**UNIVERSIDADE DE BRASÍLIA**
**FACULDADE DE TECNOLOGIA**
**DEPARTAMENTO DE ENGENHARIA ELÉTRICA**

# MODELAGEM EM NÍVEL TRANSACIONAL DE SISTEMAS EM CHIP MISTOS PARA APLICAÇÕES DE REDES DE SENSORES SEM FIO

**GILMAR SILVA BESERRA**

**ORIENTADOR: JOSÉ CAMARGO DA COSTA**

**TESE DE DOUTORADO EM ENGENHARIA ELÉTRICA**

**PUBLICAÇÃO: PPGENE.TD - 054/10**

**BRASÍLIA/DF: OUTUBRO – 2010**

**FICHA CATALOGRÁFICA**


| | |
|---|---|
| BESERRA, GILMAR SILVA | |
| Modelagem em Nível Transacional de Sistemas em Chip Mistos para Aplicações de Redes de Sensores sem Fio [Distrito Federal] 2010. | |
| xvii, 134p., 210 x 297 mm (ENE/FT/UnB, Doutor, Tese de Doutorado) – Universidade de | |
| Brasília. Faculdade de Tecnologia. | |
| Departamento de Engenharia Elétrica | |
| 1.Sistemas em Chip Mistos | 2.Modelagem |
| 3.Redes de Sensores sem Fio | 4.SystemC |
| I. ENE/FT/UnB | II. Título (série) |


**REFERÊNCIA BIBLIOGRÁFICA**


BESERRA, G. S. (2010). Modelagem em Nível Transacional de Sistemas em Chip Mistos para Aplicações de Redes de Sensores sem Fio. Tese de Doutorado em Engenharia Elétrica, Publicação PPGENE.TD - 054/10, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 134p.


## *CESSÃO DE DIREITOS*

AUTOR: Gilmar Silva Beserra.

TÍTULO: Modelagem em Nível Transacional de Sistemas em Chip Mistos para Aplicações de Redes de Sensores sem Fio.

GRAU: Doutor ANO: 2010



_____

**Dedicatória**

*À memória de meu pai, Gilvan Alves Beserra, que continua vivo no coração de todos os que tiveram a bênção de sua convivência.*

*Gilmar Silva Beserra*

## Agradecimentos

*A Deus, pela vida, pela força, pelos dons de amar, aprender e criar.*
*Ao meu pai Gilvan (em memória), pelas lições ensinadas nos últimos anos e por sempre ter me incentivado a aperfeiçoar os meus conhecimentos e a buscar novas conquistas, e à minha mãe Maria, pelo exemplo de vida e por ter me ensinado a ter sempre bom ânimo, saber perdoar e ter paciência, generosidade, modéstia e esperança. À minha tia Ires, cuja ajuda e hospitalidade me tornaram possível iniciar os estudos na universidade. Aos meus irmãos Liandra, Éder Gillian, Iuri e Nathália, e aos demais familiares, pelo apoio incondicional.*
*Ao meu orientador José Camargo da Costa, pela confiança, compreensão, incentivo, pelos exemplos de dedicação e competência e por ter aceitado me orientar no doutorado. Seu apoio e suas lições foram muito além do ambiente acadêmico e serão sempre lembrados.*
*Aos grandes amigos e colegas Edil e Heider, que muito admiro e com quem sempre posso contar a qualquer hora. Sua ajuda foi fundamental, principalmente nas etapas mais difíceis para a conclusão do trabalho e desta tese.*
*Ao amigo e colega Eusse, com quem muito aprendi e cujo trabalho, ajuda e sugestões serviram como base para os resultados aqui apresentados. A todos os colegas que contribuíram com o projeto, em especial Daniel Café, João Lucas, Arthur Sampaio, João Vitor Pimentel, Charles Costa, Matheus Belin, Leticia Maia e Jorge Carvalho.*
*Aos professores que, de uma forma ou de outra, apresentaram sugestões, idéias e direcionamentos: Ricardo P. Jacobi, Nahri Moreano, Linnyer B. Ruiz, Carlos H. Llaños, Sandro Haddad, Adson F. Rocha, Janaína G. Guimarães e, em especial, Fernando G. Moraes, cujas observações realizadas na qualificação foram de grande valia na continuidade e finalização do trabalho.*
*A todos os amigos e colegas do LDCI, em especial ao Genival Araújo e Fabricio Schlag, pelo ambiente de amizade. À minha prima Flávia Dourado, pela ajuda prestada durante seu trabalho na secretaria do LDCI. Aos colegas da INBD, especialmente ao Eduardo Borges, pela experiência na empresa e pela compreensão das minhas limitações de tempo durante a elaboração desta tese.*
*Aos amigos e colegas de doutorado Alberto López e Suélia Rodrigues, pelo constante incentivo e pelos exemplos de determinação e disciplina.*
*Às grande amigas Neda, Régia e Leticia, pelo grande coração, pela amizade, incentivo, carinho e apoio nos momentos de dúvida.*
*Ao CNPq e ao inct NAMITEC, pelo apoio financeiro.*

*Gilmar Silva Beserra*

# RESUMO

## MODELAGEM EM NÍVEL TRANSACIONAL DE SISTEMAS EM CHIP MISTOS PARA APLICAÇÕES DE REDES DE SENSORES SEM FIO.

**Autor: Gilmar Silva Beserra**

**Orientador: José Camargo da Costa**

**Programa de Pós-graduação em Engenharia Elétrica**

**Brasília, outubro de 2010**

Este trabalho apresenta a modelagem em nível de sistema de SoCs mistos que estão sendo desenvolvidos nesta instituição, voltados inicialmente para aplicações envolvendo redes de sensores sem fio (RSSF). A abordagem utilizada na modelagem combina o uso de SystemC-AMS para descrever blocos analógicos com o uso da biblioteca SCNSL (*SystemC Network Simulation Library*) para permitir a comunicação entre nós constituídos pelos SoCs e nós funcionais em um ambiente de RSSF.

Os modelos desenvolvidos foram integrados em duas plataformas virtuais. A primeira consiste em um SoC que possui um processador RISC de 16 bits, controlador de interrupções, memória e interfaces digital, analógica e de RF. A segunda possui um processador RISC de 32 bits com um conjunto de instruções baseado no do MIPS e com controle de interrupções implementado, barramento, memória, *timer*, sensor de imagem APS, módulo AES, ADC, interface de RF e blocos reconfiguráveis. Foram realizadas simulações de aplicações que permitiram verificar o fluxo de dados entre os módulos e o correto funcionamento das plataformas virtuais.

Sendo assim, foi efetuada uma melhoria no fluxo de projeto utilizado anteriormente pela nossa equipe, ao se permitir o desenvolvimento e teste de *software* embarcado em estágios iniciais do projeto, bem como foi gerada uma biblioteca de modelos (*LDCI_Modeling_Library*) que podem ser usados na implementação de outras plataformas virtuais.

# ABSTRACT

## TRANSACTION LEVEL MODELING OF MIXED-SIGNAL SYSTEMS ON CHIP FOR WIRELESS SENSOR NETWORKS APPLICATIONS.

**Author: Gilmar Silva Beserra**

**Supervisor: José Camargo da Costa**

**Programa de Pós-graduação em Engenharia Elétrica**

**Brasília, october 2010**

This work presents the system-level modeling of mixed-signal SoCs that are currently being developed at this institution, aiming initially at Wireless Sensor Networks (WSN) applications. The approach presented here combines the use of SystemC-AMS to describe analog blocks and SCNSL (SystemC Network Simulation Library) to allow the communication among nodes composed by SoCs and functional level nodes in a WSN environment.

The developed models were integrated into two virtual platforms. The first one consists of a 16-bit RISC processor, interrupt controller, memory and digital, analog and RF interfaces. The second one has a 32-bit RISC processor with a MIPS-based instruction set and an implemented interrupt controller, bus, memory, timer, APS image sensor, AES module, ADC, RF interface and reconfigurable blocks. Both virtual platforms were simulated by running applications that allowed to verify the dataflow among the modules and check their correctness.

Therefore, the design flow previously used by our team was improved by allowing the development and test of embedded software at early stages. A library with models (*LDCI_Modeling_Library*) was also generated and can be used to implement other virtual platforms.

# SUMÁRIO

# LISTA DE FIGURAS

# LISTA DE TABELAS

# LISTA DE SÍMBOLOS

### Siglas

| | |
|---|---|
| ADC | Analog-to-Digital Converter |
| ADL | Architecture Description Language |
| AMBA | Advanced Microcontroller Bus Architecture |
| AMS | Analog/Mixed Signal |
| API | Application Programming Interface |
| APS | Active Pixel Sensor |
| ASCII | American Standard Code for Information Interchange |
| ASIC | Application-Specific Integrated Circuit |
| ASK | Amplitude-Shift Keying |
| BC | Border Channels |
| BP | Border Processes |
| CMOS | Complementary-symmetry Metal-Oxide Semiconductor |
| CPU | Central Processing Unit |
| CT | Continuous-Time |
| DCT | Discrete Cosine Transform |
| DE | Discrete-Event |
| DI | Domain Interfaces |
| DSP | Digital Signal Processing |
| ELN | Electrical Linear Networks |
| EOC | End of Conversion |
| FIFO | First In, First Out |
| ForSyDe | Formal System Design |
| FPGA | Field Programmable Gate Array |
| FFT | Fast Fourier Transform |
| FSK | Frequency-Shift Keying |
| FU | Functional Unit |
| GCC | GNU C Compiler |
| HetMoC | Heterogeneous Modeling in SystemC |
| HetSC | Heterogeneous Embedded Systems in SystemC |
| IFFT | Inverse Fast Fourier Transform |
| IP | Intellectual Property |
| ISM | Industrial, Scientific and Medical |

| | |
|---|---|
| ISS | Instruction Set Simulator |
| JPEG | Joint Photographic Experts Group |
| LSB | Less Significant Bit |
| LSF | Linear Signal Flow |
| MB | MegaByte |
| MEMS | MicroElectroMechanical Systems |
| MIPS | Microprocessor without Interlocked Pipeline Stages |
| MISO | Multiple Inputs, Single Output |
| MoC | Model of Computation |
| NS-2 | Network Simulator |
| PC | Program Counter |
| PE | Processing Elements |
| PGM | Portable GrayMap |
| PV | Programmer View |
| RAM | Random-Access Memory |
| RISC | Reduced Instruction Set Computer |
| RF | Radio-Frequência |
| ROM | Read-Only Memory |
| RoSA | Reconfigurable Stream-based Architecture |
| RSoC | Reconfigurable System on Chip |
| RSSF | Redes de Sensores Sem Fio |
| RTL | Register-Transfer Level |
| SCNSL | SystemC Network Simulation Library |
| SDF | Synchronous DataFlow |
| SR | Synchronous/Reactive |
| SRAM | Static Random-Access Memory |
| SY | Synchronous |
| SoC | System on Chip |
| TDF | Timed Data Flow |
| TLM | Transaction-Level Modeling |
| UART | Universal Asynchronous Receiver/Transmitter |
| UFRJ | Universidade Federal do Rio de Janeiro |
| ULA | Unidade Lógico-Aritmética |
| UML | Unified Modeling Language |
| VHDL | Very High Speed Integrated Circuit Hardware Description Language |
| VLSI | Very Large Scale Integration |
| WSN | Wireless Sensor Network |
| WSP | Wire-Speed Processor |

# Capítulo 1

# Introdução

*Este capítulo apresenta a contextualização do trabalho realizado no doutorado. É dado um panorama geral do estado da arte e, em seguida, são apresentados os objetivos da tese, as principais contribuições e a organização do documento.*

## 1.1 Contextualização

A abstração é uma técnica poderosa para o projeto e implementação de sistemas complexos. Abstrair determinados detalhes que se mostram desnecessários em um certo momento permite lidar com a complexidade e tratar tais detalhes mais tarde. Diferentes quantidades de detalhes correspondem a diferentes níveis de abstração [3, 17].

Sistemas em *Chip* (SoC) têm sido cada vez mais explorados em produtos eletrônicos complexos, tais como telefones e câmeras digitais, devido a características como tamanho reduzido e bom desempenho. Um SoC típico integra muitos blocos, incluindo processadores, barramento, memórias e outros periféricos. Um fator adicional que aumenta a complexidade de tais sistemas é a necessidade de executar aplicações que oferecem múltiplos recursos, geralmente associados à comunicação sem fio e ao processamento de sinais.

Apesar dos SoCs explorarem o projeto em um único *chip* e eliminarem o uso de pinos externos para barramentos, o acesso ao *software* continua seguindo o mesmo padrão, ou seja, como se os componentes estivessem *off-chip*. Por isso, novas arquiteturas ainda mais complexas estão sendo apresentadas, como é o caso do *wire-speed processor* (WSP), que tem uma arquitetura heterogênea e integra múltiplos núcleos genéricos com aceleração específica ao domínio e funções de entrada e saída em um SoC. Com isso, é possível ter ganhos de desempenho com o uso de aceleradores e reduzir os custos de desenvolvimento de *software* através de um esquema de endereçamento uniforme da memória [18].

O aumento da complexidade dos sistemas, associado à necessidade de reduzir o tempo e os custos de produção, tem tornado o fluxo de projeto tradicional, no qual espera-se muito tempo para se ter um protótipo, cada vez mais defasado. Sendo assim, aumentar o nível de abstração vem se tornando uma solução para esses problemas [3, 12].

Alguns ambientes e linguagens de descrição permitem a modelagem em alto nível de sistemas. Um dos trabalhos pioneiros nessa área é o PtolemyII [19], que é um ambiente implementado em Java baseado na semântica abstrata de módulos de computação, chamados de atores. Com isso, vários Modelos de Computação (MoC) podem ser combinados, tais como *Discrete-Event* (DE) e *Continuous-Time* (CT). Outra abordagem é a *Unified Modeling Language* (UML), que é uma linguagem padrão na modelagem de sistemas de *software*, e que deu origem ao SysML, uma linguagem de propósito geral que permite a especificação, análise, projeto, verificação e validação de uma ampla faixa de sistemas que incluem *hardware*, *software*, processos e outros componentes [20].

Dentre as abordagens, destaca-se o uso de SystemC [21], uma biblioteca do C++ que permite a descrição de sistemas em diferentes níveis de abstração. Um deles é o TLM (*Transaction-Level Modeling*), que permite que módulos se comuniquem utilizando transações, uma estrutura de dados (objeto C++) que utiliza chamadas de funções, permitindo aumentar a velocidade da simulação.

SoCs possuem características interessantes para uso em redes de sensores sem fio (RSSF), tais como tamanho reduzido e baixo consumo de potência. Além disso, podem conter em um único *chip* módulos analógicos, de radio-frequência (RF) e digitais. Esta integração, contudo, é uma tarefa complexa, sobretudo quando se tenta otimizar o sistema para uma determinada aplicação.

O núcleo nativo do SystemC não permite a descrição de sistemas analógicos e contínuos no tempo. Uma possível solução é realizar co-simulações de sistemas mistos utilizando SystemC para os blocos digitais e linguagens como VHDL-AMS [22] ou Verilog-AMS [23] para os blocos analógicos e de RF. Entretanto, existem desvantagens nessa abordagem, dentre as quais uma está relacionada à performance da simulação [24]. Por isso, extensões SystemC-AMS [25] estão sendo propostas para tornar possível a realização de ambientes para a modelagem funcional, exploração de arquiteturas, integração, validação e prototipagem de sistemas embarcados mistos [26].

Como os SoCs descritos neste trabalho estão previstos para funcionar em aplicações de RSSF, surgiu a necessidade de validar os mesmos em tais ambientes. Existem várias ferramentas de simulação de rede disponíveis, tais como o *Network Simulator* (NS-2) [27] e Opnet [28]. Tais ferramentas oferecem limitações no projeto de sistemas, visto que não modelam a concorrência dentro do nó e, portanto, não é possível avaliar como a arquitetura do sistema pode afetar o comportamento da rede e vice-versa [29].

## 1.2 Definição do Problema

Os protótipos de SoC que já foram realizados em nossa instituição seguiram o fluxo de projeto tradicional e algumas dificuldades surgiram em função dessa abordagem. A análise de trabalhos anteriores demonstrou que uma limitação citada com bastante frequência é a dificuldade em se desenvolver e testar o *software* embarcado para os SoCs, já que não havia ambientes de desenvolvimentos disponíveis antes de se ter o *hardware* pronto, conforme é mostrado em [30, 31]. A realização de um primeiro modelo em SystemC RTL (*Register Transfer Level*) foi feita em [32] como uma primeira tentativa de se proporcionar uma plataforma para executar aplicações para o SoC. Devido às limitações encontradas na modelagem e a problemas de desempenho nas simulações

- dado o nível de abstração e a quantidade de detalhes da arquitetura utilizados no modelo - o autor sugeriu a elaboração de uma nova plataforma utilizando SystemC TLM.

Dessa forma, surgiu a necessidade de se ter plataformas criadas a partir de modelos em alto nível de abstração, ou seja, desprezando detalhes "desnecessários"da arquitetura e por isso permitindo um maior desempenho nas simulações. Tais plataformas também devem permitir fazer explorações preliminares de arquiteturas, desenvolver e testar aplicações antes de se ter o *hardware* disponível, de maneira a auxiliar as decisões de projeto tendo em consideração o *hardware* escolhido e o impacto do *software* sobre o mesmo. Como tais plataformas são apenas modelos, e não placas de *hardware*, este conceito é chamado de **plataforma virtual** (vide [3]) e será adotado ao longo do texto.

## 1.3 Objetivos do Projeto

- Desenvolvimento de uma biblioteca de modelos incluindo blocos digitais, reconfiguráveis e analógicos em alto nível de abstração, usando as linguagens de descrição de *hardware* SystemC e SystemC-AMS.
- Realização de uma avaliação preliminar da funcionalidade do *hardware* dos SoCs e do *software* embarcado a partir da integração desses modelos em duas plataformas virtuais, sendo uma de um SoC e outra de um RSoC. Como ambos os SoCs envolvem módulos analógicos, as plataformas virtuais devem permitir a interação de ao menos dois MoCs diferentes.
- Realização de simulações envolvendo aplicações de referência, com resultados previamente conhecidos, para fins de verificação do correto funcionamento das plataformas virtuais.

## 1.4 Principais Contribuições

Este trabalho apresenta a implementação de plataformas virtuais de um SoC e de um RSoC cujas arquiteturas diferem em alguns aspectos das conhecidas na literatura. O primeiro consiste em um processador RISC de 16 bits, memória, controlador de interrupções e interfaces. O segundo é um sistema mais complexo, no qual, além de um processador mais elaborado (RISC de 32 bits) e um barramento, inclui blocos reconfiguráveis e sensor de imagem, o que permite um maior número de aplicações. As plataformas virtuais originadas neste trabalho aumentam a flexibilidade do projetistas de *hardware* e de *software* ao oferecerem a possibilidade de desenvolvimento e teste de aplicações que, de outra forma, somente seriam possíveis em um estágio já adiantado do fluxo, no qual mudanças teriam um custo proibitivo.

Além disso, é apresentada uma metodologia de modelagem que envolve uma combinação de ferramentas diferente das que foram encontradas na literatura durante a elaboração desta tese. A abordagem utilizada aqui inclui a integração de blocos analógicos em SystemC-AMS e, sendo assim, envolveu a integração de dois MoCs diferentes. Este trabalho também tornou possível estender os conceitos apresentados para simular ambientes de rede nos quais o SoC e/ou o RSoC podem funcionar como nós, interagindo entre si ou com nós funcionais, permitindo o teste de algoritmos

que também possuem impacto na arquitetura do sistema, e vice-versa. Essa metodologia pode também ser aplicada a outros tipos de sistema.

Por fim, a plataforma virtual RSoC permite explorar a reconfigurabilidade do sistema para otimização de algoritmos relacionados às aplicações desejadas, como por exemplo, algoritmos de processamento de imagens, no caso de aplicações que utilizem o sensor de imagem do RSoC.

## 1.5 Apresentação do Manuscrito

O documento está organizado da seguinte maneira: no Capítulo 2, são abordados os principais conceitos de modelagem em alto nível de sistemas, bem como das linguagens utilizadas, enquanto no capítulo 3 são apresentados alguns trabalhos correlatos; o Capítulo 4 detalha a metodologia adotada no desenvolvimento do trabalho; o Capítulo 5 contém a descrição das arquiteturas dos SoCs e o Capítulo 6 indica como foi feita a modelagem de ambas as plataformas virtuais. As simulações são apresentadas no Capítulo 7 e a discussão, no Capítulo 8. Finalmente, no Capítulo 9, são mostradas as conclusões e algumas considerações para a continuidade do trabalho. Os anexos contêm material complementar acerca da instalação das ferramentas utilizadas e da utilização das plataformas virtuais. Um CD com os arquivos gerados neste trabalho também foi incluído como anexo.

# Capítulo 2

# Modelagem de Sistemas

*Este capítulo tem por objetivo propiciar uma visão geral da modelagem como estratégia para a descrição de SoCs, começando com a definição de modelos de computação e níveis de abstração. Em seguida, é feito um resumo das principais características da linguagem de descrição escolhida para a elaboração deste trabalho e das extensões que foram disponibilizadas ao longo do tempo para ampliar a utilização da mesma, adicionando a possibilidade de integração de blocos analógicos e simulação de RSSF. Por fim, a linguagem de descrição de arquitetura utilizada para gerar um Instruction Set Simulator (ISS) é abordada.*

## 2.1 Redes de Sensores sem Fio

Devido aos recentes avanços nas tecnologias de comunicações sem fio e sensores, o uso de redes de sensores sem fio (RSSF) tem se tornado uma boa solução em diversas aplicações, como, por exemplo, monitoramento ambiental e controle de processos industriais. Isso se deve a fatores como baixo custo e baixo consumo de potência, conforme será descrito a seguir.

Uma RSSF pode ser caracterizada pelo uso de uma grande quantidade de nós equipados com sensores, processadores embarcados e transceptores de RF, de maneira que os mesmos possuem capacidade de aquisição e processamento de sinais, e de comunicação sem fio. Esses nós podem ser colocados dentro da região de interesse ou próximos a ela [33].

Algumas características mostram como as RSSF são diferenciadas das redes de sensores tradicionais. Dentre elas, pode-se citar: maior densidade de nós; nós alimentados por baterias; grande limitação de energia, computação e armazenamento; autoconfiguração; aplicações específicas e mudanças frequentes de topologia [33].

Os sensores podem ser usados para detectar ou monitorar diferentes parâmetros ou condições físicas, tais como: luz, som, umidade, pressão, temperatura, composição do solo, qualidade da água e do ar, e atributos de um determinado objeto (tamanho, peso, posição, velocidade e direção). O

fato da comunicação ser sem fio habilita o uso das RSSF em praticamente quaisquer ambientes, o que amplia a gama de possíveis aplicações. Alguns exemplos são mostrados a seguir [33].

- Monitoramento ambiental: neste caso, os sensores são utilizados para monitorar parâmetros ou condições de um determinado *habitat*, tais como umidade, pressão, temperatura e radiação;
- Qualidade do ar e da água: os sensores podem ser posicionados no solo ou dentro da água para controle de qualidade ou de poluição;
- Monitoramento de danos e desastres: os sensores podem monitorar danos biológicos ou químicos em ambientes hostis, campos de batalha, ou ainda podem detectar danos naturais e não-naturais, como enchentes ou incêndios em florestas;
- Aplicações militares: neste caso, as RSSF podem fazer parte de sistemas de controle, comando, comunicação e inteligência;
- Aplicações de cuidados com a saúde: os sensores podem monitorar o comportamento de pacientes ou monitorar sinais vitais;
- Processos de controle industriais: os sensores podem ser colocados em linhas de montagem para controlar processos de produção ou podem ser usados para monitorar as condições de um determinado equipamento;
- Segurança: os sensores podem ser posicionados em aeroportos, prédios, metrôs e outras infraestruturas para detectar intrusos ou ataques em potencial;
- Ambientes inteligentes: RSSF podem ser usadas em casas inteligentes para controle de equipamentos e monitoramento de parâmetros do ambiente, como gás, eletricidade, água, etc.

Algumas plataformas que vêm sendo propostas ou adotadas na construção de RSSF em projetos acadêmicos ou na indústria são descritas em [34], onde é ressaltado que a tecnologia para tais projetos tende a se tornar cada vez mais acessível com a produção em larga escala de diferentes tipos de micro-sensores.

Um dos desafios no projeto de RSSF é o requisito de baixo consumo de potência [35]. Sistemas em *chip* têm se mostrado eficientes na implementação de tais sistemas por propiciarem tanto economia de energia quanto desempenho adequado, satisfazendo os critérios de processamento e de duração da bateria do nó que caracterizam as aplicações de RSSF [36].

Sendo assim, as ferramentas de simulação possuem uma importância crucial no projeto de RSSF, tanto na reprodução de seu comportamento quanto na estimativa de consumo. Em [1], um exemplo de quatro nós é apresentado para mostrar os diferentes componentes que devem ser modelados em uma RSSF em um simulador baseado na linguagem SystemC. Em primeiro lugar, assume-se que os nós podem ter níveis de detalhes diferentes, ou seja, enquanto alguns possuem modelos de *hardware* (por exemplo, periféricos, conversores, sensores, transceptores) e de *software* (por exemplo, aplicações, sistema operacional e rotina de tratamento de interrupção), outros nós podem ser apenas funcionais. Na Figura 2.1, os nós *Node1* e *Node2* são descritos como um conjunto de sub-blocos em diferentes níveis (TLM e RTL, respectivamente), enquanto o nó *Node3* é descrito apenas no nível funcional. Já o nó *Node0* possui um modelo detalhado do *hardware* e do *software*.

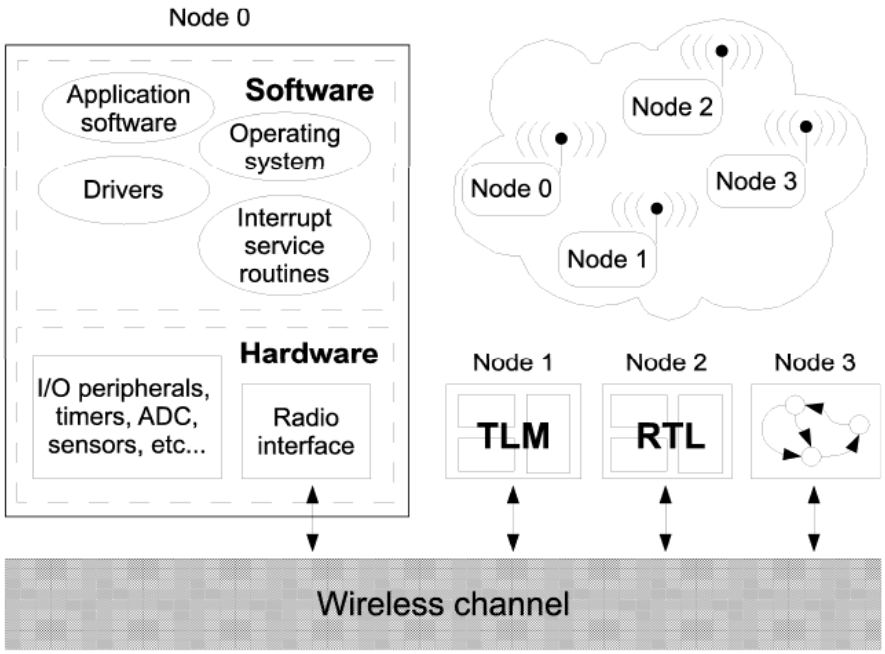


Figura 2.1: Exemplo de modelo de uma RSSF com 4 nós [1]

## 2.2 Sistemas em Chip

Com o avanço da tecnologia de fabricação VLSI (*Very Large Scale Integration*), é possível integrar bilhões de transistores em um único *chip* [18]. Sistemas em *Chip* (SoC) são caracterizados por conter uma grande variedade de componentes: lógica, memória, processadores, circuitos analógicos, etc. Seu projeto é desafiador tanto pela diversidade dos elementos quanto pela limitação de tamanho dos *chips*.

Para melhorar a produtividade, são utilizados módulos IP (*Intellectual Property*), que são componentes pré-projetados que podem ser de dois tipos: *hard* IP, que é um *layout* completo disponível, e *soft IP*, que é um módulo sintetizável modelado em uma linguagem de descrição de *hardware* como VHDL ou Verilog. Para isso, os IPs devem ser bem documentados e testados, e devem possuir uma interface padronizada que permita a sua integração ao SoC.

Processadores embarcados têm sido bastante utilizados em SoCs, já que muitas aplicações são implementadas em *software*. Além disso, muitos sistemas complexos usam sistemas operacionais para gerenciamento de arquivos e de rede [37].

Alguns aspectos importantes no projeto de SoCs são [37]:

- Projeto de sistemas mistos: um SoC interage com o mundo e, portanto, necessita de elementos que combinem processamento digital e analógico;
- Escolha do processador: há uma grande variedade de IPs, tanto *hard* quanto *soft*, disponíveis. A escolha deve ser determinada por fatores como desempenho dos programas e compatibilidade de *software* com o ambiente de desenvolvimento;

- Co-projeto *Hardware/Software*: como essas arquiteturas devem ser projetadas para atender requisitos de tempo real e baixo consumo, é necessário o uso de uma metodologia que permita o desenvolvimento simultâneo da arquitetura de *hardware* e do *software* a ser executado na mesma.

Algumas características desejáveis para SoCs são tamanho reduzido, baixo consumo de potência, alto desempenho, baixo custo e robustez. Para tanto, a tecnologia CMOS é adequada devido às suas características, dentre as quais pode-se citar: alta capacidade de miniaturização, tensão de alimentação baixa (gera menor consumo de potência), custo reduzido, e todos os componentes podem ser fabricados, o que facilita a integração do sistema.

### 2.2.1 Sistemas em Chip Reconfiguráveis

De acordo com [38], os sistemas reconfiguráveis foram introduzidos para preencher a lacuna entre ASICs e microprocessadores, visando atender os diversos requisitos de aplicações atuais e futuras. Tais sistemas utilizam conjuntos de elementos de processamento (PEs - *Processing Elements*), cuja funcionalidade e interconexões podem ser reconfiguradas, oferecendo uma flexibilidade que permite o reuso do *hardware* em muitas aplicações, evitando os custos e atrasos de novas fabricações. Sendo assim, constituem uma alternativa atraente para atender os requisitos de múltiplas aplicações e, ao mesmo tempo, reduzir custos e o *time-to-market*.

Os sistemas reconfiguráveis podem ser classificados, de acordo com a granularidade, em duas categorias. Um sistema de granularidade fina consiste de PEs e interconexões que são configuradas em nível de bit. Como implementam qualquer função lógica de 1 bit e existem recursos para realizar a comunicação entre os PEs, os sistemas de granularidade fina permitem uma grande flexibilidade e teoricamente podem implementar qualquer circuito digital. Apesar disso, tais sistemas apresentam desvantagens em termos de desempenho, área e tempo de configuração [38].

Por outro lado, os sistemas de granularidade grossa consistem em PEs reconfiguráveis que implementam operações em nível de palavra e interconexões que possuem flexibilidade suficiente para mapear tipos diferentes de aplicações. Nesses sistemas, a reconfiguração dos PEs e das interconexões é feita em nível de palavra e, sendo assim, em comparação com os sistemas de granularidade fina, oferecem desempenho maior, melhor utilização da área, menor consumo de potência e reduzem o tempo de reconfiguração [38, 39].

A vantagem de sistems reconfiguráveis em geral vem do fato de que o *hardware* é adaptado para um determinado algoritmo, isto é, para conter somente as operações que aparecem nesse algoritmo. Com isso, tais arquiteturas apresentam um desempenho melhor do que os processadores de propósito geral, que interpretam uma sequência linear de instruções e devem prover todas as possíveis operações e tipos de dados que um algoritmo pode requerer [40]. Com o avanço dos SoCs, novas maneiras de se acrescentar as vantagens da reconfigurabilidade em tais sistemas têm sido desenvolvidas para se adaptar a este novo ambiente [41].

Sendo assim, a reconfigurabilidade tem se tornado um aspecto importante no projeto de SoCs devido à crescente demanda de reuso de projeto, portabilidade, proteção contra obsolescência e capacidade de correção de *bugs*. O acréscimo de *hardware* embarcado reconfigurável de diferentes

granularidades resulta em um SoC heterogêneo, comumente referido como RSoC (*Reconfigurable System on Chip*), que possui as vantagens tanto de sistemas reconfiguráveis quanto de elementos tradicionais, como processadores de propósito geral e ASICs em geral[42, 43, 44].

Por outro lado, a enorme complexidade do desenvolvimento de tais sistemas torna o projeto mais vulnerável a erros e requer um tempo maior [45]. Por isso, novas metodologias vêm sendo constantemente propostas. Em [42], por exemplo, é apresentada uma metodologia de projeto de RSoC baseada em SystemC que pode ser facilmente acrescentada ao fluxo de projeto tradicional de SoC para permitir a exploração de diferentes alternativas de reconfigurações sem entrar em detalhes de implementação.

## 2.3 Modelos de Computação (MoC)

Um modelo de computação (MoC - *Model of Computation*) é uma forma generalizada de descrever o comportamento de um sistema de uma maneira abstrata e conceitual. Dessa forma, MoCs são a base para o raciocínio sobre comportamento, requisitos e restrições da computação a ser utilizada. Em geral, os MoCs são representados de maneira formal, usando funções matemáticas, notações de teoria de conjuntos, ou uma combinação de ambos, estabelecendo uma semântica bem definida e permitindo o uso de técnicas formais. MoCs diferentes podem, assim, ter vários graus de caraterísticas, complexidade e expressividade [3].

Apesar de haver várias definições de MoC, neste texto ele será usado para definir a representação de tempo e a semântica de comunicação e sincronização entre processos, conforme é sugerido em [2]. Assim, um MoC define como a computação se dá em uma estrutura de processos concorrentes, atribuindo uma semântica para a mesma. Esta semântica pode ser usada para formular uma máquina abstrata que é capaz de executar um modelo. As linguagens, apesar de não serem modelos computacionais, são baseadas nos mesmos. Por exemplo, as linguagens VHDL, Verilog e SystemC compartilham o mesmo modelo computacional dirigido por eventos e de tempo discreto. Por outro lado, linguagens podem ser usadas para suportar mais de um modelo computacional, como é o caso do SystemC [2].

Escolher o MoC correto é de importância crucial, visto que cada MoC possui certas propriedades. Uma rede de processos modelada em SystemC DE (*Discrete-Event*), por exemplo, pode ser mais facilmente interpretada por ferramentas automáticas se for modelada em SDF (*Synchronous DataFlow*). A seguir, são descritos alguns MoCs importantes, de acordo com a abstração de tempo, ou seja, modelos DT (*Discrete-Time*) e síncronos, nos quais um ciclo denota uma noção de tempo abstrata, são diferenciados de modelos *untimed* de acordo com o modelo de sinais e *tags*, no qual cada evento possui uma *time tag* e diferentes estruturas de *time tags* resultam em MoCs diferentes. Assim, se as *time tags* correspondem a números reais, tem-se um modelo CT (*Continuous-Time*); se correspondem a números inteiros, tem-se um modelo DT e, por fim, *time tags* originadas de um conjunto parcialmente ordenado resultam em um modelo *untimed* [2].

### 2.3.1 MoC DT (*Discrete Time*)

Estes modelos possuem uma noção global de tempo, que é representado por um conjunto discreto, como números inteiros ou naturais, e os eventos possuem uma *time tag*, ou seja, uma informação relativa ao instante em que o mesmo ocorreu. A maioria dos simuladores de circuitos digitais baseia-se neste modelo, e um simulador para este MoC geralmente mantém uma fila que ordena os eventos de acordo com a sua *time tag*. Esta abordagem não é a mais eficiente, já que ordenar as *time tags* consome bastante tempo, principalmente para sistemas mais complexos. Além disso, eventos simultâneos constituem um desafio para estes modelos, já que nem sempre os eventos serão ordenados. Neste caso, a adição de um atraso delta pode ajudar a prevenir tais situações, mas não as evita completamente. Alguns simuladores, como o Ptolemy, tentam analisar estatisticamente os precedentes dos dados que possuem a mesma *time tag* em um determinado instante.

### 2.3.2 MoC CT (*Continuous Time*)

Nestes modelos, o tempo é representado de maneira contínua, geralmente com números reais. Neste caso, equações diferenciais ordinárias e equações algébricas são utilizadas para modelar dinâmica em sistemas mecânicos, circuitos analógicos, processos químicos e vários outros tipos de sistemas físicos.

Simuladores para MoCs CT são baseados em soluções de equações diferenciais que computam o comportamento de um modelo, incluindo *feedback loops* internos. Sendo assim, as simulações desses modelos são muito lentas e apenas uma pequena parte dos sistemas é usualmente modelada em CT [2].

### 2.3.3 MoC SY (*Synchronous*)

Neste caso, o tempo também é representado por um conjunto discreto, como inteiros, mas a unidade elementar de tempo não é uma unidade física, e sim abstrata. Em cada unidade de tempo, todos os processos são avaliados uma vez e todos os eventos que acontecem durante esse processo são considerados simultâneos. Isto implica que a computação de um evento na saída é instantânea. Neste caso, processos concorrentes podem ser facilmente compostos. Entretanto, *feedback loops* com atraso igual a zero podem causar problemas de causalidade [2].

### 2.3.4 MoC *Untimed*

#### 2.3.4.1 Redes de Processos de Fluxo de Dados

Esses processos são bastante utilizados para aplicações de processamento digital de sinais. Trata-se de uma variação especial das redes de processos de Kahn (KPN - *Kahn process networks*), nas quais os processos se comunicam através de canais FIFO. A escrita desses canais é não-bloqueante, enquanto a leitura é bloqueante, o que indica que a leitura de um canal só pode

ser feita quando o mesmo contém dados suficientes (*tokens*). Os processos nas KPN somente necessitam de informações parciais nas entradas para produzirem resultados parciais na saída (ou seja, são monotônicos), o que permite o paralelismo, já que um processo não precisa do sinal inteiro de entrada para começar a computar os eventos de saída. Apesar dos eventos serem ordenados em cada sinal, não existe relação de ordem entre os eventos de diferentes sinais. Sendo assim, as KPN são parcialmente ordenadas, e portanto são classificadas como *untimed* [2].

Um programa *dataflow* é um grafo direcionado que consiste em nós (atores) que representam a comunicação, e arcos que representam sequências ordenadas (*streams*) de eventos (*tokens*), conforme mostra a Figura 2.2. Os círculos vazios representam os nós, as setas representam *streams*, e os círculos preenchidos representam *tokens*. A execução do processo é uma sequência de ativações ou avaliações. Para cada ativação, *tokens* são produzidos ou consumidos, e a sua quantidade pode variar de acordo com as regras de cada ator [2].

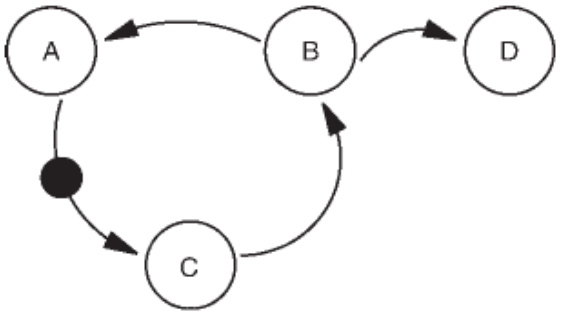


Figura 2.2: Rede de processo de fluxo de dados [2]

Nos modelos SDF (*Synchronous Data Flow*) existem mais restrições, pois, para cada ativação, um número fixo de *tokens* é produzido e consumido. Detalhes sobre SDF ficam além do escopo deste trabalho e não serão discutidos aqui.

#### 2.3.4.2 Modelos *Rendezvous*

Esses modelos consistem em processos sequenciais concorrentes que se comunicam somente nos pontos de sincronização, ou seja, para haver troca de informações, os processos devem ter alcançado um desses pontos. Cada processo sequencial possui seu próprio conjunto de *tags* e nos pontos de sincronização os processos compartilham a mesma *tag*. Sendo assim, há uma ordem parcial de eventos nesses modelos [2].

### 2.3.5 Sistemas Heterogêneos

Esta abordagem tem a vantagem de usar um MoC adequado para cada parte do sistema, o que propicia um modelo geral, flexível e útil. Por outro lado, a semântica da interação entre os MoCs deve ser definida e não é trivial, o que torna a validação mais complicada, já que o modelo

do sistema não é baseado em uma mesma semântica. Sendo assim, a simulação continua sendo a única maneira de validar tais modelos [2]. O projeto Ptolemy [19] estuda a interação de MoCs mistos utilizando um *framework* hierárquico no qual a especificação de um MoC pode conter uma primitiva que é implementada internamente usando outro MoC.

## 2.4 Níveis de Abstração

O aumento da complexidade dos sistemas tem requerido um aumento no nível de abstração por parte das metodologias e ferramentas de projeto. Durante o projeto, é necessário trabalhar com implementações de *hardware* e *software* em vários níveis de abstração. Por exemplo, um projetista pode trabalhar com um modelo em nível detalhado e utilizar *testbenches* (blocos de estímulo) em nível mais abstrato, ou pode também refinar alguns blocos de um modelo funcional para o nível RTL, enquanto outros permanecem em alto nível. Para entender melhor os níveis existentes, o diagrama Y [3] será utilizado, conforme mostra a Figura 2.3.

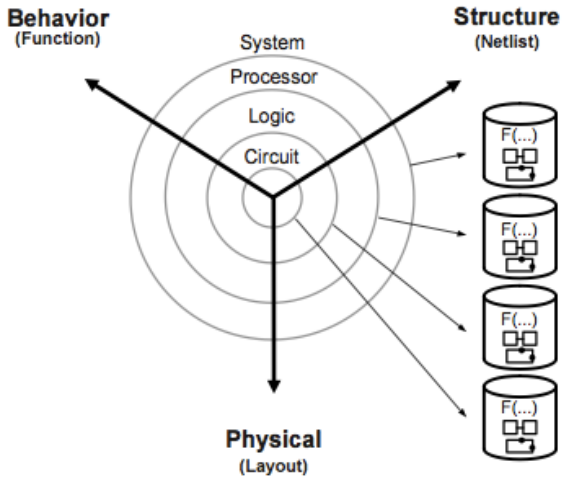


Figura 2.3: Diagrama Y [3]

Os eixos no diagrama representam três diferentes domínios de descrição, que são o comportamental, o estrutural e o físico.

- Comportamental, especificação ou funcionalidade: o modelo reflete a funcionalidade da implementação real. O projeto é representado como uma caixa-preta e suas entradas e saídas no tempo são descritas.
- Estrutural, *netlist* ou diagrama de blocos: o modelo reflete a estrutura da implementação real. A caixa-preta já é representada como um conjunto de componentes e conexões.
- Físico, *layout* ou placa: adiciona dimensão à estrutura. Especifica o tamanho e a posição de cada componente, bem como as portas e conexões entre eles.

Os níveis de abstração são representados como círculos concêntricos ao redor da origem, e quanto mais distante do centro do Y, mais abstrato é o nível. O nome de cada nível deriva do tipo

de componente gerado nele. Assim, são representados os níveis de circuitos, lógico, de processador e de sistema.

- Circuito: os componentes gerados são células padrão, que consistem em transistores.
- Lógico: os componentes gerados são portas lógicas e flip-flops, que serão integrados em registradores e unidades funcionais, como unidades lógico-aritméticas e multiplicadores.
- Processador: são gerados processadores padronizados ou customizados, ou componentes como controladores de memória, árbitros, roteadores, etc.
- Sistema: são projetados sistemas embarcados compostos por processadores, memórias, barramentos e outros componentes.

A elevação do nível de abstração para os níveis sistêmicos oculta os detalhes, tornando mais fácil descrever a funcionalidade de cada módulo e suas interconexões, e tem como vantagens: facilidade de modelagem, gerenciamento e validação do sistema descrito logo nos primeiros estágios do projeto. Os domínios de *hardware* e *software* podem ser explorados paralelamente e validados de forma homogênea [46].

## 2.5 SystemC

C++ é uma linguagem orientada a objetos que permite a abstração de dados e a elaboração de projetos hierárquicos. SystemC [21] é uma biblioteca do C++ utilizada para a descrição de *hardware* que contempla níveis sistêmicos de abstração de projeto. Por ter mecanismos que permitem a modelagem em níveis mais altos de abstração, é mais do que uma simples linguagem de descrição de *hardware*.

O uso de SystemC permite a descrição de *hardware* e *software* em um ambiente homogêneo, facilitando o processo de compreensão, descrição e validação de projeto. Esta linguagem facilita o processo de modelagem de *hardware* por possuir características como tipos de dados próprios para definição de hardware, estruturas e processos que flexibilizam a descrição de paralelismo natural em *hardware*. A base fornece um núcleo de simulação orientado a objeto e qualquer programa pode ser compilado usando um compilador C++ e produzir um arquivo executável [17].

O módulos são blocos básicos usados para particionar um projeto complexo, de maneira que o mesmo se torne mais gerenciável. Um módulo típico contém portas (para se comunicar com outros módulos), processos (que descrevem a sua funcionalidade), dados internos, canais (para comunicação entre processos do módulo), e outros módulos em níveis hierárquicos mais baixos. Podem ser instanciados seguindo as regras de instanciação de objetos e algumas adicionais do C++.

A criação de um módulo requer herança de uma classe especializada chamada *sc_module*, que define características comuns de todos os módulos, tais como os nomes. A macro *SC_CTOR* declara um construtor, que registra os processos e declara a sensitividade a eventos, dentre outras funções. Seu argumento deve ser o nome do módulo que está sendo declarado.

Um construtor alternativo é o *SC_HAS_PROCESS*, que pode ser usado em duas situações. Primeiro, quando são necessários construtores com argumentos além do nome do módulo passado para *SC_CTOR*. Segundo, quando se quer colocar o construtor no arquivo de implementação. Para usar essa abordagem, basta chamar *SC_HAS_PROCESS* antes da definição do construtor convencional.

Os processos são usados para descrever as funcionalidades dos módulos e fornecem um mecanismo para simular o comportamento concorrente requerido por sistemas eletrônicos. São sensíveis a sinais, de acordo com uma lista de sensitividade, que disparam o processo quando ocorre um determinado evento.

Existem 3 tipos de processos:

- Métodos (*sc_method*): não podem ser suspensos e podem possuir listas de sensitividade estáticas ou dinâmicas.
- Threads (*sc_thread*): são executados indefinidamente, mas podem ser suspensos pelo comando *wait()*.
- *Clocked Threads (sc_cthread)*: este tipo só pode ser sensível ao *clock* e é usado para criar máquinas de estado implícitas, nas quais um estado é definido entre dois comandos *wait()* ou *wait_until()*. Trata-se de um caso particular de *thread*.

As interfaces, portas e canais são usados para adicionar flexibilidade à linguagem. Uma porta é ligada a um canal a partir de uma interface. As interfaces são um conjunto de operações que especifica somente a assinatura de cada operação, como o nome, os parâmetros e o valor de retorno. Não especifica como as operações são implementadas. Todas as interfaces são derivadas, direta ou indiretamente, da classe *sc_interface*, que fornece uma função virtual *register_port()* que é chamada quando uma porta é conectada (via uma interface) a um canal. Os argumentos da função são: 1. uma referência para o objeto, e 2. o nome do tipo da interface que a porta espera. O nome do tipo permite ao canal assegurar a legalidade da ligação da porta ao canal.

As portas são usadas para conectar os módulos com a vizinhança, para que os mesmos possam se comunicar. São representadas por objetos,e não simplesmente ponteiros ou referências. Uma desvantagem é que a verificação das ligações durante a instanciação dos módulos não pode ser feita durante a compilação. É necessária a elaboração. Cada tipo de porta deve especificar a interface correspondente.

Enquanto interfaces e portas definem juntas que funções estão disponíveis em um pacote de comunicação, os canais definem como essas funções são executadas. Canais diferentes podem implementar a mesma interface de diversas maneiras. Além disso, um canal pode implementar mais de uma interface, contanto que possua as operações especificadas em todas as interfaces dele. Eles fornecem meios de comunicação entre módulos e entre processos dentro de um módulo, mas não está limitado a conexão ponto-a-ponto. As classes de canal são:

- Canais primitivos: são os que suportam o método de acesso *request-update*, que está relacionado à semântica de execução do SystemC para simular concorrência.

- Canais hierárquicos: são necessários para modelagem de estruturas mais complexas de comunicação. Possuem processos internos, além dos módulos que implementam uma ou mais interfaces.

Uma boa maneira de entender a estrutura da linguagem SystemC é utilizando um exemplo, conforme mostra a Figura 2.4.

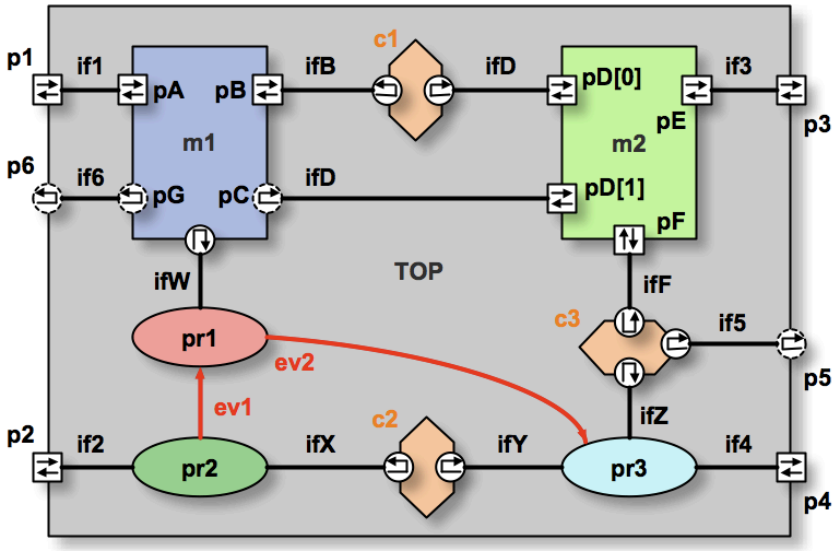


Figura 2.4: Exemplo de módulos e suas conexões em SystemC [4]

Há três retângulos representando módulos. O de hierarquia superior é chamado de *TOP*, e os dois submódulos são *m1* e *m2*. Cada módulo possui uma ou mais portas, representadas por quadrados e círculos cujas setas indicam o fluxo de informação. As portas de *TOP* são *p1, p2, p3, p4, p5* e *p6*, que estão ligadas às interfaces *if1, if2, if3, if4, if5* e *if6*, respectivamente.

O módulo *m1* possui, além das portas, as interfaces *ifW* e *if6*. O módulo *m2* mostra que é possível utilizar um *array* de portas (*pD[0]* e *pD[1]*). Pode-se notar que a interface utilizada é a mesma (neste caso, *ifD*). Os hexágonos representam canais dentro de *TOP*. Cada canal implementa uma ou mais interfaces representadas por círculos com setas. O canal *c1* implementa as interfaces *ifB* e *ifD*, por exemplo. *TOP* também possui três processos chamados *pr1, pr2* e *pr3*. Há dois eventos implícitos (*ev1* e *ev2*) usados para a sinalização entre eles.

A Tabela 2.1 [4] apresenta um resumo das possíveis interconexões entre os elementos mostrados na Figura 2.4.

### 2.5.1 A API TLM em SystemC

Em um modelo em nível de transação, os detalhes de comunicação entre os componentes computacionais são separados dos detalhes de implementação. A comunicação é modelada utilizando canais. As requisições são feitas por meio de chamadas a funções nas interfaces desses canais. Com isso, detalhes de comunicação e computação são ocultados no TLM, propiciando um aumento na

velocidade de simulação, exploração e validação de implementações alternativas.

A comunicação entre os módulos é modelada usando chamadas de funções. É precisa em termos de funcionalidade e frequentemente em termos de temporização, mas não o suficiente para ser precisa em termos estruturais. Por exemplo, no caso de um SoC, diferentes tipos de transações suportadas por barramentos *on-chip* (*burst read/write*) podem ser modeladas, mas os fios do barramento ou pinos dos módulos não são modelados.

Uma plataforma em nível de transação indica um modelo que usa o nível de transação para modelar as estruturas de comunicação de uma plataforma SoC, e os módulos correspondem estruturalmente aos da implementação real. Essa plataforma pode ser usada para modelar com precisão o desempenho geral do sistema e, ao mesmo tempo, é uma maneira rápida e efetiva de modelar a interação *hardware/software* em um estágio inicial do projeto.

O padrão TLM foi definido pela *Open SystemC Initiative* [21] e se encontra atualmente na versão 2.0. A versão 1.0 foi definida como um conjunto de interfaces para efetuar transações por valores ou por referências constantes. Apesar disso, limitações como a falta de uma classe padrão de transações e suporte para informação de tempo entre os modelos levaram à atualização para TLM 2.0, onde tais problemas são resolvidos.

A Figura 2.5 mostra, de forma resumida, os casos de uso, estilos de codificação e mecanismos do TLM. Há quatro casos de uso: desenvolvimento de *software*, análise de performance de *software*, análise de arquitetura de *hardware* e verificação da performance do *hardware*. Para os dois primeiros, o estilo *loosely-timed* é utilizado. Para o terceiro, podem ser usados tanto o *loosely-timed* quanto o *approximately-timed*. Para o último, é utilizado o *approximately-timed*.

Nas simulações *loosely-timed*, também conhecidas como PV (*Programmer View*), é utilizado um recurso conhecido como *temporal decoupling*, que permite que partes do modelo sejam executadas antes para depois sincronizarem com o restante do sistema, permitindo simulações mais rápidas. No caso do estilo *approximately-timed*, é feita uma subdivisão do tempo de vida de uma transação em diferentes fases.

Conforme pode ser visto na Figura 2.6, um *initiator* é um módulo que inicia novas transações, e um *target* é um módulo que responde às transações iniciadas por outros módulos. Um componente

Tabela 2.1: Interconexões em SystemC [4]

| **De** | **Para** | **Método** |
|---|---|---|
| Porta | Submódulo | Conexão direta via sc_port |
| Processo | Porta | Acesso direto por processo |
| Submódulo | Submódulo | Conexão com canal local |
| Processo | Submódulo | Conexão com canal local ou via sc_export ou interface implementada por submódulo |
| Processo | Processo | Eventos ou canal local |
| Porta | Canal local | Conexão direta via sc_export |

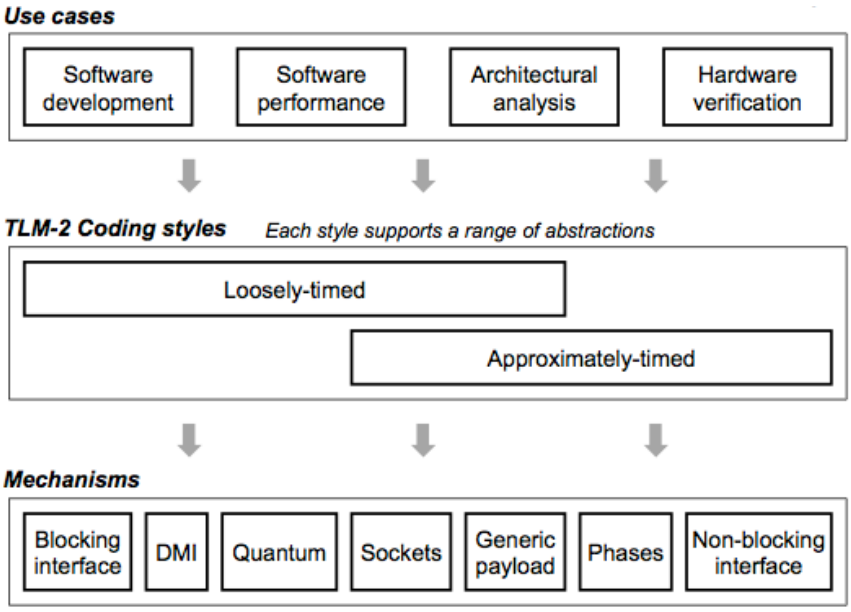


Figura 2.5: Casos de uso, estilos de codificação e mecanismos do TLM 2.0 [5]

de interconexão é um módulo que simplesmente encaminha as transações recebidas sem alterá-las. As transações são estruturas de dados passadas entre *initiators* e *targets* através de chamadas de funções. Para isso, são utilizados *sockets*. Quando a transação é passada de um *initiator socket* para um *target socket*, a sequência é conhecida como *forward path*. A transação é executada no *target* e o objeto deve retornar ao *initiator* de duas maneiras: como retorno da chamada do método de transporte (*return path*) ou passada explicitamente no sentido inverso, ou seja, do *target* para o *initiator* (*backward path*).

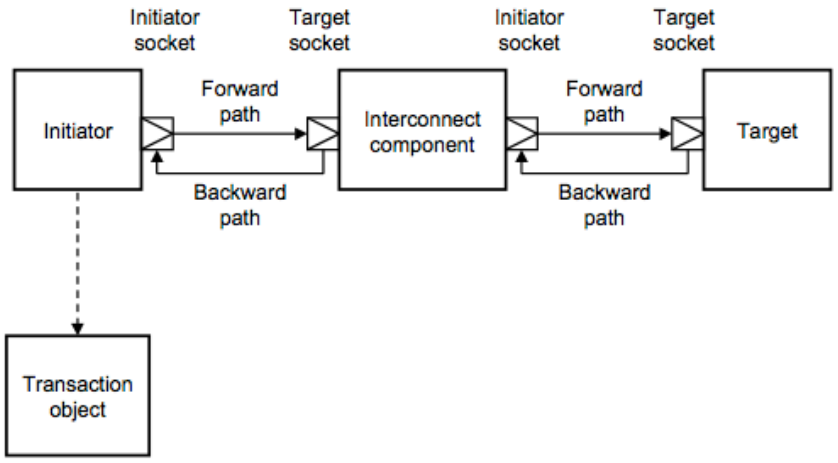


Figura 2.6: Mecanismos do TLM para o envio de transações [5]

As interfaces bloqueantes e não-bloqueantes são distintas e suportam diferentes tipos de detalhes de temporização. As primeiras são usadas no estilo *loosely-timed* e somente modelam o início e o fim de uma transação, sendo que a mesma é completada com uma única chamada de função. As

últimas são usadas no estilo *approximately-timed* e permitem que uma transação seja dividida em vários pontos, sendo que são necessárias múltiplas chamadas de funções para uma única transação.

O *generic payload* é um recurso utilizado pelo TLM 2.0 que garante a interoperabilidade entre modelos. Ele pode ser usado como um tipo de transação de propósito geral para modelos de barramentos mapeados em memória, quando se abstrai os detalhes de um protocolo de barramento em particular, e também pode ser usado como base para modelar uma série de protocolos específicos em um nível mais detalhado. Este último ponto facilita a mudança de protocolos, desde que os mesmos estejam baseados no *generic payload*.

Cada *generic payload* possui atributos padronizados, dentre os quais pode-se citar:

- *Command*: pode ser *read* ou *write*
- *Address*: este atributo define o menor valor de endereço no qual os dados são lidos ou escritos
- *Data*: é um ponteiro que aponta para um *buffer* de dados no initiator. Há um atributo *length* que define o comprimento do dado em bytes
- *Response Status*: indica o *status* da transação

### 2.5.2 SystemC-AMS

O projeto de SoCs tem ficado mais complexo pelo fato de envolver cada vez mais sistemas que são baseados em MoCs diferentes. Por isso, extensões que permitem a simulação de sistemas de sinais mistos estão sendo desenvolvidas [47]. As extensões SystemC-AMS são baseadas no padrão SystemC e definem construções de linguagem adicionais que introduzem novas semânticas e metodologias, em nível de sistema, para o projeto e verificação de sistemas mistos. Com isso, podem ser usadas em conjunto com o SystemC, formando um *framework* consistente, para criar especificações executáveis, validar e otimizar arquiteturas de sistemas mistos, explorar algoritmos e disponibilizar um protótipo virtual de todo o sistema - incluindo a funcionalidade das partes analógicas - para a equipe de desenvolvimento de *software*.

A Figura 2.7 mostra os casos de uso, as abstrações e as semânticas de execução dos MoCs das extensões SystemC-AMS [6]. Cada um deles será brevemente descrito a seguir.

Uma especificação executável serve para verificar se a especificação de um determinado sistema está sendo atendida de forma correta por meio de simulação. Os modelos utilizados aqui não estão necessariamente relacionados à arquitetura física do sistema e, por isso, são chamados de modelos funcionais. SystemC e suas extensões AMS definem tanto a modelagem em nível de sistema quanto a semântica de sua execução para fins de simulação implementadas na forma de bibliotecas C++, que são ligadas aos modelos AMS compilados para criar uma descrição executável do sistema.

O propósito dos protótipos virtuais é permitir o desenvolvimento de *software* utilizando um modelo em alto nível, temporizado ou não temporizado, que representa o *hardware* da arquitetura. Além da velocidade de simulação, existe a possibilidade de interagir com modelos em SystemC TLM.

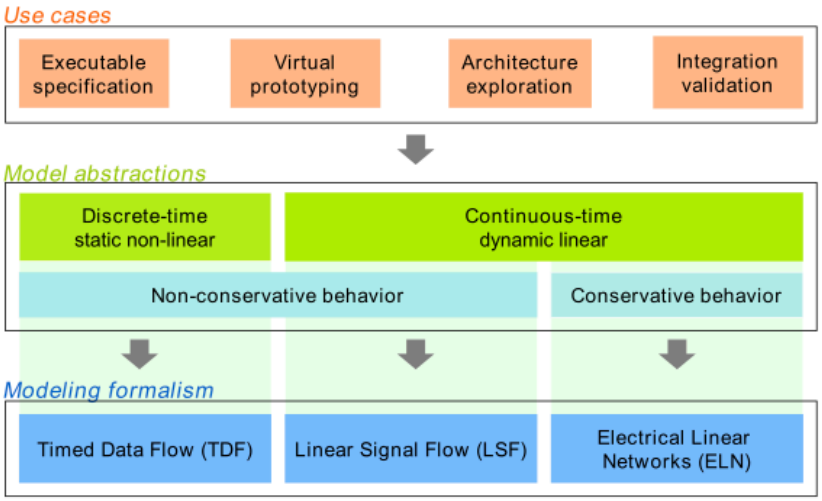


Figura 2.7: Casos de uso, abstrações e formalismos do SystemC-AMS [6]

A exploração da arquitetura avalia se e como as funções definidas nas fases anteriores podem ser mapeadas no *hardware* do sistema. Primeiramente, a especificação executável é refinada com a adição de limitações para avaliar o impacto sobre o comportamento do sistema. Em seguida, a arquitetura e suas interfaces são refinadas com a adição de novos elementos e da comunicação entre eles. Finalmente, os componentes são integrados e a funcionalidade do sistema completo é verificada. Neste caso, as interfaces e os tipos de dados utilizados nos modelos devem corresponder aos da implementação física.

Os níveis de abstrações mostrados na Figura 2.7 distiguem tempo discreto de tempo contínuo e descrições não-conservativas de conservativas. No caso de tempo discreto, os sinais são abstraídos como sequências de valores definidos em pontos discretos. Tais valores podem ser reais, inteiros ou lógicos, por exemplo. Essa abordagem é útil tanto para descrições de sistemas que envolvem processamento de sinais quanto para aproximar comportamentos contínuos no tempo. No caso de tempo contínuo, os sinais são abstraídos como funções do tempo e os comportamentos são descritos utilizando equações matemáticas, o que geralmente requer algoritmos mais complexos.

Modelos não conservativos expressam comportamentos como fluxos de sinais contínuos no tempo sobre os quais funções como filtros ou integrações são aplicadas. Já os modelos conservativos são os mais detalhados, já que as leis de conservação de energia (leis de Kirchhoff) devem ser observadas. Neste caso, o conjunto de equações a serem resolvidas é maior e mais complexo.

O padrão AMS define a semântica de execução baseada nos MoCs *Timed Data-Flow* (TDF), *Linear Signal-Flow* (LSF) e *Electrical Linear Network* (ELN). Na versão 1.0, que é o padrão atual, as definições de classes e interfaces são definidas na implementação, e o projetista pode se beneficiar do uso de classes e interface dedicadas para criar modelos TDF, LSF ou ELN utilizando módulos pré-definidos, portas, terminais, sinais e nós [48].

O MoC LSF define módulos não conservativos, tais como somadores, multiplicadores, diferenciadores e integradores. Um modelo LSF é feito através da conexão desses elementos primitivos através de sinais de valores reais no domínio do tempo.

O MoC ELN define elementos elétricos conservativos, como resistores, capacitores e indutores. Eles são usados como modelos que descrevem as relações entre tensões e correntes. Além disso, vários elementos conectando os dois domínios estão disponíveis. Em LSF, há multiplexadores, demultiplexadores e fontes. Em ELN, há chaves e fontes de tensão e corrente.

O TDF é baseado no formalismo do SDF (*Synchronous Data Flow*), mas, ao contrário do que ocorre no SDF, no TDF é utilizada a modelagem com tempo discreto, na qual os dados são considerados como sinais amostrados no tempo. Esses sinais podem ter valores discretos ou contínuos, como amplitudes. Essa amostragem no tempo permite a descrição do comportamento analógico e facilita a sincronização e comunicação com outros MoCs. Ao contrário do que ocorre com o agendamento dinâmico imposto pelo núcleo baseado em eventos do SystemC, a simulação é acelerada ao definir um agendamento estático, que é realizado antes do início da simulação. Como este trabalho utiliza apenas o MoC TDF, ele será mais detalhado na subseção a seguir.

#### 2.5.2.1 MoC TDF (*Timed Data-Flow*)

O princípio básico da modelagem TDF é descrito em [6]. A Figura 2.8 mostra um exemplo utilizando três módulos (A, B e C). Um modelo TDF é composto por um conjunto de módulos TDF conectados, que formam um grafo direcionado chamado *TDF cluster*, no qual os módulos correspondem aos vértices e os sinais TDF, às arestas. Um módulo pode conter várias portas de entrada e saída, e nos casos específicos em que contêm apenas portas de saída são chamados de produtores (*source*). Caso contenham somente portas de entrada, são chamados de consumidores (*sink*). Na Figura 2.8, A é produtor e C é consumidor.

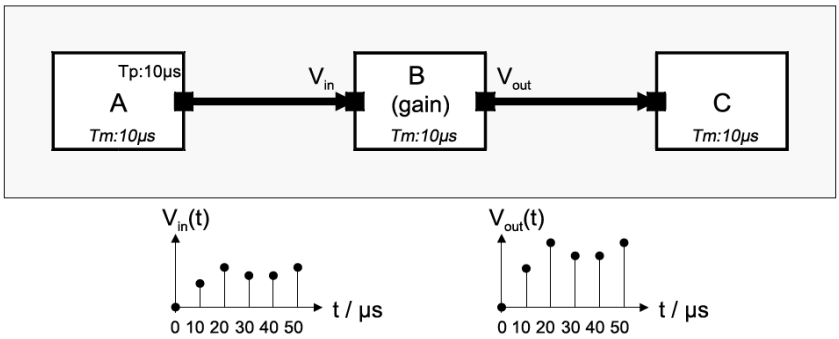


Figura 2.8: *Cluster TDF* [6]

Enquanto os sinais são utilizados para conectar os módulos, estes contêm métodos C++ que executam funções matemáticas $f$ que dependem das entradas e de possíveis estados internos. Sendo assim, o comportamento do *cluster* é determinado pela composição matemática das funções de seus módulos, na ordem apropriada, indicada como {A -> B -> C}, no caso da Figura 2.8.

Uma dada função é processada se, e somente se, há amostras suficientes nas portas de entrada. Caso esta condição seja satisfeita, as amostras nas entradas são lidas pelo módulo, cujas funções são aplicadas nas mesmas e os resultados são escritos nas portas de saída apropriadas. O número de amostras lidas ou escritas é fixo durante a simulação, e um módulo pode possuir um número

diferente de amostras na entrada e na saída. A cada amostra, um *time stamp* é associado e o intervalo fixo entre duas amostras é chamado de *time step*.

Na modelagem TDF, é possível definir o *time step* de um módulo (Tm) e de suas portas (Tp). Também é possível definir os seguintes atributos:

- *Port rate* (R): determina o número de amostras lidas ou escritas. Por exemplo, se R = 2 em uma porta de entrada, a cada ativação do módulo, duas amostras serão lidas.
- *Port delay* (D): determina o atraso com que as amostras são lidas ou escritas. Por exemplo, se D = 1 em uma porta de saída, a amostra correspondente ao *time step* anterior será escrita quando o módulo for ativado.
- *Port time offset* (Tpf): este parâmetro determina o tempo real da primeira amostra da porta e somente pode ser atribuído às portas que são conectadas ao domínio discreto (DE), ou portas conversoras.

Uma vez definidos esses atributos, é feita uma verificação de consistência do *cluster*. Caso sejam compatíveis, a ordem de ativação dos módulos e o número de amostras lidas e escritas pode ser determinado estaticamente, antes mesmo do início da simulação. Com isso, a execução de modelos TDF não se baseia no mecanismo do *kernel* do SystemC e pode ser feita com mais eficiência. Apesar de serem processados de maneira independente e usando um mecanismo local de *time annotation*, os modelos TDF podem interagir com modelos SystemC por meio de portas conversoras, conforme é mostrado na próxima subseção.

#### 2.5.2.2 Interação entre os MoC TDF e DE

Para manter a eficiência da simulação quando há módulos DE e TDF conectados, é utilizado um mecanismo chamado de sincronização de dados, no qual eventos discretos não influenciam na ativação de módulos TDF.

O primeiro caso é quando um módulo TDF necessita ler um sinal originado no domínio DE. A porta conversora utilizada é da classe *sca_tdf::sca_de::sca_in<T>* (Figura 2.9). Neste caso, a ordem de ativação do módulo TDF é determinada de maneira independente, de acordo com os atributos Tp e Tm do módulo, e R da porta conversora.

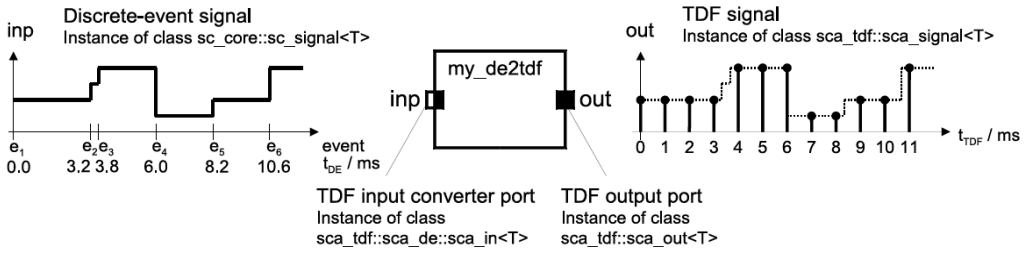


Figura 2.9: Porta conversora de DE para TDF [6]

Para que a sincronização ocorra corretamente, o valor lido pela porta conversora deve estar disponível no primeiro *delta cycle* do tempo correspondente no domínio DE. Enquanto o *cluster* TDF é executado de forma independente, existe a possibilidade de leitura de um valor anterior do domínio DE, indicando que o processo DE não escreveu o valor no canal antes do primeiro *delta cycle* e resultando em um atraso no sinal. Por isso, as portas conversoras possuem o atributo Tpf, conforme mostrado na subseção anterior, que pode ser definido utilizando a função *set_ timeoffset* e resolver o problema do atraso.

No segundo caso, o módulo TDF escreve um valor em um canal no domínio DE. A porta conversora utilizada é da classe *sca_ tdf::sca_ de::sca_ out<T>* (Figura 2.10). Os atributos Tpf e Tm da porta conversora definem o instante e o intervalo de tempo nos quais o valor é escrito no domínio DE.

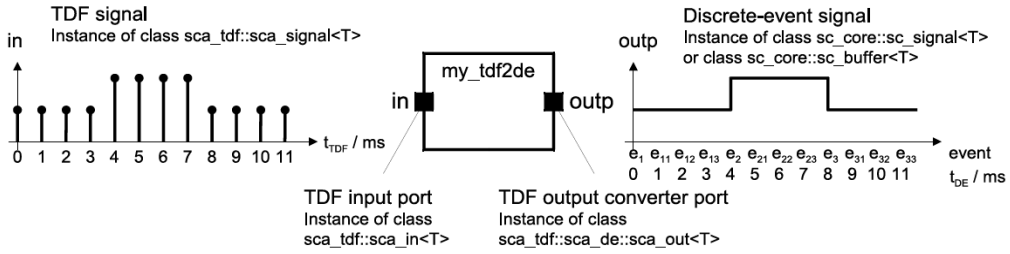


Figura 2.10: Porta conversora de TDF para DE [6]

De maneira similar ao caso anterior, a amostra escrita pela porta conversora deve ser disponibilizada no canal no primeiro *delta cycle* do tempo correspondente no domínio DE para que a sincronização seja feita corretamente. Caso o canal seja da classe *sc_ core::sc_ signal<T>*, somente um evento discreto é gerado quando o sinal muda (eventos *e1,e2,e3* na Figura 2.10). Se o canal for da classe *sc_ core::sc_ buffer<T>*, todas as amostras irão gerar um evento, como indicado na Figura 2.10 por *e11,e12,e13*, etc.

#### 2.5.2.3 Sintaxe dos Módulos TDF

A Listagem 2.1 mostra a estrutura típica de um módulo TDF [6].

```
SCA_TDF_MODULE(meu_modulo_tdf)
{
    // Declarações das portas
    sca_tdf::sca_in<double> in;
    sca_tdf::sca_out<double> out;

    SCA_CTOR(my_tdf_module) {}

    void set_attributes()
    {
    // Atributos das portas e dos módulos (opcional)
    }
```

```
    void initialize()
    {
    // Valores iniciais das portas com atraso (opcional)
    }
    void processing()
    {
    // Algoritmo de processamento de sinais no domínio do tempo.
    // Cada ativação do módulo avança o tempo em um time step
    }
    void ac_processing()
    {
    // Comportamento de pequenos sinais no domínio da frequência (opcional)
    }
};

class meu_segundo_modulo : public sca_tdf::sca_module
{
public:
    // Declarações das portas
    // ...

    meu_segundo_modulo( sc_core::sc_module_name ) {}

    // Definição das funções de maneira similar ao primeiro modulo
    // ...
};
```

Listagem 2.1: Módulo TDF típico

Os atributos das portas e do módulo, tais como taxa de amostragem, atrasos e *time step* são definidos na função *set_attributes*, que é chamada durante a fase de elaboração. A função *initialize* é executada apenas uma vez (antes da ativação do módulo) e pode ser usada para atribuir valores a variáveis locais, usadas como variáveis de estado, para ler os atributos do módulo ou para inicializar portas com um determinado atraso. A função *processing* é executada toda vez que o módulo é ativado, e é a única obrigatória, pois define o comportamento do módulo.

A macro *SCA_CTOR* define o construtor padrão (*sca_tdf::sca_module*) e o único argumento obrigatório é o nome do módulo. Um construtor regular pode ser usado, caso existam mais parâmetros a serem passados.

Algumas limitações dos módulos TDF são:

- Não é possível instanciar submódulos. A composição estrutural é possível definindo-se classes derivadas de *sc_core::sc_module* ou usando o equivalente *SC_MODULE*.
- As funções *set_attributes*, *initialize*, *processing* e *ac_processing* são chamadas como parte da semântica de execução e não podem ser chamadas diretamente pelo usuário.
- As funções que descrevem o comportamento DE em SystemC, tais como criação de métodos e *threads*, não são permitidas, pois podem interferir na execução dos módulos TDF.

- Como o tempo local é independente do *kernel* do SystemC, a função *sc_ core::sc_ time_ stamp* não pode ser utilizada. Neste caso, deve-se usar a função *get_ time*.
- É necessário usar portas conversoras, caso existam sinais em SystemC.

### 2.5.3 Biblioteca SCNSL (*SystemC Network Simulation Library*)

Como as escolhas da arquitetura do sistema podem afetar o comportamento da rede e vice-versa, é importante modelar e simular o ambiente no qual o sistema opera. A SCNSL permite modelar cenários de rede nos quais tipos diferentes de nós, ou nós descritos em níveis de abstração diferentes, podem interagir. Esta biblioteca foi descrita em SystemC para permitir uma integração mais fácil entre *hardware*, *software* e rede [29].

Ainda de acordo com [29], as ferramentas de simulação de rede não podem ser usadas para o projeto de sistemas, visto que as mesmas não modelam concorrência dentro do nó e também não é possível realizar a síntese de *hardware* ou *software* a partir das mesmas.

Para tornar a simulação de rede possível, a biblioteca conta com os seguintes elementos:

- Núcleo: executa eventos na ordem temporal correta e deve levar em conta as características do canal, tais como atraso de propagação e perda de sinal. É implementado por uma classe que gerencia as transmissões, ou seja, ela pode cancelar transmissões, mudar a posição dos nós, verificar quais nós estão disponíveis para receber um pacote e se um determinado pacote recebido foi corrompido devido a colisões. Para isso, são utilizadas as primitivas do núcleo do SystemC, tais como modelos de concorrência e eventos.
- Nó: são os elementos ativos da rede. Produzem, transformam e consomem dados transmitidos. Do ponto de vista do projetista de rede, os nós são simplesmente produtores ou consumidores de pacotes. Já do ponto de vista do projetista de sistema, a implementação do nó é de importância fundamental. Por isso, uma classe foi criada para separar a implementação do nó da simulação de rede. Cada nó é conectado a um *NodeProxy*, o que permite manter uma interface estável com a rede e, ao mesmo tempo, proporciona total liberdade para a modelagem do nó, no qual podem ser adotadas diferentes estratégias, tais como blocos interconectados ou máquinas de estado finitas.
- Pacote: é a unidade de dados que é trocada entre os nós. Possui cabeçalho e dados. Apesar de algumas características estarem sempre presentes, de maneira geral, o formato do pacote depende do protocolo a ser adotado. No projeto de sistemas, é necessária uma descrição mais precisa do conteúdo do pacote, enquanto que para a simulação de rede bastam os campos para bitrate e algumas *flags* para marcar colisões. Neste último caso, se o roteamento for feito pelo simulador, os endereços da fonte e do destino são usados também.
- Canal: é a abstração do meio de transmissão que conecta dois ou mais nós. Pode ser um *link* ponto-a-ponto ou um meio compartilhado.
- Porta: os nós usam as portas para enviar e receber dados.

Para que uma transmissão seja validada, a SCNSL possui recursos para verificar se ocorreram colisões, e neste caso outra transmissão deve ser realizada, e também para verificar se a distância entre os nós não excede o alcance máximo.

No caso de um cenário de rede sem fio, são atribuídas ao nó algumas propriedades para reproduzir o comportamento da rede. A taxa de transmissão representa o número de bits por unidade de tempo que a interface pode suportar e é usada para calcular o atraso de transmissão e a carga da rede. A potência de transmissão é usada para avaliar o alcance e a relação sinal-ruído. A porta do nó relativa ao sensor é conectada a um módulo de estímulo que representa uma fonte de dados genérica.

O *NodeProxy*, que serve de interface entre o nó e a rede, gerencia duas propriedades do nó, que são a posição e a sensibilidade do receptor. A primeira é usada para calcular a perda e também para reproduzir cenários móveis. A última é a potência mínima de sinal abaixo do qual o pacote não pode ser recebido.

Quando o nó inicia a transmissão, a sua posição relativa em relação aos outros nós é calculada e o nível do sinal nos outros nós é obtido através de uma relação que depende da distância entre eles. Para cada nó, se o nível do sinal é maior do que a sensibilidade do receptor, então o pacote pode ser detectado e, consequentemente, pode interferir em outras transmissões. Caso outras transmissões estejam sendo efetuadas ao alcance do nó receptor, é caracterizada uma colisão. Da mesma forma, se há mais transmissões em curso ao alcance do nó transmissor, também é considerada uma colisão. Como nós sem fio não podem detectar colisão, a mensagem não é interrompida e o canal continua ocupado. O tempo de transmissão depende do comprimento do pacote, da taxa de transmissão e do atraso de propagação.

Os resultados reportados em [29] mostram que simulações realizadas no nível *loosely-timed* do TLM são cerca de duas ordens de magnitude mais rápidas do que as realizadas no NS-2 [27], que é uma ferramenta clássica de simulação de redes, o que demonstra a eficiência da SCNSL nesses casos.

## 2.6 Linguagens de Descrição de Arquiteturas e ArchC

Linguagens de descrição de arquiteturas (ADL - *Architecture Description Languages*) foram inicialmente introduzidas para ajudar as equipes de projetos a explorar novas arquiteturas e validar seu *software* correspondente [49]. ArchC [7] é uma ADL híbrida *open source* que gera modelos funcionais de ISS em SystemC TLM. Ela usa tanto as informações estruturais quanto do conjunto de instruções para gerar automaticamente um conjunto de ferramentas de desenvolvimento. Além disso, ela incorpora facilidades de comunicação com modelos de *hardware* descritos em SystemC.

Uma descrição em ArchC possui dois níveis de abstração: funcional e com precisão de ciclos (ou *cycle-accurate*). O primeiro é próprio para permitir o desenvolvimento de *software* em um estágio inicial do projeto. Apesar de não incluir informações de temporização precisas nem detalhes estruturais do processador, ela consiste em uma descrição de alto nível do conjunto de instruções, que pode ser instrumentada com interfaces de comunicação, depuração e hierarquia de memória,

se desejado. Com isso, é possível executar o conjunto de instruções completo da arquitetura e, portanto, quaisquer aplicações compatíveis com a mesma. No caso de uma descrição com precisão de ciclos, os detalhes estruturais do processador, tais como *pipeline* e registradores, são incluídos. Assim, o comportamento das instruções deve ser codificado de maneira mais bem detalhada para refletir uma eventual divisão das operações em estágios de *pipeline* ou multi-ciclos [49].

Ainda em [49], é recomendado que os usuários iniciem o projeto de uma arquitetura em ArchC através de um modelo funcional e, se necessário, posteriormente desenvolvam o modelo com precisão de ciclos, pois a experiência e o conhecimento adquiridos sobre o conjunto de instruções durante a elaboração do primeiro são importantes para o desenvolvimento rápido de um modelo mais refinado. Outro motivo é o fato da verificação do modelo funcional ser mais simples, evitando a propagação de erros que tornariam a depuração do modelo em baixo nível mais complexa e demorada.

Em ArchC, o projetista fornece informações estruturais da arquitetura através da descrição *AC_ARCH*. Com isso, ferramentas de *software* podem ser geradas automaticamente. Essa descrição é razoavelmente simples, contendo campos como memória, banco de registradores e registradores de uso específico. Já as informações sobre o conjunto de instruções estão contidas na parte *AC_ISA*, cuja descrição é dividida em dois arquivos: um contendo as declarações, formatos e sequência de decodificação das instruções em ArchC, e outro contendo o comportamento das instruções em SystemC ou C++ [49].

A partir dessas informações, a ferramenta acsim é utilizada para gerar simuladores interpretados escritos em SystemC. O projetista pode, então, usar o GCC ou quaisquer compiladores C++ disponíveis, para gerar uma especificação executável de sua arquitetura, conforme está mostrado na Figura 2.11.

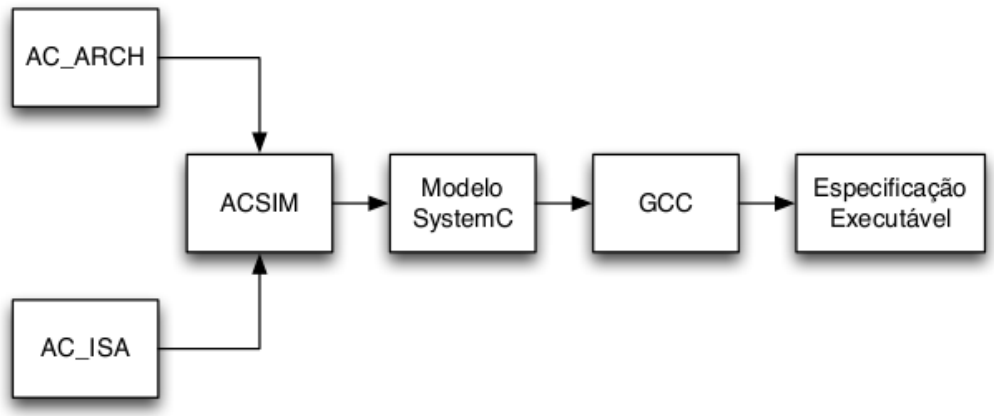


Figura 2.11: Fluxo de geração de simuladores em ArchC [7]

É possível integrar componentes diversos escritos em SystemC aos modelos gerados pelo ArchC. De fato, os processadores ArchC utilizam o padrão TLM 1.0 para comunicação com outros módulos. A versão 2.0 do ArchC permite a adição de uma porta bidirecional TLM initiator no processador, bastando declarar essa porta no arquivo de descrição de recursos da arquitetura. Com isso, a memória passa a ser um módulo externo ao simulador, e o antigo indicador do tamanho da memória

passará a indicar a faixa de endereços que estarão disponíveis para essa porta, a partir do zero (0x0). Para acessar a porta, basta utilizar os métodos *read* e *write*. É importante notar também que outras portas TLM podem ser adicionadas ao modelo e que foi definido um protocolo baseado no padrão TLM chamado de *ArchC TLM Protocol* para definir a comunicação entre módulos externos e o processador ArchC [49].

Por fim, outra porta que pode ser adicionada é a de interrupção externa, a partir da qual os processadores ArchC podem ser interrompidos por um ou mais dispositivos externos. A ferramenta *acsim* cria um conjunto extra de arquivos, um dos quais será editado pelo projetista para a inclusão da rotina de tratamento de interrupções. Assim, basta conectar a porta de saída do módulo externo na porta de interrupção e o valor contido no campo de dado do pacote utilizado no método *transport()* será o valor que chegará como argumento na chamada da rotina de tratamento [49].

# Capítulo 3

# Trabalhos Correlatos

*Grande parte dos sistemas embarcados é heterogênea. Isso motivou a busca de abordagens que englobam diferentes domínios em um mesmo framework durante a etapa de modelagem em nível de sistema. Alguns exemplos serão apresentados a seguir.*

## 3.1 HetSC

HetSC (*Heterogeneous Embedded Systems in SystemC*) [8] é uma metodologia para a especificação de sistemas embarcados em SystemC que apresenta dois níveis. O primeiro consiste em uma metodologia de especificação geral. Neste caso, foi definido no topo do *kernel* de simulação DE do SystemC um conjunto de regras e *guidelines* para a representação gráfica de estruturas em SystemC. Essas regras facilitam outras atividades do fluxo de co-projeto, tais como geração de *software*, síntese de *hardware* e geração de interfaces *HW/SW*.

Com isso, é possível escrever uma especificação executável na qual a metodologia pode ser aplicada. A estrutura da especificação é gerada antes do início da simulação e pode ter um ou mais níveis hierárquicos. O *top-level* é instanciado a partir de uma função *sc_main*, que é constituída do *testbench* e dos módulos do sistema. A comunicação é feita por meio de canais, permitindo uma separação entre computação e comunicação, e a concorrência é possível por meio do uso de processos em SystemC. As portas são utilizadas na comunicação entre processos de diferentes módulos. Além disso, módulos IP podem ser incluídos, desde que possuam portas compatíveis, caso contrário é necessário o desenvolvimento de *wrappers*. A Figura 3.1 mostra as primitivas da especificação.

O segundo nível é a especificação heterogênea que suporta vários MoCs, tais como *untimed* (*Process Networks*, *Kahn Process Networks*, *Communicating Sequential Process*, *Synchronous Data Flow*), síncronos (*Synchronous Reactive*) e *timed*. A biblioteca HetSC, como é chamada, consiste em um grupo de regras que foram acrescentadas ao SystemC padrão (Figura 3.2). Trata-se de macros, interfaces e canais contendo a semântica de comunicação específica para cada MoC e outras características. Foram implementados também, geralmente como classes transparentes ao usuário, elementos para monitoramento dos requisitos de cada MoC.

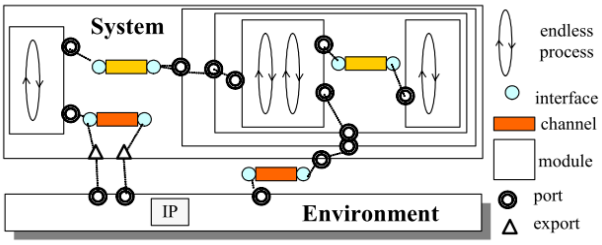


Figura 3.1: Primitivas da especificação do HetSC [8]

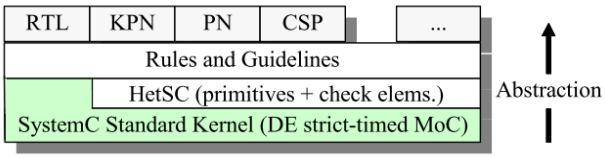


Figura 3.2: Suporte a vários MoCs preservando o *kernel* padrão do SystemC [8]

Como os subsistemas descritos em MoCs diferentes se comunicam entre si, foram definidas interfaces que são utilizadas em *border channels* (BC) e/ou *border processes* (BP). Um BP é um processo que acessa canais que pertencem a partes de MoCs diferentes, no qual a adaptação é escrita de maneira explícita no corpo da função associada ao BP. Um BC é um canal SystemC no qual a interface do MoC está concentrada. A adaptação é feita na implementação do canal de maneira que, no lado de um determinado MoC, o BC é visto como um canal associado a ele.

Uma das principais contribuições do HetCS é a sua eficiência no suporte a MoCs abstratos (*untimed* e *synchronous*) com base no *kernel* de simulação do SystemC (DE). Apesar de não permitir a descrição de componentes no MoC *Continuous Time* (CT), pode ser utilizado com SystemC-AMS, ampliando a faixa de MoCs suportados [50].

## 3.2 Ptolemy II

O Ptolemy II é um ambiente de modelagem pioneiro para sistemas embarcados heterogêneos. Baseado na semântica abstrata de módulos de computação (chamados de atores), um protótipo foi proposto para combinar estilos de modelagem heterogêneos utilizando os MoCs tempo contínuo (CT - *Continuous Time*), eventos discretos (DE - *Discrete Event*) e síncrono/reativo (SR - *Synchronous/Reactive*). O Ptolemy é implementado em Java e possui recursos de inferência de tipos e polimorfismo, que permite que os componentes projetados possam operar com vários tipos de dados [51].

O Ptolemy é bastante flexível. Ele permite a criação de novos componentes e pode ser usado para montar componentes de *software*. Também pode ser utilizado como uma ferramenta de modelagem e simulação, como um editor de diagrama de blocos, como uma aplicação de prototipagem rápida em nível de sistema, dentre outras possibilidades. Neste tópico, ele será abordado como ferramenta de modelagem e simulação.

Uma das grandes vantagens do Ptolemy é a possibilidade de modelar sistemas heterogêneos. Neste caso, é proposta uma abordagem chamada de heterogeneidade hierárquica, na qual é possível dividir um modelo complexo em uma árvore de submodelos, que são compostos separadamente em cada nível, formando uma rede de componentes que interagem entre si. A interação especificada para um determinado nível engloba tanto o fluxo de dados quanto o de controle. Além disso, o modelo completo pode se transformar em um componente que pode ser agregado em outros modelos, que inclusive podem utilizar MoCs diferentes.

Os blocos básicos de um sistema são chamados de atores, que são componentes concorrentes que se comunicam através de interfaces chamadas de portas. Essa abordagem orientada ao ator permite separar a transmissão de dados da transferência de controle. É o MoC, e não os atores em si, que define os detalhes de agendamento e comunicação. Os atores podem ser atômicos (no nível mais baixo da hierarquia) ou compostos (contêm outros atores). Ambos os tipos são executáveis, sendo que a execução de um ator composto é controlada pela composição da execução dos atores que ele contém.

A implementação de um MoC associado a um ator composto é chamada de domínio, que é o que define a semântica de comunicação e a ordem de execução entre os atores. Isso é feito por duas classes, que são o diretor e o receptor. Os receptores implementam o mecanismo de comunicação, estão contidos em portas de entrada e há um receptor para cada canal. Quando são alocados em um determinado domínio, os atores adquirem receptores específicos deste domínio, o que significa que tais atores também podem ser usados em domínios diferentes. O diretor controla a ordem de execução dos atores e também é responsável pela criação dos receptores.

A Figura 3.3 mostra um modelo hierárquico que contém dois domínios diferentes. O ator composto do nível mais alto possui um diretor D1 e dois atores A1 e A2, sendo que este último é outro ator composto com diretor D2, que controla a execução de A3 e A4 sempre que A2 é executado. O receptor na porta P1 controla a comunicação com a porta P2. De maneira similar, os receptores criados por D2 controlam a comunicação entre as portas P2 e P3, e entre P4 e P5 [9].

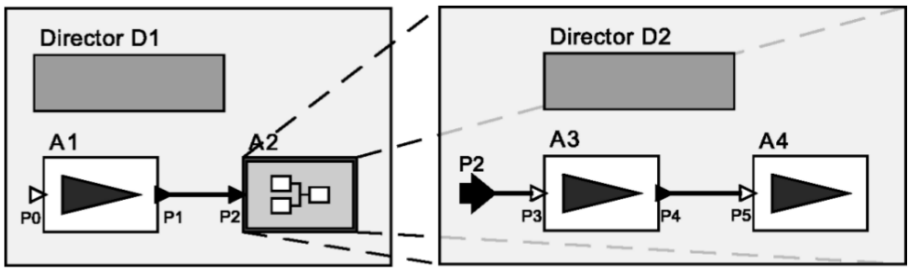


Figura 3.3: Modelo hierárquico no Ptolemy II [9]

## 3.3 ForSyDe e HetMoC

ForSyDe (*Formal System Design*) [52] é uma metodologia para a modelagem de sistemas que proporciona um conjunto de métodos transformacionais formais para um processo de refinamento do modelo de um sistema em um modelo de implementação otimizado para síntese.

A metodologia começa no domínio funcional, com o desenvolvimento de um modelo de especificação abstrato e formal. A verificação da funcionalidade se dá por simulação do modelo executável e/ou por meio de técnicas de verificação formal. O projetista, então, refina o modelo da especificação (etapa conhecida como *design refinement*), gerando um modelo de implementação a partir da transformações de projeto, que são selecionadas em uma biblioteca. Os objetivos do refinamento são otimizar o modelo de especificação e adicionar os detalhes necessários de implementação para permitir um mapeamento eficiente na arquitetura. A Figura 3.4 mostra o fluxo utilizado [10].

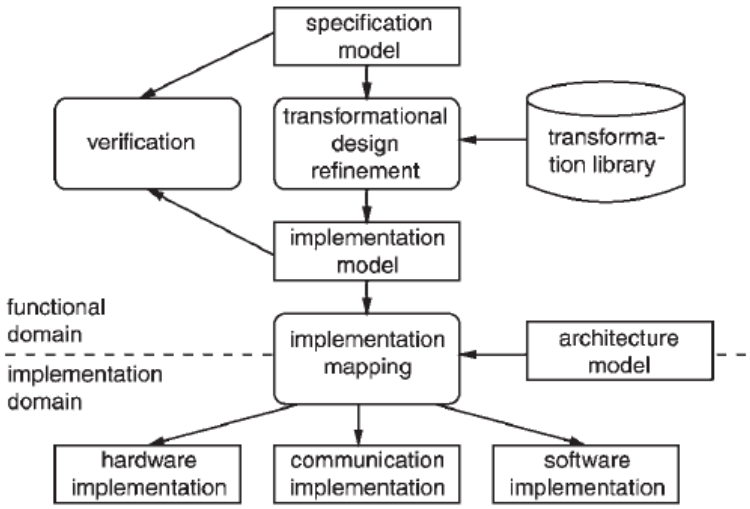


Figura 3.4: Fluxo de projeto no ForSyDe [10]

Foi definida formalmente a computação e a comunicação nos domínios *Continuous Time* (CT), *Discrete Event* (DE), *Synchronous/Reactive* (SR) e *Untimed*. Os processos heterogêneos se comunicam através de sinais implementados como canais FIFO polimórficos e, para integrar os domínios, foram definidas interfaces (DI - *domain interfaces*).

Para a modelagem de sistemas heterogêneos, foi proposto o HetMoC [11], um *framework* em SystemC que possui uma rede de processos heterogêneos que se comunicam por meio de sinais (canais FIFO) e no qual diferentes domínios são integrados por meio de DIs. Na Figura 3.5, os blocos representam os processos, que especificam a computação e as setas representam os sinais. Observa-se a utilização de DIs na comunicação entre os domínios.

Para validar o *framework* em SystemC, foi implementado um transceptor ASK (*Amplitude-Shift Keying*) como estudo de caso e as simulações foram comparadas com as realizadas em SystemC-AMS. Uma das vantagens é que, enquanto no último os domínios DE e SR somente podem ser descritos no MoC T-SDF (*Timed-Synchronous Data Flow*), o HetMoc permite a modelagem dos componentes de acordo com a especificação de seu domínio. Além disso, o HetMoC apresentou

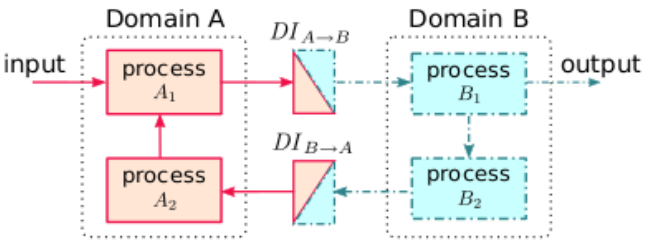


Figura 3.5: Rede de processos heterogêneos [11]

maior velocidade de simulação, o que é um fator importante na modelagem de sistemas maiores [11].

## 3.4 Trabalhos com SystemC-AMS

Conforme explicado no Capítulo 2, o SystemC-AMS é composto de extensões da linguagem SystemC com base na camada de sincronização T-SDF (*Timed Synchronous Data Flow*). Outros MoCs podem ser integrados em SystemC-AMS, e [26] apresenta alguns conversores, utilizando uma abordagem *ad-hoc*, para juntar os estilos de modelagem já existentes. No caso de modelos CT, os *solvers* utilizados são limitados aos de período de amostragem fixo, já que os modelos T-SDF possuem intervalos de tempo fixos.

Alguns trabalhos já apresentam exemplos de utilização do SystemC-AMS. Em [53], a linguagem SystemC-AMS é utilizada para a modelagem de componentes dinâmicos para a exploração da resposta em frequência e obtenção de um modelo descrito por uma função de transferência conhecida. É utilizado o MoC SDF (Synchronous Multi-Rate Data Flow Model of Computation) e um sensor químico é modelado como estudo de caso.

Em [54], são apresentados modelos de uma classe de dispositivos complexos não digitais - MEMS - com técnicas para integração para a simulação de um SoC heterogêneo em SystemC. Não foram utilizadas as extensões AMS. O MEMS, que consiste em um sensor para gases, foi modelado como um módulo SystemC padrão e integrado a uma plataforma com o processador 8051 via *on-chip interconnect*. O objetivo do trabalho foi modelar, além da funcionalidade, a potência, o desempenho e o comportamento térmico do MEMS.

Em [47], são definidos contextos, requisitos, objetivos e alguns trabalhos preliminares, mas nenhum resultado de simulação de SoC é apresentado. Já em [55], é dada uma visão geral da metodologia das extensões AMS. Há um primeiro protótipo, que é uma biblioteca que fornece um tipo de processo analógico ou de processamento de sinais. A execução desse processo não é controlada pelo *kernel* DE, e sim por uma interface.

Por fim, em [56] é feita uma primeira abordagem para conectar modelos TLM2 (*loosely timed*) com SystemC-AMS. A idéia é manter o bom desempenho na velocidade de simulação de ambos os MoCs utilizando conversores genéricos. Segundo os autores, apesar de haver vários trabalhos

relacionados com a modelagem de sistemas heterogêneos, havia escassez de trabalhos utilizando TLM e SystemC-AMS. Conforme será explicado no item a seguir, o trabalho apresentado nesta tese também busca suprir essa lacuna, oferecendo uma contribuição com modelos nos quais há uma interação entre TLM1/TLM2 e SystemC-AMS, embora com uma abordagem diferente e direcionada a aplicações de RSSF.

## 3.5 Considerações

Os ambientes e metodologias descritos neste capítulo constituem ferramentas importantes no auxílio a projeto de sistemas complexos que envolvem blocos heterogêneos. Todos foram desenvolvidos tendo em vista aspectos cruciais na modelagem de tais sistemas, como velocidade de simulação e interação entre diferentes domínios. Esses aspectos também foram levados em consideração nesta tese, mas alguns fatores foram determinantes para escolher a metodologia de implementação das plataformas virtuais dos SoCs ao invés de utilizar as soluções já existentes.

Foi realizada uma primeira padronização das extensões SystemC-AMS com uma definição completa e precisa das classes tendo em vista a sua utilização por desenvolvedores [57]. Isso, aliado à facilidade de comunicação com o kernel do SystemC e à boa performance de simulação, foi um fator determinante na sua escolha para descrever os blocos analógicos das plataformas virtuais modeladas neste trabalho. O HetSC, apesar de permitir a interação com SystemC-AMS, ainda é baseado na versão anterior à padronização, e o HetMoC ainda se encontra em fase de desenvolvimento, sendo que até o presente momento apenas um exemplo foi apresentado.

O Ptolemy permite a simulação de redes de sensores em um ambiente chamado VisualSense [58], no qual é possível descrever os nós com base nos atores ou definir o seu comportamento usando Java. Estão disponíveis também modelos específicos de canais e um *framework* para visualização. Apesar disso, optou-se pela utilização de SystemC TLM para a descrição dos blocos digitais dos SoC tendo em vista a facilidade de integração de modelos de processadores gerados pela ADL ArchC, de modelos em SystemC-AMS dos blocos analógicos, e da biblioteca SCNSL para simulações de RSSF. Assim, é possível compor nós com os modelos mistos dos SoCs, executar aplicações no ISS do processador dos mesmos e verificar os seus comportamentos em uma RSSF, conforme será explicado nos capítulos a seguir.

# Capítulo 4

# Metodologia

## 4.1 Metodologia Geral

Uma das estratégias para o desenvolvimento de sistems complexos, como é o caso de SoCs, é utilizar uma metodologia que envolve desde a especificação até o envio do *layout* para fabricação, no caso de ASICs, ou até a implementação em FPGAs. conforme mostra a Figura 4.1. Cada passo do fluxo envolve etapas de verificação funcional dos blocos utilizando ferramentas específicas.

As etapas do fluxo são descritas a seguir. É importante notar também que todo o projeto é voltado para a testabilidade, mas as etapas de verificação foram omitidas das figuras para manter a simplicidade.

- Especificação: o projeto começa com a definição da funcionalidade do sistema.
- Particionamento *hardware/software*: esta etapa visa buscar uma configuração otimizada da arquitetura do sistema.
- Modelagem em alto nível: aqui, são utilizadas linguagens de descrição de *hardware* que permitem a representação do sistema em um alto nível de abstração. Nesta fase, a plataforma em TLM funciona como uma referência para as equipes conduzirem o desenvolvimento de *software* e de *hardware*, simultaneamente. É possível também para a equipe de verificação e testes desenvolver o ambiente de verificação antes mesmo de ter o modelo RTL disponível.
- Desenvolvimento de *hardware*: durante a etapa de modelagem, os engenheiros de *hardware* desenvolvem a plataforma RTL. Neste caso, os fluxos tradicionais de projeto de *hardware* digital (ferramentas de síntese, *place and route*, análise de atrasos e verificação) e analógico (metodologia dedicada, usando as abordagens *top-down* e *bottom-up*) são seguidos normalmente.
- Desenvolvimento de *software*: ainda durante a etapa da modelagem, os engenheiros de *software* já podem desenvolver e testar o *software* embarcado utilizando a plataforma modelada para executar os programas.

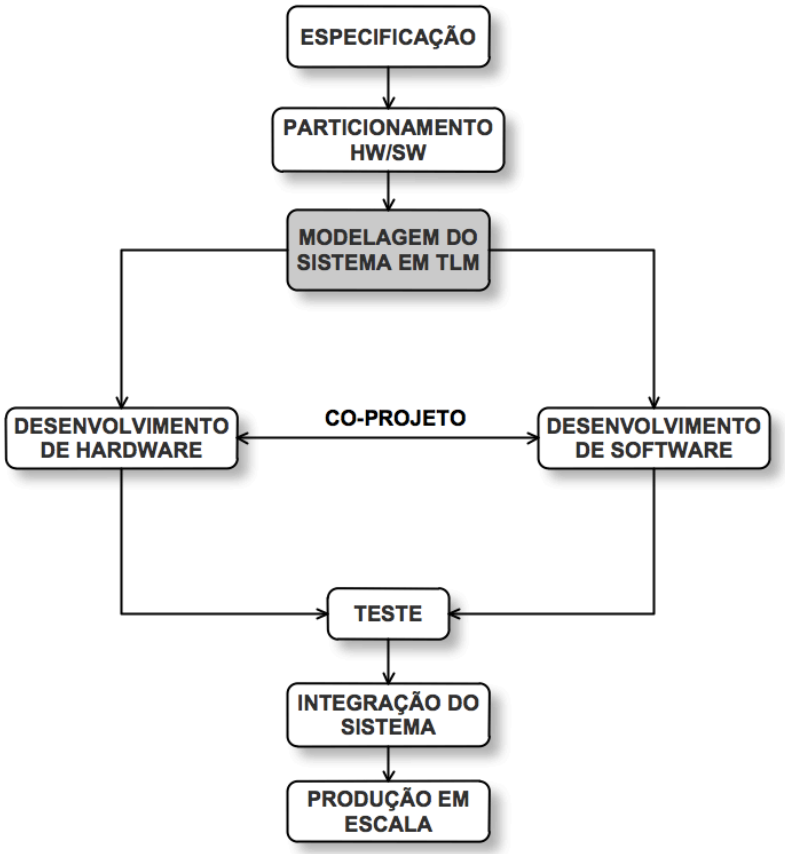


Figura 4.1: Fluxo de projeto de circuitos integrados [12]

- Teste, integração e produção: após os testes finais, o sistema é integrado e enviado para produção.

O foco deste trabalho é a etapa de modelagem em TLM. Descrições de sistemas partindo de níveis de abstração mais altos permitem uma visão geral do sistema mais rapidamente, pois diminuem a quantidade de detalhes durante a implementação e estão centradas na funcionalidade do sistema, simplificando o esforço de projeto e aumentando a velocidade de validaçao dos modelos. No nível de transação, a computação é descrita de modo comportamental, sem precisão em nível de ciclo de relógio, e a comunicação, sem detalhamento de protocolos.

As principais contribuições do trabalho estão concentradas na modelagem de sistemas em *chip* que, apesar de conterem blocos de *hardware* bem conhecidos, ainda estão em processo de desenvolvimento. O primeiro é um SoC com processador RISC de 16 bits, controlador de interrupções, memória e interfaces digital, analógica e de RF. O segundo é um RSoC com processador RISC de 32 bits baseado no MIPS, controlador de interrupções, barramento, memória, *timer*, ADC, sensor de imagem APS, blocos reconfiguráveis e transceptor de RF. Algumas simulações incluem ao menos um módulo analógico, descrito em SystemC-AMS, e possibilitam incluir um ambiente de rede no qual os SoCs funcionam como nós e se comunicam entre si e com outros nós funcionais. Para validação dos modelos, é simulada ao menos uma aplicação, ou parte de uma aplicação.

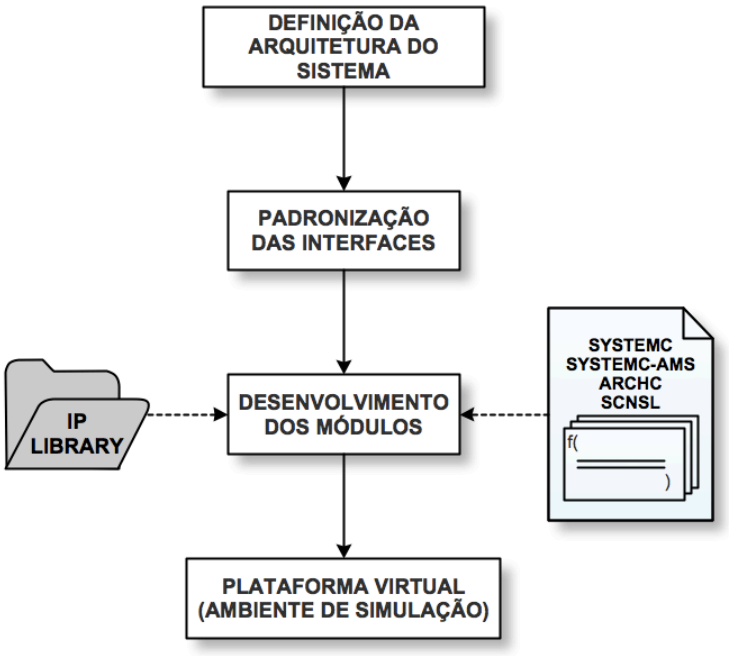


Figura 4.2: Fluxo adotado para modelagem

## 4.2 Metodologia da Modelagem

A Figura 4.2 mostra o detalhamento da etapa de modelagem do sistema em TLM do fluxo da Figura 4.1. Logo após a análise do particionamento *hardware/software* que tenha resultado em uma arquitetura otimizada, devem ser definidas as conexões entre os módulos para se determinar o fluxo de dados entre eles. Com isto, torna-se possível definir as interfaces dos módulos, o que constitui um passo importante pelo impacto que possui na portabilidade e reusabilidade dos modelos.

A partir dessas definições, é iniciado o desenvolvimento dos modelos, e é aqui que se encontram as características que tornam a metodologia flexível e abrangente. Em primeiro lugar, é possível a utilização de IPs desenvolvidos em projetos anteriores ou ainda de outras bibliotecas, bastando que sejam adaptados para o sistema. Essa característica amplia bastante o leque de possibilidades de escolha de módulos. A contribuição do presente trabalho se encontra na segunda característica, que é a utilização combinada de linguagens e extensões para a obtenção de plataformas que incluam blocos digitais, analógicos e interfaces que permitam a comunicação dos SoCs em redes sem fio.

A definição das linguagens a serem utilizadas para descrever os modelos se baseou na análise das características das mesmas. Conforme explicado nos capítulos 2 e 3, SystemC apresenta uma série de vantagens, tais como o fato de ser *open source* e ter a API (*Application Programming Interface*) TLM disponível para descrever os módulos em nível de transação.

O núcleo nativo do SystemC, entretanto, não permite a descrição e simulação de sistemas analógicos e contínuos no tempo. Para tais sistemas, pode-se utilizar linguagens de descrição de *hardware* como VHDL-AMS e/ou Verilog-AMS. Uma possível solução, inclusive, é realizar uma co-simulação de sistemas mistos utilizando SystemC e VHDL/Verilog-AMS. Entretanto, existem desvantagens nessa abordagem relacionadas à performance da simulação [24]. Por isso, as extensões

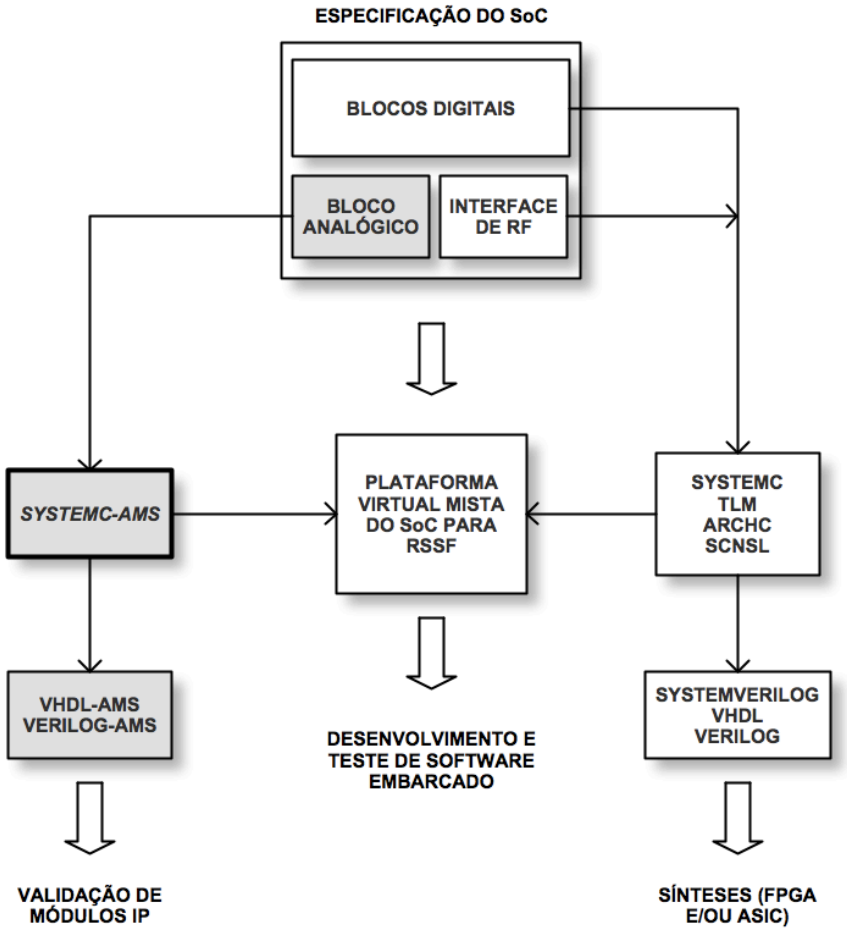


Figura 4.3: Utilização de SystemC-AMS para módulos analógicos

SystemC-AMS estão sendo introduzidas para tornar possível a elaboração de um *framework* para a modelagem funcional, exploração de arquitetura, integração, validação e prototipagem virtual de sistemas embarcados mistos [26].

As extensões AMS preenchem uma lacuna no fluxo de projeto ao inserirem uma abstração no nível de arquitetura. Com a versão SystemC-AMS1.0 [25], utilizada juntamente com SystemC, é possível realizar o co-projeto *hardware/software* do sistema com todos os seus elementos, conforme é mostrado na Figura 4.3. Os blocos digitais são modelados utilizando SystemC TLM e podem ser utilizados como base para a geração de modelos em RTL para síntese em FPGA ou ASIC. Os blocos analógicos são modelados em SystemC-AMS e, nesse caso, como não há processo de síntese, os mesmos podem ser utilizados como referência para a descrição de módulos IP em VHDL-AMS e/ou Verilog-AMS.

Com esta metodologia, uma biblioteca de modelos dos blocos em alto nível foi gerada para utilização em projetos futuros que envolvam aplicações de RSSF. O desenvolvimento dos módulos é feito utilizando uma combinação desses recursos. No caso dos SoCs descritos neste trabalho, foram utilizados:

1. SystemC TLM para os blocos digitais.

2. ArchC. O modelo do processador resultou da adaptação dos códigos gerados por essa ADL.

3. SystemC-AMS para o módulo analógico.

4. Biblioteca SCNSL para simulação do ambiente de rede.

O procedimento utilizado para instalação das ferramentas e simulação das plataformas virtuais está explicado nos anexos I e II, respectivamente. Foi criada uma biblioteca chamada **LDCI_Modeling_Library** com todos os blocos utilizados nas plataformas virtuais aqui apresentadas. Os blocos estao organizados em pastas: *ip* (IP *cores*), *processors* (processadores) e *wrappers* (blocos utlizados para conectar SystemC TLM1 e TLM2 com SystemC-AMS). Os arquivos se encontram no CD anexado à tese.

# Capítulo 5

# Descrição dos SoCs

*Este capítulo mostra a descrição das arquiteturas de dois Sistemas em Chip que foram modelados neste trabalho. O primeiro é um SoC com processador de 16 bits e o segundo, um RSoC com processador de 32 bits e blocos reconfiguráveis. Ambos estão voltados inicialmente para aplicações de RSSF.*

## 5.1 SoC

Este SoC foi inicialmente projetado para um sistema de controle de irrigação [59]. Seu projeto foi totalmente desenvolvido na universidade e foi tema de vários trabalhos de graduação e pós-graduação. Ele é composto por um processador RISC de 16 bits (RISC16) [60, 13], com uma unidade lógico-aritmética funcionando em ponto fixo [61], cuja proposta para adaptação para um módulo IP foi feita em [62]. As estruturas de armazenamento digital (memórias e banco de registradores) também foram projetadas, implementadas [63, 64] e algumas foram testadas com sucesso [65]. O *software* básico para o processador foi feito em [66] e um montador foi proposto e testado em [67, 68, 69]. As primeiras aplicações, como o programa de inicialização, foram feitas em [30] e um protocolo de comunicação foi desenvolvido e testado em [70, 31]. Um primeiro modelo feito em SystemC RTL foi proposto em [32] para permitir a execução de aplicações do SoC, mas, por conta das limitações encontradas, foi sugerida elaboração de um novo modelo em SystemC TLM.

As subseções a seguir descrevem os blocos do SoC.

### 5.1.1 Processador RISC16

Trata-se de um processador com arquitetura RISC de 16 bits cuja freqüência de operação foi definida em 16 MHz. A tecnologia de fabricação é a CMOS 0.35 um, com tensão de alimentação de 3.3V. Esta arquitetura foi descrita com detalhes em [13] e será resumida a seguir, incluindo algumas alterações. Como a modelagem do processador apresentada no Capítulo 6 foi feita de acordo com a descrição mostrada neste item, quaisquer aplicações a serem executadas no modelo

devem seguir este padrão.

O processador possui uma ULA que opera em ponto fixo e utiliza um somador do tipo *carry lookahead*, e um banco com 16 registradores (3 operandos em registradores de endereçamento). O conjunto de instruções pode ser dividido em três tipos:

- Tipo R: operação entre três registradores.
- Tipo I: operação entre dois registradores, envolvendo uma constante de 4 bits.
- Tipo J: operação entre um registrador e uma constante de 8 bits.

Os campos das instruções estão especificados abaixo:

- Operação (OP): codifica a instrução.
- Registradores fonte (REGS/REGS2): codificam os registradores fonte.
- Registrador destino (REGD): codifica o registrador destino.
- Constante (CONST): identifica um valor constante de 4 ou 8 bits.

Existem 16 instruções implementadas, entre lógicas, aritméticas, de transferência de dados e de desvios (condicionais e incondicionais). A Figura 5.1 mostra os formatos das instruções, e a Tabela 5.1 contém o conjunto proposto mostrado em [13], com algumas modificações na ordem dos registradores dos mnemônicos mostrados como exemplos.

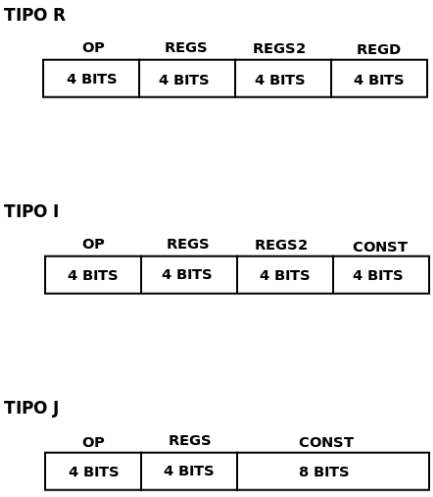


Figura 5.1: Formato das instruções [13]

A Tabela 5.2 mostra o endereçamento e a função dos 16 registradores do banco. Neste caso, a modificação efetuada foi a retirada do registrador $int e o acréscimo de mais um registrador salvo ($s4).

Este processador foi descrito, simulado e validado em VHDL [71], e sua implementação em FPGA foi apresentada em [72, 13].

Tabela 5.1: Conjunto de instruções do RISC16 [13]

| Cat. | Instr. | Cod. | Exemplo | Significado | Tipo |
|---|---|---|---|---|---|
| Aritmética | Add | 0010 | Add $s2,$s3,$s1 | $s1 = $s2+$s3 | R |
| | Sub | 0011 | Sub $s2,$s3,$s1 | $s1 = $s2 - $s3 | R |
| | Addi | 1000 | Addi $s1,100 | $s1 = $s1 + 100 | J |
| | Shift | 1001 | Sft $s1,8 | Se const>0, $s1=$s1»8; | |
| | | | | se const<0, $s1=$s1«8 | J |
| Lógica | And | 0100 | And $s2,$s3,$s1 | $s1 = $s2 and $s3 | R |
| | Or | 0101 | Or $s2,$s3,$s1 | $s1 = $s2 or $s3 | R |
| | Not | 1010 | Not $s1 | $s1 = not($s1) | J |
| | Xor | 0110 | Xor $s2,$s3,$s1 | $s1 = $s2 xor $s3 | R |
| | Slt | 0111 | Slt $s2,$s3,$s1 | Se $s2 < $s3, $s1 = 1 | R |
| Transf. | Lw | 0000 | Lw $s2,$s3,$s1 | $s1 = [$s2 + $s3] | R |
| | Sw | 0001 | Sw $s2,$s3,$s1 | [$s2 + $s3] = $s1 | R |
| | Lui | 1011 | Lui $s1,100 | $s1 = (8 MSB = const).(8 LSB de $s1) | J |
| Desvio | Beq | 1100 | Beq $s1,$s2,5 | Se $s1 = $s2, pc = pc + const | I |
| Cond. | Blt | 1101 | Blt $s1,$s2,5 | Se $s1 < $s2, pc = pc + const | I |
| Desvio | J | 1110 | J $s1,100 | pc = $s1 + const | J |
| Incond. | Jal | 1111 | Jal $s1,100 | pc = $s1 + const e $ra = origem | J |

Tabela 5.2: Registradores do RISC16 [13]

| Código | Símbolo | Função | Descrição |
|---|---|---|---|
| 0000 | $zero | Constante zero | Constante 0 de 16 bits |
| 0001 | $t0 | Temporários | Registradores auxiliares |
| 0010 | $t1 | | |
| 0011 | $t2 | | |
| 0100 | $a0 | Argumento | Argumentos para operações |
| 0101 | $a1 | | aritméticas e procedimentos |
| 0110 | $a2 | | |
| 0111 | $s0 | Salvos | Armazena valores durante |
| 1000 | $s1 | | chamadas de procedimento |
| 1001 | $s2 | | |
| 1010 | $s3 | | |
| 1011 | $s4 | | |
| 1100 | $gp | Apontador global | Aponta para variáveis globais na pilha |
| 1101 | $sp | Apontador pilha | Aponta para o topo da pilha |
| 1110 | $pc | Contador de programa | Aponta para a próxima instrução |
| 1111 | $ra | Endereço de retorno | Aponta o endereço de retorno de subrotina |

### 5.1.2 Memória

A especificação original em [13] prevê uma memória ROM de 2 kBytes, para armazenar a rotina de inicialização, e uma memória SRAM de 8 kBytes, para as aplicações. Para fins de simplificação, a memória modelada no Capítulo 6 é de apenas 1 KByte.

Algumas posições de memória são utilizadas como registradores para comunicação com a interface serial, com o ADC e com o transceptor de RF, conforme mostra a Tabela 5.3. Os endereços foram modificados para corresponderem aos utilizados no modelo.

Tabela 5.3: Registradores mapeados em memória [13]

| **Endereço** | **Interface** | **Descrição** |
|---|---|---|
| 1FEA | ADC | Dados (RX) |
| 1FEC | ADC | Setup |
| 1FEB | ADC | Status |
| 1FFA | Serial | Setup |
| 1FED | Serial | Dados (RX) |
| 1FEE | Serial | Dados (TX) |
| 1FEF | Serial | Status |
| 1FFB | RF | Dados (RX) |
| 1FFC | RF | Dados (TX) |
| 1FFD | RF | Status |
| 1FFF - 1FFE | RF | Setup |
| 1FF2 | RegInt | Salva o PC quando há interrupção |
| 1FF3 | IntCausa | Tipo de interrupção |
| 1FF1 | RegStatus | Salva o status da ULA |

A posição 1FF1 corresponde ao RegStatus, que armazena as *flags* do sistema. Este registrador é atualizado ao final das instruções lógico-aritméticas. Novamente, a especificação original foi alterada, conforme mostra a Tabela 5.4.

Tabela 5.4: Campos de RegStatus

| **Campo** | **Significado** |
|---|---|
| Bit0 | Z: resultado da operacao da ULA é nulo. |
| Bit1 | N: resultado da operacao da ULA é negativo. |
| Bit2 | C: resultado da operacao na ULA gerou carry. |
| Bit3 | Caux: houve carry no meio da palavra. |
| Bit4 | Indica que ocorreu um overflow. |

As posições 1FF2 e 1FF3 são utilizadas pelo gerenciador de interrupções, descrito na subseção a seguir.

### 5.1.3 Gerenciador de Interrupções

Foram previstas duas sinalizações de erros (exceções) e três tipos de interrupção. A primeira exceção ocorre quando há um *overflow* em uma das operações da ULA, e a segunda, quando há um erro de endereçamento de memória. As interrupções ocorrem quando uma das interfaces (ADC, RF ou serial) envia um dado para o processador.

A posição 1FF2 é chamada de RegInt e armazena o endereço da instrução que estava sendo executada quando a interrupção/exceção ocorreu, enquanto que a posição 1FF3 é chamada de IntCausa e salva o tipo de interrupção. Os valores mostrados na Tabela 5.5 foram alterados a partir da especificação original e utilizados na modelagem do gerenciador.

Tabela 5.5: Campos de IntCausa

| Campo | Significado |
|---|---|
| Bit0 | Indica se a interrupção está habilitada (0) ou desabilitada (1) |
| Bit1 | Indica interrupção requisitada pelo ADC, quando igual a 1 |
| Bit2 | Indica interrupção requisitada pela interface de RF, quando igual a 1 |
| Bit3 | Indica interrupção requisitada pela interface serial, quando igual a 1 |
| Bit4 | Indica *Overflow*, quando igual a 1 |
| Bit5 | Indica erro de endereçamento, quando igual a 1 |

O tratamento das interrupções se dá por meio de três operações: verificação da ocorrência de interrupção ao final de cada ciclo de instrução, armazenamento do endereço da instrução em execução e do tipo de interrupção gerada em registrador (no caso da existência de um pedido de interrupção), e, por fim, a transferência da execução do programa para um endereço de memória predeterminado, onde uma instrução de desvio encaminhará a execução para a rotina de tratamento da interrupção.

### 5.1.4 Transceptor de RF

A partir de uma arquitetura do transceptor de RF descrita para o SoC em [59], foi proposta uma nova arquitetura em [73]. Visando manter a simplicidade do sistema, a arquitetura foi modificada para uma topologia com baixa freqüência intermediária. A Figura 5.2 mostra o diagrama de blocos para esta nova proposta, que ainda poderá ser otimizada.

O transceptor opera na banda ISM (915MHz-927MHz) utilizando modulação FSK binária para permitir uma comunicação *half-duplex* de 50 kbps. A seção RX é implementada como uma arquitetura própria para reduzir o consumo de potência, minimizar os efeitos de ruído *flicker* e permitir a demodulação com o uso de um circuito digital simples. Uma possível solução para a seção TX seria utilizar circuitos digitais para gerar um sinal 2-FSK em banda base que será diretamente convertido para a banda RF, também para atender os requisitos de baixo consumo.

Foram previstos dois registradores de controle, utilizados para configurar parâmetros como o canal de comunicação, a potência de transmissão, modo de operação, etc. Além destes, outros 16

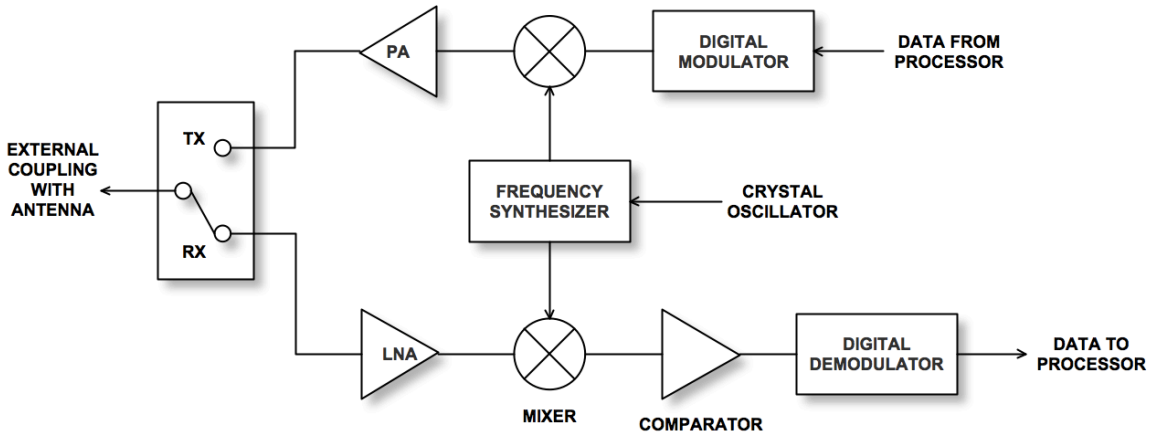


Figura 5.2: Nova proposta de arquitetura para o transceptor de RF

registradores foram previstos para comportar os pacotes de 128 bits a serem enviados ou que foram recebidos.

Para viabilizar o uso de várias taxas de transmissão, a sincronização do sistema de comunicação se torna mais difícil. Foi introduzida a utilização de um código Manchester por ser simples na codificação e decodificação, evitar longas seqüências de zeros e uns consecutivos e transmitir informação sobre o *clock* utilizado na codificação em conjunto com a informação. Sua desvantagem é a de dobrar a banda necessária a comunicação.

Mais detalhes sobre esta arquitetura podem ser encontrados em [73, 74, 75, 76, 77] e não serão abordados aqui.

### 5.1.5 ADC

Como as aplicações envolvem o sensoriamento de ambientes, um ADC (*Analog-to-Digital Converter*) foi incluído como interface com sensores externos. O conversor proposto possui arquitetura cíclica operando em modo de corrente. O dispositivo trabalha a uma taxa de conversão de 50 mil amostras por segundo, gerando 8 bits para cada amostra, e com um número efetivo de bits esperado maior ao igual a 7,5. Este ADC foi projetado e implementado em tecnologia CMOS 0,35um e seu projeto e implementação foram detalhados em [78].

A interface analógica completa também foi descrita em VHDL-AMS como um estudo de caso para mostrar como a modelagem de circuitos analógicos e mistos utilizando linguagens de descrição de *hardware* pode ser um passo importante na geração de blocos IP. Os resultados deste trabalho são mostrados em [79, 80].

### 5.1.6 Interface Serial

A primeira proposta de uma interface serial foi feita e simulada em VHDL em [81]. Posteriormente, uma serial UART para uso isolado ou integrado a outros módulos IP foi encontrada em [82] e foi adaptada para teste e possível uso posterior em conjunto com os SoCs. Ela é basicamente uma

máquina de estados, que organiza como a transmissão e a recepção devem funcionar. Ativando o sinal de leitura em nível alto, a interface habilita a recepção de dados. Ao fim da recepção, este bloco retorna ao estado inicial. Quando o sinal de escrita está em nível alto, a interface está pronta para enviar dados, voltando ao estado inicial novamente quando o fim da transmissão acontece. Tanto a escrita quanto a leitura geram sinais de interrupções em nível alto.

A função do bloco de transmissão é apenas transmitir a uma taxa fixa de 9,6 KHz os bits recebidos do processador. Para transmitir, é necessário apenas colocar um pulso em nível lógico alto no sinal de habilitação da transmissão e a unidade transmitirá, obedecendo à taxa de envio do *clock* da transmissão, no caso, 9,6KHz.

Deve-se observar que esta unidade utiliza o protocolo RS232, o que significa que, durante o estado de espera, a unidade de transmissão mantém o sinal de saída em nível lógico alto. Quando uma transmissão deve ter início, a unidade envia um primeiro bit de nível lógico baixo, chamado bit de início, indicando início da transmissão. Em seguida, a seqüência de bits é transmitida. Por fim, a unidade envia o último bit em nível lógico alto, chamado bit de parada, para indicar o fim da transmissão. O fim da transmissão é indicado com um pulso em nível alto no sinal indicador de fim de transmissão.

A unidade de recepção é responsável por receber os bits serialmente. A recepção só ocorre quando o sinal que habilita a recepção está ativo em nível lógico alto. A taxa de recepção é 16 vezes maior (153KHz) do que a taxa de transmissão. Tal característica é necessária para permitir uma recepção com uma quantidade menor de erros de bits. Ao fim da recepção, um pulso em nível lógico alto é enviado pelo sinal indicativo de fim de recepção.

## 5.2 RSoC

A Figura 5.3 mostra os componentes do RSoC-32. Este sistema é constituído por um processador RISC 32-bits com gerenciador de interrupções implementado, barramento, blocos reconfiguráveis (RoSA), memórias, *timer*, sensor de imagem APS e interfaces de comunicação. A funcionalidade do sistema será configurada por *software* e o sistema possuirá meios para a aquisição de dados analógicos e para o acionamento de subsistemas analógicos e digitais, inclusive sensores, integrados ou não, ao mesmo *chip*.

As interfaces de RF, digital e analógica do RSoC são as mesmas definidas na descrição do SoC-16 na seção anterior e, portanto, suas descrições não serão repetidas aqui. As subseções a seguir contêm uma descrição geral do processador, memórias, RoSA e do sensor de imagem.

### 5.2.1 Processador e Memórias

O processador possui arquitetura RISC de 32 bits e um conjunto de instruções compatível com o do MIPS. A idéia é obter um processador eficiente e com baixo consumo de potência, e também utilizar ferramentas de desenvolvimento já existentes para o MIPS.

Um processador MIPS consiste em uma unidade central de processamento (CPU) e uma coleção

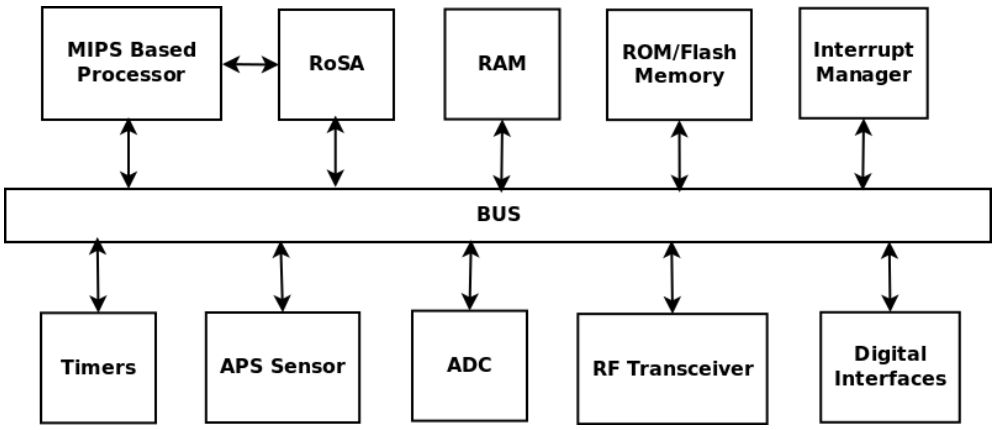


Figura 5.3: Diagrama de blocos do RSoC

de co-processadores que executam tarefas auxiliares ou operam sobre outros tipos de dados. Por exemplo, o co-processador 0 gerencia *traps*, exceções e o sistema de memória virtual, enquanto o co-processador 1 é a unidade de ponto flutuante.

O MIPS é uma arquitetura *load/store*, ou seja, somente essas instruções possuem acesso à memória. As instruções da ULA operam somente com valores em registradores. Os 32 registradores de uso geral esto descritos na Tabela 5.6.

Tabela 5.6: Registradores do MIPS e descrição

| **Registrador** | **Descrição** |
|---|---|
| $0 | Valor 0 (*hardwired*) |
| $at | Uso do montador |
| $k0 e $k1 | Uso do sistema operacional |
| $v0 e $v1 | Retornam valores de funções |
| $a0 a $a3 | Passagem dos argumentos 1 - 4 |
| $t0 a $t9 | Dados temporários |
| $s0 a $s7 | Dados que necessitam ser preservados durante as chamadas |
| $gp | Ponteiro global |
| $sp | Ponteiro da pilha |
| $fp | Ponteiro de *frame* |
| $ra | Endereço de retorno |

O gerenciador de interrupções recebe todos os pedidos gerados pelo sistema, chama a rotina de tratamento de interrupção e armazena o *status* em registradores, indicando quais eventos ocorreram. O programador pode implementar diferentes tipos de rotinas em *software*, utilizando a informação do registrador de *status*.

Após a especificação final, pretende-se utilizar memórias fornecidas pelos fabricantes, que disponibilizam os códigos através de ferramentas *memory compiler* e vendem os *layouts* separadamente. Além de memórias RAM e ROM, é possível também o uso de uma memória *flash* externa para aumentar o limite de armazenamento e permitir o uso de rotinas maiores.

### 5.2.2 RoSA

RoSA (*Reconfigurable Stream-Based Architecture*) [14] é uma arquitetura reconfigurável de granularidade grossa que combina técnicas de compilação e reuso de *hardware* para acelerar a execução de aplicações baseadas em fluxo de dados. A Figura 5.4 apresenta o diagrama de blocos da arquitetura.

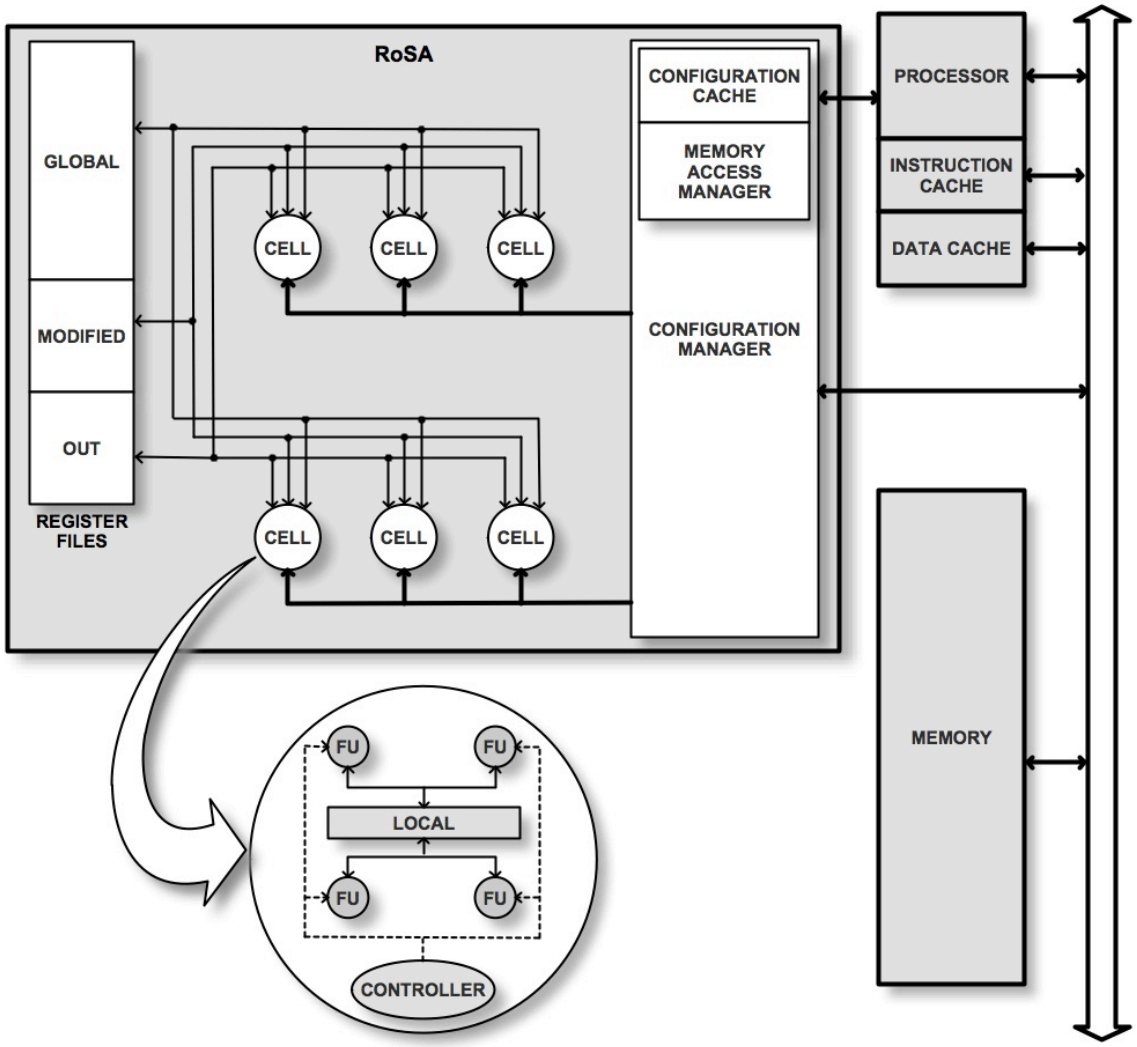


Figura 5.4: Arquitetura do RoSA [14]

A arquitetura consiste de seis unidades reconfiguráveis (células), um gerenciador de configuração, uma *cache* de configuração, interfaces para barramentos e registradores (globais, modificados, locais e de saída). Esses blocos são descritos a seguir.

1. Células. Cada célula é uma unidade reconfigurável que possui 4 FU (unidades funcionais) e executa um subgrafo independente. As células executam operações aritméticas e lógicas e se comunicam entre si através do registrador local. Até quatro operações podem ser executadas simultaneamente.

2. Registradores. Há quatro tipos:

- Globais: os dados são acessados pelas células, mas não podem ser modificados.
- Modificados: a informação é lida pela célula e modificada pelo processador.
- Saída: armazenam as saídas das células, que podem ser lidas por todas as outras.
- Locais: aqui é armazenada toda a computação intermediária.

3. Gerenciador de configuração. É responsável por buscar a configuração na *cache* e distribui-la entre as células. Ele também gerencia indica quais registradores devem ser acessados pelas células e gerencia a saída das mesmas.

4. Gerenciador de acesso à memória. É o componente responsável pelas requisições da memória. Também recebe as requisições de *read/write* das células, armazena-as em uma fila e mantém a coerência entre a memória e a *cache* quando as células requisitam acesso.

A configuração de cada célula é feita usando uma palavra de 120 bits. Sendo assim, é necessária uma palavra de 720 bits para configurar as seis células. A Figura 5.5 mostra a sintaxe de configuração da célula e o número de bits de cada campo. Também mostra a estrutura do grafo correspondente e as operações em cada nível.

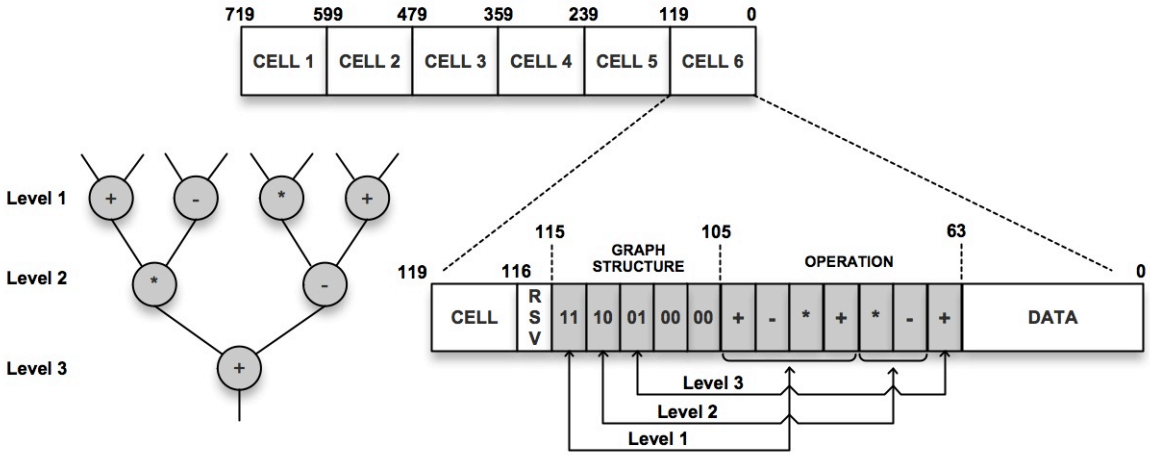


Figura 5.5: Configuração do RoSA

Cada configuração deve ser gerada para um conjunto de MISOs (*Multiple Inputs, Single Output*) que será executado no RoSA. A sua execução na arquitetura é organizada de acordo com os campos da palavra de configuração. O campo *Operation* contém as operações da FUs que serão efetuadas nos dados de entrada (*Data*) de acordo com a estrutura dos subgrafos (*Graph Structure*). No exemplo mostrado na Figura 5.5, pode-se ver a configuração de uma célula (*CELL 6*) para executar as operações mostradas no subgrafo: duas somas, uma subtração e uma multiplicação no nível 1 (*Level 1*), uma subtração e uma multiplicação no nível 2 (*Level 2*) e uma soma no nível 3 (*Level 3*).

Alguns estudos de caso foram feitos em [14] considerando os tempos de execução de quatro aplicações: FFT (*Fast Fourier Transform*), sua inversa IFFT (*Inverse Fast Fourier Transform*), codificação JPEG e DCT (*Discrete Cosine Transform*). Os estudos mostraram altas taxas de

paralelismo. Em alguns casos, houve um aumento de desempenho de até 55% em relação ao tempo de execução das mesmas aplicações rodando em um processador de propósito geral.

### 5.2.3 Sensor de Imagem

O circuito APS, do inglês, *Active Pixel Sensor* [83, 84], apresenta como principais vantagens o custo reduzido, devido à tecnologia, e a possibilidade de implementação do circuito em conjunto com circuitos digitais, microprocessadores, memórias e circuitos analógicos.

Por definição, um pixel ativo contém um fotodiodo e um ou mais transistores amplificadores. Os transistores extras são usados para prover ganho ou para carregar o pixel da grande capacitância de coluna. O pixel ativo padrão é constituído de um transistor de *reset*, um seguidor de fonte e um transistor de leitura, que permite que cada pixel seja endereçado individualmente. O transistor de leitura pode ser incluído acima ou abaixo do transistor seguidor de fonte. A operação de ambos os pixels é comparada e a escolha entre as duas arquiteturas é feita pela facilidade no *layout* e para reduzir as capacitâncias parasitárias. Existem muitas outras arquiteturas diferentes para os pixels que visam à melhoria de fatores como diminuição da área e aumento da velocidade de leitura do pixel.

Geralmente, as arquiteturas para o sensor são ditadas pela arquitetura adotada pelo pixel, que determina os sinais necessários para o controle de cada um. Apesar disto, em todas as arquiteturas encontra-se amplificadores de coluna, um circuito de endereçamento das linhas e um conversor analógico-digital, conforme mostra a Figura 5.6.

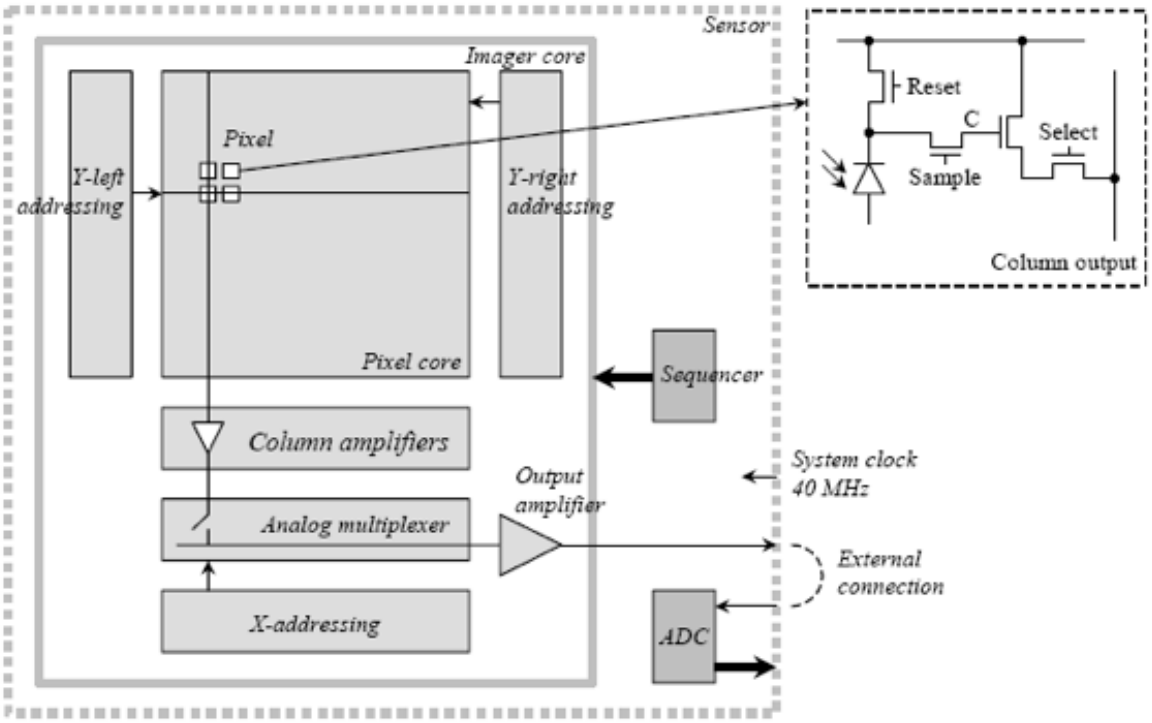


Figura 5.6: Arquitetura de um sensor APS

Considerando essas características, decidiu-se utilizar uma arquitetura projetada por pesquisadores da UFRJ, cujo protótipo foi desenvolvido em tecnologia CMOS 0,35 um C35B4 [85]. A matriz de pixels APS operacional contém 64x64 pixels. Cada pixel pode ser endereçado individualmente, seja para comandar a leitura do pixel, seja para forçar a inicialização (*reset*) da tensão do fotodiodo do pixel. Também é possível inicializar todos os pixels da matriz simultaneamente.

A lógica de endereçamento e controle permite que a matriz operacional seja endereçada tanto de modo estático quanto de maneira dinâmica e automática, através do gerador automático de endereços.

A aquisição de imagens utilizando a matriz APS operacional obedece a uma seqüência de operações de ordem bem definida. Inicialmente, deve-se comandar a inicialização (*reset*) dos pixels, que pode ser global ou por linha e, após um intervalo de tempo, é realizada a leitura da tensão de saída do pixel endereçado. O endereçamento da matriz pode ser executado remotamente via um dispositivo externo, ou de forma automática utilizando o mecanismo de endereçamento automático interno.

O capítulo a seguir mostra como foi efetuada a modelagem dos blocos do SoC e RSoC aqui descritos.

# Capítulo 6

# Modelagem dos SoCs

*Conforme exposto nos capítulos anteriores, uma maneira rápida e de baixo custo computacional é elaborar modelos em alto nível dos sistemas. Para isto, além de modelar os blocos separadamente, é preciso também definir e validar suas interfaces e estruturas de comunicação. Este capítulo mostra a modelagem do SoC e do RSoC apresentados no Capítulo 5.*

## 6.1 Plataforma Virtual SoC

Esta plataforma virtual corresponde ao modelo do SoC descrito no Capítulo 5. Este sistema já foi implementado em ASIC e o processador foi simulado em VHDL e implementado em FPGA. Dessa forma, por ter o respaldo de validações anteriores, foi escolhido como estudo de caso para aplicar a metodologia proposta neste trabalho.

Em SystemC, os processos descrevem a funcionalidade arquitetural dos componentes de processamento. Já os canais descrevem a semântica de comunicação, abstraindo a complexidade de protocolos e permitindo a transmissão de tipos de dados desejados.

A comunicação entre os blocos que formam o sistema é feita utilizando, além de sinais, o conceito de *socket*, introduzido pela versão 2.0 de TLM [21]. Os *sockets* funcionam baseados na existência de dois caminhos de dados, *forward* e *backward*. O caminho *forward* é utilizado pelo iniciador (*sockets* marcados com I), para iniciar as transações com os dispositivos alvo (*sockets* marcados com T). O caminho *backward* é usado pelo alvo para responder as transações iniciadas. A Figura 7.1 mostra o diagrama de blocos da plataforma virtual SoC modelada.

O modelo do sistema é puramente funcional e só faz anotações de tempo configuráveis para fornecer ao programador de aplicações um meio para avaliar o desempenho do sistema quando executando uma aplicação. A seguir, serão mostrados detalhes da implementação dos blocos da Figura 7.1.

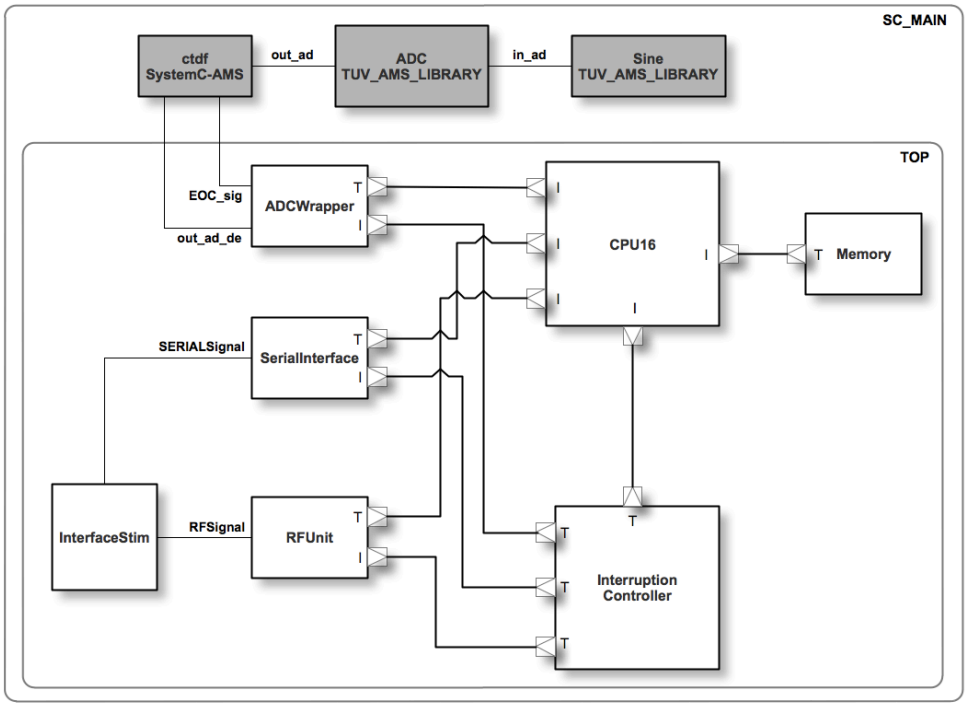


Figura 6.1: Diagrama de blocos do SoC com o ADC em SystemC-AMS

### 6.1.1 Processador

O algoritmo utilizado para a modelagem do processador é mostrado na Figura 6.2 e os passos são descritos a seguir.

1. Na leitura de memória, o algoritmo acessa o PC (variável interna representada por *regBank[14]*) para depois iniciar uma transação e ler o endereço de memória correspondente.

2. O PC é incrementado (PC = PC + 1). A Listagem 6.1 mostra o código para este passo e o anterior. Pode-se observar que é utilizada uma *flag* para testar se houve uma operação de salto (*branchOp*).

```
if(branchOp){
    unsigned int pcAux = (unsigned int)regBank[14];
    tlm::tlm_command command = tlm::TLM_READ_COMMAND;
    branchOp=false;
    memoryReadWrite(memInSocket,command,instrAux,pcAux);
    regBank[14]=regBank[14]+1;
}else{
    unsigned int pcAux = (unsigned int)regBank[14];
    tlm::tlm_command command = tlm::TLM_READ_COMMAND;
    memoryReadWrite(memInSocket,command,instrAux,pcAux);
    regBank[14]=regBank[14]+1;
}
```

Listagem 6.1: Busca da instrução na memória

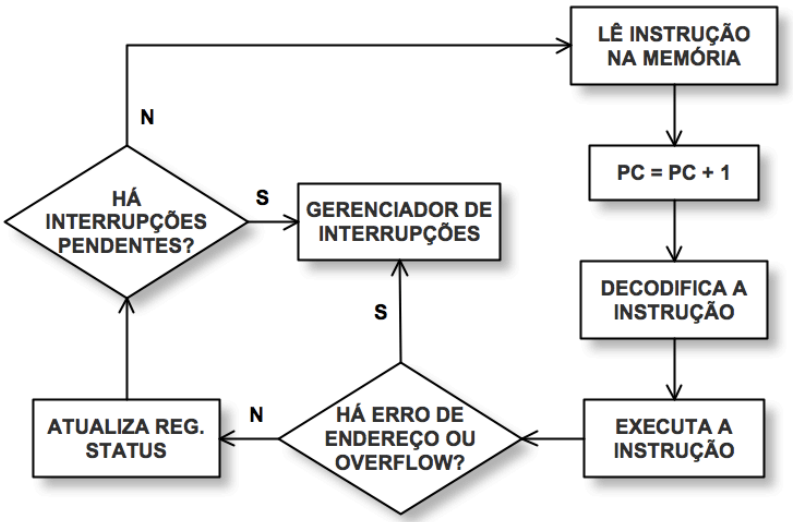


Figura 6.2: Algoritmo de modelagem do RISC16

3. É feita a decodificação da instrução atual. O algoritmo separa os campos da instrução nos seus componentes básicos (*op, regs, regs1, regs2*), para que a mesma possa ser executada. A Listagem 6.2 mostra o código correspondente.

```
operation=((sc_uint<16>)instrAux).range(15,12);
reg2=((sc_uint<16>)instrAux).range(11,8);
reg3=((sc_uint<16>)instrAux).range(7,4);
reg1=((sc_uint<16>)instrAux).range(3,0);
```

Listagem 6.2: Decodificação da instrução

4. Após a extração dos campos das operações, o algoritmo ingressa em uma FSM que, dependendo do campo operação, toma as ações necessárias para a execução da instrução (modificação do PC, do banco de registradores e execução), conforme mostra a Listagem 6.3.

```
switch(operation)
{
    case 2:  //Add: Add $s2,$s3,$s1 (adds $s2 to $s3 and stores the result in
        $s1)
        value1Aux = scuintConv(regBank[reg2]);
        value2Aux = scuintConv(regBank[reg3]);
        valueTemp = regBank[reg2]+regBank[reg3];
        valueTotal= scuintConv(valueTemp);
    (...)
    break;
    (...)
    case 0:  //Lw: Lw $s2,$s3,$s1 ($s1 = [$s2 + $s3])
        command = tlm::TLM_READ_COMMAND;
        value1 = regBank[reg2]+regBank[reg3];
        unsigned int aux1,aux2;
        aux1 = (unsigned int)value1;
```

```
        memoryMapping(command,aux2,aux1);
        regBank[reg1]=(sc_uint<16>)aux2;
    (...)
    break;
    (...)
}
```

Listagem 6.3: Execução da instrução

5. O algoritmo avalia a resposta da transação para determinar se houve um erro de endereço ou de *overflow*. No caso de erro, o algoritmo registra o mesmo no gerenciador de interrupções para ser atendido logo depois. Na Listagem 6.4, pode-se observar que o *initiator socket interruptInSocket* faz uma chamada ao método *b_transport* passando como parâmetros um atraso de 10 ns e o *generic payload transInt*.

```
if(_error_address or _error_overflow)
{
    unsigned int addrAux,dataDummy;
    dataDummy=0;
    if(_error_address)
        addrAux=5;
    else if(_error_overflow)
        addrAux=4;

    sc_time delay = sc_time(10,SC_NS);
    transInt->set_command( tlm::TLM_WRITE_COMMAND );
    transInt->set_address( addrAux );
    transInt->set_data_ptr( reinterpret_cast<unsigned char*>(&dataDummy));
    transInt->set_data_length( 1 );

    interruptInSocket->b_transport( *transInt, delay );
    (...)
}
```

Listagem 6.4: Verificação da ocorrência de exceções

6. O registrador de *status* deve ser atualizado após a execução de cada instrução, no caso de operações lógicas e aritméticas. Ele é interno ao algoritmo, e por isso a atualização é feita escrevendo-se diretamente, como mostra a Listagem 6.5.

```
regStatus.range(15,5)=0;
regStatus.bit(4)=_error_overflow;
regStatus.bit(3)=carryAux;
regStatus.bit(2)=carry;
regStatus.bit(1)=negative;
regStatus.bit(0)=zero;
```

Listagem 6.5: Atualização do RegStatus

7. O algoritmo inicia uma transação com o gerenciador de interrupções para verificar se há novas interrupções. Caso haja, o PC é salvo em RegInt, para que, depois que a interrupção for atendida, o mesmo retorne ao ponto em que o programa foi interrompido, e modificado para apontar para o endereço 0x000h, onde fica o endereço da rotina de tratamento de interrupção. A Listagem 6.6 mostra o trecho de código correspondente a este passo.

```
sc_time delay = sc_time(10,SC_NS);
intCheck->set_command( tlm::TLM_READ_COMMAND );
intCheck->set_address( 0 );
intCheck->set_data_length( 1 );

interruptInSocket->b_transport( *intCheck, delay );

if(intCheck->get_response_status()==tlm::TLM_OK_RESPONSE)
{
    unsigned int pcIntAux = regBank[14]-1;
    unsigned int addrAux = REG_INT;
    tlm::tlm_command cmdAux = tlm::TLM_WRITE_COMMAND;
    memoryMapping(cmdAux,pcIntAux,addrAux);
    regBank[14]=0;
}
```

Listagem 6.6: Verificação de novas interrupções

A subseção a seguir descreve o modelo do gerenciador de interrupções.

### 6.1.2 Gerenciador de Interrupções

O gerenciador de interrupções recebe os pedidos de interrupção do processador (por erro de endereço ou de *overflow*), da interface serial, da interface de RF e do ADC. A cada pedido de interrupção, o gerenciador avalia primeiramente se há outra requisição ainda não atendida. Caso haja alguma interrupção pendente, então o gerenciador ignora o pedido até que a mesma seja atendida; caso contrário, o gerenciador verifica se as interrupções estão habilitadas (ou seja, se o Bit0 de *IntCausa* é igual a zero), armazena a causa em *IntCausa* de acordo com a configuração mostrada na Tabela 5.5, salva o dado recebido no devido registrador (vide Tabela 5.3) e desabilita temporariamente as interrupções. A Listagem 6.7 mostra o algoritmo utilizado.

```
if ( cmd == tlm::TLM_READ_COMMAND ){
   if(_busy){
      trans.set_response_status( tlm::TLM_OK_RESPONSE );
         _busy=false;
         return;
   }else{
         trans.set_response_status( tlm::TLM_GENERIC_ERROR_RESPONSE );
         return;
   }
}else if ( cmd == tlm::TLM_WRITE_COMMAND ){
```

```
    unsigned int* aux_ptr = (unsigned int*)ptr;
    if(adr==INT_CAUSA){
        intCausa = *aux_ptr;
        if(intCausa==0)
        _busy=false;
    }else if(adr==REG_INT){
        regInt = *aux_ptr;
    }else if(!(intCausa.bit(0))){
            if(not _busy){
            intCausa.bit(adr)=1;
                data_rx=*aux_ptr;
                _busy=true;
                if(_debug)
                    printf("INTERRUPT CONTROLLER : intCausa=%x    data_rx=%x\n",(
                        unsigned int)intCausa,(unsigned int)data_rx);
            }else{
                trans.set_response_status( tlm::TLM_COMMAND_ERROR_RESPONSE );
                    return;
            }
    }else{
            _busy=false;
    }
}
```

Listagem 6.7: Algoritmo do gerenciador de interrupções

### 6.1.3 ADC

Trata-se de um módulo analógico e por isso foi modelado em SystemC-AMS. Conforme relatado no Capítulo 2, as extensões AMS do SystemC permitem a modelagem em *Timed Data Flow* (TDF), cujas simulações são mais rápidas devido ao fato do agendamento ser estático e, portanto, ser realizado antes do início da simulação. A biblioteca aberta *TUV_AMS_LIBRAY* [86] possui blocos em SystemC-AMS voltados para a área de sistemas de comunicação em radio-frequência, em particular em fontes de sinais, blocos de modulação/demodulação, filtros, e partes de medidas e observação. Eles podem ser divididos em três categorias: fontes de sinais, blocos básicos para processamento de sinais e unidades de análise de sinais. Dessas categorias, os blocos gerador de funções senoidais e um ADC foram escolhidos e adaptados para integrarem o modelo do SoC, sendo que o primeiro gera estímulos para o segundo.

O gerador de funções senoidais é um módulo TDF que possui como saída uma senóide cujos parâmetros, que são frequência, amplitude, *offset* e fase inicial, podem ser ajustados. A frequência e a amplitude podem ser alteradas durante a simulação por meio de duas portas adicionais. A Listagem 6.8 mostra as funções *set_attributes*, *initialize* e *processing* utilizadas pela classe.

```
void sine::set_attributes(){
    //atribui data rate à porta de saída
    out.set_rate(rate);
}
```

```
void sine::initialize(){
   //lê o time step da porta de saída
   sample_time = out.get_timestep().to_seconds();
}

void sine::processing(){

   if(freq_recon==true)
      //lê a frequência na porta de entrada
      freq_reg=freq_in->read();

   if(ampl_recon==true)
      //lê a amplitude na porta de entrada
      amp_reg=amp_in->read();

   // calcula o passo de acordo com o time step da porta de saida
   stepsize = sample_time*freq_reg*2.*M_PI;

   for(int i=0; i<rate; i++){
      // escreve dado (data token) para 1 bit
      out.write(amp_reg*sin( actval + phi )+offset,i);
      actval+=stepsize;
   }
}
```

Listagem 6.8: Algoritmo do gerador de funções senoidais

A Tabela 6.1 mostra os valores que foram utilizados no gerador conectado ao ADC da plataforma virtual SoC.

Tabela 6.1: Parâmetros do gerador de funções senoidais da plataforma virtual SoC

| **Parâmetro** | **Tipo** | **Valor** | **Descrição** |
|---|---|---|---|
| nm | sc_module_name | Sine | Nome do módulo |
| freq_def | double | 111111.1 | Frequência inicial |
| amp_def | double | 1.0 | Amplitude inicial |
| _phi | double | 0.0 | Fase inicial |
| _offset | double | 0.0 | Offset da saída |
| amp_c | bool | false | Amplitude configurável |
| freq_c | bool | false | Frequência configurável |
| datarate | integer | 1 | Incrementa o data rate |

Observa-se que os recursos de amplitude e frequência configuráveis não foram utilizados. A amplitude foi fixada em um valor unitário e a frequência foi ajustada para 111,1 kHz. O *time step* atribuído à porta de saída do gerador foi de 10 us, ou seja, uma amostra é gerada a cada 10 us. Estes valores foram escolhidos tendo em vista as aplicações desenvolvidas para validar o modelo do SoC (vide Capítulo 7), mas podem ser alterados de acordo com a necessidade.

A classe ADC converte sinais contínuos em valores digitais discretos de N bits. A tensão de

referência é ajustada com o parâmetro *uref*. Os parâmetros *gain_e* e *offset_e* permitem simular erros estáticos como erro do ganho e erro do *offset*, respectivamente. Esses dois parâmetros, entretanto, não foram utilizados no ADC instanciado na platarforma SoC, conforme mostra a Tabela 6.2.

Tabela 6.2: Parâmetros do ADC da plataforma virtual SoC

| **Parâmetro** | **Tipo** | **Valor** | **Descrição** |
|---|---|---|---|
| n | sc_module_name | ADConv | Nome do módulo |
| uref | double | 1.0 | Tensão de referência do ADC |
| gain_e | double | 0.0 | Erro máximo do ganho |
| offset_e | double | 0.0 | Erro máximo do offset |

A Listagem 6.9 mostra as funções *processing* e de cálculo da saída do ADC.

```
void processing()
{
   double analog;
   double temp;
   long erg;

   // lê entrada analógica
   analog = in.read();

   temp = gainError(gain_err, analog);
   temp = temp + offset_err;
   temp = temp + NlError(nl_mode, analog);

   erg = roundValue(temp / lsb);
   erg = maximumValue(erg);
   // salva como bit-vector
   bv_erg = erg;

   // escreve resultado na porta de saída
   out.write(bv_erg);
   (...)
};

// calcula o valor digital e arredonda
long roundValue(double val)
{
    return (long)lround(val);
}

// os parâmetros de erros são zero no ADC do SoC
double gainError(double err, double val)
{
    return(val * (1.0 + err));
}
```

```
// limita a saida
long maximumValue(long value)
{
    if( abs(value) > max-1){
        if(value > 0)
            value = max-1;
        else
            value = - max;
    }
     return value;
}
```

Listagem 6.9: Algoritmo do ADC

A resolução do conversor foi ajustada a partir de um *template*, que é um recurso oferecido pelo C++ que permite criar funções genéricas que admitem quaisquer tipos de dados como parâmetros. Neste caso, o parâmetro foi $N$ (número de bits), cujo valor escolhido foi 8.

Na Figura 7.1, é possível ver a integração do ADC em SystemC-AMS com a plataforma virtual SoC. O bloco *Sine* foi conectado na entrada do ADC, formando um cluster TDF, conforme foi descrito no Capítulo 2. A saída do ADC, por sua vez, deve ser conectada ao modelo que está descrito em SystemC TLM. Sendo assim, foram utilizados dois blocos extras: *ctdf*, que converte o sinal TDF em DE e gera uma *flag* para indicar o fim da conversão (*EOC*), e o *ADCWrapper* que, a partir dos valores recebidos, comunica-se via *sockets* com o processador e com o gerenciador de interrupções.

A Listagem 6.10 mostra a utilização da porta conversora entre os domínios TDF e DE do SystemC-AMS no código do *ctdf*. Neste caso, foi utilizado um sinal do tipo *sca_tdf::sca_signal<T>* para conectar os domínios TDF e DE e, sendo assim, um evento será gerado do domínio DE somente quando houver uma mudança no valor do sinal TDF (vide Figura 2.10).

```
sca_tdf::sca_in< sc_bv<N> > inTDF;
// Porta conversora
sca_tdf::sca_de::sca_out< sc_bv<N> > outDE;
sc_core::sc_out< bool > EOC;
```

Listagem 6.10: Portas do bloco usado para converter sinais entre os domínios TDF e DE

A Listagem 6.11 mostra o processo utilizado para gerar interrupção quando um dado é recebido do ADC. Neste caso, é feita uma chamada ao método *b_transport*.

```
void ADCWrapperProcess()
{
    while(true)
    {
        int dataDummy;
        unsigned int temp;

        wait(10,SC_US);
```

```
        if(!(ADCWrapperEOC.read()==EOCAux))
        {
            sc_int<sizeof(int)*8> temp = ADCWrapperDataInput.read();
            regDataRX = temp.range(15,0);

            EOCAux = ADCWrapperEOC.read();
            (...)

            tlm::tlm_generic_payload* trans = new tlm::tlm_generic_payload;
            sc_time delay = sc_time(10, SC_NS);
            trans->set_command( tlm::TLM_WRITE_COMMAND );
            trans->set_address( 1 );
            trans->set_data_ptr( reinterpret_cast<unsigned char*>(&dataDummy) );
            trans->set_data_length( 2 );

            intADCWrapperInSocket->b_transport( *trans, delay );

            (...)
        }
    }
}
```

Listagem 6.11: Algoritmo do ADCWrapper

A próxima subseção apresenta a descrição do ADC especificado para o SoC (vide Capítulo 5) utilizando a linguagem VHDL-AMS.

#### 6.1.3.1 ADC em VHDL-AMS

Um modelo VHDL-AMS do ADC foi descrito em [79, 80] e foi utilizado como estudo de caso para uma primeira simulação de um SoC misto [87]. Para isso, foi necessário utilizar uma ferramenta proprietária que permitisse a co-simulação de módulos descritos em linguagens de descrição de *hardware* diferentes, neste caso VHDL-AMS e SystemC TLM. O simulador NCSim, da Cadence [88], foi utilizado devido ao suporte oferecido para ambas as linguagens e aos recursos de visualização de formas de onda e diagramas de blocos.

A Figura 6.3 mostra o diagrama de blocos do ADC modelado. No nível mais alto de hierarquia, ele possui quatro portas: *AD_D_CLK* e *AD_D_ON_OFF* são entradas digitais para o sinal de *clock* e para a chave liga/desliga, respectivamente; *AD_D_OUTPUT* é a saída digital que disponibiliza o dado no fim do ciclo de conversão; *ad_convdone* é uma saída digital que, quando em nível lógico 1, indica que uma palavra foi convertida e está disponível na saída. O conversor tensão-corrente está implícito no bloco sinais, que é uma entidade puramente analógica que estimula a entrada *AD_A_INPUT* do ADC com um sinal senoidal, mesmo quando o ADC não está convertendo. Um bloco de estímulo (*testbench*) é responsável por gerar as entradas digitais. Entretanto, quando o ADC está integrado ao SoC, o controle da operação é feito pelo processador.

A Listagem 6.12 mostra partes do código VHDL-AMS do ADC. Em suma, o sinal de entrada do ADC é gerado internamente (*ad_d_in*) com frequência de 22 kHz e amplitude de 100uA. Se

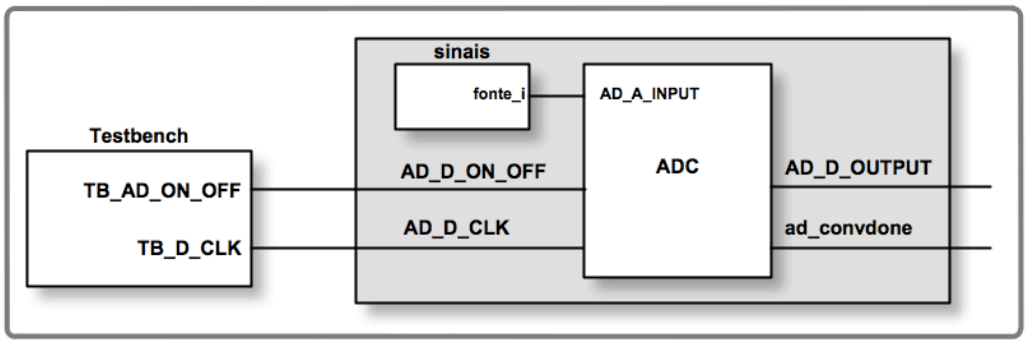


Figura 6.3: Diagrama de blocos do ADC em VHDL-AMS

a chave *AD_D_ON_OFF* estiver habilitada, este sinal é amostrado e, se estiver dentro da faixa permitida (entre -100uA e 100uA), o *loop* de conversão é iniciado, após o qual o valor convertido é escrito na saída na forma de vetor de 8 bits.

```
entity ad_ideal_real_auto is
   port (   signal AD_D_OUTPUT : out bit_vector(7 downto 0);
      signal AD_D_CLK : in bit;
      signal AD_D_ON_OFF : in bit;
      signal ad_convdone : out bit := '0');
end entity ad_ideal_real_auto;

architecture ad_ideal_real_auto of ad_ideal_real_auto is
   constant ref_zero : real :=0.0;
   constant ref_100 : real :=100.0E-6;
   constant freq_i : real := 22.0e+3;     -- frequencia do sinal

   subtype out_byte is bit_vector(7 downto 0);  -- subtype que vai ser usado para
      armazenar os bits

   (...)

   quantity ad_i_in : real;          -- declaracao do sinal interno ad_i_in

begin
   -- senoide de -100uA a +100uA, centrada em 0.0
   ad_i_in == 100.0e-06*sin(2.0*math_pi*freq_i*now);

   ad_ideal_conversao: process is

      (...)
      begin
      amostragem_e_conversao : WHILE AD_D_ON_OFF = '1' LOOP
      -- So inicia o ciclo de amostragem e conversao se AD_D_ON_OFF = '1'
      exit amostragem_e_conversao when AD_D_ON_OFF='0';

         (...)
```

```
        i_smp := ad_i_in;     -- Amostra a corrente de entrada
        i_smp_reg := i_smp;      -- Registra o valor para uso posterior

        (...)

        -- Loop de conversao de 8 bits
        conversao : FOR i IN 7 DOWNTO 0 LOOP
           IF i_smp >= ref_zero THEN
              output_byte(i) := '1';
              i_smp := i_smp*2.0 - ref_100;

           ELSE
              output_byte(i) := '0';
              i_smp := i_smp*2.0 + ref_100;

           END IF;
        END LOOP conversao;
        AD_D_OUTPUT <= output_byte;

        (...)
   end process ad_ideal_conversao;
end architecture ad_ideal_real_auto;
```

Listagem 6.12: Código VHDL-AMS do ADC

Para integrar o bloco em VHDL-AMS ao modelo em SystemC TLM, algumas adaptações foram necessárias. Primeiramente, um *wrapper* - bloco utilizado para adaptar e conectar sinais de tipos diferentes - foi gerado para conectar as entradas e saídas do ADC ao processador. Em seguida, foram feitas as ligações utilizando sinais, conforme é mostrado na Figura 6.4. Pode-se perceber que agora o bloco ADC possui duas entradas a serem estimuladas pelo bloco *InterfaceStim*.

### 6.1.4 Interfaces Serial e de RF

Como o SoC foi modelado como estudo de caso para constituir uma plataforma virtual mista, apenas o ADC teve a sua funcionalidade implementada. As interfaces serial e de RF do SoC tiveram apenas a conexão com o gerenciador de interrupções descrita em código, conforme mostram as Listagens 6.13 e 6.14.

```
void serialProcess()
{
   while(true)
   {
      wait(10,SC_MS);
      if(serialInput.read()!=0){
         unsigned int data;
         data = serialInput.read();

         tlm::tlm_generic_payload* trans = new tlm::tlm_generic_payload;
         sc_time delay = sc_time(10,SC_NS);
            trans->set_command( tlm::TLM_WRITE_COMMAND );
```

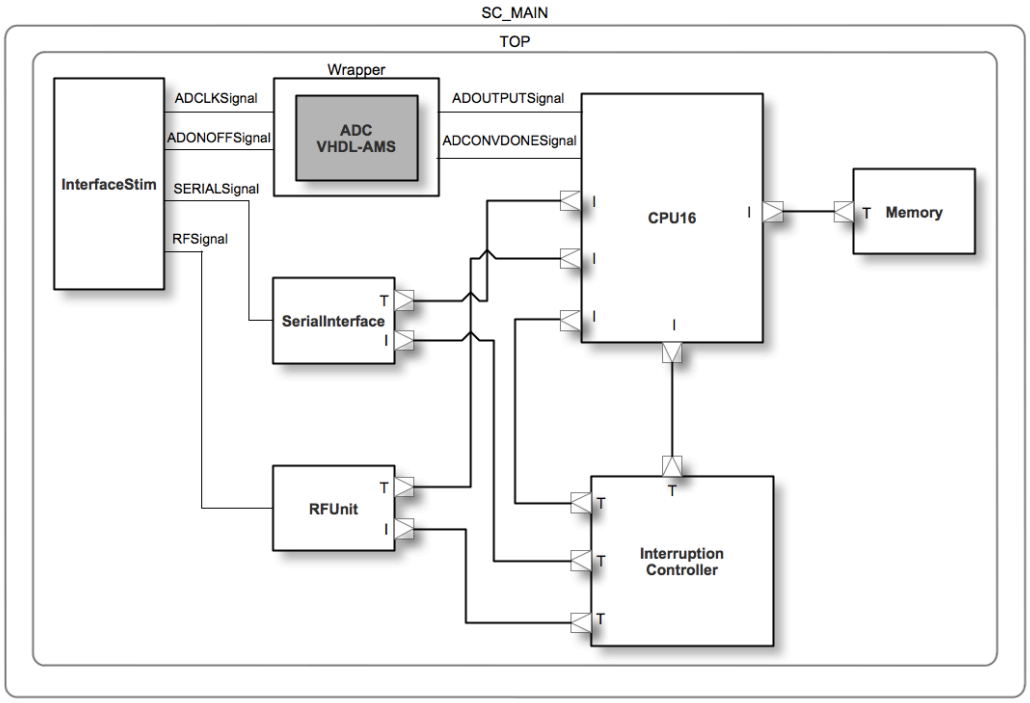


Figura 6.4: Diagrama de blocos do SoC com o ADC em VHDL-AMS

```
            trans->set_address( 3 );
            trans->set_data_ptr( reinterpret_cast<unsigned char*>(&data) );
            trans->set_data_length( 2 );

            intSerialInSocket->b_transport(*trans,delay);

            if(trans->is_response_error()){
               if(trans->get_response_status()==tlm::TLM_COMMAND_ERROR_RESPONSE)
                  if(_debug)
                     printf("Serial Interface: Attending a previous interrupt
                        request\n");
            }
      }
   }
}
```

Listagem 6.13: Algoritmo da interface serial

```
void RFProcess()
{
   while(true)
   {
      wait(10,SC_MS);
      if(RFInput.read()!=0){
         unsigned int data;
         data = RFInput.read();
```

```
            tlm::tlm_generic_payload* trans = new tlm::tlm_generic_payload;
            sc_time delay = sc_time(10,SC_NS);
                trans->set_command( tlm::TLM_WRITE_COMMAND );
                trans->set_address( 2 );
                trans->set_data_ptr( reinterpret_cast<unsigned char*>(&data) );
                trans->set_data_length( 2 );

                intRFInSocket->b_transport(*trans,delay);

                if(trans->is_response_error()){
                    if(trans->get_response_status()==tlm::TLM_COMMAND_ERROR_RESPONSE)
                        if(_debug)
                            printf("RF Interface: Attending a previous interrupt request\n
                                ");
                }
        }
    }
}
```

Listagem 6.14: Algoritmo da interface de RF

Observa-se que o processo de ambas é similar. O que muda é apenas o atributo *address* do *generic payload trans*, que vai definir o tipo de interrupção. Com isso, pode-se utilizar um gerador de estímulo nas entradas de ambas as interfaces para simular o recebimento de dados (sinais *SERIALSignal* e *RFSignal* na Figura 7.1).

## 6.2 Plataforma Virtual RSoC

Esta plataforma virtual corresponde ao modelo do RSoC-32 descrito no Capítulo 5. Neste caso, não há nenhuma implementação em *hardware* disponível para comparação de resultados e a sua validação será feita a partir de aplicações cujo resultado esperado é conhecido.

A plataforma virtual RSoC está mostrada na Figura 6.5. Esta plataforma virtual foi implementada com o intuito de executar processamentos digitais de dados adquiridos pelos sensores. Como entre as aplicações há processamento de imagens, um estudo de caso incluindo os blocos processador, APS, RoSA e memória foi escolhido para executar aplicações como, por exemplo, o JPEG [89].

### 6.2.1 Processador

Para a modelagem do processador, foi utilizada a ADL ArchC. A versão 2.0 dessa ferramenta permite adicionar uma porta TLM bidirecional ao processador (*DM_Port*) para se comunicar com módulos externos. Sempre que uma instrução do tipo *load/store* é executada, uma transação é iniciada com o envio de uma requisição ao barramento através dessa porta. Um protocolo baseado em SystemC TLM chamado ArchC TLM Protocol deve ser seguido para conectar um módulo àquela porta, e este módulo, por sua vez, deve herdar a classe *ac_tlm_transport_if*. Esta abordagem foi utilizada para conectar todos os outros blocos mostrados na Figura 6.5 ao processador. O código

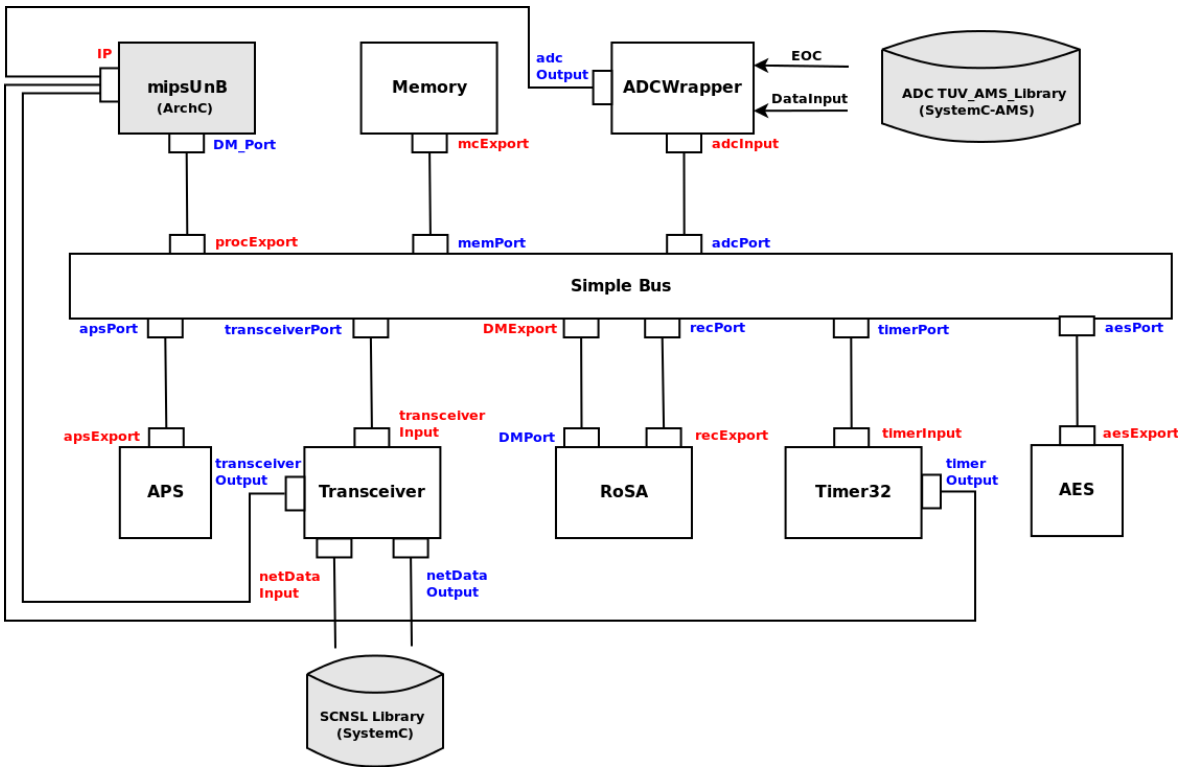


Figura 6.5: Modelo da plataforma virtual RSoC

gerado pelo ArchC foi modificado para atender às necessidades do projeto. Um segundo banco com 32 registradores foi adicionado para a implementação de funções de controle do processador e tratamento de interrupções. A Listagem 6.15 mostra o arquivo utilizado para gerar o processador no ArchC.

```
AC_ARCH(mipsUnB){

    ac_tlm_port    DM:5M;
    ac_tlm_intr_port IP;

    ac_regbank RB:32;
    ac_regbank C0_RB:32;

    ac_reg npc;
    ac_reg hi, lo;

    ac_wordsize 32;

    ARCH_CTOR(mipsUnB) {

      ac_isa("mipsUnB_isa.ac");
      set_endian("big");

};
```

Listagem 6.15: Algoritmo para geração do mipsUnB

A porta de entrada *IP* é do tipo *sc_ export* do processador e é usada para a geração de interrupção a partir dos periféricos do sistema. Quando um deles requisita uma interrupção, esta porta recebe uma transação e o processador interrompe o fluxo normal de execução do programa para executar uma rotina de tratamento de interrupção. Em seguida, o processador guarda o endereço de retorno da rotina de tratamento de interrupção no registrador *NPC*, escreve no registrador *CAUSA* o código associado ao periférico que fez a requisição, desabilita novas interrupções utilizando o registrador *CONFIG* e retoma a execução do programa no endereço 0x8h, no qual está armazenado o início da rotina de tratamento de interrupções. Ao final da rotina, as interrupções são habilitadas e o contador de programa retorna ao endereço da instrução que seria executada quando ocorreu a interrupção.

A Figura 6.6 mostra como o processo (*thread*) do processador é executado. A simulação é iniciada com a etapa de elaboração, na qual o módulo do processador lê o arquivo binário com o programa que será executado e o escreve na memória via barramento. Em seguida, vem a etapa de execução, na qual as instruções são lidas, decodificadas e executadas. Finalmente, após a execução, o tempo gasto com essas operações é calculado e um método *wait()* é chamado, permitindo que outras *threads* do modelo possam executar suas tarefas. É importante ressaltar que, neste trabalho, considera-se que cada instrução é executada em um ciclo de relógio.

Algumas alterações foram feitas na arquitetura tendo em vista a implementação de diferentes modos de operação (normal, baixa velocidade e baixo consumo). Mais detalhes podem ser encontrados em [15].

### 6.2.2 Barramento

Como, neste nível de abstração, o principal propósito é gerar uma plataforma virtual para executar *software* embarcado, um controlador de memória foi implementado para atuar como um barramento simples. Outro barramento como o AMBA [90], por exemplo, poderá ser utilizado em modelos em níveis mais baixos de abstração.

As requisições dos periféricos são roteadas de acordo com suas faixas de endereçamento, mostradas na Tabela 6.3. Uma porta *sc_ export* chamada *procExport* utilizando a interface *ac_ tlm_ transport_ if* foi incluída para conectar o controlador de memória ao processador. De maneira similar, mais portas foram adicionadas para conectar os blocos restantes.

Dessa maneira, cada periférico pode implementar até 1.048.575 registradores de uso específico, e o tamanho máximo da memória ficou em 5 MB. Pode-se observar que o RoSA e o APS estão endereçados dentro da faixa de endereços reservados para a memória.

uma chamada ao método *wait()*, o que permite que as outras *threads* do modelo executem suas tarefas. Neste trabalho, considerou-se que cada instrução é executada em um ciclo de relógio e não foi modelada nenhuma estrutura de pipeline.

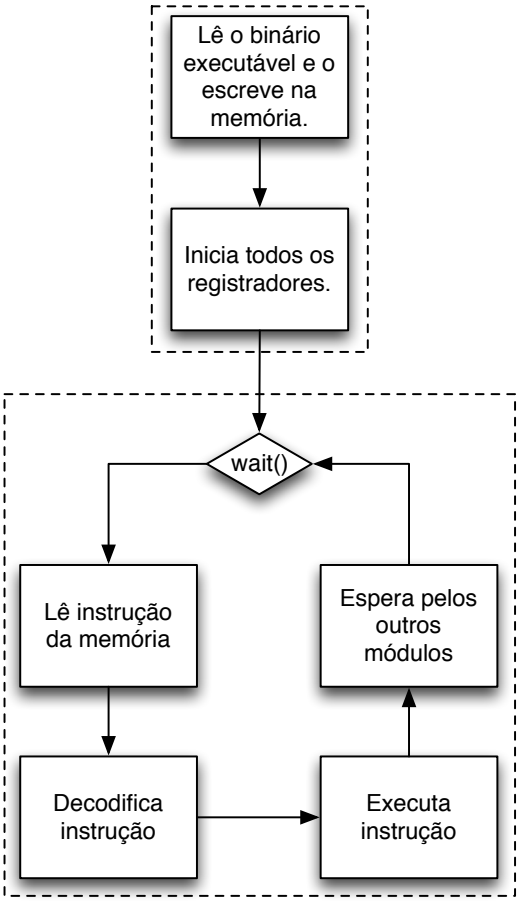


Figura 6.6: Fluxo de execução do modelo do processador [15]

Figura 4.5: Fluxo de execução do modelo do processador.

### 6.2.3 Memória

Como mostrado na Figura 4.1, o módulo processador possui duas portas para comunicação com os periféricos conectados ao sistema. A porta de saída (*sc_port*) é utilizada para o acesso à memória e outros registradores mapeados em memória. Sempre que uma instrução do tipo *load/store* é executada, uma transação é iniciada com o envio de um requisição ao barramento através desta porta. A transação termina com a resposta do módulo escravo, como descrito na Seção 4.2.1.

A memória foi implementada como um vetor de bytes dinamicamente alocado. O software de aplicação é carregado na memória durante a fase de elaboração. Duas rotinas chamadas *memRead* e *memWrite* são usadas para operar o vetor e uma *sc_export* chamada *mcExport* foi incluída para conectar a memória ao comutador. A Listagem 6.19 mostra a parte do código utilizado.

```
class tlmMem: public sc_module, public read_write_port_if<ac_tlm_req>
{
public:
  sc_export<read_write_port_if<ac_tlm_req> > mcExport;
  //Memory Reading
  virtual ac_tlm_rsp memRead(const ac_tlm_req &reading)
  {
    ac_tlm_rsp response;

    ((uint8_t*)&(reading.data))[0]=memory[reading.addr];
    ((uint8_t*)&(reading.data))[1]=memory[reading.addr+1];
```

A porta de entrada (*sc_export*) do módulo processador é utilizada para a geração de interrupções pelos periféricos conectados ao sistema. Qualquer transação recebida por esta porta faz com que o processador interrompa o fluxo normal de execução do programa e execute a rotina padrão de tratamento das interrupções. Quando isto acontece, o processador guarda o endereço de retorno da rotina de tratamento de interrupções no registrador NPC, escreve no registrador CAUSA o código correspondente ao periférico que gerou a interrupção, desabilita a ocorrência de novas interrupções através do registrador CONFIG e retoma a execução do programa de aplicação a partir do endereço 0x8. O programador é responsável por

Tabela 6.3: Mapeamento de memória dos periféricos

| Periférico | Faixa de endereços |
|---|---|
| Memória | 0x0000 0000 : 0x004F FFFF |
| RoSA | 0x0040 0000 : 0x0040 00FF |
| APS | 0x0040 0100 |
| Transceptor | 0x0050 0000 : 0x005F FFFF |
| Timer | 0x0060 0000 : 0x006F FFFF |
| AES | 0x0070 0000 : 0x007F FFFF |
| ADC | 0x0080 0000 : 0x008F FFFF |

```
        ((uint8_t*)&(reading.data))[2]=memory[reading.addr+2];
        ((uint8_t*)&(reading.data))[3]=memory[reading.addr+3];
        response.status=SUCCESS;
        response.data=reading.data;
        return response;
    }

    //Memory Writing
    virtual ac_tlm_rsp memWrite(const ac_tlm_req &writing)
    {
        ac_tlm_rsp response;

        memory[writing.addr]=((uint8_t*)&(writing.data))[0];
        memory[writing.addr+1]=((uint8_t*)&(writing.data))[1];
        memory[writing.addr+2]=((uint8_t*)&(writing.data))[2];
        memory[writing.addr+3]=((uint8_t*)&(writing.data))[3];
        response.status=SUCCESS;

        return response;
    }
...
```

Listagem 6.16: Algoritmo utilizado na memória TLM

### 6.2.4 Timer e AES

Foi modelado em [15] um *timer* de modo que o sistema possa gerar eventos periódicos. Este módulo implementa uma *thread* que periodicamente gera transações na suas porta de saída. A periodicidade dos eventos gerados é calculado a partir do período de *clock* definido em tempo de compilação e do conteúdo de dois registradores: *TIMER_PRESCALER* e *TIMER_COMPARATOR*. O primeiro tem função de simular um divisor de *clock* e o segundo armazena o valor que será comparado com o valor do contador do módulo. Um terceiro registrador, *TIMER_CONFIG*, indica se o *timer* está ou não habilitado.

Foi incluído na plataforma virtual um bloco AES implementado em [15] utilizando SystemC

TLM. Tal módulo foi acrescentado para ampliar a utilização da plataforma virtual para aplicações de RSSF e não será detalhado aqui.

### 6.2.5 RoSA

O RoSA foi modelado como estudo de caso por [91] e aperfeiçoado por [16] para ser conectado à versão 0.7.8 do MIPS-I gerado pelo ArchC. Devido à sua simplicidade e também ao aumento de performance de *hardware* para aplicações de processamento de imagens, nesta tese ele foi adaptado para ser integrado ao mipsUnB e à plataforma virtual RSoC. Este módulo pode receber e enviar requisições de leitura de memória, já que o gerenciador de configurações busca a configuração do RoSA no bloco de memória, quando a mesma não está disponível no *cache* de configuração. O gerenciador de configurações também envia um sinal às células para indicar o início e o fim de uma operação.

A Figura 6.7 mostra o diagrama de blocos indicando como o RoSA foi modelado, e a Listagem 6.17 mostra o trecho do código que corresponde à definição da sua classe. Foram criados os bancos de registradores global, modificado e de saída. Uma matriz de 6 vetores de 120 bits é utilizada para configurar as células. Há duas portas para comunicação com o barramento. *DMPort* é do tipo *sc_port* e é usada para comunicação com a memória. A porta *recExport* é do tipo *sc_export* e é usada para fazer leituras ou escritas no bloco reconfigurável.

```
class RoSA: public sc_module, public read_write_port_if<ac_tlm_req>
{
   //Register Banks
   sc_uint<32> regGlobal[48], regModified[48];
   sc_uint<64> regOut[6];;
   sc_biguint<120> arrayConf[6];
public:
   //Port to communicate with memory
   sc_port<ac_tlm_transport_if> DMPort;
   //Port to read/write to the reconfigurable array
   sc_export<read_write_port_if<ac_tlm_req> > recExport;

   //Cell instances
   recCell *cell1;
   recCell *cell2;
   recCell *cell3;
   recCell *cell4;
   recCell *cell5;
   recCell *cell6;

   //Configuration manager instances
   configurationCache *confCache;
   configurationManager *confManager;
...
```

Listagem 6.17: Classe RoSA

Há seis instâncias de células e uma instância do gerenciador de configurações. A célula reconfi-

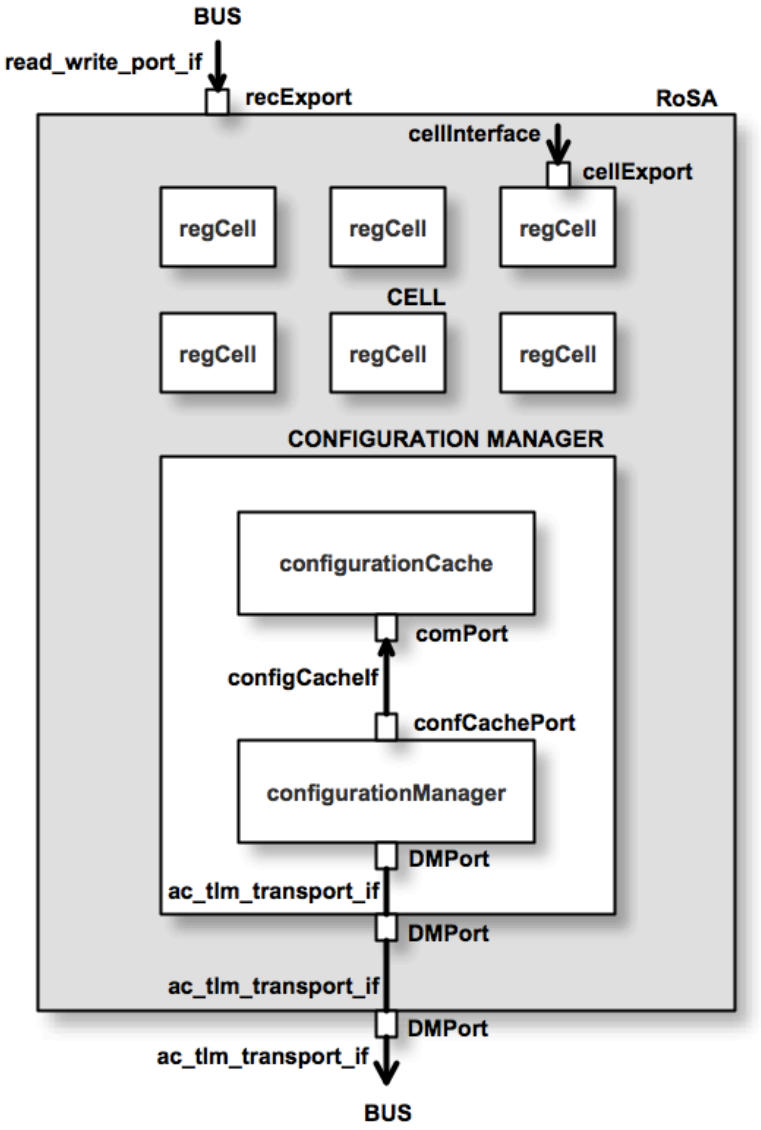


Figura 6.7: Diagrama de blocos do modelo do RoSA

gurável (6.18) foi modelada usando um método para executar a configuração da mesma de acordo com a palavra de 120 bits lida pelo gerenciador de configurações. O gerenciador de configurações, por sua vez, é um módulo mapeado na memória. Ele recebe o endereço de uma palavra de configuração e procura pela mesma na *cache* de configuração. Caso não esteja lá, lê a memória via barramento e atualiza a *cache*. Ele também é responsável por sinalizar o início das operações das células e interromper o processador no final.

```
class recCell: public sc_module, public cellInterface
{
   sc_uint<64> regInt1, regInt2, regInt3, regInt4;
   sc_uint<32> regOp[8];
   unsigned int cellNumber;
public:
```

```
sc_export<cellInterface> cellExport;

virtual void executeConfiguration(sc_biguint<120> arrayConf,
      sc_uint<32> *regGlobal, sc_uint<32> *regModified,
      sc_uint<64> &output, bool &done)
{
   sc_uint<2> level1,level2,level3,level4,level5;
   level1=arrayConf.range(115,114);
   level2=arrayConf.range(113,112);
   level3=arrayConf.range(111,110);
   level4=arrayConf.range(109,108);
   level5=arrayConf.range(107,106);

   sc_uint<6> op1,op2,op3,op4,op5,op6,op7;
   op1=arrayConf.range(105,100);
   op2=arrayConf.range(99,94);
   op3=arrayConf.range(93,88);
   op4=arrayConf.range(87,82);
   op5=arrayConf.range(81,76);
   op6=arrayConf.range(75,70);
   op7=arrayConf.range(69,64);

   (...)

   switch(level1)
   {
   //L1 - C4
   case 3:
      //4 operations on the first level
      operation(regOp[0],regOp[1],op1,regInt1);
      operation(regOp[2],regOp[3],op2,regInt2);
      operation(regOp[4],regOp[5],op3,regInt3);
      operation(regOp[6],regOp[7],op4,regInt4);
      //2 operations on the second level
      operation(regInt1,regInt2,op5,regInt1);
      operation(regInt3,regInt4,op6,regInt3);
      //1 operation on the third level
      operation(regInt1,regInt3,op7,output);
      break;
   (...)
```

Listagem 6.18: Classe da célula reconfigurável do RoSA e trecho do método de configuração das células

A Tabela 6.4 mostra as faixas de endereço das estruturas do RoSA utilizadas na plataforma virtual RSoC.

### 6.2.6 APS

Nesta plataforma virtual, considera-se que o APS possui seu próprio ADC e é integrado ao núcleo do RSoC. Ele é implementado como um módulo que lê um arquivo de imagem e o armazena

Tabela 6.4: Mapa de endereços das estruturas do RoSA

| Estrutura | Endereço |
|---|---|
| Gerenciador de configurações | 0x0040 00FF |
| Registradores globais | 0x0040 0000 : 0x0040 002F |
| Registradores modificados | 0x0040 0030 : 0x0040 005F |
| Registradores locais | 0x0040 0060 : 0x0040 00F8 |
| Registradores de saída | 0x0040 00F9 : 0x0040 00FE |

nos registradores apropriados e que serão lidos pelo programa de aplicação. O bloco APS foi mapeado em um único endereço. Quando o processador acessa esse endereço para uma operação de leitura, o APS envia 4 bytes (ou 4 pixels) lidos do arquivo da imagem. Tão logo os últimos 4 bits da imagem são enviados, o APS abre um novo arquivo. Uma *sc_ export* chamada *apsExport* foi adicionada para conectar o APS ao barramento.

O método que está implementado no módulo APS faz a leitura de arquivos de imagem no formato PGM (*Portable GrayMap*). Esse formato foi escolhido por facilitar a verificação da rotina de leitura. Cada arquivo possui no seu início dois bytes de descrição em ASCII, compostos pela letra *P* seguida de um número, especificando se o seu tipo é PGM. Após a descrição do tipo, existe um espaço para comentário e, em seguida, o número de linhas e colunas de pixels. O último item do cabeçalho é o valor de brilho que a imagem pode usar e varia entre 0 (sem brilho) e 255 (brilho máximo). Cada pixel é representado por 8 bits.

A implementação deste módulo e sua integração ao RoSA foram tema de um projeto final de graduação e são descritas com mais detalhes em [16]. Nesta tese, este módulo foi adaptado para ser integrado na plataforma virtual RSoC.

### 6.2.7 Transceptor

Como o objetivo é demonstrar a funcionalidade do RSoC em um ambiente de rede sem fio, o transceptor foi modelado para enviar e receber pacotes, que são montados e desmontados pelo processador. Após o comando de transmissão, o módulo executa o algoritmo do protocolo de acesso ao meio e inicia transações utilizando uma interface de rede definida pelo SCNSL.

Conforme é mostrado em [15], este módulo foi modelado da forma mais simplificada possível visando demonstrar a estratégia de modelagem sem incluir todos as possibilidades de funcionamento que o módulo de *hardware* poderá implementar. Dessa forma, onze registradores foram modelados, mostrados na Tabela . O registrador *TRANSCEIVER_ CONFIG* é utilizado para configurar parâmetros do módulo como taxa e potência de transmissão mas no modelo implementado nenhuma função foi atribuída a este registrador. O registrador *TRANSCEIVER_ STATUS* é utilizado para acompanhar a execução das funções do transceptor: o bit 0 (menos significativo) indica que uma transmissão está em curso e o bit 1 indica que um pacote foi recebido e não tratado pelo processador. Neste modelo, são transmitidos 128 bits por vez. Os registradores *TRANSCEIVER_ SEND_ DATA* são os que armazenam os dados a serem enviados. O formato dos pacotes enviados é deixado a

cargo do programador. Os registradores *TRANSCEIVER_RECEIVE_DATA* armazenam os dados recebidos pelo módulo. A escrita de qualquer valor no registrador *TRANSCEIVER_SEND* inicia a transmissão dos dados armazenados nos registradores *TRANSCEIVER_SEND_DATA*.

Tabela 6.5: Mapa de registradores do transceptor de RF [15]

| REGISTRADOR | Endereço |
|---|---|
| TRANSCEIVER_CONFIG | 0x0000 0000 |
| TRANSCEIVER_STATUS | 0x0000 0001 |
| TRANSCEIVER_SEND | 0x0000 0002 |
| TRANSCEIVER_SEND_DATA0 | 0x0000 0003 |
| TRANSCEIVER_SEND_DATA1 | 0x0000 0004 |
| TRANSCEIVER_SEND_DATA2 | 0x0000 0005 |
| TRANSCEIVER_SEND_DATA3 | 0x0000 0006 |
| TRANSCEIVER_RECEIVE_DATA0 | 0x0000 0007 |
| TRANSCEIVER_RECEIVE_DATA1 | 0x0000 0008 |
| TRANSCEIVER_RECEIVE_DATA2 | 0x0000 0009 |
| TRANSCEIVER_RECEIVE_DATA3 | 0x0000 000A |

## 6.3 *Wrapper* para ADC

Este módulo é similar ao apresentado na Listagem 6.11, correspondente ao modelo do SoC. Ele apenas foi adaptado para escrever o dado em um registrador de 32 bits e utilizar o método *transport()* de acordo com o protocolo ArchC TLM, ao invés de utilizar o *generic payload* e o método *b_transport()* do TLM2, como foi apresentado anteriormente. A Listagem 6.19 mostra as alterações.

```
void ADCWrapperProcess()
{
    while(true)
    {
        unsigned int temp;

        wait(2001,SC_US);

        if(!(ADCWrapperEOC.read()==EOCAux))
        {
            sc_int<sizeof(int) * 8> temp = ADCWrapperDataInput.read();
            regDataRX = temp.range(31, 0);

            EOCAux = ADCWrapperEOC.read();

            adcResponse = adcOutput->transport(adcRequest);

        }
```

```
        }
    }
```

Listagem 6.19: Trecho do algoritmo do ADCWrapper utilizando o método *transport()*

Com isso, foram descritos os módulos de ambas as plataformas virtuais. O capítulo a seguir mostra as aplicações que foram utilizadas para verificar o correto funcionamento dos modelos e os resultados obtidos nas simulações.

# Capítulo 7

# Simulação dos Modelos

*Os dois capítulos anteriores apresentaram a descrição e a modelagem dos sistemas, tendo em vista suas especificações. Uma vez definidas as arquiteturas de todos os componentes, faz-se necessário, além da validação individual, verificar se após a integração a funcionalidade desejada será mantida.*

## 7.1 Aplicações para o SoC

Para a plataforma virtual SoC apresentada no capítulo 6, que está sendo reproduzida aqui na Figura 7.1, foi elaborada uma série de aplicações que permitissem tanto verificar o fluxo de dados entre os módulos quanto validar o funcionamento dos mesmos em um ambiente de rede de sensores sem fio. Os programas consistiram em aplicações simples devido à limitação imposta pela falta de um compilador para o RISC16. Sendo assim, todos foram escritos em *assembly*, o que torna o procedimento trabalhoso e suscetível de erros.

### 7.1.1 Série de Fibonacci

O primeiro programa consistiu na série de Fibonacci, que é formada pela soma dos dois últimos termos (ex: 0, 1, 1, 2, 3, 5, 8, 13, ...). Para programar essa série no processador RISC16, são necessários 3 registradores e a operação de soma.

Um exemplo de execução é mostrado na Listagem 7.1. Pode-se ver que a série está no sexto termo, pois o registrador $s1 apresenta o valor 8, e os valores de F(n-1) e F(n-2) estão nos registradores $s0 e $a2, com os valores 5 e 3, respectivamente.

```
********************************
DMI    = { R, 3 } , data = 2678 at time 1050 ns
********************************
        DEBUG INFORMATION
********************************
OP: 2, REGS: 6, REGS2: 7, REGD: 8
Add = 8
```

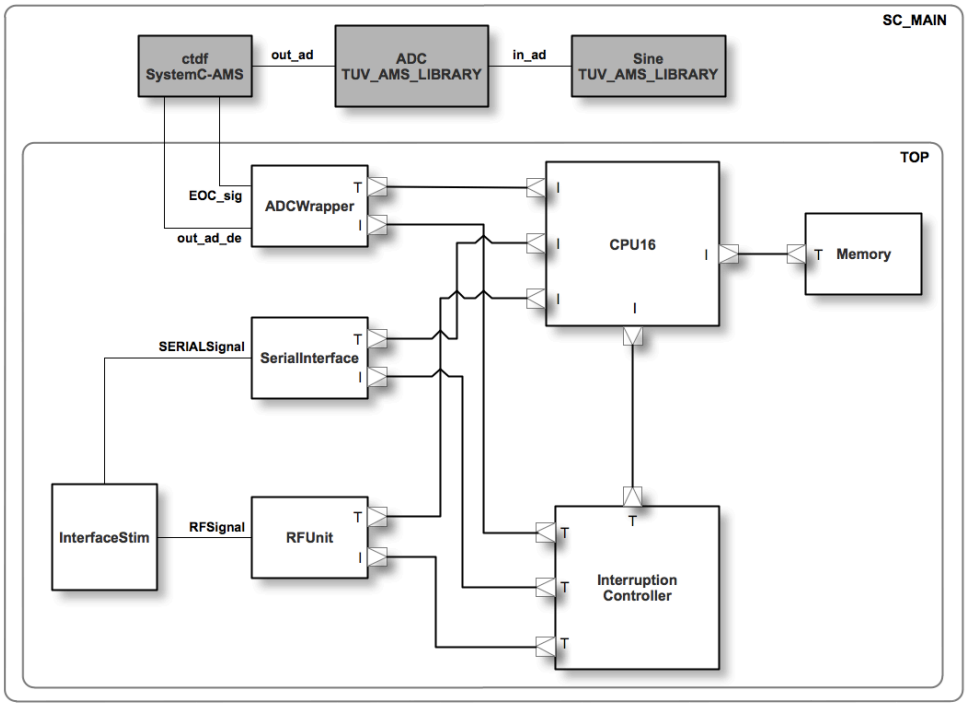


Figura 7.1: Diagrama de blocos do SoC com o ADC em SystemC-AMS

```
C: 0    C_aux: 0    Z: 0    N: 0    Bit3: 0
reg0($zero) = 0
reg1($t0) = 0
reg2($t1) = 0
reg3($t2) = 0
reg4($a0) = 0
reg5($a1) = 0
reg6($a2) = 3
reg7($s0) = 5
reg8($s1) = 8
reg9($s2) = 0
reg10($s3) = 0
reg11($s4) = 0
reg12($gp) = 0
reg13($sp) = 0
reg14($pc) = 4
reg15($ra) = 0
Int_reg=0
Int_pc=0
********************************
```

Listagem 7.1: Resultado da simulação da série de Fibonacci na plataforma virtual SoC

### 7.1.2 Teste da Interrupção com o ADC

O segundo programa testa o processo de interrupção do processador recebendo um dado convertido quando o ADC requisita uma interrupção. Enquanto não há interrupção, uma rotina fica rodando em *loop* infinito. Quando há interrupção, o dado convertido é mostrado na informação de *debug* e é carregado em um dos registradores.

A Listagem 7.2 mostra um exemplo de operação. Em t = 20 us, o ADC requisita uma interrupção. Observe que o ADC em SystemC-AMS executa uma conversão a cada 10 us, como pode ser visto na Figura 7.2. O valor que representa a tensão analógica no ciclo anterior (19,95 us) é de aproximadamente 0,64 V e foi convertido para a palavra 0x52h. Pode-se ver a operação completa realizada com sucesso, em 20,25 us. O resultado é carregado no registrador $s0.

```
*********************************
DMI   = { R, 3 } , data = 2123 at time 20 us
Received data from ADC SystemC-AMS: 052
20 us
INTERRUPT CONTROLLER : intCausa=2     data_rx=0
trans = { W, 1ff2 } , data = 3 at time 20 us
*********************************
        DEBUG INFORMATION
*********************************
OP: 2, REGS: 1, REGS2: 2, REGD: 3
Add = 5dd
C: 0   C_aux: 0   Z: 0   N: 0   Bit3: 0
reg0($zero) = 0
reg1($t0) = 4f
reg2($t1) = 58e
reg3($t2) = 5dd
reg4($a0) = 52e
reg5($a1) = 0
reg6($a2) = 0
reg7($s0) = 0
reg8($s1) = 0
reg9($s2) = 0
reg10($s3) = 0
reg11($s4) = 0
reg12($gp) = 0
reg13($sp) = 0
reg14($pc) = 0
reg15($ra) = 0
Int_reg=0
Int_pc=0
*********************************

(...)

*********************************
DMI   = { R, 8 } , data = 507 at time 20200 ns
trans = { R, 1fea } , data = 52 at time 20250 ns
*********************************
```

```
          DEBUG INFORMATION
********************************
OP: 0, REGS: 5, REGS2: 0, REGD: 7
Lw = 52
C: 0    C_aux: 1    Z: 0    N: 0    Bit3: 0
reg0($zero) = 0
reg1($t0) = 4f
reg2($t1) = 58e
reg3($t2) = 5dd
reg4($a0) = 52e
reg5($a1) = 1fea
reg6($a2) = 0
reg7($s0) = 52
reg8($s1) = 0
reg9($s2) = 0
reg10($s3) = 0
reg11($s4) = 0
reg12($gp) = 0
reg13($sp) = 0
reg14($pc) = 9
reg15($ra) = 0
Int_reg=0
Int_pc=0
********************************
```

Listagem 7.2: ADC converte dado e requisita uma interrupção

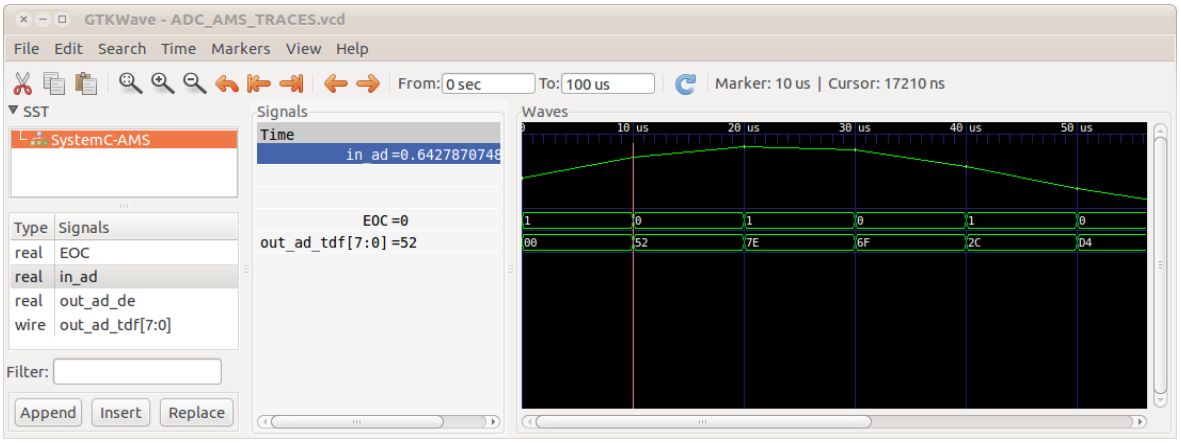


Figura 7.2: Valor convertido pelo ADC em SystemC-AMS

### 7.1.3 Média Aritmética

Uma outra aplicação calcula a média de um dado coletado em campo, como, por exemplo, a temperatura do solo, da fruta ou do ambiente. Nesta situação, o ADC faz a leitura a cada hora, por exemplo, e o processador calcula a média antes de mandar o resultado para uma central.

Precisa-se apenas de uma operação de soma, outra de subtração e uma de divisão. O processador já possui as operações de soma e subtração. Precisa-se criar, agora, uma operação de divisão.

A forma mais imediata - embora não seja a mais eficiente - é o método de somas consecutivas. Por exemplo, para dividir 12 por 3, é necessário somar 3 quatro vezes para atingir 12. Então, o resultado da divisão é 4.

Com dois dados coletados, pode-se observar que a média calculada bateu com o valor teórico. O processador recebeu os seguintes dados convertidos pelo ADC: 0x0 (em 10us), 0x52h (em 20 us) e 0x7eh (em 30 us). A média é dada por (0x0 + 0x52 + 0x7e) / 2 = 0x45 e é mostrada no registrador $t0 no instante 36,55 us. A Listagem 7.3 mostra a saída de *debug* da simulação, e os valores do ADC podem ser vistos na Figura 7.2.

---

```
*********************************
DMI   = { R, 1 } , data = e001 at time 10 us
Received data from ADC SystemC-AMS: 000
10 us
INTERRUPT CONTROLLER : intCausa=2     data_rx=0
trans = { W, 1ff2 } , data = 0 at time 10 us
*********************************
(...)
*********************************
DMI   = { R, 1 } , data = e001 at time 20 us
Received data from ADC SystemC-AMS: 052
20 us
INTERRUPT CONTROLLER : intCausa=2     data_rx=0
trans = { W, 1ff2 } , data = 0 at time 20 us
*********************************
(...)
*********************************
Received data from ADC SystemC-AMS: 07e
30 us
DMI   = { R, 1 } , data = e001 at time 30 us
INTERRUPT CONTROLLER : intCausa=2     data_rx=0
DMI   = { R, 1 } , data = e001 at time 30050 ns
trans = { W, 1ff2 } , data = 0 at time 30050 ns
*********************************
(...)
*********************************
DMI   = { R, 21 } , data = 2181 at time 36550 ns
*********************************
        DEBUG INFORMATION
*********************************
OP: 2, REGS: 1, REGS2: 8, REGD: 1
Add = 45
C: 0    C_aux: 0    Z: 0    N: 0   Bit3: 0
reg0($zero) = 0
reg1($t0) = 45
reg2($t1) = 0
reg3($t2) = 3
reg4($a0) = 0
reg5($a1) = 55
reg6($a2) = 0
reg7($s0) = 0
```

```
reg8($s1) = 1c
reg9($s2) = 0
reg10($s3) = 0
reg11($s4) = 0
reg12($gp) = 0
reg13($sp) = 0
reg14($pc) = 22
reg15($ra) = 0
Int_reg=0
Int_pc=0
********************************
```

Listagem 7.3: Média aritmética de três dados convertidos pelo ADC

Devido à dificuldade em desenvolver aplicações em hexadecimal, visto que as ferramentas do RISC16 ainda não se encontram disponíveis, as simulações da plataforma virtual SoC ficaram restritas às apresentadas aqui. Tão logo tais ferramentas sejam disponibilizadas, a plataforma virtual pode continuar a ser utilizada no desenvolvimento e teste do *software* embarcado, que foi um dos seus propósitos principais.

## 7.2 Aplicações para o RSoC

Para a plataforma virtual RSoC apresentada no capítulo 6, as aplicações simuladas consistiram na execução de compressão JPEG em imagens no formato PGM com e sem a utilização do RoSA, e na comunicação de um nó composto pelo RSoC com um nó funcional da biblioteca SCNSL, conforme será descrito a seguir. É importante ressaltar que outras aplicações foram testadas na plataforma virtual, mas não serão descritas aqui. Os arquivos estão disponibilizados no CD em anexo.

### 7.2.1 Aplicação com o APS e o RoSA: Compressão JPEG

Como uma das maiores preocupações em redes de sensores sem fio é o consumo, e como estudos mostram que a transmissão de informação via *wireless* é geralmente a tarefa que mais consome energia nesses tipos de sistema [92], um esquema de compressão é desejável para reduzir a quantidade de dados que trafegam pela rede. O fato do RSoC possuir sensor de imagem motivou a implementação de uma aplicação de processamento de imagens para validar o modelo construído.

O algoritmo de compressão JPEG opera em três estágios sucessivos: transformada DCT, quantização de coeficientes e compressão. Esses passos combinados formam um poderoso compressor capaz de comprimir imagens em até 10% do seu tamanho original [93].

O algoritmo em linguagem C de uma compressão JPEG apresentado em [93] foi modificado e adaptado para ser executado na plataforma virtual RSoC, conforme mostra a Figura 7.3. Inicialmente, o algoritmo foi executado apenas no *mipsUnB*. Em seguida, o algoritmo foi alterado para que parte dele fosse executada pelo RoSA, em conjunto com o *mipsUnB*. Imagens no formato PGM, com diferentes números de pixels, foram usadas na simulação e os resultados preliminares

utilizando o MIPS-I do ArchC foram comparados entre si [16, 94].

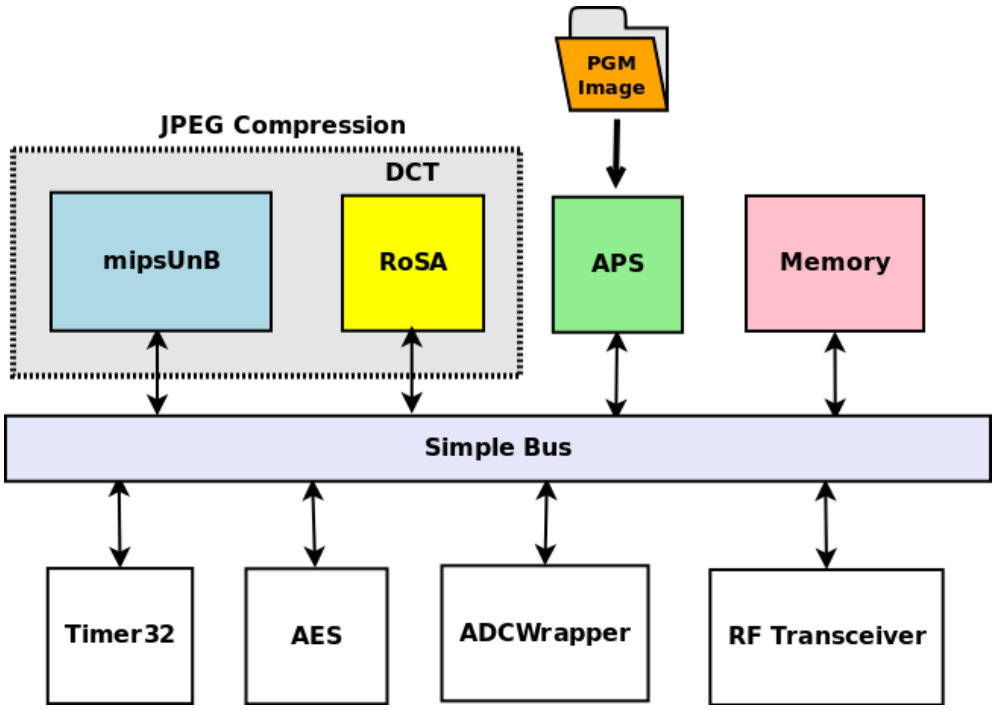


Figura 7.3: Aplicação JPEG na plataforma virtual RSoC

O primeiro passo consistiu em escolher qual estágio do algoritmo seria mapeado no RoSA. Considerando que blocos reconfiguráveis como o RoSA possuem boa performance para operações repetitivas e regulares, o estágio de multiplicação de matrizes 8x8 da DCT foi escolhido.

O RoSA possui apenas 48 registradores globais e 48 registradores modificados, totalizando 96 registradores para executar as operações. O número não é suficiente para uma multiplicação de matrizes 8x8. Por isso, a matriz resultante foi dividida em 4 partes: 6x6, 6x2, 2x6 e 2x2. O cálculo da maior parte (6x6) e a matriz resultante são mostrados na Figura 7.4, para dois ciclos de operações. O mesmo procedimento foi utilizado para calcular os valores finais da matriz.

Como cada célula suporta metade das operações necessárias para se obter cada valor da matriz resultante (cada célula possui 8 entradas, e são necessárias 16), duas células foram usadas, conforme mostra a Figura 7.5. Após as multiplicações, os valores são adicionados dentro da célula. Como os resultados de ambas as células ainda precisam ser somados, esta última operação foi executada no processador, pois, pelo fato de ser uma operação simples, não justificou o uso de outro procedimento de configuração para ser executada no RoSA.

Dois arquivos monocromáticos no formato PGM foram comprimidos: file1.pgm (160x160 pixels) e file2.pgm (32x32 pixels). A Tabela 7.1 mostra a redução obtida utilizando o algoritmo.

Os parâmetros usados para avaliar a performance com e sem o RoSA foram o tempo de simulação (em segundos) do código e o número de instruções (em milhões de instruções) executadas pelo processador. Os resultados finais são mostrados na Tabela 7.2.

É possível notar que o uso do RoSA permite uma grande redução tanto no tempo de simulação

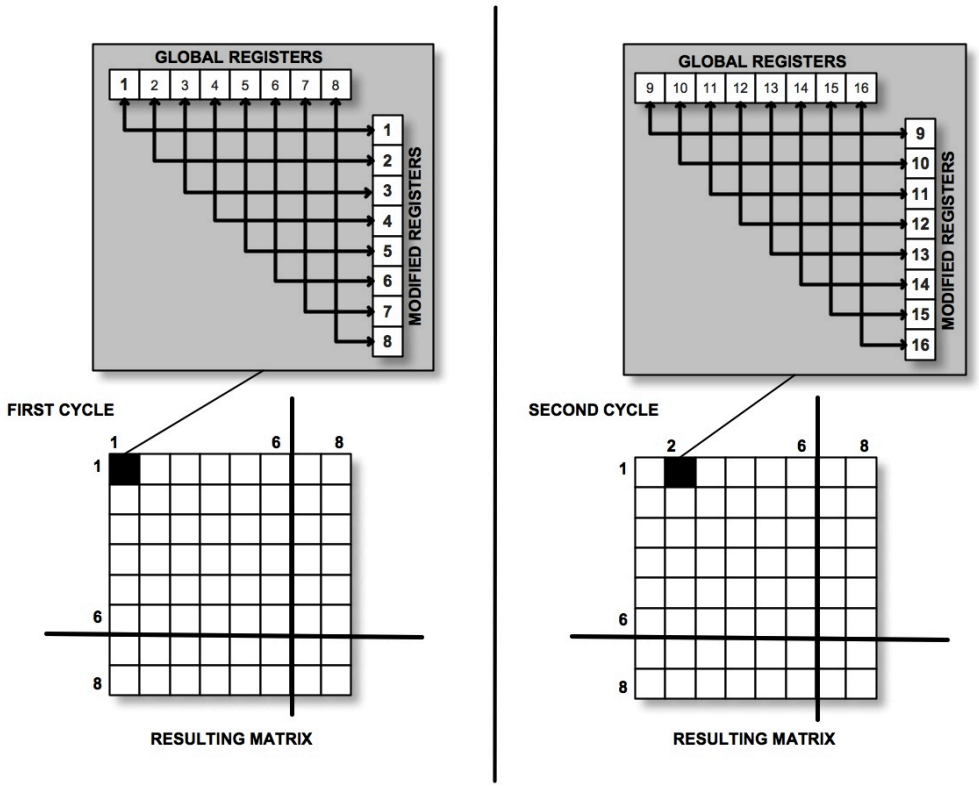


Figura 7.4: Ciclos do cálculo da matriz resultante 6x6 [16]

quanto no número de instruções executadas, mostrando que a inclusão desta arquitetura no RSoC pode ser uma boa escolha para a execução de estágios de funções DSP.

Este estudo de caso também mostrou algumas limitações da arquitetura do RoSA, tais como o número de registradores para entrada de dados e de células, que restringe os algoritmos que podem ser executados na mesma. Apesar disso, o uso do RoSA proporcionou um aumento significativo na velocidade de execução do algoritmo mapeado.

Esses resultados foram apresentados em [95] e mostram também que a plataforma virtual modelada pode ser utilizada para testar software embarcado para o RSoC mesmo sem ter o hardware disponível e, adicionalmente, permite que a equipe de *hardware* avalie o impacto do *software* na arquitetura e detecte limitações que, de outra forma, só seriam descobertas em estágios posteriores.

Tabela 7.1: Compressão JPEG [16]

| **Imagem** | **Fator de quantização** | **Tamanho inicial (bytes)** | **Tamanho final (bytes)** | **Redução de tamanho** |
|---|---|---|---|---|
| file1 | 1 | 25600 | 7727 | 70% |
| file1 | 10 | 25600 | 1876 | 93% |
| file2 | 1 | 1024 | 144 | 86% |
| file2 | 10 | 1024 | 144 | 94% |

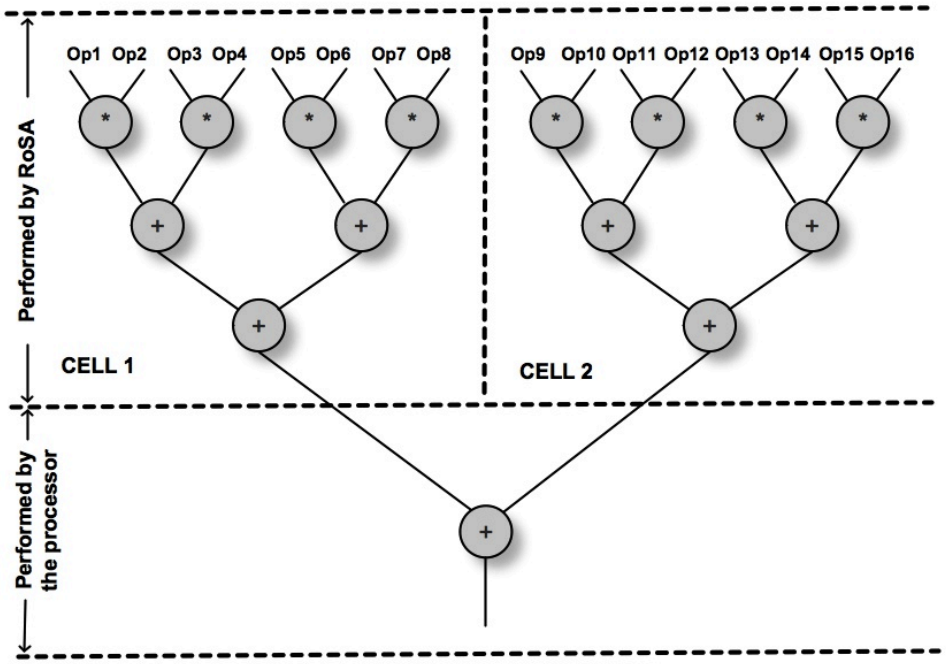


Figura 7.5: Operações executadas para obtenção dos valores finais da matriz resultante [16]

Tabela 7.2: Compressão JPEG com e sem o RoSA [16]

| **Imagem** | **Fator de quantização** | **Tempo(s) (MIPS)** | **Tempo(s) (MIPS+RoSA)** | **Instr.(x$10^6$) (MIPS)** | **Instr.(x$10^6$) (MIPS+RoSA)** |
|---|---|---|---|---|---|
| file1 | 1 | 98,72 | 15,95 | 470,41 | 68,83 |
| file1 | 10 | 96,77 | 13,84 | 460,87 | 58,29 |
| file2 | 1 | 3,81 | 0,67 | 17,15 | 2,40 |
| file2 | 10 | 3,67 | 0,64 | 17,04 | 2,28 |

### 7.2.2 Aplicação com ADC-AMS e RSSF

Para testar a comunicação entre um nó composto por blocos da plataforma virtual RSoC e um nó funcional, foi simulado o sistema mostrado na Figura 7.6. Resultados preliminares foram descritos em [96]. Um programa compilado para o MIPS é carregado na memória e executado. O código primeiramente habilita as interrupções. Em seguida, o processador é colocado no modo de baixo consumo. Um laço infinito é executado enquanto não há requisições de interrupções. Quando o ADC realiza uma conversão, ou quando o nó funcional envia um pacote através da rede sem fio e é recebido pelo transceptor, o laço é interrompido e o processador inicia uma rotina de tratamento de interrupções. Nesta rotina, o processador volta ao modo normal de operação e realiza ações de acordo com a causa da interrupção.

A Figura 7.7 mostra as formas de onda geradas pelo ADC. Nota-se que um dado é convertido a

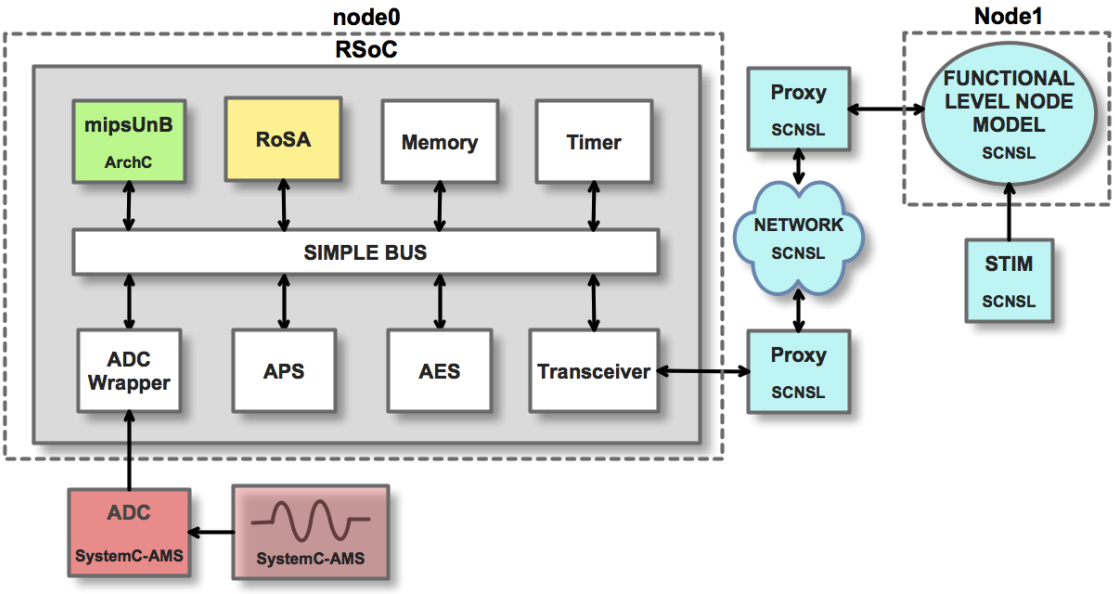


Figura 7.6: Diagrama de blocos mostrando comunicação entre o SoC e um nó funcional

cada 1s e, neste caso, uma interrupção causada pelo ADC é gerada. O primeiro dado convertido foi 0 (zero), o segundo foi 0x4Bh (ou 01001011 em binário, ou 75 em decimal) e o terceiro foi 0x7Ah (ou 01111010 em binário ou 122 em decimal).

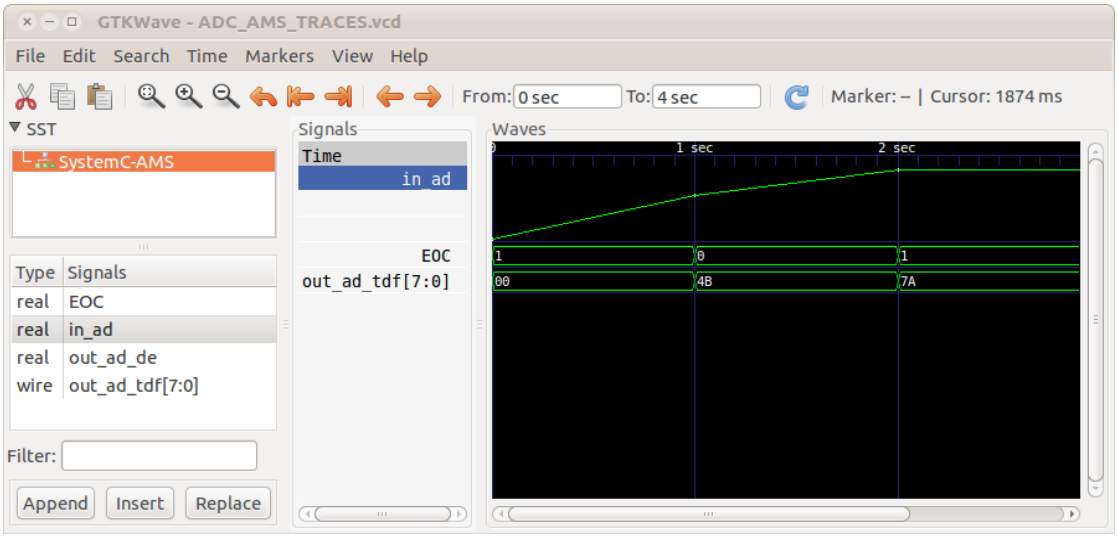


Figura 7.7: Formas de onda geradas pelo ADC em SystemC-AMS

Os resultados podem ser verificados pelas informações de *debug*, conforme pode ser visto nas listagens a seguir. Na Listagem 7.4 pode-se ver o segundo dado convertido pelo ADC e a interrupção gerada. O programa, após identificar a causa da interrupção (0x4h, que é o código para identificar o ADC como causa), vai para o endereço 0x8h, que contém uma instrução de salto para o endereço do início da rotina de tratamento de interrupção.

```
DBG: ----------------------------------------------------
DBG: Simulation Time: 1000483800 ns
DBG: Starting instruction Execution
DBG: ------ PC=0x584 NPC=0x580 ------ 130449
DBG: nop
Received data from ADC SystemC-AMS: 01001011
1000500 us0
DBG: ----------------------------------------------------
DBG: >>>>>>>>>>>>>>>>>>>>>>>>>>><<<<<<<<<<<<<<<<<<<<<<<<<<
DBG: Simulation Time: 1000500 us
DBG: MIPS: Processor interrupted.
DBG: NPC (0x584) stored.

1000500 us: Processor: ADC Interrupt.
DBG: CAUSE (0x4) stored.
DBG: Interrupts disabled. CONFIG (0) stored.
DBG: Jumping to interrupt vector address (0x8).
DBG: >>>>>>>>>>>>>>>>>>>>>>>>>>><<<<<<<<<<<<<<<<<<<<<<<<<<
DBG: ----------------------------------------------------
DBG: ----------------------------------------------------
DBG: Simulation Time: 1000560600 ns
DBG: Starting instruction Execution
DBG: ------ PC=0x8 NPC=0xc ------ 130450
DBG: j 354
DBG: Target = 0x588
DBG: ----------------------------------------------------
```

Listagem 7.4: Interrupção gerada pelo segundo dado convertido pelo ADC

A rotina *interrupt_handler* atua de acordo com a causa da interrupção. Caso a mesma seja gerada pelo ADC, o valor convertido é armazenado em um registrador e, em seguida, o nó que contém o RSoC (*node0*) envia o dado armazenado via transceptor para o nó funcional (*Node1*). Assim que recebe o dado, o *Node1* envia um dado de volta (*ack* = 0x11112222h ou 286335522 em decimal) para o *node0*. O transceptor do *node0*, então, gera uma interrupção ao receber o dado correspondente ao *ack*, conforme pode ser visto na Listagem 7.5. De maneira análoga ao que ocorreu na interrupção gerada pelo ADC, o programa identifica a causa da interrupção (0x1h, que é o código para identificar o transceptor como causa) e vai para o endereço 0x8h, que contém uma instrução de salto para o endereço do início da rotina de tratamento de interrupção.

```
1002290400 ns: node0.transceiver: sending a radio packet.

(...)

1002802400 ns: Node1: reading a radio packet: data = 75

(...)

DBG: ----------------------------------------------------
DBG: >>>>>>>>>>>>>>>>>>>>>>>>>>><<<<<<<<<<<<<<<<<<<<<<<<<<
DBG: Simulation Time: 1003315400 ns
```

```
DBG: MIPS: Processor interrupted.
DBG: NPC (0x580) stored.
1003315400 ns: node0.transceiver: reading a radio packet: data = 286335522.

1003315400 ns: Processor: RF Interrupt.
DBG: CAUSE (0).
DBG: CAUSE (0x1) stored.
DBG: Interrupts disabled. CONFIG (0) stored.
DBG: Jumping to interrupt vector address (0x8).
DBG: >>>>>>>>>>>>>>>>>>>>>>>>>>>><<<<<<<<<<<<<<<<<<<<<<<<<<<<
DBG: ------------------------------------------------------------
DBG: Simulation Time: 1003329 us
DBG: Starting instruction Execution
DBG: ------ PC=0x8 NPC=0xc ------ 130870
DBG: j 354
DBG: Target = 0x588
DBG: ------------------------------------------------------------
```

Listagem 7.5: Envio e recebimento dos dados e interrupção gerada pelo transceptor

Com isso, foram apresentados os resultados das simulações executadas em ambas as plataformas virtuais, cujo funcionamento pode ser verificado a partir da análise dos relatórios de *debug* e das formas de onda geradas.

O anexo I contém um tutorial que mostra como instalar as ferramentas necessárias para a execução das plataformas virtuais - incluindo a biblioteca gerada *LDCI_ Modeling_ Library* - e das aplicações para o RSoC. O anexo II mostra os procedimentos necessários para realizar as simulações mostradas neste capítulo. Por fim, o CD em anexo contém os arquivos, que estão listados no anexo III.

O próximo capítulo apresenta as discussões relativas à modelagem das plataformas virtuais e às simulações realizadas, e também uma discussão geral acerca das abordagens utilizadas em outros trabalhos.

# Capítulo 8

# Discussão

*Neste capítulo, são feitas considerações acerca da simulação dos modelos e das vantagens e desvantagens da abordagem utilizada na modelagem dos SoCs em relação a outras abordagens encontradas na literatura.*

## 8.1 Discussão da Modelagem e Simulações

### 8.1.1 Plataforma Virtual SoC

A plataforma virtual SoC possui uma arquitetura simples e já foi implementada em *hardware*. Por isso, a mesma foi escolhida como estudo de caso para a aplicação da metodologia descrita neste trabalho. Em [13], foram relatados a simulação VHDL do RISC16 e o procedimento para escrever programas de teste na memória e executá-los no modelo. Embora tenha servido para fins de validação da arquitetura, o modelo oferece limitações para executar as aplicações, tais como dificuldade para verificar os resultados devido à quantidade de sinais que devem ser monitorados. Com o modelo em SystemC TLM apresentado neste trabalho, é possível não só executar as aplicações usando ferramentas *open source* como também descrever os módulos analógicos em SystemC-AMS, permitindo descrever o comportamento de tais módulos com mais acurácia sem perda de desempenho na simulação.

Enquanto o procedimento de co-simulação SystemC/VHDL-AMS apresentou muitas dificuldades relacionadas à ferramenta e à sincronização (vide [87]), a simulação com SystemC e SystemC-AMS confirmou as vantagens dessa abordagem já relatadas no capítulo 2.

As aplicações que foram simuladas nesta plataforma virtual foram de baixa complexidade devido à ausência de um compilador para o RISC16. Os programas foram escritos em hexadecimal e o procedimento se mostrou bastante trabalhoso. Tão logo o compilador esteja disponível, a plataforma virtual pode continuar a ser usada no desenvolvimento de programas mais elaborados.

### 8.1.2 Plataforma Virtual RSoC

O *hardware* do RSoC ainda está em processo de desenvolvimento. O nível de complexidade é maior que o do SoC, pois já envolve um processador de 32 bits funcionando em conjunto com

blocos reconfiguráveis. Este último aspecto já permite elaborar aplicações cuja execução pode ser realizada pelo processador e pelo RoSA. O procedimento de mapeamento das aplicações (ou de parte delas) no RoSA ainda é manual na plataforma virtual RSoC e isso limitou a escolha da aplicação que foi utilizada na simulação. Mesmo assim, foi possível efetuar as análises que foram propostas inicialmente neste trabalho para demonstrar que o uso do RoSA na arquitetura proposta para o RSoC é vantajoso, já que o mesmo proporcionou um aumento significativo na velocidade de execução do algoritmo testado.

O ADC utilizado na plataforma virtual RSoC é o mesmo descrito na plataforma virtual SoC. O transceptor de RF foi descrito de maneira a permitir a comunicação do mesmo com outros nós em um ambiente de rede. Isso aumenta a abrangência do modelo de maneira a permitir o desenvolvimento e teste de aplicações de RSSF utilizando o RSoC como nó e efetuando comunicações com outro RSoC ou com nós funcionais.

Os blocos citados acima fazem do modelo RSoC uma plataforma virtual de sinais mistos, com blocos reconfiguráveis e que pode ser simulada em ambiente de rede. As aplicações executadas serviram para verificar o funcionamento da plataforma virtual, visto que envolveram todos os blocos modelados e possuíam resultados esperados conhecidos, como no caso da compressão JPEG, cuja saída é a imagem de entrada com tamanho reduzido.

## 8.2 Discussão Geral

Até a data de elaboração desta tese, os trabalhos encontrados que envolvem a modelagem em alto nível de SoCs mistos para RSSF apresentam, dentre outras, as seguintes características:

- Utilização de ferramentas proprietárias para realização de co-simulação de modelos mistos, o que restringe o seu uso a tais ferramentas (por exemplo, [88]);
- Descrição de SoC para RSSF incluindo blocos em SystemC-AMS, mas com simulação restrita a apenas dois nós (vide [24]);
- Utilização de SystemC e NS-2 para simulação de ambientes de redes, mas com a necessidade de procedimentos mais complexos do que a utilização da SCNSL, como por exemplo a alteração do núcleo das ferramentas (vide [97] e [98]).

A abordagem proposta neste trabalho combina o uso de SystemC-AMS para descrever o ADC como módulo analógico com o uso da biblioteca SCNSL para permitir a comunicação entre nós, enquanto outras ferramentas como NS-2 e Opnet são geralmente utilizadas para este último caso, mas não permitem avaliar o impacto da arquitetura do nó na rede e vice-versa.

Apesar de em [24] terem sido apresentadas simulações entre dois nós que incluem blocos descritos em SystemC-AMS, a comunicação não possui os recursos oferecidos pela biblioteca SCNSL, que facilita a realização de simulações com inclusão de mais nós. Nesta tese, a aplicação também se resumiu à comunicação entre dois nós (um nó com o RSoC e um nó funcional). Entretanto, em [15] foram apresentadas simulações com vários nós que utilizam os mesmos princípios aqui

apresentados, sendo que a rede era composta de um ou mais nós com um SoC menos complexo, mas de arquitetura parecida com as das plataformas virtuais modeladas aqui, e um ou mais nós funcionais. A adaptação das plataformas virtuais SoC e RSoC para compor RSSF mais complexas fica como sugestão de trabalhos futuros.

Outras simulações podem ser realizadas para validar aplicações para RSSF nas quais tarefas como aquisição de dados analógicos, processamento digital de sinais e comunicação sem fio são executadas, mas a implementação de tais aplicações fogem ao escopo deste trabalho.

As principais contribuições foram a implementação de modelos de plataformas virtuais de um SoC e de um RSoC mistos utilizando SystemC TLM, ArchC, SystemC-AMS e SCNSL na descrição dos módulos, que podem se comunicar em ambientes de redes, e a geração de uma biblioteca de blocos descritos em alto nível que podem ser usados em ferramentas abertas e reutilizados em projetos futuros. As plataformas virtuais apresentadas permitem o desenvolvimento, teste e otimização de *software* embarcado, bem como constituem um modelo de referência para as etapas subsequentes do desenvolvimento do *hardware*. Essas tarefas podem ser realizadas em um estágio inicial do fluxo de projeto, reduzindo custos e o tempo de projeto.

Apesar das vantagens, algumas limitações podem ser citadas em relação às plataformas virtuais. A sua utilização ainda oferece dificuldades, como a ausência de uma interface gráfica e a necessidade de recompilação para cada mudança efetuada nos blocos ou no sistema. O desenvolvimento de aplicações que utilizam os blocos reconfiguráveis também oferece dificuldades devido à necessidade de mapeamento manual das configurações das células. Por fim, no caso da simulação de RSSF, faltam parâmetros importantes, como modelos de propagação, ruídos e frequências de transmissão diferentes, o que dificulta a mudança das condições do ambiente simulado.

# Capítulo 9

# Conclusões

Neste trabalho foi apresentada a modelagem de SoCs voltados para aplicações de RSSF. Os objetivos eram: desenvolvimento de uma biblioteca de modelos incluindo blocos digitais, reconfiguráveis e analógicos em alto nível de abstração, usando as linguagens de descrição de *hardware* SystemC e SystemC-AMS; realização de uma avaliação preliminar da funcionalidade do *hardware* dos SoCs e do *software* embarcado a partir da integração desses modelos em duas plataformas virtuais, sendo uma de um SoC e outra de um RSoC, e envolvendo a interação entre dois MoCs diferentes; realização de simulações envolvendo aplicações de referência, com resultados previamente conhecidos, para fins de verificação do correto funcionamento das plataformas virtuais.

Pode-se concluir que, além dos objetivos iniciais terem sido atingidos, os resultados do trabalho tornaram possível estender a abordagem para a representação de redes de nós inteligentes, incluindo possíveis aplicações [15]. A biblioteca de modelos é *open source* e poderá ser utilizada em projetos futuros, utilizando a metodologia aqui proposta.

As principais contribuições foram a implementação de modelos de plataformas virtuais de um SoC e de um RSoC mistos utilizando SystemC TLM, ArchC, SystemC-AMS e SCNSL na descrição dos módulos, que podem se comunicar em ambientes de redes, e a geração de uma biblioteca de blocos descritos em alto nível que podem ser usados em ferramentas abertas e reutilizados em projetos futuros. As plataformas virtuais apresentadas permitem o desenvolvimento, teste e otimização de *software* embarcado, bem como constituem um modelo de referência para as etapas subsequentes do desenvolvimento do *hardware*. Essas tarefas podem ser realizadas em um estágio inicial do fluxo de projeto, reduzindo custos e o tempo de projeto.

A abordagem adotada neste trabalho proporcionou várias vantagens. Em primeiro lugar, as duas plataformas virtuais permitem a simulação de aplicações, resolvendo as limitações encontradas nos trabalhos anteriores de desenvolvimento dos SoCs aqui descritos. Elas também oferecem a possibilidade de análise do fluxo de dados entre os blocos em estágios iniciais do projeto. O resultado foi uma melhoria do fluxo tradicional de projeto que era utilizado anteriormente pela nossa equipe, que não previa uma etapa de modelagem em alto nível para permitir o desenvolvimento concorrente de *hardware* e *software*.

Outro ponto forte é que, como os modelos estão em nível transacional, a simulação é mais rápida em relação à dos modelos em nível RTL, já que a quantidade de detalhes da arquitetura é

menor. O uso do MoC TDF do SystemC-AMS para descrição de módulos analógicos em conjunto com modelos em TLM não torna a simulação mais lenta, já que o mesmo possui o recurso de agendamento estático, que é realizado antes do início da simulação. O procedimento de integração é mais simples do que o do VHDL-AMS pelo fato das extensões SystemC-AMS serem baseadas no padrão SystemC. Além disso, a possibilidade de representar sinais de valores discretos ou contínuos amostrados no tempo permite a descrição do comportamento analógico e oferece maior acurácia às plataformas virtuais do que a representação dos módulos utilizando apenas o domínio DE do *kernel* do SystemC.

O modelo de uma estrutura reconfigurável é utilizado em uma das plataformas virtuais. Com a plataforma virtual RSoC, é possível mapear aplicações no RoSA e fazer análises preliminares de desempenho em termos de tempo de simulação e número de instruções executadas pelo processador.

A plataforma virtual RSoC também permite a utilização dos recursos da biblioteca SCNSL para simulação em ambiente de rede. Apesar de só ter sido simulada uma rede com dois nós, é possível simular o nó com o RSoC interagindo em rede com nós funcionais ou com outros nós RSoC, juntamente com diversas aplicações, utilizando o mesmo procedimento apresentado em [15]. Além disso, a metodologia pode ser usada no projeto de outros tipos de sistema, além de RSSF.

A abordagem aqui adotada difere das demais encontradas na literatura em termos de combinação de recursos e ferramentas utilizadas que proporcionam as vantagens listadas acima. O uso de SystemC-AMS, ArchC e da biblioteca SCNSL permitiu a implementação de plataformas virtuais mistas que podem executar aplicações em ambientes de RSSF. Até a finalização da redação desta tese, não foi encontrado nenhum modelo de SoC com sensor de imagem, blocos reconfiguráveis e capacidade de se comunicar com outros nós para realizar uma comparação em termos de resultados de simulações.

Algumas limitações também foram observadas. Uma delas é o fato de ainda não ser possível estimar um dos parâmetros mais críticos em RSSF, que é o consumo de energia, a partir dos modelos implementados. Além disso, faltam modelos de propagação, ruídos e frequências de transmissão diferentes. Esses aspectos podem ser implementados como uma continuação deste trabalho.

Outra limitação é a ausência de interface gráfica para utilização. Os modelos ainda oferecem dificuldades para mudar as condições do ambiente simulado, pois a cada mudança dos módulos, é necessária uma recompilação.

Para a continuação do trabalho, tendo em vista a abrangência do mesmo, sugere-se a utilização das plataformas virtuais em RSSF mais complexas, incluindo mais de dois nós e implementando-se algoritmos de roteamento. Simulações com vários nós onde ao menos um deles é representado por um SoC, ou mesmo onde todos os nós são SoCs, foram efetuadas em [15] e comparadas com outras simulações nas quais foram utilizadas outras abordagens, tendo como parâmetros o tempo de simulação. Verificou-se que a abordagem utilizada nesse trabalho proporcionou uma maior velocidade de simulação.

Seria desejável também descrever todos os blocos de *hardware* do transceptor de RF em SystemC-AMS. A biblioteca *TUV_AMS_Library* oferece um conjunto de modelos que podem ser customizados e instanciados na plataforma virtual, de maneira similar à utilizada na instanciação

do ADC.

Por fim, sugere-se a elaboração de modelos em níveis mais baixos de abstração tendo como base os modelos elaborados neste trabalho, de maneira a ser possível uma análise de desempenho para tomadas de decisões em termos de arquitetura de *hardware* e também realizar a síntese lógica e física para a prototipagem do *hardware*. Algumas ferramentas, como o *C-to-Silicon* da Cadence [88], já permitem a realização de síntese automática a partir de descrições em SystemC. Como exemplo, sugere-se a utilização de *transactors* para a geração de modelos *cycle-accurate* de ambas as plataformas virtuais descritas aqui e a inclusão de um modelo de barramento, como o AMBA, na plataforma virtual RSoC.

# REFERÊNCIAS BIBLIOGRÁFICAS


[1] FUMMI, F. et al. Flexible energy-aware simulation of heterogeneous wireless sensor networks. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. [S.l.: s.n.], 2009. p. 1638–1643.

[2] JANTSCH, A.; SANDER, I. Models of computation and languages for embedded system design. *IEE Proceedings Computers and Digital Techniques*, v. 152, n. 2, p. 114 – 129, 2005.

[3] GAJSKI S. ABDI, A. G. D. D.; SCHIRNER, G. *Embedded System Design: Modeling, Synthesis and Verification*. [S.l.]: Springer Verlag, 2009.

[4] BLACK, D. C. et al. *SystemC: from the ground up*. [S.l.]: Springer Verlag, 2009.

[5] OSCI. *OSCI TLM-2.0 User Manual*. 2010. Disponível em: <http://www.systemc.org>.

[6] OSCI. *SystemC AMS extensions User Guide*. 2010. Disponível em: <http://www.systemc-ams.org>.

[7] ARCHC - The Architecture Description Language. [Online; acessado em outubro–2010]. Disponível em: <http://archc.sourceforge.net/>.

[8] HERRERA, F.; VILLAR, E. A framework for embedded system specification under different models of computation in systemc. In: *Proceedings of the 43rd annual Design Automation Conference*. [S.l.: s.n.], 2006. p. 911–914.

[9] EKER, J. et al. Taming heterogeneity-the ptolemy approach. *Proceedings of the IEEE*, IEEE, v. 91, n. 1, p. 127–144, 2003.

[10] SANDER, I.; JANTSCH, A.; LU, Z. Development and application of design transformations in forsyde. In: IET. *Computers and Digital Techniques, IEE Proceedings-*. [S.l.], 2003. v. 150, n. 5, p. 313–20.

[11] ZHU, J.; SANDER, I.; JANTSCH, A. Hetmoc: Heterogeneous modelling in systemc. In: *Proceedings of the Forum on Design Langauges (FDL)*. [S.l.: s.n.], 2010.

[12] GHENASSIA, F. *Transaction-level modeling with Systemc: TLM concepts and applications for embedded systems*. [S.l.]: Springer Verlag, 2005.

[13] COSTA, J. D. *Implementação de um Processador RISC 16 bits CMOS num Sistema em Chip*. Dissertação de Mestrado do Departamento de Eng. Elétrica, Universidade de Brasília: [s.n.], 2004.

[14] PEREIRA, M. M.; OLIVEIRA, B. C. d.; SILVA, I. S. Rosa: a reconfigurable stream-based architecture. In: *Proceedings of the 20th Annual Conference on Integrated Circuits and Systems Design, Rio de Janeiro, Brazil*. [S.l.: s.n.], 2007.

[15] MEDEIROS, J. E. G. de. *Modelagem em SystemC de um Sistema em Chip em ambiente de redes de sensores sem fio*. Dissertação de Mestrado do Departamento de Eng. Elétrica, Universidade de Brasília: [s.n.], 2010.

[16] CARNEIRO, J. L. C. *Modelagem de APS e Mapeamento de Aplicações em uma Plataforma RSoC Virtual*. Trabalho de Graduação do Departamento de Eng. Elétrica, Universidade de Brasília: [s.n.], 2009.

[17] GRÖTKER S. LIAO, G. M. T.; SWAN, S. *System design with SystemC*. [S.l.]: Springer Netherlands, 2002.

[18] FRANKE, H. et al. Introduction to the wire-speed processor and architecture. In: *IBM Journal of Research and Development*. [S.l.: s.n.], 2010. v. 54, n. 1, p. 3.

[19] BUCK, J. et al. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal in Computer Simulation*, v. 4, n. 2, p. 155–182, 1994.

[20] SYSML. [Online; acessado em outubro–2010]. Disponível em: <http://www.sysml.org>.

[21] OSCI. *Functional Specification for SystemC 2.0*. [Online; acessado em outubro–2010]. Disponível em: <http://www.systemc.org>.

[22] IEEE Standard VHDL Analog and Mixed-Signal Extensions. 2007.

[23] VERILOG-AMS. [Online; acessado em outubro–2010]. Disponível em: <http://www.vhdl.org/verilog-ams>.

[24] VASILEVSKI, M. et al. Efficient and refined modeling of wireless sensor network nodes using systemc-ams. In: *Research in Microelectronics and Electronics*. [S.l.: s.n.], 2008. p. 81–84.

[25] SYSTEMC-AMS and Design of Embedded Mixed-Signal Systems. [Online; acessado em outubro–2010]. Disponível em: <http://www.systemc-ams.org/>.

[26] DAMM, M.; HAASE, J.; GRIMM, C. Co-simulation of mixed hw/sw and analog/rf systems at architectural level. In: *IEEE International Behavioral Modeling and Simulation Workshop*. [S.l.: s.n.], 2008. p. 84–89.

[27] MCCANNE, S.; FLOYD, S. *NS Network Simulator - Version 2*. [Online; acessado em outubro–2010]. Disponível em: <http://www.isi.edu/nsnam/ns>.

[28] OPNET. [Online; acessado em outubro–2010]. Disponível em: <http://www.opnet.com>.

[29] FUMMI, F.; QUAGLIA, D.; STEFANNI, F. A systemc-based framework for modeling and simulation of networked embedded systems. In: *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*. [S.l.: s.n.], 2008. p. 49 – 54.

[30] SILVA, A. L. *Desenvolvimento de Software para a Comunicação de Sistemas Integrados em Chip (SoC) Aplicados a Sensores para Controle de Irrigação*. Trabalho de Graduação do Departamento de Eng. Elétrica, Universidade de Brasília: [s.n.], 2005.

[31] LUCHINE, G.; PICANCO, R.; ROMARIZ, A. R. S. Reliability of traffic on sensor networks using simple protocols. In: *14th International Conference on Software, Telecommunications and Computer Networks*. [S.l.: s.n.], 2006.

[32] MARTINS, A. J. O. *Ambiente de Desenvolvimento de Softwares para um SoC: Desenvolvimento de um Simulador em SystemC*. Trabalho de Graduação do Departamento de Eng. Elétrica, Universidade de Brasília: [s.n.], 2005.

[33] ZHENG, J.; JAMALIPOUR, A. *Wireless Sensor Networks: A Networking Perspective*. [S.l.]: Springer Netherlands, 2009.

[34] RUIZ, L. B. et al. Arquitetura de redes de sensores sem fio. In: . [S.l.]: XXII Simpósio Brasileiro de Redes de Computadores (SBRC), 2004. p. 167–218.

[35] GLASER, J. et al. Investigating power-reduction for a reconfigurable sensor interface. In: *Proceedings of the Austrian National Conference on the Design of Integrated Circuits and Systems, Graz, Austria*. [S.l.: s.n.], 2009. v. 7.

[36] HEMPSTEAD, M.; WEI, G.; BROOKS, D. An accelerator-based wireless sensor network processor in 130nm cmos. In: *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*. [S.l.: s.n.], 2009. p. 215–222.

[37] WOLF, W. *Modern vlsi design: system-on-chip design*. [S.l.]: Prentice Hall Press Upper Saddle River, NJ, USA, 2002.

[38] THEODORIDIS, G.; SOUDRIS, D.; VASSILIADIS, S. A survey of coarse-grain reconfigurable architectures and cad tools. In: *Fine-and Coarse-Grain Reconfigurable Computing*. [S.l.]: Springer, 2007. p. 89–98.

[39] HARTENSTEIN, R. A decade of reconfigurable computing: a visionary retrospective. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. [S.l.: s.n.], 2001. p. 642–649.

[40] GOKHALE, M.; GRAHAM, P. *Reconfigurable computing: Accelerating computation with field-programmable gate arrays*. [S.l.]: Springer Verlag, 2005.

[41] GREENBAUM, J. Reconfigurable logic in soc systems. In: *Proceedings of the IEEE Custom Integrated Circuits Conference*. [S.l.: s.n.], 2002. p. 5–8.

[42] QU, Y.; TIENSYRJA, K.; SOININEN, J. Systemc-based design methodology for reconfigurable system-on-chip. In: *Proceedings of the 8th Euromicro Conference on Digital System Design*. [S.l.: s.n.], 2005. p. 364–371.

[43] BECKER, J.; PIONTECK, T.; GLESNER, M. Adaptive systems-on-chip: architectures, technologies and applications. In: *Proceedings of the 14th symposium on Integrated circuits and systems design*. [S.l.: s.n.], 2001.

[44] BERGMANN, N. Enabling technologies for reconfigurable system-on-chip. In: *IEEE International Conference on Field-Programmable Technology*. [S.l.: s.n.], 2002. p. 360–363.

[45] VOROS, N. et al. System-on-a-chip methodology for telecom applications. In: *URSI International Symposium on Signals, Systems, and Electronics*. [S.l.: s.n.], 1998. p. 321–325.

[46] MORENO, E. I.; RODOLFO, T. A.; CALAZANS, N. L. V. Modelagem e descrição de socs em diferentes níveis de abstração. In: *X Workshop Iberchip, Cartagena*. [S.l.: s.n.], 2004. p. 1–11.

[47] VACHOUX, A.; GRIMM, C.; EINWICH, K. Systemc-ams requirements, design objectives and rationale. In: *Proceedings of the conference on Design, Automation and Test in Europe*. [S.l.: s.n.], 2003.

[48] EINWICH, K.; UHLE, T. *SystemC-AMS holistic analog, digital, hardware and software system-level modeling*. [Online; acessado em outubro–2010]. Disponível em: <http://www.edatechforum.com/>.

[49] BREITMAN, K.; ANIDO, R. *Atualizações em Informática*. [S.l.]: Editora PUC Rio, 2006.

[50] HERRERA, F. et al. Heterogeneous specification with hetsc and systemc-ams: Widening the support of mocs in systemc. *Embedded Systems Specification and Design Languages*, Springer, p. 107–121, 2008.

[51] SANGIOVANNI-VINCENTELLI, A. Quo vadis, SLD? reasoning about the trends and challenges of system level design. *Proceedings of the IEEE*, IEEE, v. 95, n. 3, p. 467–506, 2007.

[52] FORSYDE. [Online; acessado em outubro–2010]. Disponível em: <http://www.ict.kth.se/forsyde>.

[53] CENNI, F.; SIMEU, E.; MIR, S. Macro-modeling of analog blocks for systemc-ams simulation: A chemical sensor case-study. In: *17th IFIP International Conference on Very Large Scale Integration (VLSI-SoC)*. [S.l.: s.n.], 2009.

[54] VARMA, A. et al. Modeling heterogeneous socs with systemc: a digital/mems case study. In: ACM. *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. [S.l.], 2006. p. 54–64.

[55] GRIMM, C. et al. Refinement of mixed-signal systems with systemc. In: IEEE. *Design, Automation and Test in Europe Conference and Exhibition, 2003*. [S.l.], 2003. p. 1170–1171.

[56] DAMM, M. et al. Connecting systemc-ams models with osci tlm 2.0 models using temporal decoupling. In: IEEE. *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*. [S.l.], 2008. p. 25–30.

[57] DRAFT Standard SystemC AMS Extensions Language Reference Manual. 2008.

[58] BALDWIN, P. et al. *Visualsense: Visual modeling for wireless and sensor network systems*. [S.l.]: Citeseer, 2004.

[59] COSTA, J. e. a. Cmos soc for irrigation control. In: *Proceedings of the IEEE International SOC Conference*. [S.l.: s.n.], 2005. p. 51–54.

[60] BENíCIOJR, G. M. *Projeto de Microprocessador RISC 16 Bits para Sistema de Comunicação sem Fio em Chip*. Dissertação de Mestrado do Departamento de Eng. Elétrica, Universidade de Brasília: [s.n.], 2002.

[61] COSTA, J. D. *Desenvolvimento de uma Unidade Lógica Aritmética para um Sistema de Comunicação sem Fio em Chip*. Trabalho de Graduação do Departamento de Eng. Elétrica, Universidade de Brasília: [s.n.], 2002.

[62] ARAúJO, W. A. *Proposta de Adaptação de um Núcleo de Processador RISC 16 Bits CMOS ao Padrão VSIA para Propriedade Intelectual de Semicondutor*. Trabalho de Graduação do Departamento de Eng. Elétrica, Universidade de Brasília: [s.n.], 2006.

[63] BESERRA, G. S. *Projeto de Estruturas de Armazenamento Digital em um SoC para Controle de Irrigação*. Dissertação de Mestrado do Departamento de Eng. Elétrica, Universidade de Brasília: [s.n.], 2004.

[64] ARAúJO, G. M. de. *Implementação dos Circuitos Eletrônicos de uma Memória SRAM de um SoC CMOS*. Trabalho de Graduação do Departamento de Eng. Elétrica, Universidade de Brasília: [s.n.], 2004.

[65] MEDEIROS, J. E. G. d. et al. Characterization of a rom test structure designed for a soc for irrigation control. In: *7th Microelectronics Students Forum, SBMicro*. [S.l.: s.n.], 2007.

[66] LINDER, R. R. *Software Básico para Processador num Sistema em Chip (SoC)*. Dissertação de Mestrado do Departamento de Eng. Elétrica, Universidade de Brasília: [s.n.], 2002.

[67] POVOA, L. S. R. *SoC para controle de Irrigação: Ambiente para o Desenvolvimento de Aplicações*. Dissertação de Mestrado do Departamento de Eng. Elétrica, Universidade de Brasília: [s.n.], 2005.

[68] DINIZ, J. Z. F. *Desenvolvimento de Aplicação Básica para Sistemas de Comunicação*. Trabalho de Graduação do Departamento de Eng. Elétrica, Universidade de Brasília: [s.n.], 2002.

[69] RANGEL, R. M. D. *Desenvolvimento de um Programa Montador para um Processador RISC 16 bits*. Trabalho de Graduação do Departamento de Eng. Elétrica, Universidade de Brasília: [s.n.], 2002.

[70] ISHIHARA, G. L. *Protocolo de Comunicação para uma Rede de Sensores sem Fio: Teoria e Implementação em SoC*. Dissertação de Mestrado do Departamento de Eng. Elétrica, Universidade de Brasília: [s.n.], 2006.

[71] COSTA, J. D. et al. Módulo ip de um processador para aplicações embarcadas sem fio. In: *IX IBERCHIP Workshop, Havana, Cuba*. [S.l.: s.n.], 2003.

[72] COSTA, J. D. et al. Projeto de estruturas de um processador risc para aplicação em um soc para controle de irrigação. In: *X IBERCHIP Workshop, Cartagena de Indias, Colombia*. [S.l.: s.n.], 2004.

[73] MEDEIROS, J. E. G. de. *Implementação Final do módulo de Saída de um Transceptor de RF a 900MHz em SoC CMOS*. Trabalho de Graduação do Departamento de Eng. Elétrica, Universidade de Brasília: [s.n.], 2007.

[74] MADUREIRA, H. M. G. *Projeto de Oscilador Controlado por Tensão para Transceptor RF 900MHz Embarcado em SoC*. Trabalho de Graduação do Departamento de Eng. Elétrica, Universidade de Brasília: [s.n.], 2008.

[75] MADUREIRA, H. M. G.; MEDEIROS, J. E. G. d.; COSTA, J. C. d. Ring voltage controlled oscillators for an rf transceiver. In: *Proceedings of the 8th Microelectronics Student Forum, SBMicro*. [S.l.: s.n.], 2008.

[76] MADUREIRA, H. M. G.; MEDEIROS, J. E. G. d.; COSTA, J. C. d. Design and characterization of a 900mhz lc voltage controlled cmos oscillator. In: *Proceedings of the 9th Microelectronics Student Forum, SBMicro*. [S.l.: s.n.], 2009.

[77] NEVES, L. C. *Projeto de um Sintetizador de Frequência em 920MHz para um Receptor CMOS Embarcado em um SoC*. Trabalho de Graduação do Departamento de Eng. Elétrica, Universidade de Brasília: [s.n.], 2010.

[78] SOARES, V. F. et al. Cmos a/d converter for soc in wireless sensor network application. In: *Semetro, João Pessoa, Paraíba*. [S.l.: s.n.], 2009.

[79] PIMENTEL, J. V. B.; COSTA, J. C. d. A methodology for describing analog/mixed-signals blocks as ip. In: *18th IP-Embedded Systems Conference*. [S.l.: s.n.], 2009.

[80] PIMENTEL, J. V. B. *Metodologia para Descrição de Células Analógicas como IP*. Dissertação de Mestrado do Departamento de Eng. Elétrica, Universidade de Brasília: [s.n.], 2009.

[81] MARRA, J. C. S. da S. *Projeto de Interface Serial para Microprocessador RISC de 16 Bits*. Trabalho de Graduação do Departamento de Eng. Elétrica, Universidade de Brasília: [s.n.], 2004.

[82] OPENCORES. [Online; acessado em outubro–2010]. Disponível em: <http://opencores.org/>.

[83] DICKINSON, A. et al. Standard cmos active pixel image sensors for multimedia applications. In: *16th Conference on Advanced Research in VLSI*. [S.l.: s.n.], 1995.

[84] EID, E.; AY, S.; FOSSUM, E. Design of radiation tolerant cmos aps system-on-a-chip image sensors. In: *Aerospace Conference Proceedings, IEEE*. [S.l.: s.n.], 2002.

[85] BRAGA, L. H. C. e. a. Layout techniques for radiation hardening of standard cmos active pixel sensors. In: *Proceedings of the 20th Annual Conference on Integrated Circuits and Systems Design, Rio de Janeiro, Brazil*. [S.l.: s.n.], 2007.

[86] OU, J. et al. *TU Vienna SystemC AMS Communications Library Documentation*.

[87] BESERRA, G. S. et al. Tlm and vhdl-ams co-simulation of a system on chip for wireless sensor networks. In: *Proceedings of the 16th Workshop Iberchip, Iguaçu Falls, Brazil*. [S.l.: s.n.], 2010.

[88] CADENCE Design Systems, Inc. [Online; acessado em outubro–2010]. Disponível em: <http://www.cadence.com>.

[89] OFFICIAL website of the Joint Photographic Experts Group. [Online; acessado em outubro–2010]. Disponível em: <http://www.jpeg.org/>.

[90] AMBA Specification Rev 2.0. [Online; acessado em outubro–2010]. Disponível em: <http://www.arm.com/>.

[91] EUSSE, J.; HUBNER, M.; JACOBI, R. Brick: a multi-context expression grained reconfigurable architecture. In: ACM. *Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design: Chip on the Dunes*. [S.l.], 2009. p. 1–6.

[92] HILL, J. e. a. In: *ACM SIGPLAN Notices*. [S.l.: s.n.]. v. 35.

[93] NELSON, M.; GAILLY, J.-L. *The Data Compression Book, 2nd Ed.* [S.l.]: M&T Books, 1995.

[94] BESERRA, G. S. e. a. System-level modeling of a reconfigurable system on chip for wireless sensor networks applications. In: *The 3rd International Conference on Intelligent and Advanced Systems, Kuala Lumpur, Malaysia*. [S.l.: s.n.], 2010.

[95] BESERRA, G. S. e. a. Transaction-level modeling of a reconfigurable system on chip for wireless sensor networks applications. *Journal of Communication and Computer, USA*, 2010.

[96] BESERRA, G. S. et al. System-level modeling of a mixed-signal system on chip for wireless sensor networks. In: *Design, Automation and Test in Europe*. [S.l.: s.n.], 2011.

[97] FUMMI, F. et al. Modeling and simulation of mobile gateways interacting with wireless sensor networks. In: EUROPEAN DESIGN AND AUTOMATION ASSOCIATION. *Proceedings of the conference on Design, automation and test in Europe: Designers' forum*. [S.l.], 2006. p. 106–111.

[98] FUMMI, F. et al. Heterogeneous co-simulation of networked embedded systems. In: IEEE COMPUTER SOCIETY. *Proceedings of the conference on Design, automation and test in Europe*. [S.l.], 2004. p. 30168.

# ANEXOS

# I. INSTALAÇÃO DAS FERRAMENTAS DE MODELAGEM

As seções a seguir indicam como instalar no **Ubuntu** as ferramentas necessárias para compilar e executar os modelos apresentados neste trabalho. Os arquivos citados se encontram no CD anexado à tese.

## I.1 SystemC

- Instalar o pacote *build-essential* e atualizar o sistema:

  ```
  $ sudo apt-get install build-essential
  ```

  ```
  $ sudo apt-get update
  ```

- Baixar o arquivo *systemc-2.2.0.tgz* em *www.systemc.org*

- Para executar os passos de instalação, é necessário ter permissão de escrita nos diretórios. Logo, deve-se efetuar os passos como *root* (#) ou alterar as permissões para o usuário.

- Copiar o arquivo em */usr/local* e descompactar:

  ```
  # cd /usr/local
  ```

  ```
  # gzip -d systemc-2.2.0.tgz
  ```

  ```
  # tar -xvf systemc-2.2.0.tar
  ```

- Editar o arquivo *systemc-2.2.0/src/sysc/utils/sc_utils_ids.cpp*, incluindo no início:

  ```
  #include "string.h"
  ```

  ```
  #include "cstdlib"
  ```

- Instalar:

  ```
  # export CXX=g++
  ```

  ```
  # cd systemc-2.2.0
  ```

  ```
  # mkdir objdir
  ```

  ```
  # cd objdir
  ```

  ```
  # ../configure
  ```

  ```
  # make
  ```

  ```
  # make install
  ```

  ```
  # make check
  ```

- Verificar se os testes rodaram corretamente após o último comando. Se sim, o SystemC está corretamente instalado e funcionando. Se não, rever os passos anteriores.

- Para executar um pequeno exemplo, como usuário comum ($) copiar o arquivo *sc_ tutorial.tgz* na pasta do usuário (ex: */home/fulano*) e descompactar:

  ```
  $ gzip -d sc_tutorial.tgz
  $ tar -xvf sc_tutorial.tar
  $ cd sc_tutorial
  ```

- Editar o arquivo *Makefile.defs* e setar a variável *SYSTEMC* para:

  ```
  SYSTEMC = /usr/local/systemc-2.2.0
  ```

- Compilar o exemplo:

  ```
  $ cd simpleCounterSC
  $ make -f Makefile.osci
  ```

- Verificar se foi gerado o arquivo *run.exe* e executá-lo:

  ```
  $ ./run.exe
  ```

- Visualizar as formas de onda. Para isso, é preciso ter o programa *GTKWave* instalado.

  ```
  $ gtkwave trace.vcd &
  ```

## I.2 TLM 2.0 e 1.0

- Não é necessário fazer nenhuma compilação. Basta baixar o arquivo *TLM-2.0.1.tgz* em *www.systemc.org*, copiar em */usr/local* e descompactar.

  ```
  # cd /usr/local
  # tar -xvzf TLM-2.0.1.tgz
  ```

- OBS: Se for utilizar o *ArchC*, é necessário ter também o *TLM1.0*. Basta baixar o arquivo *TLM-1.0.tgz* e repetir o passo acima.

## I.3 SystemC-AMS

- Baixar o arquivo *systemc-ams-1.0BETA1.tar.gz* em *www.systemc-ams.org*

- Descompactar o arquivo em */usr/local*:

  ```
  # cd /usr/local
  # gzip -d systemc-ams-1.0BETA1.tar.gz
  # tar -xvf systemc-ams-1.0BETA1.tar
  ```

- Mudar para o diretório *systemc-ams-1.0BETA1*:

  ```
  # cd systemc-ams-1.0BETA1
  ```

- Setar a variável *SYSTEMC_PATH*:

```
# export SYSTEMC_PATH=/usr/local/systemc-2.2.0
```

- Instalar:

```
# ./configure
# make
# make install
```

- Setar a variável *SYSTEMC_AMS_PATH*:

```
# export SYSTEMC_AMS_PATH=/usr/local/systemc-ams-1.0BETA1
```

- Para executar um pequeno exemplo, como usuário comum ($) copiar o arquivo *integ.tar.gz* na pasta do usuário (ex: */home/fulano*) e descompactar:

```
$ gzip -d integ.tar.gz
$ tar -xvf integ.tar
$ cd integ
```

- Compilar:

```
$ make -f Makefile.osci
```

- O arquivo *tb_integ.exe* deve ser gerado. Executar:

```
$ ./tb_integ.exe
```

- Os arquivos *trace_u* e *trace_y* devem ser gerados. Para visualizá-los, pode-se utilizar o *Matlab* ou o *Octave*.

## I.4 TUV_AMS_Library

- Baixar o arquivo *TUV_AMS_Library.tar.gz* em:

  *http://www.systemc-ams.org/download/TUV_AMS_Library.tar.gz*

- Verificar se o *cmake* está instalado. Caso contrário, instalar.

- Como *root* (#), copiar o arquivo para */usr/local* e descompactar:

```
# cd /usr/local
# tar -xvzf TUV_AMS_Library.tar.gz
```

- Mudar para o diretório *TUV_AMS_Library*:

```
# cd TUV_AMS_Library
```

- Setar as variável *SYSTEMC_HOME* e *SYSTEMCAMS_HOME*:

```
# export SYSTEMC_HOME = /usr/local/systemc-2.2.0
# export SYSTEMCAMS_HOME = /usr/local/systemc-ams-1.0BETA1
```

- Executar o *script build.sh*:

  ```
  # sh build.sh
  ```

- Os arquivos serão instalados na pasta *output*.

- Para executar um exemplo, como usuário comum ($) copiar o arquivo *qpsk-transceiver.tar.gz* na pasta do usuário (ex: */home/fulano*) e descompactar:

  ```
  $ tar -xvzf qpsk-transceiver.tar.gz
  ```

  ```
  $ cd qpsk-transceiver
  ```

- Verificar as variáveis do *Makefile*:

  ```
  SYSTEMC = /usr/local/systemc-2.2.0
  ```

  ```
  SYSTEMCAMS = /usr/local/systemc-ams-1.0BETA1
  ```

  ```
  TUV_LIB = /usr/local/TUV_AMS_Library/output
  ```

- Compilar:

  ```
  $ make -f Makefile
  ```

- O arquivo *run.exe* deve ser gerado. Executar:

  ```
  $ ./run.exe
  ```

- Ajustar os parâmetros da simulação e comparar as saídas tendo como base o arquivo *QPSK_ transmitter.pdf*. O arquivo *tr.vcd* deve ser gerado. Para visualizá-lo, pode-se utilizar o *gtkwave*:

  ```
  $ gtkwave tr.vcd
  ```

- Antes de executar uma nova simulação, executar o *clean*:

  ```
  $ make clean
  ```

## I.5 ArchC

- Instalar antes os pacotes *Bison* e *Flex*:

  ```
  $ sudo apt-get install bison
  ```

  ```
  $ sudo apt-get install flex
  ```

- Baixar os seguintes arquivos no site *http://archc.sourceforge.net/*

  - *ArchC*: acsim 2.0 (OBS: O *mipsUnB* foi gerado com o *ArchC 2.0* e NÃO funciona com a versão 2.1 !!!)
  - *Processor Models*: MIPS-I 0.7.8
  - *Compilers for target applications*: mips-elf-tools.tgz
  - (Opcional) *Application binaries*: Qualquer *benchmark* ou programa do *MIPS-I*

- Descompactar o *ArchC*:

  ```
  $ tar -xvzf archc-2.0.tar.gz
  ```

- Instalar:

  ```
  $ cd archc-2.0
  $ ./configure -with-systemc=/usr/local/systemc-2.2.0 -with-tlm=/usr/local/TLM-2005-04-08/
  $ make
  $ sudo make install
  ```

- Gerar os arquivos de um dos processadores (neste caso, o *MIPS*):

  ```
  $ tar -xvzf mips1-v0.7.8.tgz
  $ cd mips-v0.7.8
  $ acsim mips1.ac -abi
  $ make -f Makefile.arch
  ```

- Testar o modelo com uma das aplicações binárias (por exemplo, o *Quick Sort* para o *MIPS*):

  ```
  $ tar -xvzf qsort.tar.gz
  $ cd qsort/bin
  $ (caminho de onde o mips-v0.7.8 foi descompactado)/mips1.x -load=qsort_small
  ```

- Para usar a toolchain do *MIPS*, basta descompactar a pasta no diretório */l/archc/compilers* e atualizar a variável *PATH*:

  - Criar o diretório */l/archc/compilers*:

    ```
    $ cd /
    $ sudo mkdir l
    $ sudo mkdir l/archc
    $ sudo mkdir l/archc/compilers
    $ cd /l/archc/compilers
    ```

  - Copiar o arquivo *mips-elf-tools.tgz* para o diretório criado e descompactar:

    ```
    $ sudo cp (diretório onde o arquivo mips-elf-tools.tgz está)/mips-elf-tools.tgz
    .
    $ sudo tar -xvzf mips-elf-tools.tgz
    ```

  - Editar o arquivo oculto **.bashrc**, que está no diretório do usuário (por exemplo, */home/fulano*). Acrescentar a linha:

    ```
    export PATH=$PATH:/l/archc/compilers/bin
    ```

  - Salvar e atualizar o terminal:

    ```
    $ source .bashrc
    ```

  - Verificar se os comandos estão disponíveis (por exemplo, digitando *mips-elf-gcc* no terminal).

## I.6 SCNSL

- Baixar o arquivo em *http://sourceforge.net/projects/scnsl/*
- Descompactar e entrar no diretório:

  `$ tar -xvzf scnsl-beta.tgz`

  `$ cd scnsl`
- Editar os seguintes campos do *Makefile*:

  `SYSTEMC:=/usr/local/systemc-2.2.0`

  `SYSTEMC-TLM:=/usr/local/TLM-2009-07-15`

  `INSTALL_DIR:=(Diretório de instalação.  Pode-se deixar o que já está lá.)`
- Instalar:

  `$ make all_test doc doc_dev`
- Editar o arquivo *.bashrc* e incluir o caminho das pastas *lib*. Substituir *user* pelo seu usuário.

  `$ gedit /home/user/.bashrc`

  Incluir no final do arquivo a linha:

  `export LD_LIBRARY_PATH=/usr/local/scnsl/lib:/usr/local/scnsl/scnsl_models/lib`

  Salvar e carregar:

  `$ source /home/user/.bashrc`
- Testar com o exemplo *scnsl_ test_ linux*:

  `$ tar -xvzf scnsl_test_linux.tar.gz`

  `$ cd scnsl_test_linux`

  Editar o *script compile_ linux.sh* colocando o caminho da pasta *scnsl* (por exemplo, */home/-fulano*) no lugar de (*DIRETORIO DO SCNSL*) e executá-lo:

  `$ ./compile_linux.sh`

  Executar o arquivo binário gerado:

  `$ ./test.x`

  OBS: Caso haja algum erro de permissão (*permission denied*), alterar a permissão do arquivo ou pasta em questão utilizando o comando *chmod*.
- OPTATIVO: Para executar os exemplos da SCNSL separadamente, descompactar o arquivo *scnsl_ tests* e executar o *Makefile* de cada um. Por exemplo:

  `$ tar scnsl_tests.tar.gz`

  `$ cd scnsl_tests`

  `$ cd bottleneck`

  `$ make`

  `$ ./bottleneck.x`

## I.7 Biblioteca *LDCI_Modeling_Library*

Esta biblioteca contém os blocos que foram modelados para desenvolver as plataformas virtuais dos SoCs que estão sendo implementados no LDCI. Os blocos podem ser instanciados em novas plataformas também.

A versão v0.1 está estruturada da seguinte forma:

- ip: pasta que contém os periféricos.

  *ac_tlm_AES*: módulo AES

  *ac_tlm_APS*: sensor de imagem APS

  *ac_tlm_generic_8bit_memory*: memória genérica de 8 bits

  *ac_tlm_int_generator*: módulo para gerar interrupções no mipsUnB

  *ac_tlm_int_monitor*: módulo para monitorar interrupções no mipsUnB

  *ac_tlm_memory*: memória utilizada na plataforma virtual RSoC

  *ac_tlm_RoSA*: bloco reconfigurável do RSoC

  *ac_tlm_timer32*: timer de 32 bits

  *ac_tlm_transceiver*: transceptor implementado para comunicação com os blocos da SCNSL

  *scnsl_functional_node*: nó funcional da SCNSL

- lib: pasta que contém o arquivo **.a**.

- processors: pasta com os microprocessadores modelados.

  *mipsUnB*: processador mips1 gerado pelo ArchC e modificado

  *tlm2_risc16*: processador RISC de 16 bits

- *wrappers*: pasta com *wrappers* para conectar os blocos da biblioteca com outros domínios (por exemplo, SystemC-AMS).

  *ac_tlm_adc_wrapper*: *wrapper* para conectar o ADC em SystemC-AMS usando o protocolo ArchC TLM1

  *tlm2_adc_wrapper*: *wrapper* para conectar o ADC em SystemC-AMS usando TLM2

Para instalar, basta utilizar o procedimento descrito a seguir.

- Como *root*, copiar o arquivo *LDCI_Modeling_Library_v0.1.tar.gz* para a pasta */usr/local*

  ```
  # cd /usr/local
  # cp <caminho_do_arquivo>/LDCI_Modeling_Library_v0.1.tar.gz .
  ```

- Descompactar e entrar no diretório:

  ```
  # tar -xvzf LDCI_Modeling_Library_v0.1.tar.gz
  # cd LDCI_Modeling_Library_v0.1
  ```

- Abrir o *Makefile*:

  ```
  # gedit Makefile &
  ```

- Verificar e editar(se necessário) os seguintes campos do *Makefile* e salvar:

  ```
  SYSTEMC = /usr/local/systemc-2.2.0

  TLM1 = /usr/local/TLM-2005-04-08

  TLM2 = /usr/local/TLM-2009-07-15

  SYSTEMCAMS = /usr/local/systemc-ams-1.0BETA1

  TUV_LIB = /usr/local/TUV_AMS_Library/output

  ARCHC = /usr/local

  SCNSL = /usr/local/scnsl
  ```

- Instalar:

  ```
  # make -f Makefile
  ```

- Com isso, será criada uma pasta *lib* com o arquivo **libldci_modeling_library.a**.

# II. UTILIZAÇÃO DAS FERRAMENTAS DE MODELAGEM

Neste anexo, dois exemplos são apresentados, mostrando a utilização do *mipsUnB*. Os exemplos foram compilados via *Makefile* e no ambiente Eclipse CDT. Os arquivos citados se encontram no CD anexado à tese.

## II.1 *mipsUnB_interrupt_test*

Este exemplo faz parte do trabalho apresentado em [15]. Trata-se de uma plataforma virtual que inclui o *mipsUnB*, uma memória genérica, um barramento simples, um *timer* de 32 bits, um módulo AES e algumas aplicações.

### II.1.1 *Makefile*

Para executar a simulação usando o *Makefile*, copiar da pasta *using_makefiles* o arquivo *mipsUnB_interrupt_test.tar.gz*, descompactar e mudar para o diretório *mipsUnB_interrupt_test*:

```
$ tar -xvzf mipsUnB_interrupt_test.tar.gz
```

```
$ cd mipsUnB_interrupt_test
```

Abrir o *Makefile* e editar os campos das seguintes variáveis [1]:

```
SYSTEMC = /usr/local/systemc-2.2.0
```

```
TLM1 = /usr/local/TLM-2005-04-08
```

```
ARCHC = /usr/local
```

```
LDCI = /usr/local/LDCI_Modeling_Library_v0.1
```

Abrir o terminal e executar o comando:

```
$ make -f Makefile
```

Após a compilação, o arquivo executável *mipsUnB_interrupt_test.x* será gerado. Assim, basta escolher uma aplicação na pasta *soft* e executar o comando com a opção $--load$:

```
$ ./mipsUnB_interrupt_test.x --load=soft/<nome_da_aplicação>.x
```

Para salvar a saída em um arquivo *log*, usar o comando:

```
$ ./mipsUnB_interrupt_test.x --load=soft/<nome_da_aplicação>.x &> output.log
```

Para interromper a simulação, basta utilizar **CTRL+C**.

[1] Se as ferramentas foram instaladas conforme indicado no Anexo I, não será necessário editar as variáveis, visto que os caminhos foram mantidos conforme sugerido nas instruções de instalação lá apresentados.

### II.1.2 Eclipse CDT

Copiar o arquivo *mipsUnB_interrupt_test.tar.gz* da pasta *using_eclipse* e descompactar:

```
$ tar -xvzf mipsUnB_interrupt_test.tar.gz
```

Abrir o Eclipse e importar o projeto utilizando o menu **File -> Import**.

Na janela **Import**, escolher **General -> Existing Projects into Workspace** e clicar em **Next**. Em **Select root directory**, clicar em **Browse** e indicar o caminho de onde o *mipsUnB_interrupt_test* foi descompactado (por exemplo, */home/user/projetos/mipsUnB_interrupt_test*). Marcar **Copy projects into workspace** e clicar em **Finish**. Com isso, o projeto *mipsUnB_interrupt_test* aparecerá na aba **Project Explorer**.

Antes de compilar, deve-se alterar as propriedades do mesmo. Basta clicar com o botão direito do mouse em *mipsUnB_interrupt_test* e selecionar **Properties**. Em seguida, selecionar **C/C++ Build -> Settings**.

Na aba **Tool Settings**, selecionar **GCC C++ Compiler -> Includes** e verificar os **Include paths**. Eles devem indicar os caminhos para os *headers* do SystemC, TLM1.0 (note que o arquivo *tlm.h* está em *.../TLM-2005-04-08/tlm*), *ArchC* (se o *ArchC* foi instalado de acordo com este tutorial, está em */usr/local/include*), da biblioteca LDCI e da pasta *include* do *workspace* do *mipsUnB_interrupt_test*. Por exemplo, vide Figura II.1.

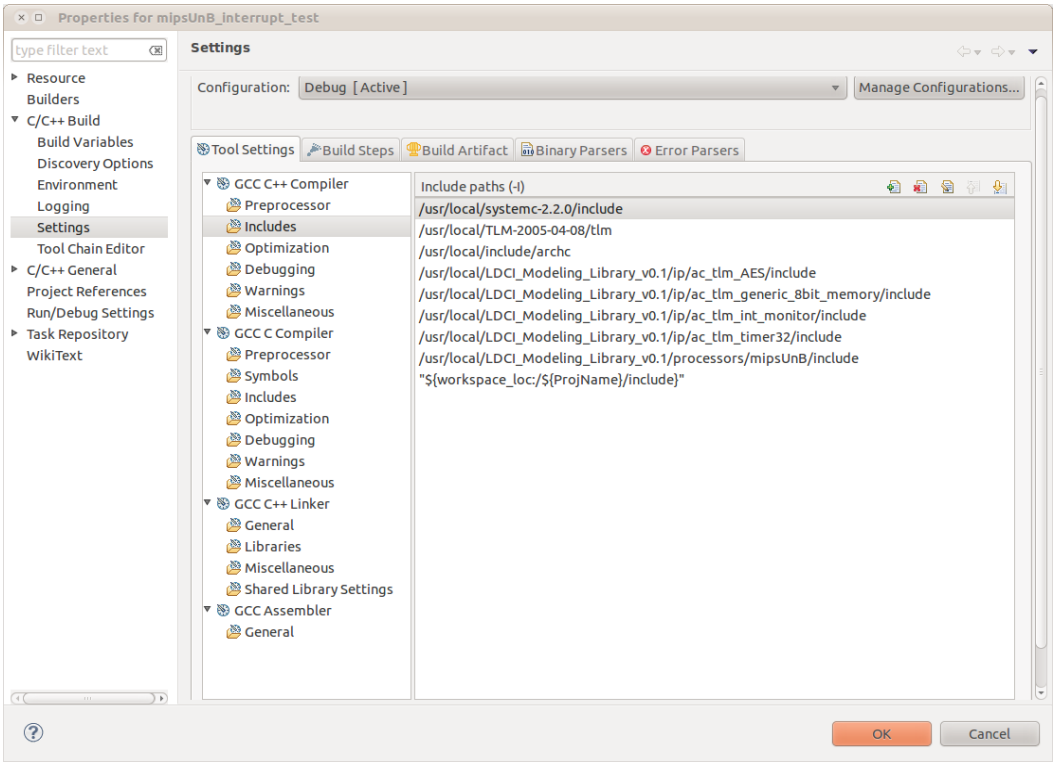


Figura II.1: *GCC C++ Compiler - Includes* do *mipsUnB_interrupt_test*

A Figura II.2 mostra as **Libraries** do **GCC C++ Linker**. Neste exemplo, adicionar as 3 que são utilizadas, **respeitando a ordem**: *ldci_modeling_library*, *systemc* e *archc*. Em **Library**

**search path**, adicionar os caminhos para os respectivos arquivos (não necessariamente na mesma ordem), que são: *libldci_modeling_library.a*, *libsystemc.a* (no caso do *Linux 32 bits*, está em *.../systemc-2.2.0/lib-linux*) e *libarchc.a* (se o *ArchC* foi instalado de acordo com este tutorial, está em */usr/local/lib*). Clicar em **Ok**.

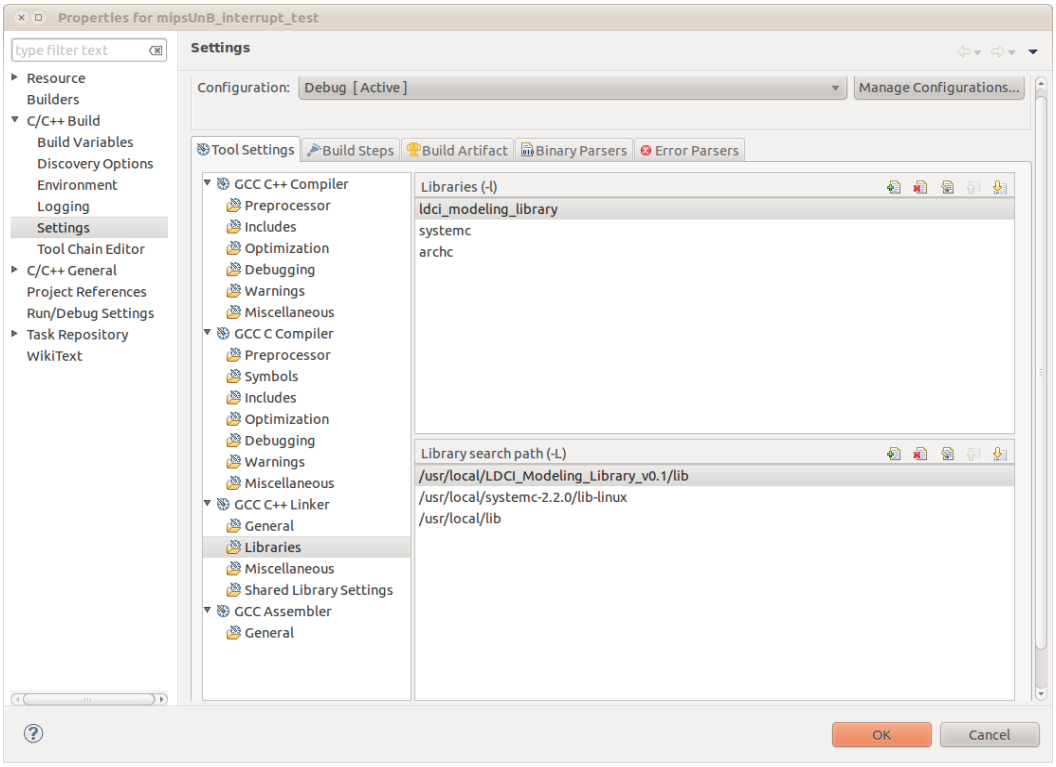


Figura II.2: *GCC C++ Linker - Libraries* do *mipsUnB_interrupt_test*

Compilar o projeto usando o menu **Project -> Clean**. Observe que a pasta *soft* deve ser excluída do **Build**. Caso haja mais de um projeto aberto, selecionar **Clean projects selected below** e marcar *mipsUnB_interrupt_test*.

Caso não haja nenhum erro, será gerado o arquivo executável *mipsUnB_interrupt_test.x* na pasta *Debug* do projeto *mipsUnB_interrupt_test*. Este arquivo será utilizado para executar as aplicações da pasta *soft*. Os arquivos que serão executados na plataforma virtual possuem a extensão **.x** e são gerados pelo *cross-compiler* do *MIPS* (ou seja, que foram compilados utilizando o comando **mips-elf-gcc**).

Para executar a aplicação *aes* da pasta *soft*, basta abrir um terminal no diretório *Debug* do projeto *mipsUnB_interrupt_test* e executar o arquivo *mipsUnB_interrupt_test.exe* com a opção *−−load*. Por exemplo:

```
$ cd /home/user/workspace/mipsUnB_interrupt_test/Debug
```

```
$ ./mipsUnB_interrupt_test.x −−load=../soft/aex.x
```

A tela deverá mostrar os dados da simulação, conforme pode ser visto na Listagem II.1.

```
        Copyright (c) 1996-2006 by all Contributors
                     ALL RIGHTS RESERVED
ArchC: Reading ELF application file: ../soft/aes.x

DBG: @@@ begin behavior @@@
ArchC: ---------------------- Starting Simulation ----------------------

DBG: ------------------------------------------------
DBG: Simulation Time: 0 s
DBG: Starting instruction Execution
DBG: ------ PC=0 NPC=0x4 ------ 0
DBG: j 4
DBG: Target = 0x10
DBG: ------------------------------------------------
DBG: Simulation Time: 62500 ps
DBG: Starting instruction Execution
DBG: ------ PC=0x4 NPC=0x10 ------ 1
DBG: nop
DBG: ------------------------------------------------
DBG: Simulation Time: 125 ns
DBG: Starting instruction Execution
DBG: ------ PC=0x10 NPC=0x14 ------ 2
DBG: ori r29, r0, 49152
DBG: Result = 0xc000
...
```

Listagem II.1: Exemplo de simulação do mipsUnB_interrupt_test

Para interromper a simulação, basta digitar **CTRL+C**.

## II.2 *mipsUnB_soc_sink*

Este exemplo também faz parte do trabalho apresentado em [15]. A plataforma virtual inclui o *mipsUnB*, uma memória genérica, um barramento simples, um *timer* de 32 bits, um transceptor de RF e um nó funcional da biblioteca SCNSL.

### II.2.1 *Makefile*

Para executar a simulação usando o *Makefile*, copiar da pasta *using_makefiles* o arquivo *mipsUnB_soc_sink.tar.gz*, descompactar e mudar para o diretório *mipsUnB_soc_sink*:

```
$ tar -xvzf mipsUnB_soc_sink.tar.gz
```

```
$ cd mipsUnB_soc_sink
```

Abrir o *Makefile* e editar os campos das seguintes variáveis [2]:

```
SYSTEMC = /usr/local/systemc-2.2.0
```

[2]Se as ferramentas foram instaladas conforme indicado no Anexo I, não será necessário editar as variáveis, visto que os caminhos foram mantidos conforme sugerido nas instruções de instalação lá apresentados.

```
TLM1 = /usr/local/TLM-2005-04-08

ARCHC = /usr/local

SCNSL = /usr/local/scnsl

LDCI = /usr/local/LDCI_Modeling_Library_v0.1
```

Abrir o terminal e executar o comando:

```
$ make -f Makefile
```

Após a compilação, o arquivo executável *mipsUnB_soc_sink.x* será gerado. Assim, basta escolher uma aplicação na pasta *soft* e executar o comando com a opção *−−load*:

```
$ ./mipsUnB_soc_sink.x −−load=soft/<nome_da_aplicação>.x
```

Para salvar a saída em um arquivo *log*, usar o comando:

```
$ ./mipsUnB_soc_sink.x −−load=soft/<nome_da_aplicação>.x &> output.log
```

Para interromper a simulação, basta utilizar **CTRL+C**.

### II.2.2 Eclipse CDT

Copiar o arquivo *mipsUnB_soc_sink.tar.gz* da pasta *using_eclipse* e descompactar:

```
$ tar -xvzf mipsUnB_soc_sink.tar.gz
```

Importar o projeto no Eclipse seguindo o mesmo procedimento do item anterior.

Alterar as propriedades do projeto clicando com o botão direito do mouse em *mipsUnB_soc_sink* e selecionando **Properties**.

A Figura II.3 mostra os **Include paths**.

A Figura II.4 mostra as **Libraries** do **GCC C++ Linker**.

Compilar o projeto usando o menu **Project -> Clean**. Caso haja mais de um projeto aberto, selecionar **Clean projects selected below** e marcar *mipsUnB_soc_sink*.

Caso não haja nenhum erro, será gerado o arquivo executável *mipsUnB_soc_sink* na pasta *Debug* do projeto *mipsUnB_soc_sink*. Este arquivo será utilizado para executar as aplicações da pasta *soft*. Os arquivos que serão executados na plataforma virtual possuem a extensão **.x** e são gerados pelo *cross-compiler* do *MIPS* (ou seja, que foram compilados utilizando o comando **mips-elf-gcc**).

Para executar a aplicação *fw_sink* da pasta *soft*, basta abrir um terminal no diretório *Debug* do projeto *mipsUnB_soc_sink* e executar o arquivo *mipsUnB_soc_sink.x* com a opção *−−load*. Por exemplo:

```
$ cd /home/user/workspace/mipsUnB_soc_sink/Debug

$ ./mipsUnB_soc_sink.x −−load=../soft/fw_sink.x
```

A tela deverá mostrar os dados da simulação, conforme pode ser visto na Listagem II.2.

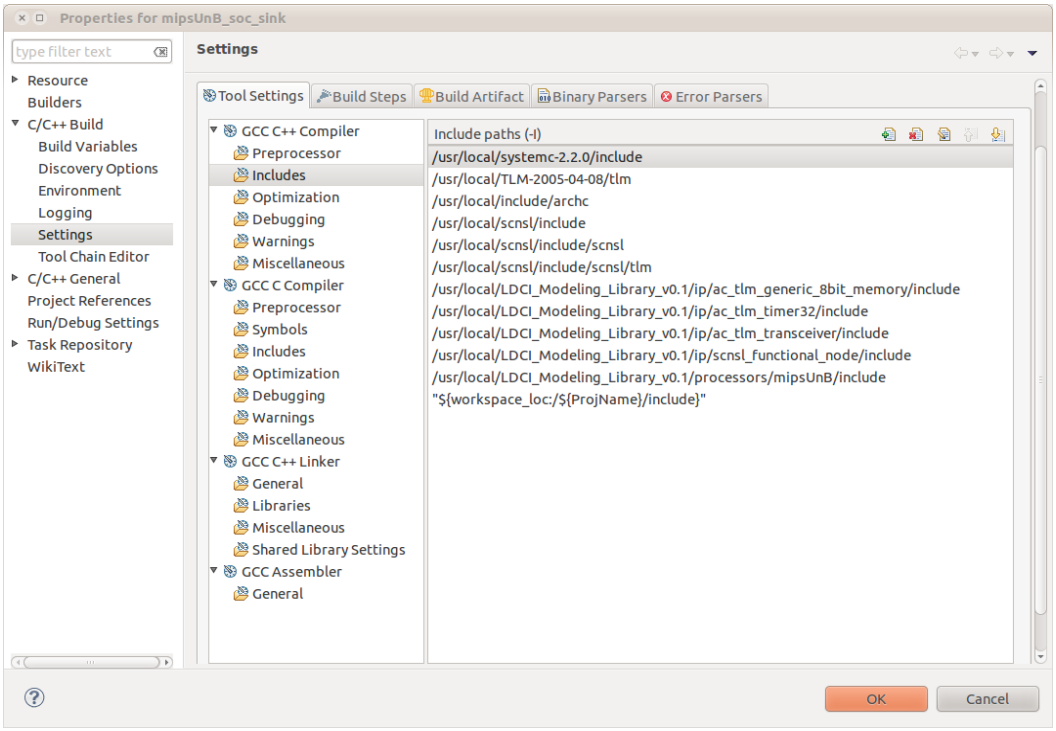


Figura II.3: *GCC C++ Compiler - Includes* do *mipsUnB_soc_sink*

---

```
                SystemC 2.2.0 --- Dec  7 2010 10:53:04
          Copyright (c) 1996-2006 by all Contributors
                      ALL RIGHTS RESERVED
ArchC: Reading ELF application file: ../soft/fw_sink.x

DBG: @@@ begin behavior @@@
ArchC: ---------------------- Starting Simulation ----------------------
x[0] = 0  y[0] = 0
x[1] = 1  y[1] = 0
x[2] = 2  y[2] = 0
x[3] = 3  y[3] = 0
x[4] = 4  y[4] = 0
x[5] = 0  y[5] = 1
x[6] = 1  y[6] = 1
x[7] = 2  y[7] = 1
x[8] = 3  y[8] = 1
x[9] = 4  y[9] = 1
Instantiating Node1.
Identifier 1.
Instantiating Node2.
Identifier 2.
Instantiating Node3.
Identifier 3.
Instantiating Node4.
Identifier 4.
Instantiating Node5.
```

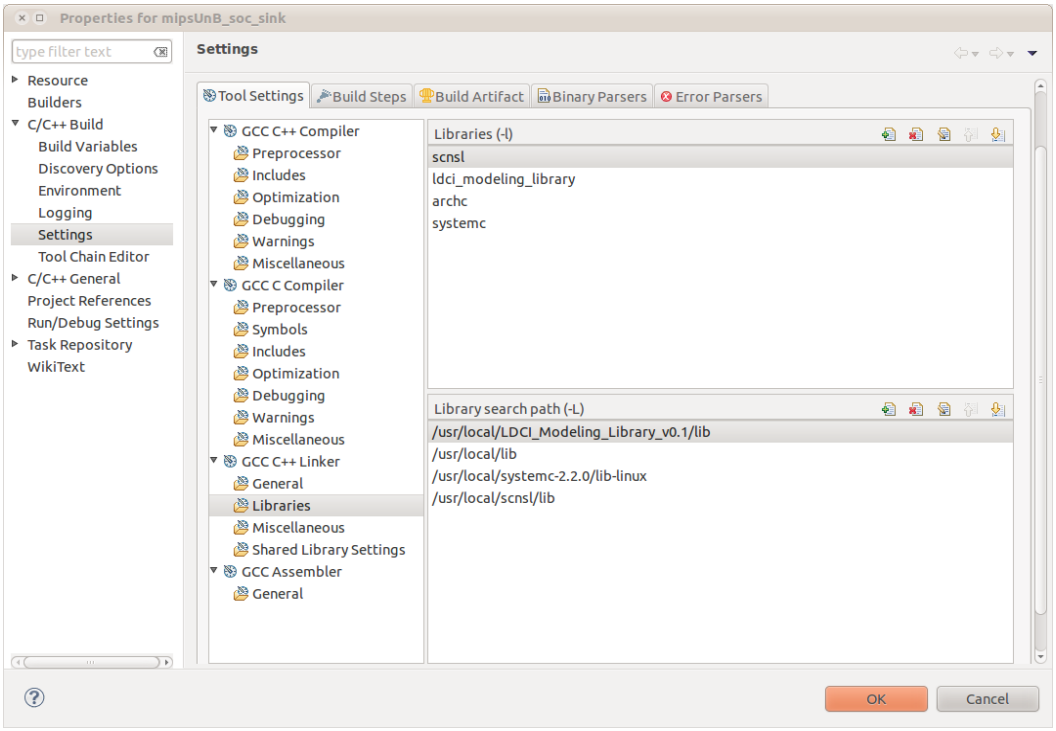


Figura II.4: *GCC C++ Linker - Libraries* do *mipsUnB_soc_sink*

```
Identifier 5.
Instantiating Node6.
Identifier 6.
Instantiating Node7.
Identifier 7.
Instantiating Node8.
Identifier 8.
Instantiating Node9.
Identifier 9.
Instantiating Node10.
Identifier 10.

DBG: ------------------------------------------------------------
DBG: Simulation Time: 0 s
DBG: Starting instruction Execution
DBG: ------ PC=0 NPC=0x4 ------ 0
DBG: j 4
DBG: Target = 0x10
DBG: ------------------------------------------------------------
DBG: Simulation Time: 0 s
DBG: Starting instruction Execution
DBG: ------ PC=0x4 NPC=0x10 ------ 1
DBG: nop
DBG: ------------------------------------------------------------
DBG: Simulation Time: 0 s
DBG: Starting instruction Execution
DBG: ------ PC=0x10 NPC=0x14 ------ 2
DBG: ori r29, r0, 49152
```

```
DBG: Result = 0xc000
DBG: ------------------------------------------------
DBG: Simulation Time: 0 s
DBG: Starting instruction Execution
DBG: ------ PC=0x14 NPC=0x18 ------ 3
DBG: addi r29, r29, 64512
DBG: Result = 0xbc00
DBG: ------------------------------------------------
...
```

Listagem II.2: Exemplo de simulação do mipsUnB_soc_sink

Para interromper a simulação, basta digitar **Ctrl+C**.

## II.3 *Plataforma SoC v0.1*

Copiar o arquivo *plataforma_soc_v0.1.tar.gz* e descompactar:

```
$ tar -xvzf plataforma_soc_v0.1.tar.gz
```

Mudar para o diretório *plataforma_soc_v0.1*:

```
$ cd plataforma_soc_v0.1
```

Observe que esta plataforma virtual utiliza o processador *tlm2_risc16* da *LDCI_Modeling_Library*, que contém os seguintes módulos:

- Microprocessador RISC16
- Controlador de interrupções
- Memória
- ADC Wrapper: bloco para conectar o ADC AMS e gerador de ondas senoidais instanciados da *TUV_AMS_Library* (SystemC-AMS)
- RF Interface
- Serial Interface
- Interface Stim (*testbench*)

Há três programas em hexadecimal, de acordo com o *assembly* do *RISC16*, disponíveis para testar esta plataforma virtual:

- instruction_file_fibonacci
- instruction_file_AD
- instruction_file_average

Esses programas são carregados na memória usando a seguinte instrução do código *memory.h*, em *LDCI_Modeling_Library/processors/tlm2_risc16*:

```
instructionFile = fopen("instruction_file_AD", "r");
```

Para testar outro programa, basta mudar o argumento *instruction_file_AD*, recompilar a biblioteca *LDCI_Modeling_Library* e recompilar e executar a plataforma virtual novamente. É importante notar que os arquivos dos programas devem estar no mesmo diretório do arquivo executável (*soc.x*).

Para compilar, primeiramente deve-se verificar as variáveis do *Makefile*:

```
SYSTEMC = /usr/local/systemc-2.2.0

TLM2 = /usr/local/TLM-2009-07-15

SYSTEMCAMS = /usr/local/systemc-ams-1.0BETA1

TUV_LIB = /usr/local/TUV_AMS_Library/output

LDCI = /usr/local/LDCI_Modeling_Library_v0.1
```

Em seguida, executar o *Makefile*:

```
$ make
```

Com isso, será gerado um arquivo executável chamado *soc.x*. Para simular, basta executar o comando:

```
$ ./soc.x
```

Caso não seja uma simulação passo-a-passo, pode-se gerar um arquivo de *log* na saída utilizando o seguinte comando:

```
$ ./soc.x &> output.log
```

Para uma nova compilação, deve-se primeiro utilizar o *clean*:

```
$ make clean
```

## II.4 *Plataforma RSoC v0.2*

A plataforma RSoC v0.2 instancia os seguintes blocos da *LDCI_Modeling_Library*:

- mipsUnB
- Memória TLM
- AES
- APS
- RoSA
- RF transceiver

- ADC Wrapper: bloco para conectar o ADC AMS e gerador de ondas senoidais instanciados da *TUV_AMS_Library* (SystemC-AMS)
- 32-bit timer
- Nó funcional SCNSL

A plataforma virtual também contém um barramento simples para conectar os módulos e uma pasta *soft* com aplicações.

Para executar a simulação usando o *Makefile*, copiar da pasta *using_makefiles* o arquivo *plataforma_rsoc_v0.2.tar.gz*, descompactar e mudar para o diretório *mipsUnB_interrupt_test*:

```
$ tar -xvzf plataforma_rsoc_v0.2.tar.gz
```

```
$ cd plataforma_rsoc_v0.2
```

Abrir o *Makefile* e editar os campos das seguintes variáveis [3]:

```
SYSTEMC = /usr/local/systemc-2.2.0
```

```
SYSTEMCAMS = /usr/local/systemc-ams-1.0BETA1
```

```
TUV_LIB = /usr/local/TUV_AMS_Library/output
```

```
TLM1 = /usr/local/TLM-2005-04-08
```

```
ARCHC = /usr/local
```

```
SCNSL = /usr/local/scnsl
```

```
LDCI = /usr/local/LDCI_Modeling_Library_v0.1
```

Abrir o terminal e executar o comando:

```
$ make -f Makefile
```

Após a compilação, o arquivo executável *rsoc.x* será gerado. Assim, basta escolher uma aplicação na pasta *soft* e executar o comando com a opção *−−load*:

```
$ ./rsoc.x --load=soft/<nome_da_aplicação>.x
```

Para salvar a saída em um arquivo *log*, usar o comando:

```
$ ./rsoc.x --load=soft/<nome_da_aplicação>.x &> output.log
```

Para interromper a simulação, basta utilizar **CTRL+C**.

OBS: Para executar as aplicações JPEG, JPEG com o RoSA e comunicação do RSoC com o nó genérico, observar as instruções dos itens a seguir.

[3] Se as ferramentas foram instaladas conforme indicado no Anexo I, não será necessário editar as variáveis, visto que os caminhos foram mantidos conforme sugerido nas instruções de instalação lá apresentados.

### II.4.1 Aplicações JPEG e JPEG com RoSA

Neste caso, deve-se desabilitar a macro de *debug* do *mipsUnB*. Basta editar o *Makefile* da *LDCI_Modeling_Library* e retirar a opção -DDEBUG_MODEL do comando de compilação. Em seguida, deve-se recompilar a biblioteca e a plataforma virtual.

Para que essas aplicações sejam corretamente executadas, é necessário também desabilitar as interrupções do ADC. Isso pode ser feito no arquivo:

`LDCI_Modeling_Library/wrappers/ac_tlm_wrapper/ADCWrapper.h`

Basta comentar o laço *while(true)* dentro do processo *ADCWrapperProcess()*. Neste caso, não é necessário recompilar a biblioteca, e sim apenas a plataforma virtual.

### II.4.2 Aplicação int_adc_rf

O procedimento é o inverso do item anterior. A macro DEBUG_MODEL deve ser utilizada e o processo do ADC deve estar ativo para que o mesmo possa gerar interrupções.

# III. CONTEÚDO DO CD

- anexo-I-instalacao
  - ex-scnsl
    - scnsl_test_linux.tar.gz
    - scnsl_tests.tar.gz
  - ex-systemc
    - sc_tutorial.tar.gz
  - ex-systemc-ams
    - integ.tar.gz
  - ex-tuv-ams-library
    - qpsk-transceiver.tar.gz
  - LDCI_Modeling_Library
    - LDCI_Modeling_Library.tar.gz
- anexo-II-utilizacao
  - usando-eclipse
    - mipsUnB_interrupt_test.tar.gz
    - mipsUnB_soc_sink.tar.gz
  - usando-makefiles
    - mipsUnB_interrupt_test.tar.gz
    - mipsUnB_soc_sink.tar.gz
    - plataforma_rsoc_v0.2.tar.gz
    - plataforma_soc_v0.1.tar.gz