Universidade Estadual Paulista Júlio de Mesquita Filho

Faculdade de Engenharia de Ilha Solteira

Departamento de Engenharia Elétrica

Comparação de Técnicas de Redes de Petri na Descrição de Um *Pipeline*

candidato: Cristiano Pires Martins

orientador: Prof. Dr. Aleardo Manacero Jr.

Dissertação submetida à Faculdade de Engenharia de Ilha Solteira da Universidade Estadual Paulista Júlio de Mesquita Filho, para preenchimento dos pré-requisitos parciais para obtenção do Título de Mestre em Engenharia Elétrica.

Março de 2004

Resumo

O projeto de sistemas digitais é, na maioria das vezes, uma tarefa complexa e que consome muito tempo do projetista. Não se concebe atualmente que esses projetos sejam inteiramente realizados sem que os sistemas projetados sejam modelados e simulados, procurando diminuir o custo de projeto.

Entre as técnicas em uso para a geração de modelos estão as baseadas em redes de Petri. O problema com redes de Petri é que existe uma grande variedade de diferentes técnicas para modelar sistemas equivalentes e, além disso, o tratamento de tempo é feito de formas muito distintas.

Nesse trabalho se apresentam algumas técnicas de redes de Petri que permitem o tratamento de tempo e se faz a comparação entre as mesmas quando aplicadas na modelagem de um *pipeline*. Dentre as técnicas examinadas está incluída uma rede híbrida, na qual se propõe a mistura não tradicional de outras técnicas mais básicas, buscando melhores resultados do que os obtidos com tais redes. Apresentam-se resultados obtidos com um protótipo de um simulador especialmente projetado para ser capaz de simular as redes de Petri híbridas propostas nesse trabalho, assim como cada uma das técnicas individuais nelas utilizadas.

Abstract

The design of a digital system often is a complex and time consuming task. Nowadays it is unconceivable that these designs would be entirely done without modeling and simulation, in order to save design costs.

Among the modeling techniques in use there are the Petri nets models. The drawback with Petri nets is that there is a wide range of different techniques to perform the modeling of equivalent systems and, moreover, the time manipulation is performed through very distinct approaches.

Some Petri net techniques that deal with time are presented in this work. Comparisons between these techniques are done through the modeling of a pipeline. Among the techniques examined there is a hybrid one, that is propposed as a composition of the basic techniques, aiming the improvement of the modeling results. Results provided by a prototype of a Petri Net simulator built to simulate the hybrid model, and its component techniques, are presented.

Agradecimentos

À Deus, por ter me trazido até aqui e dado forças nas horas em que mais precisava.

À minha mãe, meu pai e meu irmão, com quem posso partilhar toda minha vida, aflições e anseios, pelo valiosíssimo apoio que têm me dado.

Ao Prof. Aleardo Manacero Jr., pela orientação, pela paciência e principalmente pelo apoio nos momentos em que os problemas pareciam intransponíveis.

À Vanessa e sua família, por ter estado presente nos momentos alegres e difíceis desse trabalho, pelas horas perdidas de sono ao meu lado e pelas palavras de apoio e carinho que me destes quando precisei.

Ao Prof. Mauro Conti Pereira, um dos grandes responsáveis por eu estar neste programa de mestrado.

Ao meu amigo Cecilio Merlotti Rodas, pela amizade cultivada nestes anos e pelo apoio nas horas difíceis, de grande importância no meu processo de adaptação na cidade de Ilha Solteira.

Às pessoas que, mesmo não estando próximas, puderam contribuir.

Às pessoas que contribuiram direta ou indiretamente para realização deste trabalho.

Dedico esse trabalho aos meus familiares.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

Rede de Petri (RdP) é uma ferramenta gráfica e matemática que se adapta bem a um grande número de aplicações em que as noções de eventos e de evoluções simultâneas são importantes ([Cardoso e Valette, 1997] e [ICATPN, 2000]). Essas aplicações incluem protocolos de comunicação, redes de computadores, sistemas industriais, controle de processos industriais e fluxo de dados computacionais. Similarmente, RdP's têm sido usadas em estudos de propriedades de comportamento de sistemas através de simulações, avaliação de desempenho e verificação de tolerância à falhas. Outra aplicação importante de RdP's é a modelagem de uma arquitetura computacional complexa, incluindo características comportamentais como concorrência e pontos de decisão.

Como é quase impossível falar em modelagem de arquiteturas sem tocar no parâmetro tempo, é importante compreender como RdP's tratam o tempo para modelar um sistema. Existem maneiras distintas de se descrever uma seqüência de eventos temporais dentro de uma RdP, sendo importante determinar o que essas diferenças implicam no momento de modelar um sistema digital.

Para examinar a aplicação de redes de Petri em sistemas digitais escolheu-se um elemento de arquitetura conhecido como *pipeline*, que é uma técnica muito

importante para aumentar o desempenho dos processadores. O conceito fundamental envolvido em *pipeline* é tornar o processador capaz de executar diversas instruções simultaneamente, procurando fornecer ao menos o resultado de uma instrução a cada ciclo de processamento. Sem o *pipeline* cada instrução é processada individualmente e seu resultado pode consumir um número significativo de ciclos, fazendo com que as demais instruções fiquem em fila de espera, prejudicando o desempenho do processador.

1.2 Objetivo

A rede de Petri é uma ferramenta gráfica que promove uma abordagem matemática para a modelagem e análise de sistemas, com conveniente efetividade e alto poder intuitivo com linguagem gráfica para suas visualizações. Ela se adapta bem a um grande número de aplicações principalmente quando os sistemas que estão sendo modelados envolvem tempo e indeterminismos.

A modelagem de sistemas digitais complexos envolve uma grande quantidade de informações, como por exemplo ocorre com a modelagem de um *pipeline*. Sendo o *pipeline* uma técnica importante para o aumento do desempenho de processadores, existe um interesse muito grande em simulá-lo e com isso descobrir formas de se aperfeiçoar cada vez mais essa técnica. Para simular um *pipeline* é preciso trabalhar com vários parâmetros como tempo, probabilidade e concorrência, entre outros. Portanto, a técnica de rede de Petri se encaixa perfeitamente na tarefa de modelar esta tecnologia.

O objetivo deste trabalho é fazer uma comparação entre algumas técnicas de rede de Petri que tratam de tempo [Merlin e Farber, 1976] [Wang, 1998] e probabilidade [Zimmermann, Freiheit e Hommel], [Ajmone, Balbo, Chiola e Conte, 1987] e [Molloy, 1982] e mostrar a dificuldade em se conseguir modelar um sistema digital (neste caso, um *pipeline*) que envolva esses dois parâmetros utilizando apenas uma das técnicas por vez. Por fim, apresenta-se uma RdP que combine várias dessas técnicas, formando uma RdP Híbrida.

Foram desenvolvidas três redes de Petri para modelar um *pipeline*. A pri-

meira rede trata apenas do fator tempo, a segunda trata de probabilidade e a terceira utiliza uma técnica mais abrangente de RdP, que envolve tempo e probabilidade no mesmo modelo. Este modelo é também uma contribuição desse trabalho por associar parâmetros que nem sempre são tratados de forma conjunta em redes de Petri.

1.3 Descrição do Texto

Este trabalho está organizado em quatro capítulos. Na primeira parte descreve-se os conceitos fundamentais necessários para entendimento do trabalho. Começa-se, no capítulo 2, com uma descrição dos conceitos fundamentais de redes de Petri, incluindo algumas das extensões que permitem o tratamento de tempo de alguma forma. Passa-se então para uma breve introdução aos sistemas digitais e, em particular, sobre características essenciais dos *pipelines*.

A segunda parte, apresentada pelo capítulo 3, trata da modelagem de um *pipeline* usando redes de Petri. Inicia-se com uma descrição dos módulos básicos dos modelos que serão criados e por fim a apresentação dos modelos de redes de Petri usando técnicas e recursos diferentes: rede de Petri Baseada em Tempo, rede de Petri Estocástica Generalizada e rede de Petri Híbrida.

A terceira parte do trabalho trata das comparações entre os modelos de *pipeline* criados com as diferentes extensões de redes de Petri. Tais comparações são apresentadas ao longo do capítulo 4, incluindo análise qualitativas, sobre a complexidade da construção de cada modelo, e quantitativas, sobre a estabilidade e precisão dos modelos quando simulados.

Finalmente, no capítulo 5 apresenta-se a quarta e última parte do trabalho, quando se mostram as conclusões tiradas sobre a aplicação das diferentes técnicas na modelagem de um *pipeline* e indicam-se possíveis caminhos para a continuidade desse trabalho.

Capítulo 2

Conceitos Fundamentais

O mundo digital implica que a informação no computador é representada por variáveis que assumem um número limitado de valores discretos. Estes valores são processados internamente pelos componentes que podem manter um número limitado de estados discretos. O primeiro computador digital eletrônico desenvolvido no final dos anos 40 foi usado primeiramente para cálculos numéricos. Neste caso, os elementos discretos são os dígitos. A partir desta aplicação surgiu o termo computador digital. Na prática, a função computador digital é mais real se somente dois estados forem usados. Por causa das restrições físicas dos componentes e em função da tendência da lógica humana ser binária (exemplo: verdadeiro ou falso, sim ou não), componentes digitais que assumem valores discretos são obrigados a assumir somente dois valores que são ditos binários [Mano, 1993].

Sistemas digitais são usados em comunicações, transações de negócio, controle de tráfego, tratamento médico, monitoramento de tempo, a Internet e muitos outros empreendimentos comerciais, industriais ou científicos. A propriedade mais notável de um sistema digital é sua generalidade. Ele pode seguir uma seqüência de instruções, chamada programa, que opera sobre uma informação dada [Mano, 2002].

Uma característica de sistemas digitais é a habilidade de manipular elementos discretos de informação. Qualquer conjunto que é restrito a um número finito de elementos contém informações discretas. Elementos discretos de informação são representados em um sistema digital por quantidades físicas chamadas de sinais. Si-

nais elétricos tais como voltagem e corrente são os mais comuns. Um sistema digital é um sistema que recebe elementos discretos de informação (sinais, símbolos ou outros caracteres) e manipula-os através de uma representação interna em forma binária [Hill e Peterson, 1987].

Projetos de sistemas digitais são, em sua maioria, complexos por envolver uma grande variedade de parâmetros, dentre eles tempo e indeterminismos. A rede de Petri é justamente utilizada para modelar esse tipo de sistemas por conter ferramentas que lhe permite essa modelagem.

2.1 Introdução às Redes de Petri

2.1.1 Conceitos Básicos

As RdP's permitem uma abordagem matemática eficiente e intuitiva para a modelagem e análise de sistemas. A eficiência vem do tratamento algébrico do modelo e a intuitividade vem de sua representação gráfica [Buy e Sloan, 1994]. Essas redes são aplicáveis a uma ampla classe de sistemas, envolvendo características como:

- Concorrência e sincronismo;
- Sistemas determinísticos e estocásticos.

RdP's cobrem uma extensão de diversas aplicações, incluindo protocolos de comunicação, redes de computadores, sistemas industriais, controle de processos industriais e fluxo de dados computacionais. Em tais aplicações elas têm sido usadas tanto em estudos de propriedades de comportamento quanto em áreas como simulação e avaliação de desempenho [Nissanke, 1997].

Histórico

A rede de Petri é uma ferramenta gráfica e matemática que se adapta bem a um grande número de aplicações em que as noções de eventos e de evoluções simultâneas são importantes.

Esta teoria é muito recente, pois originou-se da tese intitulada "Comunicação com Autômatos", defendida por Carl Adam Petri em 1962 na Universidade de Darmstadt, Alemanha [Petri, 1962]. Anatol W. Holt foi seduzido por este trabalho e sob sua impulsão um grupo de pesquisadores do Massachussetts Institute of Technology - MIT, Estados Unidos, lança as bases, entre 1968 e 1976, do que se tornaram as redes de Petri. Entre estes pioneiros destacam-se F. Commoner e M. Hack [Cardoso e Valette, 1997].

As vantagens da utilização da rede de Petri podem ser resumidas pelas seguintes considerações:

- Pode-se descrever uma ordem parcial entre vários eventos, o que possibilita levar-se em conta a flexibilidade do sistema que está sendo modelado;
- Os estados, bem como os eventos, são representados explicitamente;
- Uma descrição precisa e formal das sincronizações torna-se possível, o que é essencial para alcançar a segurança necessária de funcionamento;.
- Uma única família de ferramentas é utilizada através da especificação, da modelagem, da análise, da avaliação do desempenho e da implementação;
- Uma única família de ferramentas é utilizada nos diversos níveis da estrutura hierárquica do controle, o que facilita a integração destes níveis.

Elementos Básicos

São quatro os elementos básicos para se definir uma rede de Petri. São polivalentes e em sua maioria, podem ser interpretados livremente. Estes elementos são os seguintes:

- Lugar (representado por um círculo, Figura 2.1): pode ser interpretado como uma condição, um estado parcial, uma espera, um procedimento, um conjunto de recursos, um estoque, uma posição geográfica num sistema de transporte, etc. Em geral todo lugar tem um predicado associado, por exemplo, máquina livre, peça em espera;

- Transição (representada por uma barra ou retângulo, Figura 2.1): é associada a um evento que ocorre no sistema, como o evento iniciar a operação;

- Marca (representado por um ponto num lugar, Figura 2.1): é um indicador significando que a condição associada ao lugar é satisfeita. Pode representar um objeto (recurso ou peça) numa certa posição geográfica (num determinado estado), ou ainda uma estrutura de dados que se manipula.

- Arcos (representados por flechas, Figura 2.1): Há dois tipos de arcos em rede de Petri, nomeados arcos de entrada e arcos de saída. Os de entrada mapeiam quais eventos são necessários para o disparo da transição, e são representados por flechas saindo dos lugares e chegando nas transições. Os arcos de saída mapeiam os eventos que seriam gerados com o disparo da transição, sendo representados por flechas saindo das transições. Um tipo especial de arco de entrada é o arco inibidor (representado por um círculo no lugar da ponta da flecha) que mapeia a situação de eventos que não podem ocorrer para o disparo de uma transição. Algebricamente os arcos têm pesos, que identificam a multiplicidade de marcas (eventos) necessárias nos lugares de entrada ou geradas nos lugares de saída. Esses pesos são determinados pela função W(p,t).

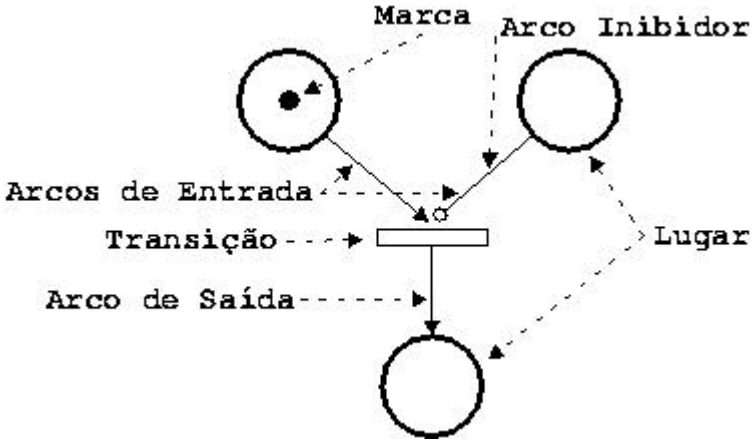

Figura 2.1: Rede de Petri.

Condições Gerais de Funcionamento de uma RdP

A evolução das RdP's é obtida por uma ordem de estados da rede representada pela função de marcação. A função de marcação M é um mapeamento

$$M : P \rightarrow N$$

que associa um número de marcas N com cada lugar P. Cada mudança dos estados da rede é trazida pelo disparo ou execução de uma transição. Diz-se que uma transição t está habilitada se, e somente se, as seguintes condições forem verdades:

- Todo o lugar de entrada p, distinto dos que tem arcos inibidores incidentes em t, tem pelo menos $W(p,t)$ marcas, que é o peso do arco de p para t.
- Todo lugar de entrada p, de onde há um arco inibidor em t, não pode ter marcas.

O disparo das transições altera o valor da função de marcação e assim, o estado da rede. Esses disparos obedecem as seguintes regras:

- Somente transições habilitadas podem disparar.
- Quando uma transição t dispara, W(pi,t) marcas são removidas de cada entrada pi da transição t (indicando que esta condição não é mais verdadeira após a ocorrência deste evento), e W(po,t) marcas são depositadas em cada saída po da transição t. O valor da função de marcação é inalterada em outro lugar.
- O disparo é instantâneo e completo com respeito à regra anterior. O disparo é também não determinístico no caso de escolha entre mais de uma transição habilitada concorrentemente. Isto é, qualquer uma das transições habilitadas pode disparar e a transição que será disparada é escolhida arbitrariamente.

2.2 Modelos de RdP

O modelo de RdP apresentado na seção anterior não abrange, explicitamente, tratamentos de tempo, probabilidade e cor. Embora uma quantidade significativa de aplicação não necessite desses parâmetros, existem aplicações em que a

obtenção de informações relevantes sobre o sistema dependem dessas condições. Nessa seção apresenta-se algumas formas de inserir os parâmetros tempo, probabilidade e cor em RdP's.

2.2.1 Redes de Petri Baseadas em Tempo (RdPBT)

Conceitos Básicos

Nesta abordagem cada marca é associada ao instante de disparo da transição que a criou. Associado ao registro de tempo existe uma função de tempo *time*, tal que dado *time(p)* dá o valor do registro do tempo da marca para o lugar p, assumindo que há somente um registro de tempo das marcas para p [Wang e Deng, 2000]. Tempo é uma função parcial

$$time : P \rightarrow T$$

e é definido somente para lugares onde há marcas.

Os intervalos de disparo permitido das transições são determinados em termos de uma condição de tempo, que é um conjunto de valores de tempo de relógio ou valores de tempo absoluto [Tew, Manivannan, Sadowski e Seila, 1994]. A condição de tempo de qualquer transição é determinada em termos dos registros de tempo do lugar de entrada e certos parâmetros de entrada associados com a transição [Ghezzi, Mandrioli, Morasca e Pezze, 1991].

Habilitação (Sensibilização) de Transição

Uma transição está habilitada se e somente se acontecer o seguinte:

- Cada lugar de entrada contém o número necessário de marcas. Se arcos inibidores estão sendo usados, então o lugar de entrada conectado com eles deve estar vazio. Esta condição é idêntica às redes de Petri sem tempo;
- Cada transição, quando necessário, deve ter indicado o tempo mínimo de espera antes de se tornar habilitada com base no tempo de criação de cada marca

no(s) lugar(es) de entrada, assim como, um tempo máximo de duração da habilitação. Todas as marcas devem chegar ao lugar de entrada antes que expire o intervalo da habilitação da transição.

Modelo de Execução

A execução das transições obedece às seguintes regras:

- Somente transições habilitadas podem disparar. O disparo de uma dada transição deve ocorrer em algum instante durante seu intervalo de habilitação;
- Quando uma transição t dispara, as marcas necessárias são removidas dos lugares de entrada de t. Esta regra também existe para redes de Petri não temporizadas, porém a diferença é que aqui, a escolha das marcas a serem removidas, em geral, depende dos registros de tempo;
- O disparo da transição entrega à saída um número apropriado de marcas, cada uma associada com o tempo do disparo e de acordo com o peso do arco;
- As demais regras das redes de Petri não temporizadas permanecem aplicáveis.

2.2.2 Redes de Petri Estocásticas Generalizadas (RdPEG)

As redes de Petri Estocástica Generalizada são redes de Petri temporizadas nas quais as transições são classificadas em dois grupos, de acordo com a sua prioridade [Granda, Drake e Gregorio, 1992]:

- Transições temporizadas têm seus atrasos distribuídos exponencialmente em variáveis aleatórias;
- Transições imediatas, que tem prioridade mais alta que as transições temporizadas, disparam em tempo igual a zero e são habilitadas com uma marcação dependente da probabilidade de disparo [Ghezzi, Mandrioli, Morasca e Pezze, 1991].

Uma marcação na qual transições imediatas são habilitadas é conhecida como marcação *vanishing*, enquanto que marcações com somente transições temporizadas são habilitadas são conhecidas como marcações tangíveis. Os estados *vanishing* têm permanência igual a zero, enquanto que estados tangíveis têm permanência diferente de zero distribuída exponencialmente.

Uma RdPEG é definida formalmente como uma sêxtupla (P,T,I,O,M0,W) em que P, T, I, O e M0 são definidas como numa RdP convencional e W é dado por

$$W : T \times M \rightarrow R$$

que é um conjunto de funções dependentes de marcações mapeando transições sobre números reais não negativos. Para transições temporizadas, esta função retorna a taxa de disparo médio da transição, enquanto que para transições imediatas retorna a probabilidade de disparo.

RdPEG são isomorfas às cadeias de Markov (CM), sendo que seus estados correspondem aos estados desta CM equivalente. Para a construção dessa cadeia deve-se primeiro eliminar os estados *vanishing*, obtendo uma CM de tempo contínuo contendo somente estados tangíveis.

Segundo Granda, Drake e Gregorio [Granda, Drake e Gregorio, 1992], a principal dificuldade envolvendo o uso de RdP é que o problema de análise se torna incontrolável quando o sistema que está sendo modelado é grande ou complexo, devido ao tamanho do processo de Markov que é gerado.

2.2.3 Rede de Petri Colorida

O modelo descrito a seguir propõe uma estruturação da parte de dados do sistema, diferenciando-se, por exemplo, os dados globais daqueles locais. Por dados locais, entendem-se dados que só intervêm em certas condições ou ações, e que são acessíveis em certos instantes. As redes coloridas associam, de forma estruturada, uma parte dos dados às *marcas*, permitindo que cada marca possa ser unicamente identificada [Jensen, 1996].

Associação de Cores às *Marcas*

Com o objetivo de diferenciar as marcas são associadas cores (números inteiros ou conjunto de etiquetas) a estas. Como conseqüência, a cada lugar se associa o conjunto de cores das marcas que podem pertencer a este lugar. A cada transição se associa um conjunto de cores que corresponde às diferentes maneiras de disparar uma transição. Nos casos mais simples, quando todos os processos possuem rigorosamente a mesma estrutura e são independentes uns dos outros, as cores das transições são diretamente associadas aos processos, e o conjunto de cores dos lugares e das transições são idênticos.

No modelo de rede de Petri colorida, à cada marca é associada uma cor. Assim, a cada lugar é associado explicitamente um conjunto de cores. Essa cor representa uma informação [Kristensen, Mitchell, Zhang e Billington, 2002].

Cada marca representa um indivíduo passando de um lugar a outro, ou representa uma informação que se interpreta de acordo com o lugar que a contém.

Modelo de Execução

Em um modelo de rede de Petri Colorida uma transição só está habilitada se e somente se há marcas sufucientes com as cores determinadas, conforme os pesos dos arcos de entrada em cada lugar de entrada. Essa condição deve permanecer verdadeira até o disparo. Com o disparo, as marcas são consumidas conforme os pesos e as regras de disparo relacionadas às cores das marcas e novas marcas são produzidas nos lugares de saída conforme as regras de cores.

2.3 Sistemas Digitais

O primeiro processador do mundo nasceu juntamente com o primeiro computador, o Eniac (*Eletronic Numerical Integrator and Calculator*), construído por John Von Neuman em 1946 [Raskin e Mattos, 1994]. O processador é a unidade principal do computador; ele controla o fluxo dos programas, executa operações lógicas

e aritméticas, acessa a memória, faz solicitações aos periféricos, sendo basicamente o que hoje chama-se Unidade Central de Processamento(CPU).

Com a evolução do transistor e da microeletrônica iniciou-se um processo de compactação dos circuitos do processador, que resultou no desenvolvimento dos microprocessadores. Em 1971, na Intel Corporation, Ted Hoff construiu um processador que tinha todas as unidades reunidas em um só *chip*, o 4004, o primeiro microprocessador, desenvolvido para uso em calculadoras.

A diferença básica entre o processador tradicional e o microprocessador é o fato de este último poder ser produzido em larga escala, diminuindo drasticamente o custo. Por isso os microprocessadores se espalharam pelo mundo e conquistaram o mercado.

Desde o Eniac, os processadores utilizam conjuntos de instruções (*instruction sets*) bastante complexos. Esse tipo de arquitetura, por sua difícil construção, exige que o processador analise as instruções e execute pequenas sub-rotinas gravadas dentro do próprio processador.

Acreditando que essas sub-rotinas ou microcódigos fossem contraprodutentes, Jonh Cocke, da IBM, teve a idéia de construir um processador mais simples, que não necessitasse de microcódigo, deixando o trabalho pesado para os programas.

Esse raciocínio foi possível a partir de um estudo em que se identificou que a maioria das instruções de máquina eram usadas com pouca freqüência (cerca de 20% delas era usada 80% das vezes). Os próprios desenvolvedores de sistemas operacionais habituaram-se a determinados subconjuntos de instruções, tendendo a ignorar as demais, principalmente as mais complexas.

Estava criada a filosofia do computador com conjunto reduzido de instruções [Hwang, 1993], um processador menor e mais barato denominado RISC. RISC é a abreviatura de *Reduced Instruction Set Computer*, computador com conjunto reduzido de instruções, e é a identificação de um tipo de arquitetura de CPU (e, conseqüentemente, de todo um sistema de computação) que se contrapõe à arquitetura até então predominante, denominada CISC - *Complex Instruction Set Computer*, ou computador com conjunto complexo de instruções [Monteiro, 2001].

Apesar de inventada em 1974, a filosofia RISC só chegou ao mercado em 1985, pelas mãos da *Sun Microsystems*, com o *Sparc*. Hoje tem-se como microprocessadores CISC grande parte da família Intel, dos tradicionais X86 ao Pentium. Do outro lado estão o consórcio *Sun, PowerPC, MIPS, HP e Digital*, cada qual com seu chip RISC.

2.3.1 Arquitetura RISC

A arquitetura RISC é constituída por um pequeno conjunto de instruções simples que são executadas diretamente pelo *hardware*, sem a intervenção de um interpretador (microcódigo), ou seja, as instruções são executadas como se fossem apenas uma microinstrução.

Existe um conjunto de características que permitem a definição de uma arquitetura RISC básica; são elas:

- o coração de todo computador é o *datapath* (ULA, registradores e os barramentos que fazem sua conexão); uma das maiores características das máquinas RISC é utilizar apenas uma instrução por ciclo do *datapath*;
- arquitetura carrega/armazena, ou seja, qualquer referência à memória é feita apenas por instruções especiais de *load/store*;
- inexistência de microcódigo; sendo assim, a complexidade de projeto está no compilador;
- instruções de formato fixo, permitindo uso consistente do formato e facilitando a decodificação de instruções por controle fixo, tornando mais rápidas as tarefas de controle;
- conjunto reduzido de instruções, facilitando a organização da Unidade de Controle de modo que esta tenha uma interpretação simples e rápida;
- utilização de *pipeline*, abrindo espaço para execução simultânea de múltiplas instruções;

A especificação de formatos e o número de instruções são os pontos mais discutidos quando se fala de arquitetura RISC. Entretanto, uma generalização bem feita da teoria RISC vai muito além, indicando uma predisposição para estabelecer livremente compromissos de projeto através das fronteiras arquitetura/implementação e tempo de compilação/tempo de execução, de modo a maximizar o desempenho medido em algum contexto específico.

2.3.2 Compiladores RISC

Os processadores RISC são geralmente mais velozes do que os processadores CISC populares. O desempenho de qualquer processador RISC está estreitamente ligado ao compilador e à tecnologia de otimização do código gerado. Compiladores orientados a CISC (como *Microsoft*, *Borland Internacional*, *Symantec*, etc.) competem com base na facilidade de desenvolvimento. Na geração de códigos por eles, otimizações independentes de processador são a regra, devido a natureza complexa de seus conjuntos de instruções.

Os compiladores RISC trabalham muito com otimização dependente do processador, para aproveitar o máximo de desempenho do *hardware*. O agendamento de instruções, o gerenciamento de *cache* e seguimento de registradores são vitais, tendo em vista a natureza canalizada e de utilização intensiva de memória dos processadores RISC [Hwang, 1993].

As ferramentas para desenvolvimento RISC se espelham naquelas usadas para desenvolvimento CISC, que ainda são mais abundantes e fáceis de usar, apesar de suas similares para RISC estarem começando a preencher essa lacuna. Como compiladores e interpretadores para linguagens de 4ª geração são usados para a construção desses aplicativos, não é possível construí-los sem ferramentas especialmente projetadas para essa arquitetura. Esses compiladores e interpretadores são a chave para o sucesso de plataformas RISC.

Como o desempenho RISC depende tanto de tecnologia de compilação, os criadores de compiladores freqüentemente trabalham em conjunto com os arquitetos de *hardware* no projeto do integrado. É por isso que as companhias que vendem os

integrados constroem também os compiladores RISC específicos para eles.

2.4 *Pipelines*

Como o *pipeline* será o caso de estudo sobre a efetividade das diferentes técnicas para tratamento de tempo em RdP's é necessário um exame mais detalhado sobre o mesmo. Uma das características mais relevantes da arquitetura RISC é o uso altamente produtivo de *pipeline*, obtido em face do formato simples e único das instruções de máquina. A técnica *pipeline* funciona mais efetivamente quando as instruções são todas bastantes semelhantes, pelo menos no que se refere ao seu formato e complexidade [Monteiro, 2001].

2.4.1 Conceitos Básicos

Pipeline é uma técnica em que múltiplas instruções são sobrepostas durante a execução. Para sobrepor estas instruções, é necessário dividir o *pipeline* em estágios. Cada estágio executa uma fase de uma instrução e como os estágios estão conectados em seqüência, as instruções passam por todos eles até que sejam completadas.

O *pipeline* não diminui o tempo de execução individual das instruções. O aumento na velocidade de processamento das instruções no *pipeline* é determinado pela freqüência das instruções que saem do *pipeline*. Através da comparação visual entre a forma seqüencial e a sobreposição de instruções (Figura 2.2), é fácil perceber a diminuição do tempo necessário para se executar o mesmo número de instruções [Milutinović, Fura e Helbig, 1991].

Para facilitar a implementação dos estágios é interessante que eles funcionem de forma sincronizada, de modo que todos estejam prontos ao mesmo tempo. O tempo necessário para mover uma instrução de um estágio para o outro no *pipeline* é chamado de ciclo de máquina. Como os estágios devem ser completados ao mesmo tempo, e portanto devem estar prontos ao mesmo tempo, o tamanho do ciclo de

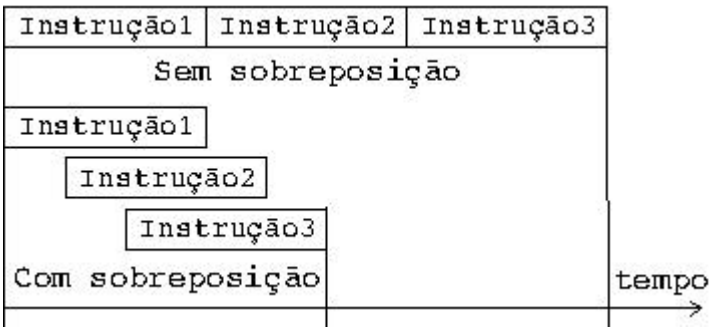

Figura 2.2: Comparação do tempo de execução de instruções com e sem sobreposição.

máquina é determinado pelo tempo requerido pelo estágio mais lento, como visto na Figura 2.3. Isso é necessário para que ocorra um sincronismo entre os estágios do *pipeline*. Desse modo é importante que o projetista de um processador faça o melhor balanceamento possível entre os tempos de execução de seus estágios, o que permitiria a sua otimização [Hayes, 1988].

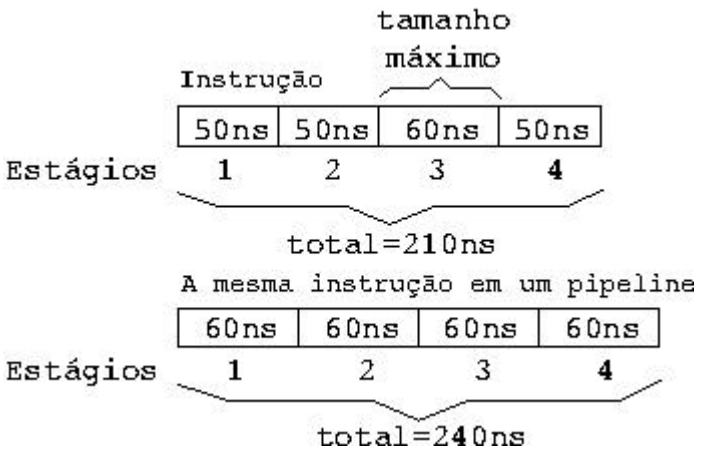

Figura 2.3: Encontrando o tamanho de um ciclo de máquina.

O ideal seria fazer com que um *pipeline* consiga executar uma instrução para cada ciclo de máquina. Entretanto a média de uma instrução por ciclo não é atingida pois, por exemplo, a execução de um teste de decisão pode indicar como próxima instrução a ser executada uma que ainda não esteja no *pipeline*. Para atenuar esses prejuízos usam-se algumas técnicas para evitar a perda de instruções carregadas

no pipeline, tais como o carregamento especulativo, descrito mais adiante, na seção 2.4.4.

2.4.2 Estágios do *Pipeline*

O conceito fundamental envolvido em *pipeline* é tornar o processador capaz de processar diversas intruções simultaneamente, procurando fornecer ao menos o resultado de uma instrução por ciclo de processamento. Para garantir o processamento simultâneo, o *pipeline* é dividido em estágios de processamento, que podem ser sobrepostos e com isso aumentar o desempenho das máquinas que contêm esse recurso. A quantidade de estágios varia de acordo com a arquitetura desenvolvida pelo fabricante. Nesse texto serão descritos apenas os quatro estágios mais básicos de um *pipeline*, embora já existam arquiteturas, chamadas de *superpipelines*, com 12 ou mais estágios, como é o caso do *Pentium 4*.

Estágio de Busca

Antes de serem executadas, as instruções são buscadas no *cache* de instruções e colocadas no *buffer* de instruções, de onde eventualmente elas irão ser selecionadas para serem decodificadas e em seguida executadas. O acesso ao *cache* é feito durante o estágio de busca. O número de instruções buscadas depende da máquina, por exemplo, o *UltraSPARC* pode buscar até 4 instruções simultâneas seguindo a informação de predição. O grande fluxo fornecido pelo *cache* faz com que o *buffer* de instruções não fique sem instruções neste estágio. Exceções para esta regra ocorrem quando ramificações são difíceis de predizer ou quando o nível de faltas (*cache misses*) for grande.

Estágio de Decodificação

Depois de buscadas, as instruções são pré-decodificadas e enviadas para o *buffer* de instruções. Os *bits* pré-decodificados gerados durante este estágio acompanharão as instruções enquanto elas estiverem no *buffer* de instruções.

Estágio de Execução

Dados do conjunto de registradores de inteiros são processados durante este ciclo. São computados os resultados, que são eventualmente disponibilizados para outras instruções no próximo ciclo. O endereço virtual de uma operação de memória também é calculado durante este estágio, em paralelo com a computação na ULA.

Estágio de Escrita

Neste estágio todos os resultados são escritos nos conjuntos de registradores (inteiro e ponto-flutuante). Todas as ações executadas durante este estágio são irreversíveis. Depois deste estágio as instruções são consideradas terminadas.

2.4.3 Conflitos (HAZARDS)

A arquitetura *pipeline* traz consigo excelentes recursos que melhoram o desempenho de uma máquina. No entanto, traz também algumas situações de conflitos, ou seja, situações em que o processamento da instrução ocorre de forma indesejada, podendo acarretar erro no resultado final da execução das instruções. Serão descritos a seguir alguns destes conflitos:

Estrutural

O primeiro conflito é chamado de estrutural. Eles ocorrem quando o *hardware* não pode suportar a combinação das instruções que estão sendo executadas no mesmo ciclo de relógio.

Um exemplo deste problema é quando uma instrução está querendo ler um dado da memória enquanto outra instrução está querendo escrever na mesma memória.

Controle

O conflito de controle surge da necessidade de se tomar uma decisão baseada nos resultados de uma instrução enquanto outras estão sendo executadas.

Um exemplo deste conflito são os desvios, quando a instrução que está sendo executada vai decidir qual será a próxima. Este problema pode ser tratado de duas formas. Na primeira, espera-se para ver o resultado (impedindo novas instruções de entrarem no *pipeline*) e depois continua-se o procedimento normal. Na segunda, admite-se que a condição examinada será sempre verdadeira ou falsa, preenchendo-se o *pipeline* para essa hipótese. Caso não se acerte a previsão, o *pipeline* será esvaziado e entrarão novas instruções, tendo um atraso por isso.

Dados

O conflito de dados ocorre quando uma instrução depende do resultado de uma instrução que ainda está no *pipeline*. Por exemplo, suponha que tenha uma instrução *add* seguida imediatamente por uma *sub* que utiliza o resultado da soma anterior, como no exemplo a seguir:

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

Sem intervenção alguma este conflito de dados poderia atrasar o *pipeline*, uma vez que a instrução *add* não escreve o resultado $s0 antes do estágio seguinte à execução. Uma forma de tratar esse conflito é utilizar os dados que acabaram de ser calculados no estágio de execução no próximo ciclo, sem que se precise esperar que os dados sejam gravados, esta técnica é conhecida como "*forwarding*" [Tanenbaum, 1996], e é vista na Figura 2.4.

Outra forma de tratar este problema é através do compilador, que colocaria outras instruções entre as instruções dependentes. Assim haveria tempo para que a primeira se complete antes que a segunda instrução necessite desses dados.

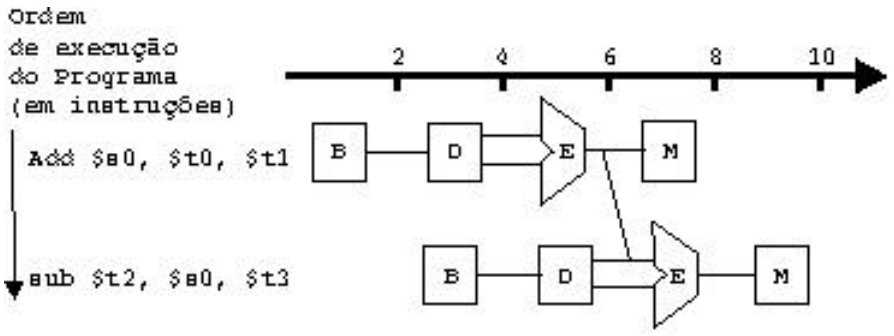

Figura 2.4: Representação gráfica de um "*forwarding*".

2.4.4 Carregamento Especulativo

Embora não diretamente associado ao *pipeline*, um fator de aceleração na execução das instruções diz respeito à disponibilidade das instruções (e dados) em memória *cache*. Em particular é necessário reconhecer que o fluxo de instruções no *pipeline* apenas pode ser mantido se as mesmas estiverem facilmente disponíveis em *cache* e não na memória RAM. Uma técnica para que isso seja possível é através do carregamento antecipado das instruções em *cache*, o que é conhecido como carregamento especulativo.

O carregamento especulativo permite a carga de uma instrução antes que o programa venha a necessitá-lo. A idéia básica é separar o carregamento da instrução do instante de sua utilização. A instrução sempre estará disponível e, por isso, o processador não se detém esperando que a memória, ou o *cache*, seja lido. O compilador é o responsável por analisar o programa buscando operações que necessitam de dados que estão na memória. Sempre que possível, o compilador insere antecipadamente uma instrução de carregamento (bem antes do ponto em que será necessário). Isso resulta que, muito antes da instrução ser necessária, ela já está sendo buscada na memória.

2.4.5 Desvios (Branches)

Infelizmente, estudos mostram que cerca de 30% de todas as instruções são desvios, e estes causam danos ao *pipeline* [Tanenbaum, 1996]. Desvios podem ser

classificados em duas categorias: incondicionais e condicionais. Um desvio incondicional diz que o computador deve parar de buscar instruções consecutivamente, e ir para algum endereço específico. Um desvio condicional testa alguma condição e desvia o fluxo de instrução se a condição é satisfeita. Um exemplo típico é uma instrução que testa algum registrador e desvia se o registrador contém zero. Se o registrador não contém zero, o desvio não é feito e o controle continua na seqüência corrente.

Instruções de laço tipicamente decrementam um contador de iteração, e então desviam de volta ao início do laço se ele é diferente de zero (isto é, ainda existem mais iterações a serem feitas). Instruções de laço são um caso especial importante de desvios condicionais, pois é conhecido a priori que os desvios quase sempre ocorrem.

2.4.6 Tratamento de Desvios

Dois tipos de previsões são possíveis: estática (tempo de compilação) e dinâmica (tempo de execução). Com previsão estática, o compilador faz uma suposição a respeito de cada instrução de desvio que ele gera. Com instruções de *loop*, por exemplo, adivinhar que ela irá desviar de volta ao início é o correto na maioria das vezes. Durante o teste de uma condição improvável, como uma chamada de sistema retornando um código de erro, a não realização do desvio é o que tem mais chance. Em muitos casos, instruções diferentes são usadas para esses casos, permitindo seu reconhecimento apenas com o exame do código da instrução [Smith, 1981].

No caso dos desvios provocados por laços, uma forma fácil de resolver é "desenrolando" o laço para que todas as iterações fiquem visíveis. Essa técnica pode ser utilizada quando o número de iterações é pequeno [Rajagopalan e Allan, 1993].

Um esquema mais elaborado exige que os projetistas da máquina providenciem dois códigos para cada tipo de desvio, fazendo com que o compilador use o primeiro se ele achar que o desvio será realizado, e o segundo, caso contrário [Ramirez, Larriba-Pey e Valero, 2001] [Smith, 1981]. Alternativamente, para programas que são muito utilizados, o programa pode primeiro ser executado em um simulador, e o comportamento de cada desvio ser realmente guardado. Então, o programa binário pode ser alterado, substituindo cada desvio com o código apropriado (prova-

velmente será realizado/provavelmente não será realizado).

O outro método de previsão é o dinâmico. Durante a execução, o microprograma monta uma tabela dos endereços contendo desvios e mantém informações a respeito do comportamento de cada um. Este método tende a tornar a máquina mais lenta por causa da manutenção do registro, de modo que alguma ajuda de *hardware* pode ser necessária. Medições mostraram que a obtenção de 90% de acerto desta maneira não é difícil [Rajagopalan e Allan, 1993] [Smith, 1981].

A previsão de desvios não é o único meio de ataque. Outro é tentar determinar de que maneira será o desvio antecipadamente no *pipeline*. Alguns testes, como desvios "se igual", são muito mais fáceis de fazer do que desvios "se menor que". O primeiro pode ser feito com um comparador, enquanto o último requer um ciclo completo pelas vias de dados para fazer uma subtração. Com esta técnica, sempre que um desvio for encontrado, o microprograma realiza uma rápida verificação num estágio inicial do *pipeline* para ver se ele pode resolver o desvio imediatamente. Se puder, ele sabe onde continuar buscando. Compiladores podem fazer muito para ajudar. Por exemplo, quando o programador escreve um *loop* fazendo i variar de 1 a 10, o compilador pode testar i para ver se i é igual a 10, em vez de ver se ele é menor que 11, de modo que o microprograma pode realizar uma comparação em vez de uma subtração.

Para tratar desvios que não podem ser resolvidos antecipadamente, o compilador pode tentar encontrar alguma rotina útil para o computador fazer enquanto estiver esperando que o desvio seja executado [Allan, Shah e Reddy, 1995]. Considere a Figura 2.5(a), um comando aritmético seguido por um teste. Um compilador com otimização muito inteligente poderia produzir código como na Figura 2.5(b), que não é válido em Pascal, mas mostra a ordem dos eventos. Ele primeiro gera código para realizar o teste, depois realiza a operação aritmética. Depois que o desvio entrou no *pipeline*, muitas instruções comuns se seguem. Estas devem ser feitas de qualquer jeito, de modo que não há necessidade de fazer uma previsão e não há necessidade de fazer *squashing* (esvaziamento do *pipeline*). Para usar esta técnica efetivamente, os microprogramadores e os escritores de compiladores devem trabalhar juntos durante o projeto.

```
a:=b+c;                    if b<c
if b<c                     a:=b+c;
  then comando;              then comando;
      (a)                        (b)
```

Figura 2.5: (a)Um trecho em Pascal. (b)Como o compilador poderia tratá-lo.

Outra técnica de previsão de desvio existente é a seguinte: existe um contador único de acertos ou erros de previsão. Para cada previsão de desvio feita é observado o acerto ou o erro. Em caso de acerto é acrescentado uma unidade no contador. Esse contador tem um limite de 5 unidades. Em caso de erro é decrementado uma unidade do contador. Enquanto o contador permanecer positivo, a previsão segue a última previsão feita. Toda vez que o contador chega a zero a previsão é trocada [Evers, 2001]. Esta técnica é a mesma utilizada nos modelos descritos nesse trabalho.

Finalmente, ainda há a possibilidade de seguir dois caminhos em paralelo. Isso requer dois *pipelines* no *hardware* e não elimina o problema de *squashing*. Nos grandes computadores, em que o gargalo é o desempenho e não o preço, este método é algumas vezes utilizado. É claro que, se qualquer caminho encontrar outro desvio pela frente antes que o primeiro seja resolvido, o tratamento desse problema fica ainda mais complicado. Ter algumas dúzias de *pipelines* para tratar do pior caso provavelmente não é uma boa idéia.

2.4.7 Tratamento de Conflitos em Algumas Tecnologias

Intel Pentium III

Por um conflito de desvio entende-se que a uma certa altura de processamento pode ocorrer a necessidade de um desvio e este necessitar de instruções que ainda estão por terminar. Considerando que este realmente ocorra, o processador cai em uma via sem saída pois deve realizar um desvio condicional sem ainda saber quais são as condições, uma vez que as instruções que forneceriam tais condições ainda estão por serem processadas. Posto isto, é necessário haver um planejamento de

como lidar com tais situações. Em muitos casos, como no do Pentium III, é utilizado um algoritmo para prever quando da ocorrência de um desvio condicional e assim evitar o respectivo conflito de desvio.

A arquitetura elaborada pela Intel para o Pentium III utiliza um algoritmo para prever quando da ocorrência de um desvio e assim não precisa necessariamente processá-lo por completo. Ao invés disso, antes mesmo deste ocorrer o processador já sabe se deve ou não realizar o desvio. Tal algoritmo é complexo e envolve a seqüência de instruções atuais bem como uma parte das já processadas. O cerne da questão consiste no fato do algoritmo gerar uma condição e testá-la logo que a instrução de desvio for "quebrada" em operações menores. Em caso positivo, ou seja, a condição prevista é a mesma que o algoritmo previu, o processador continua a execução naturalmente e descarta o desvio uma vez que já sabe quais decisões adotar. Em caso contrário, houve falha na previsão sendo necessário trocar o *status* de todas as micro operações na Unidade de Relação de Instruções e reiniciar a análise do desvio atual.

MIPS

A arquitetura MIPS não faz uso de um algoritmo complexo de previsão de desvios, sendo assim é quase que padrão processar e analisar todas as instruções de desvios. Esta configuração possui um desempenho bastante inferior ao do Pentium III, porém evita várias situações de conflitos de desvios.

O funcionamento da previsão de desvios pode ser explicado pelo seguinte conceito: O desvio é previsto ainda no estágio ID (*instruction decode*) e há também um *buffer* especial que armazena as últimas vezes em que um desvio foi previsto com sucesso, tal *buffer* é utilizado como coeficiente de previsão do algoritmo implementado. Em caso de erro de previsão apenas uma instrução (o desvio) deve ser removida do estágio de *pipeline*, minizando os estragos quando da ocorrência do respectivo erro. Em caso positivo, o desvio é aceito sendo apenas trocada a posição do contador de programa (PC) pelo valor presente no código do desvio (essa técnica será utilizada neste estudo para construir o módulo de previsão de desvios).

Alguns Outros Casos de Conflitos na Arquitetura MIPS

Dentre os conflitos da arquitetura MIPS, vale destacar o caso de conflito estrutural. Tal consiste quando um recurso de *hardware* tenta ser utilizado em duas situações ao mesmo tempo, a exemplo quando da busca de uma instrução bem como a tentativa de um acesso a memória. A arquitetura MIPS implementa uma saída bastante peculiar para este caso de forma a evitar conflitos estruturais. Existem duas regiões distintas de memória sendo que uma delas trabalha com as instruções e outra com os dados. Sendo assim é perfeitamente possível buscar uma próxima instrução enquanto se realiza o acesso a memória para o tratamento de uma instrução já sendo processada.

Outro tipo de conflito existente consiste no conflito de dados. Para esta situação o MIPS implementa uma unidade chamada *Forwarding Unit* a qual repassa o dado atual do fim de um estágio de Execução do *datapath* para o início do estágio de Execução do *datapath* seguinte, Figura 2.4. Esta solução é bastante viável porém há ainda um caso de conflito de dados. Este consiste na existência de uma instrução que lê um dado de um registrador seguida de uma instrução que grava um dado no mesmo registrador. Para solucionar tal conflito, é necessária a incluisão de ciclos de espera (*stall cycles*) para que somente seja feita a escrita quando a outra instrução já conseguiu ler o dado com sucesso, Figura 2.6. O MIPS possui uma unidade de detecção de conflitos para ser capaz de trabalhar com situações como estas. Esta unidade verifica se o registrador de destino de uma dada instrução é o registrador de origem da instrução seguinte. Em caso positivo a unidade inclui ciclos de espera para assegurar a validade dos dados.

Técnica Utilizada no IA-64

Os compiladores para as CPU's IA-64 usam uma técnica denominada "*Predication*", que permite eliminar os efeitos danosos provocados pelos desvios. Ao encontrar um desvio, diferente do que acontecia com os modelos anteriores, a CPU não precisa mais apostar um dos ramos. Ela simplesmente executa todos os desvios possíveis e depois descarta os resultados irrelevantes. Todos os desvios e suas rami-

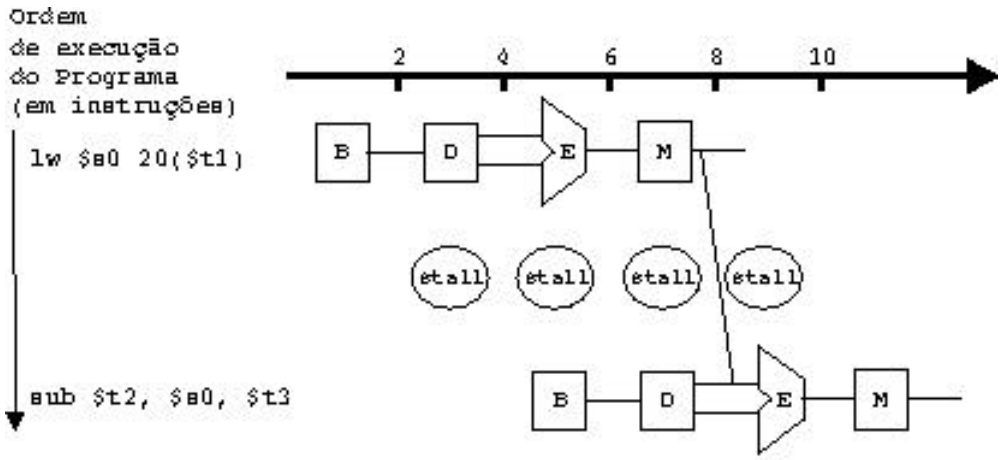

Figura 2.6: Representação gráfica da inserção de um ciclo de espera (*stall*).

ficações são preditos pelo compilador. Assim, durante a execução, quando a CPU encontra um desvio já predito pelo compilador, ela começa a executar os códigos ao longo de todos os possíveis destinos. Quando o destino correto é decidido, a CPU armazena os resultados válidos e descarta os demais.

É muito importante o papel do compilador. Quando este encontra um desvio, ele o analisa para ver se é um candidato para a predição. A grande maioria dos desvios o é. Caso positivo, marca com um *predicate* diferente as instruções de cada ramo do desvio. Por exemplo, uma instrução do tipo "salte se a variável R for igual a zero" tem duas ramificações possíveis, como é mostrado na Figura 2.7.

As instruções pertencentes a um mesmo ramo têm o *predicate* igual. Existem disponíveis 64 *predicates* diferentes. Após marcar as instruções de cada ramo do desvio, o compilador verifica quais podem ser executadas em paralelo e depois empacota as instruções em palavras de 128 *bits*, de forma que o microprocessador as receba arrumadas da melhor forma possível. Durante a execução, a CPU despacha para execução em paralelo as instruções que não têm dependências e cuida para que as que apresentam dependências sejam executadas na ordem correta.

O lado ruim desta técnica de predição é que a CPU sempre executa instruções cujos resultados serão descartados; porém, segundo especialistas, haverá um ganho em relação às arquiteturas atuais [Raskin e Mattos, 1994].

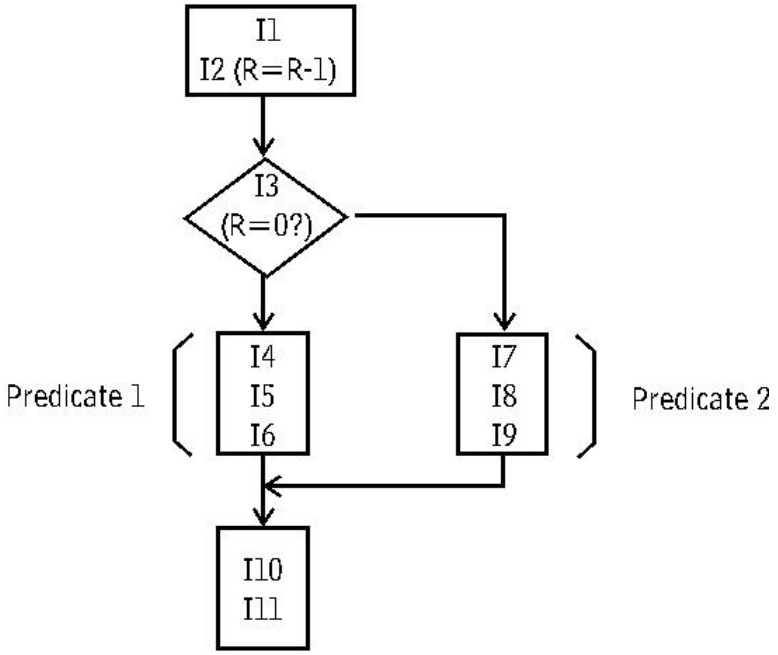

Figura 2.7: Ilustração da operação de predição dos desvios.

Capítulo 3

Modelagem do Sistema

3.1 Módulos Básicos dos Modelos

O modelo do *pipeline* foi dividido em quatro módulos: os estágios, gerenciamento de *caches*, tratamento de desvios e limpeza dos estágios em caso de erro de previsão. Para a descrição de cada um destes módulos será usada uma rede de Petri Colorida. Ao final da descrição apresenta-se o modelo completo, com a explicação de cada transição e de cada lugar que compõe o *pipeline*.

Como o objetivo deste trabalho é comparar as técnicas de rede de Petri usadas para modelar sistemas digitais complexos, foi escolhido a técnica de *pipeline* em função de sua complexidade e da importância dessa técnica na construção dos microprocessadores. Este capítulo pretende descrever cada um dos módulos que foram criados para modelar o *pipeline*.

3.1.1 Os Estágios

Um *pipeline* é formado por estágios que executam funções diferentes. Para esta modelagem escolheu-se um *pipeline* simples com quatro estágios: busca, decodificação, execução e gravação dos dados.

A função deste bloco da modelagem é apenas demonstrar o caminho que

os dados percorrem durante a execução de uma tarefa pela CPU. A Figura 3.1 mostra a modelagem dos estágios que compõem este *pipeline*.

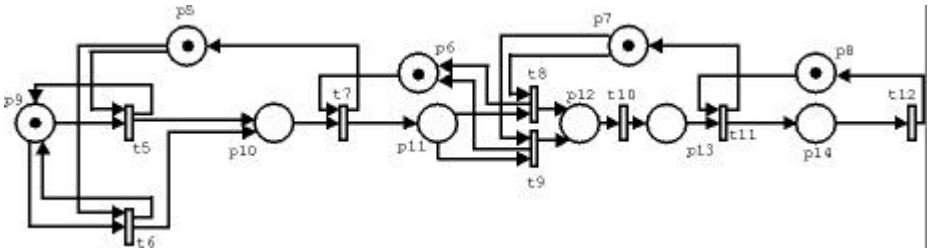

Figura 3.1: Modelagem de um *pipeline* de 4 estágios simples.

O contador de programas fornece os endereços de cada instrução que passará pelos estágios do *pipeline*. Os endereços dessas instruções ficam disponíveis (p9 com marca) aguardando a liberação do estágio de busca. Se o estágio de busca estiver pronto (p5 com marca), o endereço da instrução é passado para que o estágio busque a instrução (p10 com marca). Para que a instrução passe do estágio de busca para o estágio de decodificação, esse deve estar liberado (p6 com marca). Estando liberado o estágio de decodificação, a instrução será buscada e colocada a disposição para decodificação (marca em p11), liberando o estágio de busca para a próxima instrução (produz marca em p5). A instrução que estiver pronta para ser decodificada (p11 com marca) pode ser um desvio (disparo da transição t8) ou uma instrução simples (disparo da transição t9). Sendo essa instrução um desvio ou não, ela estará decodificada (marca no lugar p12) e aguarda a definição de erro de previsão. Depois de definida a questão do erro ou acerto da previsão (p13 com marca) e caso o estágio de escrita esteja liberado (p8 com marca) a instrução é executada (produz marca em p14). Após a execução da instrução, os dados serão gravados (produz marca em p8), terminando assim a execução de uma instrução dentro do *pipeline*.

3.1.2 Gerenciamento de *Caches*

Com a necessidade de atualização constante dos conteúdos, tanto da *cache* de dados como da *cache* de instruções, fez-se um módulo que trata da verificação dos dados presentes em cada uma das *caches*. Caso os dados presentes na *cache* estejam

incorretos, esta é esvaziada e novos dados são transferidos da memória para *cache*, permitindo que as operações referentes a esses dados sejam executadas, sejam elas sobre instruções ou dados [Smith, 1982]. O módulo de gerenciamento de dados da *cache* é definido na Figura 3.2.

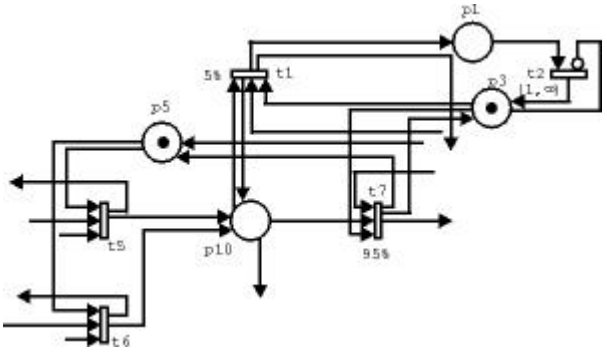

Figura 3.2: Modelagem de um bloco que gerencia *caches*.

Quando uma instrução está pronta para ser executada (marca em p10), ela habilita duas transições t1 e t7. Essas transições dependem das probabilidades associadas a elas para que uma delas possa ser disparada.

Em 5% das vezes as instruções contidas na *cache* estão incorretos (5% das vezes a transição t1 seria disparada). No intervalo de (1,∞) os dados pertencente a *cache* serão trocados por novos dados. Utiliza-se esse tempo mínimo para que a simulação possa se aproximar mais da realidade, uma vez que há um tempo mínimo gasto para a troca dos dados da *cache*. Se tiver uma marca em p1 e nenhuma marca em p3, isso significa que os dados da *cache* devem ser trocados (disparo de t2). Por último, depois de colocar novos dados na *cache* (marca em p3, se necessário), a instrução pode passar para a fase seguinte (dispara-se a transição t7 e produz uma marca no lugar do próximo estágio e em p3 avisando que mudou de instrução e é possível uma nova atualização da *cache*, se necessário).

3.1.3 Tratamentos de Desvios

Como visto no capítulo anterior, o tratamento de desvios é vital para o funcionamento do *pipeline*. Entre as várias técnicas possíveis de previsão adotou-se a

proposta por Evers, que faz a previsão de desvios baseado no histórico dos acertos e erros de previsões [Evers, 2001].

O módulo de tratamento de desvio verifica se a instrução que está no *pipeline* é um desvio ou não e, se for, qual o suposto caminho deve ser tomado. Para cada desvio que apareça no *pipeline* será proposto um caminho (continuar executando a instrução seguinte ou pular para a instrução referente ao desvio). Em caso de acerto da previsão, essa política (por exemplo, continuar executando a instrução seguinte) deve ser armazenada para que na próxima previsão, o módulo de tratamento de desvios possa levar em conta o acerto.

Existe um *buffer* que guarda as quatro últimas previsões em forma de soma para cada acerto. O cálculo é feito da seguinte forma: para cada acerto acumula-se uma marca em p17 e para cada erro retira-se uma marca e cada vez que este contador chega a zero, inverte-se o tipo de previsão (se a previsão era para que a próxima instrução fosse a do salto para o desvio, passará a ser a instrução seguinte e vice-versa). A Figura 3.3 modela o tratamento de desvios.

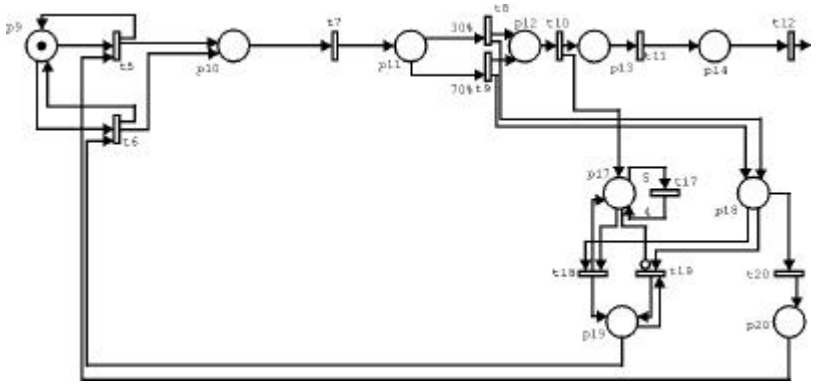

Figura 3.3: Modelagem de um bloco para predição de desvios.

A transição t10 dispara sempre que uma instrução foi prevista corretamente e produz uma marca em p17. Para cada acerto de previsão, uma marca é acumulada, porém o número de marcas em p17 não ultrapassa o valor 4. O responsável por esse limite no número de marcas é a transição t17 que caso o número de marcas chegue em 5, ela dispara, consome as 5 marcas e produz 4.

Caso não tenha nenhuma marca em p17, isso quer dizer que a previsão teve mais erros do que acertos e com isso deve-se mudar a política de previsão (se vinha usando a instrução seguinte ao desvio deve usar agora a instrução do desvio e vice-versa).

Se a última instrução decodificada for desvio (p18 com marca vermelha) permite o disparo de t18 ou t19 conforme a situação no lugar p17. Se disparar a transição t18 produz uma marca da cor azul em p19. Se disparar a transição t19 produz uma marca da cor vermelha em p19. A transição t20 só dispara se a marca em p18 for azul (a instrução decodificada não é desvio).

Se uma instrução que acabou de ser decodificada for de desvio, é analisado o que se deve fazer (qual o endereço da próxima instrução a ser buscada). Quando t6 for disparar, observa-se a cor da marca presente no lugar p19. Caso a cor da marca for azul a instrução a ser buscada deve respeitar a última previsão. Caso a cor presente em p19 seja vermelha, há uma troca da política de previsão. Quando t5 for disparar significa que a instrução decodificada anteriormente não é de desvio e a a próxima instrução a ser buscada não precisa de previsão.

3.1.4 Limpeza dos Estágios em Caso de Erro de Previsão

Nem sempre o sistema de previsão de desvios acerta. Neste caso, precisa-se limpar alguns estágios do *pipeline* para que as instruções corretas possam ser executadas. A Figura 3.4 apresenta a rede que modela a limpeza do estágio de busca em função do erro de previsão.

Assim que um desvio é decodificado, é feita uma previsão e há uma probabilidade de que esteja errada. Adota-se aqui que a probabilidade de erro dessa previsão é de 10,5% [Evers, 2001]. Baseado nisso, a transição t13 em 10,5% das vezes descartaria a instrução buscada neste instante (p10 com marca), pois provavelmente esteja errada. Sempre que ocorre isso, há uma perda do desempenho do *pipeline* pois uma nova instrução deverá ser buscada. Com a limpeza do estágio de busca, uma das marcas usadas para contar os acertos em previsões é consumida. Depois da limpeza

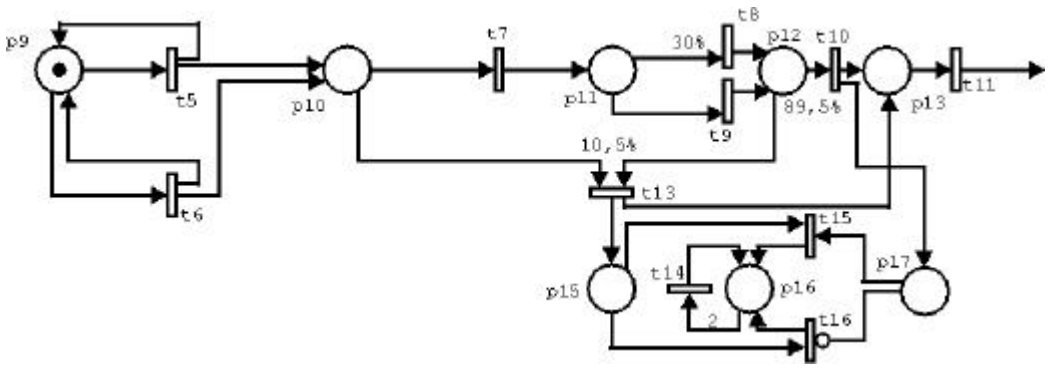

Figura 3.4: Modelagem de um bloco para limpeza do *pipeline*.

do estágio de busca uma nova marca, com outra cor, é colocada em p12 para informar que houve um erro de previsão e que o estágio referente a este erro já foi limpo para que nova instrução seja colocada nele.

Nesse modelo a instrução presente no estágio de decodificação (p11) não é retirada porque assume-se que o compilador, em caso de desvio, coloca uma instrução logo após ao desvio que será executada fora de ordem, independentemente da resposta do desvio.

3.2 Modelos Específicos

Nesta seção serão apresentados três modelos de *pipeline* usando diferentes técnicas para tratamento de tempo em redes de Petri. O primeiro modelo utiliza rede de Petri Baseada em Tempo, o segundo rede de Petri Estocástica Generalizada e o terceiro uma rede de Petri Híbrida que utiliza recursos de tempo, probabilidade e cor. A idéia central de descrever estes três modelos é demonstrar a dificuldade em simular o tempo quando se tem apenas recursos como probabilidade, simular probabilidade quando se tem apenas recursos de tempo e mostrar simplificação do modelo quando se usa tempo, probabilidade e cor. A criação de um modelo utilizando apenas rede de Petri Colorida é inviável pois é impossível simular tempo e probabilidade com esse recurso (presença de cor nas marcas).

3.2.1 Rede de Petri Baseadas em Tempo (RdPBT)

Modelo de Um *Pipeline* Usando uma RdPBT

A Figura 3.6 apresenta a modelagem de um *pipeline* de quatro estágios utilizando apenas a técnica de rede de Petri Baseada em Tempo. Ele é composto por quatro módulos: estágios, módulo de gerenciamento de *cache* de instruções e de dados, módulo de tratamento (previsão) de desvios (*branches*) e o módulo responsável por limpar o *pipeline* em caso de erro de previsão. Na tabela 3.1 apresenta-se a descrição dos lugares definidos para o modelo.

O módulo de estágios constitui os quatro estágios que compõe o *pipeline*: busca da instrução, decodificação dessa instrução, execução da instrução e gravação dos resultados. É um módulo que exige uma adaptação quanto a concorrência de disparos das transições t12 e t13 em função da necessidade de se tratar de probabilidade do disparo tendo apenas o tempo como recurso desta rede (Figura 3.6).

O módulo de gerenciamento de *cache*, que tem o funcionamento semelhante tanto para dados quanto para instruções, utiliza o recurso de tempo quando precisa provocar um atraso quando os valores presentes na *cache* precisam ser trocados.

O módulo de tratamento de desvios ficou mais complexo neste modelo em função da não utilização de cores nas marcas. Quando se utiliza cores, as marcas podem assumir "valores" diferentes e quando esse recurso não é utilizado, é necessário a criação de lugares diferentes para que transições diferentes possam tratar dessa previsão. Veja a Figura 3.5.

O módulo responsável por limpar o *pipeline* em caso de erro de previsão utiliza um recurso de tempo para modelar a questão da probabilidade de disparo envolvida. Como a probabilidade para que esse módulo entre em ação é de 11% (disparo de t19, ver Figura 3.6), foi dado um tempo de habilitação de disparo proporcional para que com isso o modelo ficasse o mais próximo possível da realidade.

Nos pontos em que existe concorrência para o disparo de transições que seriam, originalmente, coordenadas por probabilidades, como é o caso das transições

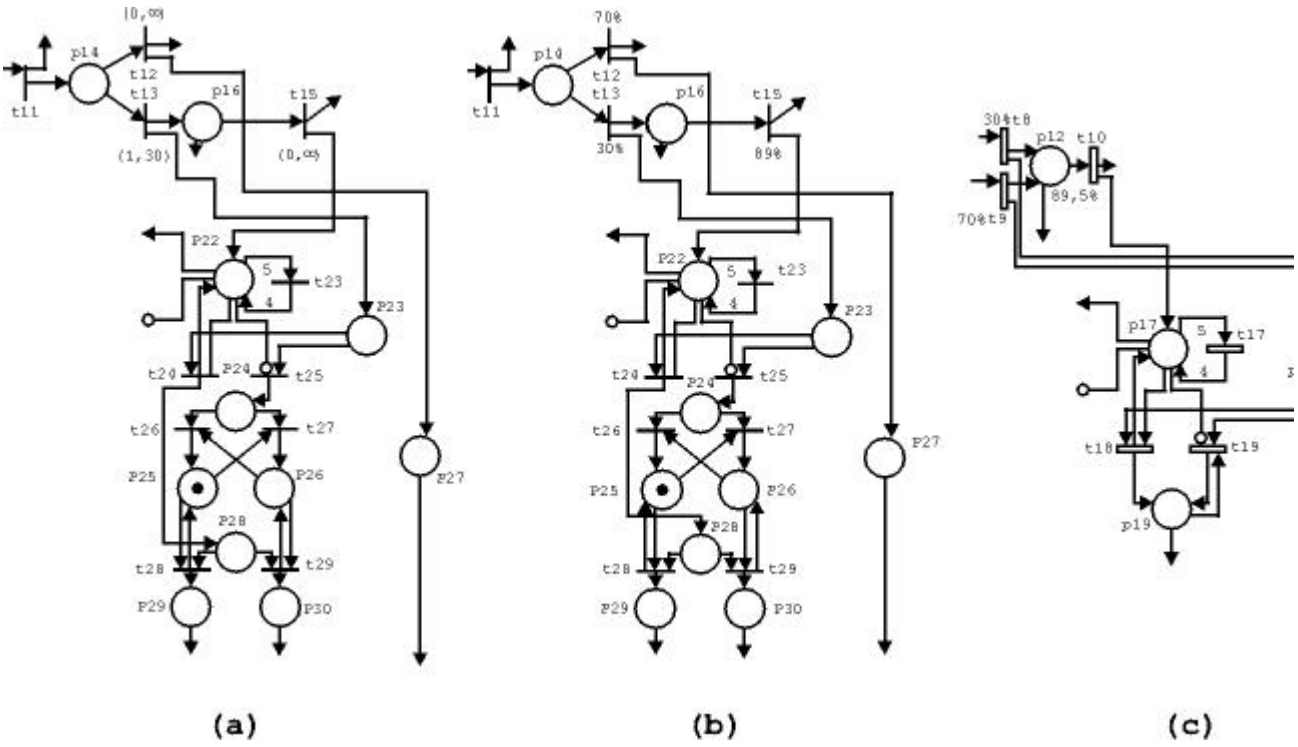

Figura 3.5: Comparação da complexidade no módulo de previsão em função da ausência de cor nas marcas. a) e b) Sem cor. c) Com cor.

t1 e t9, por exemplo, foi necessário adaptar-se intervalos de habilitação que proporcionassem taxas de disparo aproximadamente iguais às probabilidades esperadas. Os intervalos que aparecem na Figura 3.6 foram determinados empiricamente e dependem, fundamentalmente, do passo de discretização de tempo do simulador e do número médio de transições habilitadas em cada passo de simulação. A única correlação existente é de que ao determinar-se o valor ideal para uma dada probabilidade, pode-se aplicar em proporção linear esse valor de intervalo para os demais pontos de concorrência probabilística.

Lugar	Descrição
p1	Sinal de atualização dos dados da *cache* de instruções
p2	Sinal de atualização dos dados da *cache* de dados
p3	*Cache* com instruções
p4	*Cache* com dados
p5	Estágio de busca livre
p6	Estágio de decodificação livre
p7	Estágio de execução livre
p8	Estágio de gravação dos dados livre
p9	Contador de Programas
p10	Endereço das instruções
p11	Endereço das instruções esperando a verificação da *cache* de instruções
p12	Dados corretos
p13	Instrução
p14	Instrução decodificada
p15	Instrução comum
p16	Instrução de desvio
p17	Instrução esperando dados da *cache*
p18	Dados necessários para a execução das instruções estão corretos
p19	Resultado da execução das instruções
p20	Erro de previsão
p21	Descarte das marcas consumidas de p22
p22	*Buffer* de previsão
p23	Sensor de acerto de previsão
p24	Errou última previsão
p25	Instrução é desvio (*branch*), aviso de acerto de previsão
p26	Instrução é desvio (*branch*), aviso de erro de previsão
p27	Não é *branch*
p28	Acertou última previsão
p29	Endereço da instrução seguinte
p30	Endereço da instrução resultado de desvio tomado

Tabela 3.1: Descrição de cada lugar em uma Rede de Petri Baseada em Tempo.

3.2.2 Rede de Petri Estocástica Generalizada (RdPEG)

Modelo de Um *Pipeline* Usando Uma RdPEG

A Figura 3.7 apresenta o modelo de um *pipeline* de quatro estágios utilizando apenas a técnica de rede de Petri Estocástica Generalizada. Ele é composto pelos mesmos quatro módulos que compõem a rede de Petri Baseada em Tempo: estágios, módulo de gerenciamento de *cache* de instruções e de dados, módulo de tratamento (previsão) de desvios (*branches*) e o módulo responsável por limpar o *pipeline* em caso de erro de previsão. A definição dos lugares presentes nessa rede é encontrada na Tabela 3.2.

O módulo de estágios constitui os quatro estágios que compõe o *pipeline*: busca da instrução, decodificação dessa instrução, execução da instrução e gravação dos resultados. Neste módulo, usando o recurso que esta técnica permite, ficou mais fácil o tratamento dos disparos das transições t12 e t13 em função da disponibilidade do uso de probabilidades (Figura 3.7).

O módulo de gerenciamento de *cache*, que tem o funcionamento semelhante tanto para dados quanto para instruções, utiliza o recurso de probabilidade para determinar a porcentagem das vezes que os dados da *cache* serão trocados durante o andamento da simulação do *pipeline*.

O módulo de tratamento de desvios ficou mais complexo neste modelo em função da não utilização de cores nas marcas. Quando se utiliza cores, as marcas podem assumir "valores" diferentes e quando esse recurso não é utilizado, é necessário a criação de lugares diferentes para que transições diferentes possam tratar dessa previsão. Veja a Figura 3.5.

O módulo responsável por limpar o *pipeline* em caso de erro de previsão utiliza um recurso de probabilidade para modelar a questão da quantidade de vezes que o *pipeline* precisa ser limpo em função de erro de previsão.

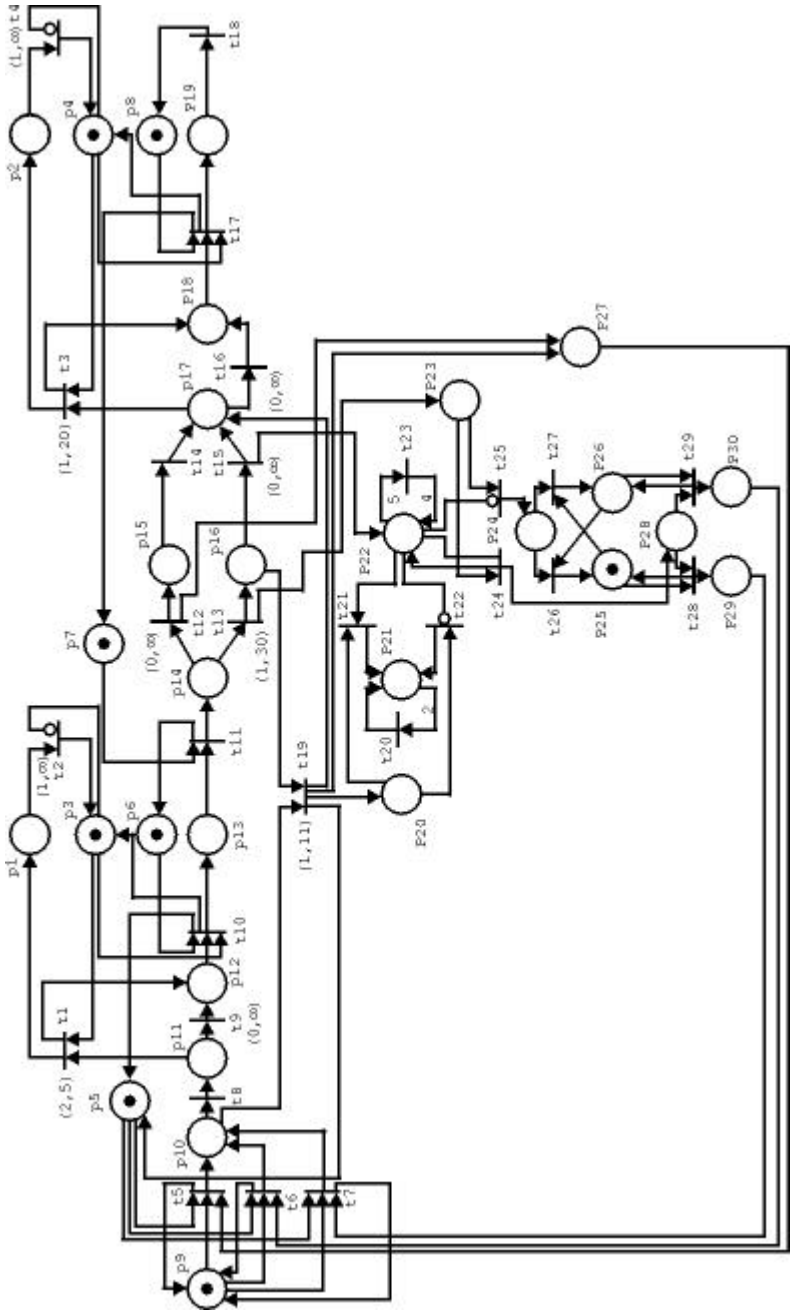

Figura 3.6: Modelagem de um *Pipeline* de 4 estágios utilizando uma rede de Petri Baseada em Tempo.

Cada um dos lugares presentes nesta modelagem está descrita na tabela 3.2

3.2.3 Modelo de Um *Pipeline* Usando Rede de Petri Híbrida (RdPH)

Define-se aqui uma rede de Petri híbrida como sendo a rede que reúne as técnicas RdP Baseada em Tempo, RdP Estocástica Generalizada e RdP Colorida, descritas anteriormente. A principal razão em se utilizar todas essas técnicas de redes de Petri em conjunto é facilitar a modelagem de um sistema, neste caso em especial um sistema digital que envolve tempo e indeterminismo, utilizando as melhores características de cada uma dessas técnicas.

Essa combinação de técnicas não aparece descrita formalmente na literatura sobre aplicações de redes de Petri, sendo portanto um diferencial entre as técnicas aqui apresentadas. O seu modelo de execução, assim como todas as restrições de habilitação e disparo de transições segue exatamente o previsto em cada um dos modelos, sempre com a restrição mais forte entre as apresentadas por cada técnica prevalecendo sobre as demais, o que permite inferir que não se fere nenhuma das exigências fundamentais de redes de Petri.

Modelo de Um *Pipeline* Utilizando Uma RdPH

Nesta seção será descrita uma rede de Petri Híbrida que modela um *pipeline* tendo como base várias técnicas disponíveis para a construção da rede, uma vez que cada uma delas trata quase que apenas de um parâmetro específico.

A Figura 3.8 é a modelagem de um *pipeline* de quatro estágios. Ele é composto pelos seguintes módulos: estágios, módulo de gerenciamento de *cache* de instruções e de dados, módulo de tratamento (previsão) de desvios (*branches*) e o módulo que trata de limpar o *pipeline* em caso de erro de previsão.

O módulo de estágios constitui os quatro estágios que compõe o *pipeline*: busca da instrução, decodificação dessa instrução, execução da instrução e gravação

Lugar	Descrição
p1	Sinal de atualização dos dados da *cache* de instruções
p2	Sinal de atualização dos dados da *cache* de dados
p3	*Cache* com instruções
p4	*Cache* com dados
p5	Estágio de busca livre
p6	Estágio de decodificação livre
p7	Estágio de execução livre
p8	Estágio de gravação dos dados livre
p9	Contador de Programas
p10	Endereço das instruções
p11	Endereço das instruções esperando a verificação da *cache* de instruções
p12	Dados corretos
p13	Instrução
p14	Instrução decodificada
p15	Instrução comum
p16	Instrução de desvio
p17	Instrução esperando dados da *cache*
p18	Dados necessários para a execução das instruções estão corretos
p19	Resultado da execução das instruções
p20	Erro de previsão
p21	Descarte das marcas consumidas de p22
p22	*Buffer* de previsão
p23	Sensor de acerto de previsão
p24	Errou última previsão
p25	Instrução é desvio (*branch*), aviso de acerto de previsão
p26	Instrução é desvio (*branch*), aviso de erro de previsão
p27	Não é *branch*
p28	Acertou última previsão
p29	Endereço da instrução seguinte
p30	Endereço da instrução resultado de desvio tomado

Tabela 3.2: Descrição de cada lugar em uma rede de Petri Generalizada Estocástica.

dos resultados. Neste módulo, usando os recursos disponíveis de tempo e probabilidade, ficou mais fácil o tratamento dos disparos das transições t12 e t13 em função dessa disponibilidade do uso da probabilidade (Figura 3.8).

O módulo de gerenciamento de *cache*, que tem o funcionamento semelhante tanto para dados quanto para instruções, utiliza o recurso de probabilidade para determinar a porcentagem das vezes que os dados da *cache* serão trocados durante o andamento da simulação do *pipeline* e utiliza o tempo para computar o atraso causado pela troca de dados na *cache*.

O módulo de tratamento de desvios ficou mais simples neste modelo em função da utilização de cores nas marcas.

O módulo responsável por limpar o *pipeline* em caso de erro de previsão utiliza um recurso de probabilidade para modelar a questão da quantidade de vezes que o *pipeline* precisa ser limpo em função de erro de previsão.

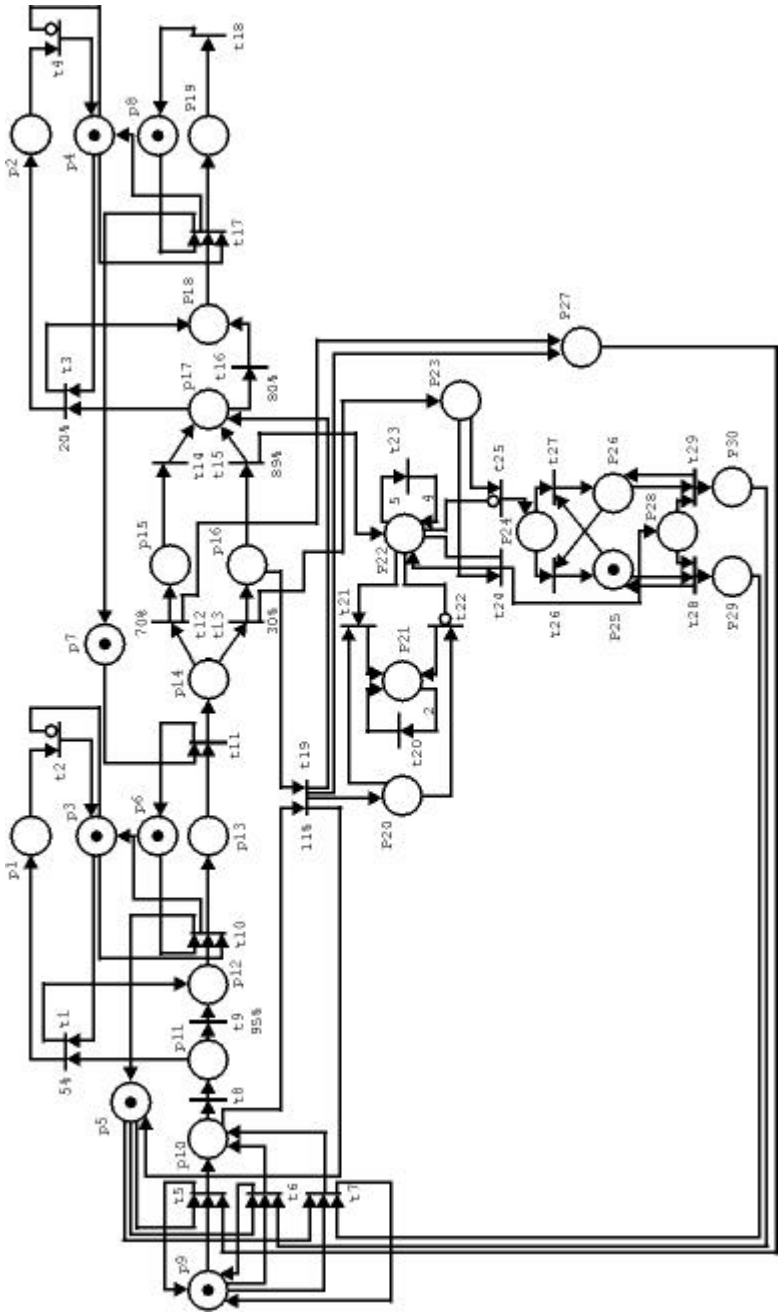

Figura 3.7: Modelagem de um *Pipeline* de 4 estágios utilizando uma rede de Petri Estocástica.

Lugar	Descrição
p1	Sinal de atualização dos dados da *cache* de instruções
p2	Sinal de atualização dos dados da *cache* de dados
p3	*Cache* com instruções
p4	*Cache* com dados
p5	Estágio de busca livre
p6	Estágio de decodificação livre
p7	Estágio de execução livre
p8	Estágio de gravação dos dados livre
p9	Contador de Programas
p10	Endereço das instruções
p11	Instrução
p12	Instrução decodificada
p13	Resultado da atualização ou não da cache de instruções
p14	Resultados da execução da instrução
p15	Erro de previsão
p16	Descarte das marcas consumidas de p17
p17	*Buffer* de previsão
p18	Sensor de *branch*
p19	Indicação de acerto ou erro da última previsão
p20	Não é *branch*

Tabela 3.3: Descrição de cada lugar em uma rede de Petri Híbrida.

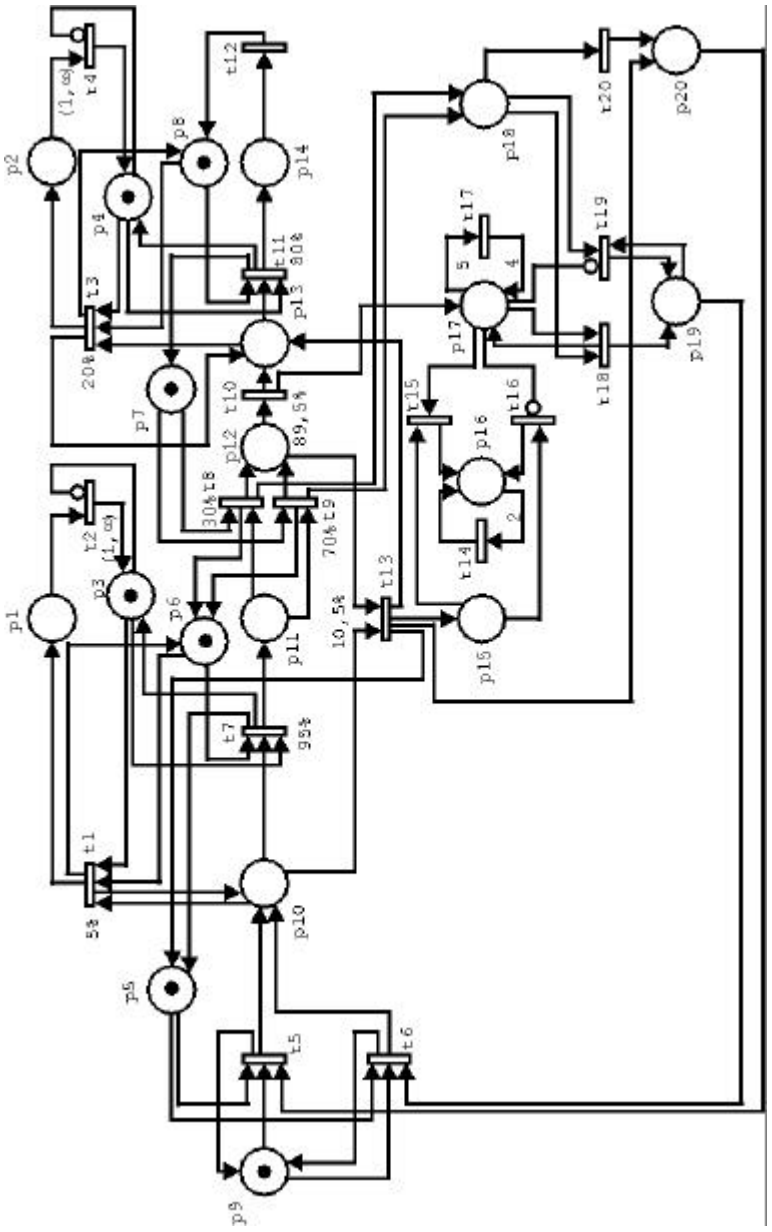

Figura 3.8: Modelagem de um *Pipeline* de 4 estágios utilizando uma rede de Petri Híbrida.

Capítulo 4

Resultados

Neste capítulo será apresentado uma comparação da utilização das redes de Petri apresentadas no capítulo anterior no que diz respeito ao seu grau de complexidade (número de transições e lugares) e as diferenças na criação dos modelos quanto ao tratamento de intervalos de tempo e eventos não determinísticos.

4.1 Comparação da Complexidade dos Modelos

Com base na Figura 4.1, nota-se o aumento do número de transições e lugares no módulo que trata da *cache* de dados em redes (Figura 4.1 (a) e (b), modelos de RdPBT e RdPEG respectivamente que possuem 8 transições e 7 lugares) que não dispõem da diferenciação de dados através de cores (marcas coloridas) disponíveis em redes de Petri Híbrida (Figura 4.1 (c), modelo de RdPH que possue 5 transições e 5 lugares).

As partes dos modelos que representam o sistema digital *pipeline* representadas na Figura 4.2 (a) e (b) mostram também o aumento do número de transições (8 transições em cada modelo (a) e (b)) e lugares (6 lugares em cada modelo) em relação ao modelo híbrido (Figura 4.2 (c), que tem 5 transições e 3 lugares), simplesmente por causa da falta de recursos de diferenciação dos dados que trafegam na rede.

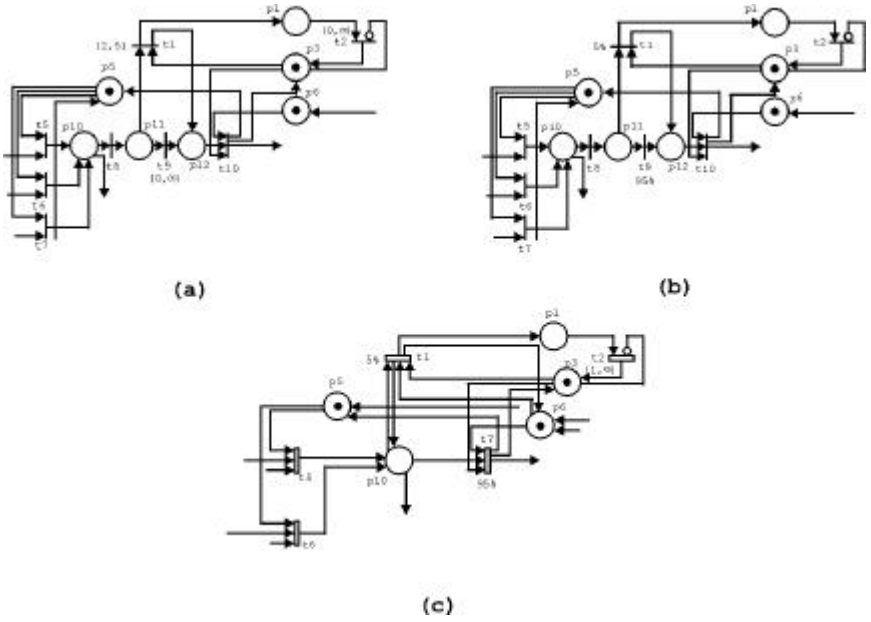

Figura 4.1: Diferença entre os modelos. (a)*Cache* de dados usando RdPBT, (b)*Cache* de dados usando RdPEG e (c)*Cache* de dados usando RdPH.

Por fim, como mostra a Figura 4.3 (a) e (b), nota-se também o aumento da complexidade da rede refletida no aumento no número de transições (10 ao todo em cada rede (a) e (b)) e lugares (11 em cada rede (a) e (b)) na tentativa de representar a função de previsão de desvios, o que no modelo criado a partir de várias técnicas em conjunto fica muito mais simples (Figura 4.3 (c) possue apenas 7 transições e 5 lugares).

O que se percebe na comparação dos módulos isolados das redes é que aquelas que utilizaram as técnicas envolvendo tempo e indeterminismo são quase que exatamente iguais, a não ser pelo fato de uma ter o tempo associado a cada uma das transições informando que há um sincronismo no disparo delas e a outra ter um determinado valor em porcentagem informando a probabilidade de disparos quando há uma concorrência entre duas ou mais transições. Isso quer dizer que o aumento da complexidade (aumento do número de transições, lugares e arcos) mostrada até agora se deve ao fato da ausência de recursos como a coloração de marcas indicando diferenças entre os dados nas técnicas que lidam com o tempo e

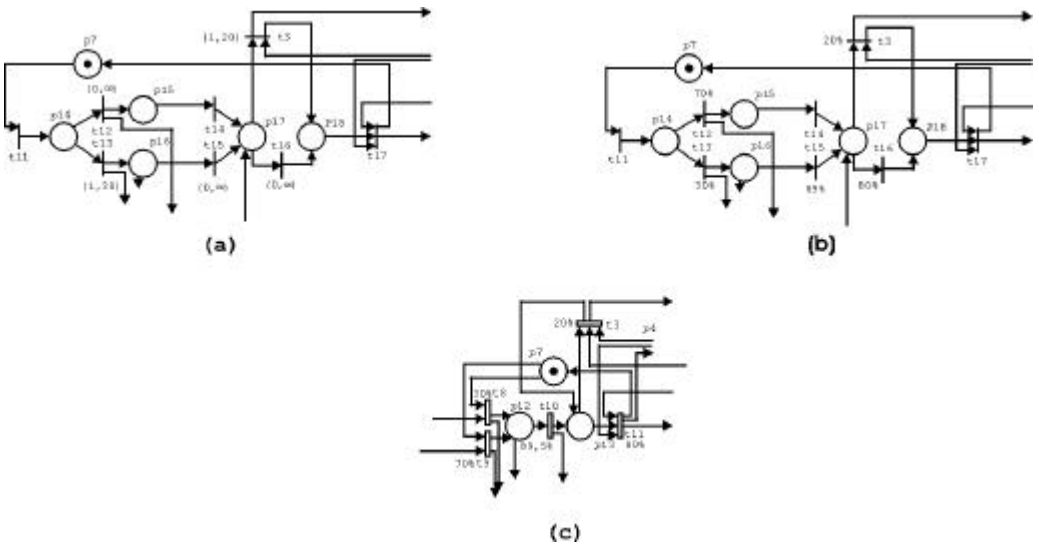

Figura 4.2: Diferença entre os modelos. (a) Estágio de execução usando RdPBT, (b) Estágio de execução usando RdPEG e (c) Estágio de execução usando RdPH.

com o não-determinismo.

Porém, essas duas técnicas que são aparentemente parecidas possuem recursos específicos e indispensáveis na construção do modelo do *pipeline* como a imposição de tempo de disparo das transições ou a utilização de probabilidade para resolver o problema de concorrência entre transições que não está disponível em técnicas como a rede de Petri Colorida, pelo menos não da forma que é necessário para a construção deste modelo.

4.2 Tratamento de Tempo e Indeterminismo

Com a técnica de rede de Petri Baseada em Tempo é mais difícil tratar indeterminismos. Para modelar um *pipeline*, foi dado um tempo diferenciado, proporcional à probabilidade de disparo de cada transição, para conjuntos de transições que continham concorrência entre si. Essa foi a técnica utilizada para substituir a probabilidade através de tempos dados às transições.

Quanto ao tratamento de tempo, quando a rede de Petri Baseada em

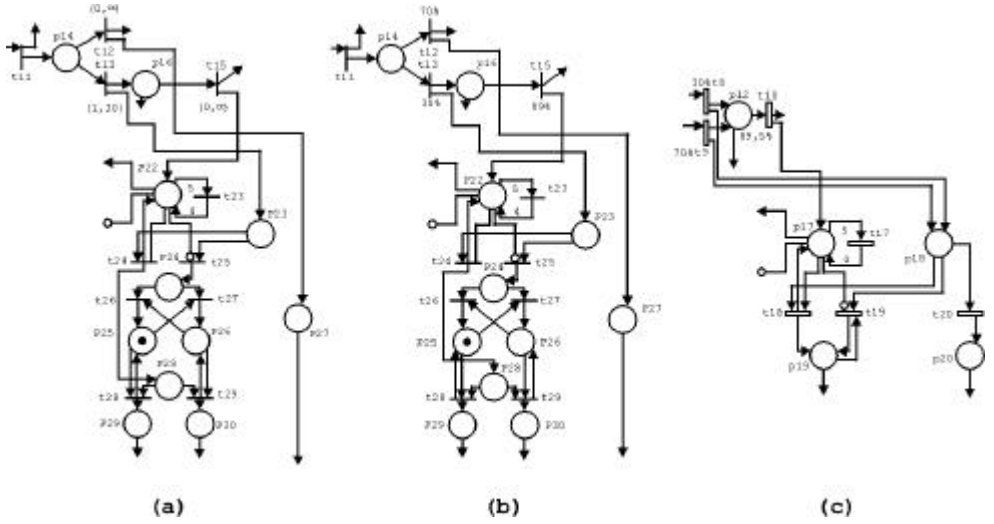

Figura 4.3: Diferença entre os modelos. (a) Previsão usando RdPBT, (b) Previsão usando RdPEG e (c) Previsão usando RdPH.

Tempo está sendo usada é fácil e prático. Basta informar o tempo mínimo de espera antes do disparo e o tempo máximo que a transição deve permanecer habilitada. O exemplo da Figura 4.4 mostra como isso é feito.

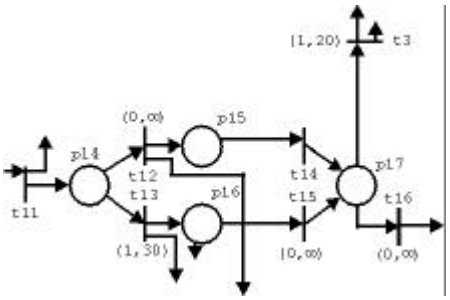

Figura 4.4: Tratamento de Concorrência utilizando tempo.

Já na questão do não-determinismo fica muito mais difícil ou quase impossível tratar do componente tempo. Porém, para tratar da concorrência entre transições cujas as probabilidades de disparo de cada uma delas são conhecidas fica extremamente simples. Basta informar as probabilidades de cada transição envolvida no disparo, como mostra a Figura 4.5.

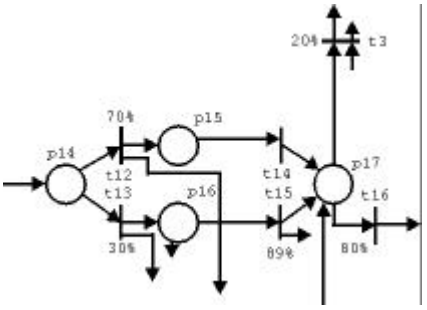

Figura 4.5: Tratamento de Concorrência utilizando probabilidade.

4.3 Comparações Quantitativas

4.3.1 Introdução

Além das comparações qualitativas sobre os modelos apresentados no capítulo 3 é preciso examinar que resultados esses modelos gerariam quando simulados. Para tanto desenvolveu-se um simulador específico para as características da rede híbrida e, a partir dele, verificou-se como os três modelos se comportam em termos de tempos de execução no *pipeline* e taxas de buscas em *cache* e acertos de desvios.

4.3.2 Descrição do Simulador

Para a simulação dos modelos apresentados neste trabalho foi desenvolvido um simulador em linguagem C. Esse simulador tem a capacidade de trabalhar com qualquer modelo que contenha as características das técnicas de redes de Petri apresentadas (tempo, probabilidade e cor), tanto de forma isolada como em combinações delas.

Para a execução de uma simulação, é necessário um arquivo que contenha a descrição de todo o modelo (lugares de entrada e saída, peso dos arcos, tempo de habilitação, probabilidade de disparo etc.) para que o programa leia essas informações e faça as simulações de acordo com as características lidas.

Depois de lidas as matrizes, simula-se a execução da rede baseado na

quantidade de iterações (disparos). Por fim produz-se um arquivo com todas as informações (tempo e quantidade de disparos de cada uma das transições da rede) necessárias para a análise sobre os resultados obtidos.

Como se trata de um protótipo, esse simulador não possui uma interface gráfica, sendo desenvolvido apenas com a finalidade de comparar, através de resultados estatísticos, a eficiência das redes de Petri propostas na atividade de modelar convenientemente um sistema digital, em particular o *pipeline* descrito no capítulo anterior.

O simulador, conforme descrito acima, está dividido em três partes fundamentais: a entrada das matrizes, a simulação propriamente dita e a geração de outras matrizes na saída. Será feita uma descrição mais detalhada a seguir de cada uma dessas partes.

A primeira parte é composta por um arquivo que contém 7 matrizes. A primeira contém informações sobre os lugares de entrada de cada transição. A segunda contém informações sobre os lugares de saída de cada transição. A terceira traz informações sobre as regras de disparos baseadas ou não em cores. A quarta matriz informa a quantidade de marcas e a que lugares essas marcas pertencem. A quinta contém apenas o tempo de criação de cada marca que faz parte da rede. A sexta contém os intervalos de habilitação de cada transição e, por fim, a sétima matriz contém as probabilidades de disparos das transições concorrentes.

A segunda parte é o simulador que está dividido em 8 funções, sem levar em conta as funções de leitura das matrizes de entrada e escrita dos valores de resposta na matriz de saída. A primeira função é responsável por verificar as transições habilitadas por cor ou, no caso dos modelos que não contém essa característica, pelas habilitadas apenas por número de marcas nos lugares de entrada. A segunda função recebe somente as transições habilitadas na função anterior e verifica as transições habilitadas por tempo, caso seja uma rede que não envolva tempo o simulador identifica e passa para a próxima função. A função seguinte é responsável por verificar a probabilidade de disparo de cada transição habilitada, isso se houver esse parâmetro no modelo simulado. Logo após a verificação das probabilidades, é feito uma verificação se há pelo menos uma transição habilitada, se houver, passa para a próxima

função. Caso não haja transições habilitadas há um incremento do tempo (em modelos que utilizam a característica tempo) ou o arquivo de saída é fechado encerrando a simulação. Caso haja mais de uma transição habilitada, a próxima função sorteia uma delas para disparar. A função seguinte faz um incremento de tempo global da rede. Após o sorteio e o incremento do tempo, as marcas dos lugares de entrada da transição sorteada são consumidas pela função próxima função que é responsável por consumir marcas. Por fim são criadas as marcas nos lugares de saída com a última função responsável por produzir essas marcas.

A terceira e última parte é responsável por salvar em arquivo, em forma de matriz, a quantidade de disparos de cada transição juntamente com os tempos de disparo de cada uma delas.

A Figura 4.6 apresenta o diagrama de blocos do simulador desenvolvido.

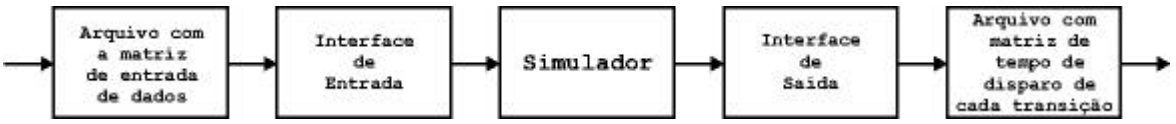

Figura 4.6: Diagrama de blocos do simulador.

4.3.3 Resultados

Para comparar quantitativamente os três modelos foram adotadas as seguintes medidas: tempo médio para fluxo no *pipeline*, freqüência de trocas de política de previsão de *branch*, freqüência de faltas de *cache*, freqüência de esvaziamento de *pipeline* e intervalo entre disparos de um mesmo estágio. A seguir apresentamos os resultados obtidos para simulações, variando o número de transições disparadas entre 1000 e 50000 disparos.

Tempo de percurso no *pipeline*

O gráfico da Figura 4.7 mostra a diferença entre os três modelos simulados.

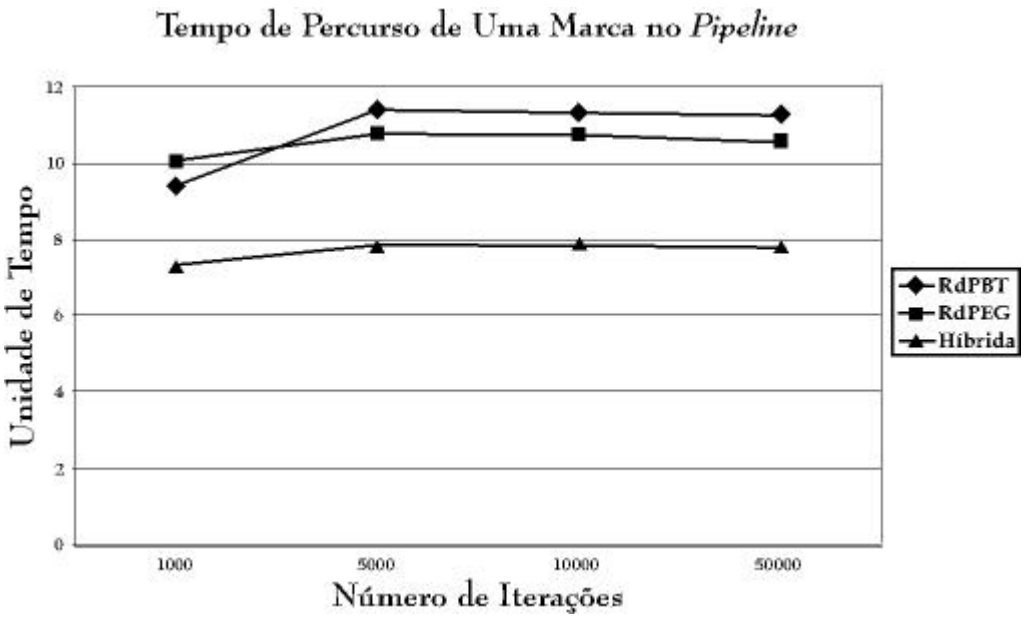

Figura 4.7: Tempo de percurso da marca no *pipeline*.

Como se pode ver, em todos os casos o valor de tempo médio foi constante, independentemente do número de transições disparadas. Em particular, o modelo híbrido apresentou menor valor médio, não muito distante dos modelos RdPEG e RdPBT.

Tempo entre disparos de um mesmo estágio

O gráfico da Figura 4.8 mostra a diferença entre os três modelos simulados.

Com base no gráfico, em todos os casos o valor de tempo médio foi constante, independentemente do número de transições disparadas. Em particular, o modelo híbrido continuou aprensentando menor valor médio, não muito distante dos modelos RdPEG e RdPBT da mesma forma que no gráfico do tempo de percurso no *pipeline*.

Vale observar que o tempo entre disparos de um estágio do *pipeline* não é aproximadamente igual ao tempo de percurso dividido pelo número de estágios. Isso se deve aos atrasos originados pelos eventuais acessos à memória e esvaziamentos de

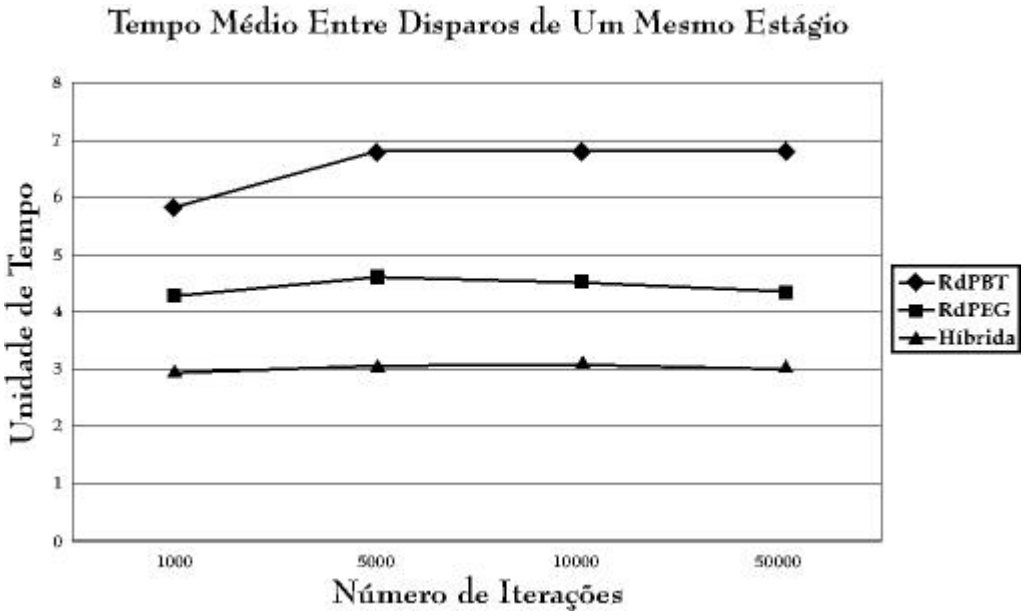

Figura 4.8: Tempo entre disparos de um mesmo estágio.

estágios, que forçam com que a transição equivalente a um dado estágio tenha que se sincronizar ao disparo do estágio seguinte. Já o tempo de percurso de cada instrução não depende do início do percurso no *pipeline* da instrução seguinte. Com relação a essas diferenças pode-se notar que para a RdPH a relação entre tempo de percurso e de estágios ficou em 2,67 (o ideal seria 4,0 pois são quatro estágios), enquanto que para a RdPEG esse valor ficou em 2,44 e para a RdPBT ficou em 1,62. Portanto mais uma vez o melhor desempenho foi apresentado pela RdPH.

Freqüência de buscas de instrução em memória

O gráfico da Figura 4.9 mostra a diferença entre os três modelos simulados.

O valor esperado para a freqüência de busca de instruções em memória (*cache misses*) era de 5%. Os resultados obtidos reafirmam a estabilidade dos modelos. Em particular, o modelo híbrido continuou apresentando o valor médio mais próximo do esperado. Os modelos RdPEG e RdPBT apresentam valores maiores, porém aceitáveis.

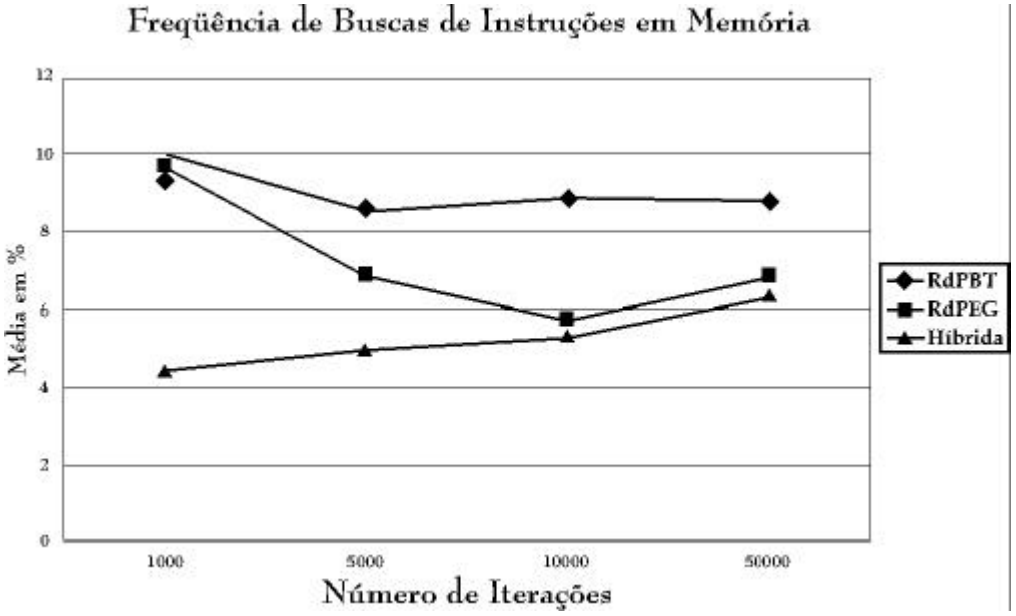

Figura 4.9: Freqüência de buscas em memória (instruções que não estão em *cache*).

Freqüência de buscas de dados em memória

O gráfico da Figura 4.10 mostra a diferença entre os três modelos simulados.

Também aqui se percebe que as redes RDPEG, RdPBT e RDPH apresentam resultados semelhantes e próximos ao esperado, que era de uma taxa de faltas em torno de 20%. Em particular a RDPH apresenta, mais uma vez, resultados melhores do que as demais.

Freqüência de esvaziamento do *pipeline*

O gráfico da Figura 4.11 mostra a diferença entre os três modelos simulados.

Nesse caso se previa uma taxa de falhas (instruções erroneamente carregadas no *pipeline* em função de instruções de desvio) de 11%. Os modelos RdPH e

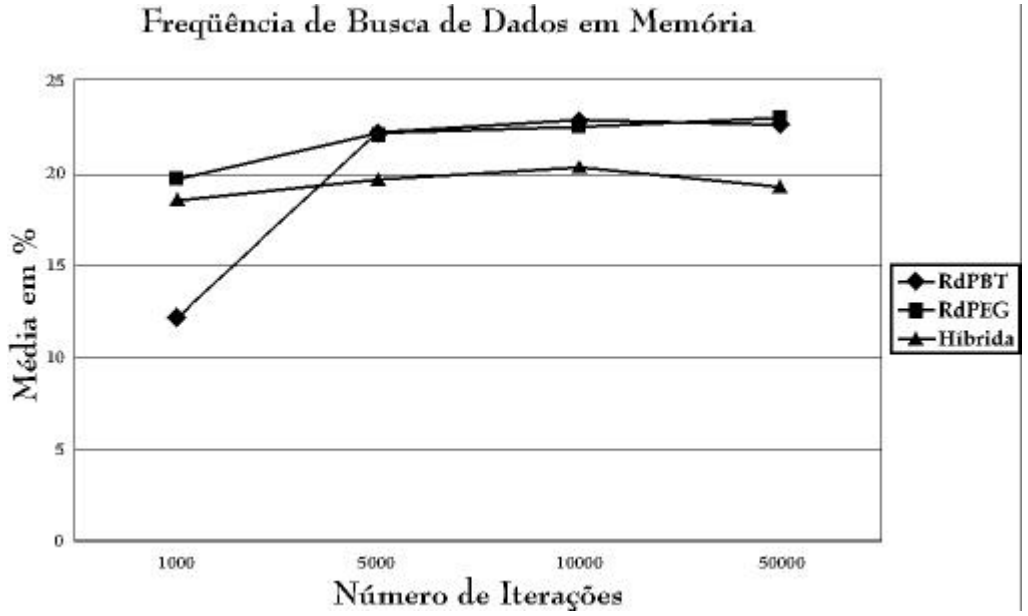

Figura 4.10: Freqüência de buscas em memória (dados que não estão em *cache*).

RdPEG ficaram abaixo desse patamar, sendo que a RDPH teve resultados entre 7 e 9%, o que significa ser o melhor modelo. A RDPEG teve resultados relativamente uniformes (2 a 3%), mas muito abaixo do patamar previsto, enquanto a RDPBT teve resultados aceitáveis variando de 16% a 8%. Esses valores comprovam a qualidade desses modelos.

Freqüência de trocas de abordagem de desvios

O gráfico da Figura 4.12 mostra a diferença entre os três modelos simulados.

Como se pode ver, não há muita variação entre os três modelos apresentados. O gráfico apresenta sinais de estabilidade na porcentagem de troca de abordagem para instruções de desvio, independentemente do número de transições disparadas. Deve-se salientar que os resultados obtidos indicam que o simulador se comporta de modo conservativo, ou seja, considera-se que uma vez adotada uma política teremos a mesma mantida por muitas iterações.

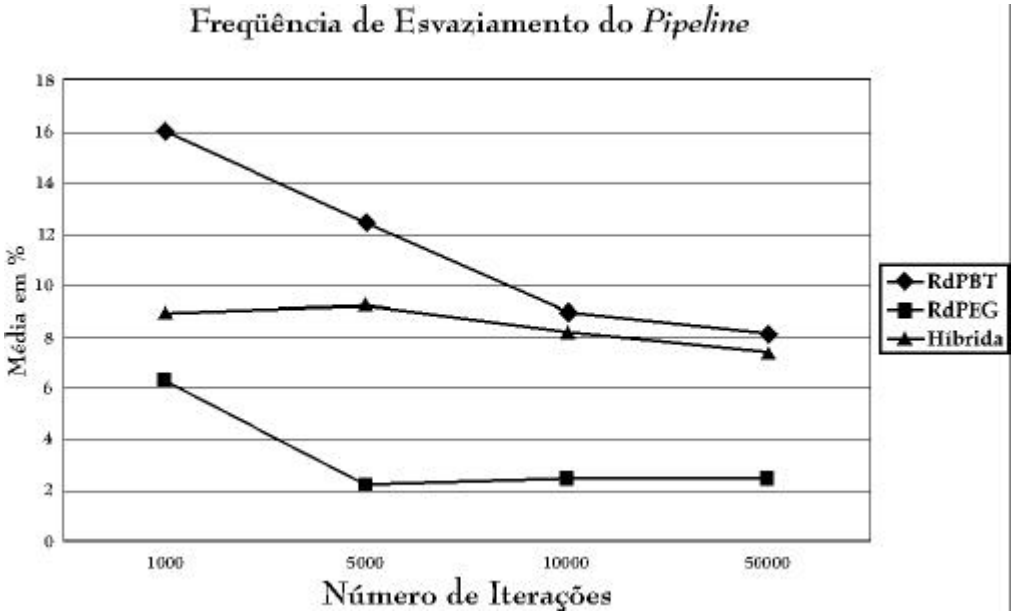

Figura 4.11: Freqüência de esvaziamento do *pipeline*.

4.3.4 Análise dos Resultados

Com base nos resultados apresentados na seção anterior, pode-se dizer que a rede que apresentou resultados mais próximos daqueles esperados para o modelo geral de *pipeline* proposto é a rede de Petri Híbrida. Os resultados obtidos com ela foram mais estáveis com a variação no número de iterações da simulação e, nos casos em que o modelo definia valores pré-determinados de comportamento, foram também mais precisos. Os modelos de RdPEG e RdPBT tiveram resultados razoáveis e aproximadamente equivalentes, tanto do ponto de vista de estabilidade quanto de precisão.

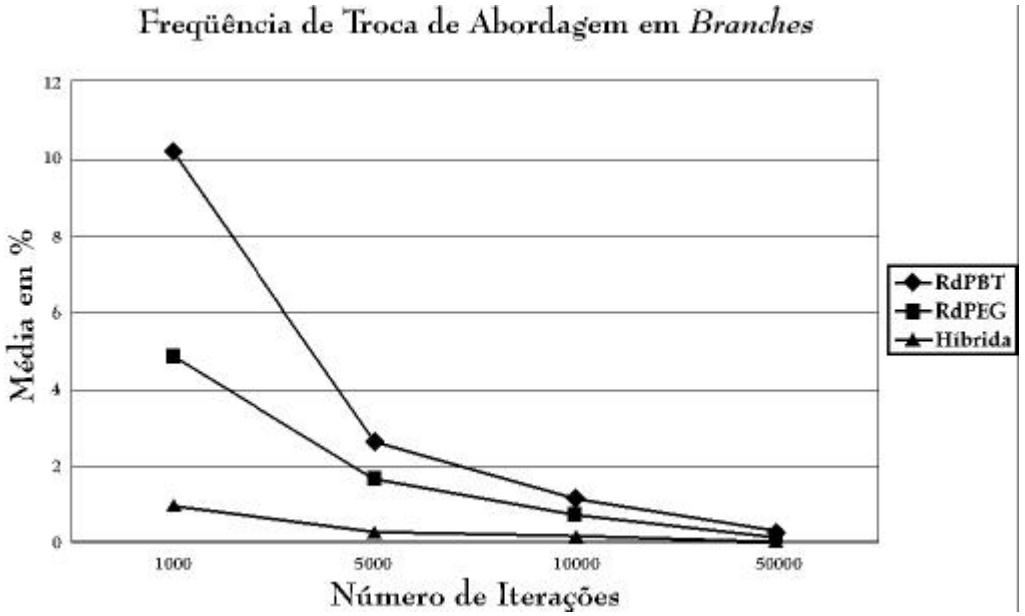

Figura 4.12: Freqüência de trocas de abordagem de *branch*.

Capítulo 5

Conclusão

Redes de Petri permitem a modelagem de muitos sistemas, incluindo sistemas concorrentes e distribuídos. Para promover bons meios de descrever aspectos temporais, várias extensões de redes de Petri têm sido propostas através da atribuição de manipulações sobre o tempo às transições, lugares e/ou arcos. Tais extensões têm aplicação em sistemas digitais, que necessitam de operadores capazes de tratar o tempo em seus modelos.

Nesse trabalho apresentou-se como algumas técnicas de redes de Petri, em especial Redes de Petri Baseada em Tempo, Rede de Petri Estocástica Generalizada e Rede de Petri Colorida podem ser utilizadas para modelar um sistema digital, em particular um *pipeline*.

Os modelos criados através dessas técnicas apresentaram dificuldades distintas, sendo que os modelos mais simples foram obtidos com técnicas mais complexas, que misturam elementos das redes mais simples. Isso ocorreu em particular com a rede híbrida, que ao compor a forma de tratamento de tempo da RdPBT, o tratamento de probabilidades da RdPEG e a capacidade de caracterização das marcas das redes coloridas, permitiu que o modelo do *pipeline* se tornasse mais compacto e funcional do que os modelos das demais técnicas.

5.1 Síntese dos Resultados

Os resultados apresentados no capítulo 4 apenas confirmaram aquilo que já se poderia esperar. Primeiramente ficou evidente a maior capacidade de produzir modelos elegantes (menos lugares e transições) das redes coloridas (aplicadas na construção das RdPH). O modelo criado com a RdPH não apenas é mais simples como também permite a combinação das melhores características dos modelos baseados em RdPBT e RdPEG.

As RdPEG e RdPBT apresentaram resultados razoáveis, mostrando-se alternativas interessantes quando as características do sistema estiverem mais fortemente relacionadas apenas com eventos probabilísticos ou de tempo respectivamente.

Finalmente, a RdPH produziu resultados significativamente melhores que as demais, principalmente no que diz respeito à precisão dos valores obtidos na simulação. Isso, aliado ao que já se disse sobre a facilidade de construção do modelo, indica que essa é uma técnica bastante atraente na modelagem de um sistema digital.

5.2 Perspectivas Futuras

Como indicado no capítulo 2, existem muitas extensões diferentes de redes de Petri que tratam do parâmetro tempo. O escopo desse trabalho se restringiu a modelos bastante básicos dessas extensões, uma vez que se trata apenas de uma investigação sobre a viabilidade da aplicação de redes de Petri na modelagem de sistemas digitais. De todos os resultados obtidos no desenvolvimento desse trabalho é possível identificar alguns aspectos que merecem uma melhor ou maior investigação. São eles:

- Ampliar o escopo de comparações, incluindo outras técnicas de manipulação de tempo em redes de Petri. Isso, é claro, permitiria um entendimento mais amplo sobre que modelos seriam efetivamente convenientes no projeto de um sistema digital.

- Aplicar as mesmas técnicas em outros componentes de um sistema digital, preferencialmente em componentes que possuam características distintas das de um *pipeline*. Com isso poderiam ser identificadas variações de adequação entre as diferentes técnicas e os sistemas modelados.

- Um outro aspecto a ser abordado é a verificação da qualidade dos resultados obtidos na simulação quando confrontados com os medidos em um sistema real. Embora de extrema importância, não foi possível realizar isso nesse trabalho pela ausência de informações sobre *pipelines* reais que fossem suficientemente simples, uma vez que seria impraticável apresentar aqui um modelo para um *pipeline* de 12 estágios, por exemplo.

- Melhorar a implementação do simulador utilizado. Embora esse simulador seja apenas um subproduto do trabalho de comparação das diferentes técnicas (que se tornou necessário em função da RdPH), a sua melhoria, com inclusão de uma interface gráfica, inclusão de novas técnicas e possibilidade de fazer um ajuste fino em seu modo de execução, permitiria um uso mais adequado do mesmo durante o processo de projeto de um sistema digital.

REFERÊNCIAS BIBLIOGRÁFICAS

[Ajmone, Balbo, Chiola e Conte, 1987] Ajmone Marsan, M., G. Balbo, G. Chiola, and G. Conte (1987). Generalized Stochastic Petri Nets Revisited: Random Switches and Priorities. Em *Proc. of the Fd Internattonai Workshop on Petri Nets and Performance Models*, pages 44-53, IEEE-CS Press.

[Allan, Shah e Reddy, 1995] Allan, V. H., Shah, U. R., Reddy, K. M., (1995). Petri Net Versus Modulo Scheduling for Software Piperline. Em *Proceedings of the 28th International Symposium on Microarchitecture*, pages 105–110.

[Buy e Sloan, 1994] Buy, U. e Sloan, R. H., (1994). Analysis of real-time programs with simple time Petri nets. Em *Comm. of ACM*.

[Cardoso e Valette, 1997] Cardoso, J., Valette R.(1997). *Redes de Petri*. Ed. Da UFSC.

[Evers, 2001] Evers, M, Yeh, T. (2001). Understanding Branches and Designing Branch Predictors for High-Performance Microprocessors. Em *Proceedings of the IEEE*, volume 89, number 11.

[Ghezzi, Mandrioli, Morasca e Pezze, 1991] Ghezzi, C., Mandrioli, D., Morasca, S. e Pezze, M. (1991). A unified high-level Petri net formalism for time-critical systems. Em *IEEE Trans. on Software Engineering*, volume 17, number 2, pages 160-172.

[Granda, Drake e Gregorio, 1992] Gandra, M., D.; Drake, J. M. and Gregorio, J. (1992). Performance evaluation of parallel systems by using unbounded generalized

stochastic petri nets. Em *IEEE Trans. on Software Engineering*, volume 18, number 1, pages 55–71.

[Hayes, 1988] Hayes, J. P. (1988). *Computer Architecture and Organization*. McGraw-Hill, San Francisco, 2ª Ed.

[Hill e Peterson, 1987] Hill, F. J. e Peterson, G. R. (1987). *Digital Systems*. Ed. Wiley.

[Hwang, 1993] Hwang, K. (1993). *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, San Francisco.

[ICATPN, 2000] 21st International Conference on Application and Theory of Petri Nets. Em *Aahus, Denmark*, June 26-30, 2000.

[Jensen, 1996] Jensen, K. (1996). *Coloured Petri Nets: Basic concepts, analysis methods and pratical use*. Ed. Springer, 2nd edition, volume 1.

[Kristensen, Mitchell, Zhang e Billington, 2002] Lars M. Kristensen, Brice Mitchell, Lin Zhang and Jonathan Billington (2002). Modelling and Initial Analysis of Operational Planning Processes Using Coloured Petri Nets. Em *Workshop on Formal Methods Applied to Defence Systems*, Adelaide, volume 12.

[Mano, 1993] Mano, M. M. (1993). *Computer System Architecture*. Los Angeles: Ed. Prentice-Hall International Editions, 3ªEd.

[Mano, 2002] Mano, M. M. (2002). *Digital Design*. Los Angeles: Ed. Prentice-Hall, 3ªEd.

[Merlin e Farber, 1976] Merlin, P.M. and Farber, D.J.(1976). Recoverability of Communications Protocols: Implications of a Theoretical Study. Em *IEEE Trans. On Communications*, volume 24, number 9, pages 1036–1043.

[Milutinovié, Fura e Helbig, 1991] Milutinovié, V. M., Fura, A. F., Helbig, W. A. (1991). Pipeline Design Tradeoffs in a 32-bit Gallium Arsenide Microprocessor. Em *IEEE Transactions on Computers*, volume 40, number 11.

[Molloy, 1982] M.K. Molloy (1982). Performance Analysis using Stochastic Petri Nets. Em *IEEE Transaction on Computers*, volume 31, pages 913–917.

[Monteiro, 2001] Monteiro, M. A. (2001). *Introdução À Organização de Computadores*. Rio de Janeiro: Ed. LTC - Livros Técnicos e Científicos Editora S.A.

[Nissanke, 1997] Nissanke, N. (1997). *Realtime Systems*. Prentice Hall series in computer.

[Petri, 1962] Petri, C.A. (1962). Fundamentals of a theory of asynchtonous information flow. Em *Comm. of ACM*, volume 5, number 6, pages 319–319.

[Rajagopalan e Allan, 1993] Rajagopalan, M. and Allan, V. H. (1993). Efficient Scheduling of Fine Grain Parallelism in Loops. Em *Proceedings of the 26th International Symposium and Workshop on Microarchitecture (MICRO-26)*, pages 2-11.

[Ramirez, Larriba-Pey e Valero, 2001] Ramirez, A., Larriba-Pey, J. L., Valero, M. (2001). Instruction Fetch Architectures and Code Layout Optimizations. Em *Proceedings of the IEEE*, volume 89, number 11.

[Raskin e Mattos, 1994] Raskin, S. F., Mattos, N. P. (1994). *Aspectos da Arquitetura de Processos RISC*. Byte Brasil, São Paulo.

[Smith, 1981] Smith, J. E. (1981). A Study of Branch Prediction Strategies. Em *IEEE Transaction*.

[Smith, 1982] Smith, A. J. (1982) Cache Memories. Em *ACM Computer Surveys*, volume 14, number 13.

[Tanenbaum, 1996] Tanenbaum, A. S. (1996). *Organização Estruturada de Computadores*. Ed. Prentice Hall do Brasil, 4a edição.

[Tew, Manivannan, Sadowski e Seila, 1994] J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, (1994). Concurrent Execution of Timed Petri Nets. Em *Proceedings of the 1994 Winter Simulation Conference*.

[Wang, 1998] Wang, J. (1998). *Timed Petri Nets: Theory and Application*. Kluwer Academic Publishers, USA.

[Wang e Deng, 2000] Wang, J. e Deng, Y. (2000). Reachability analysis of real-time systems using time Petri nets. Em *IEEE Trans. On Systems, Man, and Cybernetics-Parte B:Cybernetics*, volume 30, number 5.

[Zimmermann, Freiheit e Hommel] Zimmermann, A., Freiheit, J. and Hommel G. Discrete Time Stochastic Petri Nets for Modeling and Evaluation of Real-Time Systems. Em *IEEE Transaction.*

Apêndice A

Tabelas para referência rápida

Neste apêndice apresentam-se tabelas contendo descrição das transições de cada modelo demonstrados no capítulo três. O objetivo deste apêndice é funcionar como um guia de referência rápida para o usuário no momento de entender o funcionamento de cada modelo descrito.

Tabela A.1: Descrição de cada transição em uma Rede de Petri Baseada em Tempo.

Transição	Descrição
t1	Se for habilitada e permanecer assim durante o período (2,5) e se a transição t9 não disparar, produz marca em p12 e avisa a memória que os dados na *cache* de instrução devem ser trocados (produz marca em p1 (memória))
t2	Se a memória for avisada que os dados da *cache* de instruções devem ser trocados (p1 com marca), a *cache* estiver vazia (p3 sem marca) e o tempo de criação da marca em p1 for maior que 1, coloca os novos dados na *cache* (produz marca em p3)
t3	Se for habilitada e permanecer assim durante o período (1,20) e se a transição t17 não disparar, produz marca em p18 e avisa a memória que os dados na *cache* de dados devem ser trocados (produz marca em p2 (memória))
t4	Se a memória for avisada que os dados da *cache* de dados devem ser trocados (p2 com marca), a *cache* tiver vazia (p4 sem marca)e o tempo de criação da marca em p1 for maior que 1, coloca os dados novos na *cache* (produz marca em p4)
t5	Se o estágio estiver limpo (p5 com marca), se o endereço da instrução que deve ser buscada estiver disponível (p9 com marca) e se a instrução que acaba de ser decodificada não for desvio (p27 com marca), passa o endereço da instrução a ser buscada (produz marca em p10)
t6	Se o estágio estiver limpo (p5 com marca), se o endereço da instrução que deve ser buscada estiver disponível (p9 com marca) e se a instrução que acaba de ser decodificada for desvio e as últimas previsões estiveram erradas (p30 com marca), passa o endereço da instrução a ser buscada (produz marca em p10)

continua na próxima página

Tabela A.1: Descrição de cada transição. (continuação)

Transição	Descrição
t7	Se o estágio estiver limpo (p5 com marca), se o endereço da instrução a ser buscada estiver disponível (p9 com marca) e se a instrução que acaba de ser decodificada for desvio e as últimas previsões estiveram corretas (p29 com marca), passa o endereço da instrução a ser buscada (produz marca em p10)
t8	Se não ocorrer um erro de previsão (disparo de t19), produz marca em p11
t9	Se for habilitada e se a transição t1 não disparar, produz marca em p12 (sinal de que os dados da *cache* estavam corretos)
t10	Se o próximo estágio está limpo (p6 com marca), busca a instrução a ser decodificada (produz marca em p13) e libera o estágio (produz marca em p5)
t11	Se o estágio seguinte estiver livre (p7 com marca), produz marca em p14 (instrução foi decodificada) e em p6 (estágio de decodificação liberado)
t12	Se for habilitada e permanecer assim durante o período $(0,\infty)$, avisa que a instrução não é desvio (produz marca em p27) e aguarda o tempo do estágio (produz marca em p15)
t13	Se for habilitada e permanecer assim durante o período (1,30), avisa que a instrução é desvio (produz marca em p23) e aguarda um possível erro de previsão e conseqüente limpeza do estágio ou não (produz marca em p16)
t14	Se for habilitada, produz marca em p17.
t15	Se for habilitada sem que a transição t19 (erro de previsão) dispare, produz marca em p17
t16	Se for habilitada e permanecer assim durante o período $(0,\infty)$ e se a transição t3 não disparar, produz marca em p18 (sinal de que os dados da *cache* estavam corretos)

continua na próxima página

Tabela A.1: Descrição de cada transição. (continuação)

Transição	Descrição
t17	Se o próximo estágio está limpo (p8 com marca), executa a instrução (produz marca em p19) e libera o estágio (produz marca em p7)
t18	Se for habilitada (p19 com marca), produz marca em p8 (estágio de gravação liberado)
t19	Se for habilitada e permanecer assim até o período (1,11) sem que a transição t15 dispare (acerto de previsão), produz marca em p20
t20	Se o lugar p21 contiver duas marcas, consome as duas marcas e produz apenas uma (mantém apenas uma marca em p21)
t21	Se contiver acerto(s) na(s) última(s) previsão(ões) (p22 com marca), porém errou a última previsão (p20 com marca), diminui o número de acertos de previsões (diminui número de marcas em p22)
t22	Se não contiver acerto(s) na(s) última(s) previsão(ões) (p22 sem marca), apenas produz uma marca em p21
t23	Se o número de acertos nas previsões for maior que 4 (marcas em p22 ultrapassar 4), a transição dispara e consome uma marca
t24	Se contiver acerto(s) na(s) última(s) previsão(ões) (p22 com marca) e a instrução decodificada for desvio (p23 com marca), produz marca em p28 (continua com a mesma sugestão de previsão)
t25	Se não contiver acerto(s) na(s) última(s) previsão(ões) (p22 sem marca) e a instrução decodificada for desvio (p23 com marca), produz marca em p24 (habilita a troca da última sugestão de previsão)
t26	Se a(s) última(s) previsão(ões) estiveram erradas (p24 com marca), troca de opção (consome marca do lugar p26 e produz no p25)
t27	Se a(s) última(s) previsão(ões) estiveram erradas (p24 com marca), troca de opção (consome marca do lugar p25 e produz no p26)

continua na próxima página

Tabela A.1: Descrição de cada transição. (continuação)

Transição	Descrição
t28	Se tem acertado na(s) última(s) previsão(ões) (p28 com marca) e p25 estiver com marca (escolha a próxima instruçao após o desvio), avisa para que o endereço da próxima instrução deve ser o seguinte ao do desvio(produz marca em p29)
t29	Se tem acertado na(s) última(s) previsão(ões) (p28 com marca) e p26 estiver com marca (escolha a instruçao que o desvio indica em caso de resultado falso), avisa para que o endereço da próxima instrução deve ser do desvio(produz marca em p30)

Tabela A.2: Descrição de cada transição em uma Rede de Petri Estocástica Generalizada.

Transição	Descrição
t1	Se for habilitada, tem 5% de probabilidade de disparar se a transição t9 não disparar, produz marca em p12 e avisa a memória que os dados na *cache* de instrução devem ser trocados (produz marca em p1 (memória))
t2	Se a memória for avisada que os dados da *cache* de instruções devem ser trocados (p1 com marca) e a *cache* estiver vazia (p3 sem marca) coloca os novos dados na *cache* (produz marca em p3)
t3	Se for habilitada, tem 20% de probabilidade de disparar se a transição t17 não disparar, produz marca em p18 e avisa a memória que os dados na *cache* de dados devem ser trocados (produz marca em p2 (memória))
t4	Se a memória for avisada que os dados da *cache* de dados devem ser trocados (p2 com marca) e a *cache* tiver vazia (p4 sem marca), coloca os dados novos na *cache* (produz marca em p4)

continua na próxima página

Tabela A.2: Descrição de cada transição. (continuação)

Transição	Descrição
t5	Se o estágio estiver limpo (p5 com marca), se o endereço da instrução que deve ser buscada estiver disponível (p9 com marca) e se a instrução que acaba de ser decodificada não for desvio (p27 com marca), passa o endereço da instrução a ser buscada (produz marca em p10)
t6	Se o estágio estiver limpo (p5 com marca), se o endereço da instrução que deve ser buscada estiver disponível (p9 com marca) e se a instrução que acaba de ser decodificada for desvio e as últimas previsões estiveram erradas (p30 com marca), passa o endereço da instrução a ser buscada (produz marca em p10)
t7	Se o estágio estiver limpo (p5 com marca), se o endereço da instrução a ser buscada estiver disponível (p9 com marca) e se a instrução que acaba de ser decodificada for desvio e as últimas previsões estiveram corretas (p29 com marca), passa o endereço da instrução a ser buscada (produz marca em p10)
t8	Se não ocorrer um erro de previsão (disparo de t19), produz marca em p11
t9	Se for habilitada e se a transição t1 não disparar, produz marca em p12 (sinal de que os dados da *cache* estavam corretos)
t10	Se o próximo estágio está limpo (p6 com marca), busca a instrução a ser decodificada (produz marca em p13) e libera o estágio (produz marca em p5)
t11	Se o estágio seguinte estiver livre (p7 com marca), produz marca em p14 (instrução foi decodificada) e em p6 (estágio de decodificação liberado)
t12	Se for habilitada, tem 70% de probabilidade de disparar e, se disparar, avisa que a instrução não é desvio (produz marca em p27) e aguarda o tempo do estágio (produz marca em p15)

continua na próxima página

Tabela A.2: Descrição de cada transição. (continuação)

Transição	Descrição
t13	Se for habilitada, tem 30% de probabilidade de disparar e, se disparar, avisa que a instrução é desvio (produz marca em p23) e aguarda um possível erro de previsão e conseqüente limpeza do estágio ou não (produz marca em p16)
t14	Se for habilitada, produz marca em p17.
t15	Se for habilitada sem que a transição t19 (erro de previsão) dispare, produz marca em p17
t16	Se for habilitada, tem 80% de probabilidade de disparar e, se a transição t3 não disparar, produz marca em p18 (sinal de que os dados da *cache* estavam corretos)
t17	Se o próximo estágio está limpo (p8 com marca), executa a instrução (produz marca em p19) e libera o estágio (produz marca em p7)
t18	Se for habilitada (p19 com marca), produz marca em p8 (estágio de gravação liberado)
t19	Se for habilitada, tem 11% de probabilidade de disparar e, se a transição t15 não disparar (acerto de previsão), produz marca em p20
t20	Se o lugar p21 contiver duas marcas, consome as duas marcas e produz apenas uma (mantém apenas uma marca em p21)
t21	Se contiver acerto(s) na(s) última(s) previsão(ões) (p22 com marca), porém errou a última previsão (p20 com marca), diminui o número de acertos de previsões (diminui número de marcas em p22)
t22	Se não contiver acerto(s) na(s) última(s) previsão(ões) (p22 sem marca), apenas produz uma marca em p21
t23	Se o número de acertos nas previsões for maior que 4 (marcas em p22 ultrapassar 4), a transição dispara e consome uma marca

continua na próxima página

Tabela A.2: Descrição de cada transição. (continuação)

Transição	Descrição
t24	Se contiver acerto(s) na(s) última(s) previsão(ões) (p22 com marca) e a instrução decodificada for desvio (p23 com marca), produz marca em p28 (continua com a mesma sugestão de previsão)
t25	Se não contiver acerto(s) na(s) última(s) previsão(ões) (p22 sem marca) e a instrução decodificada for desvio (p23 com marca), produz marca em p24 (habilita a troca da última sugestão de previsão)
t26	Se a(s) última(s) previsão(ões) estiveram erradas (p24 com marca), troca de opção (consome marca do lugar p26 e produz no p25)
t27	Se a(s) última(s) previsão(ões) estiveram erradas (p24 com marca), troca de opção (consome marca do lugar p25 e produz no p26)
t28	Se tem acertado na(s) última(s) previsão(ões) (p28 com marca) e p25 estiver com marca (escolha a próxima instruçao após o desvio), avisa para que o endereço da próxima instrução deve ser o seguinte ao do desvio(produz marca em p29)
t29	Se tem acertado na(s) última(s) previsão(ões) (p28 com marca) e p26 estiver com marca (escolha a instruçao que o desvio indica em caso de resultado falso), avisa para que o endereço da próxima instrução deve ser do desvio(produz marca em p30)

Tabela A.3: Descrição de cada transição em uma Rede de Petri Híbrida.

Transição	Descrição

Transição	Descrição
t1	Se habilitada e o sorteio da probabilidade habilitá-la, produz marca verde em p10 e avisa a memória que os dados na *cache* devem ser trocados (produz marca em p1 (memória))

continua na próxima página

Tabela A.3: Descrição de cada transição. (continuação)

Transição	Descrição
t2	Se a memória for avisada que os dados da *cache* de instruções devem ser trocados (p1 com marca) e a *cache* estiver vazia (p3 sem marca) coloca os novos dados na *cache* (produz marca em p3), mas precisa respeitar o tempo habilitação para poder disparar (1,∞)
t3	Se habilitada (marca vermelha ou azul em p13) e o sorteio da probabilidade habilitá-la, avisa a memória que os dados na *cache* de dados devem ser trocados (produz marca em p2 e produz marca verde em p13 se a marca era vermelha e violeta se era azul)
t4	Se a memória for avisada que os dados da *cache* de dados devem ser trocados (p2 com marca) e a *cache* tiver vazia (p4 sem marca) coloca os dados novos na *cache* (produz marca em p4), mas precisa respeitar o tempo habilitação para poder disparar (1,∞)
t5	Se o estágio estiver limpo (p5 com marca), se o endereço da instrução que deve ser buscada estiver disponível (p9 com marca) e a instrução que acaba de ser decodificada não for previsão (p20 com marca), passa o endereço da instrução a ser buscada (produz marca em p10)
t6	Se o estágio estiver limpo (p5 com marca), se o endereço da instrução que deve ser buscada estiver disponível (p9 com marca) e a instrução que acaba de ser decodificada for previsão (p19 com marca: se for marca da cor azul, continua usando a última previsão, se a marca for vermelha, troca a previsão), passa o endereço da instrução a ser buscada (produz marca em p10)
t7	Se o estágio está limpo (p6 com marca), se o endereço está disponível (p10 com marca, verde ou preta) e se as instruções disponíveis na *cache* estão corretas (p10 com marca verde ou preta), busca a instrução para ser decodificada (produz marca vermelha em p11 se a marca em p11 for preta e se for verde, produz marca da cor azul)

continua na próxima página

Tabela A.3: Descrição de cada transição. (continuação)

Transição	Descrição
t8	Se a instrução for um desvio e o estágio seguinte estiver livre (p7 com marca), então produz marca azul em p12 e em p18
t9	Se a instrução não for um desvio e o estágio seguinte estiver livre (p7 com marca), então produz marca vermelha em p12 e em p18
t10	Se o resultado da probabilidade for positivo, se for branch (marca azul em p12) produz marca da cor azul em p13, se não for branch produz marca da cor vermelha em p13 e se houve um erro de previsão, produz marca amarela em p13
t11	Se o próximo estágio estiver limpo (p8 com marca), se os dados foram decodificados (marca azul, vermelha ou amarela em p13) e se os dados disponíveis na *cache* estiverem corretos (p4 com marca), executa a instrução (produz marca da mesma cor em p14), se a *cache* foi atualizada (p13 com marca verde ou violeta) e dados estiverem na *cache* (marca em p4), executa a instrução (produz marca da mesma cor em p14)
t12	Se a instrução for executada (p14 com marca), salva os dados na memória (produz marca em p8)
t13	Se o resultado da probabilidade for positivo e o estágio de busca ainda não foi limpo(marca em p12 for diferente de amarela), limpa o estágio de busca do *pipeline* (produz marca amarela em p12 e uma marca comum em p15)
t14	Se o lugar p16 contiver 2 marcas, consome as duas e produz uma (mantém apenas uma marca em p16)
t15	Se contiver acerto(s) na(s) última(s) previsão(ões) (p17 com marca), porém errou a última previsão (p16 com marca), diminui o número de acertos de previsões (diminui número de marcas em p17)
t16	Se não contiver acerto(s) na(s) última(s) previsão(ões) (p17 sem marca), apenas produz uma marca em p16

continua na próxima página

Tabela A.3: Descrição de cada transição. (continuação)

Transição	Descrição
t17	Se o número de acertos nas previsões for maior que 4 (marcas em p17 ultrapassar 4), a transição dispara e consome uma marca
t18	Se a previsão estiver correta (p17 com no mínimo 1 marca) e for um *branch*(p18 estiver com marca da cor azul), avisa (dispara e produz marca azul em p19) que o próximo endereço da instrução deve seguir a última previsão
t19	Se a previsão estiver errada (p17 sem marca) e for um *branch*(p18 estiver com marca azul), avisa (dispara e produz marca vermelha em p19) que o próximo endereço da instrução a ser buscado não deve seguir a última previsão
t20	Se a instrução decodificada no *pipeline* não for *branch*(cor vermelha em p18), avisa (dispara) que a próxima instrução a ser buscada deve ser a seguinte, pois não é um desvio