Tiago Rogério Mück

Uma arquitetura para implementação de SDRs em sistemas embarcados

Florianópolis – SC

Dezembro de 2009

Tiago Rogério Mück

Uma arquitetura para implementação de SDRs em sistemas embarcados

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau Bacharel em Ciências da Computação.

Orientador:
Prof. Dr. Antônio Augusto Medeiros Fröhlich

Co-orientador:
Roberto de Matos

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO

Florianópolis – SC

Dezembro de 2009

“Hold your ground, hold your ground! Sons of Gondor, of Rohan, my brothers!
I see in your eyes the same fear that would take the heart of me.
A day may come when the courage of men fails, when we forsake our friends
and break all bonds of fellowship, but it is not this day.
An hour of woes and shattered shields, when the age of men comes crashing down!
But it is not this day! This day we fight!
By all that you hold dear on this good Earth, I bid you stand, Men of the West!”

Aragorn – King of Gondor

Resumo

Existe uma grande quantidade de protocolos de comunicação sem fio sendo usados, e isso implica uma série de dificuldades aos desenvolvedores de sistemas embarcados que têm que interagir com dispositivos que podem possuir, cada um, um protocolo de comunicação diferente. Os rádios definidos por software (SDR – *Software-defined Radio*) procuram solucionar esse problema utilizando uma abordagem baseada em software para dar flexibilidade na implementação dos protocolos. Neste trabalho, uma arquitetura para implementação de SDRs em sistemas embarcados é desenvolvida. A arquitetura utiliza componentes híbridos de HW/SW em hardware programável para tornar possível o uso de SDRs em sistemas embarcados dentro dos requisitos de custo, consumo e desempenho dos mesmos. Os resultados obtidos mostram que a arquitetura impõe um overhead bastante baixo e determinístico à cadeia de processamento do SDR.

Abstract

Nowaday there's a lot of wireless communication protocols being used, and this results in a series of difficulties for developers of embedded systems that interacts with devices that can use different communication protocols. The software-defined radios(SDR) aims to solve this problem by using a software-based approach to provide flexibility on the implementation of protocols. In this work, an architecture for SDRs implementation on embedded systems is developed. The architecture uses hybrid HW/SW components and programmable hardware in order to enable the use of software-defined radios on embedded systems within their requirements of cost, consumption and performance. The results show that the architecture imposes a very low and deterministic overhead to the SDR processing chain.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

Atualmente existe uma grande quantidade de protocolos de comunicação sem fio sendo usados, e isso torna mais difícil o desenvolvimento de sistemas embarcados que têm que interagir com dispositivos que podem possuir, cada um, um protocolo de comunicação diferente. Prover adaptabilidade dos sistemas embarcados aos protocolos de comunicação é um objetivo a ser atingido. No entanto, a arquitetura baseada em hardware dos rádios tradicionais impõe uma série de limitações para atingir essa adaptabilidade.

Cada elemento de hardware da cadeia de recepção/transmissão exerce uma função específica: os componentes são projetados para operar em uma freqüência e de acordo com um padrão específico. Quando a freqüência ou algum parâmetro do padrão muda, o rádio não consegue mais codificar e decodificar a informação corretamente. Para operar sob os novos padrões, componentes de hardware tem que ser substituídos. Essa falta de flexibilidade acarreta altos custos de desenvolvimento e um *time-to-market* (tempo entre o início do desenvolvimento de um produto e o momento que ele chega ao mercado) maior. Outra limitação dos rádios tradicionais está em oferecer múltiplos serviços em um único dispositivo. Um celular, por exemplo, que oferece GSM, Bluetooth e Wi-Fi, necessita de cadeias de hardware separadas para integrar estes diferentes padrões. No entanto, as limitações físicas dos celulares, como espaço e bateria, impõe várias dificuldades ao integrar vários serviços em um único dispositivo.

Os radios definidos por software (SDR – *Software-defined Radio*) seguem uma abordagem baseada em software para eliminar limitações dos rádios tradicionais descritas anteriormente. A idéia por trás dos SDRs é utilizar um sistema multibanda que permita transmitir e receber nas frequências de rádio (RF – *Radio Frequency*) desejadas, chamado de *RF front-end*, e fazer toda a modulação e demodulação em software ao invés de utilizar circuitos dedicados, permitindo assim uma maior flexibilidade na implementação da camada física dos protocolos de comunicação.

A flexibilidade dos SDRs torna possível um grande conjunto de novas aplicações e benefícios, que variam dependendo dos utilizadores desse tipo de sistema. Para fabricantes de

equipamentos de rádio, os SDR permitem que toda uma família de produtos de rádio sejam implementadas em uma plataforma comum, reduzindo o *time-to-market* dos novos produtos e trazendo uma grande redução nos custos de desenvolvimento, graças ao reuso de software entre os vários tipos de produtos. Outra grande vantagem é a possibilidade de fazer a reprogramação do rádio remotamente, permitido que erros ou inconformidades sejam corrigidos e novas funcionalidades sejam adicionadas depois que o rádio já entrou em serviço, reduzindo o tempo e os custos associados com a operação e manutenção do mesmo (SDR Forum,). Para provedores de serviços que utilizam a comunicação via rádio, novas funcionalidades e capacidades podem ser adicionadas a infra-estruturas já existentes com um custo baixo, permitindo que os provedores deixem suas redes flexíveis em relação as novas tecnologias que poderão surgir no futuro. O uso de uma plataforma de rádio comum para múltiplos mercados também traz uma redução significativa nos custos com logística, operação e manutenção. Para o usuário final, reduz os custos e prove uma comunicação sem fio ubíqua (SDR Forum,).

Contudo, as características da maioria dos sistemas embarcados não permite o uso de SDRs da maneira que foi inicialmente idealizado por Mitola (MITOLA, 1995). A implementação em software de grande parte dos protocolos exige uma grande capacidade de processamento, o que normalmente conflita com outras métricas de desenvolvimento de sistemas embarcados, como: tamanho, custo e consumo de energia. No entanto, a maioria dos SDRs que estão sendo usados na prática são implementados para oferecer um conjunto limitado de serviços. O principal motivo que leva a isso é o fato de que a maioria dos SDRs realmente não precisa possuir essa natureza genérica. Os rádios são implementados para oferecer apenas o conjunto de serviços desejados, utilizando apenas o hardware para dar o suporte necessário à camada de software do SDR, diminuindo os custo do mesmo. Na maioria dos casos essa abordagem é suficiente, mas problemas podem ocorrer quando surgir a necessidade de oferecer novos serviços utilizando o mesmo sistema, que extrapolem nos requisitos de hardware e de arquitetura de software disponíveis no sistema original.

Ao projetar um sistema embarcado normalmente temos requisitos como: custo de produção do sistema, tamanho, consumo de energia, dissipação de calor e desempenho. Implementar um sistema que possua um SDR com a arquitetura clássica é bastante improvável, já que os requisitos de processamento para esse tipo de SDR exigiria um conjunto de componentes de hardware que impossibilitaria o cumprimento dos requisitos citados anteriormente. Isso pode ser contornado com os avanços na tecnologia CMOS (Complementary Metal-Oxide-Semiconductor), que tem permitido o sudo cada vez maior de dispositivos lógicos programáveis (PLDs – *Programable Logic Device*) – como as FPGAs (*Field Programable Gate Array*), que podem ser utilizados para mover partes da camada de software do SDR para hardware a fim de obter um

melhor desempenho e eficiência de área e energia. Como o hardware pode ser reconfigurado, o rádio ainda pode ser considerado um SDR.

O uso de um sistema híbrido para implementar um SDR pode apresentar um grande risco de projeto, devido às dificuldades para traduzir um modelo de alto nível do SDR para uma implementação em PLD. Um projeto descuidado pode apresentar muitas complicações, caso novos protocolos tenham que ser suportados pelo SDR no futuro. Para superar essas dificuldades, esse trabalho propõe uma nova arquitetura para a implementação de SDRs em sistemas embarcados.

Seguindo a metodologia ADESD (*Application-driven Embedded System Design*) (FRöHLICH, 2001) e fazendo uso do conceito de componentes híbridos (MARCONDES; FRöHLICH, 2008), a arquitetura proposta permite a implementação de sistemas específicos dentro dos requisitos de custo de hardware, desempenho e consumo da aplicação alvo do SDR, e, ao mesmo tempo, permite uma fácil expansão desses sistemas, à medida que novos requisitos forem surgindo. Um framework permite que um SDR seja implementado mapeando diretamente partes de um modelo funcional de alto nível do sistema para componentes que podem migrar entre os domínios de hardware e software de uma maneira transparente.

1.1 Objetivos

O objetivo deste trabalho é desenvolver uma arquitetura para a implementação de SDRs em sistemas embarcados utilizando a metodologia ADESD aliada com a noção de componentes de hardware/software híbridos. A arquitetura deve prover formas de implementar SDRs a partir de um modelo de alto nível do mesmo e dentro das restrições dos sistemas embarcados, movendo os componentes críticos dos SDRs para o hardware de uma forma transparente.

1.1.1 Objetivos específicos

- Modelar e desenvolver o suporte da arquitetura no sistema operacional EPOS (*Embedded Parallel Operating System*)
- Avaliar o desempenho da arquitetura, apresentando resultados que guiem os usuários da mesma no desenvolvimento dos SDRs.

1.2 Organização do trabalho

Este trabalho está organizado da seguinte forma: o capítulo 2 apresenta os conceitos básicos de sistemas embarcados, as principais características, requisitos e o cenário atual. O capítulo 3 aborda em mais detalhes os SDRs, as arquiteturas já existentes e as suas vantagens e desvantagens num contexto de sistemas embarcados, juntamente com uma revisão bibliográfica abordando os principais trabalhos na área. O capítulo 4 apresenta a metodologia ADESD, o sistema operacional EPOS e trata do desenvolvimento de sistemas utilizando componentes híbridos. O capítulo 5 apresenta a arquitetura desenvolvida. O capítulo 6 mostra a modelagem e implementação do suporte do EPOS para a arquitetura. O capítulo 7 apresenta uma avaliação da arquitetura desenvolvida. Finalmente, o capítulo 8 conclui esse trabalho, apresentando as considerações finais.

2 *Sistemas embarcados*

Na literatura podem ser encontradas várias definições para sistemas embarcados:

- Sistemas embarcados são sistemas de processamento de informação que estão embutidos em um produto maior (MARWEDEL, 2003)
- Um sistema embarcado é qualquer aparelho que possua um computador programável mas este não é projetado para ser um computador de uso geral (WOLF, 2001)
- Sistemas embarcados são sistemas onde hardware e software normalmente são integrados e seu projeto visa o desempenho de uma função específica (LI; YAO, 2003)

Apesar de ligeiramente distintas, todas as definições que podem ser encontradas concordam em pelo menos três aspectos:

1. Ao contrário de sistemas de propósito-geral, sistemas embarcados são concebidos de forma a realizar tarefas específicas e conhecidas a priori.
2. Normalmente não constituem em um produto por si só, integrando geralmente um sistema maior.
3. Geralmente possuem restrições quanto aos recursos disponíveis (i.e.: memória, processamento) e possuem uma interface para interação com o ambiente específica de acordo com a sua aplicação, seja ela para a interação homem-máquina ou mesmo para a interação máquina-máquina (através de sensores e atuadores).

Ou seja, de uma maneira geral um sistema embarcado é um sistema computacional dedicado a uma aplicação específica, realizando um conjunto de tarefas predefinidas. Já que o sistema é dedicado, o projeto do sistema normalmente é otimizado para satisfazer apenas os requisitos específicos da aplicação. São exemplos de sistemas embarcados:

- Sistemas de aviação, como sistemas de controle inercial, controle de vôo e outros sistemas integrados nas aeronaves.
- Telefones celulares e centrais telefônicas.
- Sistemas automotivos: controladores da tração e de estabilidade, freios ABS e etc.
- Eletrodomésticos, como fornos microondas, máquinas de lavar, aparelhos de TV, DVD players.
- Equipamentos de monitoramento, estações meteorológicas, satélites, entre outros.

Devido à natureza das aplicações, esses sistemas possuem uma série de requisitos e restrições que não aparecem nos sistemas de propósito geral, como: consumo de potência, consumo de memória do software, execução eficiente do software, peso e tamanho do hardware, custo de produção, entre outras métricas. Marwedel (MARWEDEL, 2003) ainda destaca que os sistemas embarcados devem ser confiáveis. Estes sistemas devem ser a prova de falhas pois interagem com o meio, causando impactos a este. Segundo Marwedel, a fiabilidade de um sistema engloba os seguintes aspectos:

Fiabilidade É a probabilidade que um sistema não irá falhar

Recuperação É a probabilidade que uma falha no sistema será corrigida em um certo intervalo de tempo

Segurança Um sistema deve ser seguro em dois aspectos. Ele deve ser seguro para o meio no qual ele está incluído, ou seja, uma falha não acarreta em danos ao meio ou as pessoas que utilizam este sistema. E o sistema deve ser seguro do ponto de vista da informação que passa por ele, informações confidenciais devem permanecer confidenciais.

2.1 Cenário atual

A maioria dos componentes semicondutores hoje fabricados são utilizados em sistemas embarcados. Segundo Pop (Paul Pop, 2005), 99% dos microprocessadores produzidos atualmente são utilizados nesse tipo de sistema e recentemente o número de sistemas embarcados em uso superou o número de habitantes no planeta. Praticamente todos os objetos com os quais as pessoas interagem possuem algum tipo de processamento digital, de um PC à uma máquina de lavar. O baixo custo e o aumento da capacidade dos circuitos integrados proporcionados pelos

avanços na tecnologia CMOS tem permitido a agregação de cada vez mais “inteligência“ aos bens de consumo utilizados no dia-a-dia.

Não apenas em quantidade, os sistemas embarcados também crescem em termos de complexidade. O uso cada vez maior dos chamados *System On Chip* (SoC) – componentes que possuem todo o sistema integrado em um único chip – é um exemplo de aumento de complexidade proporcionado pelos avanços na tecnologia de semicondutores. Esse aumento de complexidade somado ao *time-to-market* cada vez menor para esse tipo de sistema, faz com que projetar sistemas embarcados dentro das suas restrições tenha se tonado uma tarefa cada vez mais difícil. Por isso o uso de metodologias específicas para projetar esse tipo de sistema é necessário.

2.2 Metodologias de desenvolvimento

A figura 2.1 (CARRO; WAGNER, 2003) apresenta o fluxo de informação em uma metodologia típica de projeto de sistemas embarcados.

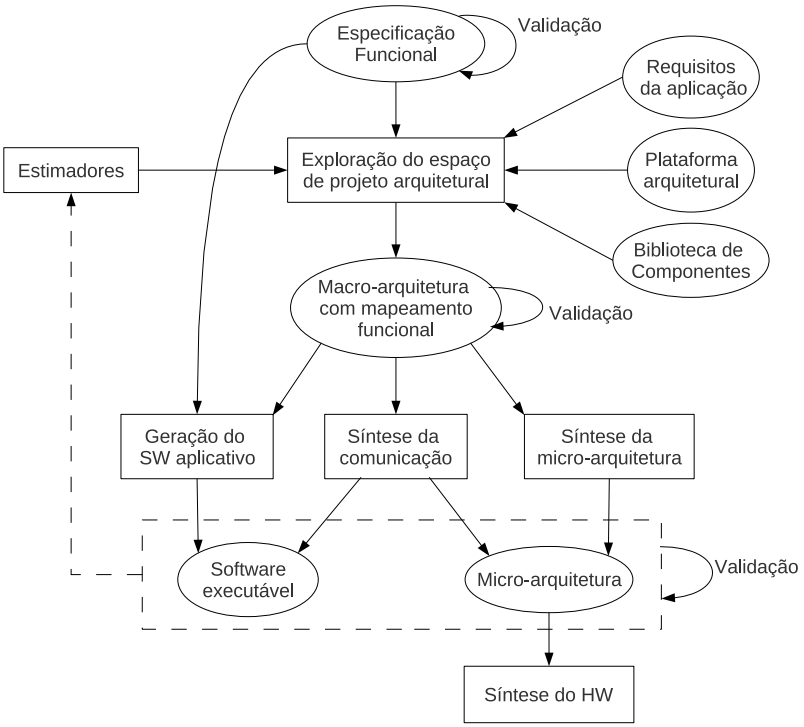

Figura 2.1: Fluxo de projeto em sistemas embarcados

O projeto normalmente começa com uma especificação funcional de alto nível do sistema. Em seguida é feita a exploração do espaço de projeto, onde o objetivo é encontrar uma arquite-

tura de hardware e software que possibilite a implementação das funcionalidades especificadas dentro dos requisitos da aplicação. O resultado da exploração do espaço de projeto é uma *macro-arquitetura* contendo os componentes do sistemas e o mapeamento das funcionalidades para esses componentes. Normalmente nessa fase ocorre o particionamento entre hardware e software das funcionalidades do sistema.

Depois da definição da macro-arquitetura é feita a geração de uma micro-arquitetura de hardware e do software baseado na especificação funcional do sistema. Idealmente, seria desejável uma síntese automática tando do hardware quanto do software. Parte dessa geração pode ser feita caso o sistema tenha sido especificado utilizando um modelo formal que suporte a geração de código para a plataforma especificada.

2.2.1 Projeto Baseado em Plataformas

O *Projeto Baseado em Plataformas* (PBD - do inglês *Platform-based Design*) (SANGIOVANNI-VINCENTELLI; MARTIN, 2001) propõe o reuso de plataformas de hardware e software, padronizadas e previamente validadas, orientadas para determinados domínios de aplicação. A idéia principal é que caso essa plataforma possa atender as restrições do projeto de um conjunto grande de aplicações, o custo da própria plataforma pode ser pulverizado dentre esse conjunto de aplicações, favorecendo assim o desenvolvimento das mesmas.

No fluxo do projeto baseado em plataformas (figura 2.2) partimos de uma instância da aplicação e é feita uma exploração do espaço de projeto inicial para definir uma plataforma para o sistema. Definida essa plataforma é feita uma exploração de projeto da plataforma para obter uma instância da plataforma que satisfaça os requisitos da aplicação.

Contudo, Vincentelli (SANGIOVANNI-VINCENTELLI et al., 2004) alerta para os desafios existentes nesta abordagem. O principal deles é especificar uma plataforma que seja reutilizável por uma gama considerável de aplicações, de forma que os benefícios do uso desta plataforma possam efetivamente justificar os custos na tarefa de especificação e desenvolvimento, sem que se use um sistema de propósito geral para isso.

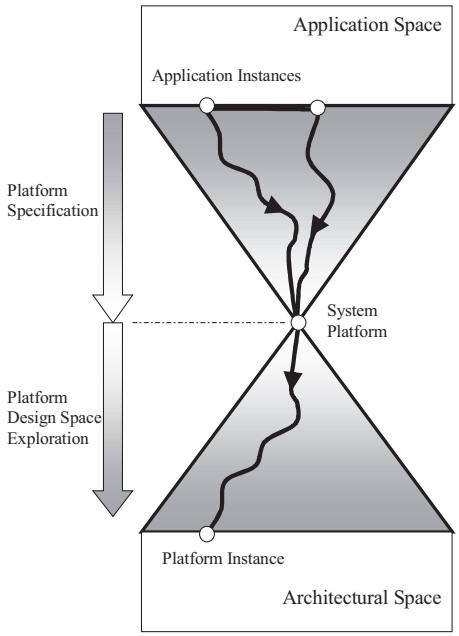

Figura 2.2: Projeto baseado em plataformas (FERRARI; SANGIOVANNI-VINCENTELLI, 1999)

3 *Radio definido por software*

3.1 Introdução

Como já foi abordado anteriormente, os sistemas embarcados precisam cada vez mais prover adaptabilidade aos vários protocolos de comunicação existentes, mas a arquitetura baseada em hardware dos rádios tradicionais impõe uma série de limitações em prover essa adaptabilidade. A necessidade de suportar novos protocolos ou mudanças nos protocolos atuais requer um reprojeto do sistema e a substituição de componentes de hardware. Portanto, novas formas de projetar rádios são necessárias.

SDRs são rádios em que os componentes da camada física dos protocolos de comunicação, tradicionalmente implementados em hardware, são implementados em software. Idealmente, um SDR possui em hardware apenas o necessário para captar o sinal a ser processado e todo o resto (moduladores, demoduladores, filtros e etc) em software. Dessa forma, um SDR pode ser, idealmente, desde um rádio AM/FM, até um receptor de TV digital, sem que nenhuma modificação de hardware seja necessária.

Para poder implementar a camada física do protocolos em software, o sinal recebido deve ser convertido para o domínio digital para que possa ser processado. Digitalização (ou amostragem) é o processo de converter um sinal analógico (uma função contínua) em uma seqüencia de números (uma função discreta) ou sinal digital. Isso é feito utilizando um conversor analógico-digital(ADC – *Analog-to-Digital Converter*). Esse sinal é então processado utilizando técnicas de processamento digital de sinais (DSP – *Digital Signal Processing*). Na transmissão, o caminho inverso ocorre: o SDR gera um sinal digital que é convertido para um sinal analógico utilizando um conversor digital-analógico(DAC – *Digital to Analog Converter*) que é enviado à antena para ser transmitido.

A figura 3.1 mostra um SDR ideal. No SDR ideal os ADCs são conectados diretamente nas antenas e o sinal completo é amostrado. Dessa maneira, podemos processar o sinal completo em software e substituir toda a cadeia de hardware apenas utilizando ADCs, processadores de uso

geral e um software de processamento de sinal. Com essa abordagem, o mesmo equipamento pode trabalhar em qualquer freqüência, padrão ou aplicação, apenas fazendo atualizações no software.

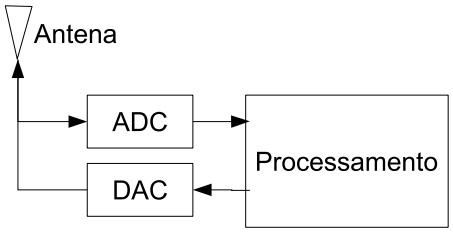

Figura 3.1: SDR ideal

No entanto, a implementação de um SDR ideal não é possível. Segundo o teorema da amostragem de *Nyquist–Shannon*, um sinal analógico que foi amostrado só pode ser reconstruído sem perdas a partir das amostras se a taxa de amostragem for pelo menos duas vezes maior que a maior frequência no sinal original, ou seja:

$$f_s \geq 2B \tag{3.1}$$

Onde f_s é taxa de amostragem e B é a maior frequência encontrada no sinal. Por exemplo, para amostrar um sinal que está na faixa dos 2.4 GHz, o ADC precisa ter uma taxa de amostragem de pelo menos 4.8 Gsps (bilhões de amostras por segundo).

Contudo, exitem limitações quanto à capacidade de amostragem dos ADCs, que não conseguem amostrar sinais com freqüências muito elevadas. Atualmente existem ADCs com taxas de amostragem na faixa de 500 Msps (milhões de amostras por segundo) (Texas Instruments,), mas isso está longe de ser suficiente para amostrar todo o espectro eletromagnético utilizado nas transmissões via rádio. Além do custo proibitivo desses ADCs, a grande capacidade computacional necessária para processar diretamente o sinal (filtrar a faixa de frequência desejada) torna esse tipo de implementação inviável para a maior parte das aplicações embarcadas.

A figura 3.2 mostra como os SDRs são implementados na prática. A idéia é filtrar a janela de frequência desejada e convertê-la para uma frequência intermediaria (IF – *Intermediate Frequency*), que possa ser amostrada pelo ADC.

Uma interface analógica é criada entre a antena e o ADC, chamada *RF front-end*. O *RF front-end* é responsável por captar o sinal de interesse e deslocar uma parte desse sinal para uma frequência intermediária. Por exemplo, para digitalizar um sinal que está na faixa 2.40–2.41 GHz o *front-end* seleciona essa janela de 10 MHz e desloca para a IF 0–10 MHz. Dessa forma o sinal pode ser amostrado utilizando ADCs de mais baixo custo.

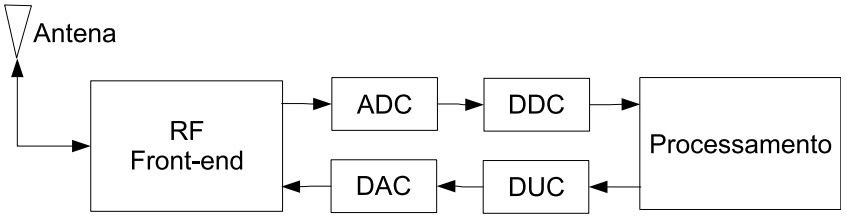

Figura 3.2: SDRs na prática

Como pode ser visto na figura 3.2, os SDRs na prática ainda possuem mais um estágio antes do processamento. Normalmente apenas uma faixa do sinal amostrado pelo ADC é necessária, por isso esses sistema incluem um *digital down-converter* (DDC) e um *digital up-converter* (DUC). O DDC é responsável por converter um sinal centralizado em uma certa frequência intermediária para um sinal complexo em banda base (um sinal cuja menor frequência é igual a zero e a maior frequência é igual a largura de banda do sinal (SCHWARTZ, 1970)). Em (YOUNGBLOOD, 2002) pode ser encontrada uma explicação mais detalhada sobre as vantagens em utilizar amostras complexas para representar sinais.

Estendendo o exemplo anterior, há um sinal de interesse em uma janela de 2 MHz centralizada em 2.405 GHz. Na frequência intermediária este sinal estará centralizado em 5 MHz. Neste caso o DDC vai fazer uma translação deste sinal para o zero e uma decimação do mesmo. O processo de decimação consiste em diminuir a taxa de amostragem do sinal para limitar a sua banda. O resultado do DDC é um sinal na faixa 0 a 2 MHz que é enviado para processamento.

O DUC faz o processo inverso do DDC. O sinal é interpolado e convertido novamente para a frequência intermediária que é transformada em um sinal analógico pelo DAC. O *RF front-end* finalmente transmite o sinal na sua frequência original.

3.2 Abordagens de implementação

Há várias abordagens para implementar a parte de processamento de sinais do SDR. De forma geral, podemos generalizar todas essas abordagens em três classes básicas (MILLHAEM, 2006): com processadores de uso geral, com hardware de processamento de sinais programável e utilizando hardware dedicado.

O modelo utilizando processadores de uso geral (GPP – *General Purpose Processor*) consiste no arquitetura básica de um SDR descrita anteriormente. Neste modelo todo o processamento do sinal é feito por um software executando no GPP. Esse modelo produz o hardware mais simples e flexível entre todos, pois a maior parte do trabalho é feita via software. O compromisso primário neste caso é entre a largura de banda suportada pelo sistema e a capacidade

de processamento disponível. Para obter uma maior largura de banda, será necessária uma capacidade maior de processamento, e, portando, um aumento nos custos do sistema.

No segundo modelo, o processamento do sinal é feito por dispositivos de hardware projetados para desempenhar funções especificas. Estes dispositivos incluem DSPs (*Digital Signal Processors*), DDCs e DUCs dedicados e IPs (*Intellectual Property* – blocos de hardware que desempenham alguma função específica) em PLDs. DSPs são mais eficientes que GPPs em cálculos numéricos, DDCs e DUCs podem fazer a decimação, interpolação e conversão de freqüência de sinais praticamente em tempo real. O projeto de hardware neste modelo tende a ser mais complicado e, tipicamente, uma FPGA é utilizada pra fazer o roteamento pelos possíveis caminhos na cadeia de processamento de sinais e para implementar alguns blocos de processamento. Em contrapartida, o software tende a ser mais simples do que no modelo anterior, pois funções matemáticas de alto nível estão disponíveis. Como o hardware usado neste modelo é mais otimizado para as suas funções, este tipo de implementação possui um melhor compromisso entre a largura de banda do sistema e o custo do hardware. A desvantagem está em um *time-to-market* maior devido à maior complexidade no projeto do hardware.

No modelo utilizando hardware dedicado, a maior parte da cadeia de processamento é *hard-coded* em um PLD, ou seja, é feita uma implementação específica do(s) procolo(s) desejado(s) em hardware. A princípio este modelo parece uma regressão aos rádios convencionais, mas, uma vez que o hardware pode ser reconfigurado, um rádio projetado dessa maneira ainda pode ser considerado um SDR. Dos três modelos, este é o que possui um maior custo de desenvolvimento, pois o ambiente para desenvolver as funções de processamento do sinal oferece mais dificuldades. Em contrapartida, com um hardware mais dedicado é possível atingir uma melhor relação entre o custo do hardware, desempenho, tamanho e consumo de energia.

3.3 Rádios Cognitivos

Um rádio cognitivo é definido como um sistema de comunicação que está ciente sobre o ambiente onde está inserido e usa uma metodologia de aprendizagem para compreender melhor o ambiente e as necessidades do usuário e se adaptar de forma a oferecer o melhor serviço possível. Para possuir essas características, um rádio cognitivo deve ser capaz de se reconfigurar dinamicamente, o que torna os SDRs a plataforma ideal para a implementação desse tipo de sistema (MITOLA; JR, 1999).

Haykin (HAYKIN, 2005) faz uma análise completa sobre rádios cognitivos, suas características, objetivos e metodologias de implementação. Uma análise profunda é feita a cerca

das três principais tarefas desempenhadas por um rádio cognitivo, que são, segundo o autor: análise do ambiente, estimação dos estados dos canais RF e modelagem preditiva, e controle da potência de transmissão juntamente com um gerenciamento dinâmico do espectro RF.

3.4 Trabalhos Correlatos

Esta seção trata de alguns trabalhos que estão sendo desenvolvidos na área, suas vantagens e desvantagens num contexto de sistemas embarcados.

3.4.1 GNU Radio

O GNU Radio (GNU FSF project, 2009) é um framework *open source* para implementação de SDRs em PCs derivado do projeto SpectrumWare (TENNENHOUSE; BOSE, 1995) do MIT. Ele oferece uma biblioteca com blocos de processamento de sinais e uma maneira de conectar estes blocos para formar um SDR. O GNU Radio permite que a camada física dos rádios seja abstraída como um grafo de fluxo acíclico, onde os nodos representam blocos de processamento e as arestas o fluxo de dados entre os nodos. Em uma cadeia de recepção, por exemplo, os dados saem do nodo inicial, o *RF front-end*, e vão fluindo através dos blocos de processamento, sofrendo transformações até chegar no nodo final na forma de bytes ou pacotes demodulados. A figura 3.3 exemplifica o grafo de SDR.

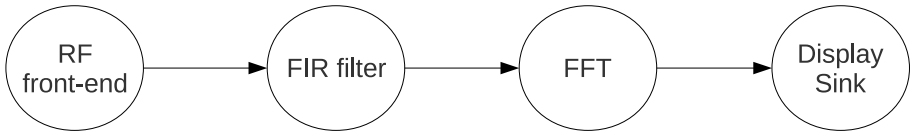

Figura 3.3: Grafo de fluxo no GNU Radio

Conceitualmente, um blocos de processamento processam um fluxo infinito de dados, fluindo das suas portas de entrada para as suas portas de saída. As portas de entrada e saída servem como fontes e sumidouros de dados no grafo. Por exemplo, há fontes que lêem de um arquivo ou de um ADC, e sumidouros para escrever em um arquivo, um DAC ou em um display gráfico. Mais de cem diferentes blocos de processamento já foram implementados no GNU Radio. O GNU Radio suporta o uso de vários dispositivos como *RF front-end*, entre eles a USRP e a USRP2 (Ettus Research, 2008).

As principais vantagens do GNU Radio estão na facilidade de desenvolvimento, na quantidade de blocos de processamento de sinal disponíveis e na grande comunidade de software livre

que suporta o framework. Os blocos de processamento de sinais são feitos utilizando a linguagem C++ enquanto que e os grafos são implementados na linguagem Python e executados em um interpretador, oferecendo, dessa forma, uma interface de alto nível ao desenvolvedor e um bom desempenho através da execução nativa dos blocos de processamento.

As desvantagens surgem a partir das próprias características básicas do GNU Radio. As aplicações feitas com o GNU Radio estarão sendo executadas em um PC com um sistema operacional de propósito geral e se comunicando com os *RF front-ends* através de barramentos compartilhados. O uso desse tipo de sistema insere um atraso não-determinístico na cadeia de processamento, o que impossibilita a implementação eficiente das camadas mais altas dos protocolos que possuem requisitos de tempo precisos (NYCHIS et al., 2009), além do atraso intrínseco do próprio GNU Radio. O GNU Radio também não é adequado para sistemas embarcados, devido à sua complexidade e requisitos de funcionamento. Existem versões do GNU Radio para Linux embarcado em algumas plataformas (BEAGLEBOARD.ORG,), mas os processadores utilizados neste tipo de sistema não têm a capacidade de processamento necessária para implementar a maioria dos blocos críticos do GNU Radio.

3.4.2 Outras arquiteturas

Nos últimos anos muitas outras arquiteturas foram propostas para a implementação de SDRs. Muitas delas consistem em SoCs que incluem um GPP para implementação das camadas mais altas dos protocolos e co-processadores SIMD (*Single Instruction, Multiple Data*) para fazer os processamento dos sinais. Alguns exemplos desse tipo de arquitetura são: Signal-processing On-Demand Architecture (SODA) (LIN et al., 2006), Cell Broadband Engine (GSCHWIND et al., 2006), Sandbridge Sandblaster (GLOSSNER; HOKENEK; MOUDGILL, 2004), Phillips EVP (BERKEL et al., 2005) e Elemental (STEVEN KELEM et al., 2007). A figura 3.4 apresenta uma visão geral de um exemplar desta família de arquiteturas.

No exemplo da figura 3.4 um processador ARM é utilizado para implementação das camadas mais altas dos protocolos enquanto que vários elementos de processamento (PE – *Processing Element*) são responsáveis pelo processamento dos sinais. Cada PE possui memória interna para instruções e dados, uma unidade de processamento escalar e uma unidade SIMD para processamento vetorial.

Apesar de muitas dessas arquiteturas atingirem os requisitos de desempenho e consumo de energia de muitas aplicações em SDR, elas impõem muitas dificuldades no desenvolvimento das aplicações. No exemplo particular da figura 3.4, o desenvolvedor que deve definir que partes do software serão executadas em cada um dos PEs e escreve-las diretamente na linguagem *as-*

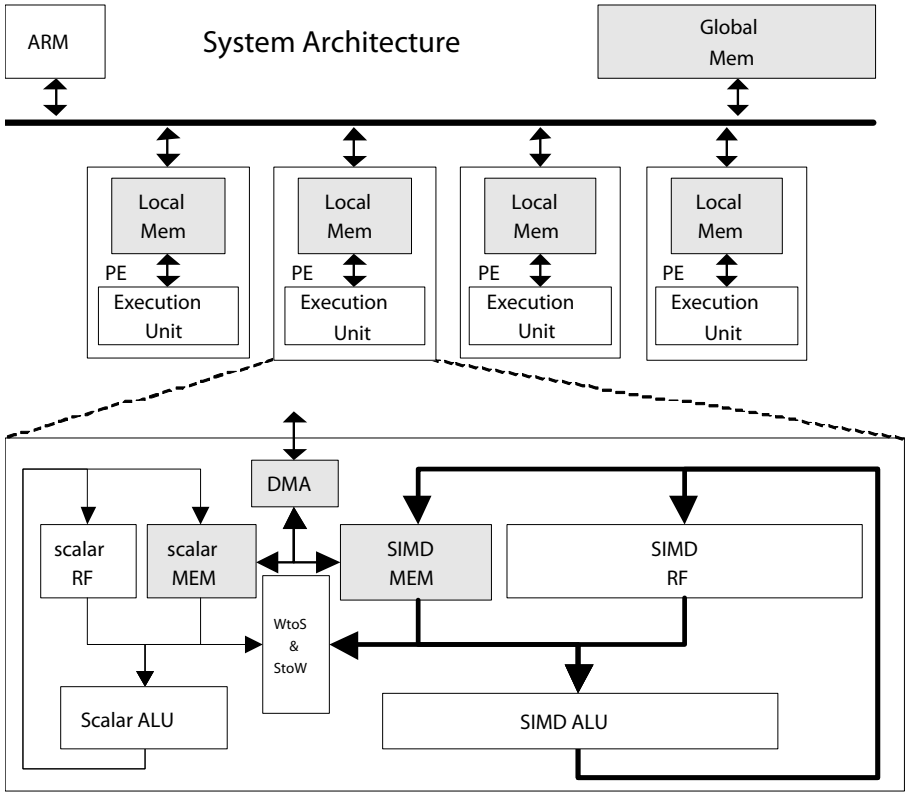

Figura 3.4: Visão geral da arquitetura SODA (LIN et al., 2006)

sembly utilizada pelos PEs. O sincronismo entre os PEs e o ARM também é bastante complexo e deve ser feito manualmente.

Existem várias outras arquiteturas. O Vanu Software Radio System (CHAPIN; BOSE, 2002) é um projeto comercial muito semelhante ao GNU Radio e também derivado do SpectrumWare. A Wireless Open-Access Research Platform (WARP) (Rice University,) da Rice University oferece uma plataforma baseada em FPGAs para a implementação de SDRs. O Chamaleonic Radio (HASAN et al., 2006) utiliza a Software Communications Architecture (SCA) (DAVIS, 1999) – um padrão baseado na arquitetura CORBA para interconexão de componentes de comunicação RF desenvolvido durante o projeto Joint Tactical Radio Systems (JTRS) (DAVIS, 1999) – para implementação de aplicações para segurança. A OpenCPI (Mercury Federal Systems,) é um framework para o desenvolvimento de aplicações baseadas em componentes heterogêneos e suporta a SCA.

4 *Projeto de sistemas embarcados dirigido pela aplicação*

A metodologia de projeto de sistemas embarcados dirigido pela aplicação (ADESD – *Application-driven Embedded System Design*) (FRöHLICH, 2001) oferece uma maneira de desenvolver sistemas dedicados a uma certa aplicação utilizando componentes pré-existentes obtidos através de uma cuidadosa engenharia de domínio. Essa engenharia de domínio identifica entidades significtivas que compõe uma certa aplicação e as organiza em famílias de componentes reusáveis que abstraem essas entidades. Mas, algumas destas abstrações ainda podem possuir dependências do cenário em que se encontram. Essa dependência pode ser reduzida utilizando o conceito de programação orientada a aspectos (KICZALES et al., 1997). Esse conceito provê meios de identificar variações em um cenário e modela-las como aspectos, esses aspectos podem ser aplicados em um componente mais abstrato para obter uma implementação do componente adequada para o cenário.

4.1 O sistema operacional EPOS

O EPOS (FRöHLICH, 2001) é um sistema operacional baseado na metodologia ADESD e tem o objetivo de automatizar o desenvolvimento de sistemas dedicados, de modo que os desenvolvedores possam dedicar-se no que realmente importa: a aplicação. O EPOS oferece um conjunto de ferramentas para selecionar, configurar e conectar componentes em um framework especifico para a aplicação, possibilitando a geração automática de uma instância do sistema operacional para a aplicação.

A figura 4.2 apresenta uma visão geral do EPOS e a figura 4.1 mostra a relação entre as principais construções resultantes da decomposição de domínio através da ADESD: famílias de abstrações independentes de cenários, adaptadores de cenários e interfaces infladas.

Para oferecer uma aplicação embarcada para cada sistema computadorizado, o EPOS utiliza os componentes de software, que implementam um conjunto de abstrações de sistema indepen-

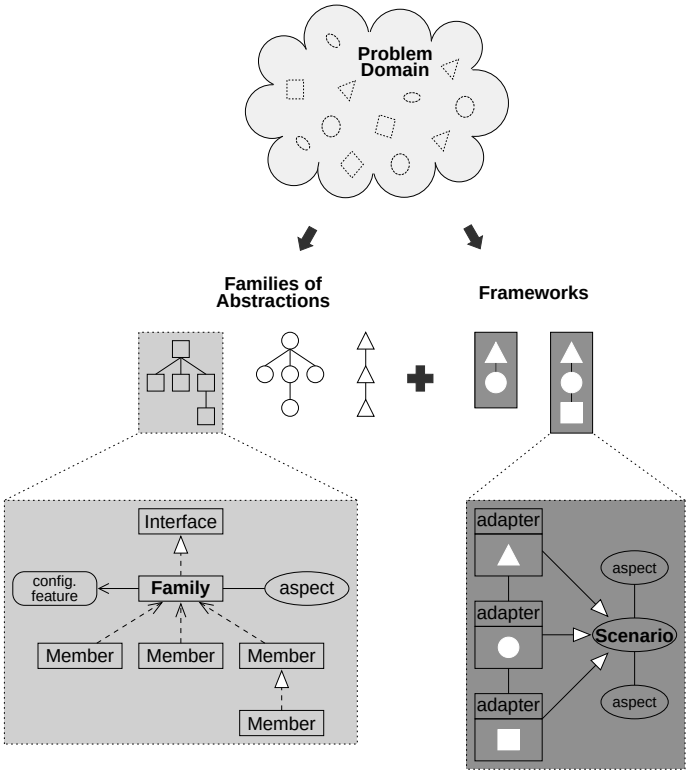

Figura 4.1: Visão geral da decomposição de domínio através da ADESD (MARCONDES; FRöHLICH, 2008)

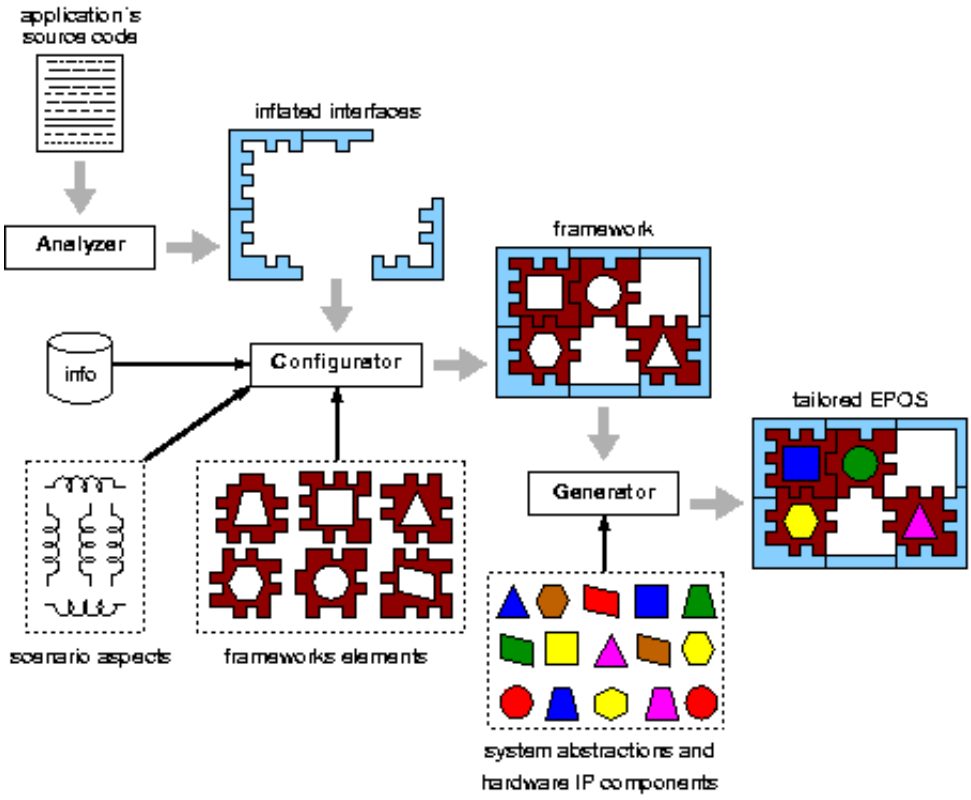

Figura 4.2: Visão geral do EPOS (LISHA, 2008)

dentes de cenário que podem ser adaptadas posteriormente para um cenário de execução, com a ajuda de adaptadores de cenário. Essas abstrações ficam em um repositório e são exportadas para a aplicação utilizando interfaces infladas. Essa estratégia permite que os programadores expressem os requisitos da aplicação independentemente do sistema alvo.

Um tipo particular de componente, chamado de mediador de hardware, é responsáveis por manter as abstrações de alto nível independentes de arquitetura. Pode-se imaginar estes componentes como parte de uma camada de abstração para um hardware configurável. Quando combinados com IPs, os mediadores de hardware fornecem uma interface software-hardware customizável que poderá dar origem a uma plataforma de hardware dedicada, ou seja, uma plataforma de hardware que inclui somente as funcionalidades necessárias para dar suporte a aplicação.

Uma aplicação projetada e implementada de acordo com essa estratégia pode ser submetida a uma ferramenta que irá fazer uma análise sintática e de fluxo de execução para extrair um guia de construção do sistema operacional a ser gerado. Esse guia é então refinado fazendo uma análise de dependência utilizando as informações do cenário de execução repassadas pelo usuário. O resultado desse processo é um conjunto seletivo de componentes e parâmetros que darão suporte a compilação do sistema operacional e a síntese dos componentes de hardware em um dispositivo de lógica programável.

4.2 Componentes Híbridos

Na ADESD os mediadores de hardware são implementados utilizando técnicas de Programação Gerativa (CZARNECKI; EISENECKER, 2000), adaptando a interface do hardware para a interface requisitada pelo sistema, ao invés de criar camada de abstração do hardware. Em (MARCONDES; FRöHLICH, 2008) é elaborado o conceito de componentes híbridos de hardware e software baseado no conceito de mediadores de hardware. Essa idéia de componentes híbridos surge do fato de que diferentes mediadores podem existir para o mesmo componente em hardware, cada qual projetado com uma série de objetivos específicos, como atingir um melhor desempenho em decorrência de um maior consumo de energia, por exemplo. Caso o sistema esteja sendo desenvolvido através de uma plataforma que possua dispositivos de lógica programáveis, a noção de componentes híbridos se torna ainda mais clara, uma vez que a combinação de mediadores poderia existir ainda com uma diferente combinação de hardware e software.

A figura 4.3 apresenta a forma geral de um componente híbrido. Cada componente agrega um mediador que faz a interface entre as diversas implementações possíveis tanto em hardware

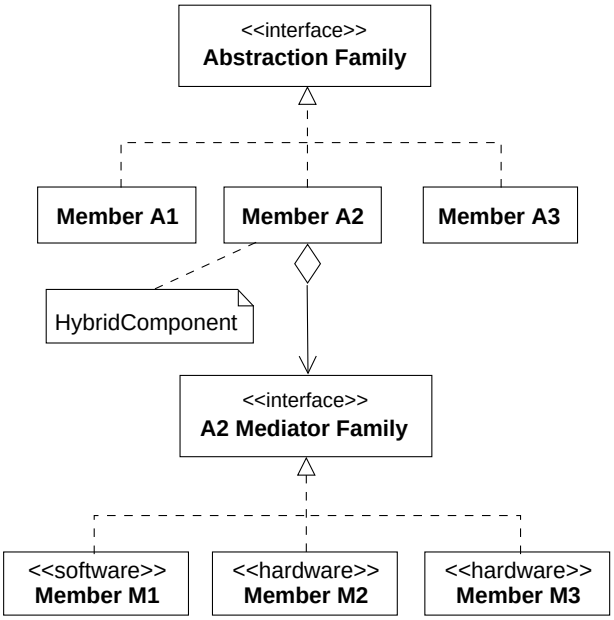

Figura 4.3: Componentes híbridos (MARCONDES; FRöHLICH, 2008)

quanto em software. Dependendo dos requisitos de custo, desempenho, consumo e etc, qualquer uma destas implementações pode ser selecionada sem alterar as camadas superiores do sistema que usam o componente. Neste trabalho os componentes híbridos ainda são separados em três categorias, segundo o seu comportamento do ponto de vista dos componentes clientes (componentes que usam o componente híbrido):

Componentes síncronos São os componentes cujos serviços são explicitamente invocados. Os clientes ficam bloqueados até a conclusão do serviço.

Componentes assíncronos Seus serviços são explicitamente invocados, mas os clientes não ficam bloqueados esperando pela conclusão. Um mecanismo de *call-back* deve ser utilizado para notificar o cliente da conclusão do serviço.

Componentes autônomos São componentes que realizam serviços sem que os mesmos tenham que ser invocados. Os serviços podem gerar eventos para componentes clientes ou ser executados de forma totalmente pervasiva.

5 *A arquitetura para o desenvolvimento de SDRs*

Como foi abordado no capítulo 2, o desenvolvimento de sistemas embarcados requer metodologias e ferramentas específicas. Quando é cogitado o uso de SDRs em sistemas embarcados, surgem uma série de complicações adicionais devido às características de ambos mostradas nos capítulos 2 e 3. Inicialmente o desenvolvimento do SDR pode começar com o fluxo de projeto apresentado no capítulo 2, criando-se um modelo de alto nível do SDR. O problema surge na hora de mapear o modelo de alto nível de algo que requer muito processamento e largura de banda para uma implementação que satisfaça os requisitos mais comuns de uma aplicação em um sistema embarcado.

Normalmente os SDRs são implementados utilizando a abordagem com hardware dedicado apresentada no capítulo 3 para que seja possível satisfazer os requisitos do sistema, entretanto essa abordagem possui um risco de projeto muito grande e pode restringir a flexibilidade do SDR. O ideal seria o uso de uma arquitetura semelhante ao GNU Radio, oferecendo uma abstração de alto nível implementável do SDR, mas, como explicado no capítulo 3, o GNU Radio não é viável em sistemas embarcados. Arquiteturas seguindo a abordagem de hardware dedicado programável, como a SODA, podem satisfazer os requisitos de um sistema embarcado, mas oferecem inúmeras dificuldades ao converter um modelo de alto nível para uma implementação.

Dadas todas essas dificuldades, este trabalho propõe uma nova arquitetura para o desenvolvimento de SDRs em sistemas embarcados. A nova arquitetura proposta usa a idéia de componentes híbridos de hardware e software através dos mediadores de hardware da ADESD para permitir a implementação de um SDR utilizando a abordagem de hardware dedicado programável, enquanto oferece um framework que possibilita a implementação a partir de um modelo de alto nível do SDR.

5.1 Definição da arquitetura

A arquitetura proposta neste trabalho apresenta uma estrutura e um comportamento semelhantes aos do GNU Radio. Ela permite que o SDR seja abstraído como um grafo de fluxo onde os nodos representam os blocos de processamento e as arestas representam canais FIFO (*First-In First-Out*) entre os blocos. De forma semelhante ao GNU Radio, um framework permite que os blocos sejam conectados para formar o grafo de fluxo e faz o gerenciamento das filas de dados entre os blocos e de quando ocorrerá o processamento em cada bloco.

De forma mais precisa, a arquitetura segue o modelo de fluxo de dados síncrono (SDF – *Synchronous data flow*) (LEE; MESSERSCHMITT, 1987b). Neste modelo, as entradas para o SDF são um fluxo infinito de dados e os nodos podem iniciar o seu processamento assim que um número suficiente de dados estiver disponível nas suas entradas. A figura 5.1 mostra a representação gráfica de um SDF. Os nodos A e B representam o processamento (operações * e +). O rótulo nas aresta indica o tamanho da FIFO entre os nodos.

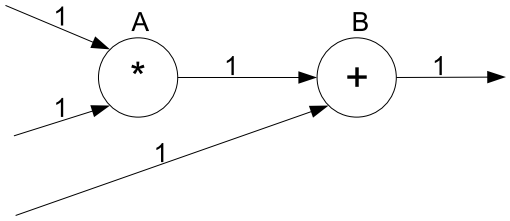

Figura 5.1: Representação gráfica de um SDF (MARWEDEL, 2003)

Neste modelo, cada execução do processamento nos nodos é chamada de *firing*. Para cada *firing* um número constante de dados (também chamados de *tokens*) é consumida e produzida. Essa propriedade permite determinar estaticamente uma ordem de execução para os nodos e os requisitos de memória (tamanho das filas), evitando o tempo extra necessário para escalonar o disparo dos blocos de processamento em tempo de execução (LEE; MESSERSCHMITT, 1987a).

A figura 5.2 apresenta uma visão geral da arquitetura proposta, na forma de uma abstração de fluxo de dados que flui de uma fonte para um sumidouro. O diferencial está em como é feito o processamento em cada bloco. A parte responsável por fazer o processamento de cada bloco é um componente híbrido (*Hybrid Component*).

A definição de quando e qual bloco será executado é feita por um mecanismo chamado de *Controlador de Fluxo* (FC – *Flow Control*). O FC define em tempo de execução qual bloco será executado ao invés de fazer escalonamento estático, pois cada um dos blocos pode estar implementado em hardwares diferentes e ter seu processamento feito em paralelo com todos

os outros blocos, seguindo o fluxo de dados. Dada essa estrutura heterogênea, um mecanismo dinâmico controlando a execução dos blocos oferece maior eficiência e flexibilidade.

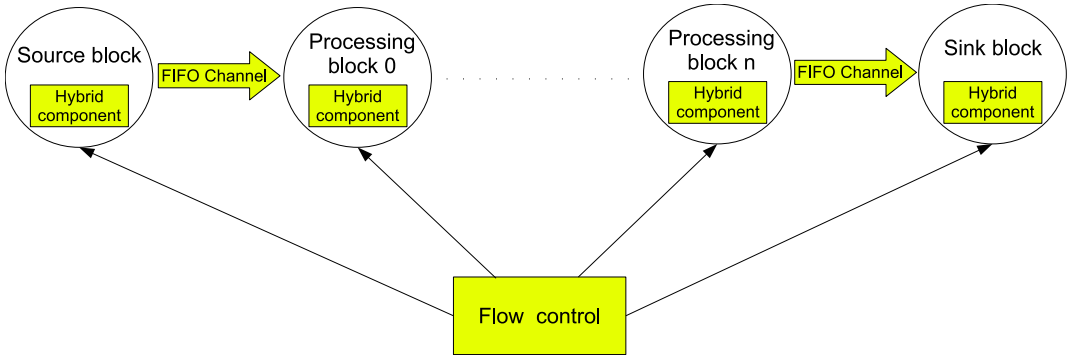

Figura 5.2: Visão geral da arquitetura proposta

Tanto o FC, quanto os canais FIFOs que ligam os blocos são componentes híbridos. Os canais FIFO podem ser em software, em hardware, ou com partes em software e em hardware, dependendo da implementação (em software ou hardware) dos componentes híbridos responsáveis pelas funcionalidades dos blocos ligados pelo canal. O FC possui mecanismo tanto em hardware quanto em software para fazer a alocação dos diferentes tipos de canais que podem existir.

Nas seções que seguem será apresentado em mais detalhes os blocos da arquitetura e o mecanismo de controle de fluxo.

5.1.1 Blocos da arquitetura

Assim como o GNU Radio, a arquitetura possui três tipos diferentes de blocos (figura 5.3):

- **Blocos-fonte:** Os blocos-fonte(SrB – *Source Blocks*) fornecem os sinais ao grafo de fluxo. São os blocos responsáveis por abstrair os *RF front-ends* e os ADCs, ou outros dispositivos que possam inserir dados no sistema (como arquivos, interfaces de rede, USB, etc).

- **Blocos de processamento:** Os blocos de processamento(PB – *Processing Blocks*) são responsáveis por fazer o processamento dos sinais.

- **Blocos-sumidouro:** Os blocos-sumidouro (SiB – *Sink Blocks*) consomem os dados dos sistema. Podem ser abstrações para um *RF front-end* e um DAC para transmissão dos dados, abstrações para algum dispositivo de saída (display, saída de áudio, por exemplo) ou algum mecanismo de empacotamento de informações e *callback* para as camadas superiores do protocolo de comunicação.

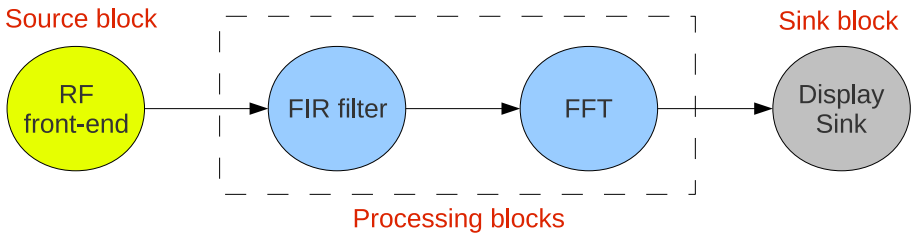

Figura 5.3: Tipos de blocos

A figura 5.4 mostra a estrutura de um PB. Cada PB possui uma interface e um bloco interno que é o componente híbrido responsável pelo processamento. A interface de bloco é parecida com o que já existe no GNU Radio. Cada bloco define o número de entradas (In_0–In_n) e saídas (Out_0–Out_n) que ele possui e o número de *tokens* consumidos e produzidos a cada *firing*, bem como o tamanho de cada *token*. Essas informações são utilizadas pelo FC para definir as relações de entrada/saída entre os blocos e o tamanho das filas FIFOs. Um componente híbrido dentro do bloco define uma interface para as funções de processamento do bloco. A estrutura para os SrB e SiB é semelhante ao do PB, exceto pelo fato de que não são definidas entradas e saídas, respectivamente.

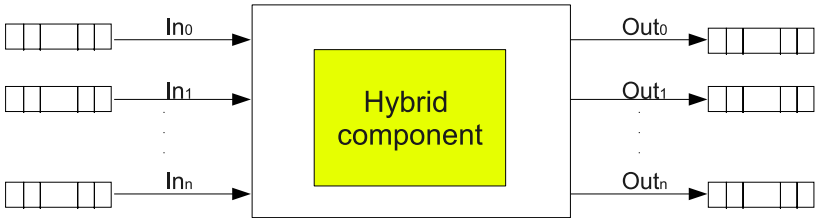

Figura 5.4: Bloco de processamento

5.1.2 Controlador de fluxo

O FC é o mecanismo responsável por:

- Criar os canais que conectam os blocos.
- Controlar a execução dos blocos.

Todas as estruturas de controle do FC são transparentes ao usuário, que deve preocupar-se apenas com as funções executadas em cada bloco e com a sua interface, de forma a definir corretamente a conexão entre os blocos.

Ao conectar uma das saídas de um bloco em uma das entradas de outro, o FC checa se ambas são compatíveis (os *tokens* possuem o mesmo tamanho) e se for possível estabelecer

uma conexão, o FC aloca um canal FIFO para a conexão. O tamanho da FIFO do canal é definida segundo a equação abaixo:

$$FIFO_{size} = (Blk_0^{outputrate} + Blk_1^{inputrate}).\alpha \tag{5.1}$$

Onde uma saída de Blk_0 está sendo conectada em uma entrada de Blk_1 e α é um fator de segurança, que deve ser configurado segundo as características de memória e desempenho da plataforma do sistema, para garantir que a FIFO vai ter espaço para todos *tokens* que serão gerados pelos *firings* necessários para que o bloco de destino tenha dados para executar.

O tipo de canal gerado depende da implementação física dos componentes híbridos que executam as funções dos blocos que estão sendo conectados. Se todos os componentes híbridos estão em hardware, então um canal em software vai adicionar um atraso extra que pode ser um gargalo para o SDR; por isso uma estrutura em hardware é necessária para permitir que blocos troquem dados diretamente sem passar por software. Essa estrutura é apresentada na figura 5.5.

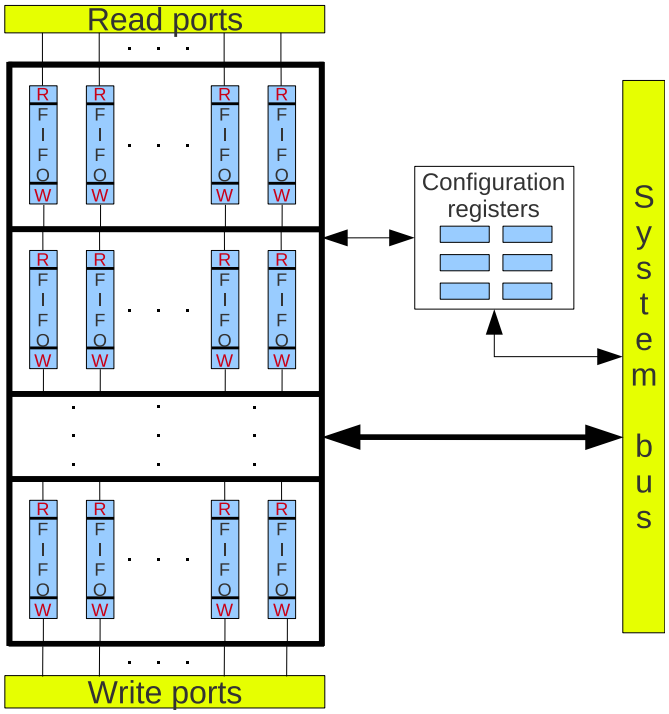

Figura 5.5: Estrutura do FC para alocação de canais em HW

Esta estrutura define uma espécie de NoC (*Network-on-a-Chip*) (KUMAR et al., 2002) onde várias FIFOS têm suas entradas e saídas interconectadas. As saídas dos blocos são ligadas às portas de escrita (*write ports*) e as entradas dos blocos são ligadas às portas de leitura (*read ports*). A definição da conexão entre uma porta de escrita e a entrada de uma FIFO, e uma porta de entrada e a saída de uma FIFO é feita através de registradores configurados pelo FC. Uma

vez definidas as conexões os dados podem fluir entre blocos em hardware sem a intervenção do software. Duas ou mais FIFOs podem ser conectadas em serial, formando uma única FIFO maior, caso as taxas de entrada/saída dos blocos tenham essa exigência. Finalmente as portas de entrada e escritas podem ser configuradas para se conectarem ao barramento do sistema para permitir a troca de dados com os canais em software.

O controle da execução dos blocos é feito através da criação de uma thread para cada bloco, onde são executadas as funcionalidades do mesmo. As threads são sincronizadas utilizando semáforos associados aos canais FIFO nas conexões dos blocos. As seções abaixo apresentam em mais detalhes como é feito esse controle da execução para cada tipo de bloco.

Controle dos blocos-fonte

A figura 5.6 apresenta a estrutura comportamental dos blocos-fonte. $Out_0 ... Out_n$ são os canais associadas a cada uma das saídas do bloco e $Sout_0 ... Sout_n$ são os semáforos associados a cada um dos canais. T_s é a thread criada onde o bloco será executado e S_s é um semáforo usado para sincronizar essa thread. *Front-end mediator* é a abstração para o *RF front-end*.

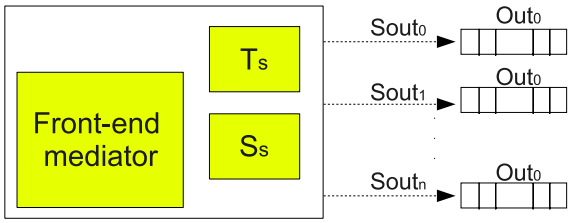

Figura 5.6: Estrutura comportamental dos blocos-fonte

O comportamento do bloco está apresentado no pseudo-código 5.1. A thread T_s fica em laço infinito, bloqueada no semáforo S_s até que uma notificação vinda do *RF front-end* indique que um novo dado está pronto. A thread T_s é desbloqueada e lê o dado através do *Front-end mediator*. Em seguida, o dado é escrito em cada um dos canais associados às saídas do bloco e o método *v()* dos semáforos associados a ele é chamado para indicar que um novo dado foi adicionado nas respectivas filas. Depois T_s volta a ficar esperando no semáforo S_s até que uma indicação de um novo dado chegue.

Pooling poderia ser usado no lugar de uma notificação e do semáforo S_s, mas a taxa de inclusão de dados no SDF deve ser mantida de acordo com o que foi configurado no *RF front-end*, e usando notificações através de interrupções é possível garantir isso (considerando que interrupções não estão sendo perdidas). Com esse esquema, a inserção de dados nos canais sempre ocorre assim que um amostra dado é gerado. Se o sistema não for capaz processar todas

Algoritmo 5.1 Comportamento dos blocos-fonte

```
Ts loop :                                   Front-end notification :
     Ss.p()                                      Ss.v()
     Read sample
     Write sample on Out0
     ...
     Write sample on Outn
     Sout0.v()
     ....
     Soutn.v()
```

as amostras em tempo, ocorrerá um transbordamento de dados nas FIFOs indicando que o SDR que deseja-se implementar está além das capacidades do sistema.

Controle dos blocos de processamento

Como pode ser visto na figura 5.7, uma parte da estrutura é semelhante a dos blocos-fonte. As diferenças estão nas entradas e nos canais associadas a elas que o bloco possui ($In_0 ... In_n$), e nos semáforos $Sin_0 ... Sin_n$ associados aos canais. Um componente híbrido é responsável por fazer o processamentos dos dados no bloco. A thread T_{pb} é a thread onde o bloco é executado.

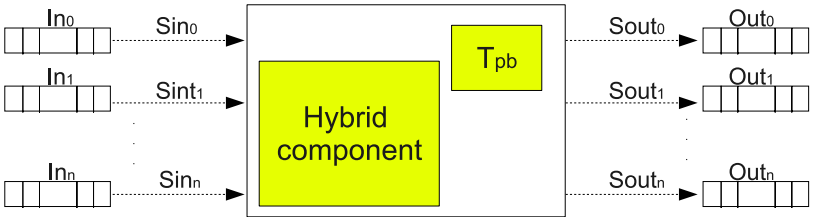

Figura 5.7: Estrutura comportamental dos blocos de processamento

O pseudo-código 5.2 apresenta o comportamento desses blocos. A thread T_{pb} fica em um laço infinito, bloqueada nos semáforos $Sin_0 ... Sin_n$. Quando dados são adicionado por algum outro bloco nos canais $In_0 ... In_n$ os seus respectivos semáforos são liberados. Em seguida o FC verifica se os dados nas entradas são suficientes para que o processamento seja feito. Se não forem suficientes, a thread T_{pb} volta a ficar bloqueada nos semáforos à espera de mais dados. Quando a condição anterior for satisfeita, os dados são retirados dos canais e enviados ao componente híbrido para processamento. Para cada saída gerada, ela é escrita nos canais de saída $Out_0 ... Out_n$ e o método *v()* dos semáforos $Sin_0 ... Sin_n$ é chamado, indicando que novos elementos foram adicionados nas suas respectivas filas.

Algoritmo 5.2 Comportamento dos blocos de processamento

```
Tpb loop :
     Sin0.p()
     ...
     Sinn.p()
     if there is enough inputs:
          Consume inputs
          Do processing
          for each output generated:
               Write outputs to Out0
               ...
               Write outputs to Outn
               Sout0.v()
               ....
               Soutn.v()
```

Controle dos blocos-sumidouro

As figuras 5.8 e o pseudo-código 5.3 mostram o comportamento dos blocos-sumidouro, que é semelhante ao dos blocos de processamento. A diferença é que esses blocos possuem apenas entradas que, ao invés de serem processadas e enviadas para as saídas, são apenas encaminhadas para um mediador de algum dispositivo de saída que consome os dados do SDF, como já foi explicado anteriormente.

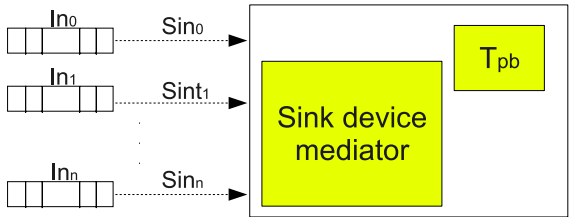

Figura 5.8: Estrutura comportamental dos blocos-sumidouro

Algoritmo 5.3 Comportamento dos blocos-sumidouro

```
Tpb loop :
     Sin0.p()
     ...
     Sinn.p()
     if there is enough inputs:
          Consume inputs
          Sink the data
```

6 *Modelagem e Implementação*

Este capítulo descreve a modelagem e a implementação do framework da arquitetura no EPOS, a qual se divide em basicamente em duas partes: a definição de uma família de abstrações para os blocos de processamento e a modelagem e implementação do controlador de fluxo.

6.1 Modelagem das abstrações dos blocos

O diagrama de classes na figura 6.1 apresenta as abstrações definidas no EPOS. Uma interface chamada *SDR_Block* foi criada contendo todos métodos de um bloco. Esses métodos serão implementados de acordo com o tipo do bloco nas classes abstratas *SDR_Source_Block*, *SDR_Processing_Block* e *SDR_Sink_Block*, que representam os blocos-fonte, blocos de processamento e blocos-sumidouro, respectivamente. Os blocos definidos pelo usuário herdarão três classes segundo o seu tipo, e o usuário proverá implementações para os métodos que fornecem informações sobre o número de entradas e saídas, as taxas de entradas e saídas e os métodos que implementam a funcionalidade do bloco. Como os blocos são componentes híbridos, a suas funções podem estar implementadas diretamente dentro do bloco ou ele pode agir como um mediador para o bloco em hardware.

Para prover uma melhor abstração dos dados ao usuário, os tipos dos dados que passam pelos blocos é definido por tipos parametrizados (GAMMA et al., 1994), permitindo que o usuário possa manipular os dados sempre através do seu tipo real ao invés de trabalhar com tipos de dados brutos, como acontece no GNU Radio, por exemplo.

6.2 Modelagem do controlador de fluxo

A figura 6.2 mostra a relação entre as classes que compõe o controlador de fluxo. A classe *SDR_Flow_Controler* abstrai o controlador de fluxo e possui três operações principais: *connect*, *start* e *stop*. O *connect* é responsável por realizar a conexão de uma porta de saída de um bloco

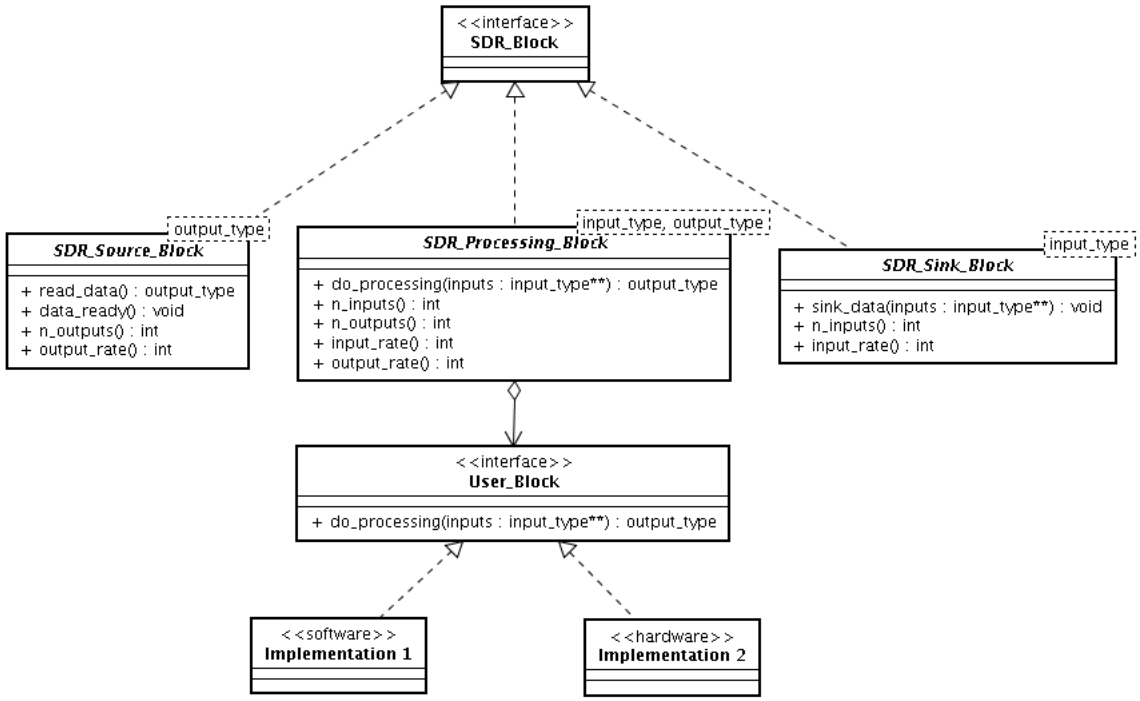

Figura 6.1: Diagrama de classes dos blocos SDR

à uma porta de entrada de outro. O *start* e o *stop* iniciam e param o grafo de fluxo do SDR, respectivamente.

A classe *SDR_Flow_Controler* possui uma lista de objetos da classe *Graph_Node*, que representam os nodos do grafo de fluxo. Os nodos possuem referências para os blocos que eles representam (*SDR_Block*), para os canais conectados ao bloco (*SDR_Channel*) e para thead que executa a funcionalidade do bloco. A classe *SDR_Channel* encapsula todas as funcionalidades de adição de dados no canais FIFO, e de sincronismo entre os blocos.

O comportamento do controlador de fluxo está definido no diagrama de seqüência da figura 6.3. Ao criar uma conexão entre dois blocos, o FC primeiro verifica se o tipo de dado da saída é compatível com o tipo de dado da entrada. Essa compatibilidade é baseada apenas no tamanho dos tipos, pois não é necessário que os tipos sejam exatamente iguais ao realizar uma conexão. As possíveis implicações semânticas de conectar blocos de tipos diferentes são de responsabilidade do usuário. Sendo compatíveis os tipos, o FC verifica se os blocos já possuem algum tipo de conexão, se não tiverem são criados os nodos para estes blocos e as threads que controlam os blocos. Essas thread implementam o comportamento definido no capítulo 5 dependendo se o bloco for um bloco-fonte, de processamento ou sumidouro. Estando os blocos presentes na estrutura do FC, os canais são então alocados para estabelecer a conexão. O tipo do canal alocado vai depender se os blocos estão em hardware ou software.

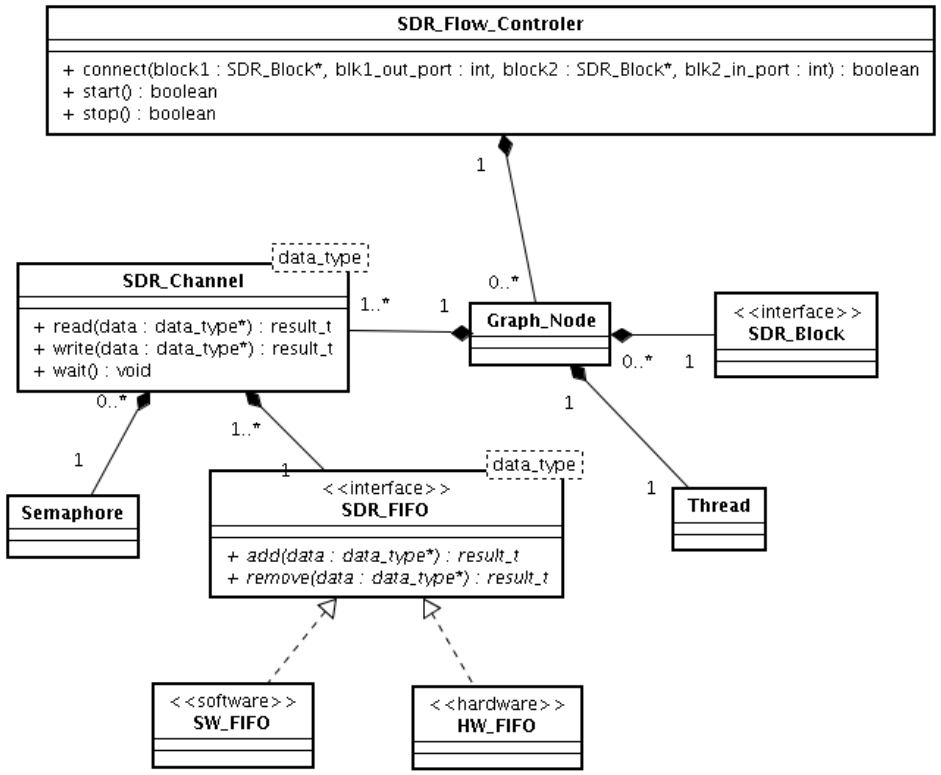

Figura 6.2: Diagrama de classes dos controlador de fluxo

Na hora de iniciar o grafo de fluxo, é feito primeiro uma verificação na estrutura do grafo para determiar se todas as entradas e saídas de todos os blocos estão conectadas. Caso afirmativo as threads são iniciadas e os dados começam a fluir pelo grafo. O método *stop* para o grafo bloqueando todas as threads no momento em que é feita a checagem da presença de novos dados para serem consumidos. Ou seja, caso um bloco esteja processando algum dado, ele vai concluir o processamento, adicionar o dado nos canais de saída (se for o caso) e só então vai parar. Dessa forma os blocos sempre são mantidos em um estado consistente, principalmente se estes estiverem em hardware.

6.3 Implementação das interface dos blocos

Para evitar a perda de desempenho em tempo de execução causada pelo uso de polimorfismo, o relacionamento entre as classes apresentado na figura 6.1 foi feito com herança paramétrica (CZARNECKI; EISENECKER, 2000) utilizando os templates da linguagem C++. Esse tipo de herança pode ser usado para atingir polimorfismo estático, ou seja, um polimorfismo resolvido em tempo de compilação que não precisa das tabelas de métodos virtuais para descobrir qual método deve ser chamado em tempo de execução. O apêndice A apresenta um

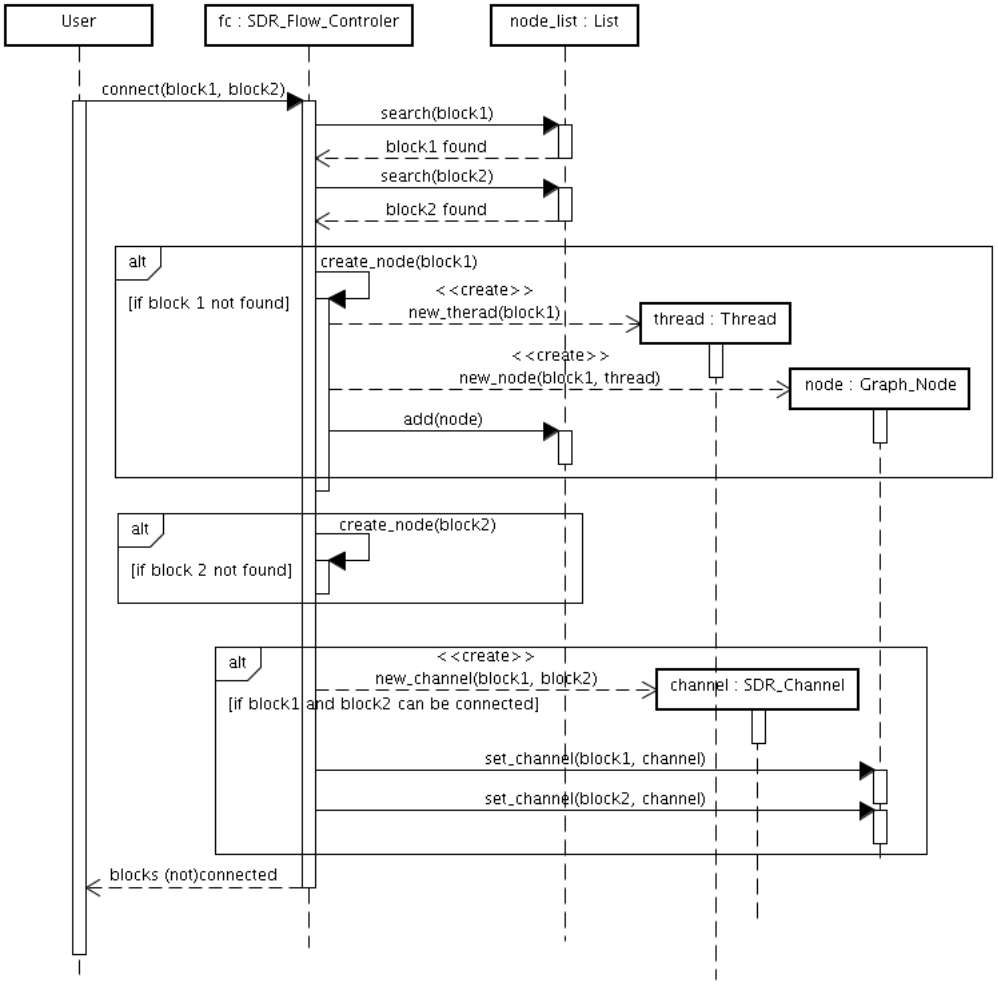

Figura 6.3: Diagrama de seqüência para o método *connect*

exemplo de uso de polimorfismo estático usando C++.

6.3.1 Exemplos de blocos

A seguir são apresentados alguns exemplos de uso das interfaces definidas para implementação de blocos. O código-fonte completo para os exemplos a seguir pode ser visto no apêndice B.

Bloco-fonte

O código 6.1 apresenta um exemplo de bloco-fonte com uma saída que gera inteiros de $1..n$ segundo uma taxa de amostragem definida posteriormente. O tipo de dado que o bloco gera é passado como parâmetro de template para a classe base *SDR_Source_Block*. Sempre que um novo dado é gerado o método *data_ready()* deve ser chamado para notificar o FC que novos dados chegaram no sistema, e o método *read_data()* retorna os últimos dados gerados.

Algoritmo 6.1 Exemplo de bloco-fonte

```
class Int_Source_Block :
public SDR_Source_Block<Int_Source_Block, int, 1> {

  ...

  int* read_data(){ return &_data; }

  ...

  static void alarm(){
    ++_data;
    singleton->data_ready();
  }

  ...
};
```

Bloco de processamento

O código 6.2 mostra um bloco que calcula a multiplicação dos valores das duas entradas e coloca o resultado na saída. O método *do_processing* é chamado pelo FC para fazer o processamento e retorna um ponteiro para os dados gerados, sendo necessário um buffer interno no bloco para armazenar esses dados até que eles sejam adicionados nos canais de saída do bloco. O tipo de todas as entradas do bloco deve ser o mesmo, mas o tipo das saídas pode ser diferente

do tipo das entradas. Isso representa uma limitação que não é muito significante na prática, pois é muito incomum o uso de blocos de processamento com essas características.

Algoritmo 6.2 Exemplo de bloco de processamento

```
class Mult_Int_Proc_Block :
public SDR_Processing_Block
<Mult_Int_Proc_Block, int, int, 2, 1>{

  ...

  int* do_processing(int** data){
    square = data[0][0] * data[1][0];
    return &square;
  }

  ...
};
```

Bloco-sumidouro

O código 6.1 apresenta um exemplo de bloco-sumidouro. Ele é muito semelhante ao exemplo anterior, exceto pelo fato de não possuir saídas. O bloco possui uma entrada e apenas imprime os dados vindos pela entrada na saída padrão.

Algoritmo 6.3 Exemplo de bloco-sumidouro

```
class OStream_Sink_Block :
public SDR_Sink_Block<OStream_Sink_Block, int, 1>{

  ...

  void sink_data(int** input){
    OStream os;
    os << "OStream_Sink_Block::sink_data = "
       <<  input[0][0] << "\n";
   }

  ...
};
```

6.4 Implementação do controlador de fluxo

O controlador de fluxo foi implementado exatamente da forma como está descrito na seção 6.2. Foram utilizadas as implementações de semáforo e thread providas pelo EPOS para as estruturas de controle. Para manter a lista dos nodos que compõe o grafo de fluxo foi utilizada a implementação de uma lista ligada simples fornecida pelo EPOS.

O método *connect* da classe *SDR_Flow_Controler* é parametrizado utilizando templates para que ele possa ser invocado utilizando como argumentos qualquer bloco que implemente as interfaces definidas nas seções anteriores. O código 6.4 mostra como ficou definida a assinatura do método *connect* em função dos templates. O códido completo do método *connect* e o código fonte completo do framework pode ser conferido no apêndice D.

Algoritmo 6.4 Método *connect* da classe *SDR_Flow_Controler*

```
template <
    typename User_Block1, typename User_Block2,
    typename input_type1, typename output_type1,
    typename input_type2, typename output_type2 >
bool connect(
  SDR_Block<User_Block1,input_type1,output_type1> *b1, int b1_output,
  SDR_Block<User_Block2,input_type2,output_type2> *b2, int b2_input );
```

6.4.1 Exemplo de SDR

O código 6.5 mostra um exemplo de um SDR implementado usando a arquitetura proposta. Os blocos utilizados são os mesmos mostrados como exemplo nas seções anteriores. Dois blocos-fonte para gerar inteiros a uma taxa de 100 S/s (amostras por segundo) são criados, juntamente com um bloco de processamento e um bloco-sumidouro. Ao conectar os blocos são especificados os blocos e qual porta de saída está ligada em qual porta de entrada. Finalmente o grafo é iniciado. O código completo deste exemplo pode ser visto no apêndice B

Algoritmo 6.5 Exemplo de SDR implementado usando a arquitetura proposta

```
int main() {
  ...
  Int_Source_Block source0(100, 0);
  Int_Source_Block source1(100, 0);
  Mult_Int_Proc_Block proc;
  OStream_Sink_Block sink;

  fc.connect(&source0, 0, &proc, 0);
  fc.connect(&source1, 0, &proc, 1);
  fc.connect(&proc, 0, &sink, 0);

  fc.start();
  ...
}
```

7 Resultados

Foram realizados testes para medir o overhead intrínseco da arquitetura, que é basicamente o overhead imposto pelo FC ao grafo de fluxo. O objetivo na realização destes testes é dar ao usuário dados que, juntamente com as informações sobre os tempos de execução dos blocos, permitam estimar o tempo total de propagação dos dados pelo grafo de fluxo.

7.1 Testes realizados

Para fazer essas medições, foi implementada a estrutura mostrada na figura 7.1. Um bloco-fonte (*Timestamp source*) gera dados que contêm uma referência do tempo que eles foram gerados. Esses dados são propagados por blocos de processamento vazios (*Dummy block*) que simplesmente repassam os dados para o próximo bloco. Um bloco-sumidouro compara a referência de tempo contida nos dados com o tempo atual, obtendo o tempo que o dado levou para passar por todo o grafo de fluxo. Como blocos vazios foram utilizados, esse tempo representa o overhead da arquitetura para um SDR com a estrutura utilizada.

Figura 7.1: SDR para avaliação do overhead com blocos seriais

Além de um SDR com vários blocos em serial, foram também implementadas estruturas com blocos em paralelo e com blocos de várias entradas, apresentadas nas figuras 7.2 e 7.3, respectivamente. Isso permite avaliar a arquitetura em relação a todas as possíveis variações estruturais que um SDR pode ter. Nestas duas estruturas foram introduzidos dois novos blocos (*Fork block* e *Join block*) para separar e multiplexar o fluxo de dados, respectivamente. Além de um bloco de processamento vazio com várias entradas e saídas.

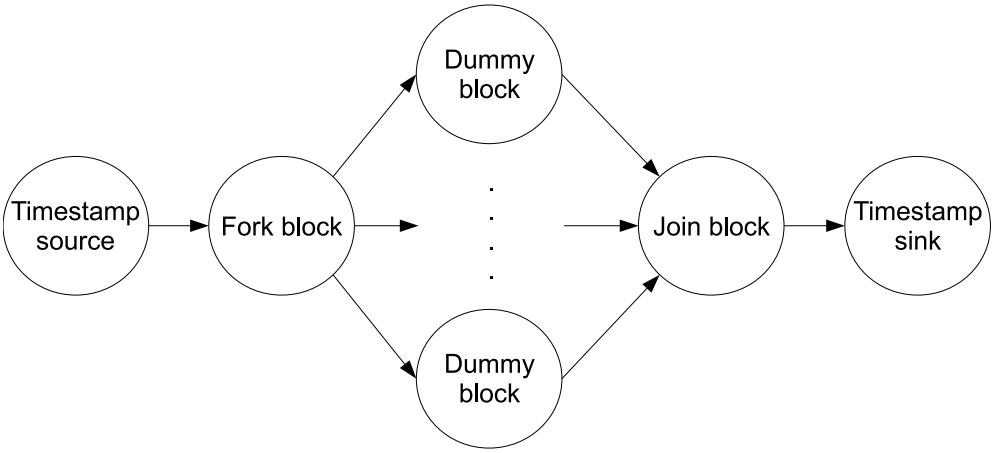

Figura 7.2: SDR para avaliação do overhead com blocos paralelos

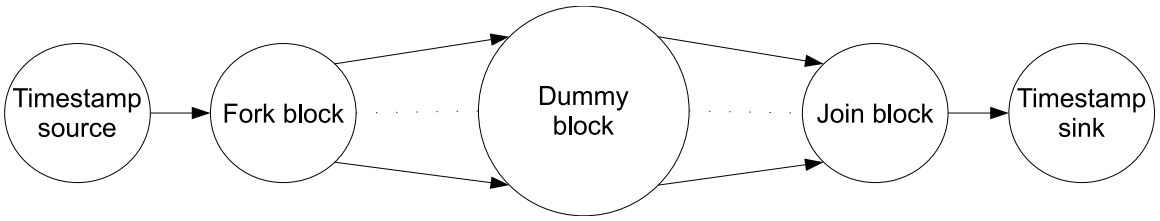

Figura 7.3: SDR para avaliação do overhead com blocos de várias entradas

7.2 Resultados obtidos

A seguir são apresentados os resultados obtidos na implementação das estruturas da seção anterior. Todas as estruturas foram implementadas utilizando apenas blocos em software, pois o overhead mais significativo está na estrutura em software do FC. Todos os tempos apresentados nesta seção são normalizados em relação ao caso mais simples, que é estrutura com apenas um bloco vazio de uma entrada, isso é feito pois os tempos são extremamente dependentes do hardware onde a aplicação está sendo executada, e valores absolutos não agregam nenhuma informação útil quando buscamos por dados mais gerais.

Para cada uma das estruturas mostradas anteriormente foram executados cinco testes. Em cada testes foram propagadas 1000 amostras pela estrutura e coletado o tempo médio e o desvio padrão do tempo de propagação das amostras.

As tabelas 7.1, 7.2 e 7.3 mostram os resultados obtidos na implementação das estruturas com blocos em serial, blocos em paralelo e blocos com multiplas entradas e saídas, respectivamente. As tabelas apresentam os valores médios dos resultados obtidos em cada um dos testes descritos anteriormente, variando o número de blocos/entradas/saídas das estruturas. A figuras 7.4 e 7.5 apresentam de forma gráfica as informações das tabelas 7.1, 7.2 e 7.3.

N° de blocos	Média	Desvio padrão
1	1	0,45
2	1,28	0,6
4	1,78	0,49
8	2,7	0,56
16	4,66	1,01

Tabela 7.1: Tempo de propagação por blocos seriais

N° de blocos	Média	Desvio padrão
1	1,7	0,83
2	2,1	1,18
4	2,88	1,26
8	4,31	1,5
16	7,37	1,64

Tabela 7.2: Tempo de propagação por blocos paralelos

N° de entradas e saídas	Média	Desvio padrão
1	1,69	1,02
2	1,97	0,93
4	2,51	1,1
8	3,5	1,14
16	5,44	1,2

Tabela 7.3: Tempo de propagação por blocos com múltiplas entradas e saídas

Como pode ser observado na figura 7.4, o crescimento do tempo de propagação é linear em ralação ao número de blocos na cadeia e em relação ao número de entradas e saídas dos blocos. Devido a inclusão dos blocos *Fork* e *Join*, as estruturas com blocos em paraleo e com blocos de multiplas entradas e saídas possuem tempos iniciais mais altos do que a estrutura com blocos em serial. Comparando a inclinação das curvas, nota-se que taxa do crescimento do tempo de propagação é bem semelhante para as três estruturas.

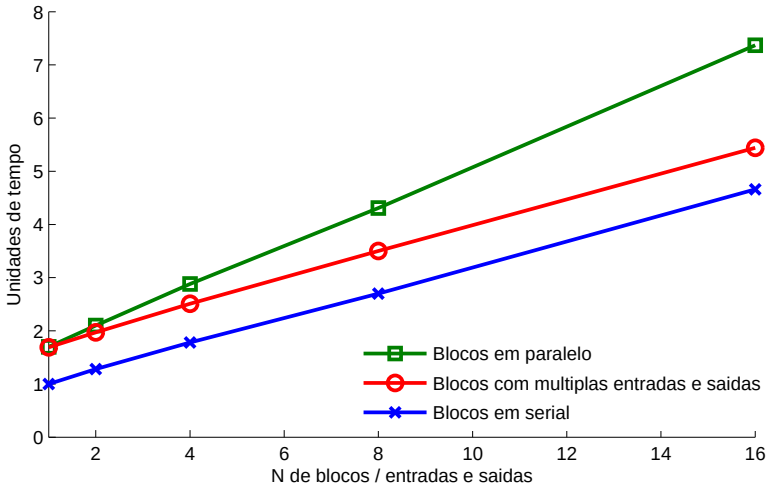

Figura 7.4: Gráfico do tempo médio de propagação

A figura 7.5 mostra o desvio padrão. Apesar de algumas variações iniciais o desvio padrão cresce linearmente. As curvas do desvio padrão para as estruturas onde o número de blocos variou apresentam uma taxa de crescimento semelhante enquanto que, quando varia-se apenas o número de entradas e saídas, o desvio padrão manteve-se praticamente constante. Esses resultados são esperados, pois o número de threads no sistema não aumenta para o último caso, diminuindo o overhead de controle e os pontos de não-determinismo em relação aos casos anteriores. Comparando as escalas das figuras 7.4 e 7.5 nota-se que o desvio padrão ficou bem baixo em relação a média, mostrando que a arquitetura apresenta um comportamento bastante determinístico. Esse não é o caso, por exemplo, do GNU Radio, onde o desvio padrão do tempo que leva para uma amostra chegar a cadeia de processamento após ter sido gerada é maior que o próprio tempo médio (NYCHIS et al., 2009).

Para facilitar a visualização do overhead da arquitetura com a estrutura do SDR variando em várias dimensões, a figura 7.6 apresenta uma superfície gerada a partir da multiplicação dos tempos de propagação da estrutura com blocos em serial e da estrutura com blocos em paralelo. Essa superfície mostra a variação do overhead em relação número cadeias de processamento serial em paralelo.

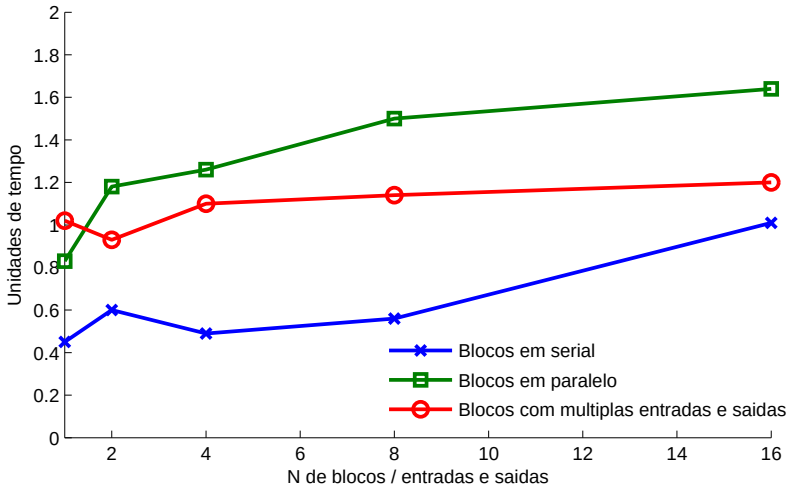

Figura 7.5: Gráfico do desvio padrão tempo de propagação em relação a média

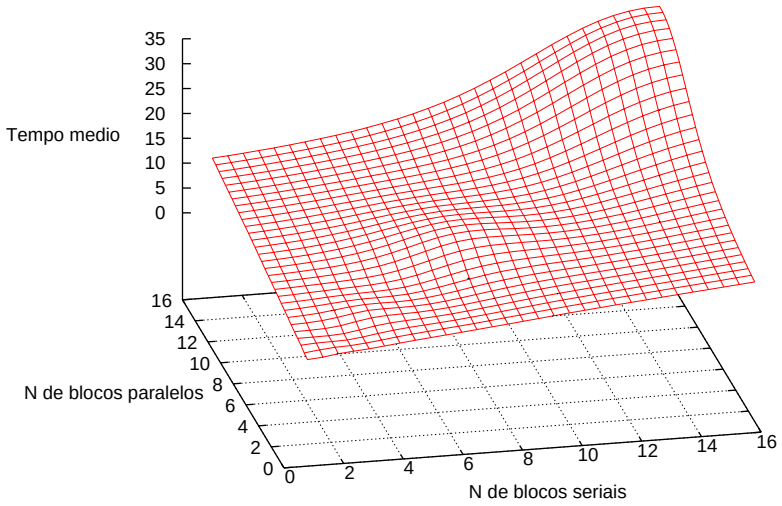

Figura 7.6: Gráfico do tempo de propagação em relação a variação do n° de blocos em serial e paralelo

8 *Considerações finais*

Neste trabalho, uma arquitetura para a implementação de SDRs em sistemas embarcados foi desenvolvida. Um framework modelado e implementado no sistema operacional EPOS utilizando a metodologia ADESD permite o desenvolvimento de SDRs a partir do mapeamento direto de componentes de um modelo de alto nível do mesmo para componentes implementáveis.

Foram implementadas estruturas que comprovam o funcionamento da arquitetura da forma como ela foi descrita e foram obtidos resultados que permitem estimar o overhead da arquitetura para uma determinada estrutura de SDR. Os testes mostraram um crescimento linear do overhead à uma taxa bastante baixa, o que ajuda na ajuda na implementação de SDRs com grandes cadeias de processamento. O desvio padrão do overhead médio também ficou bastante baixo e cresce linearmente em relação ao tamanho da estruturas. Esse desvio padrão baixo facilita muito a implementação de protocolos que apresentam requisitos de tempo preciso, por exemplo, protocolos baseados em multiplexação do tempo (TDMA – *Time Division Multiple Access*).

8.1 Trabalhos futuros

Mais testes devem ser feitos para avaliar as estruturas de comunicação e de controle em hardware. Testes utilizando estrutura de SDR com blocos que estejam em hardware e em software devem feitos para determiar melhor os gargalos da arquitetura.

Protocolos de comunicação usados na prática devem ser implementados para avaliar o desempenho da arquitetura em ambientes reais e para poder fazer uma comparação direta entre a arquitetura proposta e outros trabalhos dsenvolvidos na área.

Referências Bibliográficas

BEAGLEBOARD.ORG. *Beagle Board*. [Online; accessed 18-Junho-2009]. Disponível em: <http://beagleboard.org/>.

BERKEL, K. van et al. Vector processing as an enabler for software-defined radio in handheld devices. *EURASIP: Journal on Applied Signal Processing*, Hindawi Publishing Corp., New York, NY, United States, v. 2005, p. 2613–2625, 2005. ISSN 1110-8657.

CARRO, L.; WAGNER, F. R. *Sistemas Computacionais Embarcados*. Campinas: SBC, 2003. (JAI'03 – XXII Jornadas de Atualização em Informática).

CHAPIN, J. M.; BOSE, V. G. The vanu software radio system. In: *SDR '02: 2002 Software Defined Radio Technical Conference*. [S.l.: s.n.], 2002.

CZARNECKI, K.; EISENECKER, U. *Generative Programming: Methods, Tools, and Applications*. [S.l.]: Addison-Wesley, 2000.

DAVIS, K. V. JTRS-an open, distributed-object computing software radio architecture. In: *18th Digital Avionics Systems Conference*. [S.l.: s.n.], 1999. Volume: 2.

Ettus Research. *USRP*. 2008. [Online; accessed 21-Novembro-2008]. Disponível em: <http://www.ettus.com>.

FERRARI, A.; SANGIOVANNI-VINCENTELLI, A. System design: Traditional concepts and new paradigms. In: *Computer Design, 1999.(ICCD'99) International Conference on*. [S.l.: s.n.], 1999. p. 2–12.

FRöHLICH, A. A. *Application-Oriented Operating Systems*. Tese — Technical University of Berlin, Berlin, 2001.

GAMMA, E. et al. *Design patterns : elements of reusable object-oriented software*. Reading, Mass. [u.a.]: Addison-Wesley, 1994. ISBN 0-201-63361-2.

GLOSSNER, J.; HOKENEK, E.; MOUDGILL, M. The Sandbridge Sandblaster Communications Processor. In: *3rd Workshop on Application Specific Processors*. [S.l.: s.n.], 2004. p. 53–58.

GNU FSF project. *The GNU Radio*. 2009. [Online; accessed 08-November-2009]. Disponível em: <http://www.gnu.org/software/gnuradio>.

GSCHWIND, M. et al. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, v. 26, p. 10–24, March 2006. ISSN 0272-1732.

HASAN, S. M. et al. A Low Cost Multi–band/Multi-mode Radio for Public Safety. In: *SDR '06: 2006 Software Defined Radio Technical Conference*. [S.l.: s.n.], 2006.

HAYKIN, S. Cognitive radio: brain-empowered wireless communications. *IEEE Journal on Selected Areas in Communications*, v. 23, n. 2, p. 201–220, 2005.

KICZALES, G. et al. Aspect-Oriented Programming. In: *Proceedings of the European Conference on Object-oriented Programming'97*. Jyväskylä, Finland: [s.n.], 1997. p. 220–242.

KUMAR, S. et al. A Network on Chip Architecture and Design Methodology. In: *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'02)*. [S.l.: s.n.], 2002. p. 105–112.

LEE, E. A.; MESSERSCHMITT, D. G. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, C-36, p. 24–35, 1987.

LEE, E. A.; MESSERSCHMITT, D. G. Synchronous data flow. *Proceedings of the IEEE*, v. 75, p. 1235– 1245, 1987. ISSN 0018-9219.

LI, Q.; YAO, C. *Real-Time Concepts for Embbeded Systems*. 2 edition. ed. Oxford: CMP-Books, 2003.

LIN, Y. et al. SODA: A Low-power Architecture For Software Radio. In: *ISCA '06: 33rd International Symposium on Computer Architecture*. [S.l.: s.n.], 2006. p. 89–101. ISBN 0-7695-2608-X.

LISHA. *Epos Project*. 2008. [Online; accessed 26-Novembro-2008]. Disponível em: <http://epos.lisha.ufsc.br/>.

MARCONDES, H.; FRöHLICH, A. A. On Hybrid Hw/Sw Components for Embedded System Design. In: *Proceedings of the 17th IFAC World Congress*. Seoul, Korea: [s.n.], 2008. p. 9290–9295.

MARWEDEL, P. *Embedded System Design*. [S.l.]: Kluwer Academic Publishers, 2003. ISBN 1-4020-7690-9.

Mercury Federal Systems. *OpenCPI*. [Online; accessed 29-Junho-2009]. Disponível em: <http://opencpi.org/>.

MILLHAEM, M. D. Sdr advantages for test equipment. In: *IEE Systems Readiness Technology Conference*. [S.l.: s.n.], 2006. p. 565 – 569.

MITOLA, J. The Software Radio Architecture. *IEEE Communications Magazine*, v. 33, n. 5, May 1995.

MITOLA, J.; JR, G. M. Cognitive radio: making software radios more personal. *IEEE Personal Communications*, v. 6, n. 4, p. 13–18, 1999.

NYCHIS, G. et al. Enabling MAC Protocols Implementation on Software-defined Radios. In: *Networked Systems Design and Implementation*. [S.l.: s.n.], 2009.

Paul Pop. Embedded Systems Design: Optimization Challenges. In: *CPAIOR*. [S.l.: s.n.], 2005. p. 16–16.

Rice University. *WARP*. [Online; accessed 18-Junho-2009]. Disponível em: <http://warp.rice.edu>.

SANGIOVANNI-VINCENTELLI, A. et al. Benefits and challenges for platform-based design. In: *DAC '04: Proceedings of the 41st annual conference on Design automation*. New York, NY, USA: ACM, 2004. p. 409–414. ISBN 1-58113-828-8.

SANGIOVANNI-VINCENTELLI, A. L.; MARTIN, G. Platform-Based Design and Software Design Methodology for Embedded Systems. *IEEE Design & Test of Computers*, v. 18, n. 6, p. 23–33, 2001.

SCHWARTZ, M. *Information, Transmission, Modulation and Noise: A Unified Approach to Communication Systems*. 2 edition. ed. [S.l.]: McGraw-Hill, 1970.

SDR Forum. *Software Defined Radio Forum*. [Online; accessed 17-Junho-2009]. Disponível em: <http://www.sdrforum.org/>.

STEVEN KELEM et al. An Elemental Computing Architecture for SD Radio. In: *SDR '07: 2007 Software Defined Radio Technical Conference*. [S.l.: s.n.], 2007.

TENNENHOUSE, D. L.; BOSE, V. G. SpectrumWare – A Software-Oriented Approach to Wireless Signal Processing. In: *ACM Mobile Computing and Networking 95*. [S.l.: s.n.], 1995.

Texas Instruments. *ADS54RF63*. [Online; accessed 17-Junho-2009]. Disponível em: <www.ti.com/ads54rf63-pr>.

WOLF, W. *Computers as Components – Principles of Embedded Computing System Design*. 1 edition. ed. San Francisco: Morgan Kaufmann Publishers, 2001.

YOUNGBLOOD, G. A Software Defined Radio for the Masses, Part 1. *QEX Communications Quarterly, ARRL: Connecticut*, 2002.

APÊNDICE A – Polimorfismo estático utilizando templates em C++

O trecho de código abaixo exemplifica esse tipo de polimorfismo. Uma classe base define uma interface que é implementada usando um método de sua subclasse, que é um parâmetro de template da classe base, passado quando a classe base é especializada.

```
template <class classe_derivada>
class Base {
  void interface() {
    static_cast<classe_derivada*>(this)->implementacao();
  }
};

class Derivada : public Base<Derivada> {
  void implementacao();
};
```

Quando referênciamos objetos de subclasses da classe *Base* através de referências do tipo *Base*, temos que dizer qual o tipo do parametro de template da classe *Base*, isto é, qual a sua subclasse. Dessa forma o compilador sempre consegue resolver as chamadas em tempo de compilação. O trecho de código abaixo mostra um método que recebe como parametro qualquer objeto cuja classe seja subclasse de *Base*:

```
template <typename classe_derivada>
void foo (Base<classe_derivada> *obj) {
  obj->interface();
}
```

Ao chamar o método *foo* passando um objeto da classe *Derivada*, a invocação de interface é resolvida em tempo de compilação e convertida para uma invocação do método *implementacao* da classe *Derivada*. O uso de polimorfismo estatico proporciona uma melhora no tempo de execução em relação ao polimorfismo dinâmico, mas acarreta em um maior consumo de memória devido ao crescimento do tamanho do código. Para cada nova especialização criada, uma nova classe base, especifica para a subclasse, é gerada. No caso do exemplo enterior, é gerado um novo método *foo* para cada tipo de dado diferente usado como argumento para as suas chamadas. Cabe ao desenvolvedor avaliar o *trade-off* entre desempenho e tamanho de código para definir se vale ou não a pena usar polimorfismo estatico.

APÊNDICE B – Exemplos de uso da arquitetura

B.1 Exemplo de bloco-fonte

```
class Int_Source_Block : public SDR_Source_Block<Int_Source_Block, int, 1> {
public:
  int* read_data(){ return &_data; }

  int output_rate(){ return 1; }

public:
  Int_Source_Block(int sampling_rate, int n_samples)
    :_alarm(1000000/sampling_rate, new Handler_Function(&alarm), n_samples)
  { singleton = this; }

  static Int_Source_Block* singleton;

  static void alarm(){
    ++_data;
    singleton->data_ready();
  }

private:
  static int _data;
  Alarm _alarm;
};
```

B.2 Exemplo de bloco de processamento

```
class Mult_Int_Proc_Block :
public SDR_Processing_Block
<Mult_Int_Proc_Block, int, int, 2, 1>{
public:
  int* do_processing(int** data){
    square = data[0][0] * data[1][0];
    return &square;
  }

  int input_rate(){ return 1; }
  int output_rate(){ return 1; }

private:
  int square;
};
```

B.3 Exemplo de bloco-sumidouro

```
class OStream_Sink_Block :
      public SDR_Sink_Block<OStream_Sink_Block, int, 1>{
public:
  void sink_data(int** input){
    OStream os;
    os << "OStream_Sink_Block::sink_data = "
       <<  input[0][i] << "\n";
   }

  int input_rate(){ return 1; }
};
```

B.4 Exemplo de uso do FC

```
int main() {
```

```
  SDR_Flow_Controler fc;

  Int_Source_Block source0(100, 0);
  Int_Source_Block source1(100, 0);
  Mult_Int_Proc_Block proc;
  OStream_Sink_Block sink;

  fc.connect(&source0, 0, &proc, 0);
  fc.connect(&source1, 0, &proc, 1);
  fc.connect(&proc, 0, &sink, 0);

  fc.start();

  while(true);
  return 0;
}
```

APÊNDICE C – Aplicação de teste

```
#include <sdr/sdr.h>
#include <utility/ostream.h>
#include <alarm.h>
#include <thread.h>
#include <utility/handler.h>
#include <tsc.h>
#include <cpu.h>
#include <machine.h>
#include <chronometer.h>

__USING_SYS
using namespace SDR;

typedef struct{
  Chronometer::Microsecond time;
  unsigned int sequence;
} TimeStamp;

Chronometer chronometer;

class TimeStamp_Source :
      public SDR_Source_Block<TimeStamp_Source, TimeStamp, 1>{
public:
  TimeStamp* read_data(){
    _data.time = _time;
    _data.sequence = _sequence;
    return &_data;
```

```
  }

  int output_rate(){
    return 1;
  }

public:
  TimeStamp_Source(int sampling_rate, int n_samples)
    :_alarm(1000000/sampling_rate,
    new Handler_Function(&alarm),
    n_samples)
  {
    singleton = this;
  }

  static TimeStamp_Source* singleton;

  static void alarm(){
    _time = chronometer.read();
    ++_sequence;
    singleton->data_ready();
  }

private:
  static Chronometer::Microsecond _time;
  static unsigned int _sequence;
  Alarm _alarm;
  TimeStamp _data;
};
Chronometer::Microsecond TimeStamp_Source::_time = 0;
unsigned int TimeStamp_Source::_sequence = 0;
TimeStamp_Source* TimeStamp_Source::singleton = 0;


class TimeStamp_Sink :
```

```
      public SDR_Sink_Block<TimeStamp_Sink, TimeStamp, 1>{
public:
  void sink_data(TimeStamp** input){

    Chronometer::Microsecond diff = chronometer.read()-input[0][0].time;

    if(_first_blood && (input[0][0].sequence-1 != _prev)){
      _os << "SEQUENCE ERROR: expected " << (_prev+1)
  << " but got " << input[0][0].sequence << "\n";
    }
    else{
      _first_blood = true;
    }
    _prev = input[0][0].sequence;

    _sum += diff;
    _square_sum += diff*diff;
    _count += 1;

    _os << "Seq: "<< input[0][0].sequence
<< "    Time: " << input[0][0].time
<< "    Diff: " <<  diff << "\n";

    if(_count == _num_samples){
      _os << "Mean: " << time_mean()
  << "   Std. Deviation: " << time_std_deviation() << "\n";

    }
  }

  int input_rate(){
    return 1;
  }

private:
```

```
  int sqrt(int num) {
    int op = num;
    int res = 0;
    // The second-to-top bit is set: 1L<<30 for long
    int one = 1 << 30;

    // "one" starts at the highest power of four <= the argument.
    while (one > op)
      one >>= 2;

    while (one != 0) {
      if (op >= res + one) {
op -= res + one;
res = (res >> 1) + one;
      }
      else {
res >>= 1;
      }
      one >>= 2;
    }
    return res;
  }

public:
  unsigned int time_mean(){ return _sum/_count; }

  unsigned int time_std_deviation(){
    unsigned int aux0 = _square_sum/_count;
    unsigned int aux1 = time_mean(); aux1 *= aux1;
    return sqrt(aux0 - aux1);
  }

public:
  TimeStamp_Sink(int num_samples)
    :_first_blood(false),
```

```
      _prev(0), _sum(0), _square_sum(0), _count(0),
      _num_samples(num_samples)
  { }

private:
  bool _first_blood;
  unsigned int _prev;
  unsigned int _sum;
  unsigned int _square_sum;
  unsigned int _count;
  int _num_samples;
  OStream _os;
};


template<typename type, int num_in_out>
class Dummy_Proc_Block :
      public SDR_Processing_Block
      <Dummy_Proc_Block<type, num_in_out>,
       type, type, num_in_out, num_in_out>{
public:
  type* do_processing(type** data){
    return data[0];
  }

  int input_rate(){
    return 1;
  }

  int output_rate(){
    return 1;
  }
};

template<typename type, int num_outputs>
```

```
class Dummy_Fork_Block :
      public SDR_Processing_Block
    <Dummy_Fork_Block<type, num_outputs>,
     type, type, 1, num_outputs>{
public:
  type* do_processing(type** data){
    return data[0];
  }

  int input_rate(){
    return 1;
  }

  int output_rate(){
    return 1;
  }
};

template<int num_inputs>
class TimeStamp_Join_Block :
      public SDR_Processing_Block
     <TimeStamp_Join_Block<num_inputs>,
      TimeStamp, TimeStamp, num_inputs, 1>{
public:
  TimeStamp* do_processing(TimeStamp** data){

    for (int i = 0; i < num_inputs-1; ++i) {
      if(data[i][0].sequence != data[i+1][0].sequence)
_os << "JOIN ERROR - Inputs " << i << "," << (i+1)
    << " Seqs " << data[i][0].sequence
    << "," << data[i+1][0].sequence << "\n";
    }
    return data[0];
  }
```

```
  int input_rate(){
    return 1;
  }

  int output_rate(){
    return 1;
  }
private:
  OStream _os;
};


template<typename type>
class OStream_Sink :
      public SDR_Sink_Block<OStream_Sink<type>, type, 1>{
public:
  void sink_data(type** input){
    os << "OStream_Sink::sink_data = ";
    for (int i = 0; i < input_rate(); ++i) {
      os << input[0][i] << ", ";
    }
    os << "\n";
  }

  int input_rate(){
    return 1;
  }

private:
  OStream os;

};

enum{
  SERIAL_NUM_BLOCKS = 1,
```

```
  SERIAL_NUM_SAMPLES = 10000,
  SERIAL_SAMPLING_RATE = 400
};
void serial_test(){

  OStream os;

  os << "\nCreating blocks\n";

  TimeStamp_Source
  tm_source(SERIAL_SAMPLING_RATE, SERIAL_NUM_SAMPLES);
  Dummy_Proc_Block<TimeStamp, 1>
  dummy_blocks[SERIAL_NUM_BLOCKS];
  TimeStamp_Sink
  ts_sink(SERIAL_NUM_SAMPLES);

  os << "\nCreating FC\n";

  SDR_Flow_Controler fc;

  bool x = false;

  os << "\nConnecting blocks\n";

  fc.connect(&tm_source, 0, &dummy_blocks[0], 0);
  for (int i = 0; i < SERIAL_NUM_BLOCKS-1; ++i) {
    fc.connect(&dummy_blocks[i], 0, &dummy_blocks[i+1], 0);
  }
  fc.connect(&dummy_blocks[SERIAL_NUM_BLOCKS-1], 0, &ts_sink, 0);


  os << "\nStarting flow graph\n";

  chronometer.start();
```

```
  x = fc.start();
  if(!x) os << "ERROR fc.start()\n";

  os << "\nDone!\n\n";

  while(true)
}

enum{
  PARALLEL_NUM_BLOCKS = 2,
  PARALLEL_NUM_SAMPLES = 10000,
  PARALLEL_SAMPLING_RATE = 400
};
void parallel_test(){

  OStream os;

  os << "\nCreating blocks\n";

  TimeStamp_Source
tm_source(PARALLEL_SAMPLING_RATE, PARALLEL_NUM_SAMPLES);
  Dummy_Fork_Block<TimeStamp, PARALLEL_NUM_BLOCKS>
fork;
  Dummy_Proc_Block<TimeStamp, 1>
dummy_blocks[PARALLEL_NUM_BLOCKS];
  TimeStamp_Join_Block<PARALLEL_NUM_BLOCKS>
join;
  TimeStamp_Sink
ts_sink(PARALLEL_NUM_SAMPLES);

  os << "\nCreating FC\n";

  SDR_Flow_Controler fc;

  bool x = false;
```

```
  os << "\nConnecting blocks\n";

  fc.connect(&tm_source, 0, &fork, 0);
  for (int i = 0; i < PARALLEL_NUM_BLOCKS; ++i) {
    fc.connect(&fork, i, &dummy_blocks[i], 0);
    fc.connect(&dummy_blocks[i], 0, &join, i);
  }
  fc.connect(&join, 0, &ts_sink, 0);

  os << "\nStarting flow graph\n";

  chronometer.start();

  x = fc.start();
  if(!x) os << "ERROR fc.start()\n";

  os << "\nDone!\n\n";

  while(true);
}
void parallel_test2(){

  OStream os;

  os << "\nCreating blocks\n";

  TimeStamp_Source
tm_source(PARALLEL_SAMPLING_RATE, PARALLEL_NUM_SAMPLES);
  Dummy_Fork_Block<TimeStamp, PARALLEL_NUM_BLOCKS>
fork;
  Dummy_Proc_Block<TimeStamp, PARALLEL_NUM_BLOCKS>
dummy_block;
  TimeStamp_Join_Block<PARALLEL_NUM_BLOCKS>
join;
```

```
  TimeStamp_Sink
ts_sink(PARALLEL_NUM_SAMPLES);

  os << "\nCreating FC\n";

  SDR_Flow_Controler fc;

  bool x = false;

  os << "\nConnecting blocks\n";

  fc.connect(&tm_source, 0, &fork, 0);
  for (int i = 0; i < PARALLEL_NUM_BLOCKS; ++i) {
    fc.connect(&fork, i, &dummy_block, i);
    fc.connect(&dummy_block, i, &join, i);
  }
  fc.connect(&join, 0, &ts_sink, 0);

  os << "\nStarting flow graph\n";

  chronometer.start();

  x = fc.start();
  if(!x) os << "ERROR fc.start()\n";

  os << "\nDone!\n\n";

  while(true);
}


int main()
{
  serial_test();
  parallel_test();
```

```
  parallel_test2();

  return 0;
}
```

APÊNDICE D – Código-fonte do framework implementado

D.1 Arquivo sdr_block.h

```
#ifndef __sdr_block_h
#define __sdr_block_h

#include <system/config.h>

__BEGIN_SYS

namespace SDR{

typedef enum{
  SOURCE_BLOCK,
  PROCESSING_BLOCK,
  SINK_BLOCK,
}sdr_block_t;


template <typename Block_Type, typename input_type, typename output_type>
class SDR_Block {

public:
  SDR_Block()
    :_enabled(false), _mup(0)
  {   }
```

```
output_type* do_processing(input_type** input){
  return static_cast<Block_Type*>(this)->do_processing(input);
}

int input_rate(){
  return static_cast<Block_Type*>(this)->input_rate();
}

int output_rate(){
  return static_cast<Block_Type*>(this)->output_rate();
}

int n_inputs(){
  return static_cast<Block_Type*>(this)->n_inputs();
}

int n_outputs(){
  return static_cast<Block_Type*>(this)->n_outputs();
}

int input_size(){
  return static_cast<Block_Type*>(this)->input_size();
}

int output_size(){
  return static_cast<Block_Type*>(this)->output_size();
}

sdr_block_t block_type(){
  return static_cast<Block_Type*>(this)->block_type();
}

bool enabled(){ return _enabled;}

void enabled(bool val) { _enabled = val;}
```

```
  void* mup(){return _mup;}

  void mup(void* val){_mup = val;}

  bool* enable_ptr() {return &_enabled;}

private:
  bool _enabled;
  void* _mup;
};

}

__END_SYS

#endif
```

D.2 Arquivo sdr_source_block.h

```
#ifndef __sdr_source_block_h
#define __sdr_source_block_h

#include "sdr_block.h"
#include <semaphore.h>

__BEGIN_SYS

namespace SDR {

template <class User_Block, typename output_type, int noutputs = 1>
class SDR_Source_Block :
    public SDR_Block<SDR_Source_Block<User_Block, output_type, noutputs>, int, outp

public:
```

```
  output_type* do_processing(int** input){
    return static_cast<User_Block*>(this)->read_data();
  }

  int output_rate(){
    return static_cast<User_Block*>(this)->output_rate();
  }

public:

  void data_ready(){
    Semaphore *sem = reinterpret_cast<Semaphore*>(this->mup());
    sem->v();
  }

  int n_inputs(){return 0;}

  int n_outputs(){return noutputs;}

  int input_rate(){ return 0; }

  int input_size(){return 0;}

  int output_size(){return sizeof(output_type);}

  sdr_block_t block_type(){return SOURCE_BLOCK;}

};

}

__END_SYS

#endif
```

D.3 Arquivo sdr_processing_block.h

```
#ifndef __sdr_processing_block_h
#define __sdr_processing_block_h

#include "sdr_block.h"

__BEGIN_SYS

namespace SDR {

template <class User_Block, typename input_type, typename output_type, int ninputs
class SDR_Processing_Block :
  public SDR_Block<
    SDR_Processing_Block<User_Block, input_type, output_type, ninputs, noutputs>, i

public:
  output_type* do_processing(input_type** input){
    return static_cast<User_Block*>(this)->do_processing(input);
  }

  int input_rate(){
    return static_cast<User_Block*>(this)->input_rate();
  }

  int output_rate(){
    return static_cast<User_Block*>(this)->output_rate();
  }

public:
  int n_inputs(){return ninputs;}

  int n_outputs(){return noutputs;}

  int input_size(){return sizeof(input_type);}
```

```
  int output_size(){return sizeof(output_type);}

  sdr_block_t block_type(){return PROCESSING_BLOCK;}

};

}

__END_SYS


#endif
```

D.4 Arquivo sdr_sink_block.h

```
#ifndef __sdr_sink_block_h
#define __sdr_sink_block_h

#include "sdr_block.h"

__BEGIN_SYS

namespace SDR{

template <class User_Block, typename input_type, int ninputs = 1>
class SDR_Sink_Block :
  public SDR_Block<SDR_Sink_Block<User_Block, input_type, ninputs>, input_type, int

public:
  typedef input_type input_type_t;
  typedef int output_type_t;

public:

  int* do_processing(input_type** input){
```

```
    static_cast<User_Block*>(this)->sink_data(input);
    return reinterpret_cast<int*>(0);
  }

  int input_rate(){
    return static_cast<User_Block*>(this)->input_rate();
  }


public:

  int n_inputs(){return ninputs;}

  int n_outputs(){return 0;}

  int output_rate(){ return 0; }

  int input_size(){return sizeof(input_type);}

  int output_size(){return 0;}

  sdr_block_t block_type(){return SINK_BLOCK;}

};

}

__END_SYS

#endif
```

D.5 Arquivo sdr_fifo.h

```
#ifndef __sdr_fifo_h
#define __sdr_fifo_h
```

```
__BEGIN_SYS

namespace SDR {

class SDR_FIFO{

public:
  typedef enum{
    SUCCESS,
    FIFO_OVERFLOW,
    FIFO_EMPTY,
  } result_t;

public:
  SDR_FIFO(int fifo_size, int data_size)
    :_begin(0), _end(0), _size(0), _fifo_size(fifo_size*data_size), _data_size(data
  {
    _buffer = new char[fifo_size*data_size];
  }

  ~SDR_FIFO(){
    delete[] _buffer;
  }

  result_t add(char* data){

    db<SDR_FIFO>(TRC) << "SDR_FIFO::adding " << *reinterpret_cast<int*>(data) << "\

    if((_size + _data_size) <= _fifo_size){

      for (int i = 0; i < _data_size; ++i) {
_buffer[_begin] = data[i];
_begin = ++_begin % _fifo_size;
      }
```

```
      _size += _data_size;

      return SUCCESS;
    }
    else
      return FIFO_OVERFLOW;
  }

  result_t remove(char* dest){

    if(_size != 0){

      for (int i = 0; i < _data_size; ++i) {
dest[i] = _buffer[_end];
_end = ++_end % _fifo_size;
      }
      _size -= _data_size;

      db<SDR_FIFO>(TRC) << "SDR_FIFO::removed " << *reinterpret_cast<int*>(dest) <<

      return SUCCESS;
    }
    else
      return FIFO_EMPTY;
  }

  int size(){
    return _size/_data_size;
  }

private:
  int _begin;
  int _end;
  int _size;
  int _fifo_size;
```

```
  int _data_size;
  char *_buffer;

};

}
__END_SYS

#endif
```

D.6 Arquivo sdr_channel.h

```
#ifndef __sdr_channel_h
#define __sdr_channel_h

#include "sdr_fifo.h"
#include <semaphore.h>

__BEGIN_SYS

namespace SDR{

class SDR_Channel{

public:
  typedef SDR_FIFO::result_t result_t;

public:
  SDR_Channel(int fifo_size, int data_size)
    :_fifo(fifo_size, data_size), _sem(0)
  {   }

  result_t write(char* data){
    result_t aux = _fifo.add(data);
    if(aux == SDR_FIFO::SUCCESS)
```

```
      _sem.v();
    return aux;
  }

  result_t read(char* dest){
    return _fifo.remove(dest);
  }

  int size() { return _fifo.size(); }

  void wait(){
    _sem.p();
  }

private:
  SDR_FIFO _fifo;
  Semaphore _sem;

};

}
__END_SYS

#endif
```

D.7 Arquivo sdr.h

```
#ifndef __sdr_h
#define __sdr_h

#include "sdr_source_block.h"
#include "sdr_processing_block.h"
#include "sdr_sink_block.h"
#include "sdr_channel.h"
#include <utility/list.h>
```

```
#include <thread.h>
#include <semaphore.h>

__BEGIN_SYS

namespace SDR {

class SDR_Flow_Controler {

private:
  enum{
    BUFFER_THRESHOLD = 3,
  };

private:
  class Graph_Node{
  public:
    Graph_Node()
    :block(0), block_type(SOURCE_BLOCK), block_enabled_ptr(0),
    n_inputs(0), n_outputs(0),
    input_channels(0), output_channels(0),
    thread(0)
    {}
    ~Graph_Node(){
      delete[] input_channels;
      delete[] output_channels;
    }
  public:
    void *block;
    sdr_block_t block_type;
    bool *block_enabled_ptr;
    int n_inputs;
    int n_outputs;
    SDR_Channel **input_channels;
    SDR_Channel **output_channels;
```

```
    Thread *thread;
  };


public:
  SDR_Flow_Controler()
    : _nodes()
  {

  }

public:

  template<typename User_Block1, typename User_Block2, typename input_type1, typena
  bool connect(
    SDR_Block<User_Block1,input_type1,output_type1> *b1, int b1_output,
    SDR_Block<User_Block2,input_type2,output_type2> *b2, int b2_input)
  {

    db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::connect(b1 = " << b1 << ",
<< ", b2 = " << b2 << ", b2_input = " << b2_input << ")\n";

    if((b1->output_size() != b2->input_size()) || (b1_output >= b1->n_outputs()) ||
      db<SDR_Flow_Controler>(ERR) << "SDR_Flow_Controler::connect-> Incompatible in
      return false;
    }

    db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::connect-> checking/creating

    List<Graph_Node>::Element* node_b1 = search(b1);
    List<Graph_Node>::Element* node_b2 = search(b2);

    if(node_b1 == 0){
      Graph_Node* gnode = create_node(b1);
      node_b1 = new List<Graph_Node>::Element(gnode);
```

```
  _nodes.insert(node_b1);
}

if(node_b2 == 0){
  Graph_Node* gnode = create_node(b2);
  node_b2 = new List<Graph_Node>::Element(gnode);
  _nodes.insert(node_b2);
}

Graph_Node *gnode_b1 = node_b1->object();
Graph_Node *gnode_b2 = node_b2->object();

db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::connect-> checking connecti

if(gnode_b1->output_channels[b1_output] != 0){
  db<SDR_Flow_Controler>(ERR) << "SDR_Flow_Controler::connect-> Block 1" << b1
  return false;
}

if(gnode_b2->input_channels[b2_input] != 0){
  db<SDR_Flow_Controler>(ERR) << "SDR_Flow_Controler::connect-> Block 2" << b2
  return false;
}

int fifo_size = BUFFER_THRESHOLD*(b1->output_rate() + b2->input_rate());

db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::connect-> Creating channel

SDR_Channel *channel = new SDR_Channel(fifo_size, b2->input_size());

db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::connect-> Connecting blocks
gnode_b1->output_channels[b1_output] = channel;
gnode_b2->input_channels[b2_input] = channel;

return true;
```

```
  }

  bool start(){
    db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::start()\n";

    if(!check_graph()) return false;

    List<Graph_Node>::Iterator iter = _nodes.begin();
    do{
      Graph_Node *g = iter->object();
      *(g->block_enabled_ptr) = true;
      ++iter;
    }while(iter != _nodes.end());

    return true;
  }

  void stop(){
    db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::stop()\n";

    db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::stop() - Stopping sources\n

    List<Graph_Node>::Iterator iter = _nodes.begin();
    do{
      Graph_Node *g = iter->object();
      *(g->block_enabled_ptr) = false;
      ++iter;
    }while(iter != _nodes.end());

  }

private:

  template<typename User_Block, typename input_type, typename output_type>
  Graph_Node* create_node(SDR_Block<User_Block,input_type,output_type> *block){
```

```
  Graph_Node *ptr = new Graph_Node();
  ptr->block = block;
  ptr->block_type = block->block_type();
  ptr->block_enabled_ptr = block->enable_ptr();
  ptr->n_inputs = block->n_inputs();
  ptr->input_channels = new SDR_Channel*[block->n_inputs()];
  for (int i = 0; i < block->n_inputs(); ++i) {
    ptr->input_channels[i] = 0;
  }
  ptr->n_outputs = block->n_outputs();
  ptr->output_channels = new SDR_Channel*[block->n_outputs()];
  for (int i = 0; i < block->n_outputs(); ++i) {
    ptr->output_channels[i] = 0;
  }
  block->enabled(false);

  db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::create_node() - Block "<< b

  sdr_block_t blk_type = block->block_type();
  switch (blk_type) {
  case SOURCE_BLOCK:
    ptr->thread = new Thread(&thread_source_block, block, ptr, Thread::READY, Thr
    break;
  case PROCESSING_BLOCK:
    ptr->thread = new Thread(&thread_proc_block, block, ptr, Thread::READY, Threa
    break;
  case SINK_BLOCK:
    ptr->thread = new Thread(&thread_sink_block, block, ptr, Thread::READY, Threa
    break;
  default:
    break;
  }

  return ptr;
}
```

```
  template<typename User_Block, typename input_type, typename output_type>
  typename List<Graph_Node>::Element* search(SDR_Block<User_Block, input_type, outp
    typename List<Graph_Node>::Element *e = _nodes.head();
    for(; e && (e->object()->block != block); e = e->next());
    return e;
  }

  bool check_graph(){
    db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::check_graph()\n";

    List<Graph_Node>::Iterator iter = _nodes.begin();
    do{
      Graph_Node *g = iter->object();
      for (int i = 0; i < g->n_inputs; ++i) {
if(g->input_channels[i] == 0){
  db<SDR_Flow_Controler>(ERR) << "Input " << i << " unconnected on block " << g->bl
  return false;
}
      }
      for (int i = 0; i < g->n_outputs; ++i) {
if(g->output_channels[i] == 0){
  db<SDR_Flow_Controler>(ERR) << "Output " << i << " unconnected on block " << g->b
  return false;
}
      }
      ++iter;
    }while(iter != _nodes.end());
    return true;
  }

  template<typename User_Block, typename input_type, typename output_type>
  static int thread_proc_block(SDR_Block<User_Block, input_type, output_type> *blk,
```

```
    db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::thread_proc_block() - Block

    input_type** buffer = new input_type*[blk->n_inputs()];
    for (int i = 0; i < blk->n_inputs(); ++i) {
      buffer[i] = new input_type[blk->input_rate()];
    }

    while(true){

      if(!blk->enabled()){
Thread::yield();
continue;
      }

      db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::thread_proc_block() - Blo

      for (int i = 0; i < node->n_inputs; ++i) {
node->input_channels[i]->wait();
      }

      db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::thread_proc_block() - Blo

      bool has_data = true;
      for (int i = 0; i < node->n_inputs; ++i) {
has_data = has_data && (node->input_channels[i]->size() >= blk->input_rate());
      }

      if(has_data){

db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::thread_proc_block() - Block: "

for (int i = 0; i < node->n_inputs; ++i) {
  for (int j = 0; j < blk->input_rate(); ++j) {
    SDR_Channel::result_t r = node->input_channels[i]->read(reinterpret_cast<char*>
    if(r != SDR_FIFO::SUCCESS){
```

```
      db<SDR_Flow_Controler>(ERR) << "SDR_Flow_Controler::thread_proc_block() - Blo
      while(true);
    }
  }
}

db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::thread_proc_block() - Block: "

output_type* output = blk->do_processing(buffer);

db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::thread_proc_block() - Block: "

for (int i = 0; i < blk->output_rate(); ++i) {
  for (int j = 0; j < node->n_outputs; ++j) {
    SDR_Channel::result_t r = node->output_channels[j]->write(reinterpret_cast<char
    if(r != SDR_FIFO::SUCCESS){
      db<SDR_Flow_Controler>(ERR) << "SDR_Flow_Controler::thread_proc_block() - Blo
      while(true);
    }
  }
}

      }
    }
    return 0;
  }


  template<typename User_Block, typename input_type, typename output_type>
  static int thread_sink_block(SDR_Block<User_Block, input_type, output_type> *blk,

    db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::thread_sink_block() - Block

    input_type** buffer = new input_type*[blk->n_inputs()];
    for (int i = 0; i < blk->n_inputs(); ++i) {
```

```
      buffer[i] = new input_type[blk->input_rate()];
    }

    while(true){

      if(!blk->enabled()){
Thread::yield();
continue;
      }

      db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::thread_sink_block() - Blo

      for (int i = 0; i < node->n_inputs; ++i) {
node->input_channels[i]->wait();
      }

      db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::thread_sink_block() - Blo

      bool has_data = true;
      for (int i = 0; i < node->n_inputs; ++i) {
has_data = has_data && (node->input_channels[i]->size() >= blk->input_rate());
      }

      if(has_data){

db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::thread_sink_block() - Block: "

for (int i = 0; i < node->n_inputs; ++i) {
  for (int j = 0; j < blk->input_rate(); ++j) {
  SDR_Channel::result_t r = node->input_channels[i]->read(reinterpret_cast<char*>(&
  if(r != SDR_FIFO::SUCCESS){
    db<SDR_Flow_Controler>(ERR) << "SDR_Flow_Controler::thread_sink_block() - Block
    while(true);
  }
  }
```

```
}

      db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::thread_sink_block() - Blo

      blk->do_processing(buffer);
    }
  }
  return 0;
}

  template<typename User_Block, typename input_type, typename output_type>
  static int thread_source_block(SDR_Block<User_Block, input_type, output_type> *bl

    db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::thread_source_block() - Blo

    Semaphore wait_data_sem(0);
    blk->mup(&wait_data_sem);

    input_type** buffer = 0;

    while(true){

      if(!blk->enabled()){
Thread::yield();
continue;
      }

      db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::thread_source_block() - B

      wait_data_sem.p();

      db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::thread_source_block() - B

      output_type* output = blk->do_processing(buffer);
```

```
      db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::thread_source_block() - B

      for (int i = 0; i < blk->output_rate(); ++i) {
db<SDR_Flow_Controler>(TRC) << "SDR_Flow_Controler::thread_source_block() - Block:
for (int j = 0; j < node->n_outputs; ++j) {
  SDR_Channel::result_t r = node->output_channels[j]->write(reinterpret_cast<char*>
  if(r != SDR_FIFO::SUCCESS){
    db<SDR_Flow_Controler>(ERR) << "SDR_Flow_Controler::thread_source_block() - Blo
    while(true);
  }
}
      }

     }
    return 0;
  }


private:
  List<Graph_Node> _nodes;

};

}

__END_SYS

#endif
```

APÊNDICE E – Artigo

An Architecture for Software-defined Radio Implementation on Embedded Systems

Tiago Rogério Mück, Roberto de Matos and Antônio Augusto Fröhlich[1]

[1] Laboratory for Software and Hardware Integration (LISHA)
Federal University of Santa Catarina (UFSC)
Florianópolis – Brazil

tiago@lisha.ufsc.br

Abstract. *Nowadays there is a lot of wireless communication protocols being used, and this results in a series of difficulties for developers of systems that interacts with devices which can use different communication protocols. Software-defined radios (SDR) aims to solve this problem by using a software-based approach to provide flexibility on the implementation of the protocols physical layer. In this paper, an analysis of SDR implementation approaches is presented, and an architecture for SDR implementation is proposed. The architecture uses hybrid HW/SW components and programmable logic devices in order to enable the efficient implementation of software-defined radios from high level models. The results show that the architecture imposes a deterministic overhead to the SDR processing chain.*

Resumo. *Existe uma grande quantidade de protocolos de comunicação sem fio sendo usados, e isso implica uma série de dificuldades aos desenvolvedores de sistemas embarcados que têm que interagir com dispositivos que podem possuir, cada um, um protocolo de comunicação diferente. Os rádios definidos por software (SDR – Software-defined Radio) procuram solucionar esse problema utilizando uma abordagem baseada em software para dar flexibilidade na implementação dos protocolos. Neste trabalho, uma arquitetura para implementação de SDRs em sistemas embarcados é desenvolvida. A arquitetura utiliza componentes híbridos de HW/SW em hardware programável para tornar possível o uso de SDRs em sistemas embarcados dentro dos requisitos de custo, consumo e desempenho dos mesmos. Os resultados obtidos mostram que a arquitetura impõe um overhead bastante baixo e determinístico à cadeia de processamento do SDR.*

1. Introduction

Nowadays there is a lot of wireless communication protocols being used, and this yields several difficulties for developers of systems that interacts with devices which can use different communication protocols. Providing adaptability to the communication protocols is a goal to be achieved. However, the hardware-based architecture of traditional radios imposes several limitations in providing this adaptability.

In traditional radios, each element of the receive/transmit hardware chain has a specific function, and the components are designed to work according to a fixed protocol.

When some protocol's parameters change, the information is not transmitted correctly by the radio anymore. Hardware components needs to be replaced so the system can work with new standards. This lack of flexibility leads to higher development costs and time-to-market.

Software-defined radios (SDR) follow a software-based approach to eliminate the limitations imposed by the traditional radios. The idea behind SDRs is to use a multi-band system that allows transmitting and receiving in the frequencies of interest, called RF front-end, and does all the modulation and demodulation of the signal in software instead of hardware, thus allowing more flexibility on implementing the physical layer of the communication protocols.

The use of general purpose systems is very popular for implementing SDRs. Frameworks like the GNU Radio [2] can be used to easily implement SDRs from high level models on common PCs. Moreover, most of the protocols require a big amount of processing power to be implemented, and the requirements of cost, size and power consumption of most applications do not allow the use of a powerful PC to implement the system.

Usually, more specific solutions are used to implement SDRs, such as DSPs, GPPs with SIMD co-processors and programmable logic devices (PLD) like FPGAs. However, the use of this hybrid systems may present high project risks due to the difficulties in translating a high level model of the SDR to an implementation. To overcome these issues this paper proposes a new architecture for the implementation of SDRs.

Following the *Application-driven Embedded System Design* (ADESD) methodology [3] and using the concept of *hybrid HW/SW components* [4], the proposed architecture allows the efficient implementation of SDRs in FPGAs by mapping parts of a high level functional model of the SDR directly to components that can migrate between the hardware and software domains in a transparent way.

2. SDR implementation approaches

There are many ways of implementing SDRs. Overall, these implementations can be generalized into three basic approaches: using general purpose processors, using programmable signal processing hardware, and using dedicated hardware.

The general purpose processor approach consists in the general SDR architecture. In this model, all the processing is made using a GPP, achieving the highest flexibility and ease of development. The GNU Radio [2] framework provides means of implementing SDRs on common PCs and is an example of an architecture that follows this approach. However, using general purpose hardware usually increases the costs of the system, and the overhead imposed by general purpose operating systems can forbid the efficient implementation of time-strict protocols [5].

In the second approach, the processing is made by programmable devices designed for specific functions. These devices includes: *digital signal processors* (DSP), GPPs with SIMD co-processors, and devices for specific functions like dedicated *digital up/down converters* (DUC/DDC). The hardware project on this approach tends to be more complicated and, typically, a FPGA is used to route the signals among the possible ways in the processing chain. In the last few years many architectures have been proposed based on

this approach [6] [7] [8] [9]. The disadvantages of this architectures is the complexity of the software implementation. On SODA [6], for example, the synchronization between the GPP and the SIMD co-processors and the optimization of the signal processing functions were made manually in assembly code.

In the dedicated hardware approach, most of the processing chain is hard-coded on PLDs. The implementation of the desired protocol is made in hardware. This approach may look like the traditional radios way, but, since the hardware can be reconfigured, the radio can be considered a SDR. From all approaches, this one yields the better trade-off between the hardware cost, size and performance. However, the unfriendly environment for developing the signal processing functions and the lack of software support for controlling the data flow on the hardware may present high project risks and time-to-market.

Ideally, architectures that keep the high level of abstraction for the SDR developers should be used, for example the GNU Radio. However, as it was explained earlier, most of the applications cannot be implemented using PCs. Architectures that uses dedicated hardware may meet the applications requirements, but offer many difficulties to translate a high level model of the SDR to an implementation. Therefore, SDR developers should try to follow an approach that combines both the good trade-off of programmable hardware and the flexibility of architectures like GNU Radio.

3. An architecture for SDR implementation

Given the difficulties showed on the previous sections, this paper proposes a new architecture for SDR development. This new architecture uses the ADESD methodology [3] and hybrid hardware/software components [4] to enable the implementation of SDRs using a dedicated hardware approach. A framework allows the easy translation of a high level model of the SDR to an implementation.

The architecture is based on the *Synchronous Data Flow* (SDF) model, that is a very common model for designing digital signal processing applications. In this model, the SDR processing chain is abstracted as a flow graph, where the node represents processing blocks and the edges represents the data flow between the blocks. Figure 1 presents an overview of the proposed architecture. The processing blocks have their functions implemented using hybrid components, so the processing blocks can be in hardware, software or both, in a transparent way to the SDR developers. The blocks are connected through FIFO channels, that are allocated by the architecture's framework in hardware or software, depending on the implementation of the blocks that it connects. An entity called *Flow Controller* controls the data flow between the blocks.

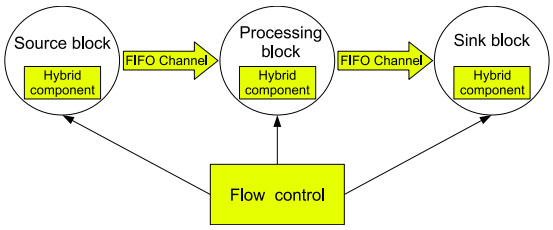

Figure 1. Overview of the proposed architecture

As depicted in figure 1, the architecture defines three different kinds of blocks: the

source blocks are responsible for the insertion of data on the flow graph. They usually abstract input devices like RF front-ends and ADCs. The *processing blocks* are responsible for the signal processing. Finally *sink blocks* consumes the data from the flow graph. It can abstracts output devices like a DAC and a RF front-end, or provide a callback mechanism for higher protocols layers.

The blocks have an interface that defines its number of inputs and outputs. For each execution of the block function, the interface defines the number of data elements consumed and generated. These informations are used by the flow controller to correctly connect the blocks and calculate the FIFO's size.

The flow controller is also responsible for controlling the data flow between the blocks. This is accomplished by creating a thread for each block, where its function is executed. The threads are synchronized using semaphores associated to the FIFO channels. They remain locked on the semaphores associated to the channels connected to the block's inputs. Every time an element is added to a channel, the *v()* method of its associated semaphore is called, unlocking the threads that consume the data from the channels.

Since this software control mechanism could be a bottleneck if both blocks are in hardware, the architecture allows hardware blocks communicate directly without software intervention. This is done through the deployment of HW FIFO channels that is supported by the hardware structure showed in figure 2. This structure has read ports, write ports and a set of FIFOs. The interconnection between this elements is defined by registers that are configured by software. All the blocks that are in hardware have its inputs connected to the structure's read ports, and its outputs connected to the write ports. This way any hardware block can be directly connected to any other hardware block through a hardware channel. The structure is also connected to the system bus, so hardware and software blocks can exchange data with each other.

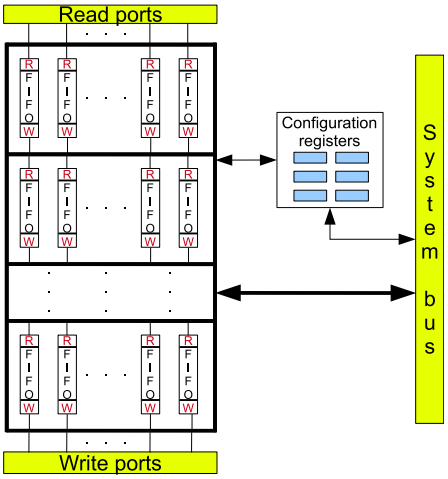

Figure 2. Flow controller structure for HW channel allocation

4. Results

Tests were made to evaluate the overhead imposed over the data flow by the control structures. The three structures shown in figure 3 were implemented to evaluate the increase of

the overhead in terms of the number of blocks in serial, the number of blocks in parallel and the number of inputs/outputs of a block, respectively.

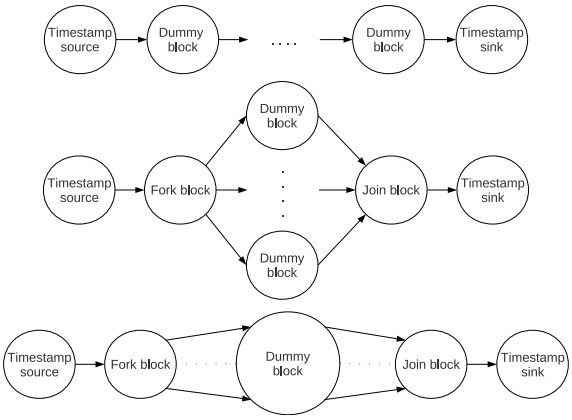

Figure 3. SDRs implemented for the overhead evaluation

A source block generates samples that contain timestamps of when they were generated. The *dummy blocks* are empty blocks that just propagate its inputs to its outputs. When the samples arrive at the sink block, the timestamps are compared with the current time, obtaining the time the sample took to go through the block chain. Since the blocks are empty, this time is the overhead imposed by the architecture to the data flow.

Figure 4 shows the results. Since absolute time values would be strongly related to the platform processing power, all the results were normalized in relation to the simplest case, where there is just one *dummy block* between the source and sink blocks. This way, more general results can be shown. The results show that, for all structures, the overhead grows linearly in relation to the increase of the number of blocks and the number of inputs and outputs. The standard deviation of the average overhead remained low compared to the mean values. This low standard deviation is especially important when implementing time-strict protocols, like TDMA based protocols. This is not the case of GNU Radio for example, where the standard deviation of the time a sample takes to get to the processing chain after being generated is higher then the average time [5].

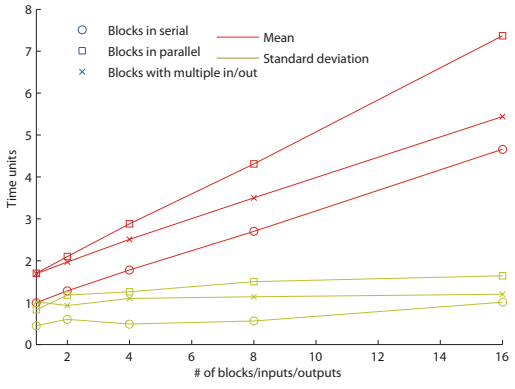

Figure 4. Mean and standard deviation of the architecture overhead

5. Conclusion

This paper presented an analysis of SDR implementation approaches and proposed a new architecture for SDR development. This new architecture enables SDR development using PLDs by directly mapping components of a high level model to implementable components. The results show that the architecture imposes a very deterministic overhead over the SDR processing chain.

References

1

2 GNU FSF project, "The GNU Radio," 2009, [Online; accessed 08-November-2009]. [Online]. Available: http://www.gnu.org/software/gnuradio

3 A. A. Fröhlich, "Application-Oriented Operating Systems," Ph.D. dissertation, Technical University of Berlin, Berlin, 2001.

4 H. Marcondes and A. A. Fröhlich, "On Hybrid Hw/Sw Components for Embedded System Design," in *Proceedings of the 17th IFAC World Congress*, Seoul, Korea, 2008, pp. 9290–9295.

5 G. Nychis, T. Hottelier, Z. Yang, S. Seshan, and P. Steenkiste, "Enabling MAC Protocols Implementation on Software-defined Radios," in *Networked Systems Design and Implementation*, 2009.

6 Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "SODA: A Low-power Architecture For Software Radio," in *ISCA '06: 33rd International Symposium on Computer Architecture*, 2006, pp. 89–101.

7 J. Glossner, E. Hokenek, and M. Moudgill, "The Sandbridge Sandblaster Communications Processor," in *3rd Workshop on Application Specific Processors*, 2004, pp. 53–58.

8 K. van Berkel, F. Heinle, P. P. E. Meuwissen, K. Moerman, and M. Weiss, "Vector processing as an enabler for software-defined radio in handheld devices," *EURASIP: Journal on Applied Signal Processing*, vol. 2005, pp. 2613–2625, 2005.

9 Steven Kelem, Brian Box, Stephen Wasson, Robert Plunkett, Joseph Hassoun, and Chris Phillips, "An Elemental Computing Architecture for SD Radio," in *SDR '07: 2007 Software Defined Radio Technical Conference*, 2007.