VIRTUALIZAÇÃO DE COMPONENTES DE HARDWARE PARA A MISSÃO VERITAS USANDO FERRAMENTAS OPEN-SOURCE

Gabriel Rodrigues Marques [1]; Rodrigo de Marca França [2]

[1] Aluno de Iniciação Científica do Instituto Mauá de Tecnologia (IMT);
[2] Professor do Instituto Mauá de Tecnologia (IMT).

Resumo. *Este trabalho apresenta o desenvolvimento de um sistema de injeção de falhas no barramento AMBA AHB integrado ao ambiente de virtualização QEMULA, visando a validação de mecanismos de tolerância a falhas em software embarcado aeroespacial. O sistema implementa seis tipos de falhas (timeout, paridade, SLVERR, DECERR, corrupção de dados e stall) por meio de um motor dedicado capaz de interceptar transações AHB e aplicar perturbações de forma determinística. A arquitetura inclui a extensão do Controlador de Simulação de Baixo Nível (LLSC), registradores adicionais no periférico AHBSTAT para logging estruturado e três ferramentas de teste: interface Python, scripts AFI e firmware bare-metal. Os resultados demonstram que o ambiente permite reproduzir cenários complexos de falhas com controle temporal preciso, possibilitando maior cobertura de testes e avaliação detalhada das rotinas de detecção e recuperação de erros no processador LEON3. A solução contribui para ampliar a confiabilidade do software de voo, oferecendo uma alternativa de baixo custo a métodos tradicionais de teste em hardware real.*

Introdução

A crescente complexidade dos sistemas embarcados aeroespaciais e a necessidade de garantir confiabilidade em ambientes hostis tornam a validação de mecanismos de tolerância a falhas um desafio crítico no desenvolvimento de software de voo. Sistemas espaciais estão sujeitos a fenômenos como *Single Event Upsets* (SEUs) causados por radiação cósmica, que podem corromper registradores, memórias e sinais de barramento (Normand, 1996). Em processadores como o LEON3, amplamente utilizados em aplicações espaciais devido à sua arquitetura SPARC V8 tolerante a falhas, a detecção e recuperação de erros depende de mecanismos integrados como códigos EDAC (*Error Detection and Correction*) e registradores de status como o *AHBSTAT* (Gaisler Research, 2022). Entretanto, testar adequadamente esses mecanismos em hardware real apresenta limitações significativas: provocar falhas controladas requer equipamentos especializados de teste por radiação ou injeção de falhas por hardware, os cenários são difíceis de reproduzir deterministicamente, e existe o risco de danos permanentes ao equipamento (Benso *et al*., 2003).

A virtualização de hardware emerge como solução promissora para essas limitações, permitindo testes exaustivos em ambiente simulado antes da integração com hardware físico. Trabalhos recentes demonstram a eficácia de emuladores como QEMU para validação de software embarcado crítico (Carvalho *et al*., 2020; Santos *et al*., 2025), incluindo aplicações aeroespaciais onde a combinação de simulação de alto nível com *Software-in-the-Loop* (SIL) permite reduzir custos e acelerar ciclos de desenvolvimento (Geletko *et al*., 2019).

Nesse contexto, estudos existentes na literatura exploram diferentes abordagens de injeção de falhas para avaliação da confiabilidade de sistemas embarcados, com ênfase em arquiteturas tolerantes a falhas. Abordagens clássicas baseadas em injeção física ou por perturbação elétrica, como as análises pioneiras de Arlat *et al*. (1990), demonstram alta fidelidade, porém apresentam elevado custo, baixa repetibilidade e risco operacional. Estudos mais recentes focam em plataformas de simulação, explorando técnicas de injeção de falhas baseadas em modelos comportamentais ou instrumentação de simuladores. Bekele, Limbrick e Kelly (2023) apresentam um levantamento abrangente de ferramentas de injeção baseadas em QEMU, destacando limitações como suporte limitado ao barramento AHB e ausência de mecanismos determinísticos de temporização. Outras iniciativas, como mcQEMU (Carvalho *et al*., 2020) e o NOS3 da NASA (Geletko *et al*., 2019), demonstram a viabilidade de simulação de plataformas complexas, mas não fornecem infraestrutura

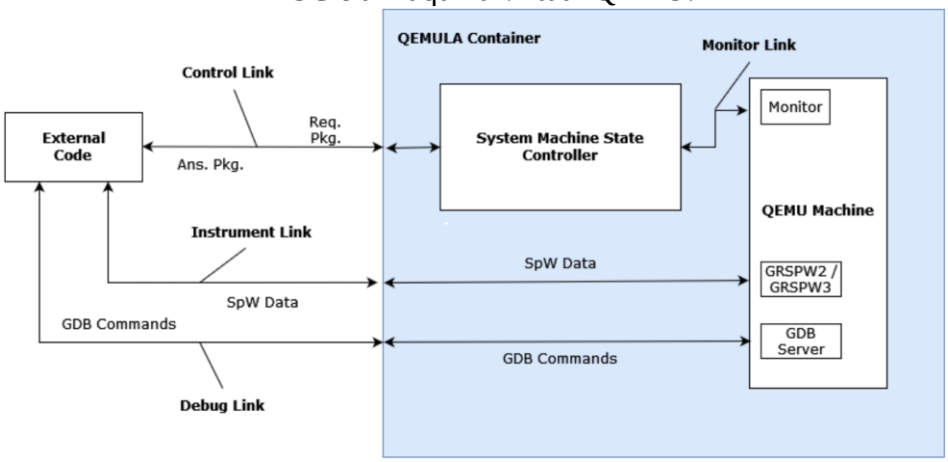
para injeção estruturada de falhas em nível de barramento. Em conjunto, esses trabalhos evidenciam a lacuna existente na literatura: embora existam ferramentas de simulação espacial e frameworks de injeção de falhas, não há soluções que integrem, em uma única plataforma, virtualização de hardware LEON3, injeção determinística no AMBA AHB (Advanced High-performance Bus) e *logging* detalhado de eventos, lacuna que este trabalho busca preencher.

No contexto do projeto QEMULA, um framework baseado no emulador QEMU desenvolvido para virtualização do instrumento VenSpec-M da missão VERITAS (Costa & França, 2024), identificou-se a necessidade de expandir as capacidades de teste para incluir injeção sistemática de falhas no barramento AMBA AHB, principal interconexão de alta performance em sistemas-on-chip baseados em LEON3.

Este trabalho apresenta o desenvolvimento de um sistema completo de injeção de falhas no barramento AHB integrado ao QEMULA, compreendendo: (i) um motor de injeção de falhas (*AHB Fault Engine*) implementado como dispositivo *QEMU Object Model* (QOM) capaz de interceptar transações de barramento e aplicar perturbações controladas; (ii) extensão do Controlador de Simulação de Baixo Nível (LLSC - *Low-Level Simulation Controller*) para processar comandos de injeção via protocolo binário; (iii) ampliação do periférico *AHBSTAT* com registradores estendidos para *logging* de eventos em FIFO; (iv) ferramentas de teste em três níveis: interface Python para controle interativo, *parser* de *scripts* AFI (AHB *Fault Injection*) para cenários automatizados, e programa *bare-metal* para validação no processador virtualizado; e (v) mecanismos de sincronização temporal baseados no relógio virtual do QEMU para garantir determinismo nas injeções.

A Figura 1 apresenta uma visão geral do ambiente QEMULA, destacando o fluxo de comunicação entre o código externo, o *System Machine State Controller* e a máquina virtual QEMU. Nessa arquitetura, o *Control Link* desempenha papel central ao encaminhar os comandos destinados ao LLSC, responsável por executar operações de controle em nível baixo, incluindo as injeções de falhas propostas neste trabalho.

Figura 1 – Visão geral do QEMULA, destacando o fluxo de comunicação entre o código externo, o LLSC e a máquina virtual QEMU.



Além de implementar a arquitetura proposta, este trabalho introduz um conjunto de contribuições relevantes para a validação de sistemas embarcados críticos. Entre elas, destaca-se o modelo estruturado de falhas no barramento AHB, que abrange seis categorias e permite reproduzir cenários realistas de erro. A arquitetura de injeção determinística desenvolvida, baseada na interceptação de transações e na aplicação de filtros múltiplos, garante precisão temporal e reprodutibilidade, enquanto a integração entre o LLSC, o AHB *Fault Engine* e o *AHBSTAT* formam uma cadeia unificada de injeção, captura e registro de eventos. Complementando essa infraestrutura, foi definido um protocolo binário otimizado para controle remoto da simulação, e desenvolveu-se um conjunto de ferramentas de validação em diferentes níveis, incluindo a interface Python, os *scripts* AFI e o *firmware bare-metal*. A aplicação prática no projeto VenSpec-M demonstra a viabilidade da abordagem em um contexto aeroespacial real.

Essas contribuições beneficiam não apenas sistemas espaciais, mas qualquer aplicação embarcada crítica baseada em barramentos AMBA, incluindo setores automotivo (ISO 26262), aviônico (DO-178C) e médico (IEC 62304). A motivação central é fornecer aos desenvolvedores um ambiente que permita validar *handlers* de exceção, rotinas de recuperação de erros e algoritmos de redundância em cenários que seriam extremamente custosos ou inviáveis de reproduzir em hardware real. Situações como sequências de múltiplos erros de paridade com intervalos de microssegundos, por exemplo, tornam-se triviais no ambiente virtualizado e podem ser repetidas quantas vezes necessário, o que facilita depuração, análise e verificação de requisitos de V&V para sistemas críticos (European Cooperation for Space Standardization, 2009).

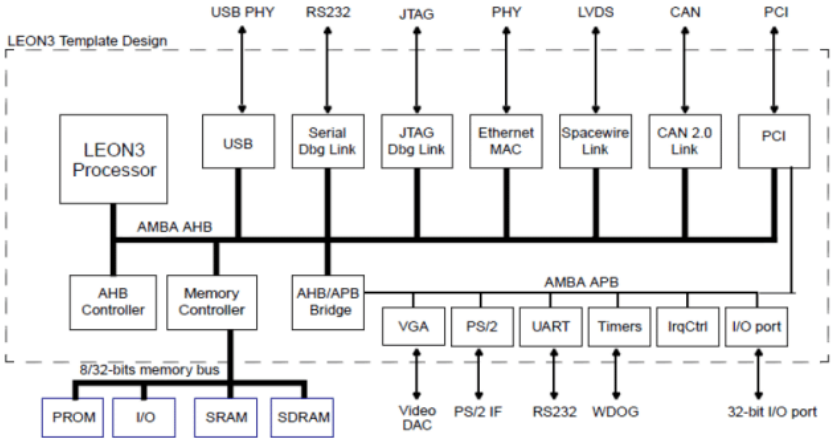
A organização deste trabalho segue a seguinte estrutura: a seção Materiais e Métodos descreve o modelo de falhas, a integração com o QEMU via QOM, o protocolo LLSC, o sistema de *logging* estendido do *AHBSTAT* e as ferramentas utilizadas na validação. A seção Resultados e Discussão apresenta os casos de uso avaliados, a análise de determinismo temporal, o impacto de desempenho e a comparação com abordagens alternativas. Por fim, as Conclusões resumem as contribuições e apresentam perspectivas para trabalhos futuros.

Material e Métodos

Nesta seção são apresentados os principais componentes e procedimentos utilizados no desenvolvimento do sistema de injeção de falhas, incluindo o modelo de falhas adotado para o barramento AHB, a arquitetura do AHB *Fault Engine*, as extensões realizadas no periférico *AHBSTAT* e as ferramentas empregadas para validação dos cenários de teste.

A Figura 2 apresenta a arquitetura de referência do LEON3, destacando os barramentos AMBA e seus componentes principais. Essa visão global facilita compreender o impacto que perturbações no AHB podem causar no fluxo de execução do sistema, justificando a necessidade de mecanismos robustos de injeção e monitoramento de falhas.

Figura 2 – Arquitetura *top-level* do processador LEON3, destacando a organização dos barramentos AMBA AHB e APB, seus principais controladores e periféricos.



Modelo de Falhas e Sistemática de Erros no Barramento AHB

O barramento AMBA AHB (ARM, 2001) define um protocolo de comunicação de alta performance entre mestres (como CPU e controladores DMA) e escravos (memórias e periféricos) em sistemas-on-chip. Cada transação envolve (*HREADY*, *HRESP*, *HSIZE*, *HWRITE*), endereçamento (*HADDR*), dados (*HWDATA*/*HRDATA*) e metadados (*HMASTER*), e diferentes tipos de falhas podem ocorrer, exigindo mecanismos específicos de detecção e recuperação no software.

Neste trabalho adotou-se um modelo composto por seis categorias de falhas, inspirado em Arlat *et al*. (1990) e adaptado ao contexto AMBA:

1. **BUS_TIMEOUT (código 0x01):** transação que excede o tempo máximo permitido, simulada mantendo *HREADY* inativo até gerar *timeout* e registro no *AHBSTAT*;
2. **PARITY_ERR (código 0x02):** paridade incorreta em sinais de dados ou controle, simulada com resposta *HRESP*=ERROR;
3. **SLVERR (código 0x03):** *erro interno do escravo ou acesso inválido, sendo a resposta de erro mais comum no protocolo*;
4. **DECERR (código 0x04):** *endereço não reconhecido por nenhum escravo, útil para testar acessos fora do mapa de memória*;
5. **DATA_CORRUPT (código 0x05):** corrupção de dados em trânsito, implementada via operação XOR com padrão configurável;
6. **STALL (código 0x06):** *atraso artificial prolongado mantendo HREADY=0, simulando contenção de barramento ou latências extremas*.

Cada falha é configurada por um conjunto de parâmetros (endereço, máscara, mestre, direção e tamanho de acesso), permitindo selecionar precisamente quais transações serão afetadas. Essa granularidade viabiliza a reprodução de cenários específicos observados em testes reais ou análises de falhas.

Arquitetura do Motor de Injeção de Falhas (AHB *Fault Engine*)

O AHB *Fault Engine*, implementado como dispositivo QOM do tipo TYPE_AHB_FAULT_ENGINE, atua como o componente central do sistema de injeção de falhas. Ele mantém uma estrutura interna capaz de armazenar até 32 falhas ativas simultaneamente, além de contadores de injeção, *timers* associados a falhas temporárias, um *script* AFI previamente carregado e ponteiros de *callback* para notificação direta do *AHBSTAT*. Essa organização garante que o motor possa lidar tanto com eventos pontuais quanto com cenários complexos compostos por múltiplas injeções distribuídas no tempo virtual do QEMU.

A função de interceptação (*ahb_fault_engine_intercept*) é a principal responsável por aplicar perturbações ao barramento. Antes de cada transação AHB relevante, essa função compara os parâmetros do acesso, endereço, mestre, direção e tamanho, com os critérios definidos em cada falha ativa. Quando há correspondência, o mecanismo aplica o efeito apropriado: no caso de DATA_CORRUPT, os dados são modificados via XOR; em STALL e BUS_TIMEOUT, o comportamento de espera é simulado pela manipulação de *HREADY*; e nos casos de SLVERR, DECERR ou PARITY_ERR, são aplicados sinais de erro adequados. Para falhas configuradas como de disparo único, o contador associado é decrementado e a entrada é automaticamente removida após ser aplicada. Esse mecanismo possibilita reproduzir tanto falhas simples quanto eventos raros e altamente específicos que ocorrem em condições reais de operação.

O motor também é responsável por processar todos os comandos vindos do LLSC, interpretados por meio do protocolo binário. Três tipos principais de instruções são suportados: INJECT_ONCE (0x10), utilizado para inserir falhas que serão aplicadas apenas na próxima transação correspondente; INJECT_TOGGLE (0x11), empregado para ativar ou desativar falhas persistentes; e CLEAR_FAULTS (0x12), que remove todas as falhas ativas e reinicia o mecanismo. Após o processamento, o motor envia ao LLSC uma resposta contendo o status da operação, a quantidade atual de falhas ativas e o número de eventos já registrados pelo *AHBSTAT*, permitindo ao cliente confirmar rapidamente se o comando foi corretamente interpretado.

Para permitir cenários mais elaborados, o AHB *Fault Engine* suporta o carregamento de *scripts* AFI. Esses arquivos em formato texto definem eventos temporais de injeção no formato *<timestamp_us> <fault_type> <address> <HMASTER> <HWRITE> <HSIZE> <duration_us>*. A Tabela 1 apresenta um exemplo de *script* AFI com seus parâmetros e descrições correspondentes. Durante o carregamento, o *parser* converte *timestamps* para o tempo virtual do QEMU, valida cada tipo de falha e armazena até 256 eventos na estrutura interna. Esse mecanismo permite que testes complexos, como *bursts* de falhas encadeadas ou sequências temporizadas, sejam executados sem intervenção manual, garantindo precisão e reprodutibilidade.

Tabela 1 – Exemplo de *script* AFI

Parâmetro	Valor Atribuído	Descrição Técnica
timestamp_us	1000	Evento ocorre em 1 ms
fault_type	3 – SLVERR	Erro de slave durante escritura
address	0x40000000	Endereço de teste na RAM
HWRITE	1	Operação de escrita
HSIZE	2 – word (32 bits)	Acesso de 32 bits
duration_us	100	Duração do erro
Detalhamento	SLVERR em write após 1 ms	—

Por fim, o motor possui um mecanismo de expiração temporal responsável por remover automaticamente falhas com duração limitada. Esse mecanismo, implementado como um *callback* periódico de *QEMUTimer*, verifica se cada falha temporária já excedeu o tempo virtual configurado. Quando isso ocorre, a falha é removida do *array* ativo, e o timer é reagendado apenas caso ainda existem falhas temporárias pendentes. Essa funcionalidade é essencial para *scripts* AFI que dependem de sequências complexas de eventos distribuídos no tempo.

Integração com o Periférico *AHBSTAT* e *Logging* de Eventos

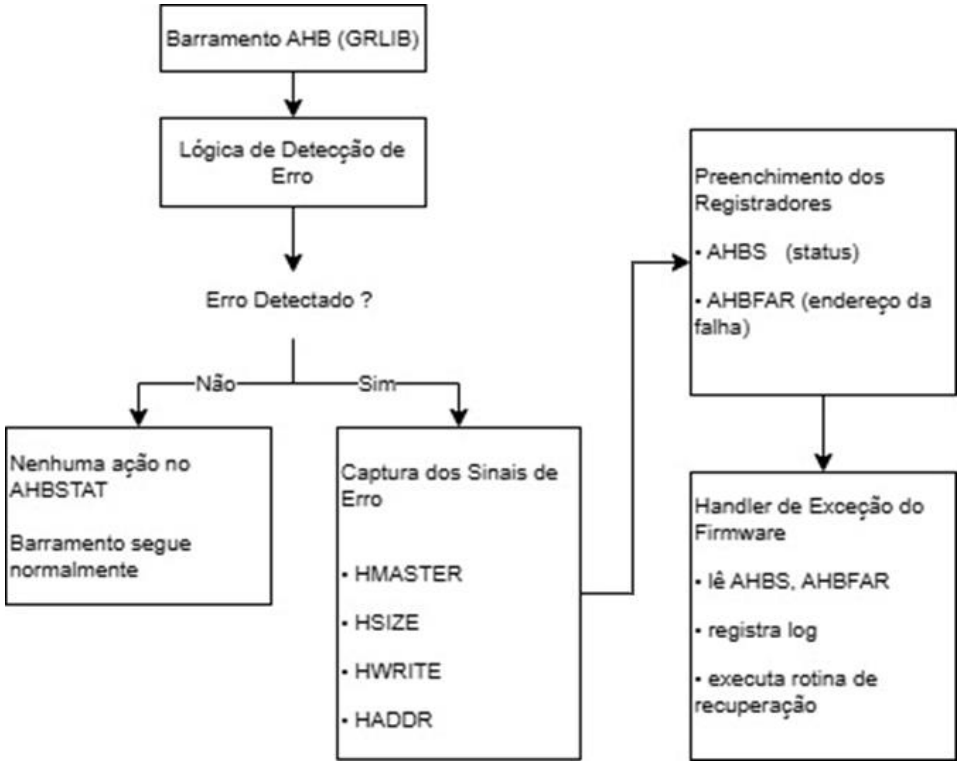
A GRLIB IP *Library*, desenvolvida pela Frontgrade Gaisler, fornece um conjunto modular de núcleos sintetizáveis para sistemas embarcados baseados nos processadores LEON2/LEON3, estruturados segundo a arquitetura AMBA. Nesse ecossistema, o periférico *AHBSTAT* exerce a função de monitorar erros no barramento AHB, oferecendo uma interface padronizada para detecção e diagnóstico de falhas. No projeto original, o módulo dispõe de dois registradores principais, *AHBS* e *AHBFAR*, responsáveis por capturar o status do erro e o endereço associado. Sempre que uma falha for detectada, o *AHBSTAT* registra os sinais relevantes da transação, como *HMASTER*, *HSIZE* e *HWRITE*, e aciona uma interrupção para o processador, permitindo que o *firmware* trate o evento de forma imediata.

Para este trabalho, o *AHBSTAT* foi ampliado a fim de suportar *logging* estruturado e captura histórica de eventos, integrando-se diretamente ao mecanismo de injeção de falhas do QEMULA. A Figura 3 ilustra a organização interna do periférico e sua interação com os sinais do barramento, destacando como erros são coletados, armazenados e encaminhados ao processador. Essa visualização reforça o papel do *AHBSTAT* como ponto central de diagnóstico dentro do ambiente virtualizado.

A extensão implementada adiciona quatro novos registradores mapeados entre os offsets 0x10 e 0x1C. O primeiro deles, *AHBSTAT*_EVENT_COUNT, é um contador global que incrementa a cada evento registrado e permite identificar perda de entradas quando a FIFO atinge sua capacidade. O registrador *AHBSTAT*_STATUS_BITMAP mantém uma máscara indicando quais tipos de falhas já ocorreram, facilitando verificações rápidas pelo software. O registrador *AHBSTAT*_EVENT_LOG_CTL controla a FIFO circular de 64 entradas, permitindo habilitar ou limpar o log e consultar seu estado (cheia ou vazia). Já *AHBSTAT*_EVENT_LOG_DATA fornece acesso sequencial às entradas da FIFO, contendo endereço, tipo de falha, parâmetros da transação e *timestamp* relativo em microssegundos.

Internamente, o módulo utiliza índices de leitura e escrita para gerenciar a FIFO de forma circular, além de manter um *timestamp* base para cálculo do tempo relativo dos eventos. A adição de novas entradas é realizada pela função *Ahbstat_log_event*, acionada diretamente pelo *callback* registrado no AHB *Fault Engine* sempre que uma falha é aplicada. Essa abordagem permite análise *post-mortem* detalhada: após a execução de um conjunto de testes, o *firmware* pode ler toda a FIFO para verificar ordem, temporalidade e consistência dos eventos registrados, auxiliando no diagnóstico de condições de corrida e na validação de sequências complexas de falhas.

Figura 3 – Visão funcional do periférico *AHBSTAT*, destacando a captura de erros no barramento AHB e o encaminhamento dessas informações ao processador.



Integração com o QEMULA e Protocolo de Comunicação LLSC

O sistema foi projetado para integração não invasiva com o QEMULA desenvolvido por Costa & França (2024), preservando compatibilidade com toda a infraestrutura existente. O AHB *Fault Engine* é implementado como um dispositivo QOM independente, instanciado em *leon3.c* via sysbus e conectado ao barramento AHB virtual por meio de propriedades padrão do QOM, permitindo sua incorporação sem alterações estruturais na plataforma. Durante a inicialização, o *AHBSTAT* registra um *callback* no *Engine*, recebendo notificações imediatas de eventos injetados sem necessidade de polling e estabelecendo uma cadeia eficiente de comunicação entre monitoramento e injeção.

No lado do controle externo, o LLSC foi estendido com novos comandos adicionados ao switch-case de llsc_read, mantendo compatibilidade com os mecanismos pré-existentes do QEMULA. A arquitetura também utiliza um *message bus* interno que possibilita comunicação assíncrona entre dispositivos simulados, facilitando futuras extensões. Por exemplo, permitindo que um controlador EDAC reporte erros corrigidos diretamente ao *AHBSTAT* ou que periféricos adicionais, como o GRSPW2 para SpaceWire, sejam incorporados ao framework sem reestruturações globais.

O protocolo binário do LLSC suporta esses novos comandos de injeção de falhas preservando o mesmo formato utilizado pelas operações originais do QEMULA. A comunicação ocorre via TCP/IP, com o LLSC atuando como servidor (tipicamente na porta 5055). Em um fluxo típico, o cliente (*fault_cli.py*) conecta-se ao QEMU, empacota um comando de 16 bytes e o envia pela rede. O LLSC recebe o pacote no *callback llsc_read*, decodifica a requisição e aciona *ahb_fault_engine_process_command*, que adiciona ou remove falhas conforme configurado. Em seguida, uma resposta compacta de 8 bytes é enviada ao cliente, permitindo validar o estado da operação. Para comandos de leitura do *AHBSTAT* (0x20 e 0x21), o protocolo suporta respostas maiores contendo os registradores estendidos e, quando necessário, o *dump* completo da FIFO de eventos, possibilitando coleta detalhada para análise e depuração.

Ferramentas de Validação e Teste

Para validar o sistema proposto, foram desenvolvidas três ferramentas complementares que operam em diferentes níveis de abstração e permitem avaliar desde o comportamento interno do mecanismo de injeção até a resposta completa do software embarcado.

A interface Python (*fault_cli.py*) fornece uma CLI (*command line interface*) para envio direto de comandos ao LLSC, possibilitando acionar injeções únicas ou persistentes, limpar estados internos e consultar o *AHBSTAT*. Por ser interativa e oferecer retorno imediato, essa ferramenta é especialmente útil durante experimentação e depuração. Um exemplo de utilização típica pode ser observado na Figura 4, que ilustra a execução de sequência de comandos e a consulta aos registradores estendidos.

Figura 4 – Exemplo de uso da interface Python *fault_cli.py*

```
python fault_cli.py --inject-once --fault-type 3 --address 0x40000000 --duration 100
python fault_cli.py --demo
```

O *parser* AFI constitui a segunda ferramenta e é integrado diretamente ao motor de falhas, interpretando *scripts* que definem eventos temporais de injeção com parâmetros configuráveis. Esse formato permite compor cenários extensos e totalmente determinísticos, adequados para campanhas automatizadas e validações que exigem sincronização temporal precisa. A Tabela 2 apresenta um exemplo de *script* AFI contendo diferentes cenários de teste, demonstrando a flexibilidade da ferramenta e a clareza do formato empregado. A simplicidade dos arquivos, com suporte a comentários e centenas de eventos, facilita documentação, reprodutibilidade e integração com *pipelines* de CI/CD.

Tabela 2 – *Script* AFI com quatro diferentes cenários de teste

timestamp_us	*fault_type*	*address*	*HMASTER*	*HWRITE*	*HSIZE*	*duration_us*	Descrição
1000	3	0x80000100	255	1	2	0	# Cenário 1: SLVERR em região de periférico
2500	5	0x40000000	255	0	2	0	# Cenário 2: DATA_CORRUPT durante leitura de memória
5000	1	0x90000000	3	1	2	50	# Cenário 3: *Burst* de *timeouts* em DMA
5100	1	0x90000004	3	1	2	50	# Cenário 3: *Burst* de *timeouts* em DMA
5200	1	0x90000008	3	1	2	50	# Cenário 3: *Burst* de *timeouts* em DMA
10000	6	0xFFFFFFFF	255	255	255	1000	# Cenário 4: STALL prolongado

Complementando essas abordagens, o *firmware bare-metal ahb_fault_tester.c* executa diretamente no LEON3 virtualizado, permitindo observar o comportamento do software embarcado frente às falhas injetadas. Ele implementa cinco testes principais: (i) *bursts* de escrita sem injeção, verificando ausência de erros espúrios; (ii) *bursts* de leitura com checagem de integridade; (iii) validação coordenada de SLVERR injetado manualmente via CLI; (iv) execução de múltiplos acessos após o carregamento de um *script* AFI, para confirmar registro e ordenação de eventos; e (v) leitura

dos registradores estendidos do *AHBSTAT*, certificando sua acessibilidade e consistência. Esses testes exercitam o caminho completo de CPU para barramento e então periféricos, refletindo o comportamento esperado em um ambiente embarcado real.

A Tabela 3 resume as características das três ferramentas, destacando seus pontos fortes, limitações e cenários de aplicação ideais. Em conjunto, elas oferecem uma infraestrutura abrangente de validação, cobrindo desde testes exploratórios até execuções determinísticas e orientadas ao comportamento real do *firmware*.

Tabela 3 – Comparativo entre as Ferramentas de Validação e Teste

Ferramenta	Pontos Fortes	Limitações	Melhor Aplicação
Interface Python (*fault_cli.py*)	Testes rápidos e exploratórios; feedback imediato no console; fácil de estender e depurar; suporte interativo.	Não garante determinismo em cenários longos; depende do tempo real do host; não cobre ciclo completo do *firmware*	Depuração manual, demonstrações, execução de falhas isoladas, verificação rápida de comportamento
Parser AFI (*scripts*.afi)	Determinismo temporal total (QEMU_CLOCK_VIRTUAL) ; suporte a cenários complexos com centenas de eventos; excelente reprodutibilidade; documenta cenários de forma legível.	Exige preparar arquivos antecipadamente; menos adequado para testes interativos; debug de tempo real mais difícil	Campanhas estruturadas, testes repetíveis, *fault injection* sistemática, validação de sequências complexas
Programa *Bare-Metal* (*ahb_fault_tester.c*)	Determinismo temporal total (QEMU_CLOCK_VIRTUAL) Exercita o caminho completo HW/SW; captura comportamento real do *firmware*; mede respostas a interrupções, *handlers* e *recovery*; usa periférico real (UART, registradores, etc.).	Evolução mais lenta: requer recompilação; depuração mais trabalhosa; visualização limitada ao console UART	Validação final, análise de integridade de *handlers*, testes funcionais completos, certificação de software

Resultados e Discussão

A validação funcional do sistema foi realizada por meio de testes cobrindo os seis tipos de falhas implementados, variando parâmetros de filtragem, persistência e duração. No primeiro cenário, avaliou-se a injeção de SLVERR durante uma escrita no endereço 0x80000100 (UART0). O LEON3 capturou corretamente a exceção de barramento e o *AHBSTAT* registrou o evento com os metadados esperados, confirmando o comportamento do mecanismo.

A simulação de corrupção de dados foi verificada com um teste de DATA_CORRUPT, no qual o *firmware* escreveu o padrão 0xDEADBEEF em 0x40001000 e realizou uma leitura subsequente. Com a falha injetada, o valor retornado foi alterado para 0x00000000, sendo identificado pelo algoritmo de checagem, demonstrando que o mecanismo modifica corretamente dados em trânsito.

Também foi avaliada a capacidade do sistema de lidar com falhas concorrentes. Dez falhas configuradas em endereços consecutivos foram todas aplicadas durante um loop de escritas, resultando em EVENT_COUNT = 10 e com a FIFO registrando os eventos na ordem correta, o que confirma a robustez do motor em cenários paralelos.

A persistência foi validada ao ativar uma falha global de STALL, que afetou 100 acessos consecutivos, todos corretamente sinalizados no *AHBSTAT*. Após desativar a falha com

INJECT_TOGGLE, os acessos voltaram ao comportamento normal, comprovando o funcionamento do modo persistente.

Por fim, o mecanismo de expiração temporal foi testado configurando BUS_TIMEOUT com duração de 500 μs. A remoção automática da falha após o tempo virtual configurado, monitorado via *llsc_read_time*, confirmou o funcionamento do timer e sua integração com o relógio virtual do QEMU.

Embora o sistema apresentado ofereça uma infraestrutura robusta para injeção de falhas, algumas limitações permanecem. A modelagem atual do barramento não inclui arbitragem completa entre múltiplos mestres, restringindo a análise de contenção complexa. O controlador de memória também não implementa códigos EDAC reais, limitando a simulação de SEUs em nível de célula. Além disso, o modelo temporal do QEMU não reflete ciclos de *clock* individuais, o que reduz a fidelidade para aplicações altamente dependentes de timing. Por fim, o conjunto de falhas implementadas cobre os cenários mais comuns, mas não inclui perturbações de baixo nível em sinais analógicos ou registradores internos.

Como trabalhos futuros, pretende-se aprimorar a modelagem de EDAC, incorporar um modelo mais completo de arbitragem AMBA, expandir a injeção de falhas para outros periféricos (como SpaceWire e interfaces seriais) e integrar ferramentas de análise de cobertura para avaliação automática de rotinas de tratamento de erros. Adicionalmente, planeja-se explorar geradores automáticos de cenários AFI baseados em modelos estatísticos de radiação e desenvolver interfaces gráficas que facilitem a visualização temporal de eventos e o depuramento de cenários complexos.

Conclusões

Este trabalho apresentou desenvolvimento, implementação e validação de sistema robusto de injeção de falhas no barramento AHB integrado ao ambiente de virtualização QEMULA. A solução proposta aborda limitação crítica de ambientes de desenvolvimento de software embarcado: dificuldade em testar mecanismos de tolerância a falhas sem acesso a equipamentos especializados de teste por radiação ou injeção de falhas por hardware.

Através da combinação de emulação em nível de registradores no QEMU, extensão do Controlador de Simulação de Baixo Nível (LLSC), e desenvolvimento de ferramentas de teste complementares, demonstrou-se ser possível reproduzir de forma controlada e determinística ampla gama de cenários de falhas em sistemas baseados em processador LEON3 e barramento AMBA AHB. Os seis tipos de falhas implementados (BUS_TIMEOUT, PARITY_ERR, SLVERR, DECERR, DATA_CORRUPT, STALL) cobrem maioria dos erros transientes encontrados em sistemas espaciais sujeitos a radiação, permitindo validação exaustiva de *handlers* de exceção e algoritmos de recuperação.

Resultados de validação confirmam fidelidade funcional da emulação: software embarcado do VenSpec-M executado no ambiente virtualizado comporta-se de forma idêntica ao esperado em hardware real, incluindo captura de interrupções, leitura de registradores de status, e acionamento de rotinas de recuperação. Determinismo temporal foi verificado através de múltiplas execuções de mesmos cenários, produzindo resultados idênticos independente de carga do sistema host.

Contribuições principais incluem a arquitetura modular e extensível do motor de injeção, o mecanismo de *logging* estruturado via FIFO no *AHBSTAT* estendido, o protocolo binário eficiente para controle remoto, e o conjunto de ferramentas cobrindo desde exploração interativa até automação de testes. O sistema permite criar cenários de teste que seriam impraticáveis, arriscados ou impossíveis em hardware real, como cascatas de múltiplas falhas com timing preciso ou erros durante rotinas críticas de interrupção.

Limitações identificadas relacionam-se principalmente a aspectos de modelagem de baixo nível (arbitragem de barramento, cálculo real de EDAC, timing em nível de ciclo) que, embora importantes para certos contextos, não impedem uso efetivo da ferramenta para validação de software, objetivo principal do trabalho. Trabalhos futuros incluem aprimoramento da fidelidade de EDAC, expansão para outros periféricos (SpaceWire, interfaces seriais), integração com ferramentas de

análise de cobertura, e desenvolvimento de geradores automáticos de cenários baseados em modelos estatísticos de radiação espacial.

Trabalhos futuros incluem: (i) Integração com ferramentas de análise de cobertura de código (ex: gcov) para verificar automaticamente quais *branches* de *error handling* foram exercitados; (ii) Desenvolvimento de gerador automático de *scripts* AFI baseado em modelos de falhas estatísticos derivados de dados de radiação espacial (utilizando distribuições de Poisson para SEUs); (iii) Implementação de injeção de falhas em outros periféricos (SpaceWire, I2C, SPI); (iv) Criação de interface gráfica para visualização temporal de eventos de falha, facilitando debug de cenários complexos.

Em contexto mais amplo, este trabalho contribui para tendência crescente de uso de gêmeos digitais e virtualização em desenvolvimento de sistemas espaciais, alinhado com paradigma *New Space* de redução de custos e aceleração de ciclos de desenvolvimento. Disponibilização de ferramentas *open-source* baseadas em QEMU democratiza acesso a capacidades de teste avançadas, anteriormente restritas a laboratórios especializados. Espera-se que metodologia apresentada seja adotada não apenas em projetos aeroespaciais, mas em qualquer domínio de sistemas embarcados críticos onde confiabilidade e robustez são requisitos essenciais.

Referências Bibliográficas

Arlat, J., Costes, A., Crouzet, Y., Laprie, J. C., & Powell, D. (1990). Fault injection and dependability evaluation of fault-tolerant systems. *IEEE Transactions on Computers*, 39(4), 504-516.

ARM Limited. (2001). AMBA Specification (Rev 2.0). ARM IHI 0011A.

Bekele, Yohannes B., Daniel B. Limbrick, and John C. Kelly. (2023). A survey of QEMU-based fault injection tools & techniques for emulating physical faults. *IEEE Access*, 11, 62662-62673.

Benso, A., Di Carlo, S., Di Natale, G., Prinetto, P., & Tagliaferri, L. (2003). A watchdog processor to detect data and control flow errors. *In Proceedings 9th IEEE On-Line Testing Symposium*, 144-148.

Carvalho, Humberto, Geoffrey Nelissen, and Pavel Zaykov. (2020). mcQEMU: Time-Accurate Simulation of Multi-core platforms using QEMU. *2020 23rd Euromicro Conference on Digital System Design (DSD)*. IEEE, 227-234.

Costa, Felipe Fazio, and França, Rodrigo Marca. (2024). Virtualização do Instrumento VenSpec-M da Missão VERITAS por meio de Emulação de Hardware com QEMU e Integração SpaceWire via TCP/IP. Relatório de Iniciação Científica – Instituto Mauá de Tecnologia.

European Cooperation for Space Standardization. (2009). ECSS-Q-ST-80C: Software product assurance. ESA Requirements and Standards Division.

Gaisler Research. (2022). GRLIB IP Core User's Manual. Frontgrade Gaisler AB, Version 2022.2.

Geletko, Dustin M., et al. (2019). NASA operational simulator for small satellites (NOS3): the STF-1 cubesat case study. *arXiv preprint arXiv:1901.07583*.

Herdt, V., Große, D., Le, H. M., & Drechsler, R. (2020). Extensible and configurable RISC-V based virtual prototype. *In Forum on specification & Design Languages (FDL)*, 1-8. IEEE.

IEEE Computer Society. (2010). IEEE Standard for Standard SystemC Analog/Mixed-Signal Extensions Language Reference Manual. IEEE Std 1666.1-2016.

Kanavouras, P., Lounis, S., Perez, M. A., & Serio, C. (2022). New Space Approach for Advancing Software-in-the-Loop Simulation Technologies. *In 2022 IEEE Aerospace Conference (AERO)*, 1-10. IEEE.

Khurana, S., & Hodges, A. (2020). Virtualization Enabling Technologies for Space System Development. *In AIAA Scitech 2020 Forum*, 0954.

Normand, E. (1996). Single event upset at ground level. *IEEE Transactions on Nuclear Science*, 43(6), 2742-2750.

Santos, Luiz Henrique Antoniassi, et al. (2025). A Virtualized Architecture for Software-in-the-Loop Testing Applied to the LEON3 Processor. *Journal of Aerospace Technology and Management*, 17, e2025.