UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

PETERSON WILGES

Verificador temporal de propriedades em tempo de execução implementado em VHDL

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Renato Perez Ribas

Coorientador: Prof. Dr. Axel Braun

Porto Alegre
2014

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Graduação: Prof. Sérgio Roberto Kieling Franco
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador do Curso de Engenharia de Computação: Prof. Marcelo Götz
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Em primeiro lugar, agradeço a minha namorada Daiane Mendes dos Santos, por estar comigo durante praticamente toda a minha caminhada universitária e pela paciência durante o desenvolvimento desta trabalho. Em seguida, agradeço a meus familiares, principalmente minha mãe que tornou a realização desse sonho possível, quando depositou sua confiança de que eu poderia entrar em uma universidade federal.

Agradeço meus irmãos, Alexandre Wilges, Beatriz Wilges e Michael Wilges e todos meus amigos mais próximos que durante toda essa caminhada compartilharam momentos difíceis e outros muitos, divertidos e de felicidade, que me ajudaram a tornar essa caminhada melhor.

Por último, gostaria de agradecer aqueles que tiveram grande importância no desenvolvimento deste trabalho, começando pelo prof. Doutor Axel Brown da universidade Tübingen, que acompanhou o desenvolvimento do projeto, a amiga e colega Yumin Zhou também da universidade Tübingen que ajudou no desenvolvimento dos testes finais deste trabalho, o amigo e colega Mateus Felipin Dalepiane pelas produtivas conversar sobre o trabalho e finalmente, o prof. Doutor Renato Ribas, que me ajudou a fazer os ajustes finais e escrita do trabalho para apresentação.

"*You see all I need is a whisper*
In a world that only shouts"

RESUMO

A verificação de projetos digitais é essencial para garantir o correto funcionamento e aumentar a confiabilidade de um sistema. Este trabalho visa fazer a verificação de sistemas reativos através de propriedades formais usando lógica temporal linear finita (FLTL) a fim de aumentar a confiabilidade de circuitos. Muitas técnicas têm sido desenvolvidas para a verificação em tempo de execução. A proposta deste trabalho é o desenvolvimento de um circuito verificador para checar a validade propriedades temporais de sistemas através da análise de sinais Booleanos. Neste sentido, um compilador será desenvolvido em linguagem C++ para criar instruções que possam ser interpretadas em um circuito verificador que será desenvolvido em VHDL. Tais instruções devem ser gravadas na memória RAM do FPGA que será o alvo para o desenvolvimento do circuito verificador HDL. O verificador será rápido o suficiente para checar as propriedades temporais de um dispositivo no exato ciclo de relógio especificado pela fórmula FLTL.

Palavras-chave: Verificação formal. Detecção de falha. Confiabilidade. Segurança.

Temporal checker of properties in execution time implemented in VHDL

ABSTRACT

Verification of digital designs is essential to ensure the correctness and to improve the reliability of systems. The checker developed makes the verification of reactive systems through formal properties using finite linear temporal logic (FLTL) in order to increase the circuits' reliability. Many techniques have been proposed in order to make verification in execution time. This approach is to make verification in VHDL to check the validity of temporal properties of systems by analyzing Booleans signals. In this way, a compiler will be developed using C++ to create instructions to be interpreted in a HDL checker circuit. This instructions should be stored in the RAM memory of the FPGA used as target architecture for our HDL developed. The checker will be fast enough to check the temporal properties of the device in the exact clock cycle specified by the FLTL formula.

Keywords: Formal Verification. Fault detection. Reliability. Safety.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

UFRGS	Universidade Federal do Rio Grande do Sul
LTL	Linear temporal logic
FLTL	Finite linear temporal logic
VHDL	VHSIC Hardware Description Language
DUT	Device under test
DUV	Device under verification
DSV	Dispositivo sobre verificação
CUT	Circuit under test
CUV	Circuit under verification
FSM	Finite state machine

SUMÁRIO

1 INTRODUÇÃO

Garantir corretude é uma das mais importantes tarefas no processo de desenvolvimento de um circuito digital. Por esta razão, durante a concepção e durante o seu tempo de vida, um sistema digital é testado e diagnosticado inúmeras vezes. Basicamente, ao projetar um produto devemos fazer suas especificações, implementar, fabricar e testar. Ao realizarmos algum teste, caso houver falha, sabemos que algo no projeto está errado e precisa ser corrigido. O motivo da falha pode ser erro na especificação funcional, na implementação, na interligação dos componentes, falha no processo de fabricação, ou até mesmo, erro no teste aplicado. Um bom processo de teste deve detectar produtos com defeito, o mais rápido possível, quando manifestado, seja por questões financeiras, funcionais ou de segurança.

Uma verificação pode acontecer em etapas distintas do processo de desenvolvimento. Podemos dizer que um projeto geralmente inicia com um modelo abstrato, descrevendo sua principal funcionalidade, e então sofre sucessivas melhorias até chegar ao produto final, por exemplo, um sistema de tempo real, onde em um primeiro momento devemos ter a resposta em um determinado número de ciclos. Depois de sucessivas melhorias até sua fabricação e utilização ele deve sempre garantir entregar a resposta dentro do tempo, ou caso for possível, antes do tempo especificado. Dentre os benefícios de verificação em sistemas digitais podemos citar: qualidade, segurança e economia; esta, porque produtos fabricados com falha ou mesmo distribuídos com falha geram um alto custo de manutenção. Todavia, para conseguir um produto de alta qualidade, segurança, bem como garantir que o funcionamento está de acordo com as especificações, vários tipos de testes podem ser aplicados.

Geralmente a maioria das verificações são feitas através de simulação usando *testbench*. Assim, um vetor ou uma sequência de vetores de teste é gerado e colocado na entrada do sistema que produzirá diferentes saídas na ausência e na presença de falhas, tornando assim, uma falha perceptível. Entretanto, somente verificação por carga de *testbenchs* não cobre todos os possíveis casos e pode deixar escapar erros graves e importantes. Dessa forma, só podemos provar que o sistema esta funcionando, sem erros, para um determinado conjunto de testes testado (DIJKSTRA, 1972), dentro das condições em que o circuito é testado. No entanto, quando confiabilidade do circuito é um parâmetro essencial na concepção de um projeto, outras técnicas devem ser utilizadas. Neste trabalho, demonstraremos uma forma de auto verificação de propriedades temporais de circuitos para detecção de falhas durante a operação normal do sistema. Essa técnica de determinação da corretude também é conhecida como *In-System Testing ou Burn-in testing*. No presente

trabalho, a verificação concorrente com o circuito é feita usando propriedades formais. Assim sendo, conseguimos aumentar consideravelmente a confiabilidade de um determinado circuito.

1.1 Contexto Específico

Sistemas, no contexto em análise, são considerados aqueles que são reativos, ou seja, são caracterizados por uma contínua interação com seu ambiente (HAREL & PNUELLI, 1985). Neste sentido, os sistemas reativos possuem como entradas, variáveis do ambiente, que podem ser botões, sensores ou mesmo variáveis de limite que quando acionados devem iniciar uma nova tarefa, interagindo de forma dinâmica com o sistema. Geralmente esses sistemas precisam responder em um espaço de tempo limitado. Tipicamente, exemplos de sistemas reativos são sistemas de controle de trafego aéreo, sistemas de tempo real e protocolos de comunicação. Esses tipos de sistemas possuem requisitos de segurança e em algumas situações, tempo crítico onde eventos, por vezes, devem ocorrer em tempo determinado.

Recentemente, métodos formais como verificação de equivalência e verificação de modelos (*model checking*) tem encontrado espaço em laboratórios de pesquisa (RUF et al., 2001). Esses métodos especificam os sistemas de forma precisa e não ambígua, dando suporte a uma posterior análise semântica do mesmo. Validação baseada em métodos formais tem um grande potencial e é bem aceita em verificação de hardware e verificação de protocolos de comunicação, por exemplo(RUF et al., 2001). Para o usuário expressar uma ou mais propriedades formais este trabalho utiliza a lógica temporal chamada FLTL (*finite linear temporal logic*) , que é uma variante da lógica LTL (*finite linear temporal logic*) (RUF et al., 2001). Esta, por sua vez, só interpreta propriedades sobre quantidades de tempo em ciclos não determinados ou infinitos. Já, por outro lado, na FLTL, o usuário verificador pode quantificar com precisão o intervalo de tempo em número de ciclos que uma determinada atividade deve acontecer. Para citar um exemplo, imagine um sistema onde um protocolo de comunicação precisa responder a uma mensagem recebida em um espaço de ciclos de relógio bem definido. Esse sistema pode ser modelado usando a lógica FLTL, onde o verificador pode quantificar o número de ciclos em que o sistema deve responder. Ao revés, usando somente LTL a única modelagem possível seria para analisar se houve uma resposta em algum momento indeterminado do futuro.

A abordagem utilizada é para qualquer complexidade de sistema, mas sabendo que o verificador desenvolvido pode apenas checar um número limitado de propriedades em cada execução. Isso se deve ao fato de que quanto mais propriedades sequências forem checadas, menor será a frequência que o sistema sobre análise vai rodar. Propriedades podem ser checadas de forma concorrente umas com as outras, mas isso implica em uma maior área para o circuito verificador. Dessa maneira, fica claro para o projetista que de acordo com as restrições do sistema, seja de área de circuito, consumo ou desempenho, que ele precisa encontrar o balanço ideal para quais propriedades serão checadas e de que maneira isso será realizado, seja sequencialmente ou paralelamente umas as outras, para conseguir atingir a confiabilidade desejada.

1.2 Motivação

Atualmente, dispositivos digitais estão cada vez mais presentes em nosso cotidiano. Para esses sistemas funcionarem de acordo com suas especificações com alta confiabilidade e disponibilidade, é preciso testá-los e diagnosticá-los de maneira rápida e eficiente. Uma maneira de conseguir esses requisitos é a implementação de verificação durante a execução normal do circuito. Outro fator bastante requisitado em sistemas confiáveis, é a verificação formal.

Com a implementação de um circuito construído para verificação formal em tempo de execução (*build-in*), o custo de hardware adicionado será balanceado pelas vantagens em termos de confiabilidade e redução de custos de manutenção.

1.3 Proposta e Objetivos do Trabalho

O trabalho aqui desenvolvido visa trazer uma proposta de verificação/validação funcional aprimorada através de um analisador temporal desenvolvido em VHDL, que faz a verificação de propriedades formais durante qualquer etapa de simulação/execução de um *hardware*. Através da verificação formal, o usuário pode especificar propriedades do sistema que serão analisadas pelo verificador. Também é possível especificar com precisão em número de ciclos de relógio que um evento ou um conjunto de eventos devem ou não ocorrer. As propriedades do sistema são escritas na memória RAM do dispositivo, através de

instruções que serão interpretadas pelo verificador, testando assim a corretude de um sistemas de forma concorrente. Dessa forma, as propriedades lógicas verificadas podem ser alteradas de forma dinâmica pelo projetista. Para a transformação de uma fórmula formal em instruções, o presente trabalho propõe a implementação de um compilador que gera o conjunto de instruções necessárias para verificação de propriedades.

Um dos principais objetivos do trabalho é o desenvolvimento do verificador de forma tão simples e rápida quanto possível. Dessa maneira, o intrusividade do analisador temporal deve ser mínima, possibilitando ao circuito sobre testes, funcionar normalmente sem sofrer grandes prejuízos no seu desempenho. Espera-se ainda, que a área adicionada pelo verificador no circuito seja mínima comparada ao circuito testado. Outro objetivo do trabalho, é a possibilidade de se alterar de forma dinâmica e fácil as propriedades funcionais verificadas pelo circuito, sem a necessidade de recompilar o código VHDL.

1.4 Estrutura do Texto

O restante deste trabalho é organizado como segue: na seção 2 apresentamos uma revisão teórica de conceitos utilizados para ajudar o leitor no entendimento do trabalho proposto. Na seção 3 descrevemos alguns trabalhos relacionados. Na seção 4 será apresentada a proposta onde as ideias a ser implementadas são apresentadas, bem como os objetivos do trabalho de forma mais precisa. Na seção 5 descreveremos como foi feita a implementação do projeto. Na seção 6 será descrito como o trabalho foi validado, demonstrando os experimentos realizados durante a implementação. Por ultimo, na seção 7, serão descritos as conclusões do trabalho bem como trabalhos futuros para este projeto.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Sistemas Reativos

Neste trabalho iremos nos concentrar em verificação de sistemas reativos, termo concedido por Pnueli (HAREL & PNUELLI, 1985). Tais sistemas interagem de forma dinâmica com o ambiente no qual estão inseridos. Em outras palavras, respondem a estímulos externos, de forma a produzir resultados em determinado período de tempo (HAREL & PNUELI, 1985). O comportamento de um sistema cíclico pode ser descrito de forma cíclica: primeiramente espera por sinais de entrada, em seguida calcula as determinadas respostas e então emite os resultados. Diferentemente dos sistemas transformativos (HAREL & PNUELI, 1985), onde o sistema só calcula o resultado através de estímulos internos, não importando variáveis externas, ou seja, não havendo interação com o ambiente externo através de sensores ou botões, por exemplo. Basicamente, isso diferencia sistemas simples de sistemas mais complexos. Desta forma, o entendimento de sistemas reativos ajuda na especificação de sistemas através de propriedades temporais.

Sistemas reativos precisam ser especificados de acordo com o comportamento do ambiente e para isso, lógica temporal fornece uma linguagem natural e completa para expressar este comportamento (MANNA, Z & PNUELI, 1995). Na abordagem deste trabalho, utilizaremos lógica linear temporal (LTL) para especificar as propriedades do um sistema reativo desenvolvendo uma metodologia para provar que um dado projeto funciona de acordo com suas especificações.

2.2 Processo de Verificação

Dentro do processo de um projeto de hardware a verificação é uma das principais etapas. Neste sentido, podemos citar verificação funcional, verificação de tempo, teste de verificação, teste de equivalência (WILE B et. al., 2005) e verificação *burn-in* (BUSHNELL & AGRAWAL, 2000) . No escopo deste trabalho é importante entender verificação funcional, teste de verificação e verificação no sistema (*burn-in*). Verificação funcional, garante que a lógica de um circuito está funcionando corretamente levando em conta todas as circunstancias estipuladas na especificação do projeto. Para tanto, ela é realizada antes da fabricação do chip (WILE B et. al., 2005). Por outro lado, teste de verificação é realizado uma vez que o circuito for fabricado de modo a testar se não houve falha durante a fabricação (WILE B et. al., 2005).

Após o circuito ser validado pelo teste de verificação ele está pronto para desempenhar sua função específica.

Entretanto, existem alguns sistemas críticos onde uma confiabilidade maior é requisitada e desta maneira, apenas verificação funcional e teste de validação são insuficientes. Sendo assim, verificação no sistema (*In-system testing*) tem ganhado importância, principalmente no mercado de missões críticas como militar, hospitalar e automotiva (GOERING, 2012). Também conhecido como *burn-in*, esse tipo de verificação é realizado continuamente ou periodicamente, após o circuito ser validado pelo teste de verificação a fim de detectar falhas que podem ocorrer no decorrer do funcionamento do dispositivo (WILE B et. al., 2005).

2.2.1 Verificação Funcional e Teste de Verificação

Existem algumas diferenças importantes entre teste funcional e teste de validação. Depois de projetar um circuito um engenheiro deve criar a lógica do circuito, escrevendo-a em uma linguagem de descrição de hardware (HDL) onde as mais conhecidas são VHDL e Verilog. Após a escrita do código HDL duas questões permanecem: a primeira, se a lógica descrita representa o que desejamos, e a segunda, o que acontece caso o projetista perder alguma condição crítica (WILE B et. al., 2005). Detectar essas incorreções é o desafio da verificação funcional. Uma vez que a lógica é verificada, o circuito é fabricado e então um novo teste é gerado, desta vez, para analisar se o circuito foi fabricado sem defeitos. Um bom teste de verificação precisa cobrir todas especificações relevantes do dispositivo (BUSHNELL & AGRAWAL, 2000) . Para isso, o engenheiro testador deve saber exatamente quais vetores de testes devem ser gerados para estimular o circuito.

Para fazer a verificação funcional podemos usar duas abordagens diferentes: verificação baseada em simulação e verificação formal (WILE B et. al., 2005). Verificação baseada em simulação é realizada através de *testbench* dentro de um ambiente de simulação para que o comportamento do hardware possa ser simulado e os resultados sejam gerados (DEPRA, 2007). Já a verificação formal é a demonstração matemática do funcionamento de um sistema e será vista com mais detalhes na sequência deste trabalho.

Para o teste de verificação, um vetor de estímulos deve ser gerado de maneira a testar os parâmetros do dispositivo sobre teste (DUT), sob condições normais de operação *(*BUSHNELL & AGRAWAL, 2000) .

2.2.2 Verificação no Sistema (In-System Testing)

Após um dispositivo ser fabricado e validado, ainda assim, falhas e defeitos podem ser manifestados. Além disso, os dispositivos que passam pelo teste de produção não são idênticos, de modo que quando colocados em uso, alguns irão falhar muito rapidamente enquanto outros irão funcionar por um longo período de tempo *(*BUSHNELL & AGRAWAL, 2000). Desse modo, verificação no sistema (*In-system testing or burn-in*) garante confiabilidade de dispositivos testados e aprovados, através de verificação contínua ou periódica (sobre um longo período de tempo) no ambiente onde o dispositivo deve ser utilizado. Estudos relacionados mostram que a ocorrência de falhas potenciais pode ser acelerada com elevadas temperaturas *(*BUSHNELL & AGRAWAL, 2000). Existem dois tipos de falhas que são facilmente detectadas por verificação *burn-in*: falhas de mortalidade infantil *(infant mortality failures)*, frequentemente causadas por uma combinação de projeto sensível e variação de projeto, e falhas anormais *(freak failures)* que requerem longos períodos de teste *(*BUSHNELL & AGRAWAL, 2000).

2.3 Verificação Formal

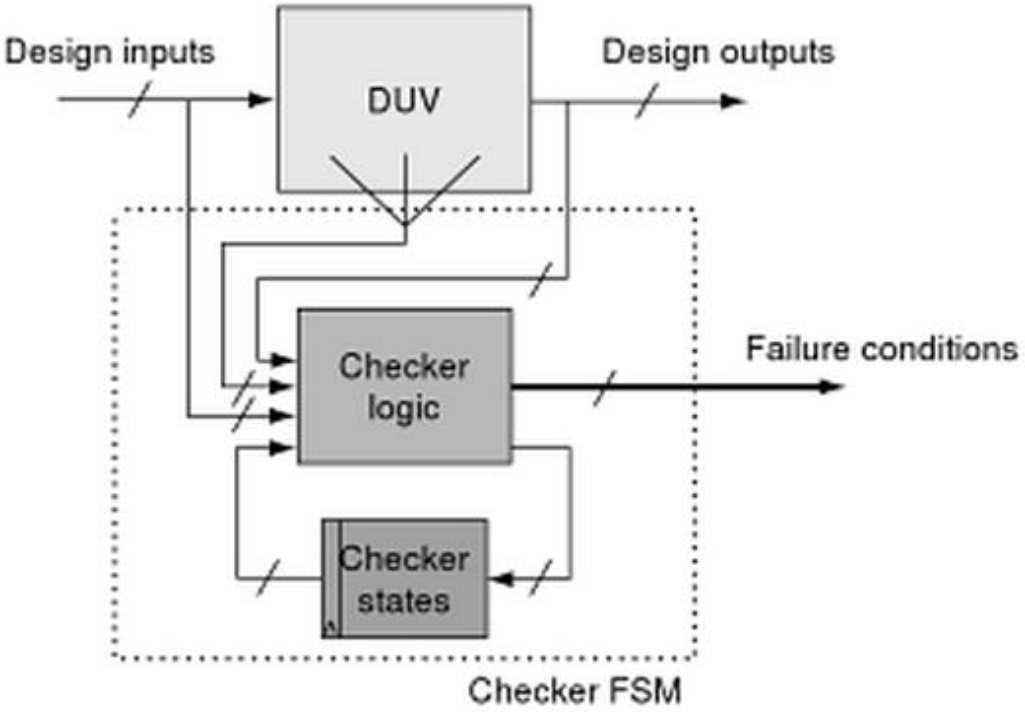
É a verificação de conformidade de uma implementação a certa propriedade ou especificação formal. Essa propriedade é descrita utilizando métodos formais que são modelos matemáticos de provar corretude (KATOEN, 1999), ou seja, é uma demonstração matemática do funcionamento de um sistema reativo. Para tanto, é preciso fazer a construção de um modelo matemático de funcionamento de um sistema a ser testado que representa o comportamento do sistema (KATOEN, 1999).

2.3.1 Verificação de Propriedades

Verificação de propriedades (*property checking*) é em grande maioria a mais importante tecnologia na indústria atualmente (WILE B et. al., 2005). Modelos baseados em

verificação de propriedades são baseados no modelo FSM (*Finite state machine*) do dispositivo sobre teste (DUT). Dessa forma, para verificar formalmente que um projeto está de acordo com uma propriedade, deve-se provar para todos os estados e para todas as transições que a FSM não violou nenhuma condição expressada na propriedade em questão (WILE B et. al., 2005). Na figura 2.1 , temos um verificador FSM, que se conecta as entradas, saídas e sinais internos de um dispositivo sobre teste, verificando-os e gerando um bit de condição de falha para cada propriedade checada.

Figura 2.1 - Ilustração de um verificador FSM



Fonte: *Comprehensive Functional Verification* (2005, p. 469)

2.4 Lógica Linear Temporal

Proposta por Pnueli (HAREL & PNUELLI, 1985), lógica temporal é um formalismo para especificar e verificar propriedades temporais de sistemas reativos através de fórmulas. Uma fórmula de lógica temporal linear descreve o conjunto infinito de sequências para que a fórmula seja verdadeira. O futuro em LTL é visto como uma sequência de estados. Para um dado sistema satisfazer uma ou mais propriedades, ele precisa satisfazer todas as sequências descritas pela fórmula. Caso contrário, o sistema não satisfaz a propriedade temporal.

O modelo de lógica temporal linear (LTL) é uma infinita sequência de estados geralmente formalizados através de um *loop*. Fórmulas temporais são avaliadas em cada *loop* da sequência de estados ao mesmo tempo, ou seja, os sinais das propriedades do sistema devem se manter inalterados durante um ciclo de avaliação (HAREL & PNUELLI, 1985). Uma fórmula em lógica temporal linear é definida por um operador de tempo e uma propriedade p do sistema "Operador p". A propriedade *p* do sistema pode ser qualquer afirmação lógica como por exemplo, se o botão b1 é pressionado, então uma porta p1 deve ser aberta (b1 -> p1, lê-se, se b1 então p1). Os operadores de tempo são definidos através das seguintes letras:

- G (Operador Global) : a fórmula "G *p*" é verdadeira, se *p* é sempre verdadeiro a partir de um tempo t onde a fórmula começa.
- F (Operador Future) : a fórmula "F *p*" é verdadeira, se *p* for verdadeiro em algum tempo no futuro a partir de onde a fórmula começa.
- X (Operador neXt): a fórmula "X *p*" é verdadeira, se *p* é verdadeiro no próximo ciclo de tempo.
- U (Operador Until): a fórmula "X *p*" é verdadeira, se *p* é verdadeiro até o próximo ciclo de tempo.

Em seguida, será definida a sintaxe e semântica da linguagem LTL.

2.4.1 Sintaxe

Para o restante desse trabalho a sintaxe das fórmulas LTL será definida como segue. Assumindo Vars = {a,b,c...} um conjunto finito de símbolos distintos chamados de variáveis de domínio.

> **Definição 1** Um traço T[n...m] (m>=n) é um mapeamento T: {n,...,m} ->2^(Vars). Caso n e m sejam livres de contexto, nós simplesmente escrevemos T ao invés de T[n..m]. O conjunto de todos os traços é denotado por τ (tal). O conjunto de todos os traços T[0,m] com n = ∞ é denotado por τ^{∞}.
>
> (RUF et al., 2001)

> **Definição 2** Assumindo T [0,m], T' [0,n] sendo dois traços com n > m. T' é chamado um extensão dos traços de T se e somente se:
>
> Para todo j com 0 <= j <= m: T(j) = T'(j)
>
> (RUF et al., 2001)

Definição 3 Seja LTL, o conjunto de toda lógica temporal linear para fórmulas, é o menor conjunto que satisfaz:

Vars ⊂ LTL, e
~f, f^g, X[m] f, | G[m,n]f, F[m,n] f ∈ LTL
Se e somente se, g ∈ LTL e m ∈ N e n ∈ N U {∞}

(RUF et al., 2001)

2.4.2 Semântica

A seguir as definições de semântica usada neste trabalho.

Definição 4 A relação de satisfabilidade é definida recursivamente sobre a seguinte estrutura de formulas LTL:

T ⊨i a se a ∈ T (i);
T ⊨i ~f se T ~ ⊨i f;
T ⊨i f^g se T ⊨i f e T ⊨i g ;
T ⊨i X[m] f se T ⊨ (i+m) f;
T ⊨i G[m,n] f se para todo j com i + m ≤ j ≤ i + n tal que T ⊨j f;
T ⊨i F[m,n] f se existe um j com i + m ≤ j ≤ i + n tal que T ⊨j f;

(RUF et al., 2001)

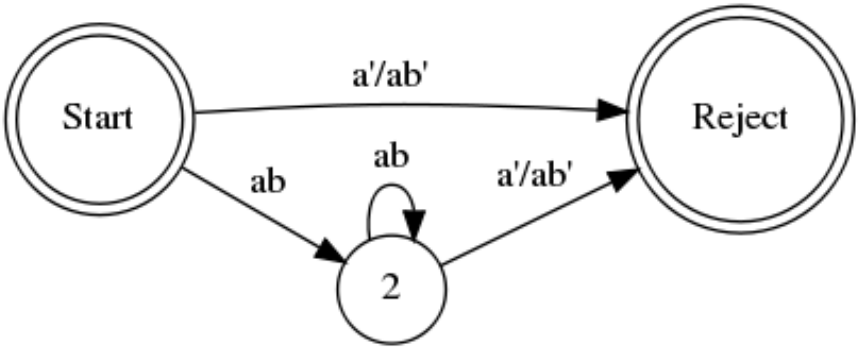
Definição 5 Assumindo f como uma fórmula LTL e T ∈ τ∞ um traço. Dizemos que T satisfaz f (T ⊨ f) se e somente se:

T ⊨0 f.

(RUF et al., 2001)

Na figura 2.2 temos um exemplo de um autômato que representa a sequência de estados para uma fórmula LTL definida por “G (a -> b)”. A lógica temporal é definida pelo operador ‘G’ que de acordo com a definição 4, significa que sempre a lógica a->b (lê-se, se ‘a’ então ‘b’) deve ser verdadeira. Assim, caso em algum ciclo de relógio os sinais de entrada não estiverem de acordo com a fórmula citada, o autômato atinge o estado rejeita. Percebemos também, que o a representação nunca aceita a fórmula provando que a mesma será checada sobre traços infinitos.

Figura 2.2 - Sequência de estados para a fórmula LTL “G (a->b)”.



Fonte: Elaborada pelo autor

2.5 Lógica Temporal Linear Finita

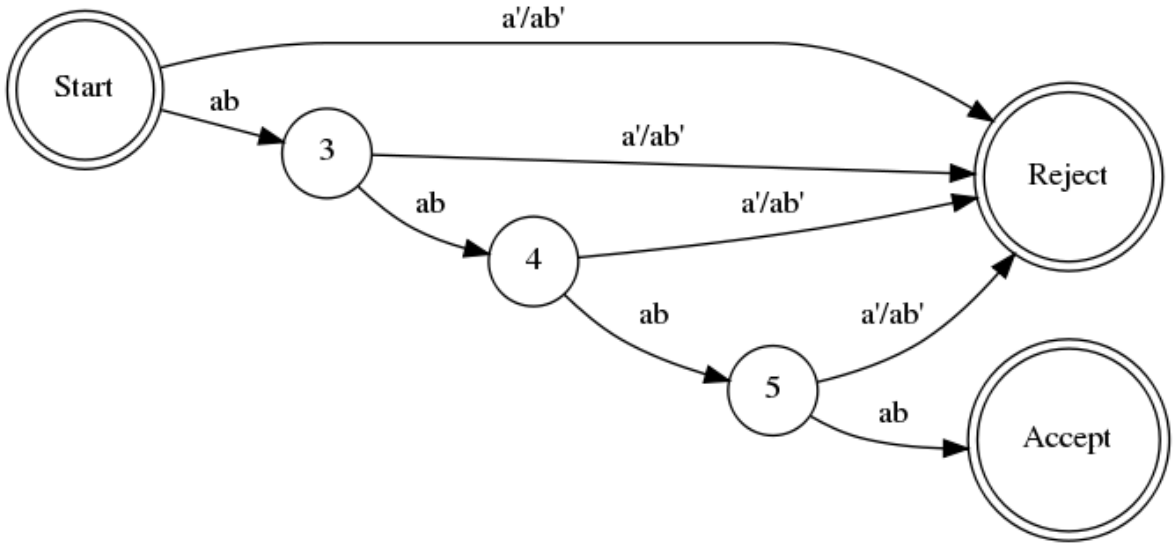
Agora vamos estender a lógica temporal linear que é descrita para infinitos traços para uma lógica temporal de traços finitos. Essa nova forma de interpretação é chamada FLTL (lógica temporal linear finita) (RUF et al., 2001). É importante notar que por ser uma extensão da LTL, ela apenas aumenta a capacidade de representação lógica. Essa lógica será usada neste trabalho porque através dela podemos modelar logicamente sistemas com maior precisão de tempo e também limitando o número de traços para validação de uma fórmula.

> **Definição 6** Assumindo T[0..n] ser um traço e f uma fórmula LTL, f é chamada verdadeira considerando T (denotado por T ⊨ f) se para todo traço de extensão T’[o..∞] de T for válido T’ ⊨ f . f é dito falso com respeito a T se não há extensão do traço T’[0..∞] de T tal que T’ ⊨ f. Caso contrário f é chamado pendente.
>
> (RUF et al., 2001)

Na figura 2.3 podemos observar um exemplo de autômato que representa a sequência de estados para a fórmula “G [3] (a -> b)”. Podemos observar que a equação lógica a->b (lê-se, se ‘a’ então ‘b’), define a lógica de decisão para o próximo estado enquanto G [3] (lê-se sempre válida durante os próximos três ciclos) define a lógica temporal. Observamos o quantificar de tempo ‘3’ que significa que a fórmula será checada para o ciclo inicial e os próximos três ciclos de relógio da sequência. Assim observamos na figura 3, quatro estados (estado inicial e estado número 3, 4 e 5) onde a fórmula lógica é verificada, um estado rejeita,

para o caso de as entradas do sistema não corresponderem à lógica citada, e um estado aceita para o caso de a fórmula ser válida para todos os ciclos testados.

Figura 2.3 - Sequência de estados para a fórmula FLTL "G [3] (a -> b)"



Fonte: Elaborada pelo autor

3 TRABALHOS RELACIONADOS

Muitas técnicas têm sido propostas na literatura para verificação temporal de propriedades de especificação em tempo de execução. Nesta seção vamos apresentar alguns trabalhos relacionados e mostrar quais os pontos motivaram o desenvolvimento do trabalho aqui apresentado. Primeiramente, na seção 3.1, analisaremos um verificador temporal de propriedades FLTL desenvolvido em C++ usando SystemC. Na seção 3.2, nós analisaremos um trabalho intitulado Projeto de verificação para teste online de protocolos de comunicação de Xilinx FPGA.

3.1 Verificador Temporal SystemC

Verificador de propriedades desenvolvido em linguagem *C++* para simulação de circuitos. Segundo o autor (RUF et al., 2001), *SystemC* foi escolhido nesta implementação porque permite a especificação em vários níveis de abstração e fornece uma rápida velocidade de simulação devido à especificação de propriedades diretamente no código a ser compilado.

Neste trabalho o testador pode adicionar propriedades com o uso de diretivas diretamente no código em VHDL ou C/C++ que será verificado durante a simulação. Essas diretivas transformam uma fórmula FLTL representada na forma de uma *string*, em um corresponde autômato que representa a sequência de validação da fórmula. As fórmulas devem incluir funções em C/C++ que representam valores booleanos ao invés de enviar valores Booleanos para verificação (RUF et al., 2001). Essas funções serão verificadas em cada ciclo de simulação e qualquer violação é imediatamente informada para o projetista.

O compilador usado por este verificador, que transforma uma fórmula FLTL em um autômato, foi usado como base para o desenvolvimento do compilador do presente trabalho, onde uma fórmula FLTL é transformada em um conjunto de instruções.

3.2 Verificador de Projeto para Teste On-line de Protocolos de Comunicação de Xilinx FPGA

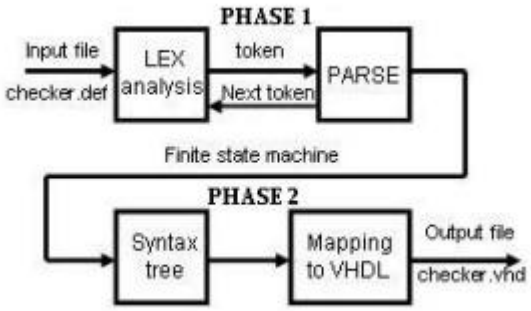
O artigo (STRAKA et al., 2005) cita o desenvolvimento de um verificador temporal para protocolos de comunicação em projetos desenvolvidos na plataforma Xilinx para FPGA.

O trabalho propõe teste concorrente online ou, em outras palavras, quando o circuito sobre teste (CUT) está rodando em sue modo de operação funcional. Para a proposta do trabalho, foram criados uma linguagem formal, com o propósito de descrever o funcionamento de um protocolo, e um gerador de código VHDL que, por sua vez, é criado a partir da lógica descrita através da linguagem formal.

A linguagem formal criada pelos autores, define um autômato a partir de um arquivo composto por duas partes: a primeira contém os símbolos de sinais de entrada que definem um conjunto de condições sobre os sinais do protocolo de comunicação. A segunda parte define a função de transição do autômato e assim, para cada estado e símbolo de entrada, a transição para o próximo estado é definida (STRAKA et al., 2005).

Após o protocolo ser definido através da linguagem em um arquivo, um compilador chamado de *Core Generator* transforma essa definição formal em um código VHDL que é o verificador. O processo de geração do verificador consiste em duas etapas como demonstrado na figura 3.1. Na fase 1, o arquivo de entrada é analisado e as condições de entrada juntamente com as funções de transição são transformados em uma FSM. Na fase 2, as transições da FSM são mapeadas para VHDL.

Figura **Erro! Nenhum texto com o estilo especificado foi encontrado no documento.**.1 - Fases da geração do verificador



Fonte: *Checker Design for On-line Testing of Xilinx FPGA Communications Protocols* (2007, p. 4).

Alguns experimentos foram realizados para testar o código gerado utilizando o FPGA Virtex2. Segundos os autores, os resultados foram muito satisfatórios, fazendo a verificação com um baixo uso de recursos do FPGA, entretanto, não foi demonstrado o quanto dos recursos foi de fato utilizado. Ainda, segundo os autores, a frequência atingida no circuito verificador desenvolvido não afetou a máxima frequência dos protocolos testados. Outro

ponto importante, foi que os circuitos gerados tinham complexidades diferentes de acordo com a verificação a ser feita, definida no arquivo de entrada do compilador.

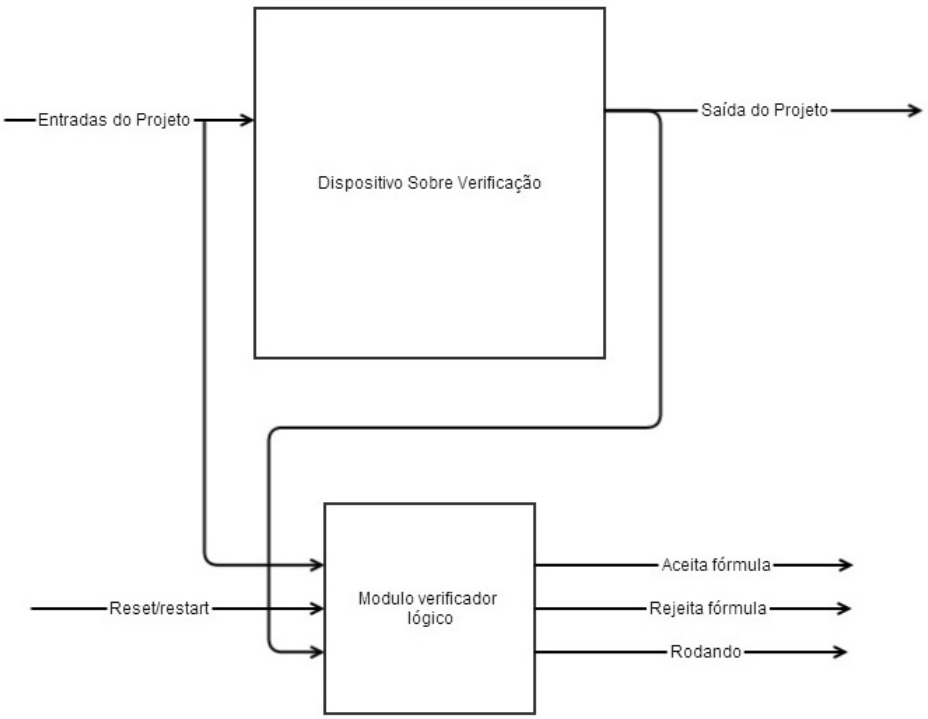
4 PROPOSTA DO TRABALHO E OBJETIVOS

4.1 Proposta

Como mencionado na seção 1, o trabalho propõe um verificador temporal de propriedades formais de forma concorrente com um circuito. Neste sentido, o trabalho será dividido em duas partes. A primeira, trata-se da implementação do verificador formal propriamente dito em VHDL; a segunda, trata-se de um compilador desenvolvido em C++ que vai transformar uma fórmula especifica pela lógica FLTL definida na seção 2 em um conjunto de instruções que serão interpretados pelo verificador temporal. Neste trabalho, implementaremos esta ideia em um FPGA escolhendo a plataforma virtex-5 como alvo.

Assim, em cada ciclo de funcionamento do sistema sobre verificação, as especificações definidas são verificadas a fim de garantir que nenhuma falha indesejada do sistema ocorra de forma despercebida. Dessa forma, foi desenvolvido um módulo verificador que deve ser compilado junto com o circuito para verificar até oito sinais de entrada/saída por módulo como mostrado na figura 4.1. Também observamos na figura 4.1 dois sinais de saída do verificador que indicam que a fórmula foi aceita ou rejeitada mais um sinal informando se a verificação foi finalizada ou não. O sinal reset serve para reiniciar a verificação.

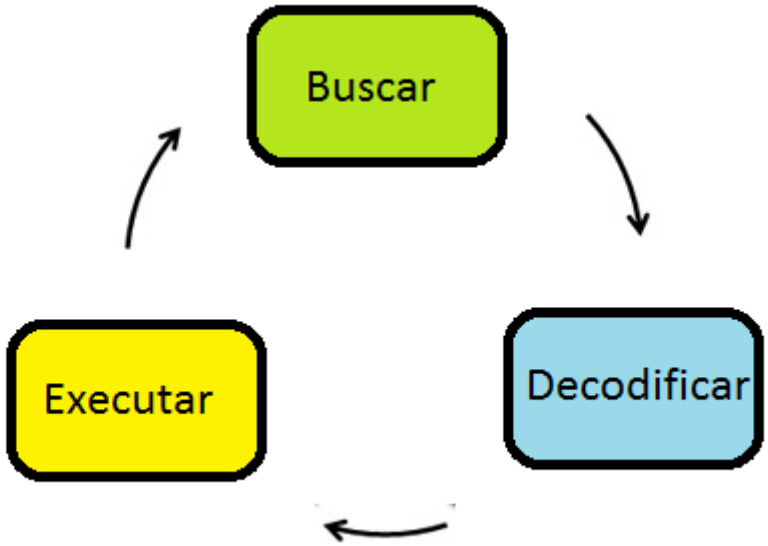
Figura 4.1- Sequência de estados para a fórmula LTL “G (a->b)”.



Fonte: Elaborada pelo autor

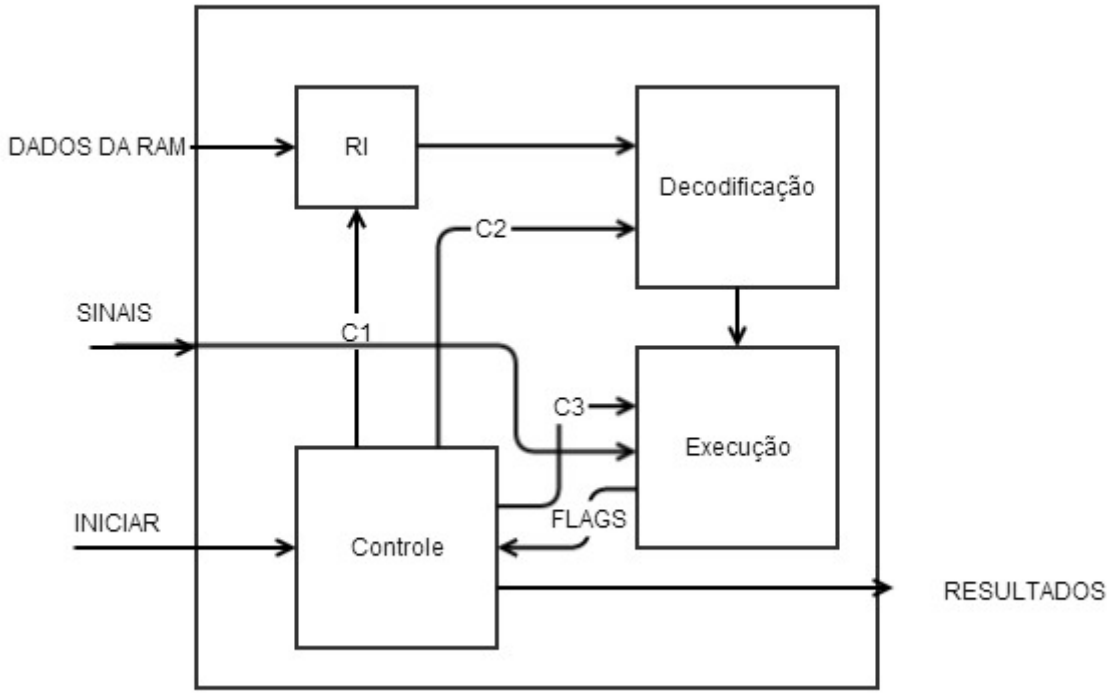
Para fazer a verificação o módulo desenvolvido interpreta, através de uma máquina virtual, um conjunto de instruções que representam uma fórmula FLTL, como demonstrado na figura 4.2. Dessa forma, cada instrução deve ser buscada da memória RAM, decodificada e então executada.

Figura 4.2 - Busca, decodificação e execução



Através do ciclo de instrução mencionado acima, dos sinais de entrada e saída, e das instruções buscadas na memória RAM, uma unidade de controle controla todas as ações necessárias à verificação. A unidade de controle é responsável por emitir sinais de controle para busca de instrução na memória RAM colocando o dado em um registrador RI, bem como sinais para decodificar e executar cada instrução no seu devido ciclo de tempo. Na figura 4.3, podemos observar uma unidade de controle, uma de decodificação e outra de execução, bem como o registrador RI para salvar a corrente instrução buscada da memória. C1, C2 e C3 são os sinais de controle emitidos pela unidade de controle, enquanto que FLAGS são sinais de resultados da execução.

Figura 4.3 - Visão geral do verificador

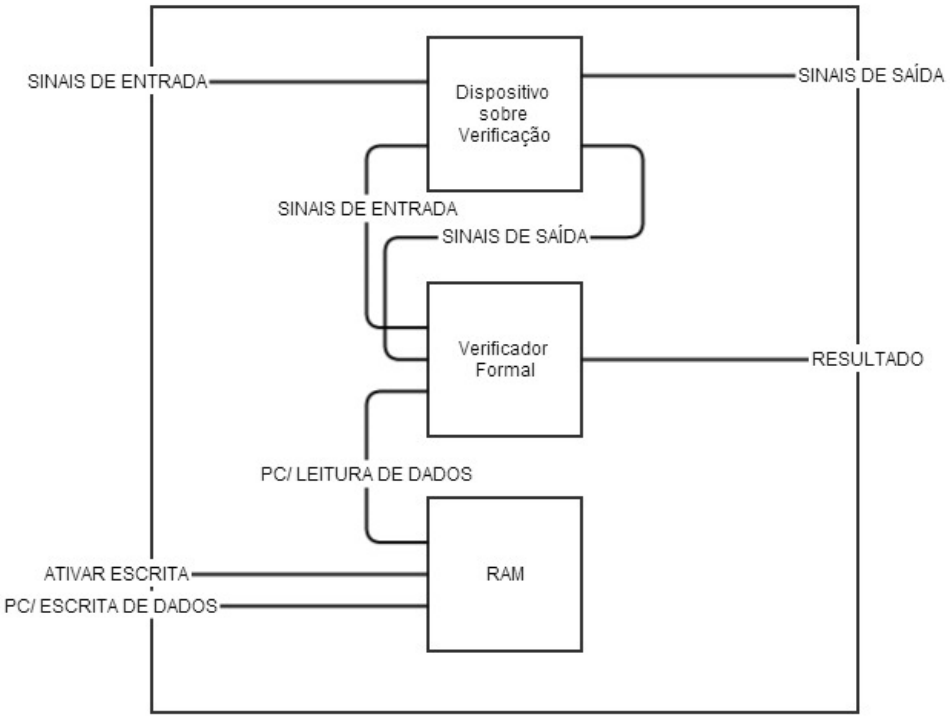


Fonte: Elaborada pelo autor

É importante notar neste ponto, que o usuário especifica uma fórmula de acordo com as regras formuladas na seção 2. Já o verificador, através da máquina virtual, interpreta instruções que representam uma fórmula FLTL. Dessa forma, faz-se necessário a transformação de fórmulas FLTL para um conjunto de instruções que a representam. Para tanto, é necessário a implementação de um compilador que transforme da maneira mais eficiente possível uma fórmula FLTL em um conjunto de instruções, para que as mesmas sejam gravadas na memória RAM do FPGA. A implementação deste compilador foi realizada em C++, onde o usuário informa um arquivo de entrada contendo a fórmula FLTL e um arquivo de saída que será gerado contendo as instruções. A tabela 4.1 representa todas as instruções criadas para a interpretação na máquina virtual.

Uma vez que as instruções são geradas, elas podem ser gravadas na memória RAM do FPGA através da interface demonstrada na figura 4.4, na qual temos uma visão geral com o DSV, o verificador formal de propriedades e a memória RAM.

Figura 4.4 - Visão geral e interface de memória RAM



Fonte: Elaborada pelo autor.

Tabela 4.1 - Conjunto de instruções

Categoria da instrução	Instrução Explícita	Código de Operação	Semântica
Tempo (*time*)	WAIT n	0111	Espera n passos
Comparação (compare)	CHK s	0100	Verifica sinal 's'
Salto (*jump*)	JMP n	0011	Salto para endereço n
	JEQ n	0010	Salto condicional para endereço n
	JNE n	0001	Salto condicional para endereço n
Returno (*return*)	RNE V/F	0101	Retorno de verificação com valores de V (verdadeiro) ou F (falso)
	RET V/F	1101	
	REQ V/F	0110	

Fonte: elaborada pelo Autor.

4.2 Objetivo e Estratégia

Para implementação do trabalho proposto, alguns objetivos foram traçados e para cada um deles uma estratégia foi tomada. Primeiramente, a verificação precisa ser feita em tempo real, sendo o tempo de resposta dos sinais de vital importância. Logo, a precisão de tempo é essencial no desenvolvimento do verificador. Assim sendo, foi escolhido realizar a implementação em VHDL, uma vez que temos mais domínio de como o circuito lógico será criado, se comparado com o uso de síntese de alto nível como *synthesizer*, por exemplo.

Segundo, é desejado que o verificador não atrapalhe o funcionamento normal do circuito. Neste sentido, a estratégia escolhida foi fazer a coleta dos sinais de entrada/saída definidos nas propriedades e concretizar a validação em um ciclo de relógio do DUV, que por sua vez, precisa rodar algumas vezes mais lento que o relógio do verificador, para que haja tempo hábil de executar todo o conjunto de instruções escritas na memória RAM.

O terceiro objetivo traçado está diretamente ligado ao segundo. Como mencionado anteriormente, será desenvolvido um compilador para as fórmulas especificadas em linguagem FLTL. É importante notar que este compilador deve ter o conjunto mínimo de instruções que representa a fórmula especificada. Também, é preciso eliminar durante essa compilação, redundâncias descritas na fórmula pelo usuário testador. Portanto, é de extrema importância para o trabalho proposto um compilador otimizado. Além dessa otimização, o uso de instruções escritas na memória RAM do FPGA permitem que as mesmas sejam mudadas de forma dinâmica, permitindo que o usuário as mude de acordo com as necessidades não exigindo uma recompilação do código.

5 IMPLEMENTAÇÃO

O verificador temporal foi implementado partindo da ideia de que ele deve se comportar exatamente como uma máquina virtual. Assim, necessita basicamente de um contador de programa, para saber qual posição da memória buscar uma instrução; uma unidade decodificadora, onde uma instrução trazida da memória RAM é decodificada e salva em registradores; uma unidade de execução onde a operação na instrução é propriamente realizada; e finalmente, uma unidade de controle, para que o fluxo de execução da máquina virtual seja controlado. Nesta seção, descreveremos como cada unidade foi implementada e por fim descreveremos como foi implementado o compilador FLTL de forma otimizada.

5.1 Unidade de Controle

A unidade de controle é responsável por controlar todas as ações do verificador. Dessa maneira, é ela que controla quando uma instrução é buscada da memória RAM, quando deve ser decodificada e quando deve ser executada. Além dos controles sobre as operações do sistema, é responsável por fazer a sincronização entre o sistema sobre verificação e o verificador e também é responsável por informar se o verificador está rodando ou parado. Em síntese, a unidade de controle foi definida como segue:

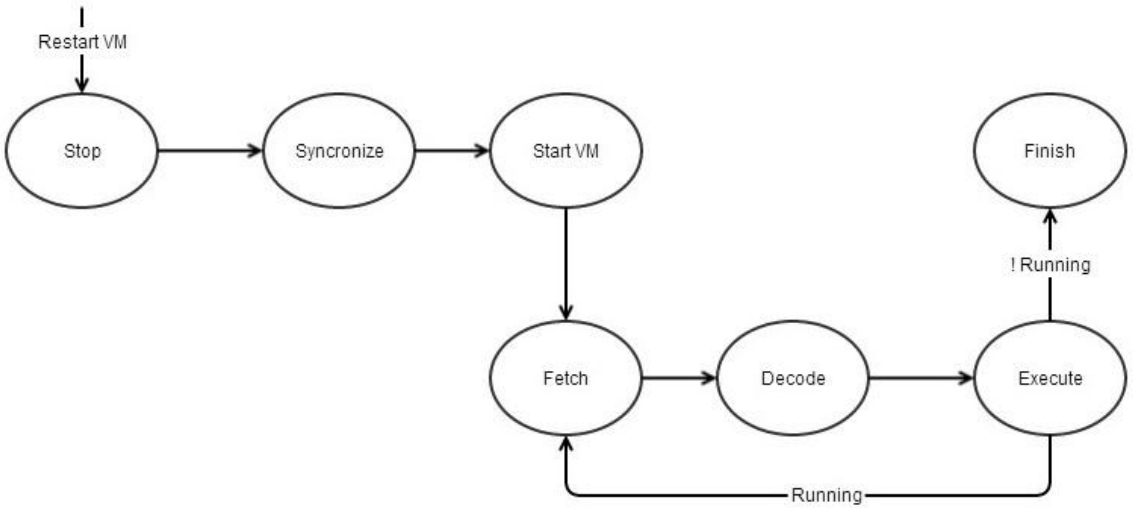
```
entity ControlUnit is
        port (          clock, reset:           in std_logic;
                        restart_VM:             in std_logic;
                        sync_wait:              in std_logic;
                        rejectingState:         in std_logic;
                        acceptingState:         in std_logic;
                        incPC:                  out std_logic;
                        fetch_op:               out std_logic;
                        decode_op:              out std_logic;
                        execute_op:             out std_logic;
                    running:                    out std_logic
                );
end ControlUnit;
```

Fonte: elaborada pelo Autor.

O *flag* "restart_VM" é usado quando se pretende iniciar/reiniciar a verificação do sistema. O *flag* "sync_wait" é usado para fazer a sincronização entre DUV e verificador. Os sinais "rejectingState" e "acceptingState" são analisados para verificar se o sistema ainda deve seguir a verificação.

Na Figura 5.1 podemos observar todos os estados da unidade de controle. Quando reiniciada, a máquina inicia seu processo no estado "Stop" indo para o estado "Syncronize", onde ocorre a sincronização e finalmente "StartVM" onde o processo de verificação é iniciado. Durante todo o processo de verificação a máquina fica entre os estados "Fetch" "Decode" e "Execute", e quando uma fórmula é aceita ou rejeitada, a máquina fica parada no estado "Finish" esperando um sinal para que seja reiniciada novamente.

Figura 5.1 - Conjunto de estados da unidade de controle



Fonte: Elaborada pelo autor

5.2 Unidade de Decodificação

Durante o estado "*Fetch*" uma instrução é buscada na memória RAM e colocada em um registrador "RI" de 16 bits. Durante o estado "*Decode*", a unidade de decodificação entra em execução com a missão de pegar o dado do registrador "RI" e decodificar, colocando-os em dois registradores: o primeiro contendo a código de operação da instrução contendo os 4

bits menos significativos; o segundo, contendo os 12 bits mais significativos e representando um dado da instrução. De modo geral, o dado pode representar ou um *flag* para instruções de RET/RNE/REQ, ou um endereço de salto (*jump)*, ou ainda, qual sinal de entrada a instrução CHK deve analisar. Abaixo segue a definição da unidade de controle:

```
entity DecoderUnit is
        Port (          decode_op :   in  std_logic;
                        reset :       in  std_logic;
                        ri:           in std_logic_vector( 15 downto 0 );
                        op_code:      out std_logic_vector( 3 downto 0 );
                        data :        out std_logic_vector( 11 downto 0 )
                    );
end DecoderUnit;
```

5.3 Contador do Programa

O contador do programa supracitado, serve para posicionar um registrador no endereço da memória RAM no qual uma instrução deve ser buscada. Em suma, é uma unidade simples como demonstrado no código da definição da entidade que segue:

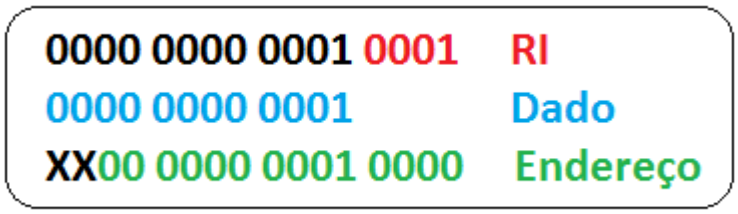
```
entity ProgramCounter is
   Port (
                    clock :          in  std_logic;
                    reset :          in  std_logic;
                    inc_PC :         in  std_logic;
                    restart_VM:      in std_logic;
                    set_jump :       in std_logic;
                    set_wait :       in std_logic;
                    jump_address:    in std_logic_vector( 11 downto 0 );
                    counter :        out std_logic_vector( 13 downto 0 )
                    );
end ProgramCounter;
```

Fonte: elaborada pelo Autor.

Quando um sinal "incPC" é recebido da unidade controladora então o endereço do Contador "Counter" deve ser incrementado. Os *flags* "setWait" e "setJump" são respectivamente, um sinal para esperar um ciclo e um sinal para receber um endereço de salto no contador. Quando um salto é realizado, o endereço contido em "jumpAddress" deve ser colocado diretamente no contador.

É possível observar que o contador (PC) tem 14 bits, isso porque a memória RAM tem 14 bits de endereçamento. Em contrapartida, observamos que o "jumpAddress" que vem do dado da unidade de decodificação possui apenas 12 bits, pois como mencionado anteriormente o dado da memória de 16 bits é dividido em 4 bits para instrução e 12 bits de dado. Neste caso, o dado representa um endereço direto de salto, então quatro bits são adicionados ao lado direito do dado, e os dois bits mais significativos são descartados, totalizando 14 bits para endereço de salto. O motivo para adicionar 4 bits no lado direito do dado é porque o salto mínimo na memória RAM é de 16 posições ("10000" em binário) e fazendo essa manipulação podemos ter saltos mais longos na memória. A figura 5.2 ilustra um exemplo dessa situação. No caso, o endereço de salto é 16 destacado em verde, e a instrução é JNE (*jump not equal*) destacado em vermelho pelos bits "0001".

Figura 5.2 - Utilização de máscara



Fonte: Elaborada pelo autor

Observamos assim, que o endereçamento máximo que o programa pode atingir é 16384 (2 elevado a 14).

5.4 Unidade de Execução

A unidade de execução possui a parte mais importante do código VHDL, pois é nela que encontramos a lógica que descreve a execução de cada uma das oito instruções. Em seguida, podemos observar todos os sinais de entrada e saída da unidade de execução conforme a definição da entidade. Podemos observar os dados "op_code" e "data_in" que representam respectivamente a instrução e o dado recebidos da unidade decodificadora.

```
entity ExecutionUnit is
        port (
                        execute_op:         in std_logic;
                        Z:                  in std_logic;
                        port_in :           in std_logic_vector(7 downto 0 );
                        op_code:            in std_logic_vector( 3 downto 0 );
                        data_in:            in std_logic_vector (11 downto 0);
                        wait_counter:       in  std_logic_vector (15 downto 0);
                        set_jump:           out std_logic;
                        set_wait:           out std_logic;
                        set_Z:              out std_logic;
                        set_accept:         out std_logic;
                        set_reject:          out std_logic
                );
end ExecutionUnit;
```

Fonte: elaborada pelo Autor.

Logo após veremos como cada uma das instruções são processadas de acordo com os sinais de entrada. Em cada instrução os 5 sinais de saída são atualizados.

5.4.1 Instrução NOP

É a instrução definida para não realizar qualquer tipo de mudança de estado na execução. Quando uma instrução NOP (*no operation*) é executada nenhuma operação é realizada pelo verificador. Todos os valores de saída citados na entidade da unidade de execução são colocados em nível lógico zero.

5.4.2 Instrução JMP

A instrução JMP (*jump*) é responsável por ativar o *flag* de salto "set_jump" que é enviado para o contador do programa, informando que um salto deve ser realizado. Os demais sinais são todos colocados em zero.

5.4.3 Instrução JEQ

A instrução JEQ (*jump equal*) é responsável por ativar o flag de salto "set_jump", caso o sinal de entrada "Z" esteja ativo, caso contrário o *flag* deve permanecer desativado. Os demais sinais são todos colocados em zero.

5.4.4 Instrução JNQ

A instrução JNQ (*jump not equal*) é responsável por ativar o *flag* de salto "set_jump" caso o sinal de entrada "Z" esteja desativado, caso contrário o *flag* de ser ativado. Os demais sinais são todos colocados em zero.

5.4.5 Instrução CHK

A instrução CHK *(check*) é a instrução que verificada um sinal do DUT, ativando o *flag* "set_Z", caso a entrada esteja ativa e desativando o *flag* caso a entrada estiver desativada. Os demais sinais são todos colocados em zero.

5.4.6 Instrução RNE V/F

A instrução RNE (*return not equal*) é responsável por retornar um valor de aceita ou rejeita, sempre que o *flag* "Z" esteja desativado. O valor de retorno é representando pelo bit mais significativo que esta contido no registrador de dados de entrada "data_in". Caso o bit

esteja em nível lógico alto, o valor de retorno é verdadeiro, caso contrário o retorno é falso. Os demais sinais são todos colocados em zero.

5.4.7 Instrução REQ V/F

Semelhante a instrução RNE a instrução REQ (*return equal*) é responsável por retornar um valor de aceita ou rejeita, entretanto o valor é retornado sempre que o *flag* "Z" esteja em nível lógico alto. Igualmente a instrução acima, o valor de retorno é representando pelo bit mais significativo que esta contido no registrador de dados de entrada "data_in". Os demais sinais são todos colocados em zero.

5.4.8 Instrução RET V/F

A instrução RET (*return*) sempre retorna o valor contido no bit mais significativo do registrador de dados de entrada "data_in", sendo os demais sinais de saída todos colocados em zero.

5.4.9 Instrução WAIT n

A instrução WAIT tem dois propósitos específicos. Primeiramente, sempre que é executada, o contador "wait_counter" é comparado com o número de espera "n" que está contido no registrador de dados "data_in", ativando o flag "set_wait" sempre que o contador for menor do que o "n" informado. Os demais sinais são todos colocados em zero. O segundo propósito é realizar a sincronização entre o verificador e o DUT após cada verificação.

5.5 Arquitetura Alvo

Para o desenvolvimento do projeto, bem como utilização de unidades primitivas dos FPGAs uma arquitetura alvo foi escolhida. Com este propósito, foi escolhido o dispositivo XC5VLX20T da família Virtex 5.

5.6 Bloco de Memória RAM

Uma unidade primitiva de bloco de memória RAM do Virtex 5 foi utilizado como memória RAM do sistema. A unidade escolhida para implementar o trabalho, possui duas portas de acesso completamente independentes, portas A e B, e consiste de 18Kb de memória (XILINX, 2012). Assim sendo, neste trabalho foi utilizado a porta A para fazer a leitura de uma instrução da memória para o verificador e foi utilizado a porta B para fazer a escrita de instruções na memória RAM. Abaixo segue a definição da interface de memória RAM utilizada:

```
entity RAM18Kb is
        Port (
                        clock :         in  std_logic;
                        reset :         in  std_logic;
                        addressA:       in std_logic_vector(13 downto 0 );
                        addressB:       in std_logic_vector(13 downto 0 );
                        data_inB:       in std_logic_vector(15 downto 0 );
                        write_B:        in std_logic_vector(1 downto 0 );
                        data_outA:      out std_logic_vector(15 downto 0 ) ;
                        data_outB:      out std_logic_vector(15 downto 0 )
                );
end RAM18Kb;
```

A memória RAM definida foi configurada de modo a fazer leitura/escrita de 18 bits em cada leitura, sendo 16 bits de dados e 2 bits de paridade que foram descartados pelo programador. Dessa maneira, temos 1 K posições de memória de 18 bits totalizando assim, 18Kb de memória. A seguir podemos visualizar como cada sinal da interface foi mapeado para memória de modo a perceber quais sinais foram utilizados e quais foram descartados.

```
port map (
                        DOA => data_outA, -- 16-bit A port data output
                        DOB => data_outB -- 16-bit B port data output
                        --DOPA => ParityA,  -- 2-bit A port parity data output
```

```
                --DOPB => ParityB,  -- 2-bit B port parity data output
                ADDRA => addressA, -- 14-bit A port address input
                ADDRB => addressB, -- 14-bit B port address input
                CLKA => clock,   -- 1-bit A port clock input
                CLKB => clock,   -- 1 bit B port clock input
                DIA => X"0000", -- 16-bit A port data input
                DIB => data_inB (15 downto 0), -- 16-bit B port data input
                DIPA => "00",  -- 2-bit A port parity data input
                DIPB => "00",  -- 2-bit B port parity data input
                ENA => '1',     -- 1-bit A port enable input
                ENB => '1',     -- 1-bit B port enable input
                REGCEA => '1', -- 1-bit A port register enable input
                REGCEB => '1', -- 1-bit B port register enable input
                SSRA => reset,  -- 1-bit A port set/reset input
                SSRB => reset,  -- 1-bit B port set/reset input
                WEA => "00", -- 2-bit A port write enable input
                WEB => write_B  -- 2-bit B port write enable input
);
```

5.7 Compilador FLTL

O compilador FLTL é o responsável por pegar uma fórmula lógica FLTL escrita em alto nível através de caracteres e transformar em um conjunto de instruções citadas na seção 5.4. Neste sentido, a compilação destas fórmulas passam pelas etapas de análise léxica, onde os lexemas são identificados, a análise sintática onde é verificado se os símbolos contidos formam uma fórmula lógica ou não, a geração de código, onde a sequência de instruções é gerada e finalmente a otimização de código. Podemos obsevar que a análise semântica bem como a geração de código intermediário não foram necessários uma vez que este compilador é relativamente mais simples que um compilador de código alto nível para código de máquina por exemplo.

5.7.1 Análise léxica

Para a geração do analisador léxico da fórmula de entrada foi usado o programa Flex. Dessa forma, a partir de um arquivo de descrição do analisador léxico, descrito conforme as regras do flex, é gerado um programa em linguagem C que realiza a análise léxica para obter

uma sequência de *tokens*. O arquivo de descrição (grammar.l) contêm a especificação de todas as expressões regulares que serão aceitas para a descrição de uma fórmula FLTL. A tabela 5.1 a seguir representa os lexemas identificados pelo interpretador:

Tabela 5.1 - Tabela de lexemas

Lexema	Tipo
G	Operador temporal *always*
F	Operador temporal *eventually*
X	Operador temporal *next*
Letras ou letras e números	Identificadores
->	Implicação a direita
<-	Implicação a esquerda
!	Operação 'Negação' (NOT)
|	Operador 'OU' (OR)
&	Operador 'E' (AND)
{0,1,2,3..9}	Inteiros

Fonte: elaborada pelo Autor.

5.7.2 Análise sintática

Para a análise sintática foi utilizado o programa Bison que é um gerador automático de analisadores sintáticos. Outrossim, foi definido um arquivo (grammar.y) contendo todas as regras gramaticais da nossa fórmula FLTL que junto com o arquivo de regra de tokens da análise léxica formam a base para o compilador. Desta maneira, uma propriedade de um sistema deve ser especificada segundo este conjunto de regras para que o compilador possa reconhecer a fórmula FLTL e então compilar no conjunto de instruções desejado.

Na tabela 5.2 abaixo, podemos ter uma visão geral de como uma propriedade deve ser especificada. Assim, observamos que uma propriedade é especificada na camada temporal pelos operados de tempo e opcionalmente seguido por inteiros entre colchetes que delimitam o ciclo de verificação do operador temporal informado. Na camada Booleana, ou podemos definir outra propriedade de forma recursiva, ou uma expressão regular booleana.

Tabela 5.2 - Descrição de uma propriedade através da linguagem FLTL

Formação de uma propriedade		
Camada temporal		**Camada boolena**
Operador temporal	**[Inteiros]**	**(Expressão booleana ou propriedade)**

Fonte: elaborada pelo Autor.

Alguns exemplos de fórmulas podem ser vistas na tabela 5.3:

Tabela 5.3 - Exemplos de propriedades

1	**G (event1 -> event2)**	**Sempre que 'event1' então 'event2'**
2	**G (requestValid -> X [5] requestAccept)**	**Sempre que 'requestValid' então, após 5 ciclos, 'requestAccept'.**
3	**X [100] (requestAccept)**	**Após 100 ciclos 'requestAccept'**

Fonte: elaborada pelo Autor.

5.7.3 Geração de Código e Otimização

Caso a verificação sintática seja bem sucedida, uma árvore sintática é criada contendo toda a sequência de *tokens* necessárias para a criação do código final. Uma vez que a árvore sintática é criada, ela é utilizada para a criação do código de instruções. Após a criação da sequência de instruções, uma otimização de código e iniciada, para retirar redundâncias desnecessárias introduzidas pelo usuário na fórmula FLTL que define uma propriedade. Alguns exemplos são citados na tabela 5.4. No primeiro exemplo o usuário digitou uma

fórmula redundante, onde caso fosse traduzida diretamente para instruções, o verificador faria duas verificações do mesmo sinal 'a' no mesmo ciclo de relógio como demonstrado na tabela 5.5, o que é claramente desnecessário. Dessa forma, o compilador e capaz de identificar a redundância e gerar um código otimizado que verifica apenas uma única vez o sinal 'a' no mesmo ciclo de relógio. Nos outros exemplos, outras redundâncias são apresentadas e igualmente otimizadas pelo compilador.

Tabela 5.4 - Otimizações

Fórmula	Verificações	Verificações após otimização
G (a & a)	a (2x)	a (1x)
G (a | b | a)	a (2x)	a (1x)
G (b -> a -> b)	a (1x), b (2x)	a (1x), b (1x)

Fonte: elaborada pelo Autor.

.

Tabela 5.5 - Otimização do código

0: CHK a 16: JNE 64 32: CHK a 48: RNE (1) 64: WAIT 1 80: JMP 0	0: CHK a 16: RNE (1) 32: WAIT 1 48: JMP 0

6 VALIDAÇÃO EXPERIMENTAL E RESULTADOS

Depois de implementados o verificador em VHDL e o compilador em C++, ambos foram testados de modo a checar o correto funcionamento. Nas seções 6.1 e 6.2, analisaremos alguns testes relevantes gerados de modo a demonstrar o funcionamento do verificador temporal de propriedades e do compilador das fórmulas FLTL. Na seção 6.3 será apresentado o consumo de LUTs do módulo verificador desenvolvido no FPGA.

6.1 Teste do Verificador Temporal

Para demonstrar o funcionamento do verificador foram gerados três avaliações de diferentes maneiras. No primeiro caso, usando o simulador ModelSim, foram gerados estímulos de entrada por *testbench* para o verificador funcional e analisado se os sinais de saída estavam de acordo com a fórmula testada. Neste primeiro teste, as fórmulas foram colocadas diretamente no código da memória RAM e compiladas. Em um segundo estágio de avaliação, foi desenvolvido um código simples em VHDL para representar um DUV e usando novamente o ModelSim foram gerados estímulos que foram analisados. Neste segundo teste, as fórmulas foram escritas na memória RAM a partir de uma memória ROM. No terceiro e teste final, foi utilizado um FPGA real, onde o código VHDL foi compilado e então testado. Nesta terceira avaliação, um programa de computador foi utilizado para gravar as instruções na fórmula diretamente na memória RAM. Os resultados dos testes gerados serão apresentados a seguir.

6.1.1 Primeiro Experimento

A fórmula escolhida para demonstrar este teste foi "F [3] ((a-> b) -> c)" cujo respectivo código está representado na tabela 6.1. Dessa forma, até o terceiro ciclo a fórmula é checada e caso em algum ciclo for verdadeira, o verificador retorna verdadeiro, caso contrário, se em nenhum dos ciclos ela for válida, retorna falso, rejeitando a fórmula. Podemos observar na tabela verdade 6.2 para a fórmula citada, o resultado gerado pelo verificador, sendo ou aceita, rejeitada ou ainda passa, onde a validação é postergada para o próximo ciclo caso ainda houver mais ciclos para validar a fórmula do tipo *eventually*.

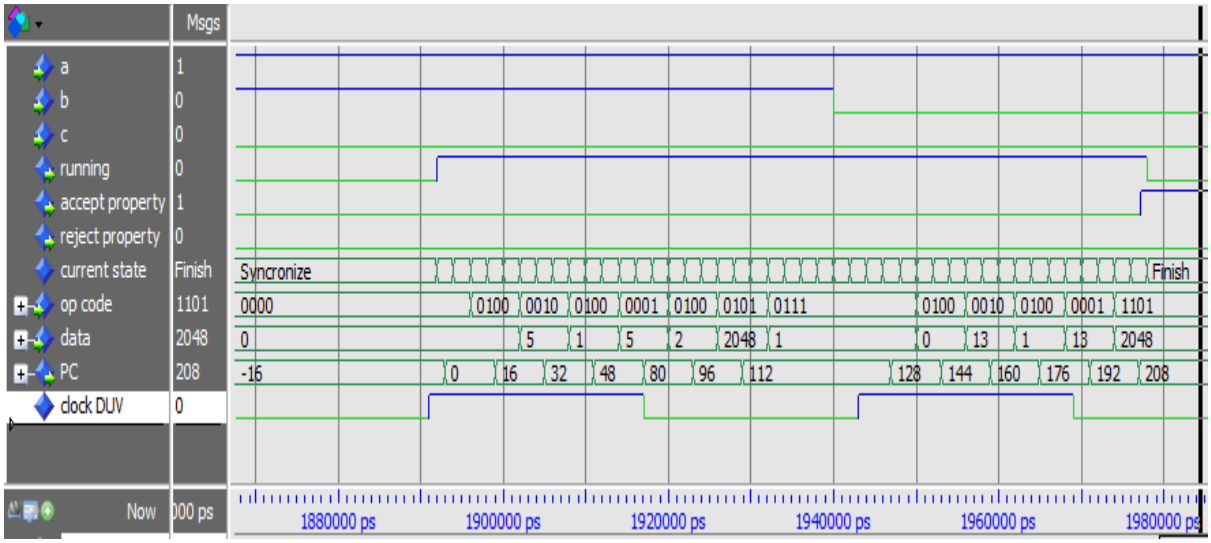
Tabela 6.1 - Código para " F [3] ((a-> b) -> c)"

Posição	Instrução e dado		Posição	Instrução e dado	
0:	CHK	0 (a)	288:	CHK	0 (a)
16:	JEQ	80	304:	JEQ	336
32:	CHK	1 (b)	320:	CHK	1 (b)
48:	JNE	80	336:	JNE	336
64:	RET	1	352:	RNE	1
80:	CHK	2 (c)	368:	CHK	2 (c)
96:	RNE	1	384:	RNE	1
112:	WAIT	1	400:	WAIT	1
128:	CHK	0 (a)	416:	CHK	0 (a)
144:	JEQ	208	432:	JEQ	464
160:	CHK	1 (b)	448:	CHK	1 (b)
176:	JNE	208	464:	JNE	464
192:	RET	1	480:	RNE	1
208:	CHK	2 (c)	496:	CHK	2 (c)
224:	RNE	1	512:	RNE	1
240:	WAIT	1	528:	RET	0

Fonte: elaborada pelo Autor.

Na figura 6.1 podemos observar a demonstração para o caso destacado em azul na tabela verdade, onde é analisado os sinais "a = 1", e "b = 0" e assim retornando aceita, uma vez que o sinal " c = x " não é necessário analisar neste caso. Observe que no primeiro ciclo do DUV os sinais a,b e c são respectivamente '1', '1' e '0', e portanto a fórmula não é aceita, e no segundo ciclo os sinais a, b e c são respectivamente '1', '0' e '0', e então a fórmula é aceita. Na tabela 6.3 temos a instrução relativa a cada código de operação. Podemos observar que na instrução WAIT ("0111") o verificador espera pelo próximo ciclo do DUV para sincronização.

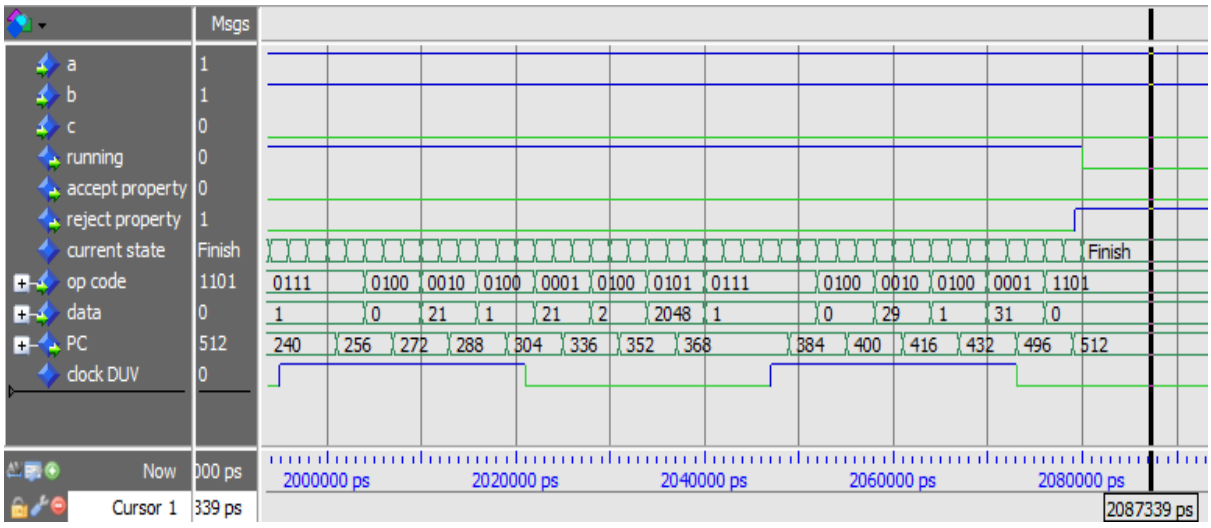
Figura 6.1 - Demonstração da verificação dos sinais em azul da tabela 6.1



Fonte - gerada pelo ModelSim a partir do HDL implementado

Na figura 6.2 podemos observar o caso em vermelho da tabela 6.1, onde a fórmula é rejeitada. Para fins de demonstração, apenas foi colocado os ciclos 2 e 3 de verificação do DUV que pode ser observado pelo nível alto do sinal clock "DUV", onde podemos observar que no ciclo 2 o verificador deixa passar a fórmula, pois ela ainda pode ser validada no terceiro ciclo, onde finalmente é rejeitada.

Figura 6.2 - Demonstração da rejeição para os sinais em vermelho da tabela 6.1



Fonte - gerada pelo ModelSim a partir do HDL implementado

Tabela 6.2- Tabela verdade da fórmula "(a -> b) -> c"

a	b	c	(a->b)->c	Verificador
1	1	1	1	Aceita
1	**1**	**0**	**0**	**Passa/Rejeita**
1	0	1	1	Aceita
1	**0**	**0**	**1**	**Aceita**
0	1	1	1	Aceita
0	1	0	0	Passa/Rejeita
0	0	1	1	Aceita
0	0	0	1	Aceita

Fonte: elaborada pelo Autor.

Tabela 6.3 - Relação entre instrução e código de instrução

JNE	JEQ	JMP	CHK	RNE	REQ	WAIT	RET
0001	0010	0011	0100	0101	0110	0111	1101

Fonte: elaborada pelo Autor.

6.1.2 Segundo Experimento

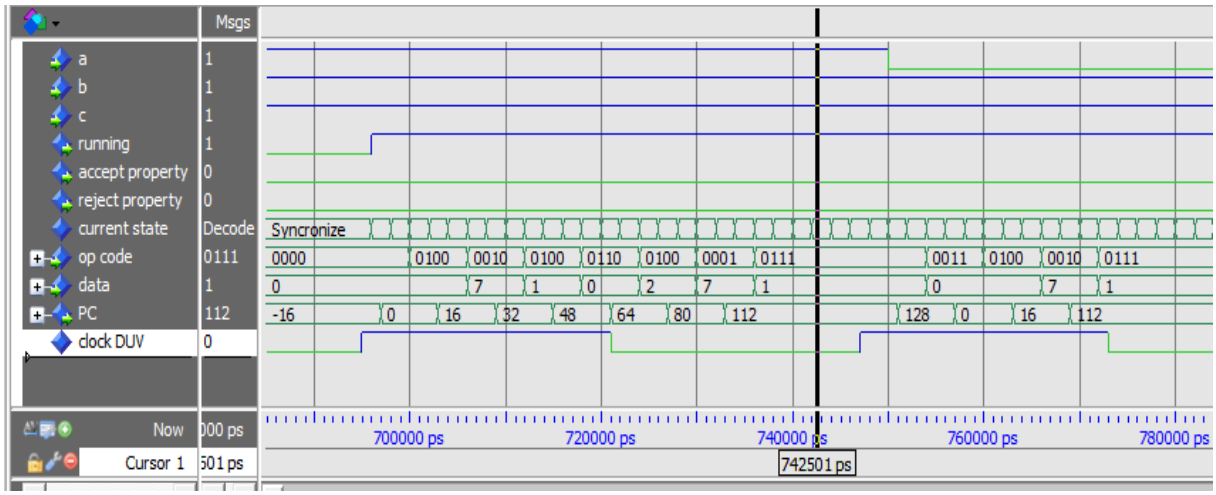
Para o seguindo caso de teste, ainda usando o ambiente de simulação ModelSim, foi criado um circuito VHDL para representar o DUV. Para demonstrar o funcionamento deste teste utilizaremos agora a fórmula "G (a -> (b & c))" que está representada no código da tabela 6.4 . Podemos observar na tabela verdade 6.5 para a fórmula citada, uma ação gerado pelo verificador para cada entrada, onde neste caso o verificador ou rejeita a fórmula, ou deixa para ser verificada no próximo ciclo (Passa). Note que para este caso, não temos o estado "aceita", uma vez que a fórmula deve ser checada em infinitos traços, só parando em caso de rejeição. Neste experimento, a fórmula foi colocada em uma memória ROM e então um controlador grava as instruções na memória RAM e inicia o verificador. Neste experimento, cujo execução está representado nas figuras 6.3 e 6.4, no primeiro ciclo foi verificada a situação em azul da tabela 6.1.2, no ciclo dois foi verificada a situação em verde, e por último, a situação em vermelho em que a fórmula foi rejeitada. Também é demonstrado nas figuras 6.3 e 6.4, o PC para leitura da RAM, bem como os dados de cada instrução (*op code*, *data*).

Tabela 6.4 - Código para "G (a -> (b & c))"

0:	CHK	0 (a)
16:	JNE	112
32:	CHK	1 (b)
48:	REQ	0
64:	CHK	2 (c)
80:	JNE	112
96:	RET	0
112:	WAIT	1
128:	JMP	0

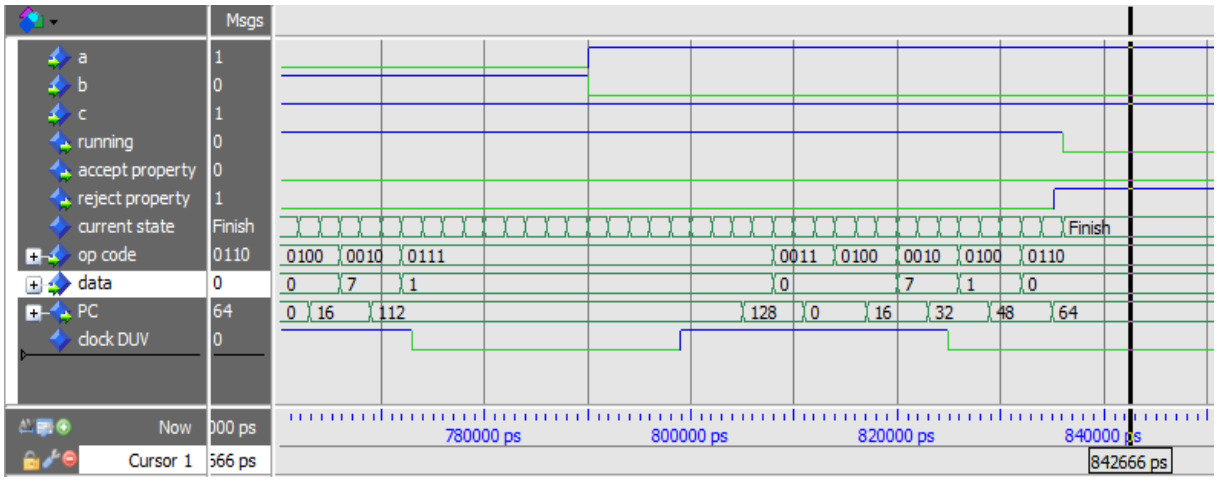
Fonte: elaborada pelo Autor.

Figura 6.3 - Demonstração de funcionamento nos ciclos 1 e 2



Fonte - gerada pelo ModelSim a partir do HDL implementado

Figura 6.4 - Demonstração do funcionamento nos ciclos 2 e 3



Fonte - gerada pelo ModelSim a partir do HDL implementado

Tabela 6.5 - Tabela verdade da fórmula " a -> (b & c)"

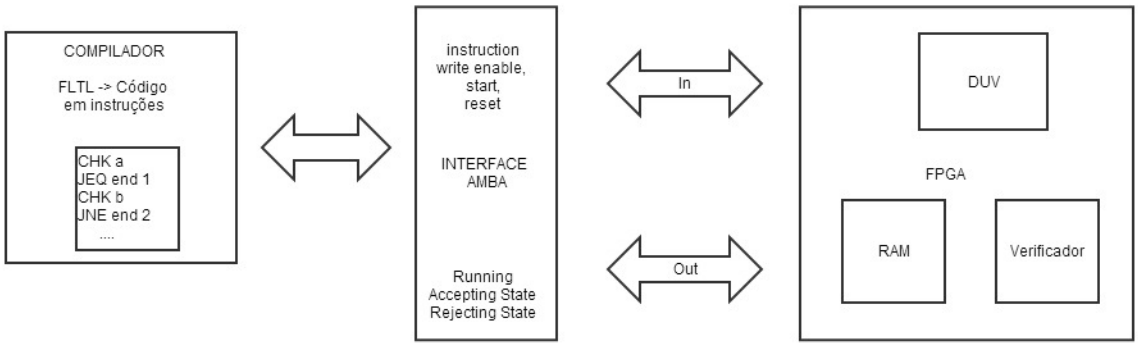
a	b	c	a -> (b & c)	Verificador
1	**1**	**1**	**1**	**Passa**
1	1	0	0	Rejeita
1	**0**	**1**	**0**	**Rejeita**
1	0	0	0	Rejeita
0	**1**	**1**	**1**	**Passa**
0	1	0	1	Passa
0	0	1	1	Passa
0	0	0	1	Passa

Fonte: elaborada pelo Autor.

6.1.3 Terceiro Experimento

No terceiro experimento, o hardware real foi usado para realizar os testes os quais demonstraremos por três fórmulas diferentes. Para fazer a interligação entre o compilador e o FPGA foi utilizado uma interface de barramento AMBA, que permitiu que as instruções fossem gravadas na memória RAM, bem como emitir e receber sinais do verificador. Uma vez que as instruções foram gravadas, um sinal foi emitido para iniciar a verificação temporal e os sinais recebidos pela interface AMBA foram analisados para verificar o funcionamento. Para controlar quais sinais eram emitidos para o verificador duas chaves do FPGA, a e b, foram utilizadas e chaveadas para teste. Na figura 6.1.3.2 podemos ter uma visão geral de como foi realizado a interligação entre o compilador, a interface AMBA e o FPGA.

Figura 6.5 - Visão geral do terceiro experimento utilizando o FPGA virtex 5



Fonte: Elaborada pelo autor

A tabela 6.6 demonstra os testes gerados para a fórmula " X [20] (a | b)", onde depois de 20 ciclos a fórmula "(a | b)" é verificada. Obviamente, não é possível perceber a passagem dos ciclos de relógio e tão logo que liga-se a verificação o resultado é emitido como demonstrado a seguir:

Tabela 6.6 - Verificação da fórmula "X [20] (a | b)" usando o FPGA

chave a	chave b	(a | b)	LED aceita	LED rejeita	LED Rodando
1	1	1	Ligado	Desligado	Desligado
1	0	1	Ligado	Desligado	Desligado
0	1	1	Ligado	Desligado	Desligado
0	0	0	Desligado	Ligado	Desligado

Fonte: elaborada pelo Autor.

Na tabela 6.7 nós podemos analisar uma fórmula do tipo *always*. Neste exemplo, a fórmula é verificada infinitamente caso não houver uma infração da propriedade "a->b". Desta maneira, podemos chavear as chaves "a" e "b" respectivamente de '11' para '01' depois para '00', e o verificador não acusará falha. Quando ligamos a chave "a" e desligamos a chave "b", então o LED rejeita liga, e o verificador finaliza.

Tabela 6.7 - Verificação da fórmula "G (a -> b)" usando o FPGA

chave a	chave b	(a -> b)	LED aceita	LED rejeita	LED Rodando
1	1	1	Desligado	Desligado	Ligado

1	0	0	Desligado	Ligado	Desligado
0	1	1	Desligado	Desligado	Ligado
0	0	1	Desligado	Desligado	Ligado

Fonte: elaborada pelo Autor.

Por último, temos na tabela 6.8 a demonstração de uma formula *eventually*. Assim, a fórmula é checada infinitamente caso não seja aceita. Percebemos que ao chaves podem ser chaveadas de modo a ter ambas desligadas ou apenas uma ligada que nada vai acontecer. Entretanto, no momento que ambas forem ligadas, o verificador aceita a propriedade e finaliza.

Tabela 6.8- Verificação da fórmula "F (a & b)" usando o FPGA

chave a	chave b	(a & b)	LED aceita	LED rejeita	LED Rodando
1	1	1	Ligado	Desligado	Desligado
1	0	0	Desligado	Desligado	Ligado
0	1	0	Desligado	Desligado	Ligado
0	0	0	Desligado	Desligado	Ligado

Fonte: elaborada pelo Autor.

6.2 Usando o Compilador FLTL

No caso do compilador de fórmulas FLTL os experimentos gerados foram bem mais simples. Uma vez que o programa foi implementado, fórmulas foram adicionadas em um arquivo de entrada e compiladas, gerando um arquivo de saída com o código de instruções a ser interpretadas pelo verificador. Uma análise foi feita sobre um conjunto de fórmulas de entrada como será demonstrado a seguir:

6.2.1 Exemplos de fórmulas LTL compiladas

Para demonstrar o funcionamento do compilador será apresentado nesta seção alguns exemplos de fórmulas LTL e seus respectivos códigos compilados. Para uma maior cobertura de apresentação, será demonstrado exemplos simples de cada tipo de fórmula.

Compilando fórmulas do tipo *always* "G (a -> b)" que significa, sempre que 'a' então 'b'. Em outras palavras, sempre que o sinal 'a' estiver ativo, o sinal 'b' também deve estar ativo, como demonstrado na tabela 6.9 onde podemos ver o código gerado pelo compilador.

Tabela 6.9 - Código gerado pelo compilador para "G (a ->b)"

Posição decimal	Código explícito	Código codificado
		Dado e instrução
0	CHK a	0000000000000100
16	JEQ 80	0000000001010010
32	CHK b	0000000000010100
48	JNE 80	0000000001010001
64	RET 0	0000000000001101
80	WAIT 1	0000000000010111
96	JMP 0	0000000000000011

Fonte: elaborada pelo Autor.

Compilando fórmulas do tipo *eventually* "F (a | b)" que significa que eventualmente no futuro 'a' ou 'b', ou seja, a fórmula será checada todos os ciclos, e caso em algum for verdadeira ela é aceita, como demonstrado no código da tabela 6.10 a seguir.

Tabela 6.10 - Código gerado pelo compilador para "F (a | b)"

Posição decimal	Código explícito	Código codificado
		Dado e instrução
0	CHK a	0000000000000100
16	RNE 1	1000000000000101
32	CHK b	0000000000010100
48	RNE 1	1000000000000101
64	WAIT 1	0000000000010111
80	JMP 0	0000000000000011

Fonte: elaborada pelo Autor.

6.2.2 Exemplos de fórmulas FLTL compiladas

Outro tipo de fórmula compilada, são as fórmulas FLTL que tem a diferença de ter um quantificador de tempo na fórmula. Desta maneira, ciclos de relógio podem ser especificados logo após um operador temporal como demonstrado nos exemplos a seguir.

Compilando fórmulas do tipo *next* "X [5] (a & b)" que significa que após 5 ciclos, 'a', 'b' e 'c' devem ser válidos, como demonstrado na tabela 6.11 e no código gerado pelo compilador.

Tabela 6.11 - Código gerado pelo compilador para "X [5] (a & b)"

Posição decimal	Código explícito	Código codificado
		Dado e instrução
0	WAIT 5	0000000001010111
16	CHK a	0000000000000100
32	REQ 0	0000000000000110
48	CHK b	0000000000010100
64	RNE 1	1000000000000101
80	RET 0	0000000000001101

Fonte: elaborada pelo Autor.

6.3 Utilização dos Recursos do FPGA

Após realizada a compilação do verificador, um relatório de utilização dos componentes do FPGA virtex 5 foi gerado usando o Xilinxs Tools. Por conseguinte, podemos observar a utilização de Registradores, *Flip-Flops*, *Latches*, *LUTs (lookup tables)* e pares de *LUT-Flip Flops*. Em todos os casos, é possível perceber um baixo uso dos componentes do virtex 5 que foi a arquitetura alvo escolhida como mencionado na seção 5.5.

Device Utilization Summary:

Number of BUFGs	1 out of 32	3%
Number of External IOBs	66 out of 172	38%
Number of LOCed IOBs	0 out of 66	0%
Number of OLOGICs	1 out of 180	1%
Number of RAMB18X2s	1 out of 26	3%
Number of Slices	34 out of 3120	1%
Number of Slice Registers	89 out of 12480	1%
Number used as Flip Flops	89	
Number used as Latches	0	
Number used as LatchThrus	0	
Number of Slice LUTS	76 out of 12480	1%
Number of Slice LUT-Flip Flop pairs	109 out of 12480	1%

7 CONCLUSÃO E TRABALHOS FUTUROS

Após a realização da implementação do trabalho proposto, bem como a realização de todos os testes para validar o funcionamento, podemos concluir que o trabalho atende de forma satisfatória todos os requisitos esperados em seu funcionamento. Através do compilador podemos facilmente obter as instruções necessárias para a execução do programa no circuito implementado em HDL, de forma eficiente, retirando redundâncias desnecessárias.

Nos experimentos demonstrados neste trabalho, o funcionamento correto para algumas propriedades escolhidas é observado, demonstrando a validação ou rejeição das fórmulas FLTL especificadas. Através do relatório de utilização do FPGA, podemos observar que o circuito implementado é compacto, pois utiliza pouco da capacidade total do FPGA, permitindo que o DUV utilize praticamente todos os recursos disponibilizados por este modelo de FPGA. Outra característica importante atingida, foi o uso da memória RAM para escrita e leitura das propriedades utilizadas. Dessa forma, podemos mudar a propriedade a qualquer tempo de execução, através de escrita de novas instruções na memória.

Para trabalhos futuros, otimizações poderiam ser realizadas a fim de obter um melhor desempenho do circuito verificador. Uma das estratégias poderia ser a construção de um conjunto de instruções mais eficientes. Um exemplo, seria a junção de instruções do tipo CHK e JEQ, CHK e JNQ, pois poderiam formar uma macro instrução, poupando um ciclo de instrução do verificador. Outra ideia sugerida, seria a partir do conjunto de instruções, a construção de um autômato em VHDL que fizesse a verificação das propriedades. Assim sendo, em cada estado vários sinais poderiam ser verificados ao mesmo tempo, deixando a verificação muito mais eficiente.

REFERÊNCIAS

DIJKSTRA, E. **Notes on Structured Programming**. Londres: Academic Press, p. 1-82, 1972.

HAREL, D.; PNUELI, A**. On the Development of Reactive Systems: logic and models of concorrent systems.** Berlin: v.F13, p.477-498, 1985.

RUF, J.; HOFFMANN D.; THOMAS, K. e ROSENSTIEL, W. Simulation based validation of fltl formulas in executable system description. **Forum on Design Languages** , [S.I.], p. 311-319, 2000.

MANNA, Z; PNUELI, A. **Temporal Verification of Reactive Systems: Safety.** [S.I.] Springer-Varlag, 1995.

WILE B.; GOSS C. J.; ROESNER W. **Comprehensive Functional Verification: The complete industry cycle**. 1. Edição. São Francisco: Morgan Kaufman, 2005.

BUSHNELL M. L.; AGRAWAL, V.D. **Essentials of Electronic Testing: for Digital Memory & Mixed-Signal VLSI Circuits.** Boston: Kluwer Academic Publishers, 2000.

GOERING, R. Logic Built-in Self Test (LBIST) is Back: but not for Manufacturing Test. **Informática Pública**, [S.I.], 2012. Disponível em: http://community.cadence.com/-cadence_blogs_8/b/ii/archive/2012/05/10/logic-built-in-self-test-lbist-is-back-but-not-for-manufacturing-test.html, acessado em 03/11/2014.

DEPRA A. D.; ZATT, B. DOS SANTOS M. B., BAMBI, S. Metodologia para Verificação Funcional de Hardware através de Co-simulação Paralela dentro de Sistemas de Software Complexos usando PLI: Decodificador H.264/AVC como Estudo de Caso. **Revista eletrônica PUC**, Porto Alegre, 2007. Disponível em: http://revistaseletronicas.pucrs.br/fo/-ojs/index.php/hifen/article/download/3898/2965, visualizado em 03/11/2014.

KATOEN, J. **Concepts, Algorithms, and Tools for Model Checking**. [S.I]: Erlangen-Nürnberg, 1999.

PNUELI, A. **The Temporal Logic of Programs**. In proceedings of the 18th Annual Symposium on Foundation of Computer Science (FOCS), pág. 46-57, 1977.

RUF, J; HOFFMANN, D. THOMAS, K. e ROSENSTIEL, W. **S. Simulation-Guided Property Checking Based on Multi-Valued AR-Automata.** In: proceedings of the 2001 Conference on Design Automation and Test in Europe pág. 742-748.

AGRAWAL, V. D.; SETH, S. D. **Tutorial - Test Generation for VLSI Chips**. IEEE Computer Society Press. Washington: D.C, 1988.

STRAKA, M.; TOBOLA, J.; KOTASEK Z. Checker Design for On-line Testing of Xilinx FPGA Communication Protocols. **22nd IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems**. Roma, Itália: p. 152-160, set. 2007.

BROWNE, C. M.; EDMUND M. C.; DILL, D. L. Automatic Circuit Verification Using Temporal Logic: two new examples. **Formal Aspects of VLSI Design**. Edinburgh, U.K: North Holland, p. 113-124, 1985.'

XILINX. Datasheet: Virtex-5 FPGA User Guide. Informática Pública, [S.I.], 2012. Disponível em: www.xilinx.com/support/documentation/data_sheets/ds202.pdf, acessado em 04/11/2014.

ANEXO A - TRABALHO DE GRADUAÇÃO I

Verificador Temporal de Propriedades em VHDL Durante a Simulação

Peterson Wilges[1], Orientador Prof. Dr. Renato Perez Ribas[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

pwilges@inf.ufrgs.br, rpribas@inf.ufrgs.br

Abstract. *Verification of digital designs is essential to ensure the correctness and to improve the reliability of systems. This work will make the verification of reactive systems through formal properties using finite linear temporal logic (FLTL). Many techniques have been proposed in order to make formal verification in execution time. This approach is to make verification in VHDL to check the validity of temporal properties of a system by analyzing booleans signals. The checker will be fast enough to check the temporal properties in one clock cycle of a device under test.*

Resumo. *A verificação de projetos digitais é essencial para garantir o correto funcionamento e aumentar a confiabilidade de um sistema. Este trabalho visa fazer a verificação de sistemas reativos através de propriedades formais usando lógica temporal linear finita (FLTL). Muitas técnicas têm sido desenvolvidas para a verificação formal em tempo de execução. A proposta deste trabalho é o desenvolvimento de um verificador em VHDL para checar a validade de propriedades temporais de um sistema através da análise de sinais booleanos. O verificador será rápido o suficiente para checar as propriedades temporais em um ciclo de relógio do dispositivo sobre teste.*

1. Introdução

Garantir corretude é uma das mais importantes tarefas no processo de desenvolvimento de um circuito digital em FPGA. Tanto para garantir o correto funcionamento de acordo com as especificações de um circuito como para garantir melhores índices de confiabilidade em um determinado sistema. Consideraremos como sistema aqueles que são reativos, ou seja, os que são caracterizados por uma continua interação com seu ambiente [Harel & Pnueli 1985]. Neste sentido, os sistemas que serão analisados neste trabalho possuem como entradas variáveis do ambiente e geralmente precisam responder em um curto espaço de tempo. Tipicamente exemplos de sistemas reativos são sistemas operacionais, sistemas de controle de tráfego aéreo e protocolos de comunicação. Esses tipos de sistemas possuem requisitos de segurança e tempo crítico onde eventos, por vezes, devem

ocorrer em determinado tempo.

Uma verificação de funcionalidade pode acontecer em etapas distintas do processo de desenvolvimento e quanto mais cedo um erro de projeto é encontrado menor será o custo de manutenção. Um projeto geralmente inicia com um modelo abstrato descrevendo sua principal funcionalidade e então sofre sucessivas melhorias até chegar ao produto final.

Geralmente a maioria das verificações/validações é feita através de simulação usando *testbench*. Um dos motivos é o seu baixo custo e fácil execução. Entretanto, verificação por carga de *testbenchs* não cobre todos os possíveis casos e pode deixar escapar erros graves e importantes. Dessa forma, só podemos provar que o sistema esta funcionando, sem erros, para o determinado conjunto de testes testado.

Recentemente, métodos formais como verificação de equivalência (*equivalence checking)* e verificação de modelos (*model checking)* tem encontrado espaço em laboratórios de pesquisa [Ruf et al. 2001]. Usando métodos formais, os sistemas podem ser especificados de forma precisa e não ambígua, dando suporte a uma posterior análise semântica do mesmo. Validação baseada em métodos formais tem um grande potencial e é bem aceita em verificação de *hardware* e verificação de protocolos de comunicação, por exemplo. Para o usuário expressar uma ou mais propriedades formais este trabalho faz uso de uma lógica temporal chamada FLTL (*finite linear temporal logic*) [Ruf et al. 2001]. Essa lógica é uma variante da lógica LTL (*linear temporal logic*) que só interpreta propriedades sobre quantidades de tempos em ciclos não determinados ou infinitos. Já por sua vez, na FLTL, o usuário verificador pode quantificar com precisão o intervalo de tempo em número de ciclos que uma determinada atividade pode ou deve acontecer. Para citar um exemplo, imagine um sistema onde um protocolo de comunicação precisa responder a uma mensagem recebida em um espaço de ciclos de relógio bem definido. Este sistema pode ser modelado usando a lógica FLTL, onde o verificador pode quantificar o número de ciclos em que o sistema deve responder. Por outro lado, usando somente LTL a única modelagem possível seria para analisar se houve uma resposta em algum momento indeterminado do futuro.

Portanto, o trabalho aqui desenvolvido visa trazer uma proposta de verificação/validação funcional aprimorada através de um analisador temporal desenvolvido em VHDL, que faz a verificação de propriedades formais durante qualquer etapa de simulação de um *hardware*. Através da verificação formal, o usuário pode especificar propriedades do sistema que serão analisadas pelo verificador. Também é possível especificar com precisão em número de ciclos de relógio que um evento ou um conjunto de eventos devem ou não ocorrer.

A abordagem aqui é para sistemas de pequena e média escala, visto que uma verificação de sistemas de larga escala poderia tornar a análise relativamente complexa ou não eficiente no escopo deste trabalho. Isso porque com o aumento do número de propriedades para análise, o sistema pode sofrer pela explosão combinacional.

O restante deste trabalho é organizado como segue: na seção 2 apresentamos uma revisão teórica dos conceitos aqui descritos. Na seção 3 descrevemos alguns trabalhos relacionados. Na seção 4 será apresentada a proposta e a metodologia a ser utilizada. Por fim, na seção 5 teremos uma conclusão.

2. Fundamentação Teórica

2.1 Sistemas Reativos

Neste trabalho iremos nos concentrar em validação para sistemas reativos, termo concedido por Pnueli [Harel & Pnueli 1985]. Esse tipo de sistema interage de forma dinâmica com o ambiente externo ao qual estão inseridos. Em outras palavras, são sistemas que reagem devido a estímulos internos e/ou externos, de forma a produzir resultados dentro de determinado período de tempo ou em um determinado período de tempo específico [Harel & Pnueli 1985]. O comportamento de um sistema cíclico pode ser descrito de forma cíclica: primeiramente espera por sinais de entrada, em seguida calcula as determinadas respostas e então emite os resultados. Diferentemente dos sistemas transformativos [Harel & Pnueli 1985], onde o sistema só calcula o resultado através de estímulos internos não importando variáveis externas ou em outras palavras não havendo interação com o ambiente externo através de sensores ou botões, por exemplo. Em sistemas reativos, as técnicas formais são as mais indicadas, visto que esse tipo de sistema deve funcionar com alto grau de confiabilidade e segurança. Técnicas de especificação formal são mais confiáveis, pois possibilitam o desenvolvimento de especificações consistentes, completas e sem ambiguidades, através de uma lógica temporal linear com uma semântica bem definida.

2.2 Verificação de Sistemas

Atualmente a corretude de muitos sistemas dependem não somente dos resultados produzidos, mas do tempo em que forem entregues. Vários sistemas são utilizados em circunstancias críticas, onde um funcionamento incorreto ou o funcionamento fora do tempo especificado pode ter consequências desastrosas [Katoen 1999]. Sistemas que processam entradas e baseado nessas entradas fornecem serviços críticos em relação a funcionamento lógico e de tempo de resposta. Por exemplo, quando um motorista aciona o freio do carro um controle de freio é acionando analisando a velocidade do carro e a superfície da estrada a fim de encontrar a frenagem ideal em uma fração de tempo muito curto. Tanto o resultado da frenagem, quanto o tempo em que aconteceu são importantes para medir a segurança do carro. Outros exemplos, onde podemos citar que o tempo pode ser crítico, são determinados protocolos de comunicação, controle de barramento e microprocessadores.

Existem várias técnicas utilizadas para fazer a verificação. Descreveremos brevemente o que é uma verificação por simulação, por teste e então verificação formal que é o escopo deste trabalho.

2.3 Verificação por Simulação

A Simulação é baseada em criar um possível comportamento de um determinado sistema. Um modelador em software chamado simulador vai criar alguns cenários, onde o comportamento do sistema será checado. Dessa forma, o usuário pode obter algumas percepções sobre a reação do sistema baseado em determinados cenários ou estímulos. Os cenários ou estímulos podem ser determinados pelo

usuário que escolhe uma determinada parte do sistema que será avaliada ou pode ser criado aleatoriamente, de forma a gerar testes mais genéricos. Simulação geralmente é uma rápida maneira de testar uma ideia. Assim, faz mais sentido usar em uma fase inicial do projeto, na qual a ideia de como um sistema deve se comportar está mais em voga. Entretanto, não é válido para encontrar erros de programação e também é impossível simular todos os estímulos ou cenários [Katoen 1999].

2.3 Verificação por Teste

Outra maneira de verificar um sistema é por teste. Este sistema é amplamente utilizado por ser muito mais confiável, visto que leva em conta a implementação do sistema como um todo ou uma parte dele [Katoen1999]. Alguns estímulos também chamados de casos de teste são colocados na entrada de uma implementação e a reação do sistema é observada. Compare-se então a saída do sistema com a saída esperada do sistema a fim de validar os testes. É importante observar que o princípio do teste é o mesmo da simulação. Entretanto, no teste é utilizado o sistema real sobre análise, onde o mesmo é executado e suas reações analisadas; ao revés, a simulação é feita levando em conta um modelo de sistema.

Apesar de amplamente utilizado, o teste considera apenas um subconjunto pequeno de testes do sistema. Dessa maneira, testes não provam o total funcionamento de um sistema, eles apenas conseguem provar que para as determinadas cargas de testes o sistema está funcionando. “Testes podem ser usados para mostrar a presença de erros, mas nunca a sua ausência” [Dijkstra 1972].

2.4 Verificação Formal

É a verificação de conformidade de uma implementação a certa propriedade ou especificação formal. Essa propriedade é descrita utilizando métodos formais que são modelos matemáticos de provar corretude. É descrito como uma maneira complementar a verificação por casos [Katoen 1999]. A verificação formal é uma demonstração matemática do funcionamento de um sistema reativo. Para tanto é preciso fazer a construção de um modelo matemático de funcionamento de um sistema a ser testado que representa o comportamento do sistema.

2.5 Lógica Linear Temporal

Lógica temporal é um formalismo para especificar e verificar propriedades temporais de sistemas reativos através de fórmulas, foi proposto primeiramente por Pnueli [Pnueli 1977]. Uma fórmula de lógica temporal linear descreve o conjunto infinito de sequências para que a fórmula seja verdadeira. O futuro em LTL é visto como uma sequencia de estados. Para um dado sistema satisfazer uma ou mais propriedades, ela precisa satisfazer todas as sequencias descritas pela fórmula. Caso contrário, o sistema não satisfaz a propriedade temporal.

O modelo de lógica temporal linear (LTL) é uma infinita sequência de estados geralmente formalizados através de um *loop*. Fórmulas temporais são avaliadas em cada *loop* da sequência de estados ao mesmo tempo, ou seja, os sinais das propriedades do sistema devem se manter inalteradas durante um ciclo de avaliação [Pnueli 1977]. Em seguida será definida a sintaxe e semântica da

linguagem LTL.

2.5.1 Sintaxe

Para o restante desse trabalho a sintaxe das fórmulas LTL será definida como segue. Assumindo Vars = {a,b,c...} um conjunto finito de símbolos distintos chamados de variáveis de domínio.

> **Definição 1** Um traço T[n...m] (m>=n) é um mapeamento T: {n,...,m} ->2^(Vars). Caso n e m sejam livres de contexto, nós normalmente simplesmente escrevemos T ao invés de T[n..m]. O conjunto de todos os traços é denotado por **τ.** O conjunto de todos os traços T[0,m] com n = ∞ é denotado por τ∞.
>
> [Ruf et al. 2001]

> **Definição 2** Assumindo T [0,m], T' [0,n] sendo dois traços com n > m. T' é chamado um extensão dos traços de T se e somente se:
>
> Para todo j com 0 <= j <= m: T(j) = T'(j)
>
> [Ruf et al. 2001]

> **Definição 3** LTL, o conjunto de toda lógica temporal linear para fórmulas, é o mais curto conjunto que satisfaz:
>
> Vars ⊂ LTL, e
> ~f, f^g, X[m] f, | G[m,n]f, F[m,n] f ∈ LTL
> Se e somente se, g ∈ LTL e m ∈ N e n ∈ N U {∞}
>
> [Ruf et al. 2001]

2.5.2 Semântica

A seguir as definições de semântica usada neste trabalho.

> **Definição 4** A relação de satisfabilidade é definida recursivamente sobre a seguinte estrutura de formulas LTL:
>
> $T \models_i a$ se $a \in T(i)$;
> $T \models_i \sim f$ se $T \sim \models_i f$;
> $T \models_i f \wedge g$ se $T \models_i f$ e $T \models_i g$;
> $T \models_i X[m]\, f$ se $T \models (i+m)$ f;
> $T \models_i G[m,n]\, f$ se para todo j com $i + m \leq j \leq i + n$ tal que $T \models_j f$;
> $T \models_i F[m,n]\, f$ se existe um j com $i + m \leq j \leq i + n$ tal que $T \models_j f$;
>
> [Ruf et al. 2001]

> **Definição 5** Assumindo f como uma fórmula LTL e T ∈ τ∞ um traço. Dizemos que T satisfaz f (T ⊨ f) se e somente se:
>
> $T \models_0 f$.
>
> [Ruf et al. 2001]

Na figura 1 temos um exemplo de um autômato que representa a sequência de estados para uma fórmula LTL definida por "G (a -> b)". A lógica temporal é definida pelo operador 'G' que de acordo com a definição quatro, significa que sempre a lógica a->b (lê-se, se 'a' então 'b') deve ser verdadeira. Assim, caso em algum ciclo de relógio os sinais de entrada não estiverem de acordo com a fórmula

citada, o autômato atinge o estado rejeita. Percebemos também, que o a representação nunca aceita a fórmula provando que a mesma será checada sobre traços infinitos.

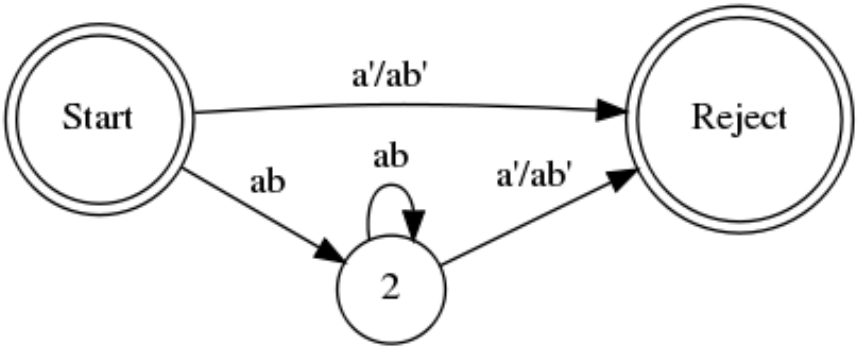


Figura 1 – Sequência de estados para a fórmula LTL “G (a->b)”.

2.6 Lógica Temporal Linear Finita

Agora vamos estender a lógica temporal linear que é descrita para infinitos traços para uma lógica temporal de traços finitos. Essa nova forma de interpretação é chamada FLTL (lógica temporal linear finita) [Ruf et al. 2000]. É importante notar que por ser uma extensão da LTL, ela apenas aumenta a capacidade de representação lógica. Essa lógica será usada neste trabalho porque através dela podemos modelar logicamente sistemas com maior precisão de tempo e também limitando o número de traços para validação de uma fórmula.

> **Definição 6** Assumindo T[0..n] ser um traço e f uma fórmula LTL, f é chamada verdadeira considerando T (denotado por T ⊨ f) se para todo traço de extensão T’[o..∞] de T for válido T’ ⊨ f . f é dito falso com respeito a T se não há extensão do traço T’[0..∞] de T tal que T’ ⊨ f. Caso contrário f é chamado pendente.
> [Ruf et al. 2001]

Na figura 2 podemos observar um exemplo de autômato que representa a sequência de estados para a fórmula “G [3] (a -> b)”. Podemos observar que a equação lógica a->b (lê-se, se ‘a’ então ‘b’), define a lógica de decisão para o próximo estado enquanto G [3] (lê-se sempre válida durante os próximos três ciclos) define a lógica temporal. Observamos o quantificar de tempo ‘3’ que significa que a fórmula será checada para o ciclo inicial e os próximos três ciclos de relógio da sequência. Assim observamos na figura 2, quatro estados (estado inicial e estado número 3, 4 e 5) onde a fórmula lógica é verificada, um estado rejeita, para o caso de as entradas do sistema não corresponderem à lógica citada, e um estado aceita para o caso de a fórmula ser válida para todos os ciclos testados.

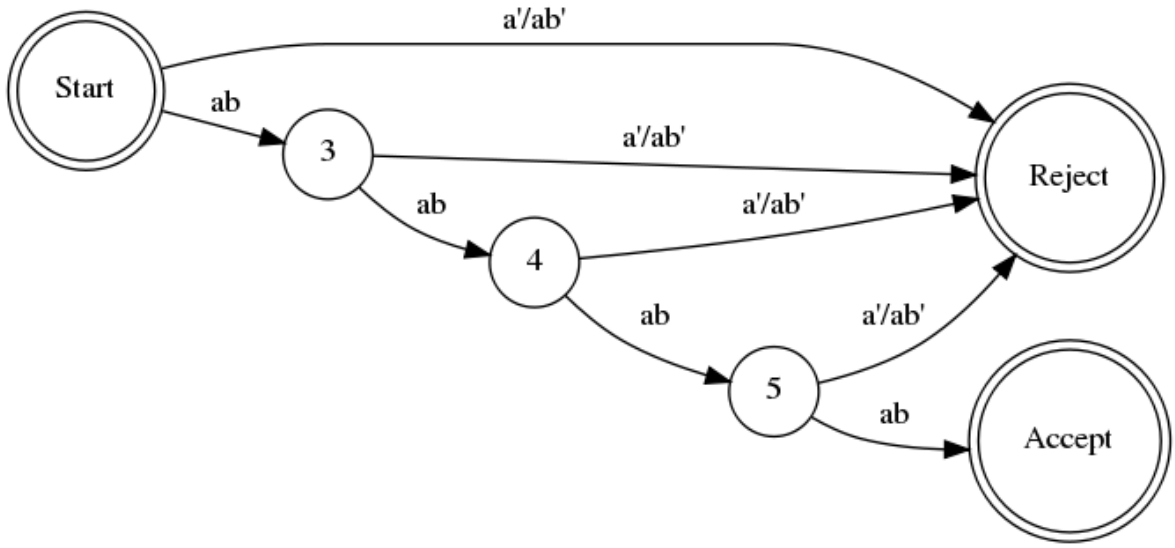


Figura 2 – Sequência de estados para a fórmula FLTL “G [3] (a->b)”

3. Trabalhos Relacionados

Muitas técnicas têm sido propostas na literatura para verificação temporal de propriedades de especificação em tempo de execução. Nesta seção vamos apresentar alguns trabalhos relacionados a este assunto e mostrar quais os pontos motivaram o desenvolvimento do trabalho aqui apresentado. Primeiramente, em 3.1, mostramos um verificador temporal chamado Verificador Temporal desenvolvido em C++ e usado para verificar propriedades de funções em linguagem C++. Em 3.2 iremos analisar uma técnica apresentada para verificação de eventos em VHDL.

3.1 Verificador Temporal

Verificador de propriedades desenvolvido em linguagem SystemC. Segundo o autor, SystemC foi escolhido nesta implementação porque permite a especificação em vários níveis de abstração e fornece uma rápida velocidade de simulação devido a compiladas especificações executáveis [Ruf et al 2001].

Neste trabalho o testador pode adicionar propriedades com o uso de diretivas diretamente no código em C/C++ ou VHDL. Essas diretivas transformam uma fórmula FLTL representada na forma de uma *string*, em um corresponde automoto que representa a sequência de validação da fórmula.

As fórmulas devem incluir funções em C/C++ que representam valores booleanos ao invés de enviar valores booleanos para verificação [RUF et al., 2001]. Essas funções serão verificadas em cada ciclo de simulação.

3.2 Verificação de Projetos VHDL usando VAL

Técnica utilizada para verificação de padrões em descrições VHDL. VAL (Linguagem de anotação VHDL) é uma extensão da linguagem VHDL que possibilita automatizar a verificação em simulações de aplicações em VHDL. Assim é possível adicionar padrões de funcionamento com comentários especiais dentro do código

VHDL. Esses padrões devem aparecer em uma ordem especificada [Augustin 1988].

Por precisar colocar padrões de lógica linearmente encadeada esse verificador possui uma lógica mais limitada em relação ao trabalho introduzido aqui. Também questões de desempenho devem ser analisadas entre VAL e o trabalho proposto.

4. Proposta e Metodologia

A proposta deste trabalho é o desenvolvimento de um verificador temporal eficiente de propriedades que seja tão rápido quanto possível. Para tanto, o trabalho será desenvolvido em VHDL e usará uma forma de especificação de propriedades formal como especificado na seção 2, chamada de lógica temporal linear finita (FLTL). O uso desta especificação lógica é justificado pela sua precisão matemática para provar o funcionamento de um sistema, além disso, é de fácil entendimento.

A proposta - para ser efetivamente eficiente em FPGA – como se pretende, será realizada da seguinte maneira: para a análise de uma propriedade o testador deverá escrever uma fórmula de acordo com as especificações na seção 2 de FLTL. Tal fórmula será compilada em C++ através de uma *string* e gerará um código que chamaremos de código intermediário. Esse código, que é uma sequência de instruções, será gravado na memória de um FPGA. O código intermediário representa assim as propriedades especificadas de um sistema a ser verificado. A partir dessa memória, com a sequência de instruções que representam as propriedades a serem checadas, este verificador temporal lerá as instruções e entregará um resultado de avaliação de acordo com os sinais recebidos de outro sistema sobre avaliação, também chamado de dispositivo sobre teste.

O desenvolvimento em linguagem de descrição de *hardware* (VHDL) foi escolhido tendo em vista que possibilita uma verificação rápida. Assim é possível analisar uma ou mais propriedades em um único ciclo de relógio do dispositivo sobre teste.

O código intermediário é a maneira escolhida de transformar uma fórmula de FLTL em um conjunto de sequências que podem ser analisadas em VHDL. Para tanto, o VHDL interpretará um conjunto de instruções através da concepção de uma máquina virtual conforme a figura 3.

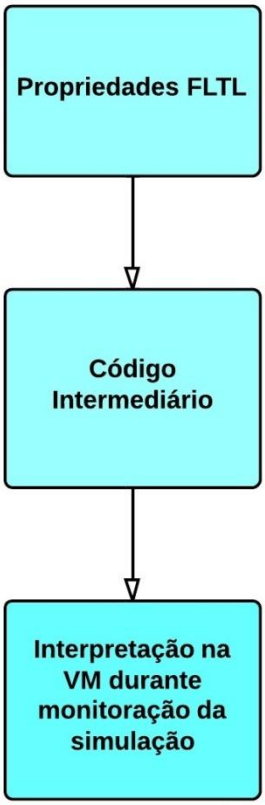


Figura 3 – Fluxo de interpretação de propriedades

Em cada ciclo de verificação do conjunto de instruções o verificador deverá validar as propriedades analisadas. Por conseguinte, se a fórmula especificada estiver de acordo com os sinais de entrada, a propriedade deverá ser aceita. Caso contrário, se os sinais de entrada do sistema não estiverem de acordo com a fórmula especificada, a propriedade deverá ser rejeitado e um sinal de alerta será emitido mostrando que houve um erro. Além disso, a cada ciclo de funcionamento do dispositivo sobre teste as suas propriedades descritas, através da fórmula, deverão ser analisadas.

5. Conclusão

Este trabalho apresenta um verificador temporal de propriedades para sistemas reativos. Geralmente os verificadores temporais são desenvolvidos em linguagens de programação e por isso não se apresentam rápidos o suficiente para checar propriedades em FPGAs. Buscando melhorias, esta proposta traz um verificador desenvolvido em linguagem de descrição de *hardware* (VHDL) que pode checar um conjunto de propriedades em FPGA de maneira rápida e eficiente.

6. Tarefas & Cronograma

A seguir serão descritas as tarefas necessárias para o desenvolvimento e estimativa de tempo, em semanas, para a finalização do trabalho proposto.

Primeiramente, serão estudados mais profundamente os trabalhos

relacionados a fim de extrair dados que serão utilizados para realizar comparações. Em seguida será estudado um conjunto de instruções para o código intermediário que representarão uma fórmula FLTL. Com o conjunto de instruções definidos, será estudado os blocos essenciais em VHDL da máquina virtual que vai interpretar essas instruções. Em seguida será feita a implementação dos blocos. Feito isso, o verificador será otimizado e testado para validação.

Tabela 1 – Cronograma de trabalho proposto

	Atividade	Duração
1	Trabalhos relacionados	2 semanas
2	Estudo das instruções de código intermediário	1 semanas
3	Definição dos blocos VHDL	1 semanas
4	Implementação dos blocos	7 semanas
5	Otimização e teste	2 semanas
6	Escrita do trabalho de graduação 2	4 semanas

Referências

Augustin, L., Gennart, B., Huh, Y., Luckham, D. and Stanculescu, A. (1988) "Verification of VHDL designs using VAL." In: *Design Automation Conference (DAC)*, pages 48–53

Dijktra, E., (1985) " Notes on structured programming." In: Structured Programming, by O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Academic Press.

Harel, D. & Pnueli, A. (1985) "On the development of reactive systems." NATO ASI Series - Logics and Models of Concurrent Systems, pag. 477—498.

Katoen, J. (1999) "Concepts, Algorithms, and Tools for Model Checking." IMMD Berichte, 32Friedrich-Alexander-Universität Erlangen-Nürnberg.

Pnueli, A. (1977) "The Temporal Logic of Programs." In proceedigs of the 18th Annual Symposium on Foundation of Computer Science (FOCS), pag. 46-57.

Ruf, J., Hoffmann D., Thomas, K. and Rosenstiel, W. (2000) "Simulation based validation of fltl formulas in executable system descriptions." In: R. Seepold, editor, *Forum on Design Languages(FDL 2000)*, pages 311–319.

Ruf, J., Hoffmann D., Thomas, K. and Rosenstiel, W. (2001) "Simulation-Guided Property Checking Based on Multi-Valued AR-Automata." In: proceedings of the 2001 Conference on Design Automantion and Test in Europe pág. 742-748.