THIAGO RADICCHI ROQUE

VERIFICAÇÃO DE EQUIVALÊNCIA DE CIRCUITOS COM ACELERAÇÃO POR LARGURA E APRENDIZADO DE CLÁUSULAS DE CONFLITO

Belo Horizonte

27 de dezembro de 2007

THIAGO RADICCHI ROQUE

VERIFICAÇÃO DE EQUIVALÊNCIA DE CIRCUITOS COM ACELERAÇÃO POR LARGURA E APRENDIZADO DE CLÁUSULAS DE CONFLITO

Dissertação apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Belo Horizonte

27 de dezembro de 2007

UNIVERSIDADE FEDERAL DE MINAS GERAIS

FOLHA DE APROVAÇÃO

Verificação de Equivalência de Circuitos com Aceleração por Largura e Aprendizado de Cláusulas de Conflito

THIAGO RADICCHI ROQUE

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Prof. Claudionor José Nunes Coelho Jr. - Orientador
Departamento de Ciência da Computação – ICEx – UFMG

Prof. Diógenes Cecílio da Silva Jr.
Departamento de Engenharia Elétrica – Escola de Engenharia – UFMG

Prof. Antônio Otávio Fernandes
Departamento de Ciência da Computação – ICEx – UFMG

Dr. Luís Humberto Rezende Barbosa
T3 Informática Aplicada Ltda.

Belo Horizonte, 27 de dezembro de 2007.

Resumo

Verificação de equivalência de sistemas digitais é uma técnica amplamente usada atualmente na indústria de fabricação e projeto de circuitos integrados. Ela permite verificar se duas implementações diferentes de uma mesma funcionalidade possuem comportamento idêntico, como por exemplo, uma versão original e sua contraparte otimizada por alguma metodologia conhecida (retiming), ou então a implementação antes e depois de sua síntese. Diversos métodos são utilizados atualmente nessa área com bastante eficiência, envolvendo algoritmos que utilizam BDDs e SAT. No entanto, a crescente complexidade dos sistemas digitais atuais e a dificuldade de se verificar a equivalência decorrente dessa complexidade motivam a busca de novas soluções. Este trabalho propõe uma ferramenta de verificação, baseada nos seguintes pontos: particionamento dos circuitos em largura, um resolvedor SAT paralelo e reaproveitamento de cláusulas de conflito encontradas pelo resolvedor SAT por indução em circuitos sequenciais. Esperamos com estas técnicas resolver problemas maiores e mais rapidamente do que as soluções existentes.

Abstract

Equivalence checking (EQ) is a very common formal verification method used in the semiconductor industry. It makes possible to verify if two different implementation of the same design have the same functional behavior which is very useful to make sure that the design still behaves correctly after optimizations (like retiming) or synthesis. Several known methods are used efficiently, involving techniques like BDDs and SAT. However, the increasing complexity of digital designs turns EQ into a harder problem, motivating the research for new alternatives in this field. This work proposes a equivalence checking tool, based on the following points: design partitioning by width, a parallel SAT solver, and SAT conflict clause learning by applying induction when unrolling sequential circuits. We aim to solve bigger problems faster than the existing solutions in some cases.

Este trabalho é dedicado à Telma Radicchi de Oliveira.

Agradecimentos

Gostaria de começar agradecendo à toda minha família, principalmente à minha mãe Telma e aos meus avós Edson e Zulma. Vocês são o alicerce da minha vida, e se eu cheguei até aqui, em grande parte foi porque vocês me apoiaram. Agradeço também à minha madrinha Belkiss, minha tia Luciana e aos meus primos Lucas, Cândida e Victor, que são como irmãos para mim.

Agradeço à todos os meus amigos, principalmente Rafael, Felipe, João Paulo, Pedro, e André pelo convívio e por todo o companheirismo desses últimos anos. Agradeço também a Juliana, Mônica, Flávio, Marcelo e Ana Laura.

Ao Luciano, Saulo, Taís, Patrícia e Ana Cristina, amigos de longa data.

À todos os amigos que fiz durante a graduação no DCC, o que inclui toda a grad001.

Ao Claudionor, meu orientador, chefe e professor, com quem aprendi muito e que sempre acreditou em mim. Obrigado por todas as oportunidades e pelo reconhecimento.

Ao Otávio, um grande professor, que está sempre de bom humor e disposto a ajudar.

Agradeço também ao Fabiano, Andrea, Luís, Georgia e Edésio. Tem sido ótimo trabalhar com vocês.

Aos amigos que fiz no Lecom: Nacif, Alessandro, Fabrício, Valdeci. Não apenas quem eu citei, mas todos os que conheci e tive a oportunidade de conviver e trabalhar.

À todos os professores e funcionários do DCC, que de uma maneira ou de outra contribuíram para a minha formação.

À todos que não foram citados mas que ajudaram de forma direta ou indireta na realização deste trabalho.

Sumário

Lista de Figuras

Capítulo 1

Introdução

O problema de verificação de circuitos integrados é historicamente complexo. Quantitativamente, é um problema da ordem de 2^n, onde n é o número de elementos do circuito integrado a ser verificado. Este número de elementos cresce obedecendo à Lei de Moore, ou seja, duplica a cada 18 meses. Neste cenário, a complexidade do problema cresce de maneira explosiva, mas levando-se em consideração o avanço nas técnicas de verificação, o esforço de verificação tem crescido super linearmente com o tamanho do projeto do circuito integrado. Uma boa aproximação para o crescimento do esforço de verificação é o fato de que a duplicação do número de portas duplica o trabalho por ciclo de clock, e a complexidade extra, no mínimo, duplica o número de ciclos necessários à obtenção da cobertura desejada [2]. Assim, em uma previsão otimista, pode-se dizer que o tempo de verificação deve dobrar a cada 18 meses.

O mercado de desenvolvimento de circuitos integrados é extremamente competitivo, e não é admissível que o tempo gasto em verificação dobre a cada 18 meses. Nos últimos 20 anos, a verificação tem sido feita em níveis cada vez mais altos de abstração: inicialmente no nível de porta lógica, passando pelo nível de transferência de registradores (RTL - *Register Transfer Level*), chegando hoje no nível de sistema. As técnicas mais usuais são a simulação e a verificação formal.

Na simulação, geram-se diversos conjuntos de entradas para o circuito randomicamente, e simula-se o mesmo a partir de cada um destes conjuntos por vários ciclos, procurando-se por algum estado inválido. Simular todas as possíveis combinações de entradas geralmente é inviável, pois o número de combinações é exponencial em relação ao número de entradas primárias do circuito.

Na verificação formal, define-se uma ou mais proprieadades que se deseja verificar. Em seguida, tenta-se provar matematicamente que as propriedades são satisfeitas ou não pelo circuito. Em alguns casos esta técnica permite encontrar erros que não seriam

detectados pela simulação, ou encontrá-los muito mais rapidamente.

Atualmente, a etapa de verificação é, sem dúvida, o gargalo no ciclo de desenvolvimento de um circuito integrado. Ela é responsável por aproximadamente de 50 a 80 por cento do tempo total de desenvolvimento do circuito integrado [3][4][5]. Mesmo utilizando-se tanto tempo, muitas vezes não é possível atingir o nível de cobertura desejado. Por exemplo, para simular apenas dois minutos de execução de um Pentium IV são necessários 200 bilhões de ciclos [6], considerando-se que este processador tem frequência de funcionamento de 1 GHz. Vários métodos têm sido propostos como alternativas aos métodos tradicionais de simulação [7]. Estes métodos têm maior eficácia se utilizados de maneira combinada. Mas mesmo utilizando diferentes métodos de verificação é impossível garantir a inexistência no circuito integrado.

A combinação dos fatores de aumento de complexidade e diminuição de tempo de desenvolvimento leva a situações indesejáveis tais como a venda de circuitos integrados contendo erros que se manifestam apenas depois da venda. O exemplo clássico é o erro de divisão de ponto flutuante do processador Pentium da Intel [8] o qual utiliza o algoritmo de divisão proposto em [9] que depende dos valores de uma tabela com aproximadamente 2000 entradas. No caso da implementação no processador Pentium, cinco entradas desta tabela foram deixadas em branco. Este erro manifestava-se apenas em combinações muito específicas de divisor e dividendo, causando perda de precisão no quarto e no décimo segundo dígitos do quociente. Apesar de não afetar diretamente a maioria dos usuários, a Intel viu-se obrigada a trocar todos os processadores defeituosos gratuitamente, tendo para tanto um custo aproximado de 475 milhões de dólares.

Atualmente, a maioria dos fabricantes publica periodicamente uma lista de erratas acerca de seus processadores [10] [11]. A Tabela 5.1 apresenta os dados publicados nas listas de errata das principais famílias de processadores da Intel [12][13][14][15]. Em [16][17][18][19][20] podem ser encontradas erratas de outros grandes fabricantes de microprocessadores.

Processador	Corrigido	A ser corrigido	Sem planos de correção	Total
Pentium	98	0	43	141
Pentium II	23	6	66	95
Pentium III	28	2	58	88
Pentium IV	50	8	34	92

Tabela 1.1: Erros em processadores

1.1 Técnicas de Verificação

Atualmente existem diversas metodologias de verificação para tentar encontrar erros em circuitos antes que o mesmo comece a ser produzido em larga escala. As vantagens de se encontrar o erro mais cedo no ciclo de desenvolvimento são diversas, sendo a financeira a mais evidente, como pode ser visto nos exemplos anteriores. Em alguns casos, a tarefa de verificação de um sistema pode levar de 60% a 80% do tempo total gasto no projeto [21].

Uma das técnicas mais utilizadas é a verificação de equivalência [22]. Nela, compara-se duas implementações de um mesmo circuito e tenta-se provar que elas são funcionalmente equivalentes, ou seja, que nenhum erro foi inserido durante uma das diversas transformações que a especificação inicial sofre, como por exemplo: otimizações, síntese, mudanças de engenharia, pipeline, retiming. Exatamente por isso, é uma técnica extensamente utilizada pela indústria, pois durante todo o ciclo de desenvolvimento o circuito sofre diversas transformações até chegar ao produto final. Vale lembrar que esta técnica pode ser aplicada a circuitos distintos, sendo a única restrição que ambos devem ter o mesmo número de saídas primárias e entradas primárias.

As técnicas atuais de verificação de equivalência se utilizam principalmente dos métodos de verificação formal SAT (Satisfabilidade Proposicional) e BDD (*Binary Decision Diagram*). Ambas as tecnologias vêm sendo utilizadas com bastante sucesso, mas apesar disso, a complexidade inerente ao problema e o aumento exponencial do tamanho e da complexidade dos circuitos digitais continuam motivando pesquisas nesta área.

Já existem resolvedores SAT paralelos [23], mas os trabalhos existentes focam basicamente no resolvedor em si, tentando encontrar algoritmos que sejam mais eficientes no caminhamento do circuito ou similaridades estruturais que possam reduzir o problema completo substancialmente. Não vimos trabalhos que foquem no particionamento eficiente do problema ou que realize o aprendizado de cláusulas SAT por indução, pois não focam especificamente no problema de verificação de equivalência de circuitos. Além disso, muitas das técnicas existentes causam falsos positivos ou não escalam bem para determinados tipos de circuitos, ou ainda dependem de determinadas características estruturais para serem eficientes. Vemos portanto diversos pontos que podem ser melhorados levando-se em conta os trabalhos atuais: dependência de similaridades estruturais, falsos positivos e utilização algoritmos seqüenciais onde o paralelismo poderia ser aplicado.

O paralelismo vem se tornando uma opção cada vez mais viável nos dias de hoje, já

que os fabricantes decidiram investir na ampliação do número de unidades funcionais dentro dos processadores, chegando, hoje, a termos quatro processadores dentro de um mesmo chip. O primeiro chip a adotar tal tecnologia foi o Athlon 64 X2 da AMD, vindo logo em seguida os Pentium 4 da Intel e atualmente os Core Duo, Core 2 Duo, e Core 2 Quad que são as novas arquiteturas que tem sua origem no Pentium M. Todas essas mudanças incentivam cada vez mais a utilização de softwares que aproveitem o paralelismo inerente dessas máquinas.

1.2 Escopo

O objetivo principal deste trabalho é desenvolver um sistema capaz de particionar circuitos utilizando heurísticas, de maneira que os particionamentos formem subproblemas que possam ser resolvidos paralelamente por um resolvedor SAT. Este trabalho faz parte de um sistema maior, que foi desenvolvido em conjunto com a aluna de mestrado Márcia C. Marra de Oliveira [24].

Inicialmente, os circuitos são lidos e armazenados numa estrutura interna. Após este passo, inicia-se o particionamento dos mesmos. Existem duas heurísticas disponíveis: por largura e por ciclos (no caso de um circuito sequencial). O particionamento por largura leva em conta quantos bits o circuito possui em suas saída. Pode-se dividi-lo até o nível de se separar um bit de largura por instância do problema. O aprendizado de cláusulas de conflito SAT pode ser aplicado em conjunto ou separadamente com a heurística de largura. Após o particionamento, efetua-se a transformação do circuito em um conjunto de cláusulas CNF (do inglês, *Conjunctive Normal Form*, ou forma normal conjuntiva) [25], sendo que cada conjunto de cláusulas representa um problema individual do ponto de vista do resolvedor SAT, ou seja, uma das partes em que o circuito original foi dividido. Os diversos subproblemas são então enviados para o resolvedor. Para cada um dos subproblemas, o resolvedor vai enviar uma resposta afirmando se uma equivalência foi encontrada, se as implementações não são equivalentes ou se nada pode ser dito e mais passos devem ser executados. No caso de não se ter um resultado conclusivo, as cláusulas de conflito retornadas podem ser utilizadas para se tentar reduzir o problema. Por exemplo, se uma cláusula indica que dois pontos do circuito são iguais, isto quer dizer que o cone de influência dos mesmos pode ser removido do conjunto de cláusulas que formam os problemas, podendo-se retornar o conjunto reduzido para o resolvedor. Atualmente, o resolvedor desenvolvido em [24] leva em conta algumas cláusulas de conflito que julgamos serem importantes:

- igualdade entre duas variáveis

- desigualdade entre duas variáveis
- variável igual a um AND de outras duas
- variável igual a um OR de outras duas
- variável igual a um XOR de outras duas

A arquitetura do sistema prosposto pode ser vista na figura 1.1.

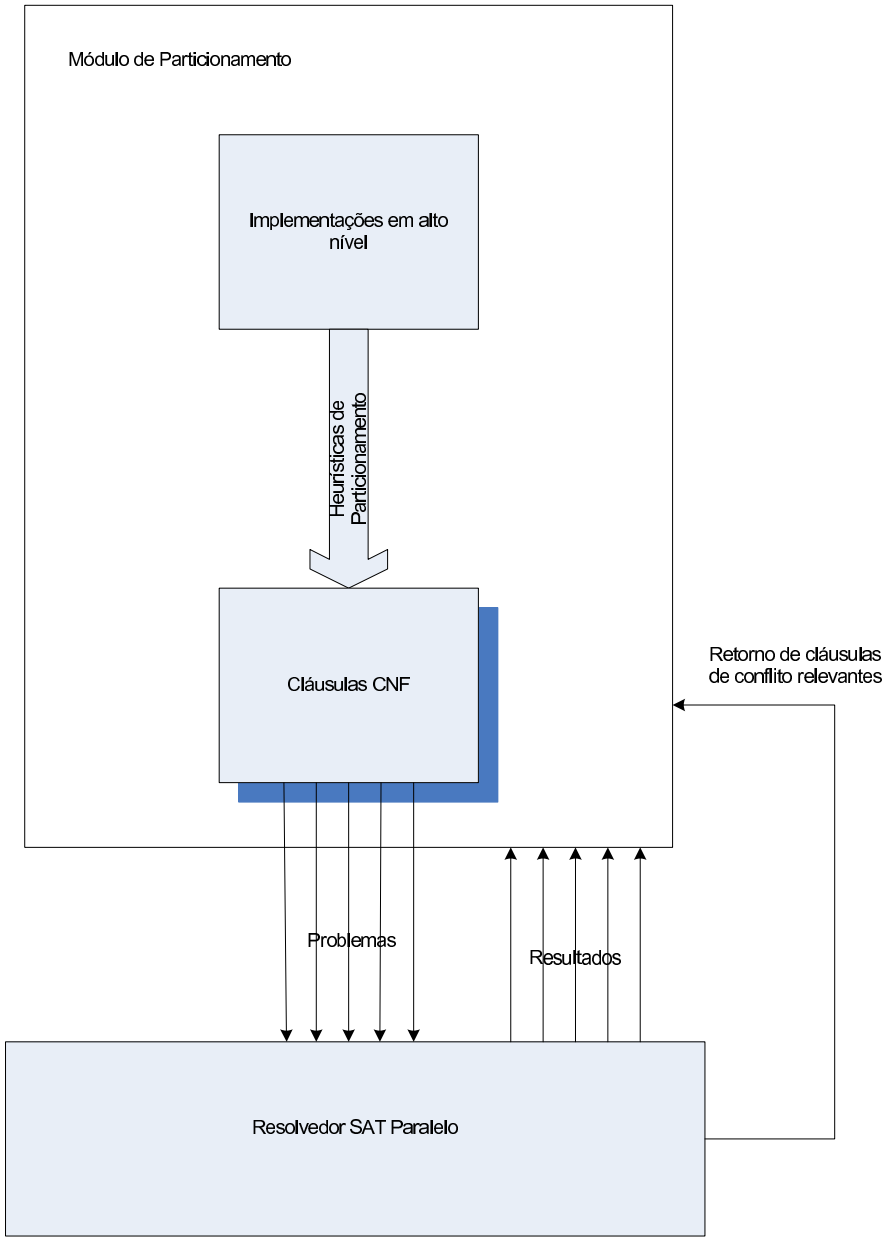

Figura 1.1: Arquitetura do Sistema

O problema a ser resolvido, pode ser visto na figura 1.2: para cada par de saídas primárias a serem comparadas, exemplificadas como PO1 e PO2, gera-se o cone de influência das mesmas (COI1 e COI2, respectivamente). O conjunto das entradas primárias é o mesmo para ambos os cones. As saídas primárias são entradas de uma porta XOR, e o resolvedor tenta provar que a saída do XOR é igual a 1, ou seja, que os cones de lógica de PO1 e PO2 são diferentes. Assumindo um circuito combinacional, caso o resultado seja UNSAT, isso quer dizer que não existe um assinalamento de variáveis onde a saída do XOR seja igual a 1, conseqüentemente $COI1 = COI2$. Se o resultado for SAT, isso significa que existe um assinalamento de variáveis que leva a resultados diferentes entre as saídas primárias, ou seja, $COI1 \neq COI2$. Se os circuitos possuem mais de uma saída primária, repete-se o procedimento para todos os pares de saídas, e para que ambos sejam considerados equivalentes, o resultado de todas as comparações deve ser UNSAT.

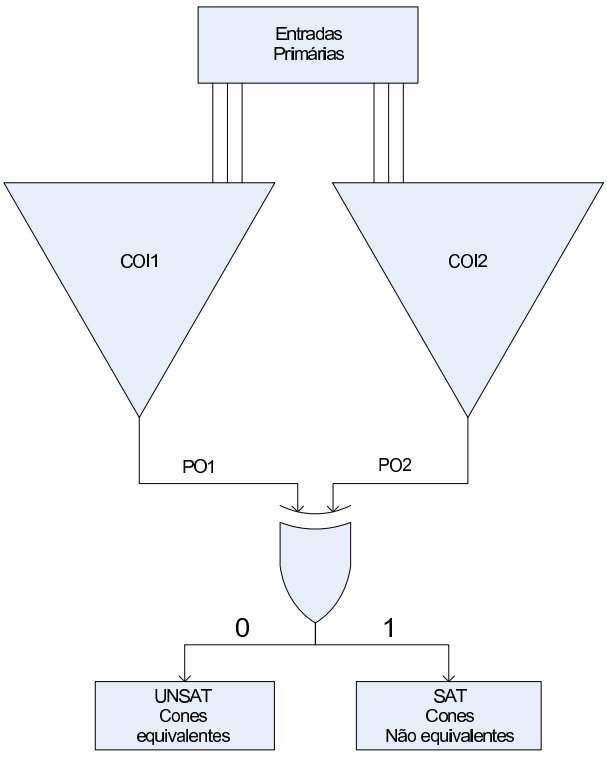

Figura 1.2: Descrição de um Problema

Na comparação de circuitos sequenciais, começamos pelo ciclo zero. Se o resultado da verificação for UNSAT, eles são equivalentes e podemos interromper a verificação. Mas se o resultado for SAT, podemos efetuar o desdobramento do circuito e efetuar a verificação de equivalência no ciclo um, e assim sucessivamente. Neste caso, um

resultado SAT pode signficar que os circuitos são diferentes, ou que precisamos efetuar mais desdobramentos até chegar em um resultado válido. O número máximo de ciclos a serem verificados antes de interromper a verificação é um parâmetro definido pelo usuário.

1.3 Contribuições

Como contribuições deste trabalho temos o desenvolvimento de uma nova metodologia para verificação de equivalência, utilizando-se paralelismo e novas heurísticas de particionamento do circuito, o aprendizado de cláusulas SAT por indução e a criação de uma ferramenta de verificação de equivalência que utiliza todas as técnicas citadas.

1.4 Organização

Este trabalho está dividido como se segue. O segundo capítulo fornece uma revisão bibliográfica sobre as técnicas existentes para verificação de equivalência atualmente. O capítulo 3 apresenta e detalha as heurísticas desenvolvidas para o particionamento dos circuitos a serem comparados. O capítulo 4 aborda a arquitetura da solução proposta como um todo. O capítulo 5 é dedicado aos resultados experimentais e o capítulo 6 apresenta as conclusões e os possíveis trabalhos futuros que podem ser desenvolvidos.

Capítulo 2

Revisão bibliográfica

Este capítulo apresenta uma revisão bibliográfica sobre verificação de equivalência e os principais métodos utilizados nesta área.

2.1 O que é Verificação de Equivalência

O processo de desenvolvimento de um circuito integrado possui diversos fases, partindo-se de uma descrição em uma linguagem de hardware de alto nível até a implementação física. Durante este processo, várias representações do mesmo circuito são geradas. Em cada uma dessas diferentes representações, diversas operações são efetuadas. Todas estas transformações aplicadas ao circuito tornam a tarefa de verificação de equivalência essencial, pois erros podem ser introduzidos em cada uma das operações efetuadas. Esta tarefa, no entanto, tem se tornado cada vez mais árdua, tanto pela crescente complexidade dos circuitos quanto pelas técnicas de transformação utilizadas.

A Figura 2.1 apresenta o fluxo tradicional de projetos de circuitos integrados, apresentando o papel da verificação no mesmo. Inicialmente, o projeto é especificado em RTL, utilizando um linguagem de descrição de hardware de alto nível, como VHDL [26], Verilog HDL [27] ou SystemC [28]. Em seguida, este modelo é sintetizado, ou seja, é gerada uma representação baseada apenas em portas lógicas. Após este passo, diversas técnicas de otimização podem ser aplicadas, para tentar otimizar área, tempo ou consumo de energia. O circuito sintetizado é então mapeado para a tecnologia que será utilizada na implementação física, utilizando-se bibliotecas específicas para tal.

Durante todo este processo de desenvolvimento, pode-se ver que várias transformações são aplicadas à especificação inicial. Portanto, é de extrema importância verificar a equivalência entre diferentes níveis de abstração do circuito, para garantir que erros não foram introduzidos durante essas transformações.

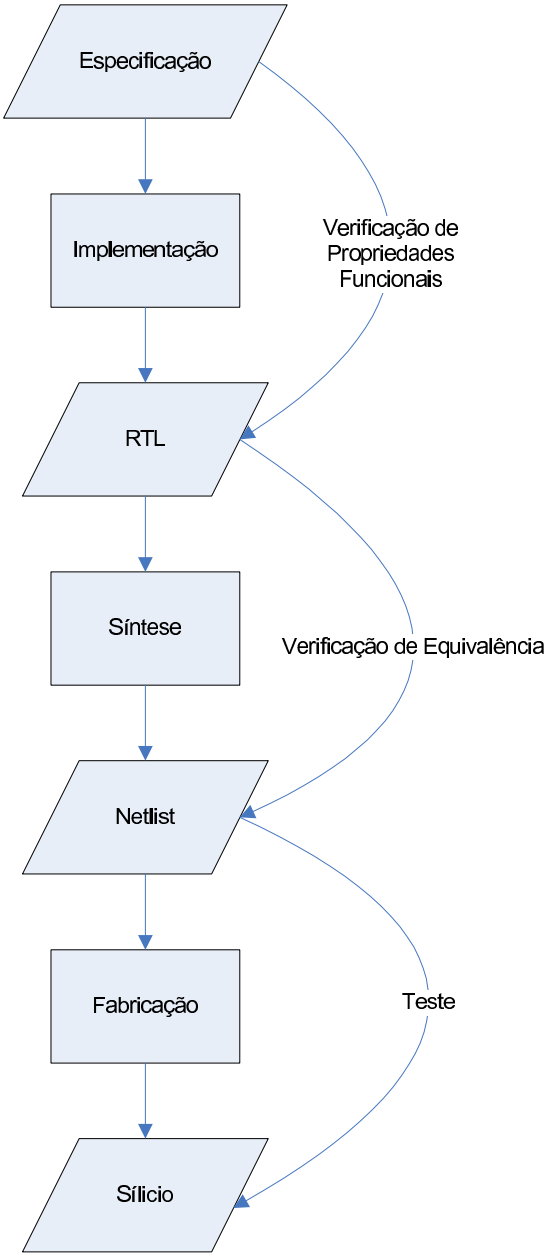

Figura 2.1: Etapas do Projeto de um Circuito Integrado

Um dos métodos existentes de EQ utiliza a técnica de simulação: aplica-se vetores de entrada iguais na descrição RTL e na descrição em portas lógicas, executa-se a simulação por um determinado número de ciclos e compara-se os resultados obtidos na saída. No entanto, esta é uma alternativa incompleta, pois é impossível simular todas as combinações possíveis de entradas em um tempo razoável, já que este é um problema exponencial com relação às entradas. Esta é uma abordagem mais utilizada para testes de regressão.

Como a simulação por si só é incompleta, não garantido a ausência de erros, técnicas alternativas e complementares vem sendo buscadas em outras áreas para suprir essas deficiências da simulação. Os métodos matemáticos tem sido utilizados com grande sucesso nesta área, tanto na verificação de circuitos combinacionais quanto seqüenciais.

A grande vantagem dos métodos matemáticos, é que eles garantem matematicamente a equivalência ou não entre duas descrições. Estes métodos são comumente chamados de Verificação Formal. Em alguns casos, estes métodos podem chegar a um resultado muito mais rapidamente do que seria possível com a simulação: o que demoraria anos para ser simulado pode ser provado com técnicas formais em alguns segundos.

Apesar da verificação de equivalência ser um problema NP-completo [29], muitas instâncias do mundo real podem ser resolvidas rapidamente. Este trabalho busca estender a capacidade da verificação de equivalência através de métodos formais, podendo-se aplica-la a problemas cada vez maiores através do paralelismo.

2.2 Como Funciona a Verificação de Equivalência

Inicialmente, os circuitos são lidos a partir de uma descrição. Em seguida, identifica-se alguns pontos em comum a ambos os circuitos que são essenciais para a realização da verificação. Estes pontos normalmente são as entradas primárias, as saídas primárias, variáveis de estado (latches e flip-flops), e quaisquer outros pontos relevantes dos circuitos analisados. Estes são os elementos importantes para se realizar a verificação de equivalência, e o fluxo tradicional de verificação pode ser visto na figura 2.2. Logo após a identificação dos pontos relevantes, parte-se para a verificação em si, que pode ser realizada através de diferentes técnicas disponíveis, como os métodos formais ou simulação. A tarefa é verificar se ambos os circuitos são equivalentes, ou caso não sejam, informar tal fato ao usuário, se possível com alguma indicação de onde existem discrepâncias.

Normalmente, a verificação da equivalência funcional de dois circuitos é feita com

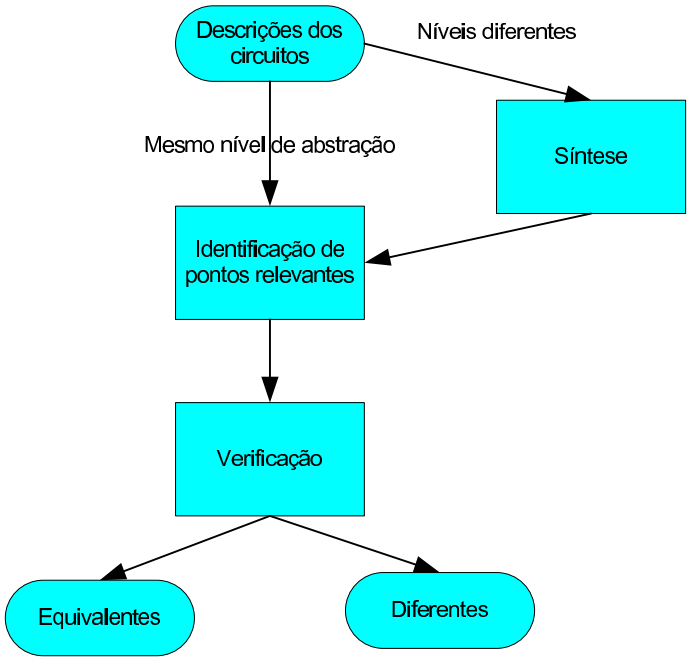

Figura 2.2: Fluxo de Verificação de Equivalência

a construção de suas representações canônicas, mais comumente na forma de BDDs. Sendo assim, os dois circuitos serão equivalentes se e somente se as representações forem isomórficas. Tomando-se os circuitos como máquina de estados finitos ou FSMs(do inglês, *Finite State Machine*), a equivalência ocorre se as FSMs possuírem exatamente os mesmos estados e transições.

A figura 2.3 mostra o modelo típico de verificação de equivalência. Constrói-se uma FSM que é o produto dos dois circuitos a serem verificados. Essa construção, se dá basicamente ligando as entradas primárias das duas máquinas aos mesmos pontos, e nas saídas primárias, faz-se um XOR bit a bit. Se o resultado do XOR for 0 para todas as entradas possíveis, isso signfica que aquele par de bits de saída é exatamente igual, caso contrário são diferentes.

Para o circuito da figura 2.3, suponha que:

- C1 e C2 são os circuitos a serem comparados, que definem uma função lógica sobre as entradas
- E é a entrada utilizada em ambos os circuitos
- S1 e S2 são as saídas de C1 e C2 respectivamente
- R é o mesmo que $S1 \oplus S2$

- o alfabeto que contém todas as entradas possíveis para os circuitos em questão é representado por α

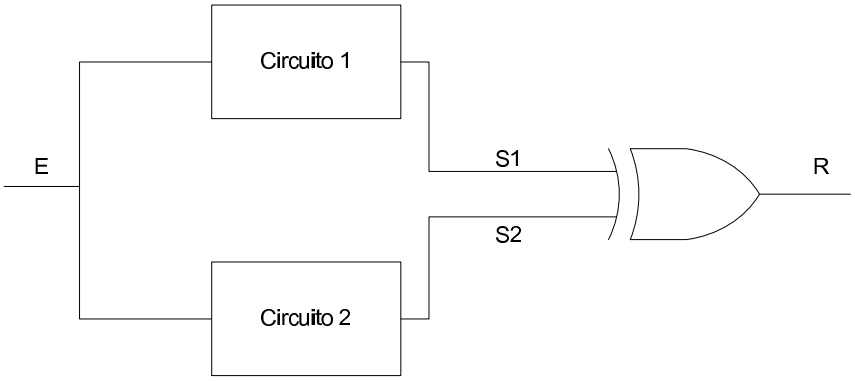

Figura 2.3: Modelo Usado na Verificação

Para que os circuitos sejam equivalentes, as equações a seguir devem ser então satisfeitas:

$$C_1 \equiv C_2 \iff \forall E \in \alpha, R = 0 \tag{2.1}$$

$$C_1 \equiv C_2 \iff \forall E \in \alpha, S1 \oplus S2 = 0 \tag{2.2}$$

$$C_1 \equiv C_2 \iff \forall E \in \alpha, C_1(E) \oplus C_2(E) = 0 \tag{2.3}$$

$$C_1 \equiv C_2 \iff \forall E \in \alpha, C_1(E) \equiv C_2(E) \tag{2.4}$$

2.3 Métodos Formais para Verificação de Equivalência

A verificação de equivalência pode ser aplicada tanto a circuitos combinacionais quanto a circuitos sequenciais. A tarefa em ambos os casos é verificar se os circuitos comparados representam a mesma função booleana.

Basicamente, existem dois grandes grupos de tipos de verificação: métodos simbólicos e métodos estruturais. No caso dos métodos estruturais, a maioria deles se utiliza de BDDs. Com relação aos métodos simbólicos, existem basicamente três alternativas: SAT, ATPG (Automated Test Pattern Generation) e aprendizado recursivo.

Cada uma das abordagens possui suas vantagens. As estruturais, apesar de dependerem de similaridades nos circuitos verificados, geralmente podem ser aplicadas a instâncias de problemas maiores. Já a abordagem simbólica tem sua vantagem em não depender dessas similaridades estruturais, podendo comparar circuitos codificados de maneiras totalmente diferentes.

Nas próximas subseções, serão abordadas algumas destas técnicas comumente utilizadas nas verificação de equivalência.

2.3.1 BDDs

Diagramas de decisão binários, ou BDDs, são uma forma canônica de representação de fórmulas booleanas. Eles são geralmente bem mais compactos do que formais normais tradicionais como a forma normal conjuntiva (CNF) e a forma normal disjuntiva (DNF), além de ser manipulados eficientemente. Dessa maneira, eles se tornaram largamente utilizados para uma variada gama de aplicações em CAD (*Computer Aided Design*), incluindo simulação simbólica, verificação de lógica combinacional e verificação de sistemas concorrentes sequenciais.

Antes de chegarmos efetivamente nos BDDs, vamos considerar primeiro uma árvore de decisão binária e a partir dela chegar ao BDD. Uma árvore de decisão binária é uma árvore direcionada com uma raiz, que possui dois tipos de vértices: terminais e não terminais. Cada vértice não terminal v é representado por uma variável var(v) e possui dois sucessores: low(v), se v for 0 ou high(v), se v for 1. Cada vértice terminal v é representado por value(v), que pode ser 0 ou 1. Uma árvore de decisão binária para um comparador de dois bits da figura 2.4, que também pode ser representado pela fórmula:

$$f(x_1, x_2, y_1, y_2) = (x_1 \iff y_1) \wedge (x_2 \iff y_2) \tag{2.5}$$

Pode ser vista na figura 2.5. Pode-se ver se determinado assinalamento para as entradas torna a fórmula verdadeira ou não caminhando pela árvore a partir da raiz até um dos vértices terminais. Se uma variável v for 0, então o próximo vértice será low(v), caso contrário será high(v). O assinalamento $(x_1 = 1, x_2 = 0, y_1 = 1, y_2 = 1)$, leva a uma folha cujo valor é 0, portanto, a fórmula é falsa para o assinalamento em questão.

Pode-se ver que árvores de decisão binárias não são uma representação concisa para fórmulas booleanas, já que são do mesmo tamanho que uma tabela da verdade. No entanto, há muita redundância nestas árvores. No exemplo dado, existem oito subárvores nomeadas b2, mas apenas três são distintas, ou seja, podemos obter uma

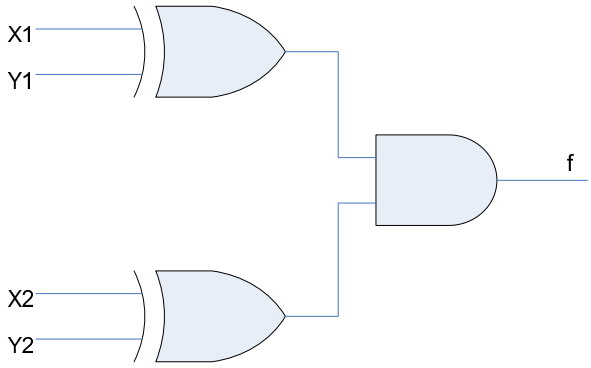

Figura 2.4: Comparador de Dois Bits

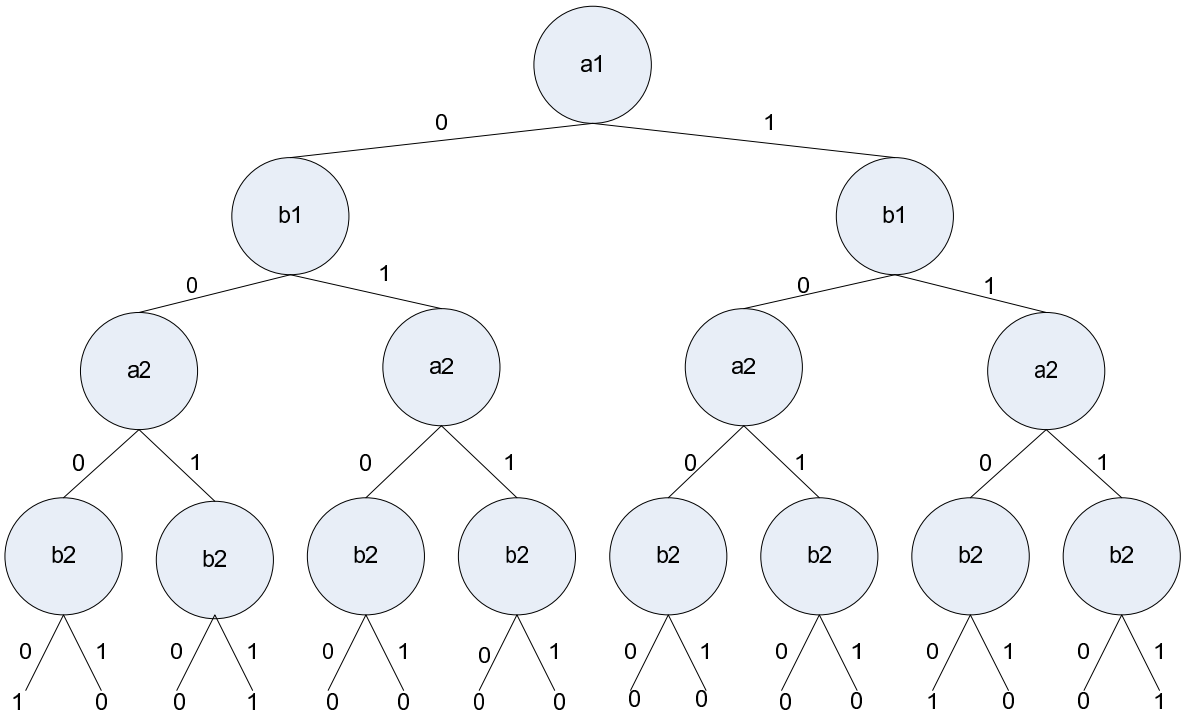

Figura 2.5: Árvore Binária para o Comparador de Dois Bits

representação mais eficiente se unirmos subárvores isomórficas. Este procedimento resulta num gráfico acíclico direto (DAG), ao qual chamamos de diagrama de decisão binário, ou BDD.

Em aplicações práticas, como a verificação de equivalência, é desejável que tenhamos uma representação canônica para as funções booleanas. A representação em questão deve ter a propriedade de que duas funções booleanas são logicamente equivalentes se e somente se elas tiverem representações isomórficas. Esta propriedade facilita a tarefa de verificar a equivalência ou provar se determinada fórmula é satisfazível ou não. Dois BDDs são isomórficos se e somente se existe uma função de mapeamento h, de um para um entre os terminais de um e os terminais de outro e os não-terminais da mesma maneira, de modo que para cada vértice terminal, value(v) = value(h(v)) e para cada não-terminal, var(v) = var(h(v)), h(low(v)) = low(h(v)), e h(high(v)) = high(h(v)).

Foi Bryant [30] quem mostrou como obter uma representação canônica para funções booleanas aplicando duas restrições aos BDDs. Primeiramente, as variáveis devem sempre aparecer na mesma ordem em todos os caminhos, da raiz até a um terminal. Além disso, não devem existir subárvores isomórficas ou vértices redundantes no diagrama. Começando de um BDD que satifaça a propriedade de ordenação de variável, basta eliminar todas as redundâncias no mesmo para se chegar ao que é comumente chamado de OBDD (Ordered Binary Decision Diagram). Este procedimento de eliminar redundâncias no diagrama é linear em relação ao tamanho do diagrama original. Por exemplo, se aplicarmos a ordenação $a_1 < b_1 < a_2 < b_2$ para a função do comparador de dois bits, obteremos o OBDD da figura 2.6.

Usando-se OBDDs como forma canônica para funções booleanas, a tarefa de verificar a equivalência é reduzida a se verificar o isomorfismo entre os BDDs. No entanto, o tamanho de um OBDD depende diretamente da ordem das variáveis. Em geral, é impraticável conseguir uma ordenação ótima para funções booleanas, pois o problema de se verificar se uma ordenação é ótima para uma determinada função booleana é NP-completo. Além disso, algumas funções booleanas tem tamanho exponencial de OBDDs não importa qual a ordenação que seja, usada, como no caso por exemplo, de multiplicadores combinacionais [31, 32].

Diversas heurísticas foram desenvolvidas para tentar encontrar uma boa ordenação de variáveis. Se a fórmula booleana representar um circuito combinacional, heurísticas baseadas em busca em profundidade podem gerar bons resultados [33, 34]. Isto vem do fato de que o tamanho dos OBDDs tende a ser menor se variáveis relacionadas estiverem perto umas das outras na ordenação. Uma outra técnica, chamada de reordenamento dinâmico [35] é utilizada quando nenhuma outra heurística se aplica. Este método

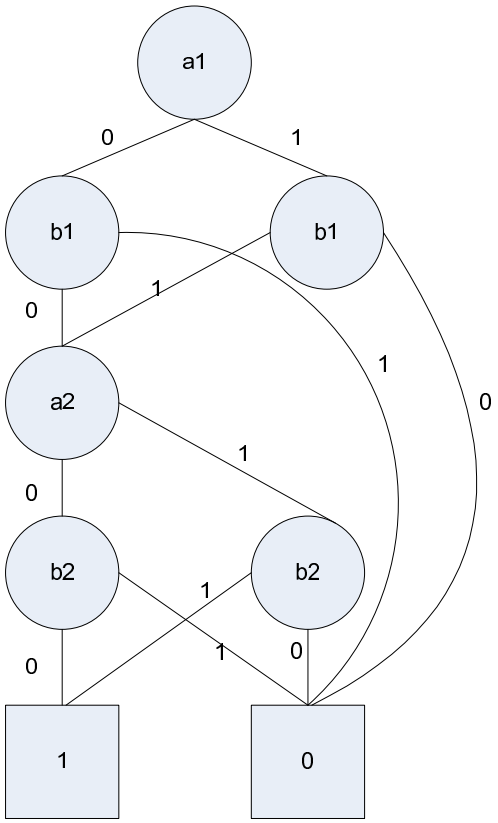

Figura 2.6: OBDD para o Comparador de Dois Bits

procura reordenar as variáveis periodicamente, para tentar reduzir o número total de vértices em uso. Este método é mais utilizado para reduzir o tempo de processamento do que para encontrar uma ordenação ótima.

2.3.2 SAT

Algoritmos SAT são utilizados hoje em dia em diversas áreas da computação. Teve suas origens em programas para prova automática de teoremas [36], mas rapidamente começou a ser utilizado em outras aplicações. Hoje em dia, vem sendo utilizado para geração de testes automáticos[37], síntese lógica, resolução de problemas BCP (do inglês *Binate Covering Problems*)[38, 39, 40, 41]. Outras área que se utilizam de algoritmos SAT são a inteligência artificial [42, 43] e pesquisa operacional [44].

Com relação às aplicações CAD, que é o nosso domínio de interesse, primeiramente, deve-se obter uma descrição inicial do circuito e também da propriedade a ser verificada, descrição esta que deve estar em um formato adequado ao resolvedor SAT. Os resolvedores atuais utilizam em sua maioria o padrão de cláusulas CNF, ou seja, os circuitos e as propriedades devem ser escritos neste formato para que o algoritmo SAT possa resolver o problema.

A fórmula CNF de um circuito combinacional são conjuntos de fórmulas CNF, uma para cada bit de saída do circuito. Na figura 2.7 podemos ver um circuito e na equação 2.6 suas fórmulas CNF correspondentes. O mapeamento de portas lógicas em cláusulas CNF pode ser encontrado em [37]. Com este mapeamento, pode-se descrever qualquer circuito como a união das cláusulas CNF de todas as suas portas lógicas.

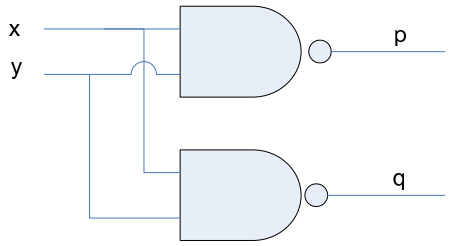

Figura 2.7: Identificador de Literais Equivalentes/Complementares

$$(x+p)(y+p)(\neg x+\neg y+\neg p)(x+q)(y+q)(\neg x+\neg y+\neg q) \tag{2.6}$$

A maioria dos algoritmos de resolução para problemas SAT tem origem no algoritmo DPLL, ou Davis-Putnam-Logemann-Loveland, que sempre tentam otimizar algum aspecto do algoritmo original. O DPLL pode ser definido como um algoritmo

de backtracking para decidir a satisfabilidade de uma fórmula booleana, que pode ser representada em CNF. Este algoritmo foi apresentado em 1962, e é um refinamento do algoritmo Davis-Putnam, que foi desenvolvido em 1960.

O algoritmo básico de backtracking primeiramente escolhe um dos literais da fórmula booleana, assinala um valor para o mesmo, simplifica a fórmula e recursivamente verifica se a fórmula é satisfazível. Se a resposta for sim, então o problema está resolvido. Caso contrário, a mesma verificação recursiva é feita assinalando-se o valor inverso para a variável escolhida inicialmente.

O DPLL introduz basicamente duas melhorias no algoritmo genérico descrito anteriormente. A primeira delas é chamada de propagação unitária: se uma cláusula é unitária, ou seja, contém apenas um literal não assinalado, ela pode ser satisfeita assinalando-se o valor desejado a este literal. Na prática, isto acaba resultando em várias outras cláusulas unitárias, diminuindo o espaço de busca. A outra melhoria é chamada de eliminação de literal puro: se uma variável ocorre apenas com uma polaridade na fórmula, ela é chamada de pura. Literais puros podem sempre ser assinalados de uma maneira que tornem todas as cláusulas que os contêm verdadeiras. Dessa maneira, estas cláusulas podem ser apagadas, deixando de fazer parte da busca. Esta segunda otimização original do DPLL vem sendo omitida nas implementações mais recentes, pois hoje em dia seu efeito pode ser muito pequeno ou até mesmo negativo devido ao overhead de se efetuá-lo.

O algoritmo DPLL depende fortemente da escolha do literal utilizado durante o branching, ou seja, o literal considerado no passo de backtracking. Desse fato resulta que existem uma série de variações do algoritmo, cada uma com uma heurística diferente para a escolha do literal que será assinalado primeiro. A eficiência do algoritmo está diretamente ligada a esta escolha.

Os trabalhos atuais geralmente buscam melhorias em três pontos principais do algoritmo original: definir diferentes políticas para a escolha dos literais, definir novas estruturas de dados para tornar o algoritmo mais rápido, especialmente na parte da propagação de cláusulas unitárias e definindo variantes do algoritmo básico de backtracking. Nesta última melhoria, pode-se incluir backtracking não cronológico e aprendizado de cláusulas.

Na listagem abaixo podemos ver a organização geral do algoritmo DPLL genérico, usado como base para todas as derivações existentes de algoritmos SAT.

```
enquanto(verdadeiro) {
    se(!decide())
        retorne(satisfazível); // todas variaveis assinaladas
```

```
    enquanto(!bcp()) {
        se(!resolveConflito())
            retorne(não satisfazível);
    }
}

bool resolveConflito() {
    d = mais recente decisão não 'tentada por ambos caminhos';
    se( d == NULO) // d não encontrado
        retorne falso;

    inverta valor de d;
    marque d como ambos caminhos tentados;
    desfazer qualquer implicação invalidada;
    retorne verdadeiro;
}
```

O método decide() seleciona uma variável que ainda não foi assinalada e atribui um valor a mesma. Este assinalamento é uma decisão. A cada decisão feita, a mesma é inserida na pilha de decisões. Este método retorna falso se nenhuma variável estiver livre, e verdadeiro caso contrário.

O método bcp() (*Boolean Constant Propagation*), identifica os assinalmentos necessários para satisfazer a fórmula booleana (vale lembrar que todas as cláusulas devem ser satisfeitas para que a fórmula também seja). Este é o passo que busca identificar as cláusulas unitárias do algoritmo DPLL. Se durante a busca um conflito surge, ou seja, no caso de se encontrar que uma variável deve ser 0 e 1 ao mesmo tempo, então o método bcp() retorna falso, e passamos ao método resolveConflito(). Basicamente, o que ele faz é tentar inverter o valor do último nível de decisão tomado. Se ambos os valores já tiverem sido tentados, então o algoritmo volta para um nível de decisão anterior e busca outro assinalamento para continuar.

2.3.3 Geração Automática de Padrões de Teste (ATPG)

Os algoritmos de ATPG são bem próximos aos de SAT. A principal diferença entre eles, é que os métodos baseados em SAT trabalham sobre fórmulas CNF, enquanto os métodos de ATPG utilizam redes booleanas. O método força-bruta para ATPG é bem simples: basta gerar todos as entradas possíveis para um determinado circuito

e simular para tentar encontrar algum erro. Claro que esta abordagem praticamente não é utilizada, pois para um circuito com n bits de entrada teremos 2^n combinações possíveis de entradas de largura n.

Uma outra abordagem mais utilizada é a geração randômica de vetores de entrada, levando-se em consideração um conjunto de probabilidades para cada bit de entrada. Não é uma verificação exaustiva, mas na prática pode-se obter bons resultados com ela. O diagrama típico dessa abordagem pode ser visto na figura 2.8.

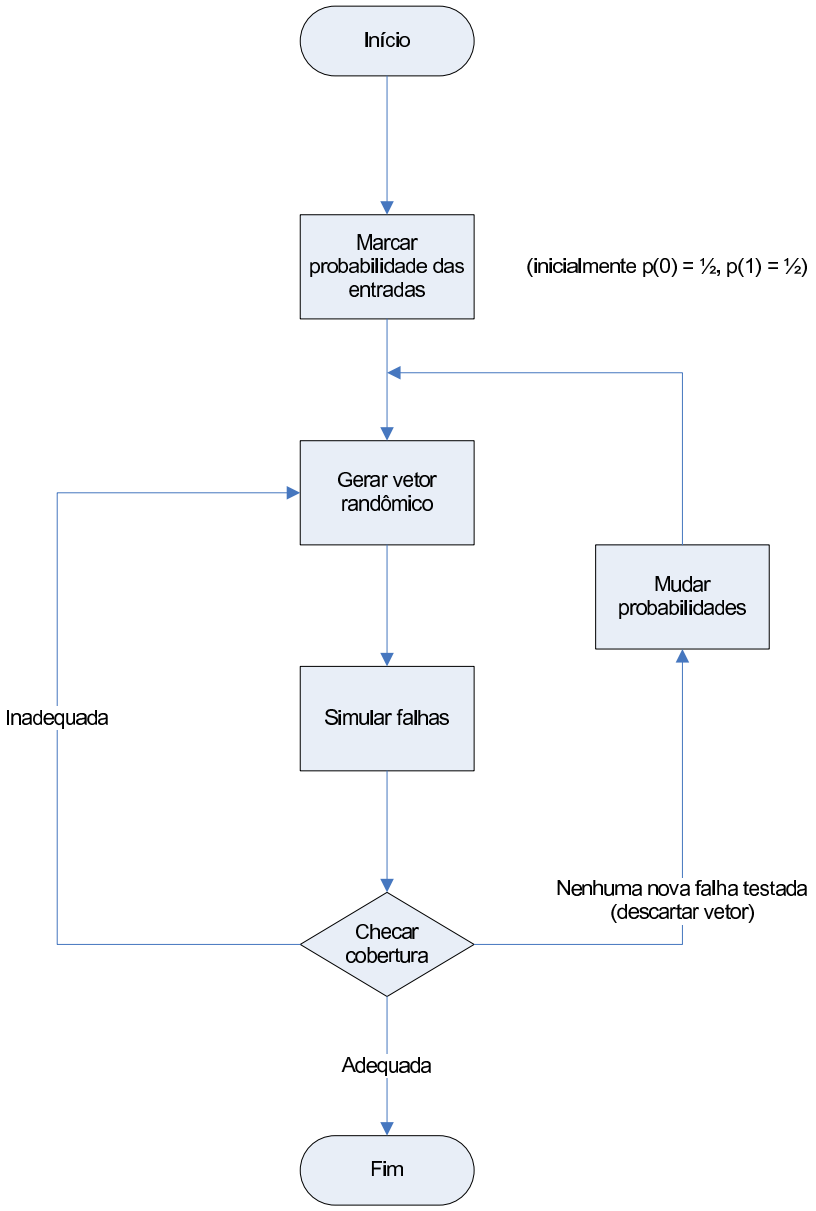

Figura 2.8: Fluxo Típico de Verificação por ATPG [1]

O modelo de falha comumente analisado nos algoritmos de ATPG é chamado de *stuck-at-fault*. É uma maneira de se modelar uma falha física em uma falha lógica, e representa o fato de um determinado sinal lógico obter um valor fixo 0 ou 1. A idéia é gerar vetores de entrada que ativem as falhas stuck-at-fault.

O primeiro algoritmo mais comum proposto para ATPG chama-se Algoritmo-D[45], proposto por Ruth em 1966. A partir deste, vários outros algoritmos eficientes foram propostos, tais como PODEM[46], FAN[47] e TOPS[1].

Capítulo 3

Heurísticas de Particionamento

Este capítulo trata das heurísticas de particionamento de circuitos que foram desenvolvidas e apresenta brevemente a metodologia de verificação proposta.

3.1 Metodologia de Verificação

A metodologia proposta permite a verificação de equivalência de circuitos integrados combinacionais ou sequenciais, descritos em linguagens de descrição de hardware utilizando-se um resolvedor SAT. Primeiramente, os circuitos são analisados e particionados com duas heurísticas: por largura e/ou por ciclos. Cada um dos particionamentos é então transformado no formato de leitura de descrições do resolvedor SAT, que é o CNF. Dessa maneira, cada um dos particionamentos é um problema diferente a ser resolvido, que é enviado para uma instância única do resolvedor SAT. Similaridades estruturais encontradas durante a verificação são realimentadas iterativamente para tentar reduzir o espaço de busca, até que a verificação esteja completa.

Inicialmente, as descrições dos circuitos estão em alto nível. As linguagens mais utilizadas atualmente para tal são Verilog, SystemVerilog e VHDL. São linguagens estruturadas com foco específico em descrição de hardware. A síntaxe de Verilog é semelhante à de C, enquanto VHDL se assemelha à linguagem ADA. SystemVerilog é uma extensão do Verilog, adicionando diversos recursos, com orientação à objetos e a possibilidade de se escrever asserções, propriedades e pontos de cobertura que se deseja verificar. O objetivo é facilitar a descrição e verificação de propriedades desejáveis nos circuitos. A maioria das ferramentas comerciais de verificação já suporta SystemVerilog e suas construções.

A implementação em alto nível deve ser lida e sintetizada, para ser criada internamente no sistema, que representa os circuitos em termos de suas portas lógicas, e não

no nível de RTL. Para tanto, é necessário efetuar a síntese do circuito. O sintetizador não faz parte do escopo deste trabalho, ou seja, nao foi construído um para a tarefa e não integramos nenhuma das opções gratuitas existentes, sendo esta uma possibilidade para trabalhos futuros. Optamos por implementar um parser simples que lê um formato de entrada onde o circuito já foi sintetizado no nível de portas lógicas, e que está descrito no apêndice.

A partir deste ponto, obtém-se o circuito, que é representado por um grafo onde os nodos são as portas lógicas, com vértices ligando-as umas as outras. Cada porta lógica pode possuir um ou mais bits de largura, dependendo da largura da estrutura que ela representa na descrição em alto nível. Esta decisão de implementação nos permite particionar o circuito em largura mais facilmente do que se tivéssemos utilizados exclusivamente lógica com apenas um bit por porta, mesmo que em implementações reais não existam portas com 32 bits de saída por exemplo. Como o objetivo não é sintetizar o circuito para produção, e sim verificar a equivalência funcional de duas descrições de um mesmo modelo, isto não invalida a nossa decisão. Nos testes realizados não utilizamos este recurso entretanto. O procedimento descrito pode ser visto na figura 3.1.

Figura 3.1: Pré-Processamento do Circuito

Para exemplificar melhor este processo, abaixo temos o código Verilog de um somador de quatro bits, e na figura 3.2 a lista de portas derivada a partir desta descrição em alto nível. Este somador é um dos circuitos do benchmark ISCAS85 [48], comumente utilizado para avaliar e comparar a performance de resolvedores SAT, BDDs e algoritmos de ATPG.

```
module Circuit74283b (C0, A, B, S, C4);

  input[3:0]    A, B;
  input         C0;

  output[3:0]   S;
  output        C4;
```

```
  TopLevel74283b Ckt74283b (C0, A, B, S, C4);

endmodule /* Circuit74283b */

/*************************************************************************/

module TopLevel74283b (C0, A, B, S, C4);

  input[3:0]    A, B;
  input         C0;
  output[3:0]   S;
  output        C4;
  wire[4:0]     CS;

  assign CS = A + B + C0;
  assign S = CS[3:0];
  assign C4 = CS[4];

endmodule /* TopLevel74182b */
```

3.2 Particionamento por Largura

O particionamento do circuito em largura, consiste em agrupar o cone de lógica dos bits de saída no número de partes especificadas pelo usuário. Cada agrupamento forma um conjunto de cláusulas CNF, que representa uma instância de um problema SAT. Por exemplo, um circuito de 32 bits de largura pode ser dividido em 4 partições contendo 8 bits cada.

Este particionamento pode ser aplicado a circuitos com mais de um bit de saída. A idéia é poder trabalhar com diversos particionamentos diferentes, deixando esta decisão a cargo do usuário. Ele deve especificar em quantas partes o circuito será particionado. O sistema trata de verificar se o valor pedido é um valor válido. Na prática, o número de partes deve ser menor ou igual ao número de bits de saída. Se for um valor pelo qual não há divisão exata, o resto da divisão se torna uma partição. Ao executar a ferramenta, um dos parâmetros que o usuário deve fornecer é quantos bits de saída devem ser agrupados em cada partição.

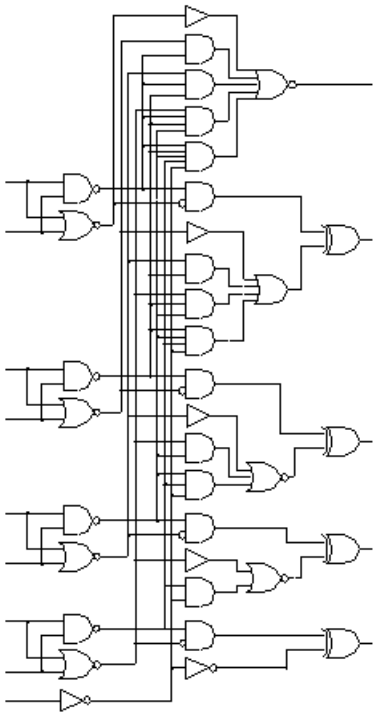

Figura 3.2: Lista de Portas do Somador de Quatro Bits

O próximo passo é a montagem dos cones de influência. Ela é feita basicamente por um algoritmo de busca em largura em grafos, ou BFS (*Breadth First Search*) [49]. A busca termina quando atingimos as entradas primárias do circuito.

Após montar os cones de lógica de cada bit de saída individualmente, agrupa-se os cones de acordo com a largura desejada. Após o agrupamento, faz-se a conversão de cada agrupamento em um conjunto de cláusulas CNF, que é então passado ao resolvedor SAT.

Do ponto de vista do resolvedor, ele recebe vários problemas individuais, um para cada particionamento gerado. O resolvedor então passa ao processamento dos problemas, retornando em seguida uma resposta para cada um dos mesmos, indicando se o resultado foi satisfazível ou não. Em caso negativo, o resolvedor também retorna as cláusulas de conflito relevantes. Veremos mais adiante como estas cláusulas podem acelerar a resolução de diversas instâncias.

Esta heurística de particionamento é mais eficiente se um bit de saída não depender diretamente do anterior. Um exemplo claro onde esta heurística não resulta em ganhos expressivos diretamente é no caso de multiplicadores combinacionais compostos por *Carry-Lookahead Adders* ou CLA. Na prática, o cone de lógica do último bit de saída inclui o circuito inteiro, e assim sucessivamente para todos os outros bits, ou

seja, cada particionamento inclui praticamente todo o particionamento anterior. No entanto, o particionamento continua sendo bastante válido, pois ele nos permite ter um aproveitamento das cláusulas de conflito relevantes.

O agrupamentos dos problemas por processos é feito dividindo-se o número de problemas pelo número de processos. Não leva-se em conta a intersecção entre cones de influência de um mesmo circuito no agrupamento. Idealmente, se a intersecção for grande, deve-se agrupar os problemas em um só, pois assim aproveita-se o aprendizado de cláusulas de conflito SAT internamente no resolvedor.

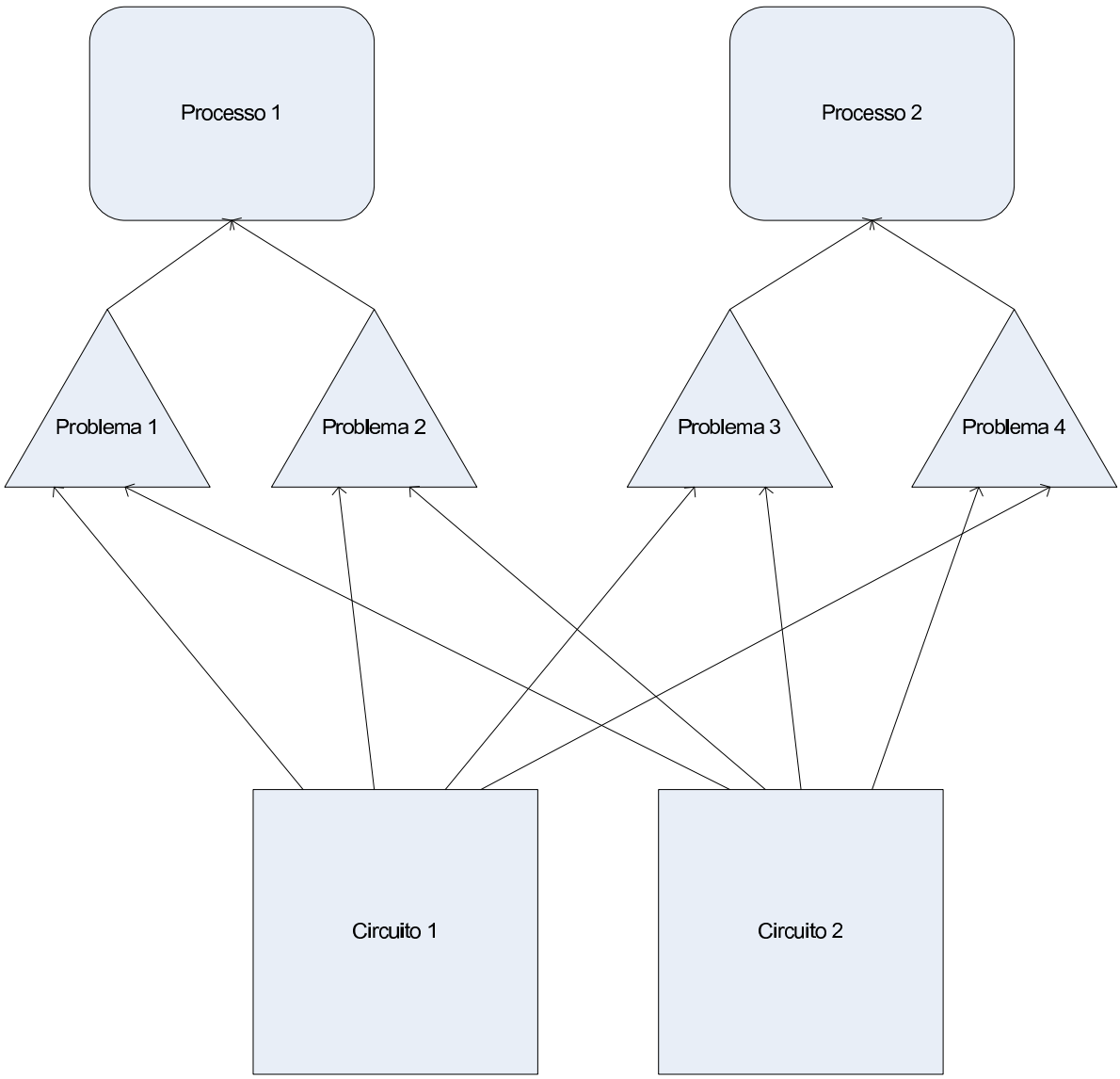

Figura 3.3: Exemplo de Agrupamento de Problemas por Processos

Veja o exemplo na figura 3.3: temos dois circuitos com quatro saídas primárias cada. Utilizando-se apenas o particionamento em largura, temos quatro problemas, um para cada par de saídas primárias a serem comparadas. Se utilizarmos dois processos na resolução do problema, os problemas 1 e 2 serão agrupados e enviados para o processo 1, e os problemas 3 e 4 formarão outro agrupamento, a ser enviado para o processo 2. O cálculo de intersecção dos cones de influência para um agrupamento mais eficiente é um trabalho futuro a ser desenvolvido.

Vamos dar um simples exemplo de como funcionaria o particionamento por largura num pequeno circuito combinacional. O circuito utilizado neste exemplo é o C17 do benchmark ISCAS85. Abaixo, temos o seu código estrutural em Verilog:

```
module nand (a,b,c);

input a,b;
output c;

assign c = !(a && b);

endmodule

module c17 (N1,N2,N3,N6,N7,N22,N23);

input N1,N2,N3,N6,N7;

output N22,N23;

wire N10,N11,N16,N19;

nand NAND2_1 (N10, N1, N3);
nand NAND2_2 (N11, N3, N6);

nand NAND2_3 (N16, N2, N11);
nand NAND2_4 (N19, N11, N7);

nand NAND2_5(N22, N10, N16);
nand NAND2_6 (N23, N16, N19);
```

```
endmodule
```

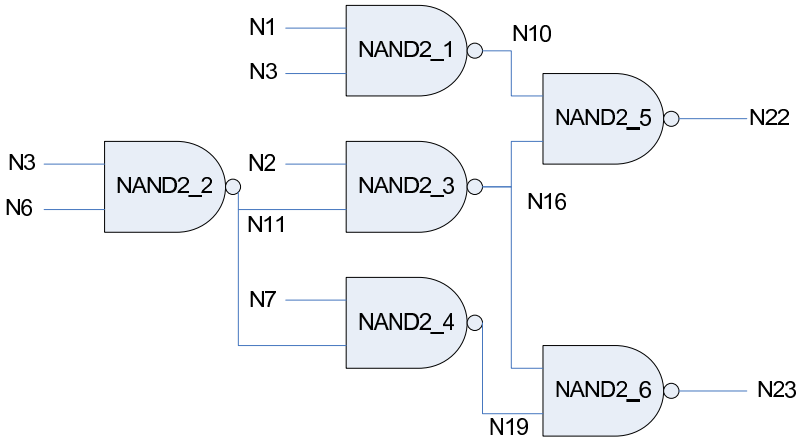

Figura 3.4: Circuito Combinacional C17 do Benchmark ISCAS85

Neste circuito simples, como podemos ver, as entradas são os sinais N1, N2, N3, N6 e N7. As saídas são os sinais N22 e N23. Supondo que iremos particionar este circuito por largura, criaremos duas partições, uma para o bit N22 e outra para o bit N23. Na figura 3.5, podemos ver o cone de lógica de cada um dos bits de saída assinalados. O cone de N22 é formado pelas portas NAND2_1, NAND2_2, NAND2_3 e NAND2_5 enquanto o cone de N23 é formado pelas portas NAND2_2, NAND2_3, NAND2_4 e NAND2_6. Estes cones são obtidos através do algoritmo de busca em profundidade, que percorre o grafo do circuito a partir da saída até chegar as entradas primárias do mesmo.

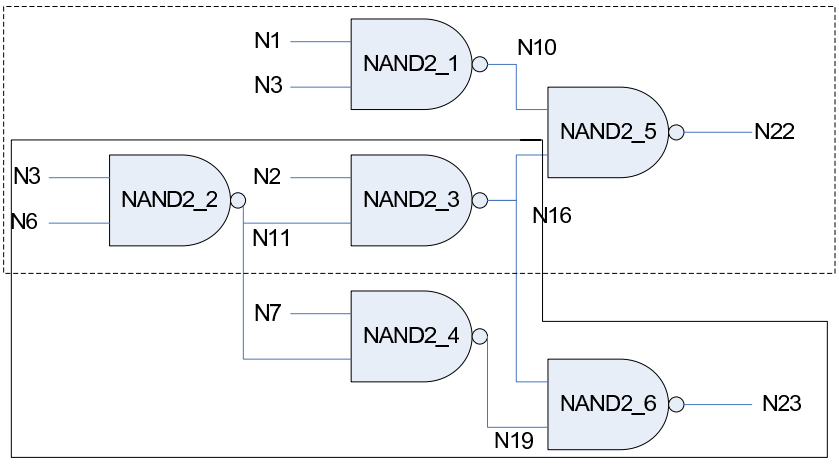

Figura 3.5: Cones de influência para N22 e N23

Após obtermos os cones de cada um dos bits de saída, podemos gerar as cláusulas

CNF correspondentes aos mesmos. Estas cláusulas são então escritas no formato de entrada do resolvedor zchaff [50], que utiliza o padrão DIMACS. O procedimento é repetido para todos os cones de lógica de ambas as descrições que se deseja comparar. Após este processo, deve-se gerar as cláusulas CNF que serão utilizadas para comparar as saídas de ambos os circuitos para verificar se as implementações são equivalentes. Isso é obtido fazendo-se um XOR das saídas, e checando se o resultado do XOR é igual a um. Se o resultado for UNSAT, então os circuitos são equivalentes. Se estivermos comparando circuitos combinacionais e o resultado for SAT, então eles não são equivalentes, mas se forem circuitos sequenciais, podemos efetuar o desdobramento dos mesmos e continuar a verificação por quantos ciclos forem necessários até atingir o resultado UNSAT, ou chegarmos ao limite máximo de ciclos definido pelo usuário.

3.3 Desdobramento de Circuitos por Ciclos

Nesta seção descreveremos como é efetuado o desdobramento de circuitos sequenciais na verificação de equivalência usando um resolvedor SAT. O desdobramento é necessário nos casos em que a equivalência não é encontrada no ciclo zero. Além disso, o aprendizado de cláusulas SAT é efetuado durante o desdobramento dos circuitos, por indução. Sem o desdobramento, este aprendizado ocorre apenas internamente no resolvedor e depende de um agrupamento em largura eficiente para que seja aproveitado.

O número máximo de ciclos a serem desdobrados é definido pelo usuário. O desdobramento ocorre nos flip-flops do circuito. Ele consiste em replicar toda a lógica na entrada dos flops, representando assim o ciclo anterior. A representação de um flip-flop em CNF para o desdobramento pode ser vista na figura 3.6.

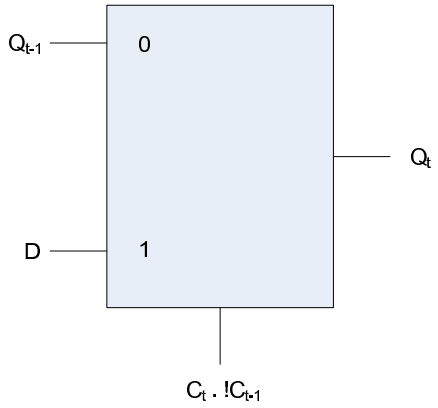

Figura 3.6: Representação de um Flip-Flop em CNF para o Desdobramento

Pela figura, vemos que um flip-flop é representado como um multiplexador, cujas entradas são a lógica do ciclo anterior (t-1), e a lógica do ciclo atual, sendo o controle um AND do clock no ciclo atual com o clock do ciclo anterior invertido. Cada uma das entradas e saídas representam uma variável CNF do problema. O mapeamento do fios do circuito em variáveis CNF é efetuado pela ferramenta, e utilizado posteriormente no desdobramento das cláusulas de conflito aprendidas. Isto significa que a cada ciclo, temos um novo mapeamento de todo o circuito em variáveis e cláusulas CNF: os fios são as variáveis e as portas lógicas definem as cláusulas. O mapeamento de portas lógicas em cláusulas CNF pode ser visto no apêndice.

Vamos agora exemplificar o desdobramento de um circuito simples passo a passo para compreender melhor como ele é feito. Suponha o circuito da figura 3.7.

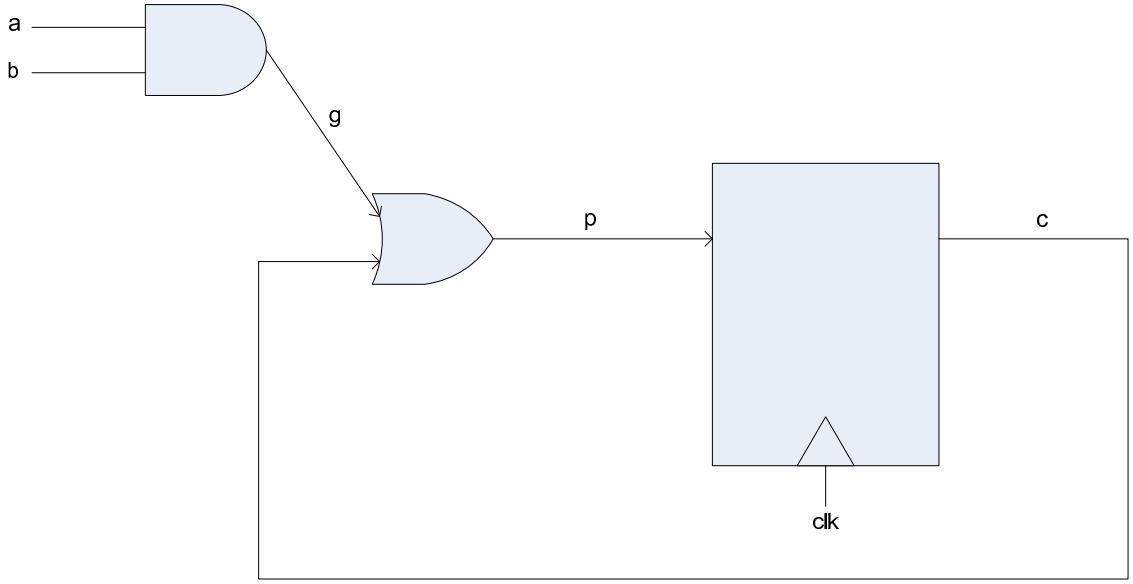

Figura 3.7: Exemplo de Circuito para Desdobramento

O primeiro passo é mapear cada um dos fios em uma variável CNF. No resolvedor que utilizamos, devemos começar o mapeamento pelo número um. Vamos utilizar o mapeamento da figura 3.8.

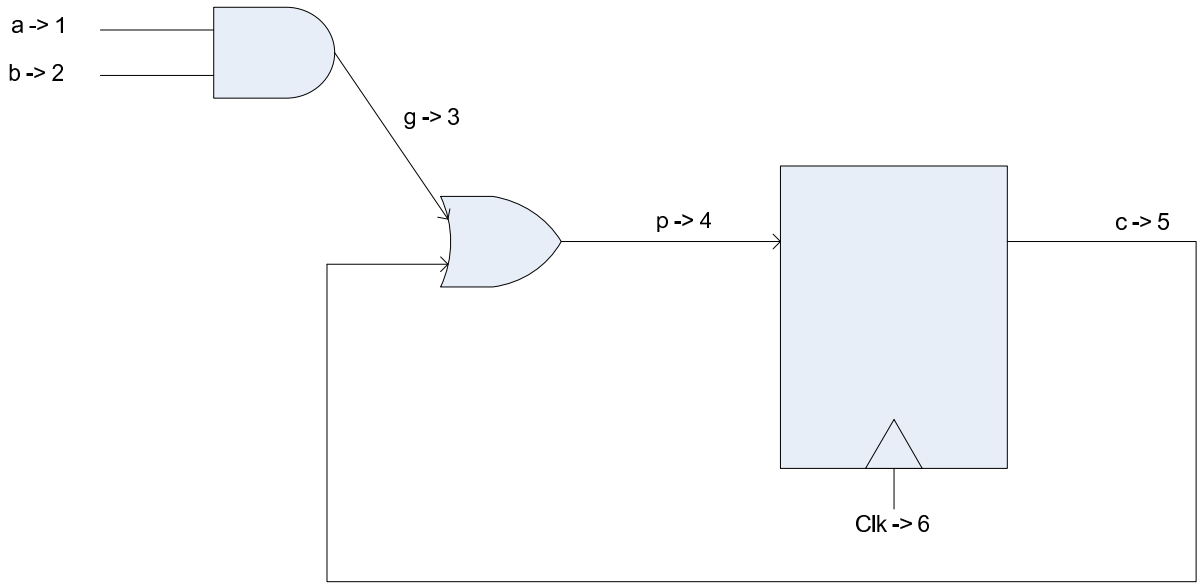

Figura 3.8: Mapeamento de Fios do Circuito em Variáveis CNF

Considerando a representação de um flip-flop em CNF na figura 3.6, temos o circuito da figura 3.9 como o nosso circuito CNF do ciclo zero. Pode-se ver que substituímos o flip-flop por um multiplexador, cuja entrada zero é uma nova variável CNF livre (sete), que representa o ciclo anterior. Esta mesma variável é a entrada da porta lógica onde há a realimentação do flip-flop no circuito.

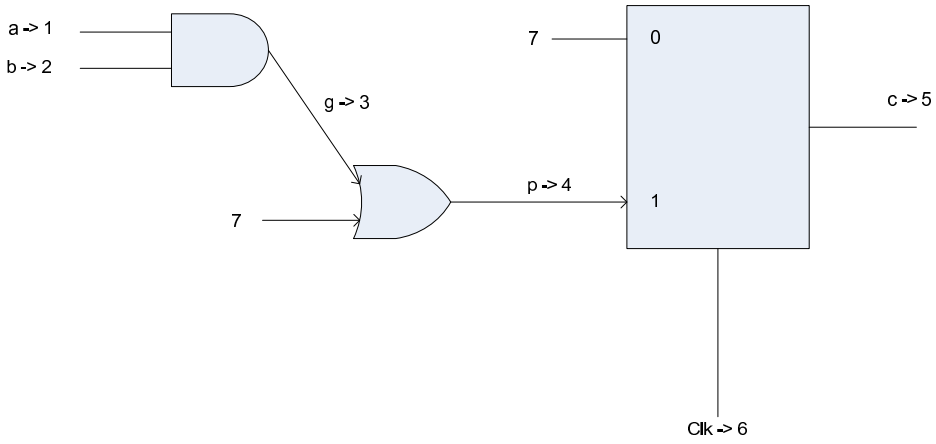

Figura 3.9: Circuito em CNF para o Ciclo Inicial

Efetuando o primeiro desdobramento deste circuito, o resultado pode ser visto na figura 3.10.

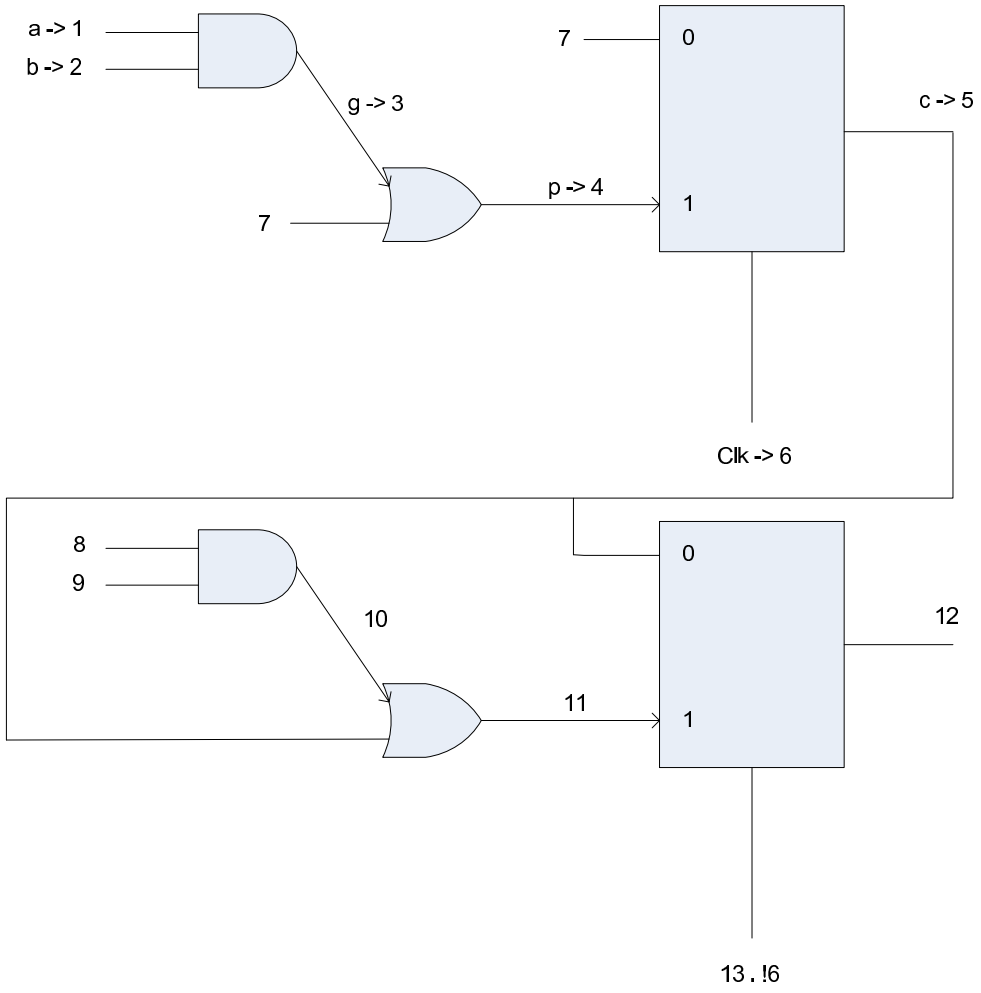

Figura 3.10: Desdobramento do Circuito para o Segundo Ciclo

Note que replicamos todas as portas lógicas, mas assinalando novas variáveis CNF a cada uma delas. Nos pontos de realimentação do flip-flop do ciclo um, conectamos a saída do flip-flop do ciclo zero. Este procedimento pode ser repetido quantas vezes for necessário.

A ferramenta guarda o mapeamento dos fios em suas variáveis CNF correspondentes por ciclo. Isto é necessário para utilizar o aprendizado de cláusulas SAT, através de indução. O desdobramento aplicado ao circuito é aplicado também às cláusulas de conflito relevantes retornadas pelo resolvedor SAT.

3.4 Aprendizado de Cláusulas de Conflito SAT

Uma cláusula de conflito é uma expressão booleana encontrada pelo resolvedor durante a resolução do problema que representa um relacionamento entre variáveis indicando uma restrição que deve sempre ser satisfeita para que se obtenha um assinalamento válido para todas as variáveis do problema. Se a cláusula de conflito não for satisfeita, esse assinalamento não existe e o problema não é satisfazível. Uma cláusula de conflito portanto, pode reduzir o espaço de busca para a resolução do problema, ao invalidar determinados assinalamentos de variáveis.

O aprendizado consiste em se aproveitar determinadas cláusulas de conflito encontradas pelo resolvedor SAT para tentar acelerar a resolução do problema como um todo, ao realimentar as cláusulas no problema inicial. Em determinados tipos de circuito, como por exemplo um multiplicador combinacional, esta técnica é muito eficiente, permitindo a verificação de circuitos bem maiores do que quando não se utiliza a mesma [24]. Isto se deve à organização estrutural do multiplicador, onde o cone de lógica do bit n possui informação sobre os cones de todos os bits anteriores, do n-1 a 0. Sendo assim, as cláusulas de conflito obtidas durante a resolução dos bits anteriores podem ser reaproveitadas, pois elas continuam valendo para as saídas de maior ordem.

O aprendizado pode ser utilizado em conjunto com o particionamento em largura (ao se agrupar diversos problemas em um mesmo processos, o resolvedor SAT utiliza as cláusulas de conflito encontradas internamente) ou por ciclos (realimentando os problemas iniciais e utilizando indução).

No caso do particionamento em largura, o próprio resolvedor SAT irá utilizar as cláusulas internamente durante a resolução do problema, assumindo que se tenha um agrupamento em largura com um determinado nível de intersecção entre os cones de influência das saídas primárias que estão sendo verificadas no mesmo agrupamento. Isso significa que cláusulas de conflito resultantes de um cone de influência potencialmente ajudam na resolução de outros, principalmente se a intersecção for grande.

O aprendizado de cláusulas se dá por indução na verificação de circuitos sequenciais. Os ciclos são resolvidos sequencialmente, começando-se do zero até o número de ciclos determinado pelo usuário. As cláusulas encontradas em um determinado ciclo n podem ser reaproveitadas no ciclo n+1 através de indução, ao se fazer o desdobramento do circuito. Vale lembrar que não se assume nenhum valor inicial para os fip-flops, pois isto impediria o reaproveitamento de cláusulas por indução.

Considere o conjunto de cláusulas dos circuitos C_n, onde n representa o ciclo a ser verificado. Chamemos a proprieadade a ser verificada de P_n, onde n novamente

representa o ciclo verificado. No nosso caso, a proprieadade representa a equivalência dos circuitos, que é a porta XOR cujas entradas são as saídas primárias dos dois circuitos. Temos uma porta XOR para cada par de saídas a serem verificadas, se o circuito tiver largura maior do que um. O problema do ciclo n pode ser representado como $C_0 \wedge C_1 \wedge ... \wedge C_n \wedge P_n$. Note que devemos concatenar todas os desdobramentos realizados no circuito até se chegar ao ciclo n.

Vejamos como a resolução e o aprendizado se dá passo a passo. Na figura 3.11, temos a definição do problema no ciclo zero, que é resolvido por uma instância SAT, que encontra um conjunto de cláusulas de conflito, definido como K_0. Esse conjunto de cláusulas vai para o banco de cláusulas da ferramenta. Caso o resultado do ciclo zero tenha sido UNSAT, os circuitos são equivalentes e para-se a execução. Caso contrário, efetua-se o desdobramento dos circuitos e continua-se a verificação até chegar no resultado UNSAT ou atingir o limite de iterações. Vale lembrar que podemos utilizar o particionamento em largura em conjunto com esta técnica.

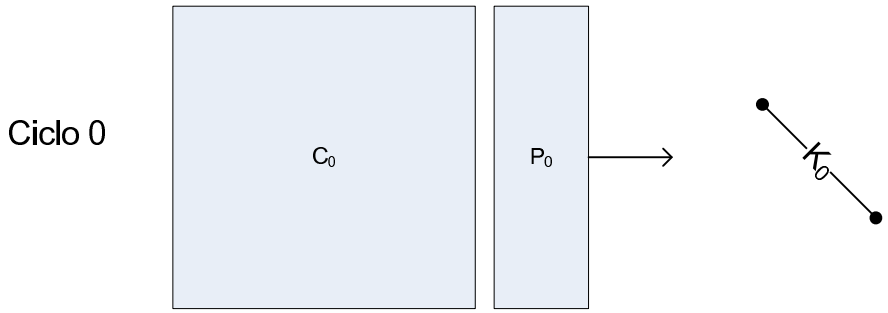

Figura 3.11: Passo 1

Neste caso, gera-se C_1 e P_1. C_1 é obtido através do desdobramento dos circuitos, e P_1 é a proprieadade no ciclo um, como podemos ver na figura 3.12.

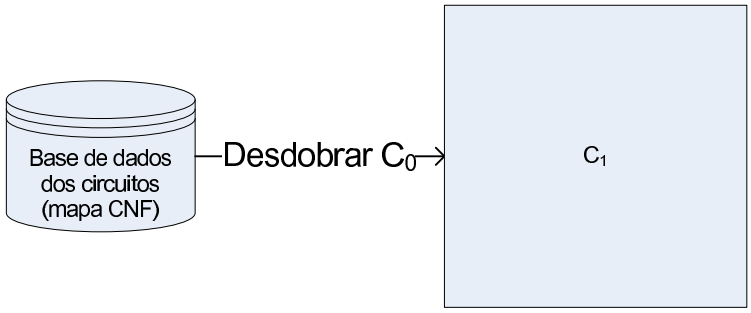

Figura 3.12: Passo 2

Agora faremos o passo indutivo, como pode ser visto na figura 3.13. K_0 diz respeito ao ciclo zero e referencia as variáveis deste ciclo. No entanto, sabemos exatamente como mapear estas cláusulas para serem utilizadas no ciclo um, pois fizemos o desdobramento do circuito previamente. Desta maneira, estamos aplicando o desdobramento nas cláusulas de conflito também obtendo K_1, e aplicando-as no problema inicial: $C_1' = C_1 \wedge K_1$.

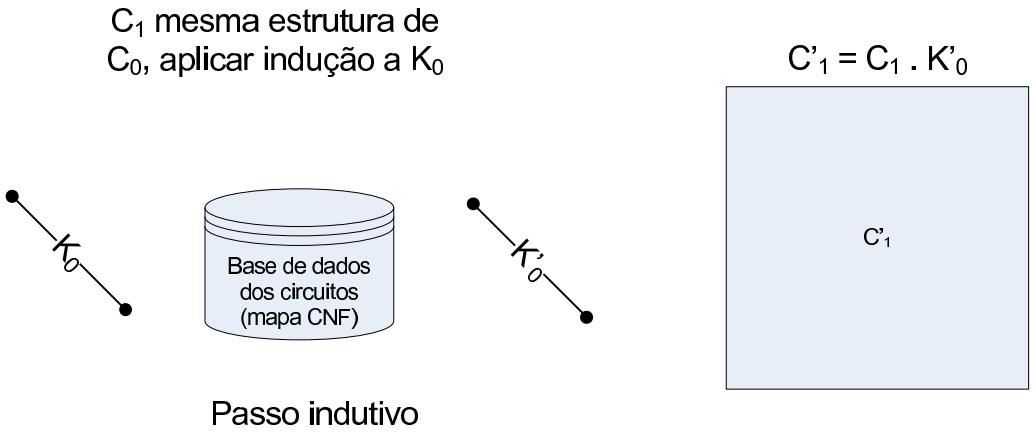

Figura 3.13: Passo 3

Desta maneira, obtemos o nosso novo problema concatenando as cláusulas do problema inicial com as cláusulas do ciclo um mais os conflitos que foram aprendidos, como podemos ver na figura 3.14': $C'_0 \wedge C'_1 \wedge P_1$, onde $C'_0 = C_0 \wedge K_0$.

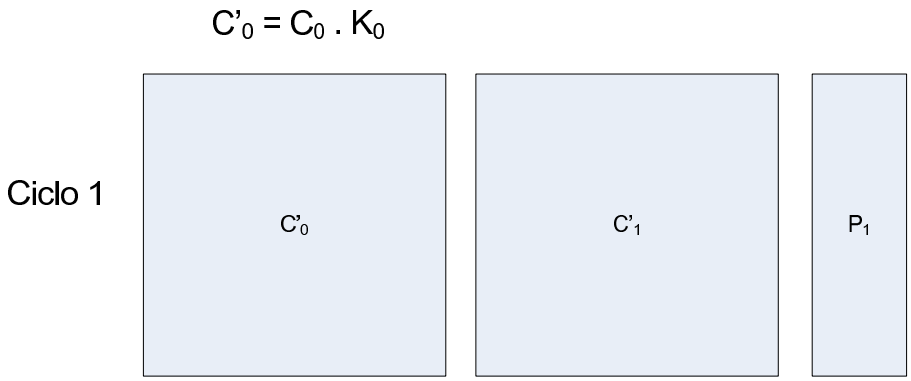

Figura 3.14: Passo 4

Capítulo 4

Arquitetura e Implementação do Sistema

Neste capítulo iremos descrever a arquitetura da ferramenta implementada como um todo, além de descrever o ambiente de implementação utilizado.

4.1 Ambiente de Implementação

Todo o trabalho foi desenvolvido utilizando-se a linguagem de programação C++ [51], e o sistema operacional Linux [52], (distribuições SuSE Linux [53] e Fedora Core[54]).

Todo o código desenvolvido neste trabalho é compatível com o padrão ISO/ANSI C++, de maneira a facilitar a portabilidade da aplicação para outras plataformas. Utilizamos o compilador GNU C [55], que é bastante difundido e utilizado, gerando código de qualidade e eficiente.

Com relação à escolha do sistema operacional, ela se justifica pelos seguintes pontos:

- É um sistema operacional gratuito
- Bastante difundido no mercado
- Relativamente fácil de se utilizar, principalmente devido ao grande número de desenvolvedores no mundo inteiro que se dedicam à constantes melhorias no sistema
- Existem vários resolvedores SAT que podem ser utilizados neste sistema operacional, ampliando as possibilidades de integração desta ferramenta à diferentes resolvedores

- O Linux suporta diversas plataformas de hardware: x86, ARM, PPC, Sparc e outras

O resolvedor SAT paralelo foi implementado utilizando o sistema LAM/MPI [56, 57].

Com relação aos testes realizados neste trabalho, utilizamos basicamente um cluster de máquinas Linux, descrito em mais detalhes no capítulo de resultados. O resolvedor paralelo utilizado foi desenvolvido em [24]. O motor principal deste resolvedor é baseado no ZChaff [50], um dos melhores resolvedores SAT disponíveis atualmente, que inclusive sempre esteve bem colocado nas últimas competições mundiais de resolvedores SAT [58].

4.2 Arquitetura do Sistema

O verificador de equivalência pode ser dividido em duas partes: o front-end e o back-end. Este trabalho trata exclusivamente do front-end da aplicação, sendo que o back-end é o resolvedor SAT distribuído [24].

O front-end é responsável pelas seguintes tarefas:

- Leitura dos circuitos em um formato de entrada específico
- Divisão dos circuitos de acordo com a heurística especificada pelo usuário
- Criação dos sub-problemas a partir da divisão realizada
- Enviar todos os sub-problemas para o resolvedor SAT paralelo
- Obter os resultados e, se aplicável, realizar o aprendizado de cláusulas de conflito

A implementação do sistema foi toda feita utilizando-se o paradigma de orientação a objetos. Sendo assim, iremos descrever cada uma das classes definidas e suas interações. Depois descreveremos a integração com o resolvedor SAT paralelo e a API entre eles. Uma visão geral da arquitetura pode ser vista na figura 4.1.

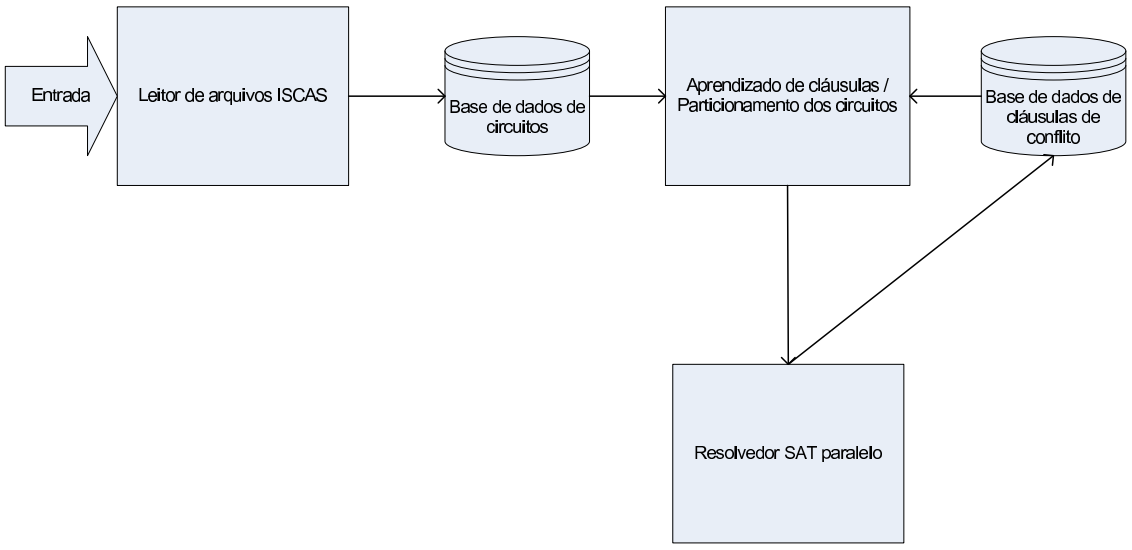

Figura 4.1: Arquitetura do Sistema

4.3 A Classe IscasParser

Para realizar testes com circuitos reais, foi necessário implementar um parser. O primeiro passo foi escolher a linguagem aceita pelo parser. As linguagens de descrição de hardware mais comuns são Verilog e VHDL. No entanto, implementar um parser para elas ou utilizar parsers existentes no mercado incorreria em um trabalho muito dispendioso, e não faz parte do escopo deste trabalho. Ao mesmo tempo, a escolha de um formato que não restringisse os testes da ferramenta era importante. A escolha recaiu nos benchmarks ISCAS (ISCAS85, 89 [59]) e ITC99 [60]. Este conjunto de benchmarks possui uma quantidade razoável de circuitos, que são muito utilizados na literatura. Existem tanto circuitos sequenciais como exclusivamente combinacionais. O formato de entrada aceito pelo parser é chamado de bench, e seus detalhes podem ser vistos no apêndice.

4.4 A classe GATE e suas derivações

A abstração que representa os circuitos internamente na memória são classes do tipo Circuit (que iremos descrever na próxima seção) compostas basicamente pela sua lista de portas lógicas e a interconexão entre as mesmas. Essa lista de portas lógicas por sua vez é formada por objetos da classe GATE, que possui as seguintes derivações: MUX, AND, NAND, OR, NOR, XOR, XNOR, DFF, BUF e NOT. Outros tipos de portas lógicas não são tratados atualmente. Cada um dos elementos representa uma classe,

sendo que todos elas derivam da classe principal GATE. Este conjunto de elementos lógicos é suficiente para representar todos os circuitos presentes no benchmark ISCAS e muitos outros circuitos também.

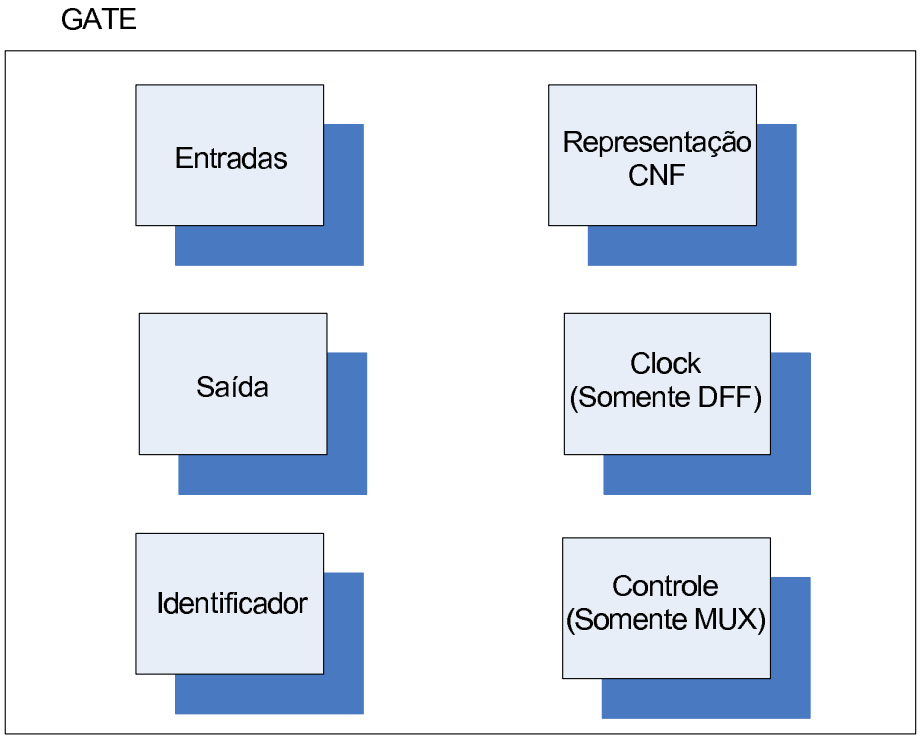

Figura 4.2: Classe GATE

Cada uma das portas possui basicamente os seguintes elementos: sinais de entrada, sinais de saída, sinal de controle no caso do MUX e sinal de clock no caso do DFF. Os sinais podem ser considerados como os fios do circuito, interligando as portas lógicas. Cada um dos elementos lógicos possui uma representação CNF própria. Um detalhamento da conversão de portas lógicas em um conjunto de cláusulas CNF pode ser visto no apêndice. As portas possuem um identificador único no circuito, que corresponde ao sinal de saída da mesma. Além disso, cada um dos fios é representado por um inteiro distinto, que correspondem diretamente às variáveis das cláusulas CNF que são enviadas para o resolvedor. Além disso, é guardado um mapeamento destes inteiros para os nomes originais que existem dentro do arquivo de entrada no formato bench. A classe GATE possui suporte para portas com largura maior do que um, mas este recurso não foi utilizado aqui, pois em todos os benchmarks as portas tem entradas e saídas de um bit apenas. Por premissas de tempo não foi possível encontrar circuitos no nível de portas lógicas que tivessem portas com mais de 1 bit de largura. É na

classe GATE e nas suas derivações que estão definidos os métodos que convertem as portas lógicas em cláusulas CNF. Esta classe possui também um campo onde pode-se guardar o valor da porta. Isto pode ser utilizado no futuro na implementação de um simulador.

4.5 A Classe Circuit

A classe Circuit representa um circuito, através de um conjunto de objetos do tipo GATE interligados. Ela também possui uma lista contendo as entradas primárias e outra com as saídas primárias. Estas listas são guardadas separadamente pois as entradas e saídas primárias são utilizadas em diversas operações, o que justifica termos um meio de acesso rápido e eficiente a esses dados. Estas listas são montadas pelo IscasParser durante a leitura do arquivo de entrada. A lista de flip-flops do circuito também é gerada separadamente, para facilitar o desdobramento do circuito.

É também na classe Circuit que existe o mapeamento entre os nomes reais do sinais e suas variáveis CNF. Este é um recurso para facilitar a depuração em caso de erros, pois o usuário saberá exatamente qual sinal está sendo verificado e quais fazem parte das cláusulas de conflito encontradas.

Os métodos mais importantes desta classe são três:

1. buildCOI(GATEID id): este método recebe como entrada o identificador de uma saída primária e gera o cone de influência da mesma. É um caminhamento em largura no grafo que representa o circuito, cuja raiz é a saída primária

2. buildSequentialCOI(GATEID id): este método também gera o cone de influência de uma saída primária, mas ele separa o cone em partes, uma para cada ciclo encontrado. É utilizado no desdobramento do circuito por ciclos

3. generateCNFMap(): este método gera o mapeamento do circuito em cláusulas CNF

4.6 A Classe EquivalenceChecker

A classe EquivalenceChecker é a classe principal do programa. Ela é responsável por chamar o parser, gerenciar os circuitos, os particionamentos e a criação dos problemas que deverão ser enviados às instâncias dos resolvedores SAT. Ela também fica responsável por obter do resolvedor as respostas relevantes (o aprendizado de cláusulas

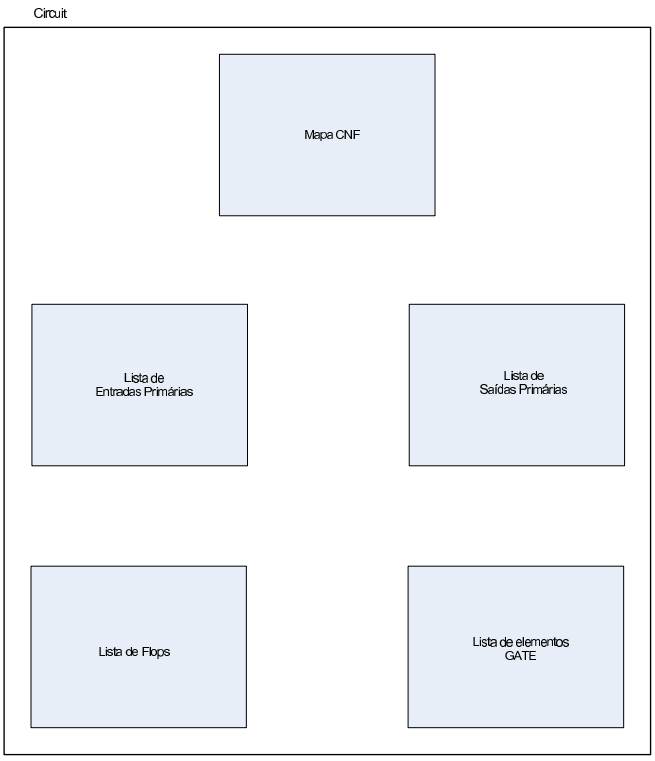

Figura 4.3: Classe Circuit

de conflito) e remontar os problemas utilizando-as de acordo com a relevância das mesmas para cada uma das partições.

Inicialmente esta classe lê os arquivos de entrada, utilizando o IscasParser, e monta os dois circuitos a serem comparados. Tendo os circuitos montados, o próximo passo é aplicar as heurísticas de particionamento na criação dos problemas. O usuário é quem escolhe o que deve ser utilizado, passando como paramêtros as heurísticas. O EquivalenceChecker irá utilizar os métodos apropriados da classe Circuit para obter as divisões dos circuitos.

Tendo as partições de ambos os circuitos prontas, passa-se a construção do problema. Isto é feito agrupando os cones de influência das saídas de maneira apropriada, mapeando as entradas primárias dos cones, que devem ser as mesmas para ambos os circuitos, e na outra ponta dos cones, faz-se um XOR das saídas primárias. Após a montagem do problema no formato CNF utilizado pelo ZChaff (DIMACS), o EquivalenceChecker gera objetos da classe ProblemData, que são os problemas a serem resolvidos mais alguns parâmetros de entrada utilizados pelo resolvedor.

O resolvedor SAT irá então tentar provar que a saída dos XORs de todos os particionamentos deve ser igual a um. Se o resultado for UNSAT, isto quer dizer em

nenhuma das 2^n combinações das entradas primárias o resultado do XOR é um, ou seja, os cones de influência das saídas comparadas são idênticos em funcionalidade. Se o resultado for UNSAT para todos os bits de saída, então os circuitos são equivalentes.

4.7 A Classe ProblemData

Esta classe é a interface de comunicação entre o EquivalenceChecker e o resolvedor SAT paralelo. O EquivalenceChecker deve montar um objeto da classe ProblemData para cada problema a ser resolvido. Esta classe contém o problema descrito em CNF, já pronto para ser resolvido, e também os parâmetros a serem utilizados na resolução, sendo eles:

- Limite de tempo
- Limite de memória
- Solver a ser utilizado (no futuro, se tivermos mais de um resolvedor além do Zchaff, este campo poderá ser utilizado)
- Número de variáveis: basicamente é o número de fios individuais presente no problema. Isto inclui a partição de cada um dos dois circuitos a serem comparados mais a cláusula SAT a ser provada pelo resolvedor, que é basicamente um XOR das saídas e a cláusula unitária onde dizemos que a saída do XOR deve ter o valor 1.
- Número de cláusulas: o número de cláusulas presentes no circuito. Cada elemento lógico do circuito é representado por um conjunto de cláusulas. O mapeamento entre o elemento e suas cláusulas CNF pode ser visto no apêndice
- ProblemID: cada problema possui um identificador único
- problem: aqui temos o string representando o problema a ser resolvido, que é o conjunto de cláusulas CNF.

4.8 A Classe ConflictClause

Esta classe representa o banco de dados de cláusulas de conflito que poderá vir a ser utilizado na resolução dos problemas, caso se deseje utilizar o recurso de aprendizado

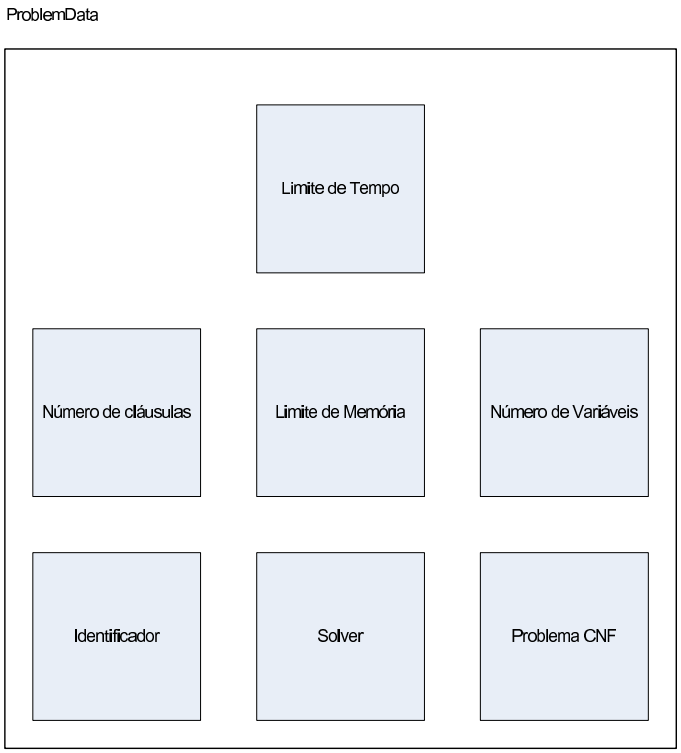

Figura 4.4: Classe ProblemData

das mesmas para acelerar a resolução de algumas partições do problema. Ela basicamente contém um string CNF com a cláusula em si, o tipo da cláusula, que indica qual elemento lógico ela representa e também as suas variáveis em separado. O resolvedor gera objetos desta classe para cada problema resolvido e os retorna para o EquivalenceChecker, mas apenas se conflitos forem encontrados (quando o resultado é SAT, nenhum conflito existe).

4.9 O Resolvedor SAT Paralelo

O resolvedor foi implementado baseado no ZChaff e o paralelismo foi implementado utilizando-se LAM/MPI. Além disso, foi adicionado ao resolvedor um identificador de cláusulas de conflito SAT relevantes. Este identificador procura por cláusulas que representem os seguintes elementos lógicos: inversores, buffers, and, nand, or e nor. As outras cláusulas são descartadas. As que forem encontradas, são então encapsuladas em objetos da classe ConflictClause e enviadas para o EquivalenceChecker.

A idéia é reaproveitar estas cláusulas em partes do problema para acelerar a resolução. A API entre o resolvedor e o particionador é composta basicamente pelas

classes ProblemData e a ConflictClause. O particionamento gera objetos do tipo ProblemData, que são usados pelo resolvedor. Este por sua vez, retorna objetos do tipo ConflictClause, que podem ser utilizados no particionamento, caso o recurso de aprendizado de cláusulas SAT seja habilitado pelo usuário e as cláusulas encontradas sejam relevantes. A arquitetura do resolvedor pode ser vista na figura 4.5.

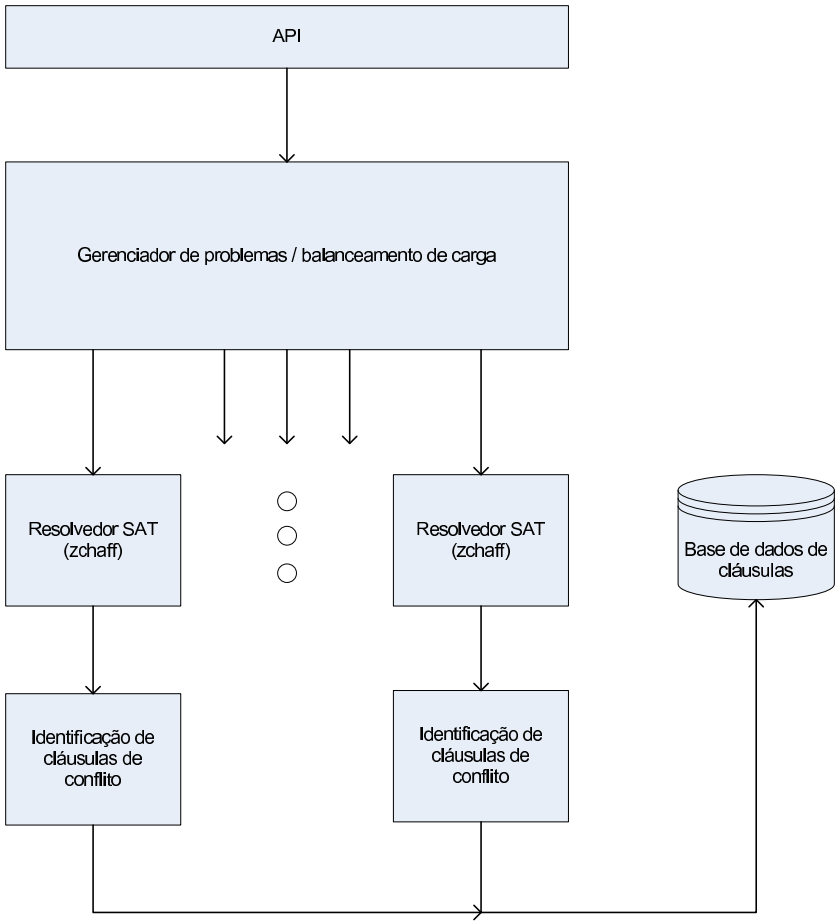

Figura 4.5: Resolvedor SAT paralelo

Capítulo 5

Resultados

Neste capítulo iremos apresentar os resultados obtidos com o trabalho. Antes, iremos descrever em detalhes o ambiente de testes e como funciona o paralelismo do programa.

Foi utilizado um conjunto de máquinas relativamente heterogêneo. São elas:

- Athlon 64 X2 4200+ (dual core) com 2GB de RAM rodando Fedora Core 5 32 bits
- Core 2 Duo E6300 (dual core) com 1GB de RAM rodando SuSE 10 32 bits
- 2 x Athlon 64 3000+ com 2GB de RAM rodando Fedora Core 3 32 bits
- Athlon 64 3500+ com 4GB de RAM rodando Fedora Core 3
- Pentium 4 HT 3.0 GHz com 8GB de RAM rodando Red Hat Enterprise Linux 3 64 bits

Com relação ao ambiente paralelo, foi utilizado o LAM/MPI versão 7.1.3. Ao configurá-lo, define-se a lista de máquinas disponíveis, e durante a alocação dos processos, ele utiliza um escalonamento do tipo round-robin para distribuir as tarefas entre as máquinas. Vale lembrar que todas estas máquinas suportam o conjunto de instruções x86-64 bits, mas apenas uma possui o sistema operacional com o suporte adequado, de maneira que não pudemos aproveitar esta vantagem, compilando todo o código em 32 bits.

Na execução da ferramenta, especifica-se quantos processos deseja-se utilizar. Nos testes realizados, disparamos um processo por processador apenas. No caso de máquinas dual-core, disparamos dois processos por máquina. Isso significa que o número de processos pode ser maior que o número de máquinas disponíveis, mas não excedemos em nenhum momento o número de processadores existentes. Um dos processos é o mestre,

responsável pela leitura dos arquivos de entrada contendo a descrição dos circuitos, o particionamento dos circuitos de acordo com as heurísticas escolhidas e o aprendizado de cláusulas SAT. Os outros processos são chamados de escravos, que disparam instâncias do resolvedor SAT para cada um dos particionamentos, coletam as cláusulas de conflito relevantes e retornam o resultado juntamente com as cláusulas coletadas para o processo mestre. Toda a comunicação entre os processos pelo MPI se dá via rede, ou seja, as máquinas estão numa mesma LAN (*Local Area Network*)[61]. O MPI utiliza o protocolo SSH (*Secure Shell*)[62] para comunicação entre as máquinas. Um exemplo da execução pode ser visto na figura 5.1.

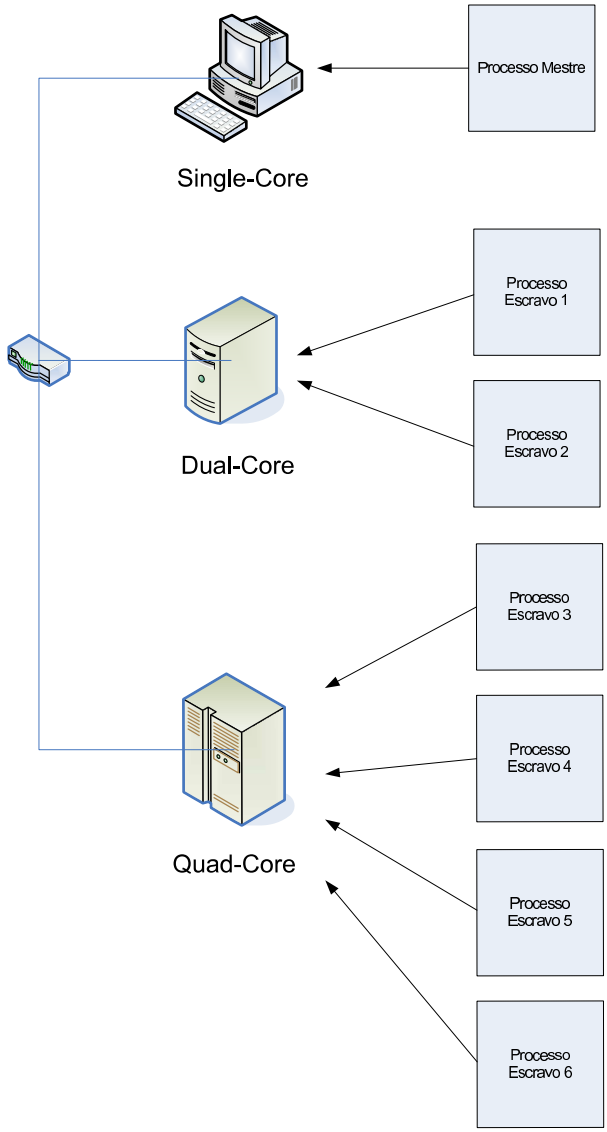

Figura 5.1: Paralelismo Processo x Máquina

Foi constatado que em alguns casos o paralelismo não traz ganho de desempenho, pois existe um custo de processamento associado à comunicação entre processos. Em alguns casos, os resultados pioram ao se adicionar mais processadores. Isso ocorre geralmente com problemas menores, onde o tempo de resolução pelo resolvedor é pequeno em relação aos outros fatores, como a divisão do problema, leitura do circuito, comunicação entre os nodos e retorno de cláusulas de conflito.

Os tempos foram obtidos rodando-se 5 vezes cada um dos testes e tirando-se a média aritmética das execuções. Todos os testes foram feitos comparando-se um circuito com ele mesmo, de maneira que em todas as instâncias a ferramenta indicou que os circuitos são equivalentes. Este artifício não invalida os testes, não havendo vício, como pode-se ver em [63]. Nos testes realizados utilizamos apenas o particionamento em largura, por premissas de tempo. Todos os gráficos possuem a medida do tempo de setup do problema. Estamos considerando como setup as seguintes tarefas:

- Leitura dos circuitos pelo parser
- Geração de todos os cones de influência dos bits de saída do circuito
- Conversão dos cones em conjunto de cláusulas CNF
- Particionamento do problema
- Envio das partições para o processo mestre do resolvedor SAT

O gráfico 5.2 mostra os resultados obtidos para o circuito b18, que faz parte do benchmark ITC99. Neste caso, a execução mais rápida, com 7 processadores, gastou 44% do tempo da da execução com apenas um processo. O setup ficou em 20s aproximadamente. Um grande problema dos benchmarks utilizados é a falta de informação sobre os circuitos, sendo que em muitos casos não há nenhum tipo de dado ou descrição. Neste caso específico, o b18 é a interconexão de circuitos menores para se obter um circuito maior de benchmark. Ele representa cinco cópias de parte de um processador 80386 e duas cópias de parte de um processador viper. Ele possui:

- 88954 portas lógicas (10387 and, 74205 nand, 657 or, 325 nor, 19528 inversores)
- 37 bits de entrada
- 23 bits de saída
- 3320 flip-flops D

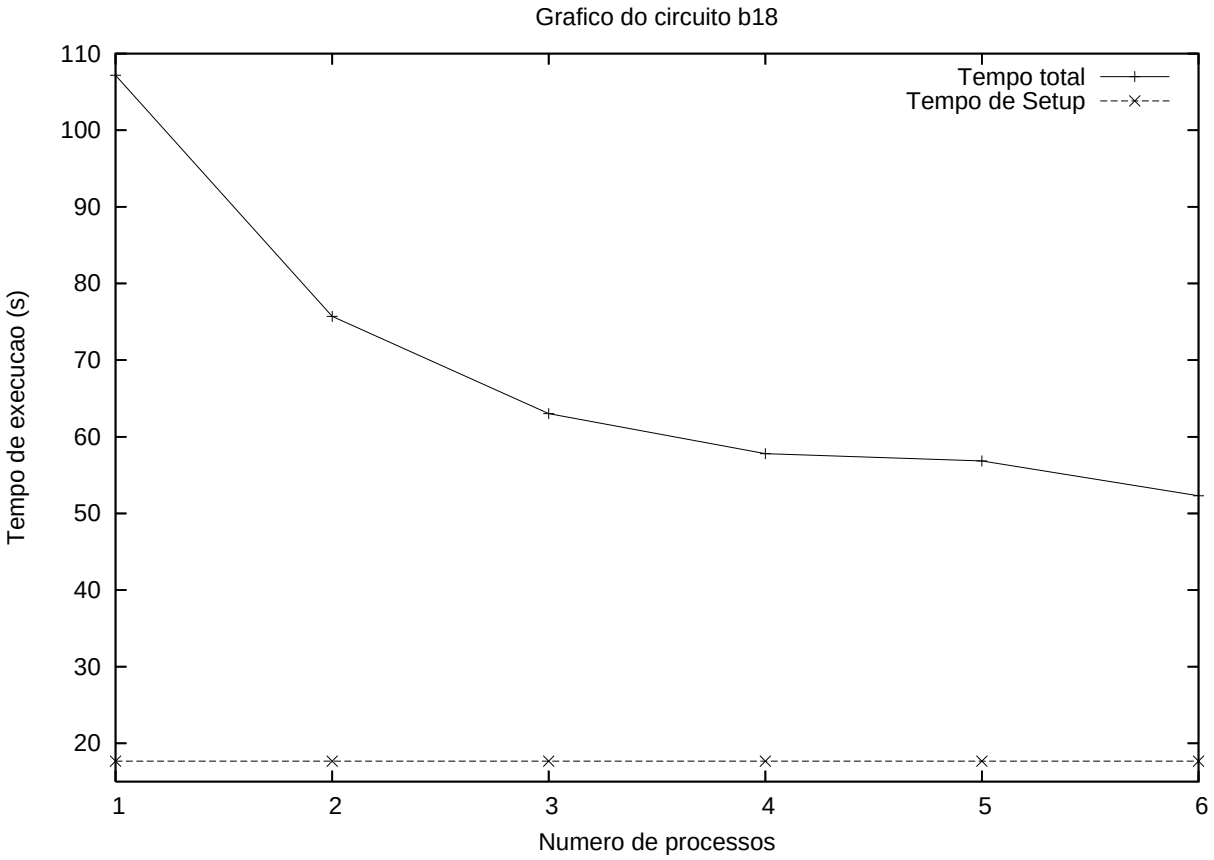

Figura 5.2: Tempos de Execução para b18

O circuito c7552 é um somador/comparador de 32 bits, totalmente combinacional. É composto por:

- 4922 portas lógicas(1310 ANDs, 1904 NANDs, 244 ORs, 54 NORs, 534 buffers, 876 inversores)
- 207 bits de entrada
- 108 bits de saída

É um circuito pequeno, onde o overhead de processamento e comunicação do resolvedor paralelo não se justifica, pois sempre que adicionamos um processo, o tempo de execução aumenta, como pode ser visto no gráfico 5.3.

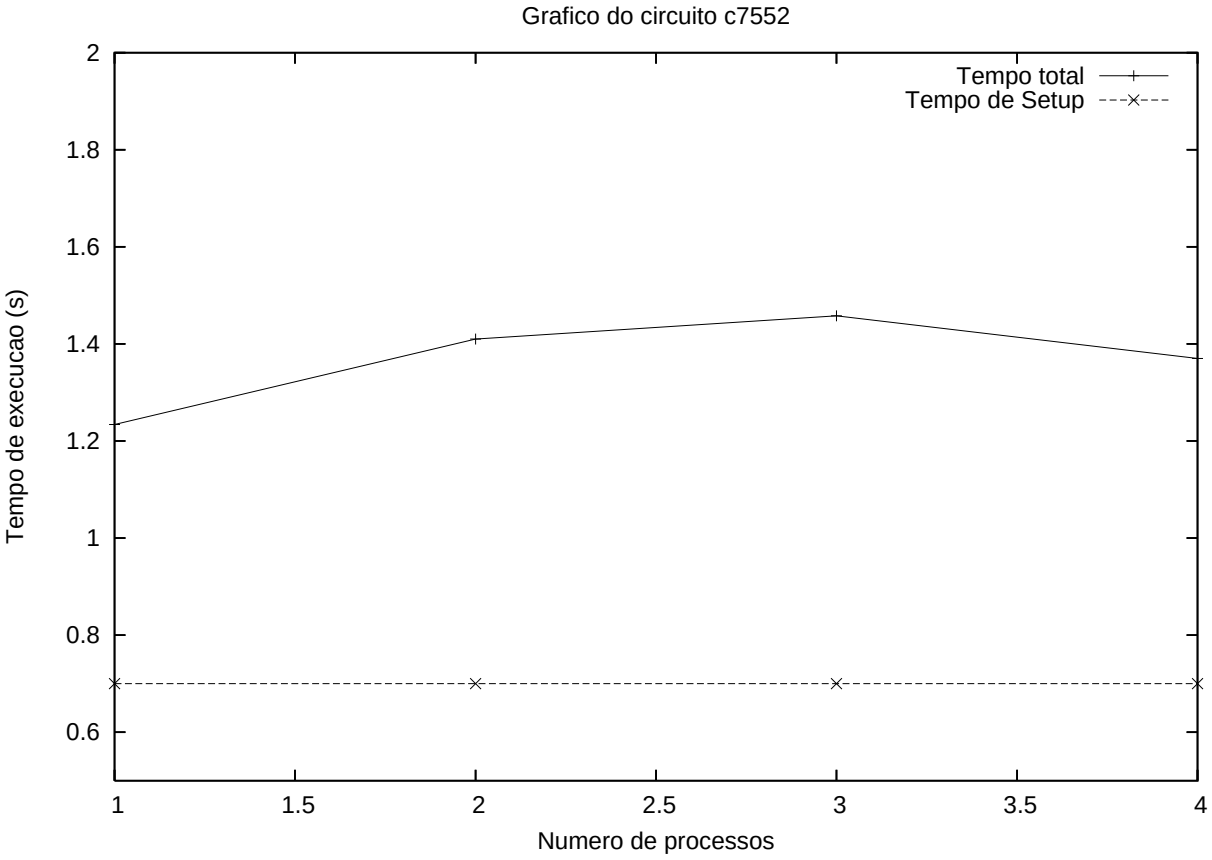

Figura 5.3: Tempos de Execução para c7552

Podemos ver o circuito na figura 5.4.

O circuito s13207.1 é um dos casos onde temos pouca informação sobre sua estrutura. Este problema se repete com praticamente todos os circuitos sequenciais do benchmark ISCAS89, pois a página que continha os benchmarks originalmente não

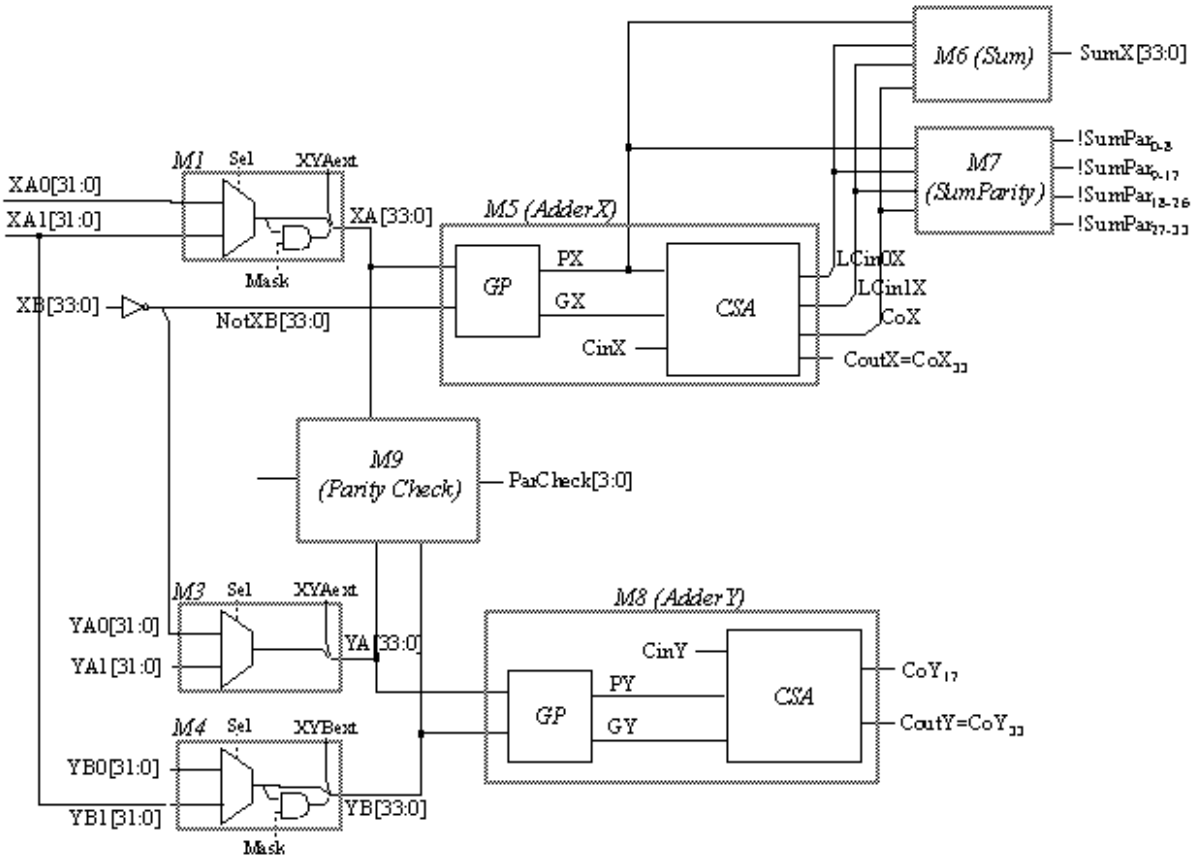

Figura 5.4: Arquitetura do circuito c7552

existe mais. É um problema de fácil resolução, dados os tempos de execução, mas ao contrário do circuito c7552, foi possível obter algum speedup, como pode ser visto no gráfico5.5. Estruturalmente, ele é composto por:

- 62 entradas primárias
- 152 saídas primárias
- 8131 portas lógicas (1114 ANDs, 849 NANDs, 512 ORs, 98 NORs, 5378 inversores)
- 638 flip-flops D

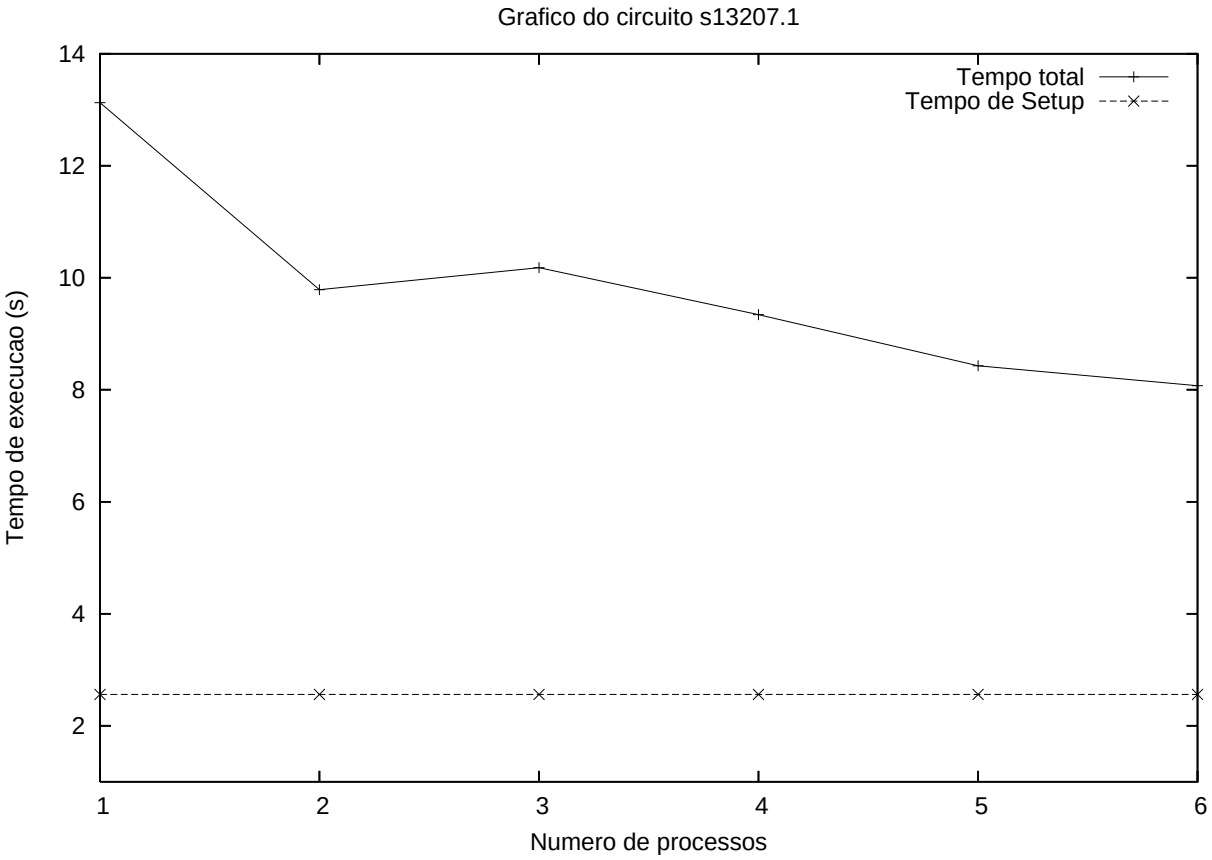

Figura 5.5: Tempos de Execução para s13207

O circuito b19 é uma variação do circuito b18, mas já representa um problema um pouco mais difícil de ser resolvido. Este é o caso onde a utilização de paralelismo mais se justifica, devido ao bom speedup obtido, como pode ser visto no gráfico 5.6. Este gráfico começa com três processadores, porque com apenas um ou dois ocorreu falta de memória, ou seja, não foi possível resolver o problema. Este foi o maior circuito

utilizado nos testes, como pode ser visto pelo número de portas lógicas que o mesmo possui:

- 24 entradas primárias
- 30 saídas primárias
- 190213 portas lógicas (22774 ANDs, 158088 NANDs, 1768 ORs, 887 NORs, 41107 inversores)
- 6642 flip-flops D

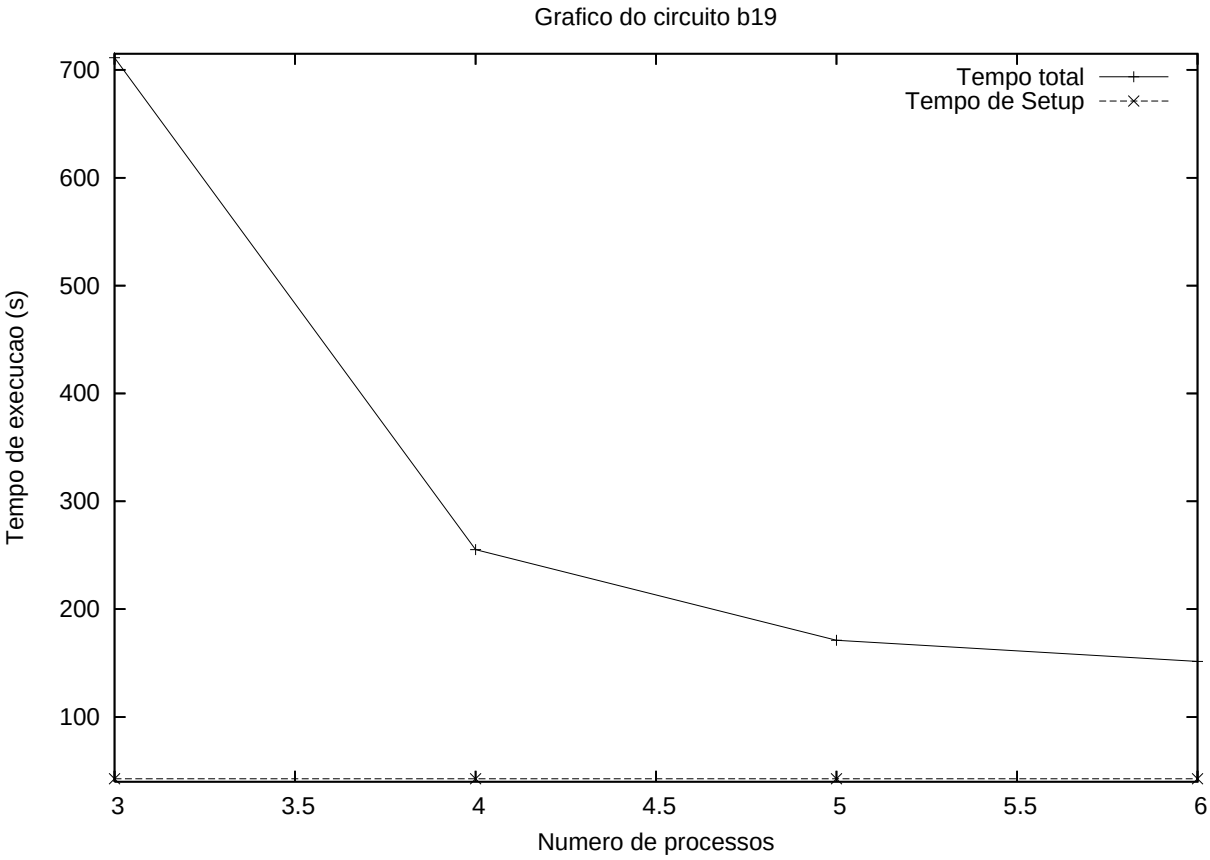

Figura 5.6: Tempos de Execução para b19

O gráfico 5.7 mostra os tempos de execução para o circuito s38584, que faz parte do benchmark ISCAS89. Este foi outro caso onde não se conseguiu executar o problema com 1 ou dois processadores, pois ocorreu falta de memória, só rodando com 3 ou mais. Ele possui as seguintes características estruturais:

- 38 entradas primárias

- 304 saídas primárias
- 19253 portas lógicas (5516 ANDs, 2126 NANDs, 2621 ORs, 1185 NORs, 7805 inversores)
- 1426 flip-flops D

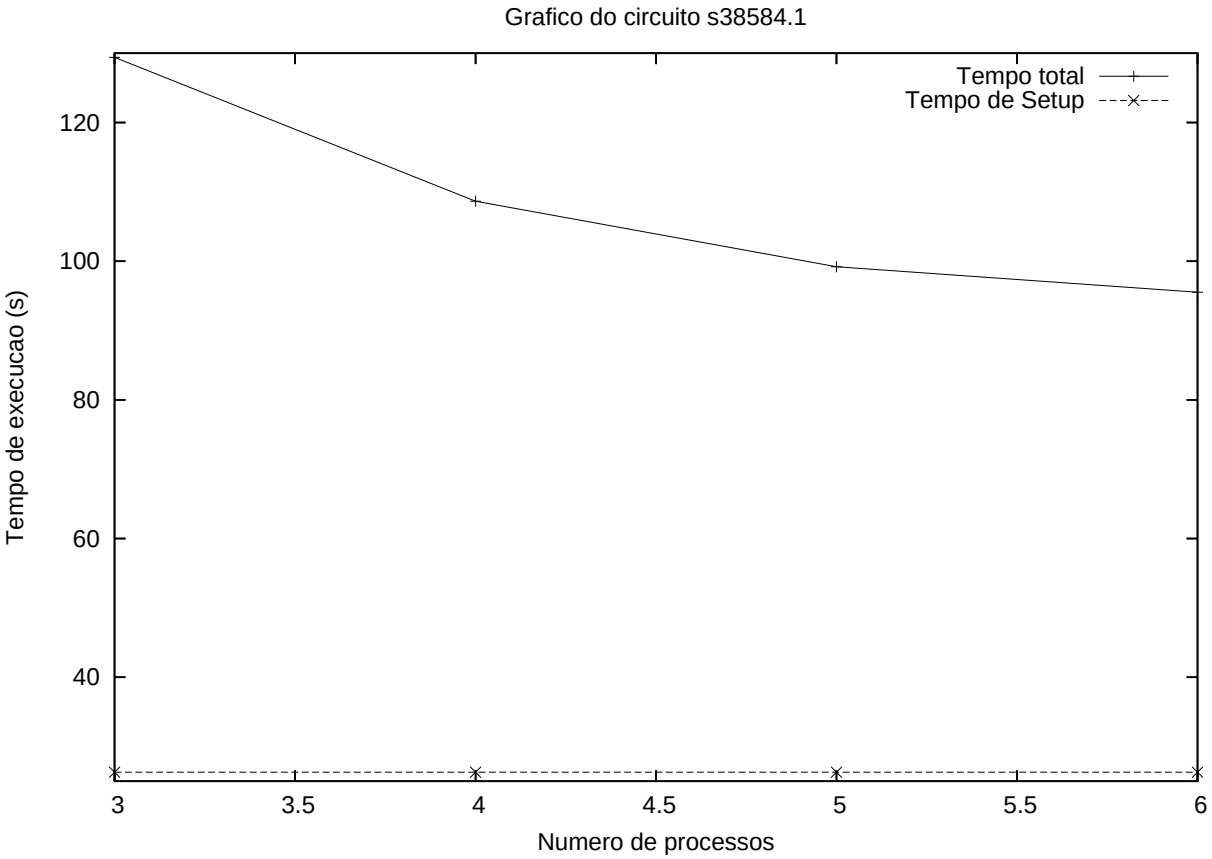

Figura 5.7: Tempos de Execução para s38584

Um detalhe importante ao ser levado em conta ao analisar os resultados: como o conjunto de máquinas é heterogêneo, o tempo de execução pode ser limitado pela máquina mais lenta, pois ao enviar os problemas para os diferentes processos, não estimamos o tamanho dos mesmos e o poder de processamento do nodo correspondente. Em alguns casos, se adicionarmos uma máquina mais lenta e com menos memória, os tempos de execução podem piorar bastante. Fizemos um teste adicionando um Pentium 4 com 512MB de RAM. Quando ele entrava no conjunto de máquinas, praticamente todos os tempos de execução sofreram uma piora sensível, pois esta máquina se tornou o gargalo do sistema. Um trabalho futuro é melhorar a distribuição de carga, ao escalonar os problemas levando em conta o tamanho deles e o poder de processamento

dos nodos utilizados. Obviamente, estimar a dificuldade de resolução não é trivial, pelo fato dos problemas serem NP-completo, mas pode-se utilizar o número de cláusulas e variáveis contidas no problema como métricas na estimativa.

Na tabela 5.1, podemos ver o speedup máximo obtido nos testes, e o tempo gasto no setup de cada um deles. Apenas o teste c7552 não foi listado, pois não houve melhora no tempo de execução. A tabela contém apenas os melhores tempos de execução, informando com quantos processadores este tempo foi obtido. Vale lembrar que ao nos referirmos a processadores, estamos falando de número de processadores físicos existentes e não número de máquinas utilizadas, pois algumas das máquinas possuem processadores dual-core.

Circuito	Número de Processadores	Speedup	Tempo de Setup (s)	Tempo Total (s)
b18	7	2,04	17,66	52,31
s13207.1	7	1,62	2,56	8,07
b19	7	4,69	42,79	151,51
s38584.1	7	1,35	26,31	95,51

Tabela 5.1: Tempos de execução e speedup

Em todos os casos, os melhores tempos foram obtidos utilizando-se o número máximo de processadores disponíveis: 6 alocados para os resolvedores SAT e 1 para o processo mestre, ou seja, o EquivalenceChecker em si. O maior speedup obtido foi de 4,69, no teste cujo circuito era o maior dentre todos os benchmarks, o b19. Ficaram pendentes testes com outros conjuntos de circuitos, circuitos diferentes entre si e o aprendizado de cláusulas SAT com desdobramento de circuitos sequenciais. Apesar disto, consideramos os exemplos utilizados significativos, pois conseguimos obter uma melhora no tempo de resolução dos problemas, e além disso os exemplos são amplamente utilizados na literatura relacionada.

Capítulo 6

Conclusões e Trabalhos Futuros

A verificação de equivalência é um dos componentes essenciais na metodologia de verificação de projetos de circuitos integrados. É uma das técnicas mais utilizadas atualmente pela indústria de EDA para a comparação de duas descrições em diferentes níveis de abstração ou otimizadas de um mesmo circuito. Apesar dos avanços obtidos nos últimos anos nas técnicas de verificação de equivalência, a complexidade inerente aos problemas desta classe, e a crescente complexidade dos circuitos digitais ainda motivam pesquisas nesta área.

Neste trabalho foi proposto, desenvolvido e analisado um ambiente de verificação de equivalência que se utiliza de heurísticas de particionamento e aproveitamento de cláusulas de conflito SAT para processamento paralelo do processo de verificação. A metodologia proposta foi validada a partir da verificação de circuitos existentes nos benchmarks ISCAS85, ISCAS89 e ITC99. Estes circuitos demonstram que as técnicas de particionamento e aprendizado de cláusulas são corretas e eficientes, permitindo a verificação de instâncias de problemas onde antes não era possível se aplicar métodos formais de verificação de equivalência.

Como trabalhos futuros, pretende-se integrar a ferramenta atual a um parser Verilog/VHDL, para automatizar a tarefa de síntese do circuito, além de aumentar consideravelmente a flexibilidade de utilização da ferramenta. Outro trabalho interessante é adicionar suporte à simulação do circuito. Neste caso poderiámos realizar uma simulação randômica para encontrar pontos interessantes de serem comparados nos circuitos, ou até mesmo para provar a não equivalência dos mesmos em casos mais simples. Verificar qual a combinação ideal de ciclos/largura para diversas classes de circuitos é outra sugestão. Um ponto interessante é com relação às cláusulas de conflito. Atualmente diversas expressões booleanas de interesse estão sendo identificadas a partir das cláusulas de conflito geradas. No entanto, os resultados obtidos utilizaram apenas as

cláusulas de conflito cujas expressões booleanas são de igualdade e diferença (inversores e buffers), descartando os outros tipos previstos. Aproveitar as outras cláusulas pode melhorar os resultados do aprendizado. Outro trabalho que poderia ser realizado é verificar a viabilidade de se estender a ferramenta para utilizar outros métodos de verificação, como BDDs por exemplo, ou até mesmo SAT e BDDs de maneira intercalada. Pode-se também buscar integrar a ferramenta com outros tipos de resolvedores SAT.

Com relação ao resolvedor paralelo, algumas melhorias podem ser implementadas. Atualmente o processo mestre não resolve nenhuma instância SAT, ele apenas fica esperando os resultados dos problemas que são resolvidos pelos processos escravos. Isto pode ser alterado, pois ele fica parado a maior parte do tempo após ler os circuitos e fazer a divisão dos mesmos e montar os problemas. Especialmente em problemas maiores, o aproveitamento do processo mestre pode resultar em ganhos expressivos.

No caso do EquivalenceChecker, é possível tornar a leitura e particionamento dos circuitos uma tarefa paralela. Os arquivos de entrada contendo os circuitos podem ser lidos em paralelo, além disso a divisão por largura (a montagem dos cones de lógica dos bits de saída) pode ser feita em paralelo também. No entanto, não seria necessário utilizar MPI para essas tarefas, pois neste caso é muito mais rápido e eficiente utilizar um sistema de threads comum.

Apêndice A

Apendice 1

Neste apêndice iremos descrever o formato de entrada dos circuitos, o formato de entrada do resolvedor SAT e o algoritmo de conversão de portas lógicas em cláusulas CNF.

A.1 Formato de Entrada dos Circuitos

O formato de entrada de circuitos utilizados é chamado bench, e foi definido pelo benchmark ISCAS. Este formato descreve o circuito em nível de portas lógicas, ou seja, são circuitos sintetizados. Este é um nível de abstração abaixo do RTL, que pode ser escrito em Verilog ou VHDL. Os motivos que levaram a escrever um parser para este formato são:

- A implementação ou adaptação de um parser para VHDL ou Verilog está fora do escopo deste trabalho, e aumentaria o tempo de implementação consideralvemente
- O benchmark ISCAS é muito usado na literatura
- O formato bench é de fácil conversão para a estrutura interna do circuito

Vamos a descrição do formato. Inicialmente temos a lista das entradas primárias do circuito. Cada entrada é descrita como

```
INPUT(a)
```

e assim sucessivamente. Logo após temos a lista de saídas primárias, descrita de maneira similar:

```
OUTPUT(b)
```

Após listar todas as saídas primárias, vem a descrição do circuito em si, em linhas do tipo:

```
b = GATETYPE(a, c, d)
```

Neste exemplo, a é a saída da porta lógica, GATETYPE é o tipo da porta lógica (BUF, AND, NAND, XOR, OR, NOR, MUX, DFF, NOT), e os sinais a, c, d são as entradas da porta. Comentários são linhas que começam com o caractere #.

Um exemplo de circuito descrito em bench pode ser visto logo abaixo:

```
#Comentario
INPUT(1)
INPUT(2)
INPUT(3)
INPUT(6)
INPUT(7)

OUTPUT(22)
OUTPUT(23)

10 = NAND(1, 3)
11 = NAND(3, 6)
16 = NAND(2, 11)
19 = NAND(11, 7)
22 = NAND(10, 16)
23 = NAND(16, 19)
```

A.2 Conversão de Portas Lógicas em CNF

Nesta seção iremos descrever em detalhes os elementos lógicos que foram implementados e como foi feita a sua descrição em CNF. Tomemos como exemplo uma porta lógica AND. Começamos por sua fórmula básica:

$Z = X \wedge Y$

Como a fórmula $P = Q$ é logicamente equivalente a $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$, podemos transformar a fórmula inicial em:

$(Z \Rightarrow (X \wedge Y)) \wedge ((X \wedge Y) \Rightarrow Z)$

Agora transformamos todas as implicações em disjunções ao utilizar o fato de que $P \Rightarrow Q$ é equivalente a $\neg P \vee Q$ para chegar a fórmula

$(\neg Z \vee X) \wedge (\neg Z \vee Y) \wedge (\neg X \vee \neg Y \vee Z)$

A fórmula acima é igual a um se os valores das variáveis forem consistentes com a tabela da verdade de uma porta AND. Utilizando o raciocínio acima, podemos encontrar as fórmulas para os outros elementos lógicos definidos. Na figura A.1, podemos ver os elementos e suas fórmulas CNF correspondentes.

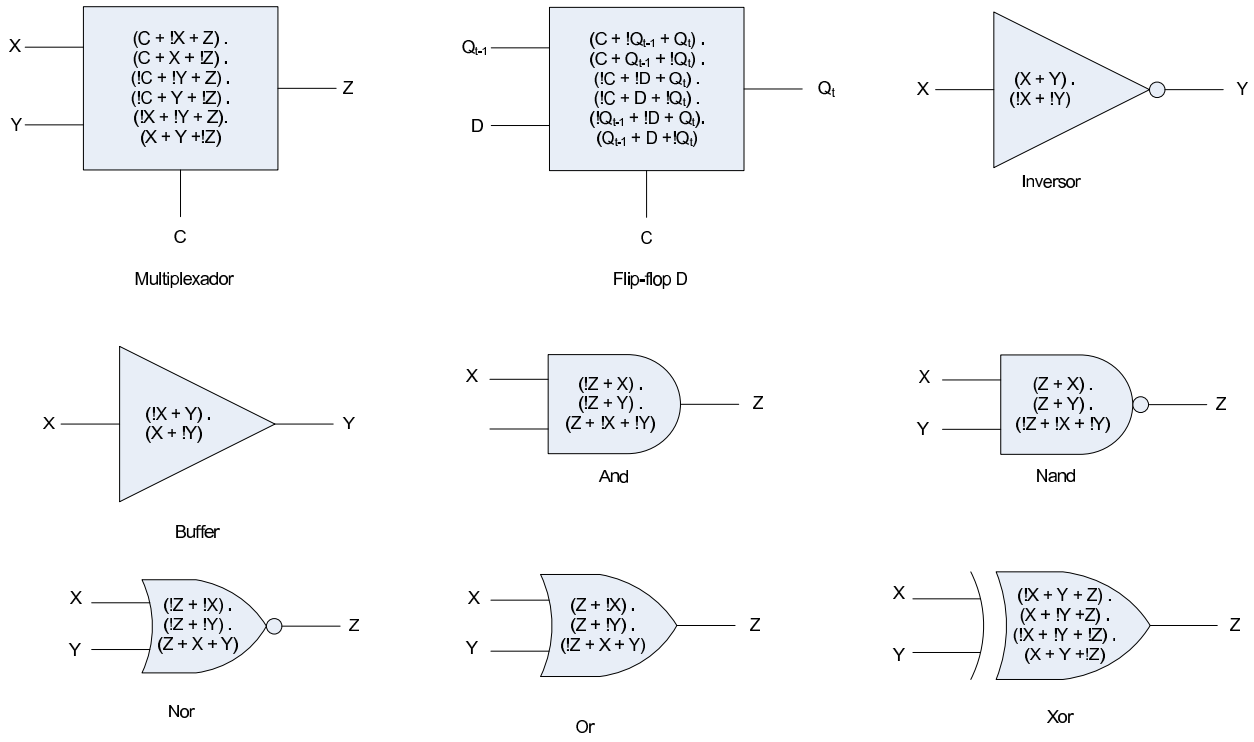

Figura A.1: Portas Lógicas e Suas Fórmulas CNF Correspondentes

A.3 O Formato DIMACS

Os resolvedores SAT atuais em sua maioria utilizam como formato de entrada de dados o padrão DIMACS[64]. Este formato foi criado com o objetivo de diminuir o esforço requerido para se testar e comparar algoritmos e heurísticas, provendo um conjunto de entradas padronizadas. Ele define basicamente dois formatos para problemas de satisfabilidade. Vamos abordar apenas o formato CNF, que é o formato utilizado neste trabalho.

Um problema de satisfabilidade em forma normal conjuntiva (CNF), consiste de

uma conjunção de cláusulas, onde uma cláusula é uma disjunção de um número de variáveis ou suas negações. Considerando que x_i representa variáveis que possam assumir apenas valores verdadeiro ou falso, um exemplo de fórmula no formato CNF seria:

$$(x_1 \vee x_3 \vee \neg x_4) \wedge (x_4) \wedge (x_2 \vee \neg x_3) \tag{A.1}$$

Dado um conjunto de cláusulas $C_1, C_2, ..., C_m$ sobre as variáveis $x_1, x_2, ..., x_n$, o problema de satisfabilidade consistem em determinar se a fórmula:

$$C_1 \wedge C_2 \wedge ... \wedge C_m \tag{A.2}$$

é satisfazível, ou seja, se existe um assinalamento de valores para todas as variáveis que torne a fórmula acima verdadeira. Isto implica que cada C_j seja verdadeiro.

Para representar instâncias de tais problemas, definimos um arquivo ASCII que contém duas seções: o preambulo e as cláusulas. O preâmbulo contém informações sobre a instância do problema. Esta informação é contida em linhas. Cada linha começa com um único caractere, seguido de espaço, que determina o tipo da linha. Os tipos existentes são:

- Comentarios. Os comentários contém informações úteis ao usuário, e são ignorados pelos programas. Eles normalmente aparecem no começo do preambulo. Cada linha de comentário começa com um caractere minúsculo c.

```
c Este é um exemplo de comentário.
```

- Linha de descrição do problema. Cada arquivo deve conter apenas uma linha deste tipo. Ela deve aparecer antes de qualquer declaração de cláusulas. Para instâncias CNF, a linha tem o seguinte formato:

```
p FORMAT VARIABLES CLAUSES
```

No nosso caso, no campo FORMAT, utilizamos a palavra cnf. O campo VARIABLES indica o número de variáveis que o problema contém, e o campo CLAUSES o número de cláusulas. Esta é a última linha do preâmbulo.

Logo após a linha do problema, começam as cláusulas. Assume-se que as variáveis são númeradas de 1 a n. Não é necessário que todas as variáveis apareçam em uma instância. Cada cláusula é uma sequência de números, separados por espaço. A versão

não negada de uma variável i é representada por i, e a versão negada por -i. Cada cláusula é terminada com o valor 0. Isto permite que as cláusulas ocupem várias linhas, ao contrário dos formatos onde o caractere de nova linha indica o começo de uma nova cláusula.

O arquivo DIMACS do conjunto de cláusulas $(x_1 \vee x_3 \vee \neg x_4) \wedge (x_4) \wedge (x_2 \vee \neg x_3)$, é:

```
c Exemplo de arquivo CNF
p cnf 4 3
1 3 -4 0
4 0
2 -3
```

Referências Bibliográficas

[1] T. Kirkland and M. R. Mercer. A topological search algorithm for atpg. In *DAC '87: Proceedings of the 24th ACM/IEEE conference on Design automation*, pages 502–508, New York, NY, USA, 1987. ACM Press.

[2] D. MacMillen, M. Butts, R. Camposano, D. Hill, and T. W. Williams. An industrial view of electronic design automation. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, pages 1428–1447, 2000.

[3] Harry Foster, Adam C. Krolnik, and David J. Lacey. *Assertion-Based Design*. Kluwer Academic Publishers, 2004.

[4] Rolf Drechsler. *Advanced Formal Verification*. Kluwer Academic Publishers, 2004.

[5] Bob Bentley, Kurt Baty, Kevin Normoyle, Makoto Ishii, and Einat Yogev. Verification: What works and what doesn't. In *Proceedings of Design Automation Conference*, 2004.

[6] B. Bentley. Validating the intel pentium 4 microprocessor. In *Procedings of Design Automation Conference*, pages 244–248, 2001.

[7] K. L. McMillan. Interpolation and sat based model checking. In *Proceedings of the 15th Conference on Computer Aided Verification*, pages 250–264. Springer-Verlag, 2003.

[8] B. Beizer. The pentium bug - an industry watershed. *Testing Techniques Newsletter*, Sep 1995.

[9] Daniel E. Atkins. Higher radix division using estimates of the divisor and partial remainder. *IEEE Transactions on Computers*, pages 925–934, 1968.

[10] Algirdas Avizienis and Yutao He. *Dependable Computing for Critical Applications*, volume 12, chapter 1, pages 3–23. IEEE Computer Society, 1999.

[11] David Van Campenhout, Trevor Mudge, and John P. Hayes. Collection and analysis of microprocessors design errors. *IEEE Design & Test of Computers*, pages 51–60, 2000.

[12] Intel Corp. Intel pentium processor specification update. Technical report, Jan.

[13] Intel Corp. Intel pentium processor specification update. Technical report, Jul.

[14] Intel Corp. Intel pentium processor specification update. Technical report, Nov.

[15] Intel Corp. Intel pentium processor specification update. Technical report, Aug.

[16] AMD. Amd athlon processor model 10 revision guide (c). Technical report, Oct 2003.

[17] AMD. Amd athlon processor model 6 revision guide (g). Technical report, Oct 2003.

[18] AMD. Amd athlon processor model 8 revision guide (e). Technical report, Oct 2003.

[19] AMD. Revision guide for amd athlon 64 and amd opteron processors. Technical report, Jun 2004.

[20] Freescale Semiconductors. Mask set errata for mc9s12dp512 microcontroller, mask 0l00m. Technical report, Aug 2004.

[21] Paul Molitor and Janett Mohnke. *Equivalence Checking of Digital Circuits*. Springer, Mar 2004.

[22] S.Y. Huang and K.T. Cheng. *Formal Equivalence Checking and Design Debugging*. Kluwer, 1998.

[23] Zaher Andraus Valeria Bertacco Trevor Mudge Stephen Plaza, Ian Kountanis. Advances and insights into parallel sat solving. In *International Workshop on Logic Synthesis*, January 2006.

[24] Márcia Carolina Marra de Oliveira. Um núcleo inteligente para processamento distribuído de resolvedores sat em verificação por equivalência. Master's thesis, Universidade Federal de Minas Gerais, 2006.

[25] Eric W. Weisstein. Conjunctive normal form. http://mathworld.wolfram.com/ConjunctiveNormalForm.html.

[26] IEEE. Ieee standard vhdl language reference manual. Technical report, 2002. IEEE Std 1076-2002.

[27] IEEE. Ieee standard for verilog hardware description language. Technical report, 2006. IEEE Std 1800-2005.

[28] Guido Arnout. Systemc standard. In *Proceedings of the ASP-DAC 2000*, 2000.

[29] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1983.

[30] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, pages 677–691.

[31] R. E. Bryant. On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, pages 205–213.

[32] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, pages 293–318.

[33] M. Fujita, H. Fujisawa, and N. Nakato. Evaluation and improvements of boolean comparison method based on binary decision diagrams. In *Proceedings of the IEEE International Conference on Computer Aided Design*. IEEE Computer Society Press, 1998.

[34] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincenteli. Logic verification using binary decision diagrams in a logic synthesis environment. In *International Conference on Computer-Aided Design*, pages 6–9, 1988.

[35] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *International Conference on Computer Aided Design*, 1993.

[36] U. Hustadt. An empirical analysis of modal theorem provers.

[37] Tracy Larrabee. Test Pattern Generation Using Boolean Satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):6–22, 1992.

[38] Olivier Coudert. On solving covering problems. In *Design Automation Conference*, pages 197–202, 1996.

[39] F. Ferrandi, A. Macii, E. Macii, M. Poncino, R. Scarsi, and F. Somenzi. Symbolic algorithms for layout-oriented synthesis of pass transistor logic circuits. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 235–241, New York, NY, USA, 1998. ACM Press.

[40] Paulo F. Flores, Horácio C. Neto, and João P. Marques-Silva. An exact solution to the minimum size test pattern problem. *ACM Transactions on Design Automation of Electronic Systems.*, 6(4):629–644, 2001.

[41] Gary D. Hachtel and Fabio Somenzi. *Logic Synthesis and Verification Algorithms.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[42] Roberto J. Jr. Bayardo and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208, Providence, Rhode Island, 1997.

[43] Hantao Zhang. SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI*, pages 272–275, 1997.

[44] Peter Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Technical Report MPI–I–95–2–003, Saarbrücken, 1995.

[45] J. P Roth. Diagnosis of automata failures: A calculus and a method. *IBM Journal of Research and Development*, 1966.

[46] D. Bhattacharya and J. P. Hayes. *Hierarchical Modeling for VLSI Circuit Testing*. Boston: Kluwer, 1990.

[47] H. Fujiwara and T Shimono. On the acceleration of test generation algorlthms. *Twenty-Fifth International Symposium on Fault-Tolerant Computing*, 1995.

[48] M. Hansen, Yalcin H., and Hayes J. Unveiling the iscas-85 benchmarks: A case study in reverse engineering. In *IEEE Design & Test of Computers*, volume 16, pages 72–80, 1999.

[49] Cormen T., Leiserson C., Rivest R., and Stein C. *Introduction to Algorithms*. MIT Press, 2001.

[50] Princeton University. Sat research group. http://ee.princeton.edu/ chaff/zchaff.php.

[51] Stroustrup B. *The C++ Programming Language*. Addison-Wesley.

[52] Inc. Linux Kernel Organization. The linux kernel archives. http://www.kernel.org/.

[53] Novell Inc. Suse linux. http://www.novell.com/linux/.

[54] Red Hat Inc. Fedora core linux. http://fedoraproject.org.

[55] Free Software Foundation. Gcc, the gnu compiler collection. http://gcc.gnu.org/.

[56] G. Burns, R. Daoud, and J Vaigl. Lam: An open cluster environment for mpi. In *Supercomputing Symposium*, 1994.

[57] Indiana University. Lam/mpi parallel computing team. Disponível em http://www.lam-mpi.org/.

[58] Sat competitions. http://www.satcompetition.org/.

[59] NCSU. Benchmark circuits. http://www.fm.vslib.cz/ kes/asic/iscas/.

[60] F. Corno, Reorda M., and Squillero J. Rt-level itc 99 benchmarks and first atpg results. In *IEEE Design & Test of Computers*, pages 44–53, 2000.

[61] Local area network. http://en.wikipedia.org/wiki/Local_area_network.

[62] The ssh protocol. http://www.snailbook.com/protocols.html.

[63] R. Arora and M. S. Hsiao. Enhancing SAT Based Equivalence Checking with Static Logic Implications. *IEEE*, 2003.

[64] DIMACS Challenge. Satisfiability suggested format, 1993.