UNIVERSIDADE FEDERAL DO TOCANTINS
CÂMPUS UNIVERSITÁRIO DE PALMAS
CURSO DE CIÊNCIA DA COMPUTAÇÃO

SISTEMATIZAÇÃO DAS METODOLOGIAS ESL COM ENFOQUE EM SYSTEMC

FELIPE SOUSA BARBOSA

PALMAS (TO)
2018

FELIPE SOUSA BARBOSA

SISTEMATIZAÇÃO DAS METODOLOGIAS ESL COM ENFOQUE EM SYSTEMC

Trabalho de Conclusão de Curso II apresentado à Universidade Federal do Tocantins para obtenção do título de Bacharel em Ciência da Computação, sob a orientação do(a) Prof.(a) Me. Tiago da Silva Almeida.

Orientador: Me. Tiago da Silva Almeida

PALMAS (TO)

2018

FELIPE SOUSA BARBOSA

SISTEMATIZAÇÃO DAS METODOLOGIAS ESL COM ENFOQUE EM SYSTEMC

Trabalho de Conclusão de Curso II apresentado à UFT – Universidade Federal do Tocantins – Câmpus Universitário de Palmas, Curso de Ciência da Computação foi avaliado para a obtenção do título de Bacharel e aprovada em sua forma final pelo Orientador e pela Banca Examinadora.

Data de aprovação: 7 / 12 / 2018

Banca Examinadora:

Prof. Dr. Alexandre Tadeu Rossini da Silva

Prof. Dr. Rafael Lima de Carvalho

Prof. Me. Tiago da Silva Almeida

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da Universidade Federal do Tocantins

B238s Barbosa, Felipe Sousa.

Sistematização das metodologias ESL com enfoque em SystemC. / Felipe Sousa Barbosa. – Palmas, TO, 2018.

52 f.

Monografia Graduação - Universidade Federal do Tocantins – Câmpus Universitário de Palmas - Curso de Ciências da Computação, 2018.

Orientador: Tiago da Silva Almeida

1. Projeto eletrônico em nível de sistema. 2. Linguagem de descrição de hardware. 3. Linguagem de descrição de arquitetura. 4. Modelagem em nível de transação. I. Título

CDD 004

TODOS OS DIREITOS RESERVADOS – A reprodução total ou parcial, de qualquer forma ou por qualquer meio deste documento é autorizado desde que citada a fonte. A violação dos direitos do autor (Lei nº 9.610/98) é crime estabelecido pelo artigo 184 do Código Penal.

Elaborado pelo sistema de geração automática de ficha catalográfica da UFT com os dados fornecidos pelo(a) autor(a).

RESUMO

O presente trabalho visa servir como um material de referência sintetizado sobre as metodologias em Nível de Sistema Eletrônico (*Electronic System-Level* ou ESL) para projetos de sistemas digitais, tendo como enfoque o uso da da biblioteca SystemC do C++. Já existem inúmeras metodologias e ferramentas presentes na literatura que propõem soluções de projeto ESL. Essa grande quantidade de material torna difícil a decisão de quais metodologias e ferramentas utilizar. O SystemC, por sua vez, é uma biblioteca para projetos de hardware escrita na linguagem C++ e padronizada pelo *IEEE* que permite inúmeros níveis de abstração de projeto por integrar funcionalidades de linguagens de alto nível e de linguagens de descrição de hardware tradicionais, tais como VHDL e Verilog. Por tal, ela é uma ferramenta adequada para suportar projetos ESL. Também foi integrado ao material de referência um estudo de TLM (*Transaction-level Modeling*), visando discutir e apresentar uma solução para integração de componentes em diferentes níveis de abstração em um projeto, permitindo assim a valoração mais cedo do projeto.

Palavra-chave: Projeto eletrônico em nível de sistema. Linguagem de descrição de hardware. Linguagem de descrição de arquitetura. Modelagem em nível de transação.

ABSTRACT

The present work aim to be a synthesized reference material about Electronic System-Level (ESL) methodologies for digital system projects, focus on the use of the C++ library SystemC. There are many methodologies and tools presents in literature that propose solutions for ESL projects. This large amount of material difficulties the decision of which methodologies and tools to use. The SystemC in turn is a IEEE standardized C++ library for hardware projects that allows many design abstraction levels by integrating high-level language and traditional hardware description language functionalities, such as VHDL and Verilog. It has also been integrated in the reference material a study about TLM (Transaction-level Modeling), aiming discuss and present a solution for integration of components in differents level abstractions in a project, thus allowing an early evaluation of the project.

Keywords: Electronic system level design. Hardware description language. Architecture description language. Transaction-level modeling.

LISTA DE SIGLAS

DSP	Digital Signal Processor, p. 27
ESL	Electronic System Level, p. 13
HDL	hardware Description Language, p. 14
IEEE	Institute of Electrical and Electronics Engineers, p. 5
IP	Intellectual Property, p. 30
ISS	Instruction Set Simulation, p. 24
PE	Processing Element, p. 27
PRD	Product Requirements Document, p. 29
RTL	Register-Transfer Level, p. 14
SAM	System Architecture Model, p. 29
VHDL	Very high speed integrated circuit hardware Description Language, p. 14
VLSI	Very-Large-Scale Integration, p. 17

LISTA DE FIGURAS

Figura 1 – Metodologia antiga de projeto a nível de sistema 15

Figura 2 – Metodologia de projeto a nível de sistema com SystemC 15

Figura 3 – Diagrama Y 19

Figura 4 – Representação estrutural de um somador de 1 bit 20

Figura 5 – Disposição física dos componentes de uma implementação para um somador de um bit 20

Figura 6 – Diagrama de telhado duplo 21

Figura 7 – Uma versão possível do diagrama de telhado duplo 23

Figura 8 – Gráfico de modelagem de sistemas 24

Figura 9 – Metodologia TLM 30

Figura 10 – Processo de simulação SystemC 33

Figura 11 – Exemplo de uso do tipo `sc_bv<w>` 36

Figura 12 – Estrutura básica da função `sc_main()` 38

Figura 13 – Estrutura básica de um módulo SystemC sem o uso da macro `SC_MODULE` 39

Figura 14 – Estrutura básica de um módulo SystemC com o uso da macro `SC_MODULE` 40

Figura 15 – Módulo com construtor e *thread* 41

Figura 16 – Implementação do módulo `module_name` 42

Figura 17 – Função principal com instanciação do módulo `module_name` 42

Figura 18 – Conjuntos de processos 45

Figura 19 – Sintaxe de `sc_event` e `sc_event::notify` 46

Figura 20 – Agendamento de notificação do mesmo evento para tempos diferentes: apenas o segundo agendamento prevalece 47

Figura 21 – Principais formas de uso da função `wait()` para observar eventos . 47

Figura 22 – Principais formas de uso da função `next_trigger()` para observar eventos 49

Figura 23 – Exemplo de sensitividade estática 50

LISTA DE TABELAS

Tabela 1 – Tabela verdade de um somador completo de 1 bit 19

Tabela 2 – Características dos diferentes modelos abstratos 27

Tabela 3 – Principais funções de manipulação para tipos de dados SystemC . . . 36

Tabela 4 – Principais formas de representação de tipos SystemC em forma de string 38

Tabela 5 – Resoluções de tempo do SystemC 43

SUMÁRIO

1 INTRODUÇÃO **13**

2 METODOLOGIAS DE PROJETO **17**

2.1 Três escolas de abordagem **17**

2.2 Modelos de representação e Diagrama Y **18**

2.3 Metodologia de telhado duplo e a separação entre hardware e software **20**

2.4 Duas metodologias essenciais e um problema não abordado . . . **23**

3 MODELAGEM EM NíVEL DE TRANSAÇÃO (TLM) **24**

4 NíVEL DE SISTEMA ELETRÔNICO E MODELAGEM EM NíVEL DE TRANSAÇÃO **28**

4.1 Necessidades da ESL e da TLM **28**

4.1.1 Metodologia TLM 29

5 SYSTEMC: INTEGRANDO O PROJETO DE HARDWARE A UMA LINGUAGEM DE ALTO NíVEL **32**

5.1 Visão geral sobre o SystemC **32**

5.2 O *Kernel* SystemC **33**

5.3 Tipos de dados SystemC **34**

5.3.1 Tipos lógicos 35

5.3.2 Tipos inteiros 36

5.3.3 Representações literais no SystmC 37

5.4 A classe módulo do SystemC e processos registrados no construtor 38

5.4.1 `SC_MODULE` : a entidade básica de um projeto SystemC 39

5.5 A passagem de tempo dentro do kernel de simulação **43**

5.6 Concorrência 44
5.6.1 Disparando eventos 46
5.6.2 Capturando eventos de processos *thread* 47
5.6.3 Processos de simulação método do SystemC 48
5.6.4 Capturando eventos de processos método 48
5.6.4.1 Sensitividade estática para Processos 49
6 CONSIDERAÇÕES FINAIS 51
REFERÊNCIAS 52

1 INTRODUÇÃO

A miniaturização e a capacidade de encapsular cada vez mais componentes eletrônicos, principalmente transistores, em um mesmo circuito integrado, vem sendo ao longo das últimas décadas um dos principais fatores que alavancou o poder computacional dos microprocessadores atuais. Tais avanços são dados de forma exponencial, dobrando a capacidade de processamento/armazenamento dos computadores sem grandes alterações de custo, em média a cada dois anos. Tal comportamento foi previsto pelo co-fundador da *intel*, Gordon E. Moore, em 1965 (MOORE, 2006), e ficou conhecida como Lei de Moore.

Após mais de 50 anos em que a Lei de Moore vem se concretizando, chega-se a um ponto em que ela corre o risco de não se cumprir por muito mais tempo, devido aos limites físicos da tecnologia baseada em silício. Como demonstram Chien e Karamcheti (2013), tem-se indicativos do fim da Lei de Moore, como, por exemplo, a perda do tempo de vida de escrita e leitura de memórias *flash*, para que houvesse ganhos de densidade. Porém, a Lei de Moore, devido ao seu tempo de vigência, ainda é um forte definidor da janela de tempo para novos lançamentos e melhorias no mercado.

Além dos limites previstos para as tecnologias de silício, dois problemas surgem devido aos avanços tecnológicos na área de sistemas eletrônicos: o aumento da complexidade desses sistemas e; o aumento da quantidade de aplicações de propósito específico, ou seja, que necessitam de hardware dedicado ou até mesmo de um projeto que desenvolva hardware e software em conjunto (*hardware/software codesign*). Toda essa gama de projetos deve ser desenvolvida visando desempenho, custo e potência, e diminuindo ao máximo o tempo de lançamento do produto final no mercado.

Por tal, fez-se evidente para grande parte do mercado de semicondutores a necessidade do desenvolvimento de novas metodologias de projeto, onde o alto nível de abstração é prezado (GERSTLAUER et al., 2009). Aumentar o nível de abstração até o chamado nível de sistema (*Electronic System-Level* - *ESL*) tornou-se o foco e inúmeras abordagens e ferramentas de síntese surgiram para este fim.

O processo de produção de sistemas eletrônicos, assim como a produção de software para diferentes domínios, pode ser visto por inúmeros níveis de abstração. Esses níveis de abstração definem o nível de especificação do sistema dentro do projeto. No projeto de sistemas eletrônicos digitais, uma descrição possível, tendo em vista apenas o projeto de hardware para sistemas digitais, divide os níveis de abstração em quatro (BLACK et al., 2009). A seguir esses níveis são listados do mais alto nível de abstração ao mais baixo.

- **Arquitetural**: São componentes de alto nível, como processadores de propósito geral, hardware dedicado, memórias e etc., não possuindo informações específicas do seu funcionamento interno.

- **Comportamental**: Apresenta uma descrição de alto nível das operações executadas nos componentes da arquitetura, mas geralmente sem qualquer definição de tempo/ciclo dos componentes.

- **RTL** ou **nível de transferência de registradores**: Implica que o nível de abstração do sistema é visto como um fluxo de dados entre os registradores que os armazenam e as operações lógicas efetuadas sobre eles, podendo apresentar temporização por ciclo de clock.

- **Portas lógicas**: Este nível está relacionado a tecnologia de construção das portas lógicas de um circuito. Mais precisamente da tecnologia usada nos transistores, o que é determinante para o consumo energético, área ocupada e eficiência.

Uma metodologia ESL consiste em construir uma descrição do sistema no nível mais alto de abstração e iterativamente ir refinando tal descrição até que se chegue ao produto final do projeto. Durante esse processo de refinamento do sistema é comum certas funcionalidades serem implementadas em software, o que diminui o custo de produção. Porém, a decisão de quais funcionalidade implementar em software também tem impacto sobre o desempenho e potência, o que a torna uma decisão difícil. Sendo assim, qualquer metodologia dita ESL deve levar em consideração o projeto de software.

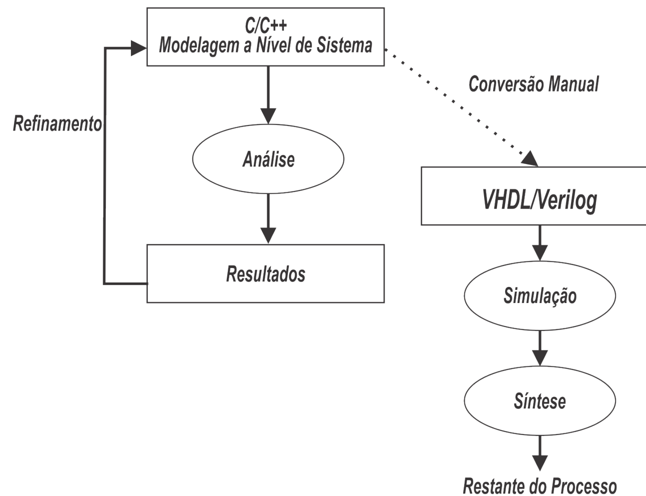
De fato, projetos de sistemas eletrônicos modernos quase sempre apresentam implementações em software. Classicamente, o desenvolvimento do software acontecia apenas nas fases finais do projeto de hardware, o que geralmente estendia o projeto como um todo. Metodologias ESL também permitem o projeto de software mais cedo, o que além de gerar ganho de tempo no projeto, implica que recursos, tanto de hardware quanto de software, possam ser otimizados para seu funcionamento conjunto (BLACK et al., 2009).

Atrelado às metodologias ESL estão as ferramentas de síntese que possibilitam a aplicação destas metodologias. No passado, era comum o projeto iniciar com uma descrição em alguma linguagem de programação de propósito geral, como C/C++, e após um processo de validação do comportamento do sistema, o projetista (ou alguém da equipe de projeto) ficava responsável pelo refinamento de tal descrição dentro de uma linguagem que permitisse síntese, geralmente uma HDL (*hardware Description Language* ou linguagem de descrição de hardware), como VHDL ou Verilog (Figura 1). Nesse processo é necessário reescrever todo o código escrito na linguagem de alto nível e dificilmente o conjunto de testes codificado na primeira etapa é reaproveitável.

É perceptível que a abordagem da Figura 1 necessita do conhecimento de duas linguagens com características distintas (no exemplo, C/C++ e VHDL), além do que o processo de reescrita do código fonte acarreta grande suscetividade a erros.

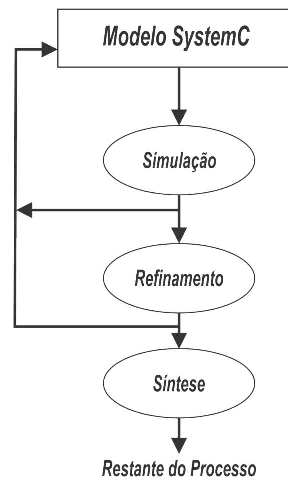
Tendo em vista a problemática do retrabalho atrelado ao uso de duas ferramentas (linguagens) para construção do projeto, o SystemC foi desenvolvido (INSTITUTE OF

Figura 1 – Metodologia antiga de projeto a nível de sistema



ELECTRICAL AND ELECTRONICS ENGINEERS, 2012). O SystemC é uma biblioteca para o C++ que integra as características das linguagens de alto nível às definições das linguagens de descrição de hardware, assim possibilitando especificação em nível de sistema e especificação para síntese dentro da mesma linguagem. A Figura 2 resume a metodologia de projeto introduzida pelo SystemC.

Figura 2 – Metodologia de projeto a nível de sistema com SystemC



Na abordagem possibilitada pelo SystemC, a codificação do sistema é feita em uma única linguagem. Isso permite que o conjunto de testes para as simulações em alto nível, utilizado para a validação do sistema, possa ser reutilizado em conjunto com o código de definição do sistema nas fases de refinamento, assim evitando retrabalho.

Outro problema que surge do amadurecimento do mercado é a integração de uma variada gama de componentes *IP* (*Intellectual Property* ou de Propriedade Intelectual) a modelos em desenvolvimento. A ideia é que tais componentes sejam "acoplados" à descrição do sistema independentemente de seu nível de abstração e do próprio sistema,

assim permitindo uma avaliação do projeto o mais cedo possível, e a criação de ambientes de simulação onde as aplicações em software podem ser modeladas mais cedo e avaliadas junto com o modelo de hardware. Esta última característica desejada, em geral, diminui o tempo de projeto, e consequentemente o tempo de lançamento no mercado, visto que quase sempre o projeto de software em sistemas eletrônicos é a fase que fecha o ciclo de projeto.

A TLM (*Transaction-Level Modeling*)é uma metodologia que possibilita a comunicação entre blocos funcionais de diferentes níveis de abstração de forma independente desses níveis (BLACK et al., 2009). Essa ideia impacta tanto no projeto de refinamento do próprio bloco quanto na comunicação entre módulos/blocos dentro do sistema. Isso é importante, pois o mais recorrente é que, durante a produção do sistema, diferentes módulos tenham granularidades diferentes. Sendo assim, a TLM possibilita avaliar o sistema como um todo independentemente das diferenças de granularidade dos módulos que o compõem.

Tendo em vista a grande variedade de metodologias ESL este trabalho tem por objetivo ser um material teórico de referência sobre as metologias de projeto de sistemas eletrônicos (por exemplo, ESL, Double Roof, TLM) com a linguagem SystemC. Depois de elencar três escolas de abordagem para projeto de sistemas eletrônicos, o Capítulo 2 descreve o modelo de diagrama Y, importante para a definição das principais formas de representação hoje utilizadas. Como uma generalização do diagrama Y o Capítulo apresenta a metodologia de telhado duplo, sendo uma metodologia que faz a separação clara entre o processo de síntese de hardware e software. Em seguida o Capítulo 3 apresenta a metodologia TLM como uma metodologia complementar as metodologias ESL, discutindo o problema de comunicação entre componentes em diferentes granularidades. O Capítulo 4 Discute as necessidades das metodologias ESL e TLM e apresenta o SystemC como uma ferramenta que suporta essas necessidades. O Capítulo 5 faz uma introdução do SystemC demonstrado suas principais características. Por fim o Capítulo 6 apresenta as considerações finais do trabalho.

2 METODOLOGIAS DE PROJETO

Este Capítulo compreende um primeiro estudo sobre as metodologias para projeto de sistemas eletrônicos digitais. Como já mencionado, o estudo dessas metodologias se faz necessário devido ao aumento da complexidade dos sistemas eletrônicos e da quantidade de aplicações que vem surgindo. Toda essa problemática já é antiga, como sugerem Gajski e Kuhn (1983), onde elencam as três principais abordagens da época para superar a chamada "crise de projeto VLSI de 80".

A seção a seguir faz uma breve descrição dessas três abordagens utilizadas na década de 1980 durante tal crise e discute os motivos da terceira abordagem ser a mais utilizada hoje.

2.1 Três escolas de abordagem

A **primeira escola** segue uma linha de projeto assistido por computador. Nessa escola acredita-se que as decisões de projeto devam ser feitas por projetistas humanos que possuam boas práticas de projeto acumuladas em anos de trabalho no próprio mercado. As principais ferramentas utilizadas nessa abordagem são:

- Editores gráficos;
- Ferramentas de simulação e verificação;
- Bases de dados.

Essa abordagem tende a ser *bottom-up*, isto é, contrói-se primeiro blocos menores que serão utilizados posteriormente em estruturas de mais alto nível (GAJSKI; KUHN, 1983).

A **segunda escola** segue uma linha de abordagem guiada por base de conhecimento. A ideia é construir uma base de dados formada por regras de projeto, que nada mais são do que o conhecimento de especialistas humanos em projeto de sistemas eletrônicos descrito na forma de regras. Essas regras descritas na base devem fazer a tomada de decisões do projeto, não abordando processos "mecânicos", como simulação e verificação. Essa base de dados pode ser constantemente incrementada e melhorada por especialistas que façam seu uso para apoio a decisões em seus projetos. Os sistemas que utilizam essas bases de dados são chamados de Sistemas Especialistas (GAJSKI; KUHN, 1983).

Por último, a **terceira escola** segue uma linha de pensamento baseada na síntese a nível de sistema (ESL). Essa abordagem é uma das mais comuns atualmente (GERSTLAUER et al., 2009) (MEEUS et al., 2012). Em adição a tudo que já foi dito sobre

ESL, a ideia é que o conhecimento de especialistas, principalmente no que tange a especificação, além de poder ser descrito como regras (segunda escola), também pode ser traduzido de forma automatizada em uma implementação do sistema, ou seja, a partir de uma descrição de alto nível do problema a solução pode ser alcançada por meio de síntese automática.

A vantagem da ESL vai além da diminuição do tempo de produção uma vez que gera a possibilidade da exploração de vários modelos de projeto de forma rápida. Outra vantagem dessa abordagem está fundamentada na ideia de que projetistas humanos, apesar de serem muito bons para otimizar uma pequena quantidade de elementos no projeto, são extremamente limitados para encontrar o ótimo quando se trata de um projeto com uma larga quantidade de componentes (GAJSKI et al., 1992), que é uma realidade que só aumentou ao longo dos anos. O trabalho Teich (2000) apresenta uma seção sobre o problema de exploração do espaço de projeto para sistemas eletrônicos e o trabalho Geilen et al. (2007) faz a mesma discussão de forma mais abrangente para sistemas computacionais em geral. O problema de alocação ótima de recursos conflitantes em um espaço de soluções é comumente chamado na literatura de pontos de pareto.

Apesar das vantagens da abordagem ESL, como identificaram Gajski et al. (1992), existem inúmeros desafios para que ela seja eficiente, como: definição de linguagens de descrição, modelos de projeto, algoritmos de síntese e ambientes para síntese iterativa onde os projetistas possam facilmente reduzir o espaço de busca pela solução.

2.2 Modelos de representação e Diagrama Y

Outra contribuição de Gajski e Kuhn (1983) foi a formalização e agrupamento de três domínios de descrição de sistemas eletrônicos, cada um com seus respectivos níveis de granularidade, o que permite definir fluxos de síntese entre esses níveis, além de representar graficamente os tipos de síntese.

Esses três domínios podem ser resumidos no diagrama Y (Figura 3) e são descritos a seguir:

- **Comportamental**: Também chamado de funcional, este eixo de representação do sistema descreve apenas o que o sistema faz, sem qualquer definição de como é feito. Pensando em um somador de um bit, a tabela verdade (Tabela 1) a seguir representa o comportamento de tal componente. Essa tabela verdade indica apenas o comportamento da saída do sistema de acordo com as possíveis entradas. Note que nada é dito sobre quais componentes irão ser utilizados para implementação, ou como eles serão ligados - essa afirmação fica mais clara ao observar a definição do domínio estrutural. Como indicado pela Figura 3, comumente os níveis do eixo comportamental são: sistema, algoritmo e expressão booleana.

Figura 3 – Diagrama Y

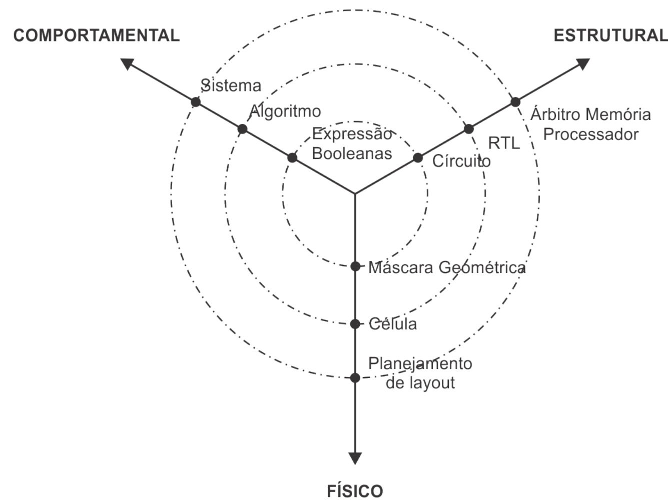


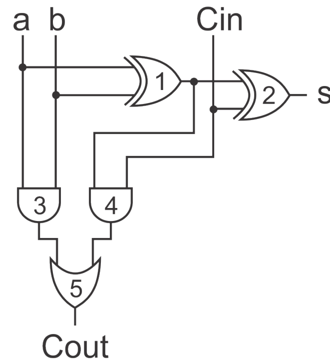
Tabela 1 – Tabela verdade de um somador completo de 1 bit

cin	a	b	s	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- **Estrutural**: neste eixo é introduzido um conjunto de componentes e ligações sujeitos a restrições, que, normalmente são, área, custo e tempo. Esses componentes e ligações são mapeados a partir do comportamento do sistema, mas sem especificar informações físicas da implementação, como disposição dos componentes e ligações. Seguindo o exemplo da Tabela 1 e considerando os componentes, duas portas lógicas *XORs*, duas *ANDs* e uma porta *OR*, uma representação estrutural possível é a da Figura 4. Note que a representação não fornece qualquer informação da disposição das portas lógicas e da disposição de suas ligações. Geralmente os níveis do eixo estrutural são: seletores memória-processador, RTL e circuitos. Às vezes, representações estruturais podem ser utilizadas como comportamentais, geralmente em simulações onde se usa esta representação para determinar comportamentos do sistema, como atrasos de tempo.

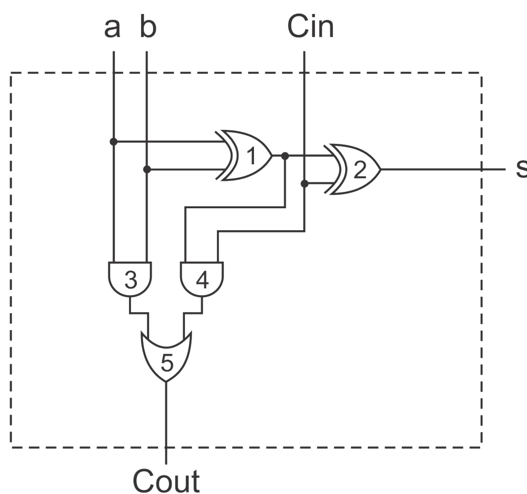
- **Físico**: Na representação física o foco está em definir a disposição dos componentes da representação estrutural e suas ligações, no espaço interno de um encapsulamento ou até mesmo no próprio silício em projetos mais completos, sem preocupar-se com o que de fato os componentes representam ou fazem. A Figura 5 mostra a disposição

Figura 4 – Representação estrutural de um somador de 1 bit



física para as portas lógicas e suas ligações do exemplo da Figura 4 no layout de uma possível célula de um CI ou até mesmo de um encapsulamento.

Figura 5 – Disposição física dos componentes de uma implementação para um somador de um bit

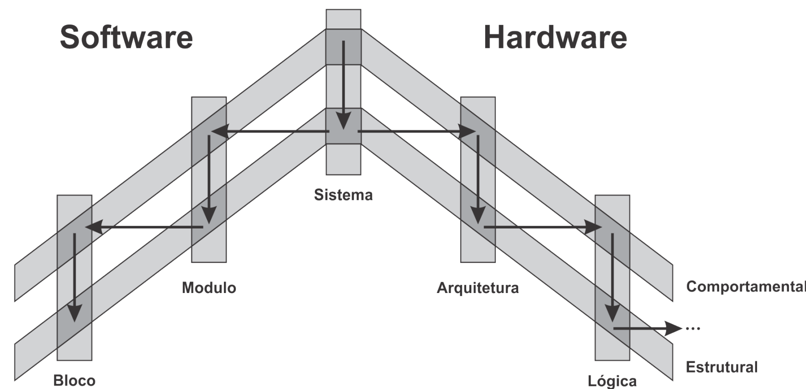


Vale lembrar que os níveis de abstração descritos em cada eixo da Figura 3 constituem apenas um exemplo geral e eles podem e devem ser adaptados de acordo com cada projeto e visão dos projetistas.

2.3 Metodologia de telhado duplo e a separação entre hardware e software

O modelo de telhado duplo (*double roof*) é uma metodologia *top-down* para projetos de sistemas embarcados que pode ser vista como uma variante do diagrama Y, apresentando apenas os domínios de representação comportamental (telhado superior) e estrutural (telhado inferior) (Figura 6). Segundo Teich (2000), esses dois domínios são suficientes para o entendimento do problema de síntese, foco principal das metodologias ESL.

Figura 6 – Diagrama de telhado duplo



Além dessa divisão entre domínios em dois telhados, outro diferencial, quando comparado ao diagrama Y e a outras metodologias, é a divisão clara entre hardware e software, representados como lados do telhado na Figura 6. Como exposto no Capítulo 1, uma metodologia dita ESL deve tratar também do projeto de software. Isso é explicitamente feito no diagrama de telhado duplo, como demonstra a Figura 6.

Na figura 6 os retângulos representam alguns dos principais níveis de abstração de um projeto de sistemas embarcados. A seguir é apresentada uma descrição geral desses níveis segundo Teich (2000).

- **Sistema**: Modelos neste nível de abstração são descrições completas do sistema, geralmente apresentadas como módulos responsáveis por partes do comportamento do sistema (como algoritmos). Alguns exemplos de tais módulos são processadores, unidades de hardware dedicado, ASICs, barramentos, memórias, etc..

- **Arquitetura**: Como apresentado na Figura 6 este nível está do lado do hardware e compreende componentes funcionais que se comunicam. Esses componentes implementam funções lógicas e aritméticas de alta granularidade. Tais blocos funcionais também são conhecidos como blocos a nível de transferência de registrador.

- **Lógica**: Também do lado do hardware, modelos neste nível geralmente são descritos como redes de conexões (*netlists*) de portas lógicas e registradores que implementam máquinas de estado e funções booleanas.

- **Módulo**: Nível pertencente ao lado de software. Compreende modelos que vão de redes de processos, grafos de nível de tarefas, linguagens com suporte à threads, etc., ou seja, modelos que desempenham comunicação entre funções de comportamento complexo. Esses modelos são mapeados para processadores de núcleo único, com ou sem suporte a sistemas operacionais multitarefa, ou arquiteturas com multiprocessador em software.

- **Bloco**: Este nível apresenta modelos comportamentais, que são tipicamente funções e procedimentos de linguagens de alto nível, que devem ser refinados em modelos estruturais na forma de instruções de máquina para o processador que será utilizado para execução.

Assim, o processo de desenvolvimento de um sistema embarcado complexo consiste de uma sequência de refinamentos dentro de um mesmo nível de abstração, adicionando informações estruturais a uma descrição comportamental, concebendo uma descrição estrutural de mesmo nível. Tais informações estruturais são na verdade informações sobre a implementação do sistema. Sendo assim, o comportamento do sistema é iterativamente refinado nos próximo níveis de abstração de mais baixo nível (TEICH, 2000).

Ainda segundo Teich (2000), o problema de síntese pode ser definido como o processo de refinamento de uma representação comportamental em uma representação estrutural dentro de algum nível de abstração. Esse processo de refinamento, por sua vez, pode ser subdivido em três partes principais:

- **alocação** de recursos, que são objetos estruturais responsáveis por executar determinada tarefa definida na representação comportamental;

- **ligação** ou mapeamento entre os recursos alocados e os objetos estruturais e;

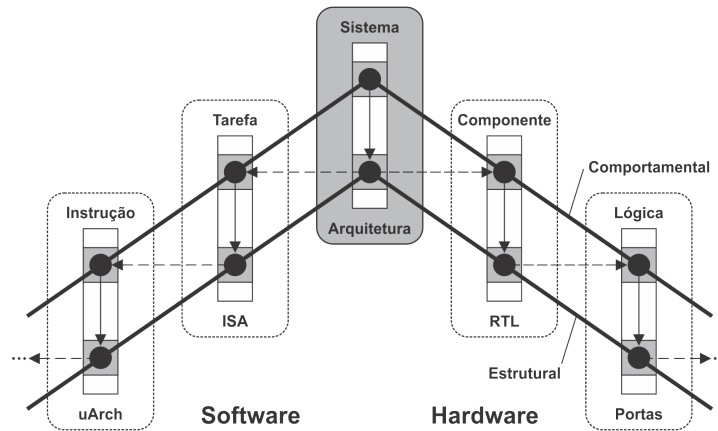
- **agendamento** (ou escalonamento) dos objetos comportamentais sobre os recursos alocados (objetos estruturais). O escalonamento pode ser definido como o intervalo de tempo de execução dos componentes comportamentais, ou somente a ordem de execução sobre os componentes alocados.

As três tarefas descritas anteriormente são independentes do nível de abstração, com variação apenas em relação à granularidade dos objetos comportamentais e estruturais, aos algoritmos de otimização que desempenham as tarefas de síntese e aos objetivos a serem otimizados em cada fase da síntese. Uma discussão mais profunda do problema de síntese é abordada no trabalho de Teich (2000).

A Figura 7 foi traduzida do trabalho de Gerstlauer et al. (2009) e é uma representação um pouco mais descritiva - e recente - do modelo original de telhado duplo (Figura 1). Dados os cursos e livros atuais de arquitetura de computadores, provavelmente essa versão apresenta conceitos mais concretos e familiares às pessoas com conhecimento básico em tal área. Os objetos comportamentais e estruturais são explicitamente nomeados no diagrama. Por exemplo, o domínio que na Figura 6 é o de mais alto nível, agora nomeia seus objetos estruturais como arquitetura, o que soa mais adequado, devido à suas granularidades (processadores, barramentos, memórias, etc.).

Por fim, vale ressaltar que, em essência, as duas representações do modelo de telhado duplo (Figura 6 e Figura 7) são naturalmente semelhantes, sendo a segunda uma

Figura 7 – Uma versão possível do diagrama de telhado duplo



representação mais moderna da primeira - o que é sustentado, pois o autor do modelo original (TEICH, 2000) faz parte do grupo de autores do artigo que apresenta essa adaptação do primeiro modelo (GERSTLAUER et al., 2009).

2.4 Duas metodologias essenciais e um problema não abordado

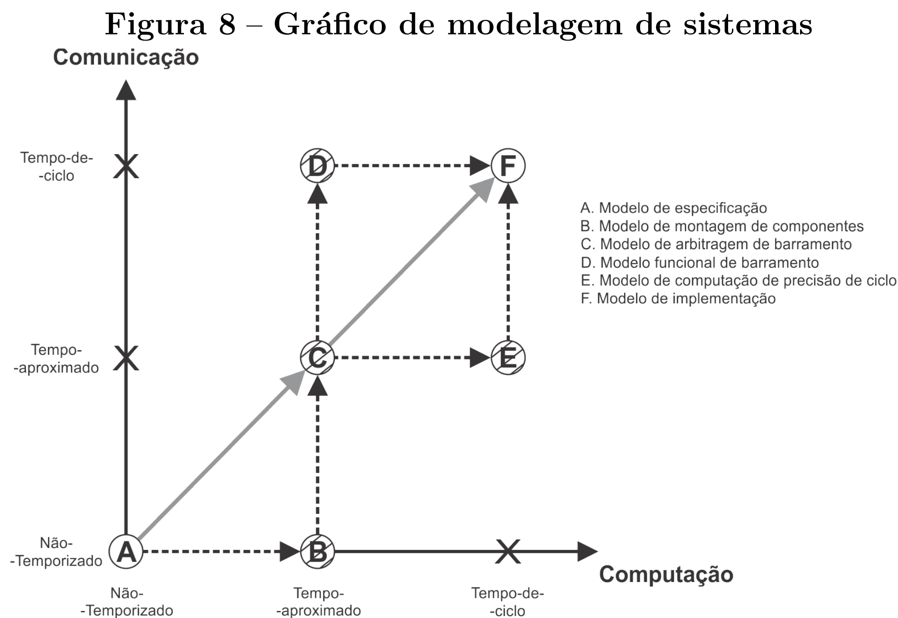
Neste Capítulo foram discutidas duas metodologias centrais no desenvolvimento de sistemas eletrônicos (diagrama-Y e telhado-duplo). Sob ótica *top-down*, as duas metodologias tratam do processo de refinamento ou síntese de modelos de componentes até o nível mais baixo de representação. O que essas metologias não tratam é o problema de integração de componentes com diferentes níveis de granularidade. Esse problema surge em dois cenários diferentes: equipes distintas trabalham em partes diferentes do sistema e; reutilização de componentes de projetos "prontos" em novos projetos, vide componentes IP.

Pensando nessa problemática, o próximo Capítulo apresenta uma metodologia de projeto para comunicação entre componentes com diferentes níveis de abstração, o que visa avaliar o desempenho e validar o sistema já nas primeiras fases do projeto.

3 MODELAGEM EM NÍVEL DE TRANSAÇÃO (TLM)

No artigo de Cai e Gajski (2003), os autores fazem uma descrição resumida de seis modelos de especificação de projeto presentes em Cai, Verma e Gajski (2003), sendo quatro modelos TLMs (*Transaction-level Modeling*), e apresentam uma introdução ao uso destas TLMs em diferentes domínios de projeto.

Figura 8 – Gráfico de modelagem de sistemas



O gráfico da Figura 8 correlaciona esses seis modelos. Os eixos horizontal e vertical representam, respectivamente, componentes de computação e comunicação. Esses dois tipos de componentes são descritos nos eixos em relação a suas precisões de tempo. Essas precisões de tempo são divididas em: Não-temporizada, de tempo-aproximado e de tempo-de-ciclo. A computação/comunicação não-temporizada representa apenas a funcionalidade do sistema, sem qualquer detalhe de implementação presente. A computação/comunicação de tempo-aproximado apresenta implementações em nível de sistema, como a arquitetura selecionada para o sistema e o mapeamento de relações entre processos da especificação do sistema e os elementos de processamento da arquitetura selecionada. Por último, a computação/comunicação de tempo-de-ciclo apresenta detalhes de implementação e de estimativa de tempo tanto em nível de sistema quanto em RTL(*Register Transfer Level* ou Nível de transferência de registrador)/ISS(*Instruction Set Simulation* ou simulação do conjunto de instruções), o que não ocorre com a computação/comunicação de tempo-aproximado, que apresenta apenas estimativa de tempo em nível de sistema (CAI; GAJSKI, 2003).

Dentro de um projeto com TLMs, a comunicação entre os módulos de computação é feita por meio de chamadas de funções de interface a um canal de comunicação (*channel*

ou simplesmente canal) (CAI; GAJSKI, 2003). Dessa forma, os detalhes de computação ficam totalmente separados dos detalhes de comunicação.

Tratando agora dos modelos contidos na Figura 8, indicados por círculos, as quatro TLMs (círculos rasurados) são modelos intermediários entre o modelo de especificação (nível de sistema) e o modelo de implementação (nível de implementação de fato). Vale ressaltar que as setas interligando os modelos representam o processo de síntese entre modelos de diferentes níveis. Nesse cenário, uma ferramenta/metodologia de síntese ESL deve chegar ao modelo de implementação partindo do modelo de especificação. A seguir é apresentada uma breve descrição de cada modelo.

Ao observar a Figura 8 é possível perceber que existem três modelos não especificados (dois sobre o eixo da comunicação e um no eixo da computação). Eles não são abordados no artigo original e também não serão abordados neste trabalho porque, segundo Cai e Gajski (2003), esses modelos são pouco utilizados.

(A) **Modelo de especificação**: Este modelo representa a pura funcionalidade do sistema, sem conter qualquer detalhe sobre sua implementação. Como indica a Figura 8 não há estimativa de tempo nesse modelo. Nesse nível toda a comunicação pode ser feita pelo acesso à variáveis, sem haver o uso de canais. Geralmente este modelo é feito em uma linguagem como C/C++, o que facilita a conversão para linguagens como SystemC durante o processo de refinamento para outros modelos.

(B) **Modelo de montagem de componentes**: Neste modelo a computação apresenta grau de tempo-aproximado e comunicação não-temporizada. As entidades no nível mais alto do modelo são memórias globais e elementos de processamento (*Processing Elements* ou PEs) com execução concorrente. Os chamados PEs podem ser processadores de propósito geral, DSPs (*Digital Siginal Processor*), IPs ou até mesmo um hardware dedicado. Os canais não apresentam qualquer implementação de barramento/protocolo e representam apenas transferência de dados ou sincronização entre PEs, que é feita por meio de passagem de mensagens. A estimativa de tempo é feita especificamente para cada PE por alguma ferramenta para tal propósito e, posteriormente, é anotada no código por meio de declarações de espera (*wait*).

(C) **Modelo de arbitragem de barramento**: Como sugere a Figura 8, este modelo é um refinamento do modelo de montagem de componentes em relação à comunicação. Diferente do modelo de montagem de componentes, no modelo de arbitragem de barramento os canais de comunicação são representações de barramentos chamados de barramentos abstratos. Os canais continuam implementando transferência de dados por meio de passagem de mensagens, porém o diferencial é a implementação de barramentos, que são simples representações de E/S com dois estados exclusivos: bloqueando e não bloqueando. A estimativa de tempo nos canais é feita por

meio de declarações de espera para cada transação. Especificação de protocolos de comunicação, tais como pinagem e precisão de ciclo, não estão presentes. Como é comum o agrupamento de vários canais de comunicação para um mesmo canal de barramento abstrato, dois elementos são inseridos às funções de interface dos canais: endereço lógico e prioridade de barramento, que são, respectivamente, utilizados para distinguir chamadas de funções de interface de diferentes PEs, ou processos, e determinar a sequência de acesso do barramento quando ocorrem conflitos de acesso ao barramento. Outro elemento inserido na arquitetura do sistema é o árbitro de barramento, que é tratado como um novo PE e é responsável pelo tratamento de conflitos no barramento.

(D) **Modelo funcional de barramento**: Neste modelo a comunicação é especificada em nível de precisão de tempo/ciclo e a computação é de tempo-aproximado. Este modelo se divide em duas especificações sucessivas: modelo com comunicação de precisão de tempo e com precisão de ciclo. A primeira especifica as limitações de tempo da comunicação, que são definidas pelo diagrama de tempo do protocolo do componente. A segunda é um refinamento da primeira com a estimativa do tempo sendo feita em relação ao ciclo de *clock* do componente principal (ou mestre) da arquitetura. Esse processo de refinamento é chamado de refinamento de protocolo. Mais especificamente, os canais de passagem de mensagens são substituídos por canais de protocolo, que tem como diferencial apresentar precisão de tempo/ciclo e precisão de pinagem. Aqui, um fio de barramento dentro do canal de protocolo é representado pela instanciação de variáveis ou sinais. Na interface de um canal de protocolo são fornecidas funções para todas as transações de barramento abstratos. Um canal abstrato que contém um canal de protocolo é chamado de canal de barramento detalhado. Uma última ressalva em relação ao modelo funcional de barramento é que nem todos os canais de barramento abstratos precisam ser refinados em canais de barramento detalhados. Alguns canais podem ser refinados enquanto outros não são alterados.

(E) **Modelo de computação de precisão de ciclo**: Este modelo apresenta computação de precisão de tempo-de-ciclo e comunicação de tempo aproximado. Aqui os PEs tem precisão de pinagem e executam com precisão de ciclo. O projeto dos componentes de hardware dedicado é feito em nível de transferência de registradores e componentes como DSPs e processadores de propósito geral são modelados levando em conta precisão de ciclo da arquitetura do conjunto de instruções. Em relação à comunicação, envólucros que convertem dados de níveis de abstração altos para níveis baixos, e vice-versa, são inseridos para conectar os PEs e as interfaces de barramento, o que permite a comunicação de componentes especificados com precisão de tempo-de-ciclo às interfaces de comunicação dos canais de barramento abstratos.

Como no modelo funcional de barramento, não é necessário o refinamento de todos os PEs para precisão de tempo-de-ciclo.

(F) **Modelo de implementação**: Este modelo representa a implementação precisa do sistema. Neste modelo, computação e comunicação apresentam precisão de tempo-de-ciclo. Os componentes são definidos em termos de transferência de registradores ou em termos da arquitetura do conjunto de instruções.

A Tabela 2 resume os seis modelos e descreve suas principais características.

Tabela 2 – Características dos diferentes modelos abstratos

Modelos	Tempo de comunicação	Tempo de computação	Esquema de comunicação	Interface de PE
Modelo de especificação	não possui	não possui	variável/canal	sem PE
Modelo de montagem de componentes	não possui	aproximado	canal de passagem de mensagem	abstrato
Modelo de arbitragem de barramento	aproximado	aproximado	canal de barramento abstrato	abstrato
Modelo funcional de barramento	precisão de tempo/ciclo	aproximado	canal de barramento detalhado	abstrato
Modelo de computação de precisão de ciclo	aproximado	precisão de ciclo	canal de barramento abstrato	precisão de pinagem
Modelo de Implementação	precisão de ciclo	precisão de ciclo	fio	precisão de pinagem

Abstraindo as definições dadas anteriormente sobre TLM, pode-se imaginar um modelo de TLM como um possibilitador do projeto conjunto de diferentes componentes em diferentes níveis de abstração. Um modelo TLM não só possibilita o projeto paralelo de diferentes componentes de hardware com uma valoração total do sistema, como também possibilita o projeto de software mais cedo.

No Capítulo 4 é discutido um pouco das pressões por ciclos mais curtos de desenvolvimento, a necessidade do projeto conjunto entre hardware e software e como a ESL e a TLM são ferramentas poderosas para alcançar melhorias no desenvolvimentos dos complexos sistemas eletrônicos atuais. Nesse sentido, como é demonstrado, o SystemC é uma ferramenta essencial para suprir as necessidades da ESL e da TLM.

4 NÍVEL DE SISTEMA ELETRÔNICO E MODELAGEM EM NÍVEL DE TRANSAÇÃO

As necessidades das metodologias ESL e da TLM são diferentes das providas por HDLs, como VHDL e Verilog, e de linguagens convencionais de desenvolvimento de software, como C/C++ e Java. A seção a seguir descreve quais são essas necessidades, introduzindo a importância do SystemC para a atual conjectura do desenvolvimento de sistemas eletrônicos.

4.1 Necessidades da ESL e da TLM

Além do crescimento na complexidade, os sistemas eletrônicos atuais normalmente integram hardware e software de aplicação específica, o que pressupõe a necessidade do projeto conjunto desses dois componentes. Esse projeto está sujeito a prazos curtos de desenvolvimento, restrições de desempenho de tempo real e pressões por baixo consumo energético. Outra necessidade é a verificação funcional de forma cuidadosa a fim de garantir que o sistema esteja livre de falhas e, obviamente, esteja de acordo com as especificações do sistema (BLACK et al., 2009).

Sistemas eletrônicos modernos são formados por inúmeros subsistemas e componentes de hardware e de software. Segundo Black et al. (2009), ESL tem foco principal no desenvolvimento de hardware, software e algoritmos em nível de arquitetura do sistema. Cada um desses assuntos tem sua complexidades por si só, o que é aumentada pelos conflitos de balanceamento entre eles (*trade-offs*). Questões como "qual a precisão mínima necessária para o algoritmo funcionar adequadamente?" podem ser solucionadas tanto com um hardware com maior precisão, um software que faça o tratamento desta precisão para o hardware ou até mesmo a mudança no algoritmo utilizado para gerar a solução.

No passado, quando, em geral, os sistemas eletrônicos possuíam tamanho tratável por uma única pessoa, era comum que este mesmo indivíduo ficasse responsável por todas as decisões envolvendo os três componentes principais do sistema - hardware, software e algoritmos. Pelo próprio escopo "reduzido" do sistema, que permitia o trabalho com uma única pessoa ou equipe reduzida - com provavelmente um ou mais *experts* nos três componentes -, a integração e tratamento dos componentes era facilitada. A medida que esses sistemas foram crescendo, as equipes também cresceram. Como usualmente os processos eram organizados com alta dependência entre etapas, havia uma serialização (ou cascata) (BLACK et al., 2009). Esse processo é análogo ao dos primeiros fluxos de software tradicional, que também vieram a apresentar seus problemas. Uma forma de melhorar o desenvolvimento dos sistemas, foi mitigar essa serialização.

Os primeiros sistemas tinham como prazo/processo final do sistema o desenvolvi-

mento do software. Por tal, fazer o desenvolvimento do software acontecer mais cedo, ou seja, paralelo a outras atividades dentro do ciclo de desenvolvimento do sistema se tornou um objetivo crucial da ESL. Em termos monetários, lançar um produto, mesmo que seja um mês antes, pode gerar ganhos milionários a uma empresa (BLACK et al., 2009).

Um modelo TLM, por exemplo, possibilita que diferentes disciplinas do projeto, como hardware e software, trabalhem de forma paralela. Essa TLM, ou protótipo virtual do sistema, representa uma especificação comum entre os grupos de desenvolvimento.

4.1.1 Metodologia TLM

No primeiro Capítulo do livro de Black et al. (2009), os autores discutem uma metodologia de desenvolvimento baseada na TLM. Como a Figura 9 demonstra, existe nessa metodologia uma divisão clara entre ambiente de projeto, desenvolvimento de software e uma fase de refinamento de hardware, suportada pelo desenvolvimento de um ambiente de verificação de hardware. Essa característica é semelhante à divisão entre hardware e software da metodologia de telhado duplo (Figuras 6 e 7).

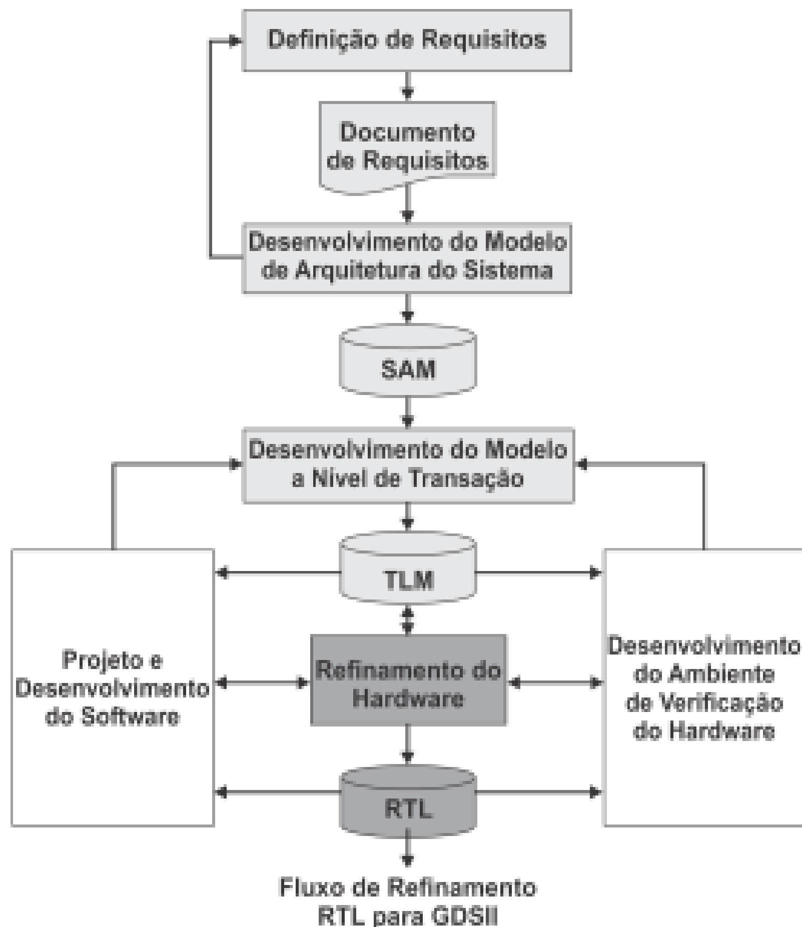
Na Figura 9, a primeira fase do fluxo de projeto é o levantamento de necessidades do sistema, ou definição do requerimento. Essa fase gera um documento de necessidades do produto (*Product Requirements Document* ou PRD). No próximo passo é definido um modelo para a arquitetura do sistema (*System Architecture Model Development*). Em seguida é definida uma TLM para comunicação entre os componentes da arquitetura do sistema (tanto de hardware quanto de software). Depois de várias fazes de refinamento dos modelos de hardware e do desenvolvimento do modelo de software que se adéqua, é gerado um modelo RTL do sistema. Isso fecha o fluxo de desenvolvimento de alto nível do sistema e os modelos RTL podem continuar sendo refinados, pois representam uma modelagem do sistema de mais baixo nível.

Vale ressaltar que a TLM produzida inicialmente é refinada constantemente à medida que for necessário junto com o progresso do desenvolvimento do software e do desenvolvimento do ambiente de verificação do hardware.

Ainda de acordo com Black et al. (2009), aplicando de maneira adequada uma linguagem e técnicas de projeto ao longo de todo esse fluxo de desenvolvimento, a TLM pode ser refinada e reutilizada, assim servindo para inúmeras tarefas de grande importância.

1. **Modelagem algorítmica**: Esta modelagem pode ser vista como a modelagem comportamental em nível de sistema. Nela são definidos os algoritmos específicos da aplicação, que posteriormente serão refinados em hardware ou software. Inúmeras questões são levantadas nessa fase, tais como: “Quais algoritmos implementar em hardware ou software?”, “Podeira uma solução apenas de software ser suficiente?”, etc. Tudo isso é feito dentro de um ambiente de desenvolvimento de software, que, comparado a um ambiente RTL, trás maior flexibilidade e facilidade para tratar

Figura 9 – Metodologia TLM



níveis de abstração. No desenvolvimento tradicional, a modelagem algorítmica seria executada em uma linguagem de alto nível (preferivelmente C/C++).

2. **Modelagem da Arquitetura**: Nesta modelagem são tomadas, majoritariamente, decisões de particionamento do sistema (hardware/software) e são tratadas questões sobre consumo energético, geralmente baseadas na análise do desempenho de barramentos. Na modelagem de arquitetura se faz necessário a existência de componentes de propriedade intelectual (IP), além de outros produtos/soluções para gerenciamento de parâmetros críticos e *trade-offs*. Uma ferramenta importante para executar essa tarefa é uma linguagem com suporte a modelagem e simulação de concorrência. Em tempos passados, os times de projetistas utilizavam majoritariamente C/C++, que possui expressividade de abstração de componentes (orientação a objetos), porém era necessária a implementação de *kernels* de simulação para o suporte de concorrência. Atualmente o SystemC é preferido por integrar todas as funcionalidades do C/C++, juntamente com o conceito de concorrência e algumas características de HDLs.

3. **Plataforma virtual de desenvolvimento de software**: Esta plataforma provê

um desenvolvimento dos componentes de software do sistema ainda nas fases de desenvolvimento/refinamento do hardware, ou seja, no começo do ciclo de desenvolvimento do projeto. Isso, além de proporcionar um ciclo de vida de desenvolvimento mais curto do projeto, possibilita melhor aproveitamento e integração dos componentes de hardware, sendo possível de acordo com necessidades vistas apenas na fase de software mudar o projeto do hardware, o que ajuda a garantir critérios de desempenho do sistema final. Nesse contexto, o SystemC se apresenta como uma escolha adequada, pois nativamente integra o C/C++, o que permite que os projetistas integrem um ISS dentro do modelo. Sendo assim, código C/C++ pode ser construído e executado dentro do processador em fase de modelagem. Essa técnica é chamada de Execução direta. O trabalho de Krishnan e Torrellas (1998) apresenta uma discussão sobre tal técnica.

4. **Refinamento do hardware**: Nesta tarefa, o foco está em manter a coerência com a especificação de alto nível. Dependendo da natureza do sistema pode ser construída uma especificação executável do hardware para manter a coerência entre a especificação e o refinamento. Isso pode evitar trabalho excessivo sobre o projeto, assim diminuindo tempo de trabalho e custo do projeto final. Questões como a latência do sistema e velocidade de ciclo são tratadas.

5. **Verificação funcional e arquitetural**: Durante o processo de refinamento do sistema é necessária verificação de que o modelo projetado reflete as necessidades da aplicação pretendida. Nisso, a aplicação da TLM e da ESL é crucial. Segundo Black et al. (2009), esse tipo de verificação no passado só era executada após a conclusão do projeto de software, que por sua vez dependia do hardware. Como já discutido, todas essas etapas são paralelizadas com uma modelagem ESL com TLM.

Grande parte dessas necessidades/tarefas pressupõe uma ferramenta/linguagem com abstrações de alto nível para a modelagem do hardware e criação de componentes de software, como também a simulação de atividades concorrentes. Nesse sentido, o SystemC é um grande auxiliador pois, junto com seu *kernel* de simulação, as características do C/C++ e algumas abstrações de HDL atinge bem essas necessidades.

5 SYSTEMC: INTEGRANDO O PROJETO DE HARDWARE A UMA LINGUAGEM DE ALTO NÍVEL

Este Capítulo apresentar o SystemC (atualmente na versão 2.3) de maneira objetiva. O SystemC, assim como o projeto eletrônico em nível de sistema, abarca um estudo multidisciplinar. O SystemC permeia assuntos como o projeto de sistemas digitais, arquitetura de computadores, noções de soluções para sistemas operacionais e programação paralela. Por tal, o presente Capítulo não aborda o SystemC de maneira completa, mas expõe seus fundamentos, a fim de formar uma base inicial para seu estudo. Para uma abordagem mais completa recomenda-se a própria documentação do SystemC (INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, 2012) e livros como os de Black et al. (2009) e Banerjee e Sur (2014).

5.1 Visão geral sobre o SystemC

O SystemC foi construído como uma biblioteca C++, que por sua vez tem como alvo questões de software. Por tal, o SystemC é na verdade um extensão do C++ para projetos que envolvam questões de hardware e sua interface com software, mas não somente. De fato o SystemC facilita o projeto de sistemas que em seu comportamento possuam atividades relacionadas à passagem de tempo, concorrência e reações a eventos (BANERJEE; SUR, 2014). Características como tempo, concorrência e eventos são fornecidas pelo *Kernel* de simulação do SystemC.

Os principais conceitos que o SystemC adiciona ao C++ são apresentados a seguir.

- **Tipos de dados para hardware**

  Diferente dos tipos de dados nativos do C++ que tem largura de bits dependentes do processador, o SystemC fornece tipos de dados com largura de bits reguláveis, além de suportar operações lógicas, de seleção e atribuição de bits. O SystemC fornece tipos de dados para valores inteiros e tipos de dados de ponto-fixo, além disso fornece tipos de dados para representação de hardware digital não binárias (valor desconhecido e alta impedância).

- **Conceito de temporização para simulação**

  O registro do tempo em SystemC é feito utilizando a classe `sc_time` , que possui 64 bits de precisão. O tempo decorre internamente ao *kernel* de simulação e são fornecidos mecanismos para implementar atrasos específicos de tempo.

- **Estrutura e Hierarquia**

A hierarquia em SystemC se dá pela instanciação de módulos dentro de outros módulos. Isso é feito de forma semelhante a instanciação de objetos dentro de classes. O que de fato ocorre é que módulos, na verdade, são instâncias de objetos de uma classe especial, no sentido da orientação a objetos, que o SystemC fornece.

- **Comunicação entre módulos**

  Esquemas de comunicação podem ser representados por canais. Esses canais podem ser mapeados para estruturas complexas como barramentos de hardware padronizados, ou mesmo simples estruturas como fios ou uma fila FIFO (*First-in First-out* ou primeiro a entrar, primeiro a sair).

- **Concorrência**

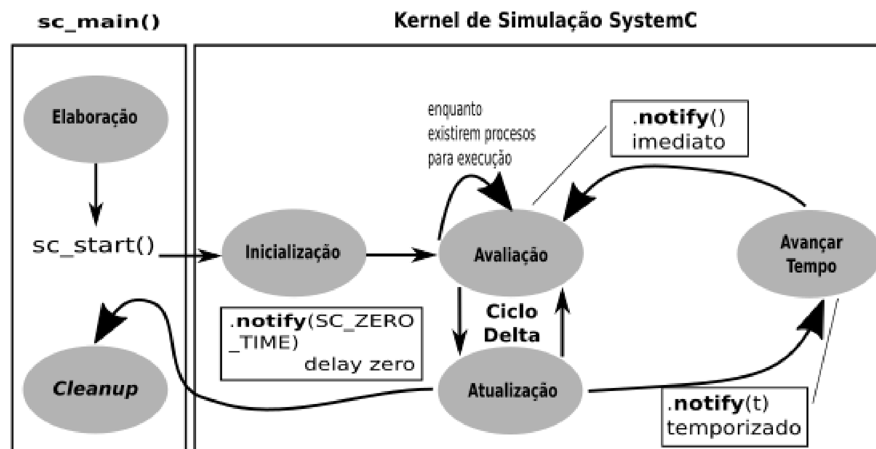
  A representação de concorrência é feita pela simulação de unidades concorrentes. A cada unidade é permitida a execução até que a simulação de outra unidade seja necessária para manter o comportamento alinhado com o tempo (responsabilidade do projetista). O *kernel* de simulação segue o modelo multitarefa cooperativo, ou seja, a troca entre processos durante a simulação é determinado pelo próprio código.

5.2 O *Kernel* SystemC

Antes de comentar sobre questões de sintaxe do SystemC é importante compreender como se dão as fases do processos de simulação de um modelo SystemC.

A execução de uma simulação SystemC é dividida em duas partes principais: elaboração e execução. Ainda é possível pensar em uma terceira fase se chama pós-processamento ou *cleanup*. Ela ocorre no fim da execução da simulação e tem o objetivo de tratar os dados gerados pela simulação.

Figura 10 – Processo de simulação SystemC



A Figura 10 descreve as fases do processo de simulação. O quadro a esquerda (`sc_main`) é análogo à função `main` do C++. Nessa função se inicia o programa e dentro dela todos os comandos executados antes da função `sc_start()` fazem parte da fase de elaboração. Na fase de **elaboração** é feita a inicialização das estruturas de dados, o estabelecimento de conectividade entre módulos e demais atividades de preparação para a fase de execução.

Na fase de **execução** os processos são executados dentro do *kernel* de simulação, que tem por objetivo criar uma ilusão de concorrência. Depois da execução da função `sc_start()` , todos os processos de simulação, com algumas exceções, são invocados de forma determinística não especificada na inicialização. Em outras palavras, a documentação não impõe uma ordem para a execução dos processos, porém nas implementações da biblioteca é necessária uma garantia de que execuções repetidas de uma mesma descrição de modelo tenham os mesmos resultados.

Depois da inicialização (Figura 10), os processos de simulação são executados quando ocorrem eventos aos quais eles são sensíveis. Como dito anteriormente, o SystemC implementa um ambiente multitarefa cooperativo. Uma vez iniciado, um processo em execução permanece nesse estado até que ele passe o controle. Vários processos de simulação podem iniciar no mesmo instante em termos de tempo de simulação. Nesse caso, todos os processos de simulação são avaliados e suas saídas são atualizadas. Uma avaliação seguida de uma atualização é conhecida como ciclo delta (mais comumente chamado de *delta cycle*).

No momento em que não há mais processos para avaliar naquele instante de tempo de simulação, então o tempo de simulação avança. Quando nenhum outro processo de simulação precisa ser executado, então a simulação é finalizada.

5.3 Tipos de dados SystemC

O SystemC fornece tipos de dados lógicos (binário e não-binário), inteiros e de ponto-fixo para projetos que envolvam modelagem de hardware. Os tipos de ponto-fixo são mais utilizados em projetos que envolvem DSPs (*Digital Signal Processors* ou Processadores de Sinais Digitais), o que está fora do deste trabalho e, por isso, não são abordados neste trabalho.

Além dos tipos de dados da biblioteca, as simulações podem utilizar tipos de dados nativos do C++, de outras bibliotecas ou tipos criados pelo usuário. Apesar disso, a escolha dos tipos de dados afeta diretamente a velocidade das simulações e a capacidade de síntese dos modelos. De forma direta, os tipos de dados do C++ aumentam o desempenho do tempo de simulação ao custo da fidelidade ao hardware e capacidade de síntese. Todos os tipos de dados do SystemC fazem parte do espaço de nomes `sc_dt` . A forma usual para o uso de tipos de dados da biblioteca SystemC é `sc_dt::sc_logic` , tomando o

tipo `sc_logic` como exemplo. Na maioria dos códigos apresentados adiante, o espaço de nomes sera omitido por questões de simplicidade (pressupõe-se o uso do comando `using namespace sc_dt`).

Com exceção do tipo `sc_logic` (bit único), todos os tipos de dados fornecidos pelo SystemC apresentam configuração de largura de bits mais flexível do que os tipos nativos do C++. O SystemC também fornece operadores de atribuição e inicialização com conversão de tipos, o que permite que tipos de dados C++, SystemC e *strings* C++ sejam utilizados na inicialização ou em operações de atribuição à tipos de dados SystemC. Todos os tipos de dados do SystemC implementam igualdade e operações bit-a-bit.

Os tipos de dados inteiros do SystemC também permitem operações aritméticas e relacionais. A implementação dessas operações é sematicamente compatível com as mesmas operações nativas do C++. Outro diferencial em relação aos tipos de dados nativos do C++ é a seleção de bit único e de intervalo de bits para operações de atribuição e como operandos de operações lógicas e aritméticas.

Antes de falar sobre os tipos, é importante lembrar que um programa SystemC nada mais é do que um programa C++ que faz uso das funcionalidades presentes na biblioteca SystemC. Para usar essas funcionalidades é necessário a instalação da biblioteca e sua inclusão no arquivo fonte (`#include <systemc.h>`).

5.3.1 Tipos lógicos

Os tipos de vetores lógicos do SystemC baseiam-se em dois tipos primitivos: `bool` do C++ e `sc_logic` do SystemC. São eles, respectivamente, `sc_bv<w>` e `sc_lv<w>`. Esses tipos de vetores lógicos foram projetados para modelos de baixo nível (RTL) e, por isto, não implementam operações aritméticas. Por outro lado implementam um conjunto bastante completo de operações lógicas e de atribuição.

O tipo `sc_bv<w>` (arqui `w` representa a largura de bits) suporta as operações bit-a-bit mais comuns: AND, OR, XOR e NOT, utilizando a sintaxe padrão do C++ (`&`, `|`, `^`, `~`). Além de seleção de bit único e de intervalo, `sc_bv<w>` também suporta operações binárias de redução (vide Tabela 3). O código da Figura 11 demonstra o uso de `sc_bv<w>` e de algumas funções especiais.

Tabela 3 – Principais funções de manipulação para tipos de dados SystemC

Seleção de bits	`bit(index), [index]`
Seleção de intervalo	`range(high, low), (high, low)`
Conversão para tipos C++	`to_int(), to_int64(), to_long() , to_uint(), to_uint64(), to_ulong() , to_string(sc_numrep)`
checagem	`is_zero(), is_neg(), length()`
Redução a um bit	`and_reduce(), nand_reduce(), or_reduce() , nor_reduce(), xor_reduce(), xnor_reduce()`

Figura 11 – Exemplo de uso do tipo `sc_bv<w>`

```
sc_bv<6> v1 = "000111";
sc_bv<6> v2 = "110000";
sc_bv<6> v3 = v1 | v2; // 110111
bool vReduce = v3.and_reduce(); // SC_LOGIC_0
v3[3] = "SC_LOGIC_1"; // 111111
vReduce = v3.and_reduce(); // SC_LOGIC_1
```

O tipo `sc_lv<w>` possui dois valores além do zero lógico e um lógico. A seguir estão descritos os valores aceitos por `sc_lv<w>` e os literais aceitos para suas representações.

- 0 lógico: “ `SC_LOGIC_0` ”, “ `Log_0` ” ou ‘ `0` ’;
- 1 lógico: “ `SC_LOGIC_1` ”, “ `Log_1` ” ou ‘ `1` ’;
- alta impedância: “ `SC_LOGIC_Z` ”, “ `Log_Z` ”, ‘ `Z` ’ ou ‘ `z` ’;
- desconhecido: ‘ `SC_LOGIC_X` ”, “ `Log_X` ”, ‘ `X` ’ ou ‘ `x` ’.

Com exceção dos valores adicionais, o tipo `sc_lv<w>` possui as mesmas características do tipo `sc_bv<w>` , o que implica dizer que as operações da Tabela 3 e as operações lógicas AND, OR, XOR e NOT também são aceitas.

5.3.2 Tipos inteiros

O SystemC apresenta versões com sinal e sem sinal em complemento de dois para inteiros. Uma das versões apresenta tamanho máximo de 64 bits. A outra tem precisão finita superior a essa. Esses tipos de dados fornecem funcionalidades não disponíveis nos inteiros nativos do C. Os tipos nativos do C++ possuem larguras que são dependentes do processador e do compilador, apesar de serem otimizados (em relação aos tipos do SystemC) para o conjunto de instruções da máquina. Além da configuração de bits,

os tipos inteiros do SystemC apresentam seleção de bit, seleção de intervalo de bits e operações de concatenação.

Os dois tipos com largura de até 64 bits são o `sc_int<w>` e `sc_uint<w>`, respectivamente com e sem sinal. Alguns projetos podem necessitar de tipos de dados com largura maior do que a dos nativos do C++, como é o caso de projetos que envolvam VLWI (*Very Long Word Instruction*), e, por tal, o SystemC apresenta os tipos `sc_bigint<w>` e `sc_biguint`, respectivamente com e sem sinal. Para ambos os tipos, com e sem sinal, as operações da Tabela 3 são aceitos. Também são aceitas as operações aritméticas e de atribuição aritméticas dos tipos aritméticos nativos do C++ (soma, subtração, divisão e etc.).

5.3.3 Representações literais no SystmC

O SystemC usa como base para suas representações literais as representações do C++. Por isso, tipos como inteiros do SystemC podem ser inicializados como inteiros do C++. Outra característica do SystemC é o uso das strings literais.

As strings literais do SystemC podem ser usadas para atribuir valores para qualquer tipo de dados SystemC. Elas consistem de um prefixo, uma magnitude e um caractere de sinal opcional ('+' ou '-'). O caractere de sinal não é permitido aos tipos sem sinal, binários, octal e hexadecimal. No caso dos três últimos, deve-se representar valores negativos por meio de complemento de dois.

Para a conversão de tipos de dados SystemC para strings C++ utiliza-se o seguinte método:

```
string to_string(sc_numrep rep, bool wprefix)
```

O parâmetro `rep` especifica qual a representação pretendida. A Tabela 4 apresenta as principais constantes para esse parâmetro, que estão presentes no espaço de nomes `sc_dt` do SystemC. Já o parâmetro `wprefix` é um valor booleano que indica à função se o prefixo é necessário na string resultado.

Por fim, as representações literais de string do SystemC e o *streaming* (entra/saída padrão) do C++ representam dados da mesma forma, o que implica na possibilidade de usar dados do SystemC como dados para E/S usando os operadores padrão do C++ (`>>` para entrada e `<<` para saída). Na saída, o mais comum é o uso da função `to_string()` com seus devidos parâmetros para definir a formação da saída dos dados.

Tabela 4 – Principais formas de representação de tipos SystemC em forma de string

sc_numrep	prefixo	significado	sc_int<5> = 13 sc_int<5> = -13
SC_DEC	0d	Decimal	0d13 -0d13
SC_BIN	0b	Binário	0b01101 0b10011
SC_OCT	0o	Octal	0o15 0o63
SC_HEX	0x	Hexadecimal	0x0d 0f3

5.4 A classe módulo do SystemC e processos registrados no construtor

Antes de entrar na criação de módulos do SystemC é necessário apresentar o início de todo programa. Assim como em programas C++ convencionais, os programas construídos em SystemC iniciam sua execução em uma função principal. Essa função é chamada de `sc_main()` e com exceção de seu nome, tem assinatura igual a função `main()` do C++. A seguir é apresentada sua estrutura básica (Figura 12).

Figura 12 – Estrutura básica da função `sc_main()`

```
#include <systemc.h>

int sc_main(int argc, char* argv[])
{
    // Elaboração

    sc_start(); // Simulação inicía e termina
                // nesta função

    // Pós-processamento (opcional)
    return 0;
}
```

O fluxo de execução de `sc_main()` geralmente é dividido em três fases distintas: elaboração, simulação e pós-processamento.

Durante a **elaboração** é estabelecida a conectividade do modelo (conexão entre módulos). A elaboração invoca o código para registrar processos de simulação e faz a conexão entre módulos do projeto. O código citado para o registro de processos de simulação é um construtor especial da classe módulo base do SystemC e é discutido na

subseção 5.4.1.

No fim da elaboração, a função `sc_start()` inicia o estágio de **simulação**. Nesse estágio o código que representa o comportamento do modelo é executado e o escalonador do *kernel* de simulação fica responsável pela execução dos processos.

Por fim, depois do retorno de `sc_start()`, o estágio de **pós-processamento** é iniciado. Essa parte, como já mencionado, é opcional. Durante o pós-processamento são lidos dados criados durante a simulação e então verifica-se a concordância dos resultados com o esperado.

5.4.1 `SC_MODULE`: a entidade básica de um projeto SystemC

Assim como em HDLs convencionais, como VHDL, o SystemC possui uma unidade básica para seus projetos. Segundo Black et al. (2009) um módulo SystemC é o menor contêiner de funcionalidade com estado, comportamento e estrutura para conectividade hierárquica. Um módulo SystemC nada mais é do que uma classe que herda da classe base `sc_module`. A Figura 13 demonstra isso.

Figura 13 – Estrutura básica de um módulo SystemC sem o uso da macro `SC_MODULE`

```
#include <systemc.h>

class module_name: public sc_module
{
  public:
    // compo do módulo
};
```

Apesar da Figura 13 deixar explícito do que se trata um módulo SystemC do ponto de vista de uma classe C++, o SystemC apresenta uma macro (equivalente a um apelido) que desempenha o mesmo papel de forma mais sintética. A Figura 14 apresenta essa forma.

Figura 14 – Estrutura básica de um módulo SystemC com o uso da macro SC_MODULE

```
#include <systemc.h>

SC_MODULE(module_name)
{
    // Corpo do módulo
};
```

Geralmente os códigos apresentados nas Figuras 13 e 14 ficam em um arquivo *header* (.h), que representa a definição do módulo. A implementação do módulo geralmente fica em um arquivo fonte separado (.cpp). Dentro da definição da classe módulo pode-se incluir vários tipos de elementos:

- Portas;
- Instâncias de canais;
- Instâncias de dados;
- Instâncias de módulos (sub-módulos);
- Construtor **(Obrigatório)**;
- Destrutor;
- Processos (métodos) de simulação;
- outros métodos.

O construtor é o único item obrigatório para constituir um módulo SystemC. A Figura 15 apresenta a sintáxe básica do construtor SystemC dentro de um módulo com um único processo (método) de simulação.

Figura 15 – Módulo com construtor e *thread*

```
//ARQUIVO: module_name.h

#include <systemc.h>

SC_MODULE(module_name)
{
    SC_CTOR(module_name)
    {
        SC_THREAD(thread_process);
    }

    void thread_process();
};
```

A macro `SC_CTOR` representa o construtor da classe módulo. Dentro do exemplo da Figura 15 também foi incluído um processo de simulação (`my_thread_process()`). Um processo de simulação nada mais é do que um método da classe módulo que é registrado dentro do construtor SystemC. No exemplo é utilizada uma *thread* (`SC_THREAD`). Processos de simulação geralmente não possuem retorno ou argumentos.

A fim de criar a estrutura básica de um modelo de simulação SystemC executável, é apresentado a seguir o restante do que seriam os arquivos que completam o projeto com base no que já foi exposto. A Figura 16 apresenta a implementação do módulo descrito na Figura 15. Em resumo é apresentada a implementação do processo de simulação `my_thread_process` .

Figura 16 – Implementação do módulo `module_name`

```
//ARQUIVO: module_name.cpp

#include "module_name.h"
#include <iostream>

void module_name::thread_process()
{
   std::cout << "thread_process executado na instância "
             << name() // nome da instancia do módulo
             << std::endl;
}
```

Por fim, a Figura 17 apresenta a função SystemC principal com a instanciação do modelo construído. O construtor padrão da classe `SC_MODULE` possue um único argumento: uma string representando o nome da instância para fins de exibição de resultados da simulação.

Figura 17 – Função principal com instanciação do módulo `module_name`

```
//ARQUIVO: main.cpp
#include <systemc.h>
#include "module_name.h"

int sc_main(int argc, char* argv[])
{
    // o construtor recebe um único argumento.
    // uma string contento o nome da instância
    module_name my_instance("my_instance");

    sc_start();

    return 0;
}
```

Assim se completa a estrutura básica de um modelo de simulação SystemC. A próxima seção discute a passagem de tempo dentro de uma simulação SystemC e como tomar vantagem de suas características.

5.5 A passagem de tempo dentro do kernel de simulação

Ao falar de simulação de sistemas eletrônicos e simulação em SystemC é importante entender alguns conceitos sobre tempo. O tempo de uma simulação pode ser visto de três formas distintas: tempo de "relógio de parede", tempo de processador e tempo de simulação.

- **Tempo de "relógio de parede"**: compreende o tempo decorrido entre o início da execução do programa e seu fim. A sua execução em computadores convencionais pode incluir tempo em que o processador esteve envolvido em outras atividades (outros programas) durante a execução do programa SystemC.
- **Tempo de processador**: represente somente o tempo gasto pelo processador executando operações referentes a simulação.
- **Tempo de simulação**: o mais importante para a avaliação do modelo, compreende o tempo interno ao *kernel* de simulação. Em outras palavras compreende o comportamento do modelo e indica características de desempenho.

O SystemC utiliza uma classe chamada `sc_time` com resolução de 64 bits para armazenar o tempo de simulação e para especificar atrasos e interrupções na simulação. Objetos desse tipo podem ser usados nas quatro operações aritméticas básicas. Também possuem métodos para sua conversão em `double`, `to_double()` (sem argumentos), e conversão para segundos em `double`, `to_seconds()` (sem argumentos). O SystemC também fornece a classe `sc_clock` para projetos que utilizem *clock*.

O SystemC utiliza uma única resolução de tempo para todos os objetos `sc_time`. Por padrão ela é de um picossegundo (`SC_PS`). A tabela 5 mostra todos as resoluções que o SystemC, por meio da enumeração (`enum`), `sc_time_unit` fornece.

Tabela 5 – Resoluções de tempo do SystemC

`enum`	Unidade	Magnitude
`SC_FS`	Femtossegundos	10^{-15}
`SC_PS`	Picossegundos	10^{-12}
`SC_NS`	Nanossegundos	10^{-9}
`SC_NS`	Microssegundos	10^{-6}
`SC_NS`	Milissegundos	10^{-3}
`SC_SEC`	Segundos	10^{0}

A resolução de tempo global do SystemC pode ser alterada por meio da seguinte função `sc_set_time_resolution()`. Essa função possui dois argumentos. O primeiro argumento deve ser um valor de ponto flutuante de precisão dupla representando uma potência de dez. O segundo argumento deve ser uma das unidades enumeradas da Tabela

5. A definição da resolução de tempo global só pode ser feita uma única vez durante a execução de um programa SystemC e deve ser feita antes da instanciação do modelo.

```
sc_set_time_resolution(double, sc_time_unit)
```

O tempo de simulação não pode ser alterado diretamente. Caso seja necessário o acesso ao tempo corrente de simulação deve-se usar a função `sc_time_stamp()` .

```
sc_time sc_time_stamp()
```

Dentro do SystemC, modelos utilizam atrasos no tempo de simulação para modelar o comportamentos do mundo real, como, por exemplo, propagação de sinais. O método `wait()` pode ser utilizado em processos de simulação `SC_THREAD` para cumprir esses objetivos de modelagem do sistema. O que ocorre quando o método `wait()` é invocado é a suspensão do processo de simulação até que o atraso especificado como parâmetro para o método tenha passado dentro do tempo de simulação - nesse tempo outros processos podem ser executados.

```
wait(double, sc_time_unit)
```

5.6 Concorrência

Nesta seção é descrita a modelagem com processos concorrentes. O que ocorre é a sincronização de processos dentro de um mesmo instante de tempo da simulação (mas com tempos diferentes de execução no processador) a fim de garantir o comportamento esperado do modelo. Para entender esse conceito de concorrência é importante relembrar a estrutura básica do fluxo de execução de uma simulação dentro do *kernel* do SystemC (Figura 10).

A simulação de processos concorrentes é feito dentro do *kernel* de simulação, que segue o modelo de multitarefa cooperativo orientado a eventos. De forma mais objetiva, os processos tomam o controle da simulação enquanto estão em execução (um por vez) e ficam responsáveis por passar este controle para o *kernel* após executar suas tarefas pretendidas dentro do instante de tempo de simulação que lhes é necessário. Processos entram em execução a partir da ocorrência de eventos aos quais eles são sensíveis.

Processos de simulação SystemC nada mais são do que métodos C++ com características peculiares. Como já mencionado, existem dois tipos. `SC_THREAD` e `SC_METHOD` . O primeiro já foi apresentado e o segundo é discutido mais adiante nesta seção.

Processos de simulação SystemC têm a característica comum de serem métodos sem retorno e sem lista de argumentos. Contudo, para que se comportem como processos

de simulação é necessário que eles sejam registrados dentro do construtor da classe módulo em questão. Já foi apresentando como registrar um processo `SC_THREAD` na Figura 15.

```
SC_THREAD(member_function)
```

Como dito, a concorrência é modelada a partir da passagem do controle da simulação por um processo em execução para o *kernel* a fim de que ele delegue a execução a outros processos. Existem duas formas de um processo fazer a liberação do controle: finalizando sua execução (`return`) ou entrando em espera (permitido apenas à `SC_THREAD`). A primeira forma é pouco utilizada para processos de simulação `SC_THREAD` , pois uma vez que o processo é finalizado ele nunca mais volta à execução dentro na simulação vigente.

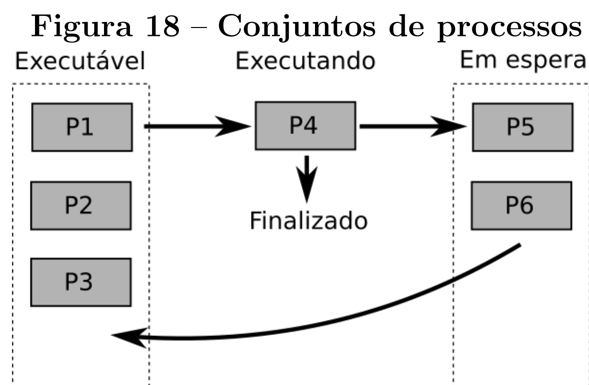
O processo de passagem de controle com a função `wait()` suspende o processo temporariamente (seu estado é salvo), outros processos são executados e, posteriormente, o processo tem sua execução resumida. Mais a frente, na seção 5.6.2, é discutido os argumentos que uma função `wait()` pode receber e seu comportamento em cada um desses arranjos.

Geralmente um processo de simulação *thread* tem sua execução inicial no início da execução da simulação e continua neste estado, por meio de um loop infinito, até o fim da simulação.

O estágio de elaboração (Figura 10) é também onde o registro dos processos de simulação é feito. No momento da chamada da função `sc_start()` a elaboração é concluída e o *kernel* de simulação é invocado, e então se começa a fase de inicialização.

Na fase de inicialização, o *kernel* identifica os processos de simulação em um de dois conjuntos: processos executáveis e processos em espera (Figura 18).

Figura 18 – Conjuntos de processos



O primeiro conjunto compreende os processos prontos para execução no tempo de simulação atual. O outro conjunto contempla processos que aguardam algum evento de tempo (instante de tempo) ou outro tipo de evento para entrar no estado executável.

Após todos os processos do primeiro conjunto terem sido executados, a fase de avançar tempo ocorre e o tempo de simulação é movido para o instante mais próximo em

que há um evento agendado. Em seguida os processos sensíveis a esse evento são postos no conjunto de processos executáveis. Normalmente essa dinâmica se repete até que todos os processos sejam executados e então a fase de pós-processamento é iniciada. Outras duas formas de finalizar a simulação é quando algum processo executa a função `sc_stop` ou quando a variável global de tempo (64 bits) tem seu valor máximo extrapolado.

5.6.1 Disparando eventos

O SystemC fornece a classe `sc_event` e seu método `notify()` como forma explícita de causar eventos.

Figura 19 – Sintaxe de `sc_event` e `sc_event::notify`

```
sc_event event_name;

event_name.notify(); // Imediato
event_name.notify(SC_ZERO_TIME); // Com atraso
event_name.notify(sc_time); // Temporizado (tempo > 0)
event_name.notify(double, units); // igual anterior
```

Como sugerem os comentários de código da Figura 19, a primeira forma do método `notify()` causa o evento de forma imediata a sua execução. A terceira e quarta forma agendam o disparo da notificação do evento assim que o tempo de simulação alcançar o tempo futuro especificado. A segunda forma é um pouco mais sofisticada e nela a notificação só acontece depois de todos os processos no estado de execução terem entrado no estado de espera no instante atual. Esse mecanismo resolve o problema de uma notificação ser executada antes do processo que deveria estar aguardando executar o comando `wait()` para o evento. Isso pode ocorrer pois, como já dito, a escolha da ordem dos processos a serem executados em um determinado instante de tempo não é especificada pela documentação (fica sob responsabilidade da implementação do simulador).

Caso um mesmo evento tenha notificação agendada para diferentes tempos de simulação prevalece o agendamento mais recente. Na Figura 20 esse caso é ilustrado.

Figura 20 – Agendamento de notificação do mesmo evento para tempos diferentes: apenas o segundo agendamento prevalece

```
sc_event event_name;

event_name.notify(10, SC_NS); // Agendado para 10 ns
event_name.notify(5, SC_NS); // Re-agendado para 5 ns
event_name.notify(15, SC_NS); // ignorado
```

Eventos também podem ser cancelados como demonstrado a seguir.

```
event_name.cancel()
```

Vale lembrar que um evento imediato não pode ser cancelado, pois sua notificação é feita no instante de simulação imediato, e cancelamentos só ocorrem em instante posterior.

5.6.2 Capturando eventos de processos *thread*

Como apresentado, processos de simulação do tipo *thread* entram em modo de espera por meio de declarações `wait()`. Além de ser possível a espera por eventos explícitos de tempo, também é possível a espera por eventos do tipo `sc_event`. Eventos nada mais são do que acontecimentos em um instante de tempo, não possuindo valor ou duração. A Figura 21 apresenta as principais formas possíveis de espera por um evento.

Figura 21 – Principais formas de uso da função `wait()` para observar eventos

```
wait(time);
wait(double, time_unit);
wait(event);
wait(event1 | event2 | ... eventn);
wait(event1 & event2 & ... eventn);
wait(time, event);
wait(time, event1 | event2 | ... eventn);
wait(time, event1 & event2 & ... eventn);
```

A primeira e a segunda forma, como já apresentado neste trabalho, são equivalentes. A terceira forma aguarda a ocorrência de um único evento. A quarta é um exemplo de espera por um de vários eventos e nela qualquer dos n eventos é válido para mudar o status do processo de em espera para executável. A quinta forma aguarda a ocorrência

de n eventos. A sexta e sétima forma são semelhantes a quarta e quinta forma, respectivamente, com a diferença de aguardar por um (sexta forma) ou todos os eventos (sétima forma), ou o tempo especificado para mudar o status do processo. Vale ressaltar que nas situações em que há mais de um evento assistido pela função `wait()` não é possível identificar qual dos eventos ocorreu primeiro.

Em um processo em que funções `wait()` são chamadas com a passagem de eventos como argumentos, diz-se que o processo é sensível a tais eventos. Em outras palavras, a ocorrência dos eventos especificados em `wait()` acarreta na execução do processo. Ao conjunto de eventos ao qual um processo é sensível dá-se o nome lista de sensitividade.

5.6.3 Processos de simulação método do SystemC

O processo de simulação `SC_METHOD` é semelhante a um método comum de classes C++. Uma das diferenças é que processos `SC_METHOD` não possuem valor de retorno e lista de argumentos. O registro de um `SC_METHOD` também é feito junto ao *kernel* dentro do construtor da classe módulo. A seguir a sintaxe de registro é apresentada.

```
SC_METHOD(process_name);
```

Diferente de `SC_THREAD` , não é permitida a suspensão da execução de `SC_METHOD` . Sempre que um `SC_METHOD` inicia sua execução ele é finalizada no mesmo instante de tempo de simulação. Por outro lado, um processo `SC_METHOD` pode ser chamado pelo *kernel* mais de uma vez dentro de uma simulação. Isso é feito por meio de uma lista de eventos ao qual o processo método é sensível (assim como ocorre com `SC_THREAD`).

5.6.4 Capturando eventos de processos método

Uma forma de especificar sensitividade para um `SC_METHOD` a eventos é feita por meio do método `next_trigger()` e ocorre a cada execução do próprio processo. A sintaxe e o comportamento de `next_trigger()` é semelhante ao do método `wait()` (Figura 22).

Figura 22 – Principais formas de uso da função `next_trigger()` para observar eventos

```
next_trigger(time);
next_trigger(double, time_unit);
next_trigger(event);
next_trigger(event1 | event2);
next_trigger(event1 & event2);
next_trigger(time, event);
next_trigger(time, event1 | event2);
next_trigger(time, event1 & event2);
```

É importante a cada execução de um `SC_METHOD` executar um `next_trigger()` , pois só assim o processo método será executado no futuro. Como veremos mais adiante é comum especificar pelo menos a sensitividade a um evento de forma estática (no registro do processo).

5.6.4.1 Sensitividade estática para Processos

Até agora só foi discutida a sensitividade dinâmica, tanto para processos método quanto para processos *thread*. A sensitividade estática é definida para cada processo imediatamente após o seu registro no construtor junto ao *kernel* de simulação. A Figura 23 mostra um exemplo com `SC_THREAD` e `SC_METHOD` .

Figura 23 – Exemplo de sensitividade estática

```
SC_MODULE(module_name)
{
        sc_event event1, event2, event3;

        SC_CTOR(module_name)
        {
                // forma de função
                SC_THREAD(thread_process);
                        sensitive(event1);

                // forma de stream
                SC_METHOD(method_process)
                        sensitive << event2 << event3;
        }

        void thread_process();
        void method_process();
```

Para `SC_THREAD` o uso de sensitividade estática é sobrescrito sempre que a função `wait()` , em sua forma com argumentos, é executada dentro do processo *thread*. Para definir o uso da sensitivade estática é necessário o uso da função `wait()` sem argumentos.

Já para `SC_METHOD` a lista de sensitividade estática é a usada por padrão, porém ela é sobrescrita quando um `next_trigger()` é executado na chamada do processo método. Para reestabelecer o uso da lista estática basta executar o método `wait()` sem argumentos dentro do processo método.

6 CONSIDERAÇÕES FINAIS

Este trabalho apresentou uma discussão sobre metologias para projeto de sistemas eletrônicos, o porquê do uso da TLM e da ESL e os motivos do SystemC ser uma ferramenta adequada para suportar projetos que as utilizem.

No Capítulo 2 foram descritas três abordagens (ou escolas) possíveis: projeto assistido por computador, abordagem guiada por base de conhecimento para tomada de decisões de projeto e síntese a nível de sistema eletrônico. Esta última é a mais utilizada atualmente para tratar projetos de sistemas complexos. Pois, a partir da descrição de alto nível do sistema por meio de refinamento se gera a implementação do sistema, o que torna possível explorar várias soluções a partir da mesma descrição.

Ainda no Capítulo 2 foi descrita a metodologia de diagrama Y, que é uma metodologia com três domínios de especificação: comportamental, estrutural e físico. A metodologia de telhado duplo foi apresentada como uma generalização do diagrama Y, tendo ela apenas os domínios comportamental e estrutural no modelo apresentado no trabalho, que são suficientes para o entendimento da problemática de síntese.

No Capítulo 3 quatro modelos TLM foram apresentados e discutidos. Já No Capítulo 3, a metodologia TLM foi discutida como uma metodologia complementar às metodologias ESL que possibilita, por meio de interfaces de comunicação, integrar componentes de diferentes granularidades. O que resulta na criação de uma plataforma virtual do sistema que permite valoração mais cedo do modelo de hardware e o desenvolvimento do projeto de software, que usualmente é o fim do ciclo de vida do projeto.

Também foi discutido o uso do SystemC como ferramenta para apoiar projetos que utilizam as metodologias ESL e TLM, o que levou à conclusão de que ela é uma ferramenta que supre as necessidades das duas metodologias. E por fim, no Capítulo 5 um estudo introdutório do SystemC foi apresentado, demonstrado suas principais características.

O objetivo desse trabalho foi compor um referencial teórico sobre as principais metodologias de projeto de hardware. Logo, espera-se que ele possa servir como material de estudos para futuras pesquisas em áreas como, arquitetura de computadores, CPLDs, co-design em FPGA, ou mesmo em projetos ASIC.

REFERÊNCIAS

BANERJEE, A.; SUR, B. **SystemC and SystemC-AMS in practice: SystemC 2.3, 2.2 and SystemC-AMS 1.0**. [S.l.: s.n.], 2014. 1-460 p.

BLACK, D. C. et al. **SystemC: From the Ground Up, Second Edition**. 2nd. ed. [S.l.]: Springer Publishing Company, Incorporated, 2009. ISBN 0387699570, 9780387699578.

CAI, L.; GAJSKI, D. Transaction level modeling: An overview. In: **Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis**. New York, NY, USA: ACM, 2003. (CODES+ISSS '03), p. 19–24. ISBN 1-58113-742-7. Disponível em: <http://doi.acm.org/10.1145/944645.944651>.

CAI, L.; VERMA, S.; GAJSKI, D. Comparison of specc and systemc languages for system design. **Technical Report CECS-03-11**, 07 2003.

CHIEN, A. A.; KARAMCHETI, V. Moore's law: The first ending and a new beginning. **Computer**, v. 46, n. 12, p. 48–53, Dec 2013. ISSN 0018-9162.

GAJSKI, D. D. et al. **High-level Synthesis: Introduction to Chip and System Design**. Norwell, MA, USA: Kluwer Academic Publishers, 1992. ISBN 0-7923-9194-2.

GAJSKI, D. D.; KUHN, R. H. Guest editors' introduction: New vlsi tools. **Computer**, v. 16, n. 12, p. 11–14, Dec 1983. ISSN 0018-9162.

GEILEN, M. et al. An algebra of pareto points. **Fundam. Inf.**, IOS Press, Amsterdam, The Netherlands, The Netherlands, v. 78, n. 1, p. 35–74, jan. 2007. ISSN 0169-2968. Disponível em: <http://dl.acm.org/citation.cfm?id=1366007.1366010>.

GERSTLAUER, A. et al. Electronic system-level synthesis methodologies. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 28, n. 10, p. 1517–1530, Oct 2009. ISSN 0278-0070.

INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. Ieee standard for standard systemc language reference manual. **IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)**, p. 1–638, Jan 2012.

KRISHNAN, V.; TORRELLAS, J. A direct-execution framework for fast and accurate simulation of superscalar processors. In: **Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.98EX192)**. [S.l.: s.n.], 1998. p. 286–293. ISSN 1089-795X.

MEEUS, W. et al. An overview of today's high-level synthesis tools. **Design Automation for Embedded Systems**, v. 16, n. 3, p. 31–51, Sep 2012. ISSN 1572-8080. Disponível em: <https://doi.org/10.1007/s10617-012-9096-8>.

MOORE, G. E. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. **IEEE Solid-State Circuits Society Newsletter**, v. 11, n. 5, p. 33–35, Sept 2006. ISSN 1098-4232.

TEICH, J. **Embedded System Synthesis and Optimization**. 2000.