);        //Mostra o número de Lugares.
 printf("-------------------------------------------------------\n");
 return i;
}
//-------------------------------------------------------------------------------

//-----------------Obtêm as Transições no arquivo PNML-------------------------

int get_transitions(char *str)
{
 printf("\t\tTransicoes da Rdp\n\n"); // Mostra título da tabela.
 char *onde,*atual;
 atual=str;              // Aponta para início da string com o arquivo lido.
 int i=0;
 while((onde=strstr(atual,"<transition id="))!=NULL){// Localiza tag,
  atual=onde+16;                                    // aponta dado,
  ler(atual,trans[i].id);                           // lê e
  printf("Transicao = %s\n",trans[i].id);           // mostra id da Transição.
  onde=strstr(atual,"<rate>\n<value>");             // Localiza tag,
  atual=onde+14;                                    // aponta dado,
  trans[i].rate=ler_float(atual);                   // lê e
  printf("Taxa = %f\n",trans[i].rate);              // mostra Taxa.
  onde=strstr(atual,"<timed>\n<value>");            // Localiza tag,
  atual=onde+15;                                    // aponta dado,
  trans[i].timed=ler_val(atual);                    // lê e
  printf("Temporizada = %d\n",trans[i].timed);      // mostra Tempo.
  onde=strstr(atual,"<infiniteServer>\n<value>");   // Localiza tag,
  atual=onde+24;                                    // aponta dado,
  trans[i].infiniteServer=ler_val(atual);           // lê e
  printf("infiniteServer = %d\n",trans[i].infiniteServer);// mostra Tipo.
  onde=strstr(atual,"<priority>\n<value>");         // Localiza tag,
  atual=onde+18;                                    // aponta dado,
  trans[i].priority=ler_val(atual);                 // lê e
  printf("Prioridade = %d\n\n",trans[i].priority);  // mostra Prioridade.
  i++;                                            //Conta o número de Transições.
 }
 printf("\nA Rdp contem %d transicoes.\n",i);   //Mostra o número de Transições.
 printf("-------------------------------------------------------\n");
 return i;
}
//-------------------------------------------------------------------------------
```

```
//-----------------Obtêm os Arcos no arquivo PNML-------------------------------

int get_arcs(char *str)
{
 printf("\t\tArcos da Rdp\n\n");   // Mostra título da tabela.
 char *onde,*atual;
 atual=str;    // Aponta para início da string com o arquivo lido.
 int i=0;
 while((onde=strstr(atual,"<arc id="))!=NULL){     // Localiza tag,
  atual=onde+9;                                    // aponta dado,
  ler(atual,arco[i].id);                           // lê e
  printf("Arco = %s\n",arco[i].id);                // mostra id do Arco.
  //--
  onde=strstr(atual,"source=");                    // Localiza tag,
  atual=onde+8;                                    // aponta dado,
  ler(atual,arco[i].source);                       // lê e
  printf("Source = %s\n",arco[i].source);          // mostra origem.
  //--
  onde=strstr(atual,"target=");                    // Localiza tag,
  atual=onde+8;                                    // aponta dado,
  ler(atual,arco[i].target);                       // lê e
  printf("Target = %s\n",arco[i].target);          // mostra destino.
  //--
  onde=strstr(atual,"<inscription>\n<value>Default,"); // Localiza tag,
  atual=onde+29;                                       // aponta dado,
  arco[i].weight=ler_val(atual);                       // lê e
  printf("Peso = %d\n",arco[i].weight);                // mostra peso.
  //--
  onde=strstr(atual,"<tagged>\n<value>");          // Localiza tag,
  atual=onde+16;                                   // aponta dado,
  arco[i].tagged=ler_val(atual);                   // lê e
  printf("Tag = %d\n",arco[i].tagged);             // mostra rotulação.
  //--
  onde=strstr(atual,"<type value=");               // Localiza tag,
  atual=onde+13;                                   // aponta dado,
  ler(atual,arco[i].type);                         // lê e
  printf("Tipo = %s\n\n",arco[i].type);            // mostra tipo.
  //--
  i++;                                    //Conta o número de Arcos.
 }
 printf("\nA Rdp contem %d arcos.\n",i); //Mostra o número de Arcos.
 printf("-----------------------------------------------------\n");
 return i;
}

//-------------------------------------------------------------------------------
```

```
//-------------------------Lê um nome-----------------------------------------

void ler(char *atual,char *p)
{
 int j=0;
 while((p[j]=*atual)!='"'){    // Lê os caracteres de um nome
  j++;                         // antes das aspas para uma string.
  atual++;
 }
 p[j]='\0';                    // Encerra string.
}

//------------------------------------------------------------------------------

//-------------------------Lê valor Inteiro-----------------------------------

int ler_val(char *atual)
{
 char p[10];
 int j=0;
 while((p[j]=*atual)!='<'){    // Lê os caracteres de um valor inteiro
  j++;                         // antes do tag < para uma string.
  atual++;
 }
 p[j]='\0';                    // Encerra string.
 if(strstr(p,"true")!=NULL)return 1; //Se encontrar o texto "true" retorna 1.
 if(strstr(p,"false")!=NULL)return 0;//Se encontrar o texto "false" retorna 0.
 else return atoi(p); //Caso contrário, converte o texto para um valor inteiro.
}

//------------------------------------------------------------------------------

//-------------------------Lê valor Decimal-----------------------------------

float ler_float(char *atual)
{
 char p[10];
 int j=0;
 while((p[j]=*atual)!='<'){    // Lê os caracteres de um valor decimal
  j++;                         // antes do tag < para uma string.
  atual++;
 }
 p[j]='\0';                    // Encerra string.
 return atof(p);    // Converte os caracteres para um valor decimal.
}

//------------------------------------------------------------------------------
```

```
//--------------------Constrói Matrizes da Rede de Petri-----------------------

void make_matriz(int matrix[][20], int num_places, int num_transitions,
int num_arcs, int tipo)
{
 int i,j,k;
 for(i=0;i<num_places;i++)// Monta uma matriz com todos os valores zerados
 for(j=0;j<num_transitions;j++)matrix[i][j]=0;
 if(tipo==1 || tipo==3)  // Se a matirz for de entrada (tipo 1) ou
 {                       // se a matriz for de incidência (tipo 3)
  k=0;                   // verifica quando um arco tem origem em um Lugar i
  do{                    // e destino em uma Transição j.
     if((i=index_p(arco[k].source,num_places))==-1)continue;
     if((j=index_t(arco[k].target,num_transitions))==-1)continue;
     matrix[i][j]-=arco[k].weight; // Decrementa o peso do arco para esta
    }                              // posição da matriz.
  while(k++<num_arcs);
 }
 if(tipo==2 || tipo==3)  // Se a matirz for de saída (tipo 2) ou
 {                       // se a matriz for de incidência (tipo 3)
  k=0;                   // verifica quando um arco tem destino em um Lugar i
  do{                    // e origem em uma Transição j.
     if((i=index_p(arco[k].target,num_places))==-1)continue;
     if((j=index_t(arco[k].source,num_transitions))==-1)continue;
     matrix[i][j]+=arco[k].weight; // Incrementa o peso do arco para esta
    }                              // posiçao da matriz.
  while(k++<num_arcs);   // Repete o processo para todos os arcos.
 }
}

//------------------------------------------------------------------------------

//----------------------Indexa Lugares------------------------------------------

int index_p(char *id, int max)
{                   // Retorna o índice i de um lugar cujo nome
 int i=0;           // é recebido como parâmetro.
 while (i<max && strcmp (id,lugar[i].id) != 0)i++;
 if (i < max)return i;
 else return -1;    //Caso não encontre, retorna -1.
}

//------------------------------------------------------------------------------

//----------------------Indexa Transições---------------------------------------

int index_t(char *id, int max)
{                   // Retorna o índice i de uma transição cujo nome
 int i=0;           // é recebido como parâmetro.
 while (i<max && strcmp (id,trans[i].id) != 0)i++;
 if (i < max)return i;
 else return -1;    //Caso não encontre, retorna -1.
}

//------------------------------------------------------------------------------
```

```
//---------------------Mostra Matrizes da Rede de Petri-------------------------

void mostra_matriz(int matrix[][20], int num_places, int num_transitions,
int tipo)
{
 if(tipo == 1)printf("\t\tMatrizes da Rdp\n\n"); // Mostra título geral.
 printf("\nMatriz de ");
 if(tipo == 1)printf("entrada.\n"); // Mostra nome da Matriz
 if(tipo == 2)printf("saida.\n");   //de a cordo com o tipo.
 if(tipo == 3)printf("incidencia.\n");
 int i,j;
 printf("\n");
 printf("%4s",""); // Utiliza espaços fixos de 4 caracteres.
 for(j=0;j<num_transitions;j++)printf("%4s",trans[j].id); // Mostra Transições.
 for(i=0;i<num_places;i++)
 {
  printf("\n");
  printf("%4s",lugar[i].id);               // Mostra Lugares e posições
  for(j=0;j<num_transitions;j++)printf("%4d",matrix[i][j]);// da Matriz.
 }
 printf("\n");
 if(tipo != 3)return; // Se a Matriz não for a última, não imprime tracejado.
 printf("\n-------------------------------------------------------\n");
 getchar();
}

//------------------------------------------------------------------------------


//------------Ajusta prioridades para solução de conflitos----------------------

void ajusta_prioridade(int num_places, int num_transitions)
{
 int i,j,k,n;
 for(j=0;j<num_transitions;j++)
     for(i=0;i<num_places;i++)
         if(matriz_i[i][j]!=0) // Localiza os conflitos de entrada.
             for(k=0,n=0;k<num_transitions;k++)
                 if(matriz_i[i][k]!=0 && k!=j){
                     n++;
                     if(trans[k].priority == trans[j].priority)//Se prioridade
                     trans[k].priority = trans[j].priority + 1;//é igual,
                 }                                             // altera.
 for(j=0;j<num_transitions;j++)
     for(i=0;i<num_places;i++)
         if(matriz_o[i][j]!=0) // Localiza os conflitos de saída.
             for(k=0,n=0;k<num_transitions;k++)
                 if(matriz_o[i][k]!=0 && k!=j){
                     n++;
                     if(trans[k].priority == trans[j].priority)//Se prioridade
                     trans[k].priority = trans[j].priority + 1;//é igual,
                 }                                             // altera.
}

//------------------------------------------------------------------------------
```

```
//------------Solicita valores de Período e Frequência de Clock-----------------
int le_periodo()
{
 printf("Digite o periodo entre trasicoes em milisegundos: ");
 scanf("%d",&periodo); // Obtém valor do período entre transi□□es.
 printf("Periodo = %d ms\n\n",periodo);
 getchar();
 if(periodo==0)return periodo;
 printf("Digite a frequencia de clock em Hertz: ");
 scanf("%d",&fclock); // Obtém valor do Frequência de clock.
 printf("Frequencia = %d Hertz\n\n",fclock);
 getchar();
}
//-------------------------------------------------------------------------------
//----------Determina número de bits para conter um valor----------------------
int num_bits(int a)
{
 int n;
 if(a==0)a=255; // Se a é 0 faz a igual a 255 (capacidade não estipulada).
 n=(int)ceil(log10(a+1)/log10(2)); //Calcula número de bits.
 return n;
}
//-------------------------------------------------------------------------------
//----------Cria nome do arquivo a ser gerado----------------------------------
void le_file_name()
{
 char extension[]=".vhd";     // Cria extensão ".vhd" .
 int i=0;
 while((name[i]=nome[i])!='.')i++;
 name[i]='\0';
 strcpy(file_name,name);           //Copia nome do arquivo PNML.
 strcat(file_name,extension);      // Anexa extensão ".vhd" .
}
//-------------------------------------------------------------------------------
```

```
//------------------Gera o arquivo VHDL----------------------------------------

void escreve_file(int num_places, int num_transitions, int num_arcs)
{

//------------------Abre arquivo para escrita----------------------------------

 le_file_name(); // Lê o nome do arquivo a ser gravado.
 FILE *fp;
 if((fp=fopen(file_name,"w"))==NULL){ // Abre arquivo para escrita.
  printf("Read error ocurred");
  getchar();
  exit(1);
}

//-----------------Codifica Cabeçalho e Entidade-------------------------------

 int i,j,k,m,n;
 fprintf(fp,"library ieee;\n"); // Cria cabeçalho com bibliotecas "ieee".
 fprintf(fp,"use ieee.std_logic_1164.all;\n");
 fprintf(fp,"use ieee.std_logic_arith.all;\n");
 fprintf(fp,"use ieee.std_logic_unsigned.all;\n\n");
 fprintf(fp,"entity %s IS\n",name); // Codifica a "entidade".
 fprintf(fp,"\tPORT\n");
 fprintf(fp,"\t(\n");
 if(periodo == 0)fprintf(fp,"\tCLK : in std_logic;\n");// Se período é 0
 else{    // utiliza o clock da placa de teste.
      fprintf(fp,"\tCLK : buffer std_logic;\n"); // Caso contrário cria um
      fprintf(fp,"\tCLK1 : in std_logic;\n");    // sinal de clock auxiliar.
     }
 fprintf(fp,"\tRESET : in std_logic;\n");
 for(i = 0;i<num_places;i++){// Codifica terminais para os Lugares da Rede.
       n=num_bits(lugar[i].capacity);
       fprintf(fp,"\t%s : buffer std_logic_vector(%d downto 0)",
       lugar[i].id,n-1);
       if(i<num_places-1)fprintf(fp,";\n");
       else fprintf(fp,"\n");
       }
 fprintf(fp,"\t);\n");
 fprintf(fp,"end %s;\n\n",name);
```

```
//-----------------Codifica Sinais e Constantes---------------------------------

 fprintf(fp,"architecture behavioral OF %s IS\n\n",name);
 for(i = 0;i<num_places;i++){ // Codifica sinais para armazenamento temporário
     n=num_bits(lugar[i].capacity);// das marcas de acordo com a capacidade
     fprintf(fp,"\tsignal %sTemp : std_logic_vector(%d downto 0);\n",// de
     lugar[i].id,n-1);             // cada Lugar da Rede de Petri.
     }
 fprintf(fp,"\n");                 // Codifica sinais de
 for(i = 0;i<num_transitions;i++)  // disparo e habilitação das Transições.
   fprintf(fp,"\tsignal %s, %s_ENABLE : std_logic;\n",trans[i].id,trans[i].id);
 fprintf(fp,"\n");
 for(i = 0;i<num_arcs;i++)         // Codifica constantes de peso dos Arcos.
   fprintf(fp,"\tconstant W%s%s\t : integer := %d;\n",arco[i].source,
   arco[i].target,arco[i].weight);
 fprintf(fp,"\n");
 for(i = 0;i<num_transitions;i++)  // Codifica as constantes de prioridades.
   fprintf(fp,"\tconstant PRIOR_%s\t : integer := %d;\n",trans[i].id,
   trans[i].priority);
 fprintf(fp,"\n");
 for(i = 0;i<num_places;i++){      // Codifica as constantes de capacidades.
     if(lugar[i].capacity==0)k=255;
     else k=lugar[i].capacity;
     fprintf(fp,"\tconstant K_%s\t : integer := %d;\n",lugar[i].id,k);
     }
 fprintf(fp,"\n");       // Codifica a constante PERÍODO se diferente de zero:
 if(periodo != 0)fprintf(fp,"\tconstant PERIODO : INTEGER := %d;\n",periodo);
 fprintf(fp,"\n\n");

//-----------------Codifica Cálculo da Marcação---------------------------------

 fprintf(fp,"begin\n");
  for(i=0;i<num_places;i++){// Para todos os Lugares, calcula a nova marcação
     fprintf(fp,"\n");      // considerando o número de marcas atuais
     fprintf(fp,"\t%sTemp <= conv_std_logic_vector((conv_integer(%s)",
     lugar[i].id,lugar[i].id); //somados às marcas adicionadas pelo
     for(j=0;j<num_transitions;j++){ // disparo das transições habilitadas
          if(matriz_o[i][j]!=0){ // na matriz de saída
             fprintf(fp,"\n\t\t+ W%s%s*conv_integer(%s)",trans[j].id,
             lugar[i].id,trans[j].id);
             }
          }                          // e subtraindo as marcas retiradas pelo
     for(j=0;j<num_transitions;j++){// disparo das transições habilitadas
          if(matriz_i[i][j]!=0){    // na matriz de entrada.
             fprintf(fp,"\n\t\t- W%s%s*conv_integer(%s)",lugar[i].id,
             trans[j].id,trans[j].id);
             }
          }
     fprintf(fp,"),%d);",num_bits(lugar[i].capacity)); // Ajusta a quantidade
     fprintf(fp,"\n");                                 // de bits do valor calculado
     }                                  // de acordo com a capacidade do Lugar.
 fprintf(fp,"\n\n");
```

```
//-------------------Codifica Atribuição da Marcação----------------------------

 for(i = 0;i<num_places;i++){// Codifica o processo de atribuição de marcas que
  fprintf(fp,"process (CLK, RESET) begin\n\t");// responde ao Clock e ao RESET.
  fprintf(fp,"if (RESET = '1') then %s <= conv_std_logic_vector(%d,%d);\n\t",
  lugar[i].id,lugar[i].marking,num_bits(lugar[i].capacity));// No RESET, cada
  fprintf(fp,"elsif (CLK'event and CLK='1') then\n\t\t"); // Lugar recebe a
  fprintf(fp,"%s <= %sTemp;\n\t",lugar[i].id,lugar[i].id);// marcação inicial.
  fprintf(fp,"end if;\n"); // Na borda de subida do sinal de clock cada Lugar
  fprintf(fp,"end process;\n\n"); // recebe a marcação atualizada contida nos
 } // sinais Temp.
 fprintf(fp,"\n");

//-------------------Codifica Habilitação das Transições-----------------------

 for(j=0;j<num_transitions;j++){ // Para cada transição verifica e atribui
     fprintf(fp,"\t%s_ENABLE <= '1' when\n",trans[j].id); // 1 ao sinal de
     fprintf(fp,"\t\t("); // habilitção...
     for(i=0,m=0;i<num_places;i++){
          if(matriz_i[i][j]!=0){// se para cada Lugar de entrada desta
             m++;// Transição o numero de marcas é maior ou igual ao peso do
             if(m>1)fprintf(fp," and "); //respectivo Arco de ligação e
             fprintf(fp,"%s >= W%s%s",lugar[i].id,lugar[i].id,trans[j].id);
             }
          }
     for(i=0,n=0;i<num_places;i++){
          if(matriz_o[i][j]!=0){ // se cada Lugar de saída desta Transição
             n++; // possui capacidade disponível maior ou igual
             if(n==1&&m>0)fprintf(fp," and\n\t\t");// ao peso do respectivo
             if(n>1)fprintf(fp," and "); // Arco de ligação.
             fprintf(fp,"K_%s-%s >= W%s%s",lugar[i].id,lugar[i].id,
             trans[j].id,lugar[i].id);
             }
          }
     fprintf(fp,")\n");
     fprintf(fp,"\t\telse '0';\n"); // Caso contrário o sinal de habilitação
     fprintf(fp,"\n"); // recebe o valor 0.
     }
```

```
//----------Codifica Disparo e Solução de Conflitos das Transições-------------
 fprintf(fp,"-------------------------------------\n\n");
 for(j=0;j<num_transitions;j++){// O sinal de disparo de cada Transição recebe
     fprintf(fp,"\t%s <= '1' when (%s_ENABLE = '1'",trans[j].id,trans[j].id);
                               // o valor 1 se ela estiver habilitada e
     for(i=0;i<num_places;i++){
          if(matriz_i[i][j]!=0){   // em caso de conflito de entrada, o número
             for(k=0,n=0;k<num_transitions;k++){ // de marcas no lugar de
               if(matriz_i[i][k]!=0 && k!=j){    // entrada é suficiente para
                  n++;              // disparo de todas as Transições envolvidas
                    if(n==1)fprintf(fp," and\n\t\t((conv_integer(%s) >=
                    W%s%s + ",lugar[i].id,lugar[i].id,trans[j].id);
                    if(n>1)fprintf(fp," + ");
                    fprintf(fp,"conv_integer(%s_ENABLE)*W%s%s",
                    trans[k].id,lugar[i].id,trans[k].id);
                    }
             }
             if(n>0)fprintf(fp,")"); // ou a prioridade da Transição é maior
             for(k=0,n=0;k<num_transitions;k++){// que as outras transições
               if(matriz_i[i][k]!=0 && k!=j){ // habilitadas em conflito.
                  n++;
                    if(n==1)fprintf(fp," or\n\t\t(");
                    if(n>1)fprintf(fp," and ");
                    fprintf(fp,"PRIOR_%s > conv_integer(%s_ENABLE)*PRIOR_%s",
                    trans[j].id,trans[k].id,trans[k].id);
                    }
             }
             if(n>0)fprintf(fp,"))");
          }
     }
     for(i=0;i<num_places;i++){
          if(matriz_o[i][j]!=0){        // e em caso de conflito de saída,
             for(k=0,n=0;k<num_transitions;k++){// a capacidade disponível
               if(matriz_o[i][k]!=0 && k!=j){   // nos lugares de saída é
                  n++;         // suficiênte para receber as marcas de todas as
                  if(n==1)fprintf(fp," and\n\t\t((K_%s-%s >= W%s%s + ",
                  lugar[i].id,lugar[i].id,trans[j].id,lugar[i].id);
                  if(n>1)fprintf(fp," + ");    //trancições em conflito.
                  fprintf(fp,"conv_integer(%s_ENABLE)*W%s%s",trans[k].id,
                  trans[k].id,lugar[i].id);
                  }
             }
             if(n>0)fprintf(fp,")"); // ou a prioridade da Transição é maior
             for(k=0,n=0;k<num_transitions;k++){// que as outras transições
               if(matriz_o[i][k]!=0 && k!=j){   // habilitadas em conflito.
                  n++;
                  if(n==1)fprintf(fp," or\n\t\t(");
                  if(n>1)fprintf(fp," and ");
                  fprintf(fp,"PRIOR_%s > conv_integer(%s_ENABLE)*PRIOR_%s",
                  trans[j].id,trans[k].id,trans[k].id);
                  }
             }
             if(n>0)fprintf(fp,"))");
          }
     }
 fprintf(fp,")\n\t\telse '0';\n\n");
 }
```

```
//-------------------Codifica Divisor de Clock---------------------------------

 if(periodo != 0){// Se PERÍODO é diferente de zero codifica o divisor de clock.
  fprintf(fp,"\nprocess (CLK1)\n"); // Codifica contador para temporizar 1ms:
  fprintf(fp,"\tVARIABLE CONTADOR : INTEGER RANGE 0 TO %d;\n",fclock/1000);
  fprintf(fp,"\tVARIABLE CONTA_MS : INTEGER RANGE 0 TO PERIODO;\n"); // Codifica
  fprintf(fp,"\nbegin\n"); //outro contador para contar o número de ms desejado.
  fprintf(fp,"\tif (PERIODO = 0) then\n"); // Se o PERÍODO é zero utiliza o
  fprintf(fp,"\t\tCLK <= CLK1;\n\t");       // sinal de clock da placa de teste.
  fprintf(fp,"elsif(CLK1'event and CLK1='1') then CONTADOR := CONTADOR + 1;\n");
  fprintf(fp,"\t\tif(CONTADOR = %d/2) then CONTA_MS := CONTA_MS + 1;\n",
  fclock/1000);    // Caso contrário o sinal de clock da placa é utilizado para
  fprintf(fp,"\t\t\tCONTADOR :=0;\n"); // acionar os contadores, produzindo um
  fprintf(fp,"\t\t\tif(CONTA_MS = PERIODO) then\n"); // um novo sinal de clock
  fprintf(fp,"\t\t\t\tCONTA_MS :=0;\n");    // com o período desejado.
  fprintf(fp,"\t\t\t\tCLK <= NOT CLK;\n"); // Chavea sinal de clock dividido.
  fprintf(fp,"\t\t\tend if;\n");
  fprintf(fp,"\t\tend if;\n");
  fprintf(fp,"\tend if;\n");
  fprintf(fp,"end process;\n\n");
 }

//-------------------Codifica Final da Arquitetura-----------------------------

 fprintf(fp,"\n\n");
 fprintf(fp,"end behavioral;\n");

//-------------------Fecha arquivo---------------------------------------------

 fclose(fp);
 printf("O arquivo %s foi gerado com sucesso\n\n",file_name);
 getchar();
}

//---------------------------------Fim---------------------------------------
```

REFERÊNCIAS

ANDRADE, M. C. **RTEV - ambiente de desenvolvimento de aplicações reconfiguráveis com o kernel de tempo real Virtuoso**. Dissertação (Mestrado) — Universidade Federal de São Carlos, 2006. Disponível em: <http://www.bdtd.ufscar.br/htdocs/tedeSimplificado//tde$_b$*usca*/*arquivo*.*php*?*codArquivo* = 2246¿.

BERKELEY DESIGN TECHNOLOGY, INC. **The AutoESL AutoPilot High-Level Synthesis Tool: An Independent Evaluation of**. [S.l.], 2009.

BILLINGTON, J. et al. **The Petri Net Markup Language: Concepts, Technology, and Tools**. 2003.

BRÆK, R. Sdl basics. **Computer Networks and ISDN Systems**, v. 28, n. 12, p. 1585–1602, 1996. Disponível em: <http://www.sciencedirect.com/science/article/pii/0169755295001190>.

BUNDELL, G. A. An fpga implementation of the petri net firing algorithm. In: **The 1997 Australasian Conference on Parallel and Real-Time Systems, Springer-Verlag, Singapore, Newcastle, pp. 434-445.** [S.l.: s.n.], 1997.

BUNKER, A.; GOPALAKRISHNAN, G.; MCKEE, S. A. Formal hardware specification languages for protocol compliance verification. **ACM Trans. Des. Autom. Electron. Syst.**, ACM, New York, NY, USA, v. 9, p. 1–32, January 2004. ISSN 1084-4309. Disponível em: <http://doi.acm.org/10.1145/966137.966138>.

CONG, J. et al. High-level synthesis for fpgas: From prototyping to deployment. **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on**, v. 30, n. 4, p. 473 –491, april 2011. ISSN 0278-0070.

DéHARBE, D.; MEDEIROS, S. Aspect-oriented design in systemc: implementation and applications. In: **Proceedings of the 19th annual symposium on Integrated circuits and systems design**. New York, NY, USA: ACM, 2006. (SBCCI '06), p. 119–124. ISBN 1-59593-479-0. Disponível em: <http://doi.acm.org/10.1145/1150343.1150378>.

DIAS, G. L. **Ferramentas para a Integração de Redes de Petri e VHDL na Síntese de Sistemas Digitais**. Dissertação (Mestrado) — UNESP, 2007.

GILLENWATER, J. et al. Synthesizable high level hardware descriptions: using statically typed two-level languages to guarantee verilog synthesizability. In: **Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation**. New York, NY, USA: ACM, 2008. (PEPM '08), p. 41–50. ISBN 978-1-59593-977-7. Disponível em: <http://doi.acm.org/10.1145/1328408.1328416>.

GOKHALE, M.; GRAHAM, P. S. **Reconfigurable Computing - Accelerating Computation with Field-Programmable Gate Arrays**. [S.l.]: Springer US, 2005.

HAREL, D. **Statecharts: A Visual Formalism For Complex Systems**. 1987.

HAREL, D. et al. Statemate: A working environment for the development of complex reactive systems. **IEEE Transactions on Software Engineering**, v. 16, p. 5, 1990.

HAREL, D.; MARELLY, R. **Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine.** [S.l.]: Springer-Verlag, 2003.

HAREL, D.; THIAGARAJAN, P. S. Message sequence charts. In: **In UML for Real: Design of Embedded Real-Time Systems**. [S.l.]: Kluwer Academic Publishers, 2003. p. 77–105.

IEEE. Standard for verilog hardware description language. **IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)**, p. C1–560, 2006.

IEEE. Standard for system verilog-unified hardware design, specification, and verification language. **IEEE STD 1800-2009**, p. C1 –1285, 2009.

IEEE. Standard vhdl language reference manual. **IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)**, p. c1 –626, 26 2009.

ITU-T. Recommendation Z.100. Specification and Description Language. [S.l.]: ITU, 1993.

ITU-T. Recommendation Z.120: Message Sequence Chart (MSC). [S.l.]: ITU, 1999.

KINDLER, E. The epnk: an extensible petri net tool for pnml. In: **Proceedings of the 32nd international conference on Applications and theory of Petri Nets**. Berlin, Heidelberg: Springer-Verlag, 2011. (PETRI NETS'11), p. 318–327. ISBN 978-3-642-21833-0. Disponível em: <http://dl.acm.org/citation.cfm?id=2022192.2022213>.

KUBáTOVá, H. Modeling by petri nets. **Journal of Advanced Engineering - Acta Polytechnica 2005/2 - Advanced Logic Design**, Czech Technical University in Prague, v. 45, p. 6 – 13, 02 2005. Disponível em: <http://ctn.cvut.cz/ap/download.php?id=41>.

LIMA, P. **Geração de código VHDL a partir de especificações IOPT PNML2VHDL**. Dissertação (Mestrado) — FCT-UNL, May 2009. Disponível em: <http://oa.uninova.pt/1955/>.

LIN, M. The amorphous fpga architecture. In: **Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays**. New York, NY, USA: ACM, 2008. (FPGA '08), p. 191–200. ISBN 978-1-59593-934-0. Disponível em: <http://doi.acm.org/10.1145/1344671.1344700>.

LIN, M. et al. Performance benefits of monolithically stacked 3d-fpga. In: **Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays**. New York, NY, USA: ACM, 2006. (FPGA '06), p. 113–122. ISBN 1-59593-292-5. Disponível em: <http://doi.acm.org/10.1145/1117201.1117219>.

LIU, S.; ZENG, R.; HE, X. Pipe+ - a modeling tool for high level petri nets. In: **SEKE**. Knowledge Systems Institute Graduate School, 2011. p. 115–121. ISBN 1-891706-29-2. Disponível em: <http://dblp.uni-trier.de/db/conf/seke/seke2011.htmlLiuZH11>.

LLOYD, L. et al. Asynchronous microprocessors: From high level model to fpga implementation. **Journal of Systems Architecture**, v. 45, n. 12-13, p. 975 – 1000, 1999. ISSN 1383-7621.

MACIEL, P. R. M.; LINS, R. D.; CUNHA, P. R. F. **Introdução às Redes de Petri e Aplicações**. [S.l.]: 10ª Escola de Computação Campinas, Julho, 1996.

MäKINEN, E.; SYSTä, T. An interactive approach for synthesizing uml statechart diagrams from sequence diagrams. In: **Proceedings of OOPSLA 2000 Workshop: Scenario based round-trip engineering**. [S.l.: s.n.], 2000. p. 7–12.

MURA, M. et al. Statecharts to systemc: a high level hardware simulation approach. In: **Proceedings of the 17th ACM Great Lakes symposium on VLSI**. New York, NY, USA: ACM, 2007. (GLSVLSI '07), p. 505–508. ISBN 978-1-59593-605-9. Disponível em: <http://doi.acm.org/10.1145/1228784.1228904>.

MURATA, T. Petri nets: Properties, analysis and applications. **Proceedings of the IEEE**, v. 77, n. 4, p. 541 –580, abr. 1989. ISSN 0018-9219.

PADERBORN, W. M.; MUELLER, W. **The Formal Execution Semantics of SpecC**. 2002.

PETRI, C. A. **Kommunikation mit Automaten.** [S.l.]: Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

RADETZKI, M. **Synthesis of Digital Circuits from Object Oriented Specifications**. Tese (Doutorado) — Oldenburg University, 2000.

SILVA, C. et al. Methodology to implement logic controllers with both reconfigurable and programmable hardware. In: **Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on**. [S.l.: s.n.], 2007. p. 324 –328.

SOARES, S. N. **T&D Bench - Explorando o Espaço de Projetos de Processadores em Ensino e Pesquisa**. Tese (Doutorado) — Univesidade Federal do Rio Grande do Sul, 2005.

SOTO, E.; PEREIRA, M. Implementing a petri net specification in a fpga using vhdl. In: **The International Workshop on Discrete-Event System Design**. [S.l.: s.n.], 2001.

SUDACEVSCHI, V.; ABABII, V.; NEGURA, V. A hardware implementation of safe petri net models. In: **Proceedings of the 8th International Conference on DEVELOPMENT AND APPLICATION SZSTEMS "DAS-2006", Suceava, Romania**. [S.l.: s.n.], 2006. p. 25–27.

SUDACEVSCHI, V.; GUTULEAC, L.; ABABII, V. A hardware implementation of petri nets models. In: **Proceedings of the 7th International Conference on DEVELOPMENT AND APPLICATION SZSTEMS "DAS-2004", 27-29 may, 2004, Suceava, Romania**. [S.l.: s.n.], 2004. p. pp 24–28.

SUTHERLAND, S. Verilog, the next generation: Accellera's systemverilog. In: ____. **Proceedings of the HDL Conference**. [s.n.], 2002. Disponível em: <http://www/1sutherland.com/papers/2002-HDLCon-paper-SystemVerilog.pdf>.

WEBER, M. et al. The petri net markup language. In: **Petri Net Newsletter**. [S.l.]: Springer, 2000. p. 124–144.

WEGRZYN, M. Implementation of safety critical logic controller by means of fpga. **Annual Reviews in Control**, v. 27, n. 1, p. 55 – 61, 2003. ISSN 1367-5788.

ZEIDMAN, B. **Designing with FPGAs and CPLDs**. [S.l.]: C M P Books, 2002. ISBN 1578201128.