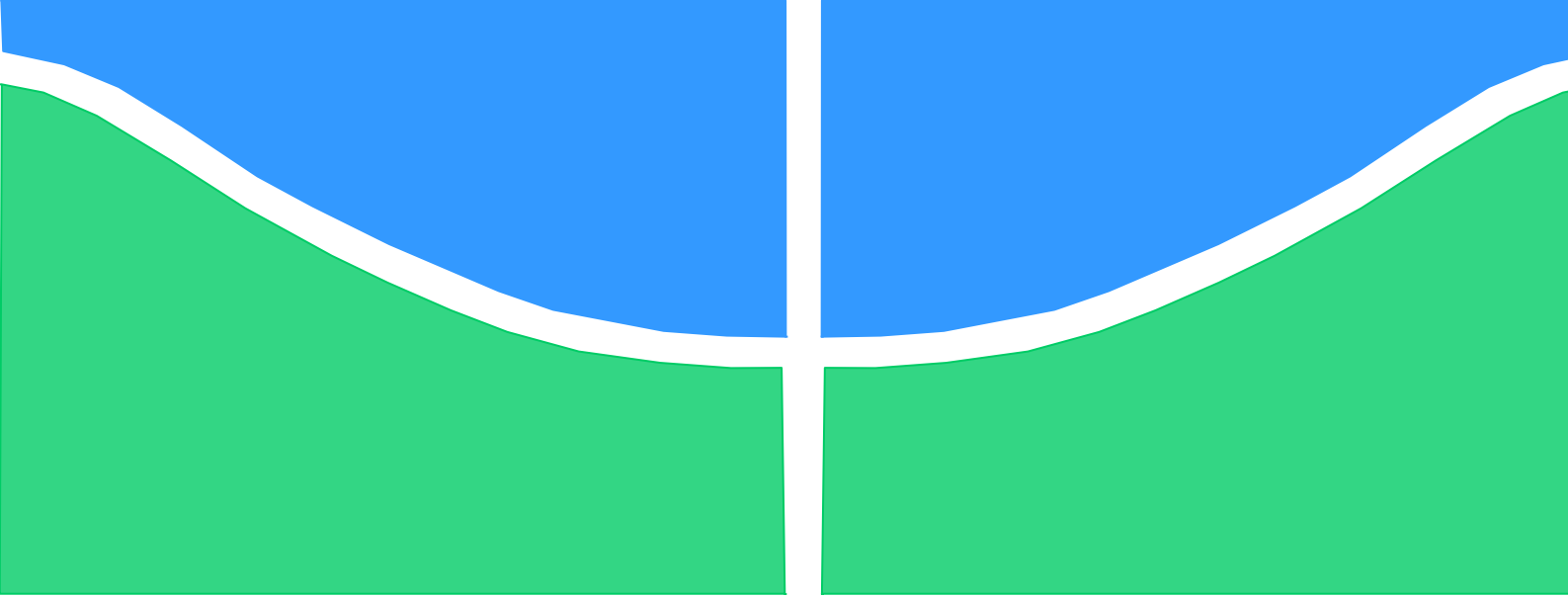

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Curso de Engenharia Eletrônica

PROJETO E SÍNTESE EM ALTO NÍVEL DE CIRCUITOS DIGITAIS

Autor: João Pedro Alexandroni Cordova de Sousa
Orientador: Gilmar Silva Beserra

Brasília, DF
2017

João Pedro Alexandroni Cordova de Sousa

Projeto e Síntese em Alto Nível de Circuitos Digitais

Monografia submetida ao curso de graduação em engenharia eletrônica da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em engenharia eletrônica.

Universidade de Brasília - UnB

Faculdade de Gama

Programa de Graduação

Orientador: Gilmar Silva Beserra

Brasil

2017, v-1.0

João Pedro Alexandroni Cordova de Sousa
Projeto e Síntese em Alto Nível de Circuitos Digitais/ João Pedro Alexandroni Cordova de Sousa. – Brasil, 2017, v-1.0-
90 p. : il. ; 30 cm.

Orientador: Gilmar Silva Beserra

Tese (Graduação) – Universidade de Brasília - UnB
Faculdade de Gama
Programa de Graduação, 2017, v-1.0.

1. Circuitos Digitais 2. Síntese I. Orientador. Gilmar Beserra Silva II. Universidade de Brasília III. Faculdade do Gama IV. Projeto e Síntese em Alto Nível de Circuitos Digitais

CDU xx:xxx:xxx.x

João Pedro Alexandroni Cordova de Sousa

Projeto e Síntese em Alto Nível de Circuitos Digitais

Monografia submetida ao curso de graduação em engenharia eletrônica da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em engenharia eletrônica.

Trabalho aprovado. Brasil, 11 de dezembro de 2017:

Gilmar Silva Beserra
Orientador

Dr. Daniel Muñoz
UnB/FGA

Dr. Daniel Chaves Café
UnB/ENE

Brasil
2017, v-1.0

Este trabalho é dedicado a todos aqueles que acreditam nos seus sonhos, mesmo quando tudo parece ir contra.

Agradecimentos

Agradeço primeiramente a Deus, que me permitiu seguir este caminho; à minha família, que sempre me apoiou em fazer aquilo que eu gostava e ainda gosto de fazer. Agradeço também aos meus colegas da universidade, Luan Henrique, Henrique Berilli, David Ferreira, e todos os colegas que estiveram comigo, que hoje se tornaram amigos, por todo o tempo que passamos juntos nesta luta para seguir nossos sonhos. E, por fim, agradeço àqueles que estiveram comigo durante esta longa caminhada, mas já não estão mais comigo.

"The limit is your imagination."
(Desconhecido)

Resumo

O aumento da complexidade de sistemas que contêm circuitos integrados atingiu níveis que tornam o projeto de *hardware* cada vez mais desafiador, com chips contendo bilhões de transistores. Sendo assim, uma solução para lidar com a complexidade é utilizar o conceito de níveis de abstração. Particularmente, aumentar o nível de abstração de um sistema do RTL (*Register-Transfer Level*) para o nível sistêmico permite a redução de detalhes em sua descrição e, consequentemente, a redução no tempo de projeto. Considerando esse contexto, a realização de síntese em alto nível e a automatização de um fluxo de projeto que permita executar a síntese RTL a partir de descrições em linguagens em alto nível de abstração vêm se tornando cada vez mais necessárias.

Sendo assim, a proposta deste trabalho é a utilização de ferramentas de síntese em alto nível para implementar circuitos simples (inicialmente contadores e ULAs), descritos na linguagem SystemC, em FPGAs. Apesar de simples, esses exemplos mostram os passos necessários para utilizar essa metodologia, que pode ser adaptada para implementar circuitos mais complexos. Os circuitos sintetizados foram testados no FPGA Artix-7 do kit Basys3.

Palavras-chaves: Circuitos Digitais, FPGA, HLS

Abstract

Nowadays, hardware design has become more challenging due to the increasing complexity of electronic systems, which can contain integrated circuits with more than a billion transistors. A feasible solution to deal with complexity is to use the concept of abstraction levels. In particular, increasing the abstraction level of a system from RTL (Register-Transfer Level) to systemic allows to reduce the details in its description and consequently the development time. Considering this context, high-level synthesis and an automated design flow which allows to perform the RTL synthesys starting from a high abstraction level description are becoming increasingly more necessary.

Therefore, the proposal of this work is to use high-level synthesis tools to implement in FPGA simple circuits (initially counters and ALUs), described in SystemC language. Although simple, these examples show the steps required to use this methodology, which can be adapted to implement more complex circuits. The synthesized circuits were tested in the Artix-7 FPGA in the Basys3 kit.

Key-words: Digital Circuits, FPGA, HLS

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

AMS	Anolog/Mixed Signal
ASIC	Application Specific Integrated Circuits
CMOS	Complementary Metal Oxide Semiconductor
FPGA	Field Programmable Gate Array
IP	Intellectual Property
RTL	Register-Transfer Level
VHDL	Very High Speed Integrated Circuit Hardware Description Language
ALU	Arithmetic Logical Unit
UVM	Universal Verification Methodology
TLM	Transaction Level Modeling

Sumário

1 Introdução

Desde que os transistores surgiram nos anos cinquenta, a indústria de microeletrônica tem crescido de forma significativa no mundo em todos os setores, principalmente após o aparecimento do primeiro microprocessador de uso geral em meados dos anos setenta [Tanenbaum (2007)]. Curiosamente, apesar de ser apenas uma previsão estatística, essa indústria vem seguido a “Lei de Moore” desde os anos sessenta, a qual apresentava uma projeção futura de que a cada dois anos o número de transistores dentro de chip dobraria [Schaller (1997)]. Apesar da capacidade de processamento dos circuitos ter aumentado drasticamente ao longo dos anos, a sua complexidade de projeto também crescia a uma proporção ainda maior. Como exemplo, no inicio de 1995 os primeiros microprocessadores com palavras de 64 bits surgiam no mercado, contendo alguns milhões de transistores. Hoje, pouco mais de vinte anos depois, é possível encontrar circuitos contendo mais de 15 bilhões de transistores [NVIDIA (2016)]. Chips com essa complexidade, e que tendem a crescer cada vez mais, necessitam de métodos de produção, desenvolvimento e verificação cada vez mais sofisticados para que as empresas consigam projetar e implementar seus sistemas em tempo hábil e continuar competitivas no mercado.

Comumente, o projeto de um *hardware* de alta complexidade é dividido em duas partes. A primeira é responsável pela parte da fabricação física do chip, ou, em outras palavras, tudo o que envolve diretamente o processo de fabricação do circuito, desde a escolha dos materiais utilizados para a construção do circuito e processos de fabricação, até o aperfeiçoamento da tecnologia dos componentes (normalmente transistores em circuitos digitais). A segunda parte é responsável pelo projeto lógico do sistema digital, onde são definidas as funcionalidades internas e externas, bem como suas entradas e saídas, além do processo de verificação funcional e simulações físicas [CALAZANS (1995)].

No decorrer dos anos, os avanços da tecnologia de fabricação do chip cresceram de forma significativa, o que permitiu a criação de chips com transistores que chegam a ter 16 nm de comprimento do canal, reduzindo a área total, as capacitâncias parasitas, a energia dissipada e consumida pelo circuito, além de aumentar a velocidade de transferência de sinal. Em contrapartida na perspectiva do projetista responsável por desenvolver a parte lógica do sistema, diversos avanços ocorreram no desenvolvimento de ferramentas, bem como mudanças de paradigmas consideráveis durante o mesmo período. Pode-se dividir este tempo em três períodos: a era da "Invenção", que ocorreu no inicio da indústria de microeletrônica, com o aparecimentos dos primeiros CIs (Circuitos Integrados) contendo centenas de transistores, projetados à mão; a era da “Implementação”, que foi onde começaram a surgir as primeiras ferramentas de otimização, síntese e verificação de *hardware*, bem como os algoritmos e técnicas para que elas fossem implementadas; e, finalmente, a

era da “Integração”, onde os passos para o desenvolvimento do circuito estão integrados e são executados por ferramentas (EDAs) [Lavagno, Scheffer e Martin (2006)].

A mudança mais significativa no processo de desenvolvimento de *hardware* está na criação de diversos níveis de abstração de projeto, permitindo ao projetista se concentrar nas particularidades de cada nível. No nível mais baixo (físico), o projetista se preocupa em como serão colocados os transistores e sua tecnologia. Já no nível arquitetural, o projetista se concentra no comportamento dos componentes digitais (registradores, ULA, multiplexadores, etc.) que compõem o sistema [CALAZANS (1995)].

Uma das formas para melhorar o desenvolvimento lógico do circuito é aumentando o nível de abstração [Ghenassia et al. (2005)]. Atualmente, o nível de abstração mais alto é onde o projetista se dedica às funcionalidades do sistema, sem muitos detalhes técnicos de como o mesmo será implementado. Entretanto, um dos grandes problemas é que, aumentando o nível de abstração, necessita-se também de um meio para sair de um nível para outro até finalizar o projeto. Existem diversas ferramentas que fazem essa “ponte”, ou a síntese, do circuito para outro nível. No entanto, até pouco tempo atrás, poucas ferramentas faziam a síntese de um circuito em alto nível de forma eficiente, requerendo a atuação manual direta do projetista no processo, algo comumente visto em circuitos de microeletrônica analógica.

1.1 Contextualização

O número de aplicações que usam FPGA vem aumentando, bem como sua importância no mercado de microeletrônica, como é possível perceber pela recente aquisição da Altera pela Intel. Uma das principais razões é o fato dos projetistas poderem programar e reprogramar a placa de acordo com suas necessidades. Entretanto, uma limitação considerável é o processo de aprendizagem de desenvolvimento de *hardware* com FPGA, no qual são utilizadas linguagens de descrição de *hardware* (HDL), como VHDL e Verilog, para criar projetos em nível RTL. Isso requer um conhecimento maior para especificar a funcionalidade do sistema em baixo nível de abstração, no qual o seu comportamento é detalhado a cada ciclo.

Apesar desse processo ter sido facilitado pela disponibilidade crescente de *IP cores*, o aumento da complexidade dos sistemas e das demandas do mercado vem exigindo o apoio de ferramentas que possam propiciar uma redução ainda maior do tempo de desenvolvimento. Nesse sentido, as ferramentas de síntese em alto nível (HLS) tem se mostrado uma boa alternativa para melhorar a produtividade, visto que permitem usar códigos-fonte descritos em C/C++, ao invés de conhecimentos avançados de *hardware*, para gerar automaticamente arquivos RTL otimizados para um determinado dispositivo.

Sendo assim, a combinação de ferramentas HLS com projetos voltados para FPGA

é particularmente interessante, visto que as implementações de *hardware* podem ser facilmente refinadas e reprogramadas nos dispositivos. Seu uso tem se tornado mais frequente tanto na indústria quanto na academia, permitindo um processo de otimização rápido e sem custos adicionais, visto que projetistas de *hardware* podem verificar e testar funcionalmente o hardware descrito, além de testar em um protótipo funcional e poder realizar mudanças rapidamente, e projetistas de *software* podem participar, sem necessariamente precisar aumentar seu conhecimento de *hardware*, dendo capaz de testar o software em um hardware descrito e em protótipo real.

No contexto do curso de Engenharia Eletrônica da Faculdade do Gama da Universidade de Brasília, observa-se que o uso de ferramentas HLS ainda não foi introduzido nas disciplinas da área. O desenvolvimento de *hardware* com FPGA é ensinado nas disciplinas Prática de Eletrônica Digital 1 e 2 (obrigatórias) e Projeto com Circuitos Reconfiguráveis (optativa). Em todos os casos, o fluxo de projeto é iniciado com a descrição RTL dos sistemas, usando-se a linguagem VHDL. Para ajudar a suprir essa lacuna, e dada a importância do tema, a proposta deste trabalho é propiciar uma abordagem inicial da síntese em alto nível para FPGA, de maneira que este material possa ser aprofundado e futuramente utilizado por alunos dessas disciplinas.

1.2 Objetivos Geral e Específicos

Considerando esse contexto, o objetivo geral deste trabalho é demonstrar o processo de síntese em alto nível de circuitos digitais de baixa complexidade, iniciando com sua descrição em SystemC, utilizando em seguida a ferramenta *Vivado HLS* (*High Level Syhntesis*) e finalmente implementando os circuitos em FPGA.

Os objetivos específicos são:

- Escolha de circuitos dependentes de clock e lógicos/aritméticos para serem descritos em SystemC;
- Comparação entre os resultados da síntese dos circuitos descritos utilizando as ferramentas *Vivado HLS* e *Vivado Design*;
- Escrita de um tutorial de fluxo de projeto para FPGA usando HLS.

1.3 Estrutura do Trabalho

O Capítulo 2 apresenta os fundamentos teóricos, com base na literatura acadêmica, necessários para o desenvolvimento deste trabalho. São abordados os temas de projeto de circuito digitais e seus níveis de abstração, os tipos de linguagens de descrição de *hardware*

e síntese em alto nível. No Capítulo 3, são abordados os aspectos metodológicos de projetos de circuitos digitais com FPGA, bem como a ferramenta que será utilizada para realizar o projeto. No Capítulo 4, são descritas as codificações dos circuitos a serem utilizados como demonstração. No Capitulo 5, são apresentados os resultados da síntese e dos testes das implementações. Finalmente, o Capítulo 6 contém as conclusões e propostas para a continuidade do trabalho.

2 Fundamentação Teórica

2.1 Projeto de um Circuito Digital e Níveis de Abstração

O avanço da complexidade dos chips cresce quase que de maneira exponencial em relação ao seu desempenho, o que tem se tornado um grande problema para que os projetistas continuem projetando chips na mesma velocidade, capacidade e qualidade que o mercado necessita [Gajski et al. (2009)]. Uma forma de tentar resolver este problema é aumentando-se o nível de abstração do projeto e usando-se boas ferramentas (EDAs) de descrição, simulação, verificação e síntese de *hardware*.

Um projeto de um circuito digital pode ser dividido hierarquicamente em vários níveis de abstração. Cada nível representa as mesmas informações, de formas diferentes, sendo que o nível mais alto possui uma quantidade de detalhes inferior à do nível mais baixo [Moreno, Ares e Calazans (2004)]. Dessa forma, ao se dividir o projeto de um sistema em diversos níveis, pode-se estabelecer como o nível mais alto aquele que contém a descrição de como o sistema atuará, ou seja, da sua funcionalidade, e o nível mais baixo como o chip físico, com os seus componentes colocados na ordem necessária para atender às funcionalidades definidas no nível mais alto.

Para entender melhor como dividir um projeto em níveis hierárquicos de abstração, utiliza-se o diagrama em Y proposto por Gajski e Kuhn em 1983, que pode ser visto na Figura 1 [CALAZANS (1995)].

Neste diagrama, é possível notar a divisão de um projeto em três vias ou domínios (Funcional, Estrutural e Físico), com diferentes níveis, representados pelos círculos [CALAZANS (1995), CGajski et al. (2009)]:

- **Nível de Circuito (Elétrico)** – Trata-se do nível mais baixo. Descreve a tecnologia utilizada, tal como cada transistor interagirá segundo as leis físicas (e equações), e qual é a sua disposição na placa.

- **Nível Lógico (Lógico)** – Aqui, o sistema é descrito utilizando-se portas lógicas e expressões *booleanas*.

- **Nível de Processo (Arquitetural)** – Neste nível, são descritos componentes um pouco mais complexos com funções simples, como caixas que podem ser usadas em sistemas mais complexos. Alguns exemplos são contadores, multiplexadores e decodificadores.

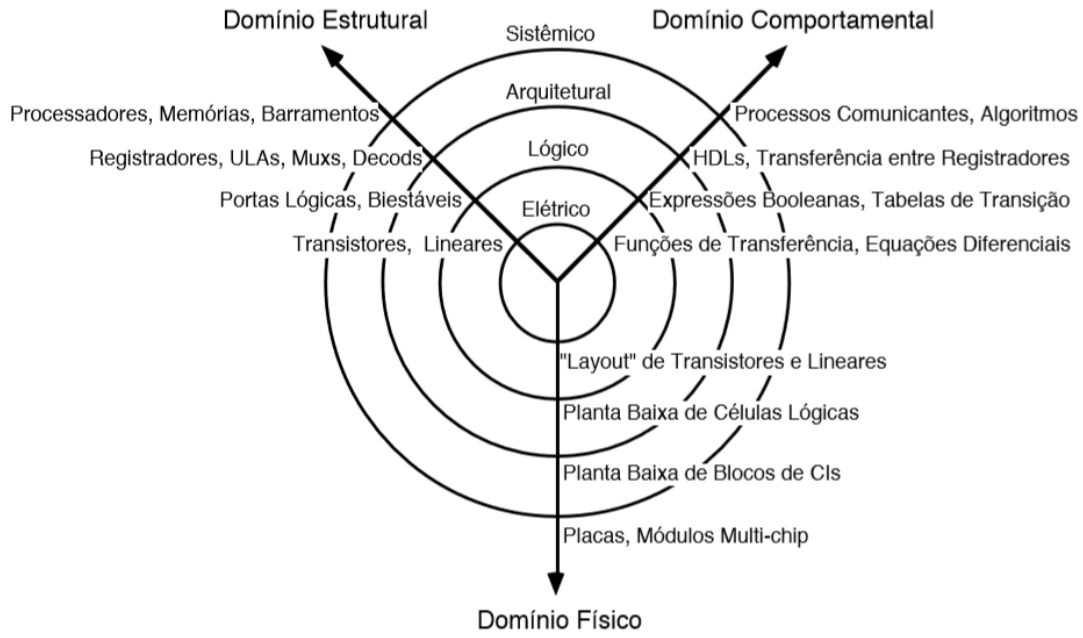

Figura 1: Gráfico Y para Projetos de Circuitos Digitais[CALAZANS (1995), Gajski et al. (2009)]]

- **Nível de Sistema (Sistêmico)** – Este é nível mais alto, no qual o sistema é descrito como um todo, utilizando-se como base os componentes do nível de processo para criar blocos com funções mais complexas. Neste nível, o foco é a funcionalidade do sistema.

Apesar de cada nível possuir um modo para descrever o sistema, cada via pode representar o sistema de uma forma diferente. Por exemplo, no nível mais alto, sistêmico, o circuito é descrito como funcional quando se concentra apenas na funcionalidade, sem se importar em como a mesma será feita. Já do ponto de vista da via estrutural, pode ser representado como um diagrama de blocos. Finalmente, do ponto de vista físico, pode-se ver os blocos de circuitos interagindo fisicamente entre si.

Para o projeto de sistemas complexos, geralmente é utilizada a metodologia de projeto *Top-Down*, que começa com a modelagem do sistema em alto nível de abstração para verificar e validar o sistema nos primeiros estágios, acrescentando-se mais detalhes ao longo do processo de projeto [Kundert e Zinke (2006)]. Assim, aplicando-se esta metodologia ao desenvolvimento de circuitos, pode-se verificar e validar o sistema em estágios iniciais do fluxo de projeto [Beserra (2011)]. Outra grande vantagem de se dividir o projeto em níveis de abstração está na reutilização de módulos IPs (*Intellectual Properties*), que são suficientemente detalhados para validar funcionalmente um ou mais blocos em níveis mais abstratos [Beserra (2011)].

2.2 Linguagens de Descrição de *Hardware*

As linguagens de descrição de *hardware* descrevem o funcionamento de um *hardware* em nível de *software*, tanto para circuitos digitais, quanto analógicos e de sinais mistos (AMS). Com elas, é possível implementar o comportamento de um *hardware* em diferentes níveis de abstração durante o projeto, além de ser possível simular e verificar o *hardware* sem a necessidade de se construir um protótipo físico. Com isso, os custos de projeto são reduzidos e a velocidade de desenvolvimento é aumentada.

Existem várias HDLs, para diversos propósitos e níveis de abstração. As linguagens mais comuns no mercado são VHDL, Verilog, SystemVerilog, C, C++ e SystemC, brevemente descritas nas subseções a seguir. Linguagens de sinais mistos (AMS) são utilizadas para descrever circuitos que necessitam de processamento digital e analógico no mesmo chip. Chips como *tags* RFID, ou sistemas para conexão de Internet sem-fio em dispositivos móveis são alguns exemplos de sistemas de sinais mistos.

2.2.1 VHDL

VHDL, ou VHSIC (*Very High Speed Integrated Circuits*) *Hardware Description Language*, é uma linguagem de descrição geral de circuitos digitais, ou seja, ela pode ser escrita da mesma forma, independente da ferramenta utilizada. No entanto, um lado negativo é a falta do controle de simulações e monitoramento da linguagem, sendo necessário, nesta parte, utilizar uma ferramenta específica para simular o circuito descrito [Bailey (2003)].

É uma linguagem criada a partir da linguagem de programação de algoritmos ADA. A última atualização feita na linguagem foi em 2008 pelo padrão IEEE 1076-2008. O foco da utilização desta linguagem está na reutilização de código, além da portabilidade entre ferramentas.

2.2.2 Verilog

Verilog é uma HDL que nasceu durante os anos oitenta, semelhante ao VHDL, quando os projetistas começavam a usar *software* para projetar circuitos cada vez mais complexos. É uma linguagem amplamente utilizada para descrever e verificar *hardware* digital, analógico e de sinais mistos (AMS). Atualmente, sua versão mais estável foi lançada em 2005, pelo padrão IEEE 1364.

Uma das características mais marcantes desta linguagem são os tipos de atribuições que existem, sendo bloqueante (=) e não bloqueante (<=), o que permite descrever facilmente máquinas de estados. Apesar de ser uma HDL, o Verilog se assemelha com a linguagem de programação C, contendo funções condicionais como "*if/else*", ou *loops*

como "*for/while*.

Em Verilog, é possível criar *modules* (módulos) para definir um bloco de funcionamento. Suas entradas e saídas são definidas por variáveis semelhantes ao C, como inteiros, além de valores binários como *wire*. Além disso, é possível criar blocos de execução paralela com o comando *fork*, de maneira semelhante à criação de processos paralelos de um programa em execução.

2.2.3 SystemVerilog

SystemVerilog veio logo após o Verilog e herdou todas as funcionalidades da versão Verilog-2005. Pode-se ver, então, SystemVerilog como uma extensão da linguagem Verilog, porém com mais funções quanto à verificação, que encontram-se em linguagens como C e C++, como também a definição de mais tipos de dados e a de definição de erros, quando o algoritmo implementado não está correto [Bailey (2003)].

SystemVerilog permite que usuários definam seus próprios tipos de dados, além de trazer tipos abstratos como inteiros e de permitir a conversão de dados (*type casting*), facilitando o uso de dados complexos e reduzindo o erro de codificação por parte de projetistas.

Sendo uma linguagem única que mescla o desenvolvimento de *hardware* com a sua verificação, SystemVerilog tem se tornado cada vez mais popular. Um dos motivos é a sua utilização no padrão universal de verificação UVM (*Universal Verification Methodology*), que permite verificar o *hardware* desenvolvido em várias outras linguagens além de SystemVerilog, como VHDL e SystemC.

2.2.4 SystemC e TLM (*Transaction-Level Modeling*)

SystemC não é uma linguagem em si, mas uma extensão para a linguagem de programação C++, criada para projetar e verificar *hardware* modelado em alto nível de abstração, mas que também pode descrever sistemas em nível RTL (ou arquitetural)[Black et al. (2009)]. As principais vantagens na utilização desta biblioteca são os tipos de dados para descrição de *hardware*, e seus tipos de processos que viabilizam o paralelismo existente no *hardware* [Beserra (2011)]. Como se trata da linguagem C++, outra vantagem está no fato de que a sua simulação pode ser gerada a partir de um arquivo executável, que pode ser compilado na maioria dos sistemas operacionais sem a adição de nenhum *software* específico.

Atualmente, o padrão SystemC é mantido pela empresa Accellera [Accellera (2016)], definido pelo padrão IEEE 1666-2011 [IEEE... (2012)] e de código-fonte livre. A adoção do SystemC pelas empresas se dá pela interoperabilidade entre *software* e *hardware* em nível de projeto, sem a necessidade de se produzir de fato um chip para que seja testado

o *software* embarcado. Assim como a linguagem Verilog, SystemC também possui uma versão AMS (SystemC-AMS) para a modelagem de sistema de sinais mistos.

Como se trata de C++, a descrição de um sistema em SystemC utiliza as mesmas regras de programação, tais como orientação a objetos, porém utiliza mecanismos próprios para a descrição de um *hardware* e não de um algoritmo. Sendo assim, a estrutura que define um bloco que possui alguma função é um módulo. Por exemplo, um filtro FIR pode ser um módulo que pode ser o sistema desejado a ser descrito ou parte de um sistema maior (outros módulos). Um módulo é, na verdade, uma classe em C++ que herda as funções da classe descrita na biblioteca SystemC, (**sc_module**). Como se trata de uma classe, pode-se definir os seus atributos como sendo sinais de entrada e saída do módulo, as funções a serem executadas, e o construtor, (**SC_CTOR**) ou (**SC_HAS_PROCESS**) no caso do SystemC, que define os tipos de processos e os tipos de sensitividade a eventos [Beserra (2011)].

Os processos definem quais as funcionalidades que o sistema executará, são sensíveis a sinais e são passíveis de simulação do comportamento de um *hardware*. Existem três tipos de processos [Black et al. (2009), [Beserra (2011)]:

- **SC_METHOD**: Declarados como funções em `C++`, são sensíveis a determinados eventos e limitados a um ciclo. É comumente usado para definir sistemas sequenciais simples.
- **SC_THREAD**: É executado por um período indeterminado de tempo, podendo conter *loops* infinitos e podendo ser suspensos pelo comando **wait**. É muito utilizado para descrição de *testbenchs*. Não é sintetizável.
- **SC_CTHREAD**: Totalmente dependente do *clock*, é utilizado na maioria das descrições de *hardware* em alto nível. Diferente do **sc_thread**, é sintetizável e pode utilizar mais de um sinal de *clock* para executar uma função. Também é muito utilizado para descrever máquinas de estado.

Em SystemC, módulos se comunicam utilizando portas, interfaces e canais. Uma porta se comunica com outra conectando módulos, e é ligada a um canal por meio de uma interface. Interfaces são classes abstratas virtuais, e os canais herdam as interfaces [Black et al. (2009)].

Entretanto, um grande gargalo no desenvolvimento de *hardware* é a comunicação entre os diferentes módulos funcionais criados. Mesmo que o nível de abstração seja alto, a simulação funcional de um sistema descrito em uma linguagem HDL depende totalmente de como cada bloco se comunicará com outros blocos, do tamanho da palavra e do que cada valor representa. A descrição de como cada bloco se comunicará acaba fazendo com

que o projetista detalhe o sistema e fuja do escopo do que é o nível de sistema, que significa menos detalhes. Para resolver este problema, surgiu o padrão TLM.

O nível de transação está no circulo de nível sistêmico no gráfico Y [Moreno, Ares e Calazans (2004)]. A proposta da modelagem de sistemas em TLM é de servir como se fosse um protocolo de barramento, comunicação entre os módulos do sistema. Pode-se definir uma transação como um modo de comunicação de qualquer tamanho de palavra, permitindo assim reduzir o tempo de projeto, o desenvolvimento de *software* embarcado antecipado, além da rápida verificação funcional do sistema [Ghenassia et al. (2005)].

Tradicionalmente, o fluxo de desenvolvimento dos sistemas era feito sem que projetistas de *hardware* e de *software* interagissem entre si até que o primeiro protótipo fosse finalizado, tornando o desenvolvimento lento, principalmente do ponto de vista do *software*, que dependia da especificação do *hardware* para ser produzido. Outro grande problema estava no custo da prototipagem. Caso houvesse algum erro (e as chances são grandes sem uma interação entre os times) tanto no *software* quanto no *hardware*, o processo retornaria praticamente ao início [Ghenassia et al. (2005)].

A introdução do TLM permitiu uma mudança fundamental no fluxo de desenvolvimento, em que os times de desenvolvimento de *hardware* e de *software* trabalham ao mesmo tempo, tendo como referência funcional o nível TLM. O time de desenvolvimento de *software* não mais necessita esperar o protótipo para testar suas funcionalidades, permitindo assim reduzir os custos e o tempo de desenvolvimento do sistema.

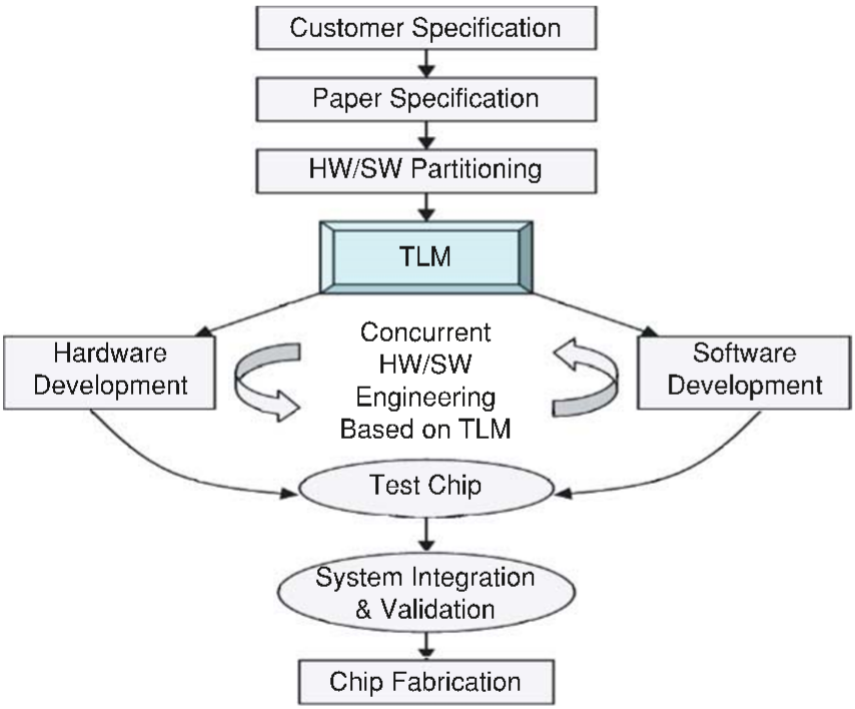

Figura 2: Fluxo de Projeto com TLM [Ghenassia et al. (2005)]

Atualmente, a biblioteca SystemC suporta a versão TLM 2.0.1 [Accellera (2016)], que define interfaces para que as transações ocorram [Beserra (2011)]. Conforme será mostrado na seção a seguir, além de ser uma excelente opção para a modelagem de sistemas em alto nível para validação da funcionalidade usando TLM, o SystemC também pode ser usado para a implementação do sistema usando ferramentas de síntese em alto nível.

2.3 Síntese em Alto Nível

A modelagem de sistemas em alto nível (sistêmico) vem sendo feita desde o final dos anos setenta. No entanto, um grande *gap* existia entre o nível de sistema e o nível RTL. Poucas ferramentas faziam a síntese em alto nível para o nível RTL.

Síntese em Alto Nível, ou *High Level Synthesis*, é o processo que interpreta um sistema descrito funcionalmente em uma linguagem de alto nível (normalmente C, SystemC, C++ e Matlab), e que o implementa em *hardware*. A descrição em alto nível permitiu acelerar o desenvolvimento de circuitos digitais, uma vez que sua sintaxe é mais flexível e permite abstrair mais detalhes que são necessários no nível RTL como a comunicação entre blocos (TLM no caso de SystemC).

No caso de um circuito integrado, a ferramenta HLS faz a síntese funcional, e o *layout* pode ser obtido realizando as sínteses lógica e física do circuito. o caso de projetos voltados para FPGA, a síntese em alto nível serve para acelerar o processo de desenvolvimento de *IP cores*, possibilitando a síntese para FPGA de modelos em alto nível sem a necessidade de criar manualmente a descrição RTL intermediária.

O capítulo a seguir aborda os aspectos metodológicos e ferramentas comumente utilizadas em ambos os casos.

3 Aspectos Metodológicos e Ferramentas

O sucesso da implementação de um projeto depende da metodologia aplicada durante seu desenvolvimento, ou seja, do fluxo de projeto para se chegar ao produto final. Em se tratando de circuitos digitais, duas formas de implementação são as mais comuns: FPGA (*Field Programmable Gate Array*) e ASIC (*Application-Specific Integrated Circuit*). A Figura 3 apresenta um gráfico mostrando o fluxo de projeto para ambas as implementações, e as seções a seguir descrevem brevemente as etapas de cada caso.

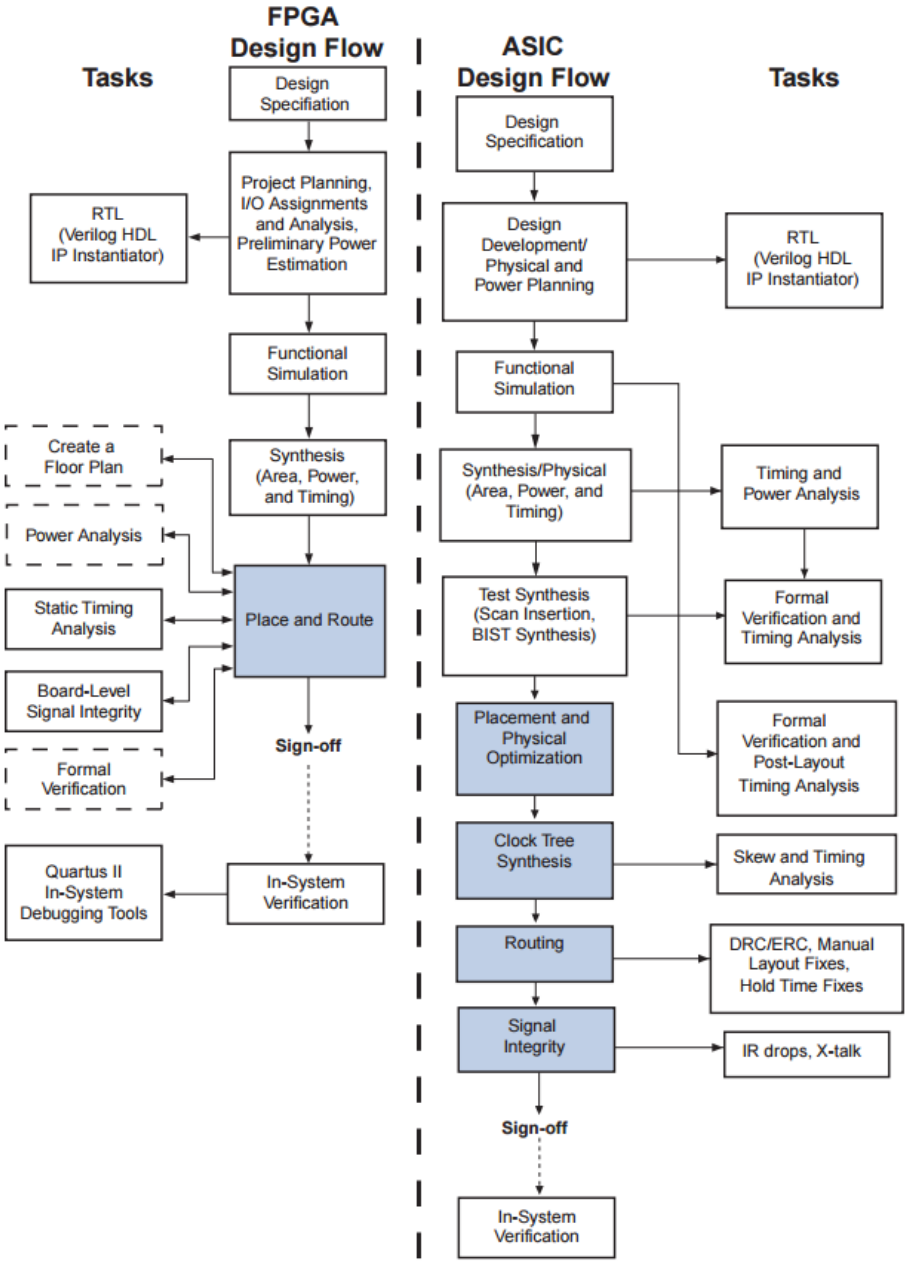

Figura 3: Fluxo de Projeto FPGA - Altera/ASIC [Altera (2016)]

3.1 ASIC

ASIC é um circuito integrado para uma aplicação específica. Muito utilizado na produção de sistemas embarcados, possui um custo elevado em relação a outras metodologias. No entanto, suas aplicações tendem a ser mais eficientes.

As fases mais importantes deste fluxo são: Especificação, Codificação, Simulação Funcional, Síntese Lógica, Simulação Pós-Síntese Lógica, Síntese Física, Simulação Pós-Layout [Franzon, Perelstein e Hurst (1999)]. Cada fase é brevemente descrita a seguir.

1. **Especificação**: Independente do tipo de projeto, a especificação é sempre a parte mais crítica. É onde se determina o que se quer e planeja-se os próximos passos para se obter o resultado desta especificação.

2. **Codificação**: Aqui, é realizada a primeira descrição funcional do circuito a ser projetado com base nas especificações criadas. Pode ser criado tanto em nível RTL quanto em nível de sistema.

3. **Simulação Funcional**: Entrando para a área do processo de verificação, aqui é realizada a verificação funcional do circuito, ou seja, verifica-se se a descrição realizada na etapa anterior realmente atende todos os requisitos da especificação. O sucesso do projeto depende deste passo. Caso o circuito não seja verificado de forma correta e apresente um erro posteriormente, o custo de projeto subirá consideravelmente.

4. **Síntese Lógica**: Neste passo, são criados os blocos de circuitos lógicos internos do sistema como um todo, utilizando as bibliotecas da tecnologia na qual o circuito será fabricado [Lavagno, Scheffer e Martin (2006)].

5. **Simulação Pós-Síntese Lógica**: Utilizando-se o mesmo ambiente de verificação para a simulação funcional, é realizada outra simulação funcional, porém contendo mais informações de atraso do circuito entre os blocos funcionais.

6. **Síntese Física**: É o passo final para o desenvolvimento do circuito. Nesta etapa, o mesmo é organizado em nível de componentes, indicando como o circuito será criado fisicamente.

7. **Simulação Pós-Síntese Física**: Último passo de verificação, é feito usando-se o mesmo ambiente de verificação funcional. Aqui, é realizada uma verificação quanto às funcionalidades do sistema, levando em consideração os atrasos criados pelos blocos de circuitos e atrasos relacionados à geometria física.

3.2 FPGA

A utilização de FPGA para o desenvolvimento de circuitos permitiu uma redução considerável no custo e tempo de projeto de circuitos digitais de pequeno porte, principalmente por permitir a simulação real em um protótipo funcional de maneira muito mais rápida do que produzindo um chip. Além disso, é possível encontrar no mercado FPGAs capazes de implementar grandes blocos de circuitos que antes só podiam ser implementados utilizando-se a metodologia para ASIC [Altera (2016)].

As etapas do fluxo de projeto de um circuito digital a ser implementado um em FPGA são, de forma resumida:

1. **Especificação, Codificação e Simulação Funcional**: Os primeiros três passos são semelhantes aos da metodologia ASIC. Especifica-se o sistema a ser implementado e, logo após, descreve-se o sistema de acordo com as especificações em uma linguagem de descrição de *hardware* ou em alto nível, e finalmente realiza-se a primeira simulação para validar as funcionalidades do sistema descrito.

2. **Síntese e Implementação**: É realizado o primeiro processo de síntese lógica do sistema, incluindo-se as informações quanto à área dos blocos e ao tempo de atraso entre os mesmos.

3. **Programação e Verificação na Placa**: Após a verificação funcional pós-síntese, programa-se a placa FPGA com o circuito projetado e realiza-se a última verificação funcional do sistema em um protótipo real.

Apesar de ser implementado em menos passos do que na metodologia ASIC, um circuito projetado para um FPGA não necessariamente será desenvolvido mais rapidamente. Entretanto, como é possível implementar e reutilizar a mesma placa, o FPGA é muito utilizado como plataforma para desenvolvimento de *hardware* e teste de funcionalidade.

3.3 Ferramentas HLS

Como descrito no Capitulo 1 deste trabalho, a complexidade de circuitos digitais tomou proporções em que o seu projeto tornou-se mais difícil utilizando-se métodos tradicionais. De acordo com Scheffer, Lavagno e Martin [Lavagno, Scheffer e Martin (2006)], nenhum chip composto por mais de um bilhão de transistores haveria sido feito sem a utilização de ferramentas EDA.

Existem diversas ferramentas de síntese em alto nível hoje em dia, como por exemplo o STRATUS HLS (Cadence), o Synphony C Compiler (Synopsys), para implementa-

ção em ASIC ou FPGA, e o Vivado HLS da Xilinx, para implementação em FPGA e o Catapult HLS (Mentor Graphics).

3.3.1 Cadence

Como uma das maiores empresas no ramo de produção de *software* voltado ao desenvolvimento de circuitos em microeletrônica, possui um conjunto de ferramentas para a realização de todas as sínteses de um projeto ASIC. Dentre elas, pode-se citar três: *STRATUS HLS*, *Encounter RTL Compiler* e *SoC Encounter*.

3.3.1.1 *STRATUS HLS*

É a evolução da ferramenta CTOS (*C-to-Silicon Compiler*). Possui como objetivos a descrição de um sistema em alto nível (C, C++, SystemC) e a implementação de um circuito em nível RTL, reduzindo o tempo de projeto, uma vez que o projetista precisa descrever o circuito apenas uma vez[Cadence (2016a)]. A Figura 4 mostra o fluxo de projeto dessa ferramenta.

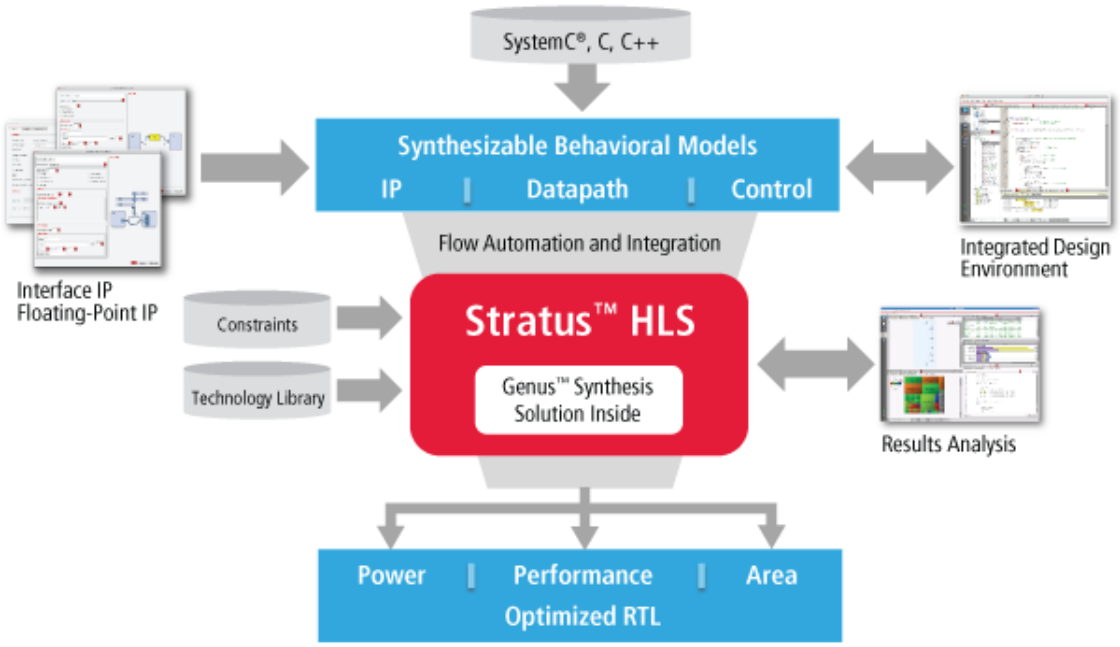

Figura 4: Fluxo HLS no Ambiente Cadence [Cadence (2016a)]

3.3.1.2 *Encounter RTL Compiler*

O objetivo desta ferramenta é a realização da síntese lógica de um circuito. Em outras palavras, é transformar um código RTL, que pode ter sido o resultado da utilização de uma ferramenta HLS, para criar a descrição lógica do sistema [Cadence (2016b), Albrecht et al. (2006)]. O fluxo de projeto desta ferramenta pode ser visto na Figura 5.

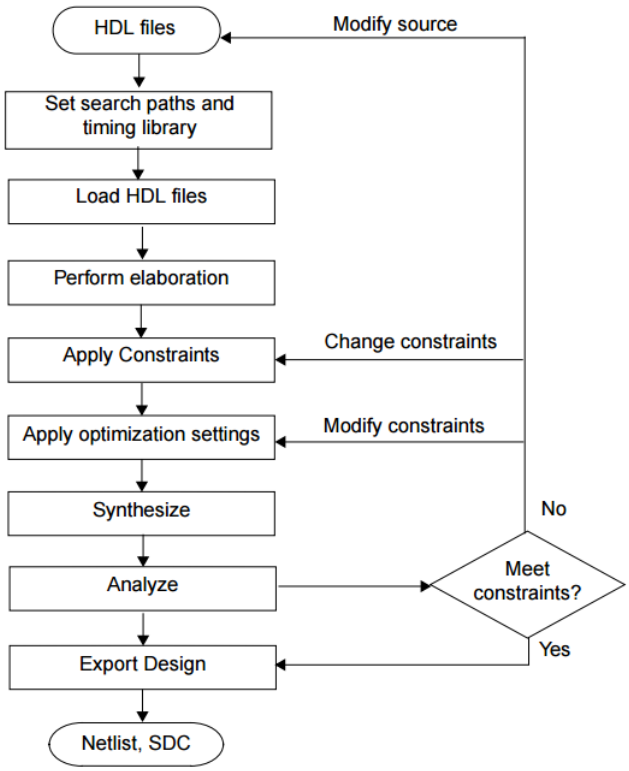

Figura 5: Fluxo de Projeto no Cadence Encounter [Cadence (2016a)]

Esta ferramenta utiliza arquivos da tecnologia na qual o circuito será implementado para gerar as primeiras simulações quanto ao atraso do circuito.

3.3.1.3 *SoC Encounter RTL-to-GDSII System*

Esta ferramenta tem como objetivo realizar a síntese física do sistema, ou seja, utiliza os *netlists* criados pelo *Encounter RTL Compiler* para projetar o posicionamento físico dos componentes em um chip e criar o *layout*.

A criação do *layout* leva em conta a posição e tamanho dos dos componentes do circuito, bem como a distribuição da rede elétrica e o posicionamento de acordo com o *clock*, para que todos recebam o sinal ao mesmo tempo, ou o mais próximo disto.

3.3.2 Vivado

Este ambiente foi criado pela empresa Xilinx com o objetivo de fornecer suporte para o desenvolvimento de circuitos em FPGAs. Ele possui três programas: *Vivado System*, *Vivado Design* e *Vivado HSL*, sendo este último o que realiza a síntese em alto nível. Assim como o *Cadence Stratus HLS*, ele permite utilizar a descrição de um sistema realizada em alto nível (C, C++, SystemC) e produz um circuito em nível RTL, sem a necessidade de fazê-lo manualmente [Vivado (2016)]. Para isso, suporta a versão 2.1 do SystemC e seu

subset sintetizável 1.3 [SSC (2009)]. Com o *Vivado HLS*, é também possível gerar um *IP core* que pode ser usado no *Vivado System* e no *Vivado Design*.

O *Vivado Design* é usado para os dois últimos passos do fluxo de desenvolvimento de *hardware* com FPGA mostrado na seção anterior: síntese lógica, implementação e verificação do circuito descrito em nível RTL. Ele suporta as linguagens VHDL, Verilog e System Verilog.

A Figura 6 mostra o fluxo de projeto no *Vivado HLS*, no qual uma descrição em C/C++ ou SystemC pode ser simulada e sintetizada. É possível executar a simulação RTL usando-se o mesmo *testbench* da descrição em C/C++ ou SystemC, bem como é possível gerar um *IP core* a partir da síntese.

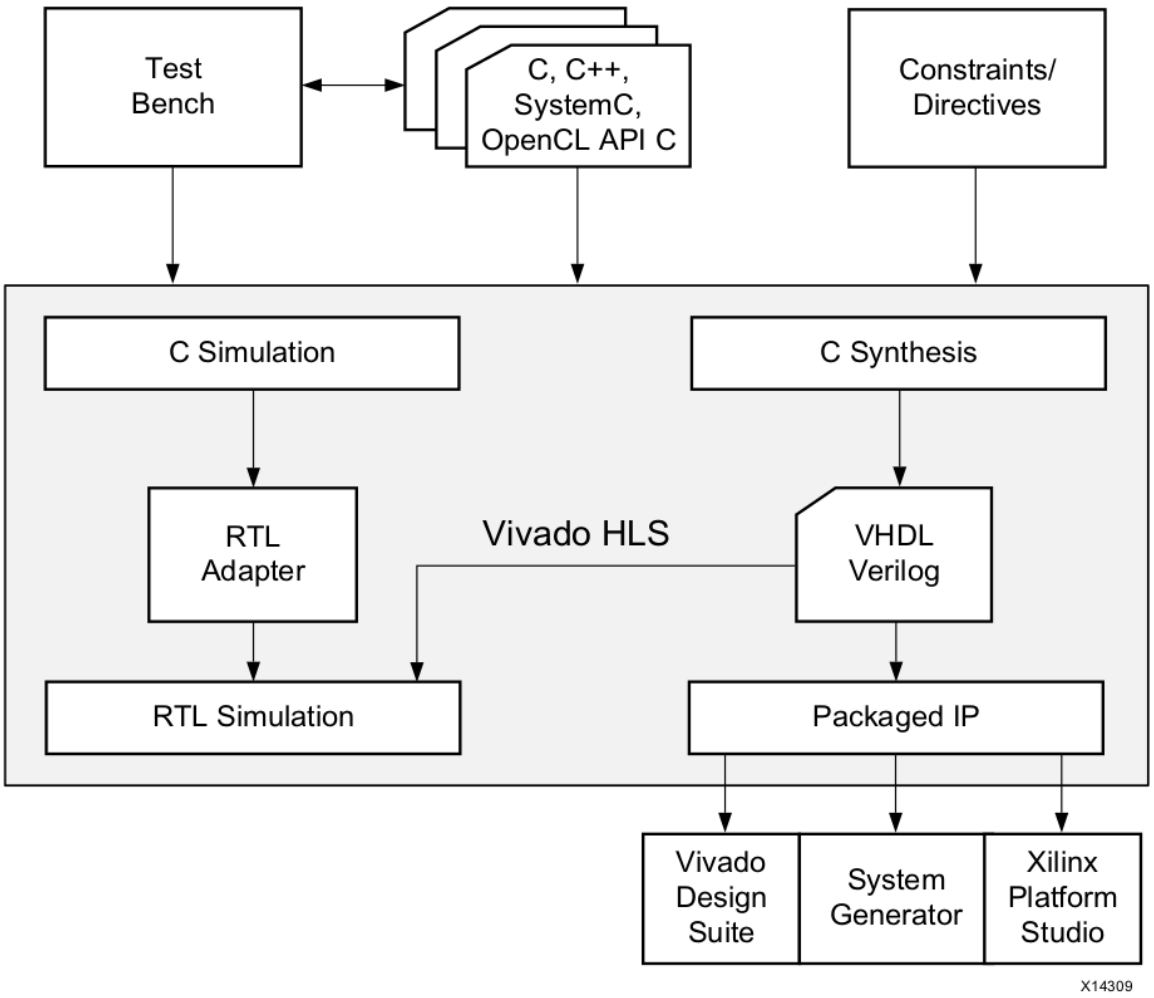

Figura 6: Fluxo de Projeto no Vivado HLS[Vivado (2016)]

No melhor caso possível, como a descrição em alto nível não depende de um programa em si, a compilação de um código criado em alto nível pelas duas ferramentas (*Cadence Stratus HLS* e *Vivado HLS*) permitiria projetar o circuito para aplicação à

qual melhor se encaixa o sistema (ASIC ou FPGA), sem que haja perda de tempo no desenvolvimento do projeto.

Para este trabalho, foram selecionadas as ferramentas *Vivado HLS* e *Vivado Design*, da Xilinx, para implementar e testar em FPGA circuitos descritos na linguagem SystemC.

3.4 Metodologia

A metodologia escolhida para este projeto foi a *top-down*, em que se começa com a especificação e modelagem do sistema em alto nível, passando-se posteriormente para os níveis mais baixos de abstração [Kundert e Zinke (2006)]. Visto que as ferramentas HLS possuem uma série de restrições em relação aos códigos que podem ser sintetizados, não foi possível completar a síntese de circuitos mais complexos. Sendo assim, primeiramente foram especificados circuitos combinacionais e sequenciais de menor complexidade que permitissem demonstrar todos os passos do fluxo de projeto, iniciando-se com a descrição em alto nível até chegar ao teste em um FPGA.

Em seguida, os circuitos foram descritos em SystemC (versão 2.3.2) e simulados utilizando um *testbench* simples para validação visual. Após a validação, foi utilizada a ferramenta *Vivado HLS* para realizar a síntese em alto nível para FPGA. O subproduto dessa síntese foi um *IP core*, que pode ser utilizado em outras ferramentas da família Vivado. Após a síntese, o *IP core* resultante foi utilizado no programa *Vivado Design* com um *wrapper*, ou seja, uma descrição que envolve o *IP core*, e em seguida sintetizado e implementado na placa Basys3. Nesta etapa, também foi utilizado um *testbench*, similar ao criado na primeira etapa de descrição em SystemC, para validação visual dos dados. Por fim, o circuito descrito foi testado na placa Basys3. A Figura 7 mostra o fluxo de projeto utilizado.

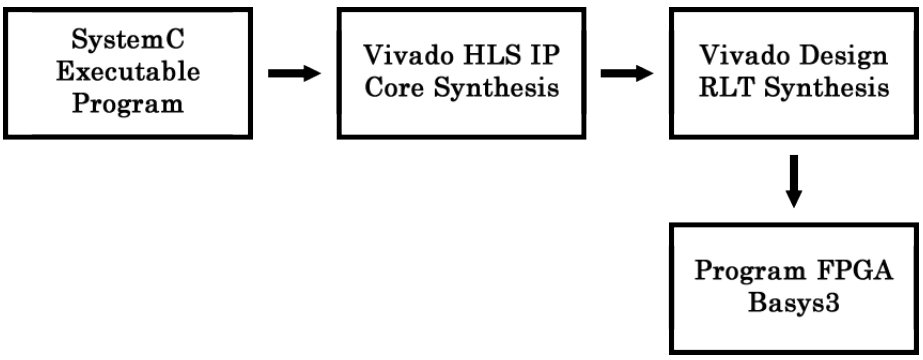

Figura 7: Fluxo de Projeto para a Implementação de Circuitos Digitais em FPGA usando o *Vivado HLS*

4 Descrição de Circuitos Digitais para HLS

Este capítulo apresenta exemplos de descrições em SystemC dos circuitos que foram sintetizados no *Vivado HLS*. Como este trabalho tem fins didáticos e o foco é o processo de realização da síntese, e não o projeto dos circuitos propriamente ditos, foram escolhidos como exemplos circuitos digitais simples, desenvolvidos para serem diretamente sintetizados no *Vivado HLS* sem a necessidade de seguir *coding guidelines*, diretrizes da própria ferramenta que precisam ser observadas na descrição de circuitos mais complexos, a fim de que os mesmos possam ser corretamente sintetizados.

Para realização da síntese, foi utilizado o fluxo de projeto mostrado na Figura 7. A única restrição adotada, que está relacionada ao padrão observado na ferramenta em si, foi a divisão de cada módulo em dois arquivos: o *header* (arquivo **.h**), que contém a descrição de entradas e saídas, o construtor e os protótipos de funções, e o arquivo **.cpp**, que contém as funções declaradas no *header*. Os arquivos aqui descritos podem ser encontrados no GitHub. [1].

4.1 Contador

O contador síncrono foi escolhido como exemplo por ser frequentemente utilizado em circuitos digitais que precisam de um sinal de *clock* para realizar suas funções. Um contador pode ser definido como um somador que incrementa sua saída a cada ciclo de *clock*.

Foram implementados dois contadores: o primeiro é um contador simples, de 10 bits (1024), e o segundo é um contador utilizado como divisor de *clock*.

4.1.1 Contador Simples

Este contador simples incrementa sua variável a cada pulso de *clock*. Sua saída é o valor atual do contador. A Figura 8 mostra o diagrama com as entradas e saídas.

[1] https://github.com/Alexandroni/systemC

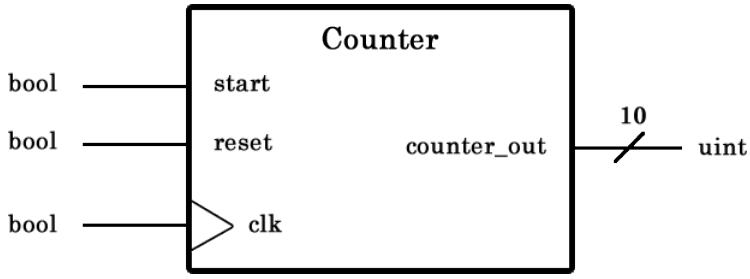

Figura 8: Contador simples de 10 bits

Conforme é possível observar na Listagem 4.1, por se tratar de um circuito dependente de um sinal de *clock*, foi utilizado o processo **SC_CTHREAD**. Diferentemente de um processo que utiliza **SC_METHOD**, pode-se ter uma fase inicial na função que é executada apenas uma vez.

```
1  //SIMPLE COUNTER HEADER--
2  //
3  #ifndef SC_COUNTER_H
4  #define SC_COUNTER_H
5  #include "systemc.h"
6
7  SC_MODULE (simple_counter) {
8
9    //I/O
10   sc_in  <bool> start, clk, reset;
11   sc_out <sc_uint<10> > count_out;
12   //variable auxiliar
13   sc_uint<10>  aux;
14
15   //function to be implemented
16   void counting();
17
18   //constructor
19   SC_CTOR(simple_counter) {
20       //Process clock dependent
21       SC_CTHREAD (counting, clk.pos());
22       //reset funciton
23       reset_signal_is(reset,true);
24   }//end contrusctor
25 };
26 #endif
```

Listing 4.1: Código-fonte do contador simples de 10 bits

A Listagem 4.2 mostra a descrição da funcionalidade do contador, e o resultado da simulação pode ser visto na Figura 9. Para o *Vivado HLS*, foi feita a divisão entre os arquivos *header* (**.h**) e os arquivos **.cpp**, sendo que os primeiros contêm a descrição dos módulos e os últimos, a implementação da função declarada no *header*.

```
27 //SIMPLE COUNTER FUNCTION FILE--
28 //
29 #include "counter.h"
30
31 //function declared on header file
32 void simple_counter::counting(){
33
34     //initiation - this step is execute once
35     aux = 0;
36     count_out.write(aux);
37     wait();
38
39     //infinity loop
40     //clocked thread executes run indefinitely
41     //and stops when reachs a wait statment
42     while(true){
43
44         //start signal
45         if(start.read()){
46
47             //increment the counter
48             aux = aux + 1;
49             //updates the output
50             count_out.write(aux);
51         }
52         wait();
53
54     }//end inifinity while
55
56   }//end counting
```

Listing 4.2: Descrição do contador simples de 10 bits

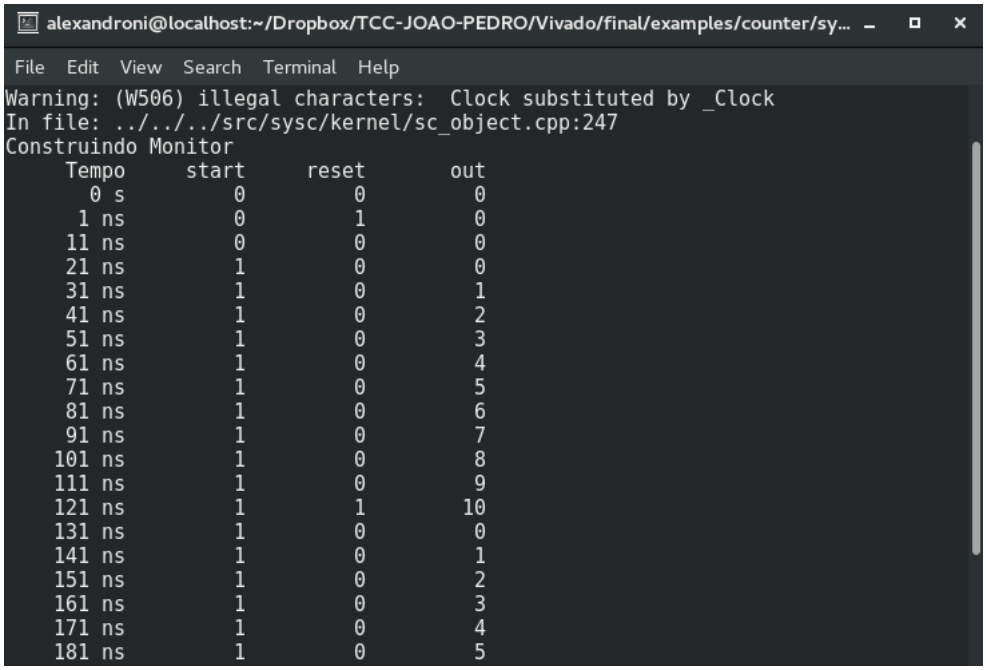

Figura 9: Resultado da simulação do contador simples de 10 bits

A Figura 9 mostra o resultado da simulação realizada sobre a descrição em SystemC do contador simples de 10 bits. É possível avaliar visualmente que o contador obedece as regras descritas. A cada ciclo de *clock*, o contador é incrementado (saída **out**). Além disso, o contador funciona apenas quando a entrada **start** está em nível alto. Quando a entrada **reset** é acionada, o contador reinicia a contagem.

4.1.2 Divisor de Clock

FPGAs normalmente possuem um circuito de *clock* interno, como por exemplo a placa Basys3, que possui um gerador de *clock* interno de 100 MHz. No entanto, diferentes projetos podem precisar de diferentes frequências de *clock*. Assim, um circuito divisor de *clock* é amplamente utilizado em projetos desenvolvidos para FPGA.A Figura 10 mostra o diagrama do divisor que foi implementado como exemplo.

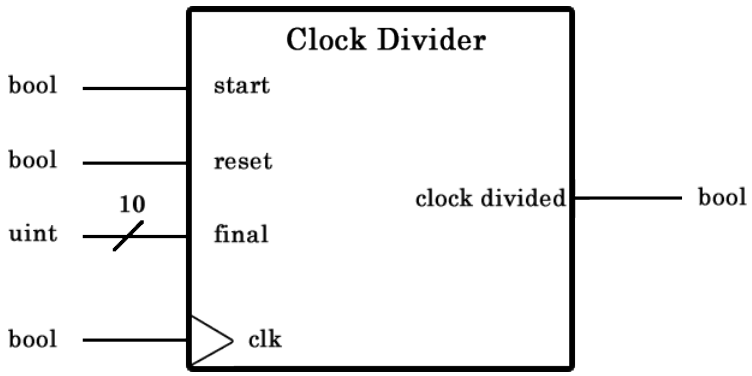

Figura 10: Divisor de clock

O código mostrado na Listagem 4.3 foi descrito com base no contador apresentado anteriormente. A mudança está em relação à sua saída que, ao invés de retornar o valor do contador, retorna os níveis alto ou baixo toda vez que o contador atinge o valor definido pela variável **final**.

```
57 //CLOCK DIVIDER HEADER FILE
58 //
59 #ifndef SC_COUNTER_H
60 #define SC_COUNTER_H
61 #include "systemc.h"
62
63 SC_MODULE (divider) {
64
65   //i/o
66   sc_in  <bool> start, clk, reset;
67   sc_in <sc_uint<10> > final;
68   sc_out <bool> count_out;
69
70   //variable auxiliar
71   sc_uint <32>  aux;
72   bool saida;
73
74   //function to be implemented by the module
75   void counting();
76
77   SC_CTOR(divider) {
78       //clocked thread
79       SC_CTHREAD (counting, clk.pos());
80       reset_signal_is(reset,true);
81   }
82
83 };
84 #endif
```

Listing 4.3: Divisor de *clock*

A função **counting** inverte a saída toda vez que o contador atinge o valor da variável **final**, que, como é de 10 bits, vai até 1024. Sendo assim, esse divisor de textitclock pode dividir o valor em até mil vezes. A Listagem 4.4 mostra a descrição da funcionalidade do divisor de *clock*.

```
85 //CLOCK DIVIDER FUNCTION FILE--
86 //
87 #include "counter.h"
88
89 void divider::counting(){
90
91     //initial phase
92     //executes once
93     aux = 0;
94     saida = false;
95     count_out.write(false);
96     wait();
97
98     //infinity loop
99     //clocked thread executes run indefinitely and stops when ↩
          reachs a wait statment
100    while(true){
101        if(start.read()){
102            //increments the counter
103            aux = aux + 1;
104            //if it reaches the final variable
105            if(aux >= final.read()){
106                //the counter restart
107                aux = 0;
108                //updates the output
109                if (saida == true){
110                    saida = false;
111                }else{
112                    saida = true;
113                }
114                count_out.write(saida);
115            }
116
117        }
118        wait();
119    }//end while (1)
120  }//end counting
```

Listing 4.4: Descrição do divisor de *clock*

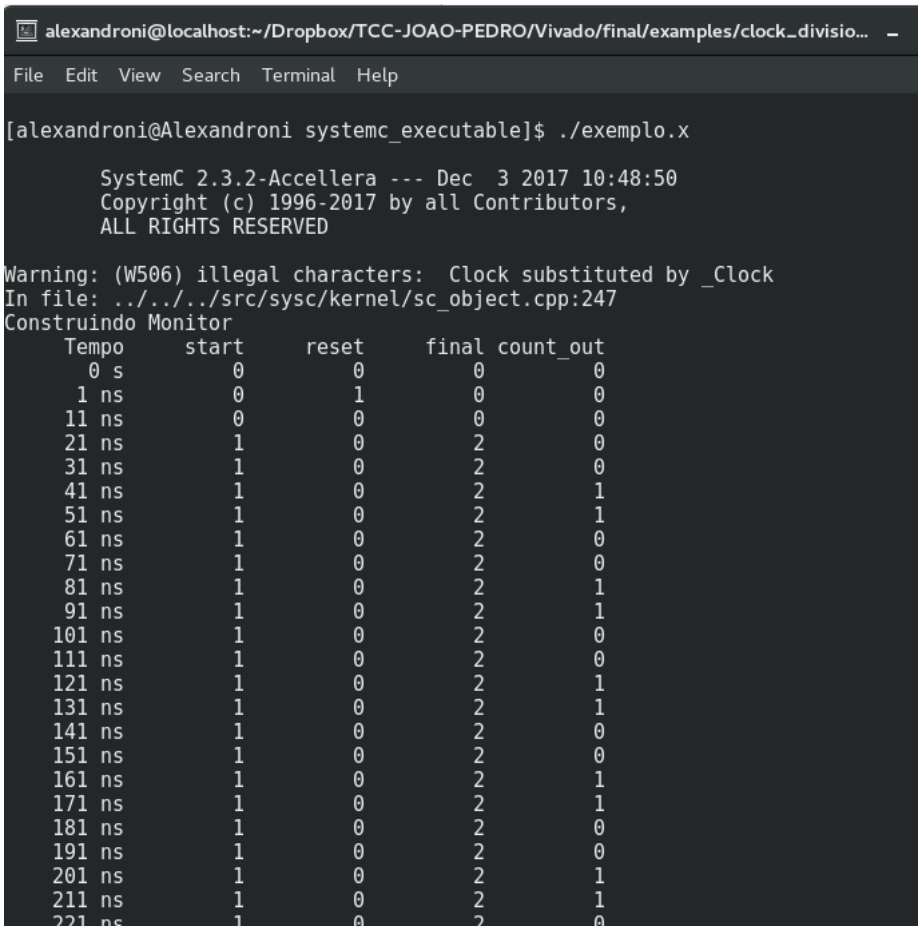

Figura 11: Resultado da Simulação do Divisor de *clock*

A Figura 11 mostra o resultado da simulação sobre a descrição em SystemC do divisor de *clock*. De maneira similar ao contador simples, o divisor de *clock* possui as entradas **reset**, para reiniciar a contagem, e **start**, que habilita o inicio da contagem. Além disso, possui uma entrada de 10 bits declarada como **final**. O valor dessa entrada define até onde o contador vai contar antes de alternar a saída, que dessa vez é do tipo *bool*, e não a saída do próprio contador. Observa-se que o contador segue a saída **final '2'**, contando dois ciclos de *clock* em nível alto, e dois em nível baixo.

4.2 ULA Simples

Hennessy, Patterson e Larus (2000) propõem uma Unidade Lógica e Aritmética (ULA) simples e *'cascateavel'*, ou seja, um circuito que realiza operações de 1 bit podendo cascatear para realizar operações de N bits. O modelo dessa ULA é apresentado na Figura 12 e possui a capacidade de realizar soma e as operações lógicas "E"e "OU". O diagrama correspondente está mostrado na Figura 13.

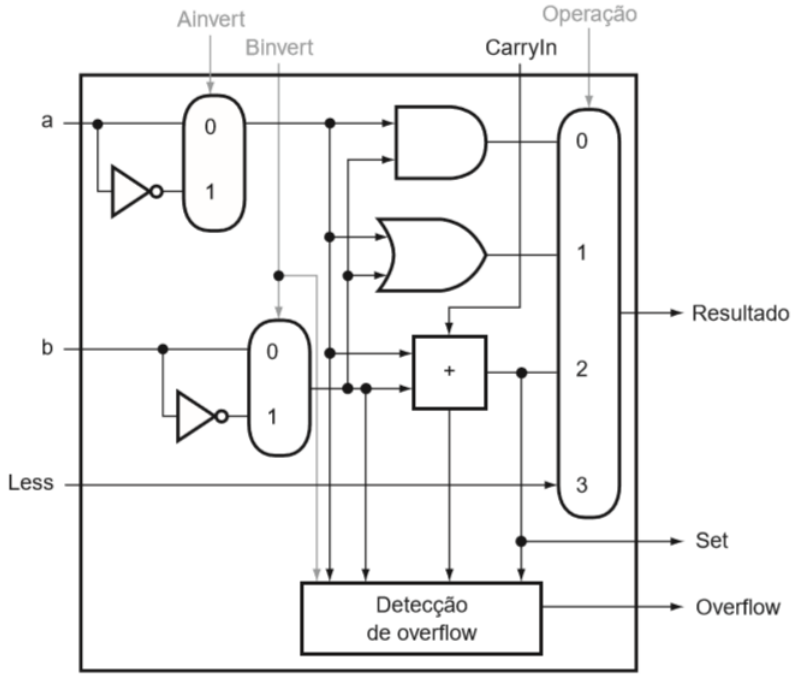

Figura 12: Especificação da ULA Simples de 1 Bit[Hennessy, Patterson e Larus (2000)]

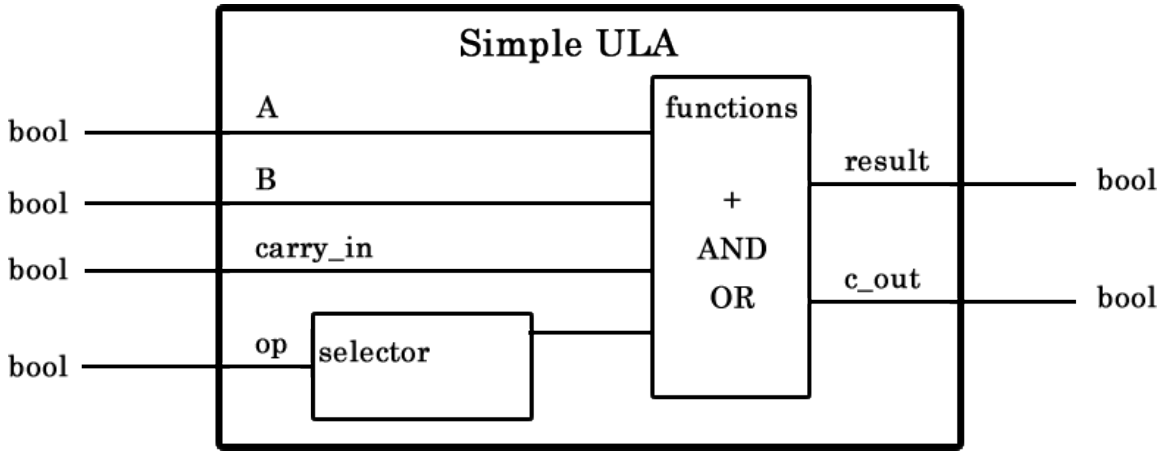

Figura 13: Diagrama da Descrição da ULA simples de 1 bit

A Listagem 4.5 mostra a descrição do circuito proposto na Figura 13, onde a variável **op** representa o seletor de operação mostrado na Tabela 1.

Tabela 1: Operação da ULA simples

op	**Função**
0	-
1	Soma
2	Lógico "E"
3	Lógico "OU"

```
121 //HEADER FILE
122 //
123 #ifndef ULA_H
124 #define ULA_H
125
126 #include "systemc.h"
127
128 SC_MODULE(ula_new)
129 {
130     //i/o
131     sc_in <bool> A, B;
132     sc_in <bool> carryIn;
133     sc_in <sc_uint<2> > op;
134     sc_out <bool> C, carryOut;
135
136     //function to be implemented by the module
137     void opUla();
138
139     SC_CTOR(ula_new)
140     {
141         //sequential method - it executes once
142         SC_METHOD(opUla);
143         //it's sensitive by the inputs
144         sensitive << A << B << carryIn << op;
145     }
146 };
147
148 #endif
```

Listing 4.5: ULA simples

A Listagem 4.6 mostra a descrição da funcionalidade da ULA, e a Figura 14, o resultado da simulação. Da mesma forma que nos códigos das seções **4.1** e **4.2**, foi feito um *testbench* para validar a funcionalidade da ULA simples de 1 bit. Por ser um circuito sequencial, ele responde às mudanças nas entradas (lista de sensibilidade). Foram verificados os resultados para todas as operações.

```
149 #include "ula.h"
150 //SC_METHOD executes once the sensitive list changes
151
152 void ula_new::opUla(){
```

```
153
154     //function variables
155     bool aux, cAux = false;
156
157     //select which funciton to be performed
158     switch (op.read()){
159         case 1:
160             //Add
161             aux = A.read() ^ B.read();
162             cAux = A.read() && B.read();
163             break;
164         case 2:
165             //AND
166             aux = A.read() && B.read();
167             break;
168         case 3:
169             //OR
170             aux = A.read() || B.read();
171             break;
172         default:
173             aux = 0;
174             break;
175
176     }//end switch case
177
178     //ouput
179     C.write(aux);
180     carryOut.write(cAux);
181
182 }//end opUla
```

Listing 4.6: Descrição da Funcionalidade da ULA simples

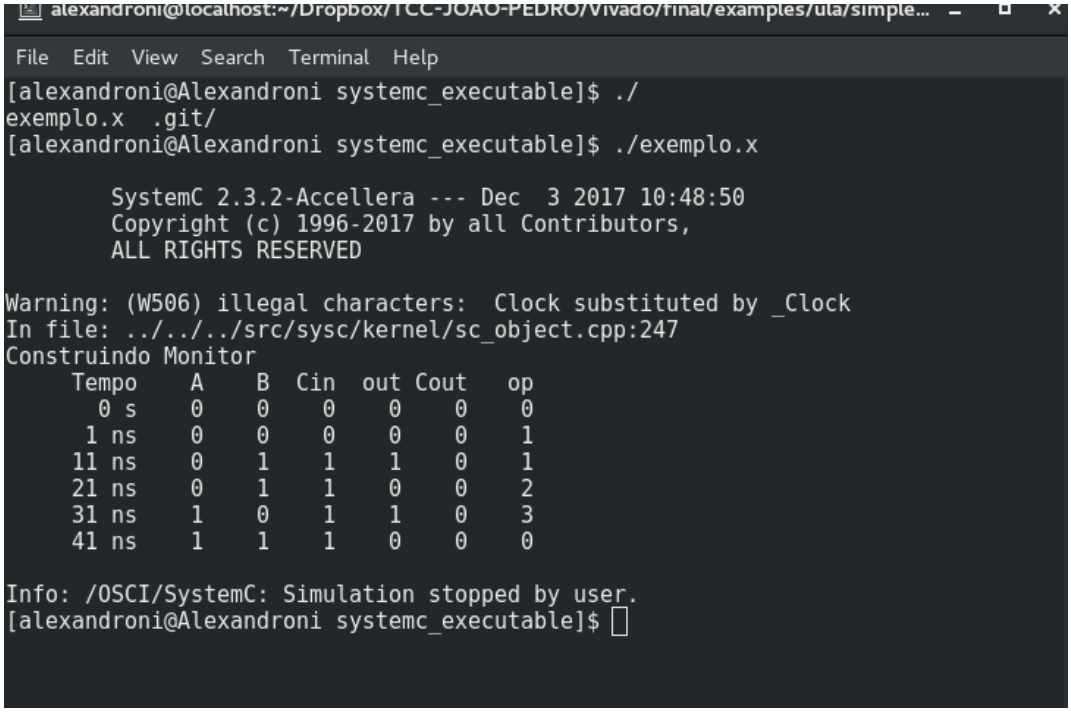

Figura 14: Resultado da Simulação da ULA Simples

4.3 ULA de 16 Bits

A representação de uma ULA de 16 bits a partir do cascateamento da ULA simples descrita na subseção anterior foge do objetivo que se deseja atingir utilizando uma linguagem de descrição em alto nível. Percebe-se que, nesse caso, sua descrição se assemelharia muito a descrições realizadas em nível RLT.

Assim, foi realizada a descrição de uma ULA de 16 bits conforme mostram as Listagens 4.7 e 4.8. A ULA possui duas entradas inteiras de 16 bits, sendo que podem haver números negativos, e realiza cinco operações: adição, subtração, multiplicação, maior que e a operação lógica AND. O resultado é um inteiro também de 16 bits.

```
184 #ifndef ULA_H
185 #define ULA_H
186
187 #include "systemc.h"
188
189 //Module ULA 16 bits
190 SC_MODULE(ula_new)
191 {
192     //i/o
193     sc_in <sc_int<16> > A, B;
194     sc_in <sc_uint<3> > op;
```

```
195     sc_out <sc_int<16> > C;
196 
197     //function to be performed by the module
198     void opUla();
199 
200     SC_CTOR(ula_new)
201     {
202         //sequential method with sensitive list
203         SC_METHOD(opUla);
204         sensitive << A << B << op;
205     }
206 };
207 
208 #endif
```

Listing 4.7: ULA de 16 Bits

```
209 #include "ula.h"
210 
211 void ula_new::opUla(){
212 
213     //funciton variables
214     sc_int<16> aux;
215     int a,b;
216 
217     //select which function is going to be performed
218     switch (op.read()){
219 
220         case 1:
221             //Add
222             aux = A.read() + B.read();
223             break;
224 
225         case 2:
226             //Sub
227             aux = A.read() - B.read();
228             break;
229 
230         case 3:
231             //Multply
232             aux = A.read() * B.read();
```

```
233             break;
234         case 4:
235             //A Bigger then B
236             if (A.read() > B.read()){
237                 aux = 1;
238             }else{
239                 aux = 0;
240             }
241             break;
242 
243         case 5:
244             //AND
245             a = A.read();
246             b = B.read();
247             aux = a & b;
248             break;
249 
250         default:
251             aux = 0;
252             break;
253 
254     }//end switch case
255 
256     //ouput
257     C.write(aux);
258 
259 }//end opUla
```

Listing 4.8: Descrição da Funcionalidade da ULA de 16 Bits

A Figura 15 mostra o resultado da simulação, onde **C** é a saída e **op** é o código da operação, que funciona de acordo com os valores mostrados na Tabela 2. As funções aritméticas selecionadas foram a soma, a função mais comum em ULAs, "maior que", que pode ser usada para passar o valor máximo de uma saída (e.g., *overflow memory*), multiplicação e a operação lógica "E".

Tabela 2: Operação da ULA de 16 Bits

op	**Função**
1	Soma
2	Subtração
3	Multiplicação
4	Maior que
5	Operação "E"

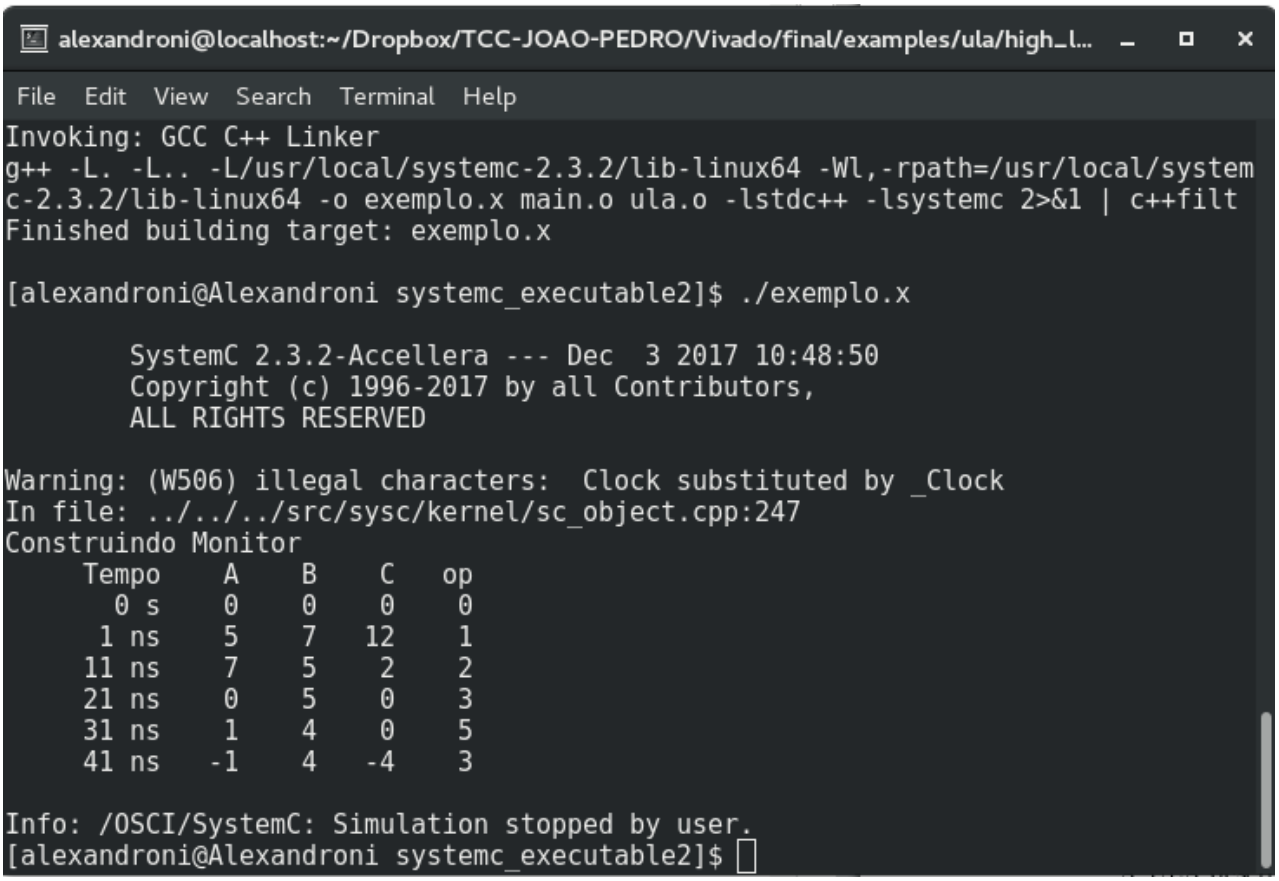

Figura 15: Resultado da Simulação da ULA de 16 bits, incluindo números negativos

5 HLS e Implementação em FPGA

Neste capítulo, são descritos os processos de HLS, simulação pós-síntese e implementação em FPGA dos circuitos descritos no Capítulo 4, bem como são analisados os resultados obtidos.

O processo é composto basicamente por duas partes: geração de um *IP core* como resultado da síntese em alto nível no *Vivado HLS*, e simulação, síntese e implementação em FPGA no *Vivado Design*. Nessa última parte, os *IPs* sintetizados são utilizados em projetos com linguagem VHDL e Verilog. Todos os *IPs* funcionam para ambas as linguagens, dependendo apenas de qual delas foi selecionada no *Vivado Design*. O Apêndice A contém um tutorial descrevendo com detalhes o passo-a-passo de todo o processo para uma porta AND2.

Nas seções seguintes, são mostrados e comparados os resultados da síntese em alto nível e da implementação em RTL de cada circuito descrito em SystemC no capítulo anterior.

5.1 Contador

Com os arquivos mostrados nas listagens 4.1 e 4.2, foi possível criar um projeto no *Vivado HLS*, seguindo os passos descritos no tutorial do Apêndice A. O projeto foi criado dentro da mesma pasta que contém os arquivos *header* e **.cpp** que descrevem o módulo, além dos arquivos de *testbench*.

Após a síntese em alto nível, foram descritos em Verilog um *wrapper* (Listagem 5.1), para receber o *IP core* resultante da síntese no *Vivado HLS*, e um *testbench* (Listagem 5.2) para realizar a sua simulação.

```
260 module top(
261     clk,
262     reset,
263     start,
264     count_out
265     );
266
267     input wire clk;
268     input wire reset;
269     input wire [0 : 0] start;
270     output wire [9 : 0] count_out;
```

```
271
272     simple_counter_0 inst(
273         .clk(clk),
274         .reset(reset),
275         .start(start),
276         .count_out(count_out)
277       );
278
279 endmodule
```

Listing 5.1: *Wrapper* para simulação em Verilog do *IP* contador simples

```
280 module top_tb();
281
282      reg clk;
283      reg reset;
284      reg [0 : 0] start;
285      wire [9 : 0] count_out;
286
287     top dut(.clk (clk),
288             .reset (reset),
289             .start (start),
290             .count_out (count_out));
291
292     initial begin
293         clk = 1'b0;
294         forever #10 clk = ~clk;
295     end
296
297     initial begin
298         reset = 1'b1;
299         #10
300         reset = 1'b0;
301         #10
302         start = 1'b1;
303         #10
304         #200
305         \$stop;
306     end
307
308 endmodule
```

Listing 5.2: *Testbench* para simulação em Verilog do *IP* contador simples

A Figura 16 mostra o resultado da simulação do contador simples no *Vivado Design*, utilizando o *wrapper* e o *testbench*.

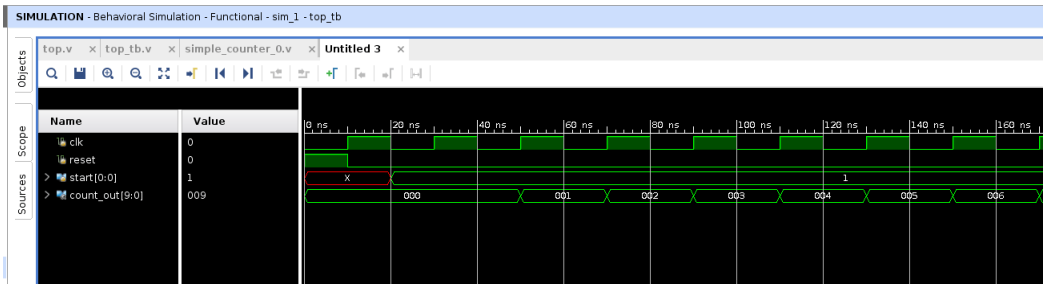

Figura 16: Resultado da Simulação do Contador Simples no Vivado Design

A Figura 17 mostra o resultado da síntese em alto nível do contador, realizada no *Vivado HLS*, e a Figura 18, o resultado da síntese RTL realizada no *Vivado Design* a partir do seu *IP core*.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	-	13	47
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	10	-
Total	0	0	23	47
Available	730	740	269200	129000
Utilization (%)	0	0	~0	~0

Figura 17: Resultado da Síntese no Vivado HLS do Contador Simples

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	10	20800	0.05
FF	22	41600	0.05
IO	13	106	12.26

Figura 18: Resultado da Implementação do Contador na Basys3

5.2 Divisor de Clock

De maneira análoga ao contador simples, o divisor de *clock* foi sintetizado utilizando o *Vivado HLS*. As listagens 5.3 e 5.4 mostram o *wrapper* e o *testbench* utilizados para simular o *IP core* gerado após a síntese. O resultado da simulação final é mostrado na Figura 19.

Listing 5.3: Módulo Divisor de Clock - top (Verilog)

```
1  module top(
2      clk,
3    reset,
4    start,
5    final,
6    count_out);
7
8    input wire clk;
9    input wire reset;
10   input wire [0 : 0] start;
11   input wire [9 : 0] final;
12   output wire [0 : 0] count_out;
13
14   division inst(
15       .clk(clk),
16       .reset(reset),
17       .start(start),
18       .final(final),
19       .count_out(count_out)
20     );
21
22 endmodule
```

Listing 5.4: Testbench Divisor de Clock - Verilog

```
1  module top_tb();
2      reg clk;
3     reg reset;
4     reg [0 : 0] start;
5     reg [9 : 0] final;
6     wire [0 : 0] count_out;
7
8      top dut(.clk(clk),
9        .reset(reset),
10       .start(start),
11       .final(final),
12       .count_out(count_out));
13
14     initial begin
15         clk = 1'b0;
16         forever #10 clk = ~clk;
17     end
18
19
20
21     initial begin
22         reset = 1'b1;
23         #10
24         reset = 1'b0;
25         #10
26         start = 1'b1;
27         final = 10'b10;
28         #600
29         \$stop;
30     end
31
32 endmodule
```

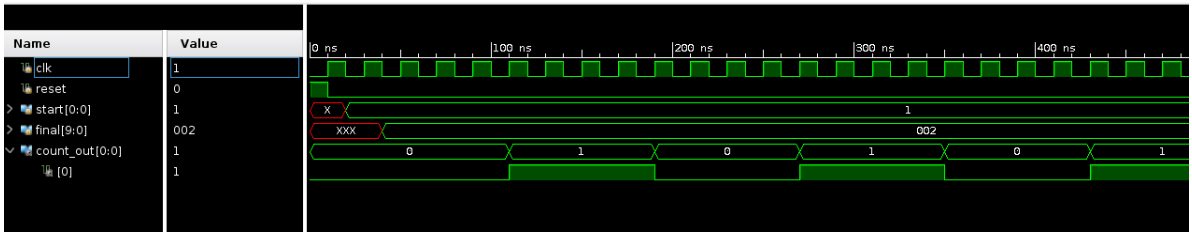

Figura 19: Resultado da Simulação do Divisor de Clock no *Vivado Design*

Os resultados das sínteses podem ser observados nas Figuras 20 e 21, sendo que esta última mostra a estimativa de utilização dos recursos da placa após a implementação do *IP core* em FPGA, realizada no *Vivado Design*.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	-	39	125
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	1	-
Total	0	0	40	125
Available	730	740	269200	129000
Utilization (%)	0	0	~0	~0

Figura 20: Resultado da Síntese no Vivado HLS do Divisor de Clock

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	20	20800	0.10
FF	39	41600	0.09
IO	14	106	13.21
BUFG	1	32	3.13

Figura 21: Resultado da Implementação do Divisor de Clock na Basys3

5.3 ULA de 1 Bit

As listagens 5.5 e 5.6 mostram o *wrapper* e o *testbench* utilizados para simular o *IP core* da ULA de 1 bit, gerado após a síntese em alto nível. Os resultados da simulação estão mostrados na Figura 22.

Listing 5.5: Módulo ULA 1 Bit - Verilog

```
1  module top_ula(
2      A,
3      B,
4      carryIn,
5      op,
6      C,
7      carryOut);
8
9
10     input wire [0 : 0] A;
11     input wire [0 : 0] B;
12     input wire [0 : 0] carryIn;
13     input wire [1 : 0] op;
14     output wire [0 : 0] C;
15     output wire [0 : 0] carryOut;
16
17       ula_new_0 inst (
18         .A(A),
19         .B(B),
20         .carryIn(carryIn),
21         .op(op),
22         .C(C),
23         .carryOut(carryOut)
24       );
25
26 endmodule
```

Listing 5.6: Testbench ULA 1 Bit Verilog

```
1  module tb_ula();
2
3      reg [0 : 0] A;
4      reg [0 : 0] B;
5      reg [0 : 0] carryIn;
```

```
6       reg [1 : 0] op;
7       wire [0 : 0] C;
8       wire [0 : 0] carryOut;
9
10        top_ula dut (
11          .A(A),
12          .B(B),
13          .carryIn(carryIn),
14          .op(op),
15          .C(C),
16          .carryOut(carryOut)
17        );
18
19        initial begin
20                  A = 1'b0;
21                  B = 1'b0;
22                  op = 2'b0;
23                  carryIn = 1'b1;
24                  #10
25
26                  A = 1'b0;
27                  B = 1'b1;
28                  op = 2'b1;
29                  carryIn = 1'b0;
30                  #10
31
32                  A = 1'b1;
33                  B = 1'b0;
34                  op = 2'b10;
35                  #10
36
37                  A = 1'b1;
38                  B = 1'b1;
39                  op = 2'b11;
40                  #10
41
42                  $stop;
43              end
44  endmodule
```

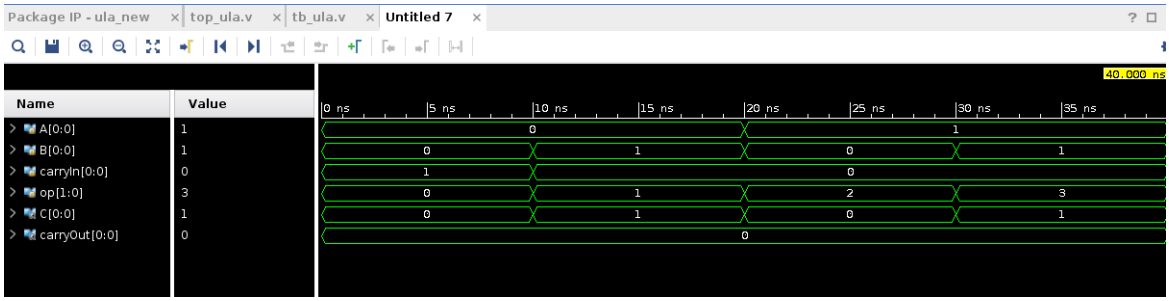

Figura 22: Resultado da Simulação da ULA de 1 Bit no *Vivado Design*

A ULA de 1 bit foi sintetizada no *Vivado HLS*, como mostra a Figura 23, e o *IP core* gerado foi utilizado na implementação no *Vivado Design* (Figura 24).

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	-	0	50
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	0	0	0	50
Available	730	740	269200	129000
Utilization (%)	0	0	0	~0

Figura 23: Resultado da Síntese no Vivado HLS da ULA de 1 Bit

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	2	20800	0.01
IO	6	106	5.66

Figura 24: Resultado da Implementação da ULA de 1 Bit na Basys3

5.4 ULA de 16 bits

Dentre os circuitos escolhidos para demonstrar o processo de síntese, a ULA de 16 bits é o mais complexo, pois sua representação em nível RTL seria composta pelo cascateamento de 16 ULAs de 1 bit. Em SystemC, entretanto, sua descrição ficou simples, bem como a simulação do *IP core* gerado pela HLS. As listagens 5.7 e 5.8 mostram, respectivamente, o *wrapper* e o *testbench* utilizados na sua simulação, cujos resultados podem ser vistos na Figura 25.

Listing 5.7: Módulo ULA 16 Bits - Verilog

```
1  module ula_high(
2       A,
3       B,
4       op,
5       C);
6
7      input wire [15 : 0] A;
8      input wire [15 : 0] B;
9      input wire [2 : 0] op;
10     output wire [15 : 0] C;
11
12   ula_new_0 inst (
13     .A(A),
14     .B(B),
15     .op(op),
16     .C(C)
17   );
18
19 endmodule
```

Listing 5.8: Testbench ULA 16 Bit Verilog

```
1  module ula_high_tb();
2
3      reg [15 : 0] A;
4      reg [15 : 0] B;
5      reg [2 : 0] op;
6      wire [15 : 0] C;
7
8      ula_high dut (
9          .A(A),
```

```
10          .B(B),
11          .op(op),
12          .C(C)
13      );
14
15      initial begin
16                      A = 16'd5;
17                      B = 16'd8;
18                      op = 3'b1;
19                      #10
20
21                      A = 16'd55;
22                      B = 16'd118;
23                       op = 3'b10;
24                      #10
25
26                      A = 16'd55;
27                      B = 16'd0;
28                      op = 3'b11;
29                      #10
30
31                      A = 16'd7;
32                      B = 16'd2;
33                      op = 3'b100;
34                      #10
35
36                      A = 16'd1;
37                      B = 16'd1;
38                      op = 3'b101;
39                      #10
40
41                      $stop;
42    end
43  endmodule
```

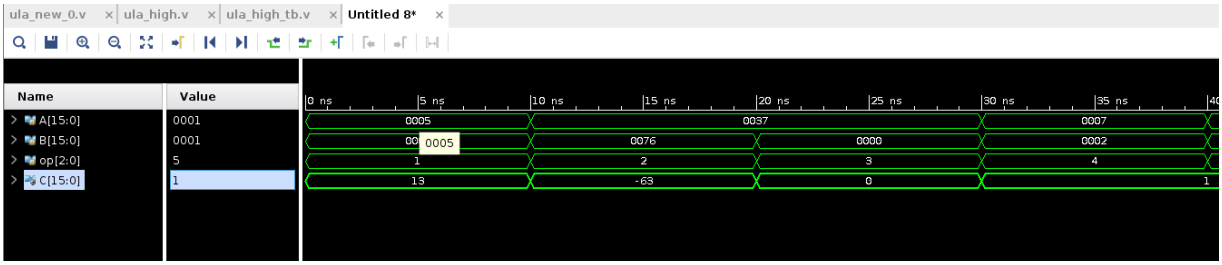

Figura 25: Resultado da Simulação da ULA de 16 bits em Verilog no *Vivado Design*

As estimativas de utilização de recursos da placa após as sínteses em alto nível e RTL são mostradas, respectivamente, nas Figuras 26 e 27.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	1	0	120
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	0	1	0	120
Available	730	740	269200	129000
Utilization (%)	0	~0	0	~0

Figura 26: Resultado da Síntese no Vivado HLS da ULA de 16 Bits

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	72	20800	0.35
DSP	1	90	1.11
IO	51	106	48.11

Figura 27: Resultado da Implementação da ULA de 16 bits na Basys3

5.5 Discussão

Os dados mostrados após as sínteses estão de acordo com a seleção do dispositivo no qual os circuitos serão implementados. No caso do *Vivado HLS*, foi selecionado o FPGA Artix-7. Já no *Vivado Design*, foi possível selecionar diretamente a placa Basys3. Essa diferença pode ser percebida pelo número de recursos disponíveis em ambos os casos, sendo que, para o *Vivado HLS*, havia 740 DSPs, 269.200 flip-flops e 129.000 LUTs, e para o *Vivado Design*, havia 90 DSPs, 41.600 flip-flops e 208.000 LUTs. A utilização de IOs não foi considerada nesta comparação.

É possível observar, para todos os circuitos, que houve uma redução do número de LUTs na implementação em FPGA realizada pelo *Vivado Design*, em relação ao obtido na síntese realizada pelo *Vivado HLS*. Isso era esperado, visto que a primeira envolve processos de otimização.

Vale notar também que foi utilizado um DSP na implementação da ULA de 16 bits em FPGA. Nesse caso, observa-se que o número de flip-flops foi zerado, pois as saídas do DSP já são registradas e, portanto, não houve a necessidade de instanciar flip-flops.

6 Conclusão

Apesar de uma grande vantagem do aumento do nível de abstração no desenvolvimento de circuitos digitais ser a redução do tempo de projeto de circuitos mais complexos, a dificuldade de se utilizar ferramentas de síntese em alto nível faz com que sua utilização seja pouco aproveitada. Além disso, as particularidades de cada ferramenta para tornar a síntese possível reduzem ainda mais esse aproveitamento.

O Vivado HLS demonstrou ser uma boa ferramenta para a implementação de circuitos digitais simples através da síntese em alto nível. Todos os circuitos propostos foram sintetizados e programados na placa Basys3. Porém, em algumas tentativas para sintetizar circuitos um pouco mais complexos, foi possível observar que algumas particularidades da ferramenta, como a necessidade de utilizar bibliotecas próprias do *Vivado HLS* na descrição de circuitos de memória, são pontos negativos. Além disso, a falta de documentação e exemplos para a descrição de circuitos em SystemC tornaram-se uma barreira para sua utilização neste trabalho. Finalmente, outro ponto a ser ressaltado é o fato dessa ferramenta ter como objetivo sintetizar circuitos para serem utilizados em suas próprias famílias de dispositivos, não sendo possível utilizar o código gerado (Verilog ou VHDL) em outros programas como o Soc Encounter, da Cadence, para a fabricação de circuitos integrados, por exemplo.

Recentemente, a Universidade de Brasília adquiriu a licença do Stratus HLS, da Cadence. Com isso, foi possível realizar alguns testes preliminares com essa ferramenta. O Stratus HLS é uma ferramenta otimizada para síntese de circuitos descritos em SystemC, apesar de também sintetizar circuitos descritos em C e C++. Uma vantagem sobre o Vivado HLS é a possibilidade de sintetizar circuitos que cujos códigos resultantes podem ser usados em outras ferramentas, tanto para circuitos integrados quanto para projetos com FPGA. Um ponto negativo é a quantidade reduzida de exemplos e uma documentação mais densa e específica que a do Vivado HLS. Por esse motivo, não foi possível utilizar esta ferramenta para a comparação entre os resultados deste trabalho.

Para a continuidade deste trabalho, sugere-se que sejam feitas as descrições, sínteses e implementações de circuitos mais complexos, compostos por mais de um módulo, como microprocessadores, a partir do processo descrito neste documento e do tutorial produzido, que também pode ser usado como base para a introdução deste tema em disciplinas que abordam tanto o projeto com FPGAs quanto o projeto de circuitos integrados digitais. Outra sugestão é a utilização de ferramentas diferentes que já se encontram disponíveis, como o Stratus HLS e o Catapult, para comparação e escolha do melhor resultado, bem como para a realização de co-simulações.

Referências

ACCELLERA. *SystemC Official Web Site*. 2016. Accellera SystemC. Disponível em: <http://accellera.org/downloads/standards/systemc>. Acesso em: 10.11.2016. Citado 2 vezes nas páginas 30 e 33.

ALBRECHT, C. et al. Sequential logic synthesis with retiming in encounter rtl compiler (rc). *Cadence Design Systems and Focus Semiconductor, Tech. Rep*, Citeseer, 2006. Citado na página 38.

ALTERA. *FPGA-ASIC Design Flow*. 2016. Altera FPGA-ASIC Design Flow. Disponível em: <https://www.altera.com/content/dam/altera-www/global/en_US/pdfs-/literature/an/an311.pdf>. Acesso em: 10.11.2016. Citado 3 vezes nas páginas 15, 35 e 37.

BAILEY, S. Comparison of vhdl, verilog and systemverilog. *Available for download from www. model. com*, 2003. Citado 2 vezes nas páginas 29 e 30.

BESERRA, G. S. Modelagem em nível transacional de sistemas em chip mistos para aplicações de redes de sensores sem fio. *Unviersidade de Brsília, Tese de Doutorado*, 2011. Citado 4 vezes nas páginas 28, 30, 31 e 33.

BLACK, D. C. et al. *SystemC: From the ground up*. [S.l.]: Springer Science & Business Media, 2009. Citado 2 vezes nas páginas 30 e 31.

CADENCE. *Cadence Stratus HLS Datasheet*. 2016. Cadence Stratus HLS. Disponível em: <https://www.cadence.com/content/dam/cadence-www/global/en_US/documents-/tools/digital-design-signoff/stratus-ds.pdf>. Acesso em: 10.11.2016. Citado 3 vezes nas páginas 15, 38 e 39.

CADENCE. *Cdence Encounter Compiler Syhtesis Flow*. 2016. Cadence Encounter Compiler. Disponível em: <http://www.csee.umbc.edu/~tinoosh/cmpe641/tutorials/rc-/rc-flow.pdf>. Acesso em: 10.11.2016. Citado na página 38.

CALAZANS, N. Métodos e ferramentas para o projeto de sistemas digitais. *Escola Regional de Informática*, p. 34–53, 1995. Citado 5 vezes nas páginas 15, 23, 24, 27 e 28.

FRANZON, P.; PERELSTEIN, S.; HURST, A. Tutorial 1-introduction to asic design methodology. 1999. Citado na página 36.

GAJSKI, D. D. et al. *Embedded system design: modeling, synthesis and verification*. [S.l.]: Springer Science & Business Media, 2009. Citado 3 vezes nas páginas 15, 27 e 28.

GHENASSIA, F. et al. *Transaction-level modeling with SystemC*. [S.l.]: Springer, 2005. Citado 3 vezes nas páginas 15, 24 e 32.

HENNESSY, J. L.; PATTERSON, D. A.; LARUS, J. R. *Organização e projeto de computadores: a interface hardware/sofware*. [S.l.]: LTC, 2000. Citado 3 vezes nas páginas 15, 49 e 50.

IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, p. 1–638, Jan 2012. Citado na página 30.

KUNDERT, K.; ZINKE, O. *The designer's guide to Verilog-AMS*. [S.l.]: Springer Science & Business Media, 2006. Citado 2 vezes nas páginas 28 e 41.

LAVAGNO, L.; SCHEFFER, L.; MARTIN, G. *EDA for IC implementation, circuit design, and process technology*. [S.l.]: CRC press, 2006. Citado 3 vezes nas páginas 24, 36 e 37.

MORENO, E. I.; ARES, T. R.; CALAZANS, N. L. Modelagem e descrição de socs em diferentes níveis de abstração. In: SN. *X Workshop Iberchip, Cartagena*. [S.l.], 2004. p. 1–11. Citado 2 vezes nas páginas 27 e 32.

NVIDIA. *NVIDIA P100 GPU Datasheet*. 2016. NVIDIA P100 GPU Datasheet. Disponível em: <http://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-datasheet.pdf>. Acesso em: 10.11.2016. Citado na página 23.

SCHALLER, R. R. Moore's law: past, present and future. *IEEE spectrum*, IEEE, v. 34, n. 6, p. 52–59, 1997. Citado na página 23.

SSC, A. *SystemC Synthesizable Subet 1.3*. 2009. Accellera SystemC Synthesizable Subset 1.3. Disponível em: <http://www.accellera.org/images/downloads/drafts-review-/SystemC_Synthesizable_subset.1.3.pdf>. Acesso em: 12.12.2017. Citado na página 40.

TANENBAUM, A. S. *Organização Estruturada de Computadores*. São Paulo, SP, Brasil: Pearson Pretenice Hall, 2007. Citado na página 23.

VIVADO. *Vivado HLS Official Web site*. 2016. Vivado HLS. Disponível em: <https:-//www.xilinx.com/products/design-tools/vivado/integration/esl-design.html>. Acesso em: 10.11.2016. Citado 3 vezes nas páginas 15, 39 e 40.

Apêndices

APÊNDICE A – Vivado HLS Tutorial - v1.0

A.1 Objetivos

Este tutorial tem como objetivos mostrar a utilização do *Vivado HLS* para síntese de módulos descritos em SystemC, bem como a utilização do *IP core* resultante no *Vivado Design* para programar uma placa Basys3. O fluxo utilizado está mostrado na Figura 28.

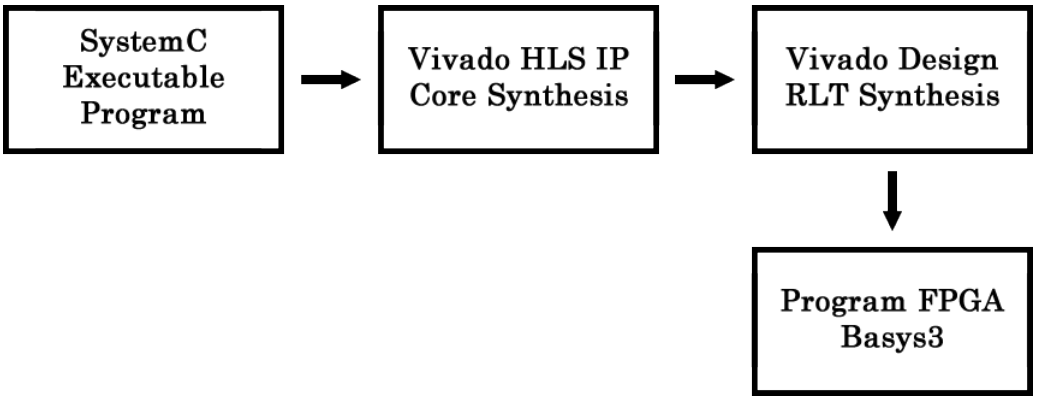

Figura 28: Fluxo de Projeto

A.2 Requisitos

Para a realização deste tutorial, foram utilizados os seguintes recursos:

1. Sistema Operacional CentOS 7
2. Vivado HLS 2017.3
3. Vivado Design 2017.3
4. SystemC 2.3.2 (instalado no sistema)
5. Kit Basys3
6. Licensa de uso: Web Pack
7. Códigos utilizados: disponíveis em https://github.com/Alexandroni/systemC

A.3 AND2

O projeto escolhido para demonstração é um bloco AND2. A Figura 29 mostra o bloco a ser criado.

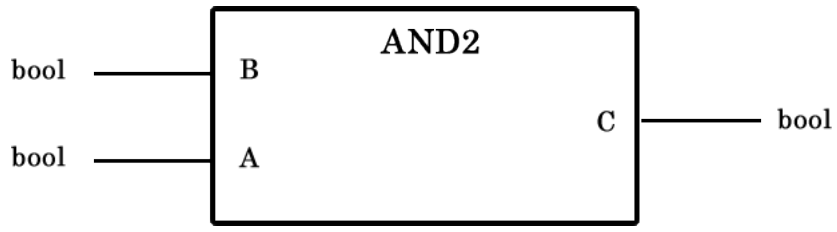

Figura 29: AND2

Seguindo o fluxo de projeto, primeiramente é realizada a descrição do módulo desejado e sua execução utilizando apenas o compilador **gcc**. Os códigos a seguir descrevem as duas partes do bloco AND2, sendo que o arquivo **and2.h** contém a descrição de IO, o construtor e os protótipos de funções, e o arquivo **and2.cpp** contém as funções declaradas no *header*.

Para a execução do arquivo em SystemC, não é necessário dividir um módulo em dois arquivos. No entanto, como o Vivado HLS segue este padrão, decidiu-se adotar o mesmo.

Listing A.1: and2.h – header file

```
1  #ifndef AND2_H
2  #define AND2_H
3
4  #include <systemc.h>
5
6  SC_MODULE(and2)
7  {
8      sc_in<sc_uint<1> > A, B;
9      sc_out<sc_uint<1> > F;
10
11     void func();
12
13     SC_CTOR(and2)
14     {
15         SC_METHOD(func);
16         sensitive << A << B;
17     }
18
19 };
```

```
20 #endif
```

Listing A.2: and2.cpp – function file

```
1 #include "and2.h"
2
3 void and2::func(){
4
5     bool f1;
6     f1=(A.read() && B.read());
7     F.write(f1);
8
9 }
```

Os códigos acima descrevem um bloco operacional de uma função AND para duas entradas de 1 bit. Para executar uma simulação funcional, foram utilizados os seguintes códigos: **main.cpp**, **monitor.h**, **estimulos.h**, e um *makefile* contendo os comandos para compilar os arquivos. O arquivo **estimulos.h** gera diferentes entradas a cada pulso de *clock*, declarado no arquivo **main.cpp**. O arquivo **monitor.h** recebe todas as saídas do bloco testado e as imprime na tela. O arquivo **main.cpp** junta todos os arquivos para criar uma função executável em C++.

Listing A.3: estimulos.h – Gera estímulos

```
1 #include "systemc.h"
2
3 SC_MODULE(Estimulos)
4 {
5     sc_out <sc_uint<1> > A, B;
6     sc_in <bool> Clk;
7
8     void GeraEstimulos()
9     {
10        A.write(0);
11        B.write(0);
12        wait();
13
14        A.write(0);
15        B.write(1);
16        wait();
17
```

```
18          A.write(1);
19          B.write(0);
20          wait();
21
22          A.write(1);
23          B.write(1);
24          wait();
25
26          sc_stop();
27      }
28
29      SC_CTOR(Estimulos)
30      {
31          SC_THREAD(GeraEstimulos);
32          sensitive << Clk.pos();
33      }
34  };
```

Listing A.4: monitor.h – Recebe as saídas do bloco testado

```
1   #include "systemc.h"
2   #include <iostream>
3   #include <iomanip>
4
5   using namespace std;
6
7   SC_MODULE(Monitor)
8   {
9       sc_in <sc_uint<1> > A, B, F;
10      sc_in <bool> Clk;
11
12      void monitor()
13      {
14          cout << setw(10) << "Time";
15          cout << setw(2) << "A";
16          cout << setw(2) << "B";
17          cout << setw(2) << "F" << endl;
18
19          while(true)
20          {
21              cout << setw(10) << sc_time_stamp();
```

```
22              cout << setw(2) << A.read();
23              cout << setw(2) << B.read();
24              cout << setw(2) << F.read() << endl;
25              wait();
26          }
27      }
28
29      SC_CTOR(Monitor)
30      {
31          cout << "Building.. " << name() << endl;
32          SC_THREAD(monitor);
33          sensitive << Clk.pos();
34      }
35  };
```

Listing A.5: main.cpp – Cria uma função executável para o testbench

```
1  #include "and2.h"
2  #include "monitor.h"
3  #include "estimulos.h"
4
5  int sc_main (int argc , char *argv[])
6  {
7
8
9      //Test ports
10     sc_signal <sc_uint<1> > sinalA, sinalB, sinalF;
11     sc_clock clock(" Clock", 10, SC_NS,0.5, 1, SC_NS);
12
13     Estimulos est("Stimuli");
14     and2 m("And2");
15     Monitor mon("Monitor");
16
17     est.A(sinalA);
18     est.B(sinalB);
19     est.Clk(clock);
20
21     m.A(sinalA);
22     m.B(sinalB);
23     m.F(sinalF);
24
```

```
25      mon.A(sinalA);
26      mon.B(sinalB);
27      mon.F(sinalF);
28      mon.Clk(clock);
29
30      sc_start();
31      return 0;
32  };
```

A Figura 30 mostra o resultado da compilação e execução dos 5 códigos mostrados acima. Uma vez validado o bloco, passa-se para a próxima etapa no *Vivado HLS*.

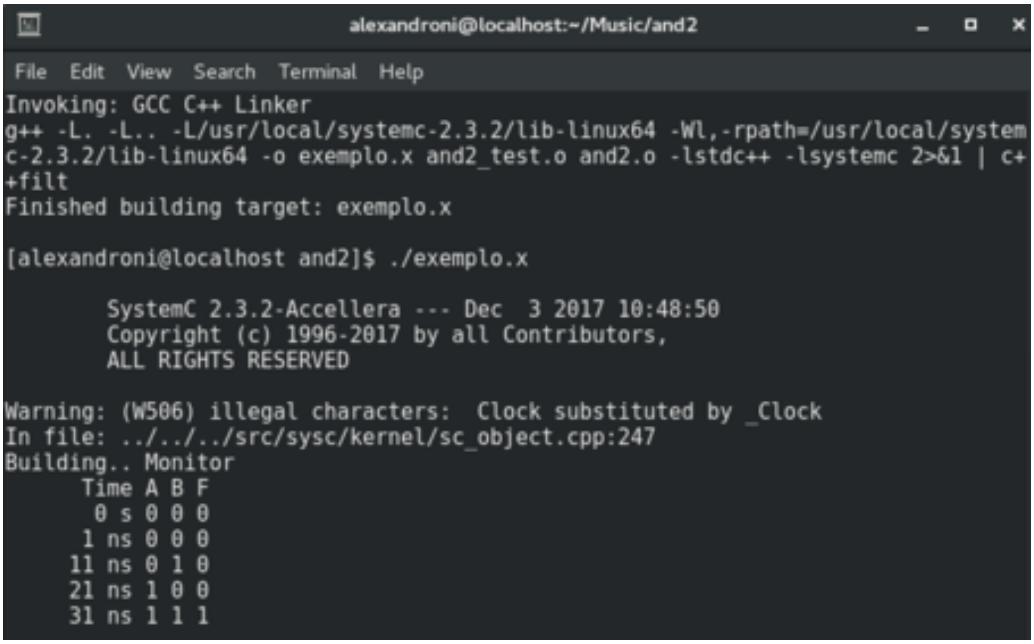

Figura 30: AND2 Testbench

A.4 Vivado HLS

Abrindo-se o *Vivado HLS*, cria-se um novo projeto dentro do mesmo diretório que contém os arquivos descritos acima. **O vivado indexa todos os arquivos *header* dentro de um diretório para que se possa usar a diretiva *include* sem a necessidade de se colocar o caminho inteiro do arquivo. Porém, caso o arquivo *header* não esteja na pasta onde foi criado o projeto do *Vivado HLS*, o programa pode não conseguir fazer a indexação corretamente.**

A Figura 31 mostra o passo de selecionar os arquivos já existentes no diretório. Além disso, é necessário designar uma *top function* para que seja sintetizável. O *Vivado HLS* utiliza essa função como parte principal do projeto. Por padrão, o *Vivado HLS* tenta

achar uma declaração de função em C ou C++. No entanto, em se tratando de SystemC, é necessário colocar o nome do módulo desejado (**SC_MODULE**), e não da função. Nesse caso, o nome é "and", conforme foi definido no primeiro código.

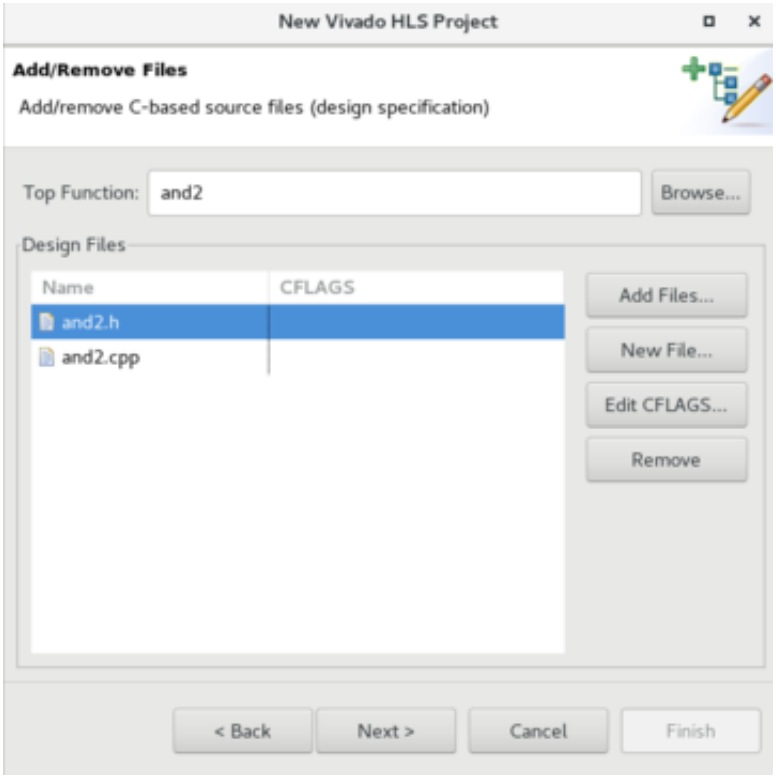

Figura 31: Selecionando arquivos do módulo construído

O próximo passo é semelhante ao anterior, porém dessa vez seleciona-se os arquivos de teste: **main.cpp**, **estimulos.h** e **monitor.h**. O último passo para finalizar a criação do projeto é selecionar o FPGA alvo do projeto. Um *IP core* sintetizado para um FPGA Spartan-3 pode não funcionar em um FPGA Artix-7. Neste caso, como a placa desejada é a Basys3, o FPGA selecionado foi o Artix-7.

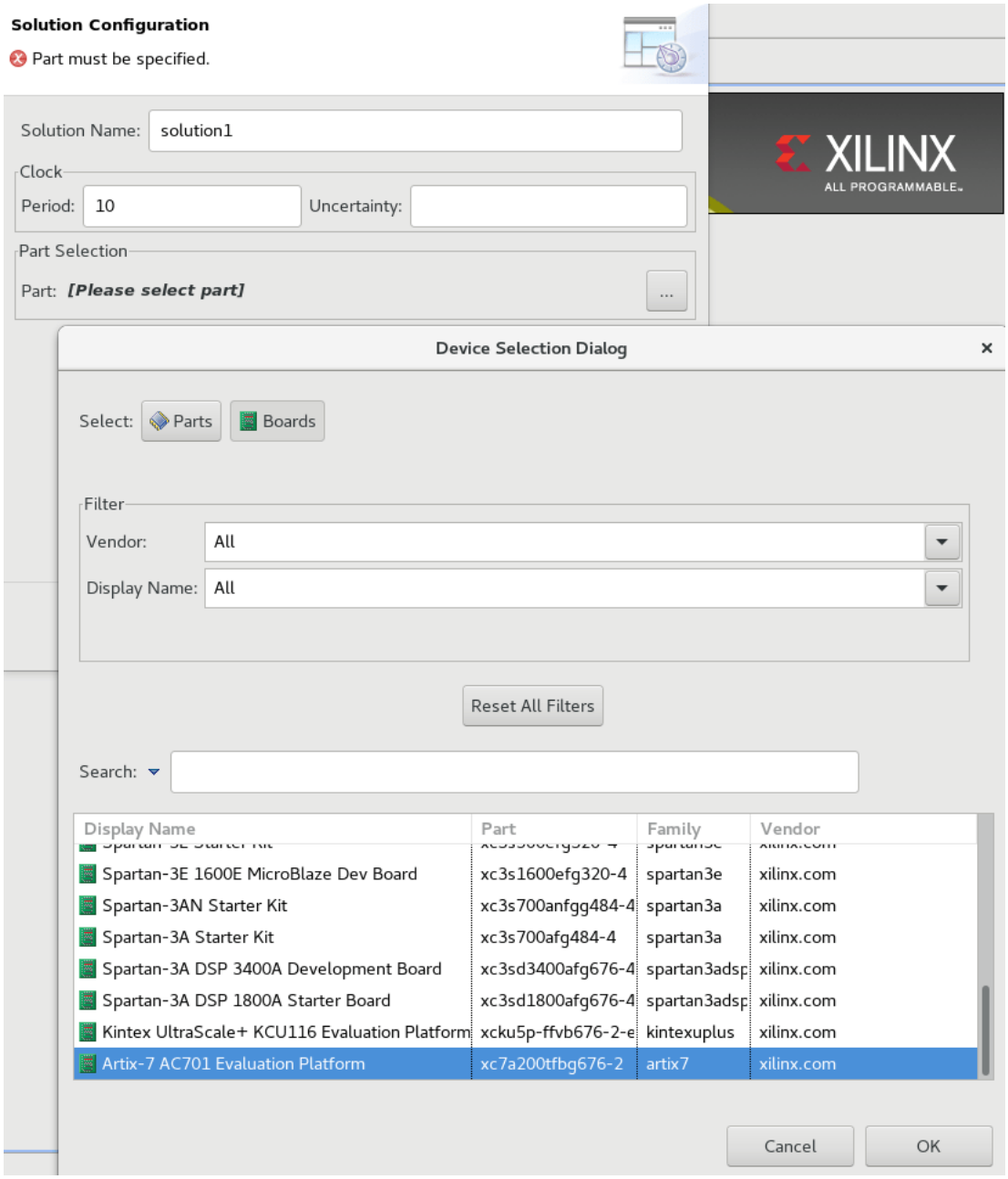

Figura 32: Selecionando Placa FPGA

Uma vez finalizado o projeto, é possível visualizar o arquivo *header* do seu projeto dentro do diretório **includes > /home/user/...** no primeiro tópico da barra lateral esquerda. Caso esteja tudo correto, o ícone de síntese (semelhante a um triângulo verde) do *Vivado HLS* vai ser habilitado no menu principal, tornando possível sintetizar o módulo (Figura 33).

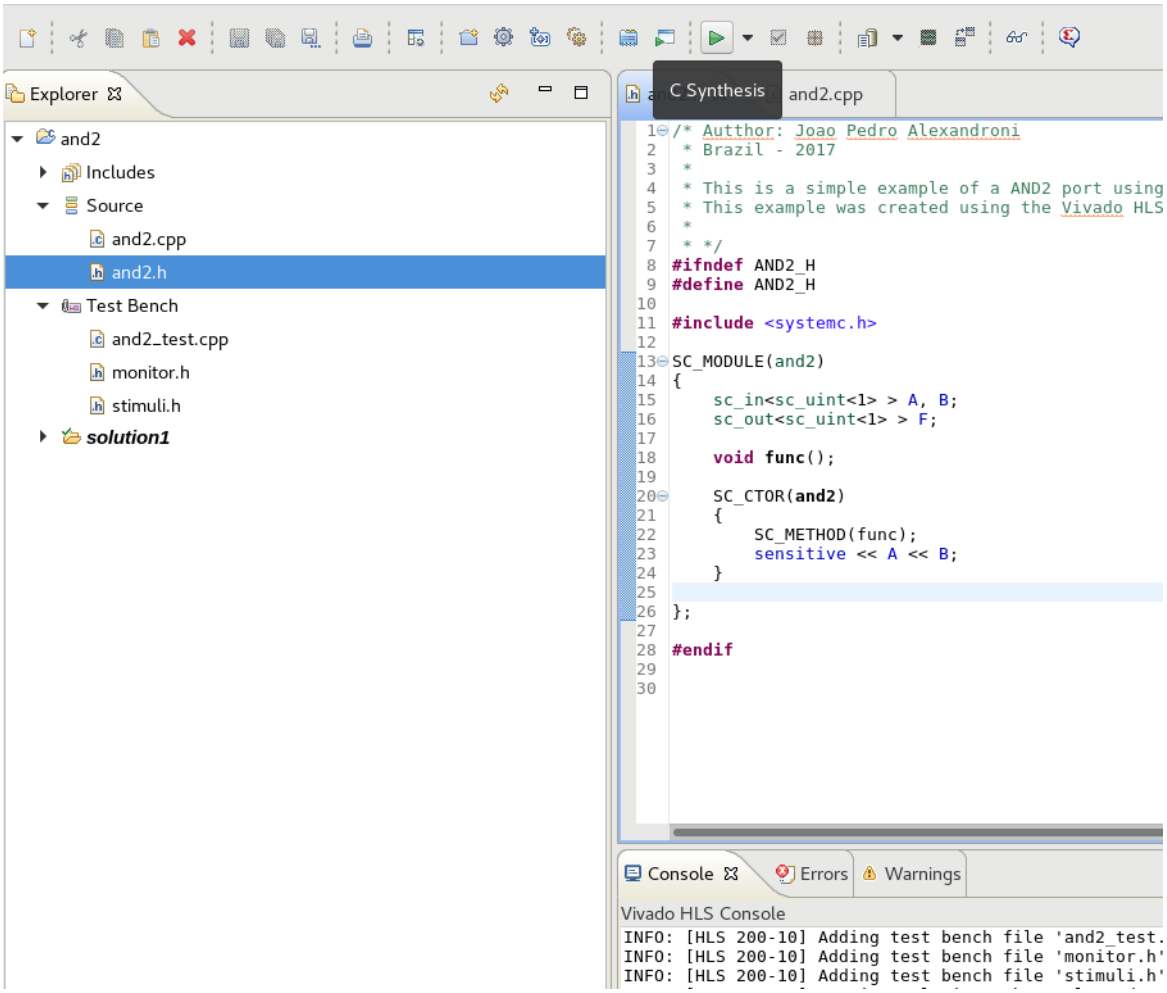

Figura 33: Iniciando processo de síntese do bloco AND2

Após realizada a síntese, o *Vivado HLS* mostrará uma página de *report* contendo os dados da síntese, dentre os quais estão as entradas e saídas utilizadas, o tempo de execução e a estimativa de uso dos recursos da placa selecionada (e.g., número de flip-flops e LUTs). Após a síntese, é possível executar a simulação funcional. O *Vivado HLS* executará sem erros a simulação da mesma forma apresentada anteriormente. No entanto, para executar a co-simulação com o bloco RTL sintetizado, é necessário adicionar algumas linhas de código na função *main* do arquivo de teste, mostradas na listagem a seguir.

Listing A.6: Mudanças na função Main

```
1 #include "and2.h"
2 #ifdef __RTL_SIMULATION__
3 #include "and2_rtl_wrapper.h"
4 #define and2 and2_rtl_wrapper
5 #else
6 #endif
7 #includes...
```

```
8
9  int sc_main (int argc , char *argv[])
10 {
11     //this is necessary to be able to run simulation and ↩
          cosimulation
12     sc_report_handler::set_actions("/IEEE_Std_1666/deprecated", ↩
          SC_DO_NOTHING);
13     sc_report_handler::set_actions( SC_ID_LOGIC_X_TO_BOOL_, ↩
          SC_LOG);
14     sc_report_handler::set_actions( ↩
          SC_ID_VECTOR_CONTAINS_LOGIC_VALUE_, SC_LOG);
15     sc_report_handler::set_actions( SC_ID_OBJECT_EXISTS_, SC_LOG↩
          );
```

Essas linhas são instruções encontradas no *User Guide* UG902 do *Vivado HLS*. Além disso, para a co-simulação funcionar, utiliza-se o modelo de *testbench* descrito nos exemplos em SystemC. Não é necessário realizar a co-simulação para exportar o *IP core* sintetizado.

Para exportar um *IP core*, basta clicar no botão **EXPORT RTL**, como mostra a Figura 34. O IP exportado possui suporte tanto em Verilog quanto em VHDL.

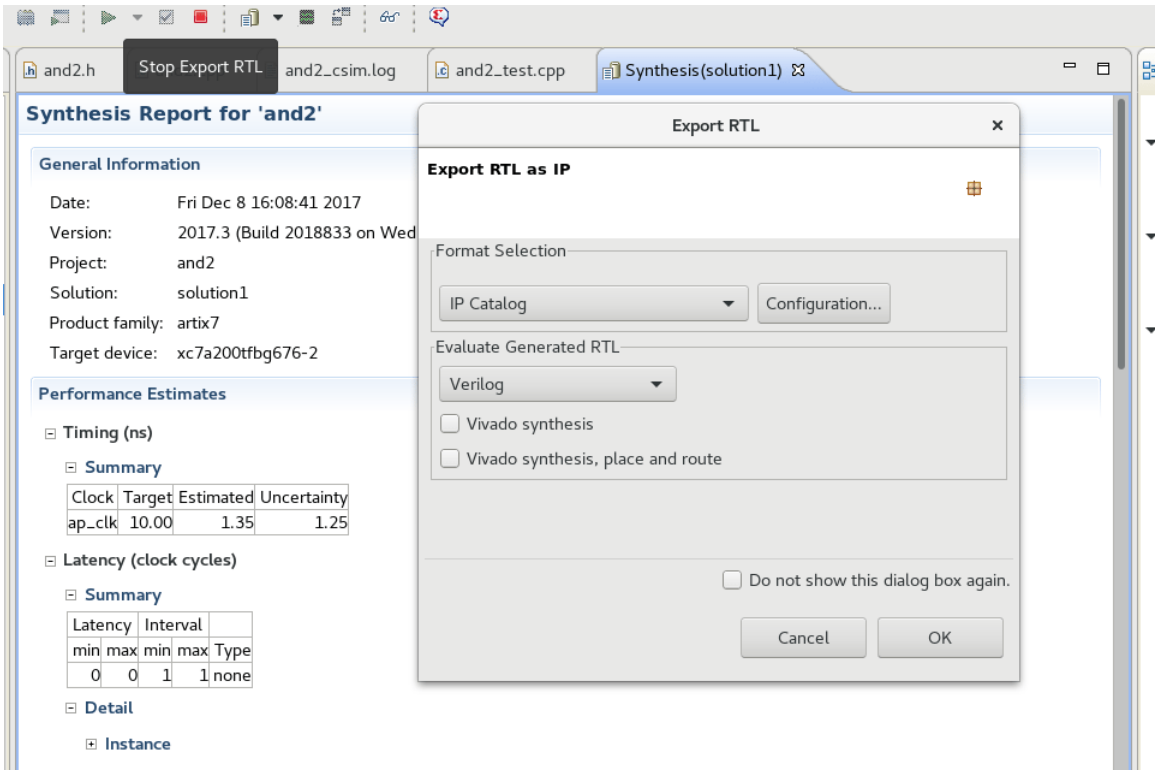

Figura 34: Exportando RTL IP

Uma vez finalizada a exportação do *IP core*, passa-se para o programa *Vivado Design*.

A.5 Vivado Design

Nesta etapa, o objetivo é criar um projeto em VHDL ou Verilog que utilize o *IP core* gerado no *Vivado HLS*, realizar a síntese e programar o FPGA.

Primeiramente, cria-se um novo projeto com a linguagem VHDL ou Verilog, direcionado para a placa Basys3.

***Nota: Caso a Basys3 não esteja instalada, é possível seguir o tutorial de instalação da Digilent: https:// reference.digilentinc.com/reference/software/Vivado/board-files?redirect=1**.

Dentro do projeto criado no *Vivado Design*, é necessário incluir o *IP core* gerado no *Vivado HLS* no **IP-Catalog**. Para isso, basta selecionar a opção **Arquivo > Abrir arquivo IPXACT**. O arquivo desejado está dentro da pasta do projeto sintetizado no *Vivado HLS* (**projeto hls > solution > impl > ip**). O arquivo desejado é chamado de **component.xml**. Após clicar em adicionar, é possível visualizar a seguinte página mostrada na Figura 35.

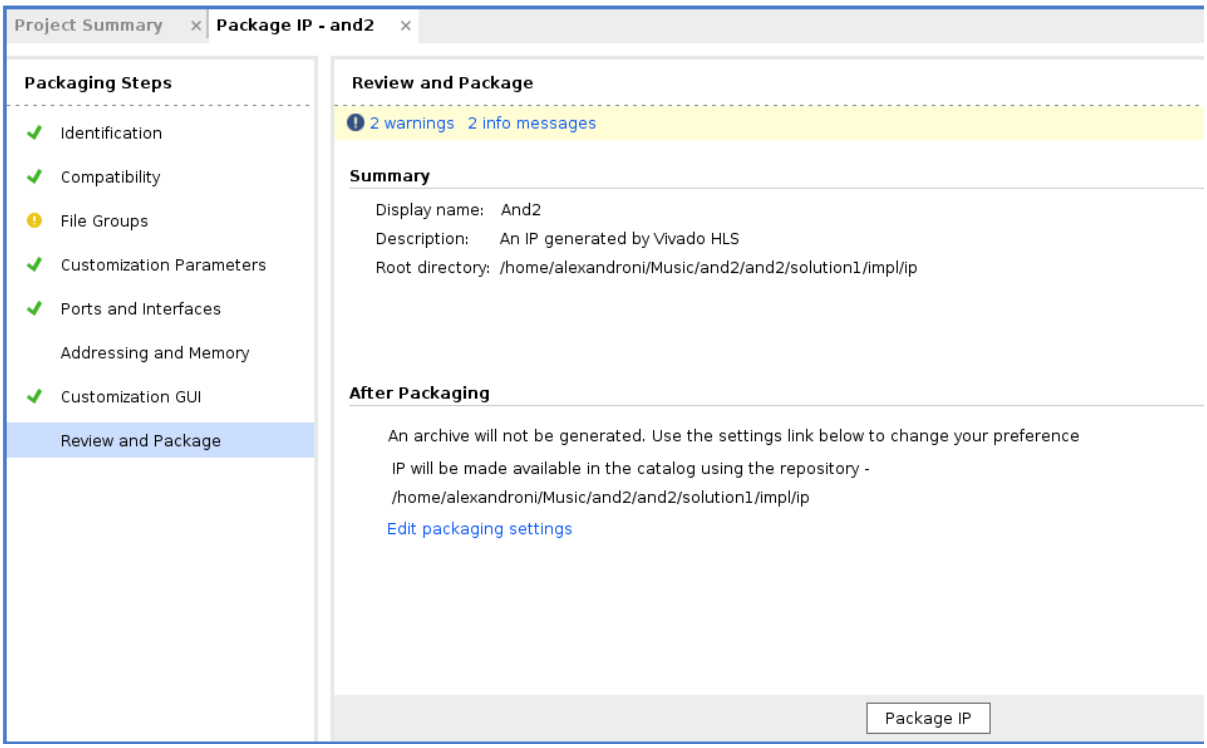

Figura 35: Importando o IP gerado

Clicando no botão **Package IP**, o *IP core* será adicionado ao **IP Catalog** e pode ser utilizado como qualquer outro IP. Para este projeto, criou-se apenas um *wrapper* e um *testbench*, mostrados a seguir, para simular o módulo. O resultado da simulação está mostrado na Figura 36.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity top_and is
5      Port ( A : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
6             B : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
7             F : OUT STD_LOGIC_VECTOR(0 DOWNTO 0));
8  end top_and;
9
10 architecture Behavioral of top_and is
11
12     component and2_0 IS
13       PORT (
14         A : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
15         B : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
16         F : OUT STD_LOGIC_VECTOR(0 DOWNTO 0)
17       );
18     end component;
19
20 begin
21
22     AND_NEW: and2_0 port map
23         (A => A,
24          B => B,
25          F => F
26          );
27
28 end Behavioral;
```

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  entity porta_and_tb is
4  --  Port ( );
5  end porta_and_tb;
6
```

```
7  architecture Behavioral of porta_and_tb is
8
9      component top_and
10     PORT (A : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
11           B : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
12           F : OUT STD_LOGIC_VECTOR(0 DOWNTO 0));
13     end component;
14
15     signal e: STD_LOGIC_VECTOR(0 DOWNTO 0);
16     signal f: STD_LOGIC_VECTOR(0 DOWNTO 0);
17     signal resposta: STD_LOGIC_VECTOR(0 DOWNTO 0);
18
19
20 begin
21
22     dut :  top_and PORT MAP(
23         A => e,
24         B => f,
25         F => resposta);
26
27     simulation : process
28     begin
29
30         e <= "0";
31         f <= "0";
32         wait for 10us;
33
34         e <= "0";
35         f <= "1";
36         wait for 10us;
37
38         e <= "1";
39         f <= "0";
40         wait for 10us;
41
42         e <= "1";
43         f <= "1";
44         wait for 10us;
45
46
47     end process simulation;
48
```

```
49 end Behavioral;
```

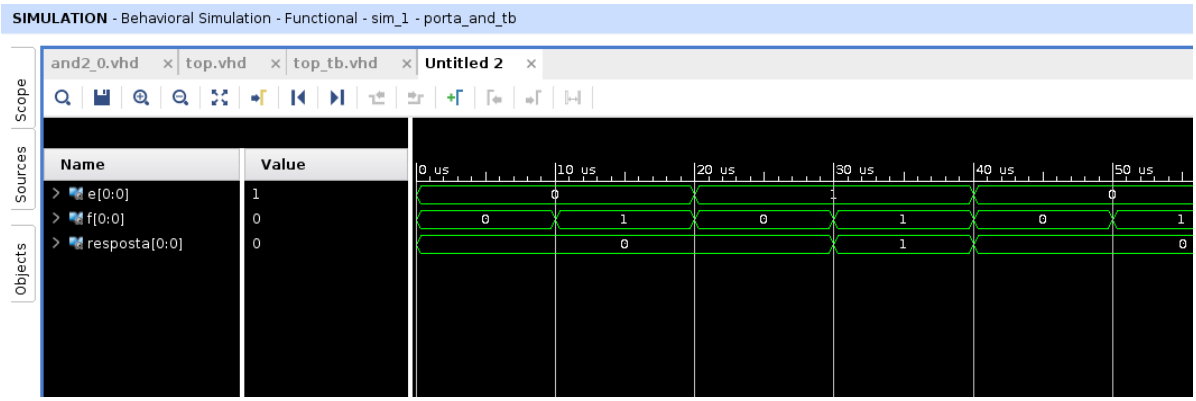

Figura 36: Resultado da simulação VHDL

Os últimos passos são criar o arquivo *bitstream* e, finalmente, programar o FPGA.