PROVA DE CONCEITO DE UM SISTEMA HÍBRIDO PARA ATUALIZAÇÃO REMOTA DA CONFIGURAÇÃO (BITSTREAM) EM PLATAFORMAS EMBARCADAS BASEADAS EM FPGA

RELATÓRIO FINAL

Autor: Marcio Barbosa
Programa: EmbarcaTech | Data: 08/02/2026
Orientador: Prof. Dr. David Ciarlini C. Freitas | Mentor(a): Luana Moura

1. INTRODUÇÃO

A evolução dos sistemas computacionais modernos é caracterizada pela convergência entre a flexibilidade dos microcontroladores de propósito geral e o determinismo do hardware programável. No cenário da Internet das Coisas (IoT) e de sistemas de alta integridade, a implementação de atualizações de firmware e hardware de forma remota *(Over-the-Air)* constitui um requisito estratégico para a longevidade de dispositivos em campo, conforme discutido na proposta inicial deste trabalho (BARBOSA, 2025).

O presente projeto detalha a Prova de Conceito (PoC) de um sistema embarcado heterogêneo composto pelo microcontrolador RP2040 (Dual-Core ARM Cortex-M0+), atuando como host de coordenação, e pela FPGA Lattice ECP5, responsável pela execução de um ecossistema *System-on-Chip* (SoC) customizado. A arquitetura fundamenta-se na integração de um processador soft-core RISC-V (PicoRV32) a um IP Core original de comunicação SPI Slave, desenvolvido para mitigar os desafios de sincronismo e integridade de sinal inerentes a barramentos de alta velocidade em ambientes de prototipagem (BARBOSA, 2026).

Ao longo do ciclo de desenvolvimento, a pesquisa evoluiu de uma integração funcional básica para um estudo rigoroso sobre resiliência de hardware. Diante das limitações físicas identificadas na etapa de evidências (BARBOSA, 2026), adotaram-se metodologias para o tratamento de metastabilidade e técnicas de amostragem de sinais (*Delayed Sampling*), em conformidade com as recomendações de design digital de Harris & Harris (2012) e Sutherland (2017).

Este relatório final apresenta a fundamentação teórica, a análise da problemática física enfrentada e a implementação definitiva do firmware em linguagem C e do hardware em SystemVerilog. A validação sistêmica, realizada através da integração entre as plataformas BitDogLab e Colorlight i9, demonstra a viabilidade de arquiteturas abertas e robustas, alinhando-se aos requisitos técnicos e de inovação do programa EmbarcaTech.

2. OBJETIVOS

O presente trabalho tem como propósito a consolidação de uma arquitetura System-on-Chip (SoC) heterogênea, validando a integração entre hardware programável e software embarcado sob requisitos de alta resiliência. Conforme estabelecido na proposta inicial (BARBOSA, 2025), a Prova de Conceito

fundamenta-se na comunicação robusta entre um microcontrolador host (RP2040) e um processador soft-core RISC-V (PicoRV32), assegurando a integridade dos dados e a estabilidade sistêmica frente às restrições físicas da prototipagem em bancada. Pontos relevantes de integração de IP Core para armazenamento de dados provenientes de sensor de temperatura, estudo para a atualização de bitstream e interligação física das soluções em uma protoboard, foram as ações realizadas para a entrega deste relatório.

2.1. Objetivos Específicos

1. **Arquitetura de Hardware (HDL):** Projetar e implementar um IP Core original de comunicação SPI Slave em SystemVerilog, incorporando mecanismos de sincronização de múltiplos estágios para mitigação de metastabilidade e tratamento rigoroso de Clock Domain Crossing (CDC).
2. **Desenvolvimento de SoC**: Integrar o processador PicoRV32 a um barramento de dados interno com endereçamento mapeado em memória (Memory-Mapped I/O), permitindo que o firmware gerencie de forma eficiente os módulos de hardware, conforme a arquitetura base validada em etapa anterior (BARBOSA, 2026).
3. **Integridade de Sinal e Resiliência:** Validar a robustez da camada física através da aplicação de técnicas de Delayed Sampling e filtros de glitch, compensando distorções de sinal e "capacitâncias parasitas" inerentes ao ambiente de teste em protoboard.
4. **Integração de Software:** Desenvolver uma stack de software em linguagem C para o RISC-V, capacitada para processar pacotes via SPI e executar rotinas de controle e telemetria, atendendo aos requisitos técnicos de programação embarcada do programa.
5. **Validação Sistêmica:** Realizar testes de loopback e telemetria de hardware (contagem de bits e análise de duty cycle) para comprovar a eficácia da integração entre as plataformas BitDogLab (RP2040) e Colorlight i9 (ECP5), consolidando os resultados parciais documentados no Relatório de Evidência de Desenvolvimento (BARBOSA, 2026).

3. JUSTIFICATIVA

A crescente complexidade dos sistemas embarcados modernos exige arquiteturas que combinem a flexibilidade de software com o determinismo de hardware. No entanto, a integração de sistemas heterogêneos impõe desafios críticos de comunicação e integridade de dados que podem comprometer a confiabilidade de aplicações de missão crítica e o sucesso de atualizações remotas (Over-the-Air), conforme discutido na proposta inicial deste trabalho (BARBOSA, 2025).

A fundamentação técnica escolhida para este projeto e metodologias justifica-se pelos seguintes pilares:

- **Confiabilidade na Camada de Transporte (OTA):** O design de atualizações remotas exige uma análise rigorosa de trade-offs para garantir a segurança e a integridade do sistema (BROWN, 2018). Em ambientes de prototipagem, a ocorrência de ruídos e capacitâncias pode corromper o bitstream

durante a transmissão. Para mitigar tais riscos, a implementação de sincronizadores de múltiplos estágios e o tratamento de Clock Domain Crossing (CDC) fundamentam-se na literatura de Harris & Harris (2012), assegurando transições de sinais determinísticas entre o microcontrolador host e a FPGA;

- **Mitigação de Falhas Físicas e Diagnóstico:** A literatura de Sutherland (2017) enfatiza que sinais assíncronos em barramentos de alta velocidade são suscetíveis a disparos falsos se não houver um tratamento rigoroso de bordas. No presente projeto, o uso de técnicas de Delayed Sampling justifica-se pela necessidade de compensar distorções de sinal e estabilizar a amostragem de dados antes do processamento pela máquina de estados (FSM). Complementarmente, a estratégia de isolamento de falhas através de telemetria interna (contagem de bits) alinha-se às recomendações de Koopman (2010) sobre a importância da observabilidade em sistemas embarcados robustos;
- **Tecnológica e Eficiência:** A opção pelo processador PicoRV32 (RISC-V) justifica-se por sua maturidade e otimização para área em FPGAs comerciais, permitindo que a lógica programável supere a função de periférico passivo e assuma capacidades de processamento local (SHAH et al., 2019). Esta abordagem permite o gerenciamento inteligente do recebimento de dados e a integração de sensores de telemetria, conforme as evidências de desenvolvimento previamente validadas (BARBOSA, 2026);
- **Uso de Plataformas Padronizadas:** A utilização do microcontrolador RP2040 como coordenador do sistema provê uma plataforma eficiente que futuramente poderá oferecer conectividade, permitindo que a FPGA foque no processamento paralelo e na execução de hardware dedicado. Essa sinergia entre o ecossistema ARM e RISC-V cria um ambiente escalável para o desenvolvimento de soluções industriais e acadêmicas de alta performance.

4. PROBLEMÁTICA

O desenvolvimento de sistemas híbridos que integram microcontroladores e FPGAs enfrenta desafios inerentes à natureza distinta desses dispositivos e às condições de implementação física. No contexto desta prova de conceito, pesquisas foram necessárias para compreender o mau funcionamento dos testes realizados na protobord, estudos que possam compreender a problemática, que divide-se em quatro eixos técnicos críticos, identificados durante a fase de pesquisa e testes de funcionalidade:

A. **Sincronismo e Metastabilidade em Sistemas Heterogêneos:** Comunicação via protocolo SPI entre o RP2040 (Host) e a Lattice ECP5 (Slave) ocorre de forma assíncrona em relação ao clock interno da FPGA. Como o sinal de clock externo (SCK) não possui relação de fase com o clock de sistema da FPGA (25 MHz), a amostragem direta pode resultar em **metastabilidade**, onde o sinal não atinge um nível lógico estável em tempo hábil para o processamento, quando flip-flops falham em definir um sinal ‘0’ ou ‘1’ em tempo do clock. Conforme Harris & Harris (2012), a falha no tratamento do Clock Domain Crossing (CDC) acarreta comportamentos erráticos na Máquina de Estados (FSM), comprometendo a integridade da recepção do bitstream;

B. **Degradação de Sinal em Ambientes de Prototipagem:** Diferente de sistemas implementados em Placas de Circuito Impresso (PCB) otimizadas, a prototipagem em matriz de contatos (protoboard) introduz efeitos de capacitivos indesejado (capacitâncias parasitas) e indutâncias que degradam a qualidade dos sinais digitais. Durante a execução do projeto, identificou-se que o "arredondamento" das bordas do clock impede que o sinal de dados (MOSI) esteja estável no instante da amostragem (violação de Setup and Hold Time). Sutherland (2017) adverte que ruídos e glitches em sinais de controle podem gerar pulsos espúrios, resultando na perda de sincronismo da contagem de bits;

C. **Restrições de Recursos e Estabilidade de Baixo Nível:** A implementação de um SoC baseado no PicoRV32 em ambiente com memória RAM interna limitada (1KB) resultou em falhas críticas de Stack Overflow. A sobreposição do ponteiro de pilha (Stack Pointer) sobre a região de dados globais provocou travamentos e corrupção de variáveis durante o boot. De acordo com Koopman (2010), a ausência de mecanismos de observabilidade em sistemas bare-metal dificulta a distinção entre falhas de hardware e erros de software, exigindo uma arquitetura que ofereça telemetria em tempo real. O efeito desta problemática parece ser identificada neste projeto e portanto exploradas em próximas etapas;

D. **Integridade no Processo de Atualização Remota (OTA):** Enquanto o processo OTA é consolidado em microcontroladores, sua aplicação em FPGAs exige a substituição da configuração lógica completa (bitstream), e não apenas do código-fonte. A problemática reside em assegurar que o canal de transporte entre o RP2040 e a memória Flash externa da FPGA seja resiliente o suficiente para evitar a corrupção de dados durante a transmissão, sob o risco de tornar o dispositivo inoperante em campo (BARBOSA, 2025). Neste projeto tentativas de integração de troca de dados via SPI não sendo perfeitamente executadas inviabilizou a investigação nesta etapa;

5. ARQUITETURA DO SOC

A arquitetura do sistema seguiu um modelo de System-on-Chip (SoC) heterogêneo, fluxo de dados é gerenciado por um processador soft-core e os periféricos críticos são implementados como módulos de hardware dedicados (IP Cores). A organização do projeto reflete a segregação de domínios, com diretórios específicos para o suporte do microcontrolador host (01.RP2040), a descrição de hardware (02.FPGA) e o código-fonte do processador embarcado (03.Firmware).

Pastas estruturadas na bio do projeto.

```
├── 01.RP2040
├── 02.FPGA
├── 03.Firmware
├── 04.Docs
└── build
```

5.1. Descrição dos Módulos

Processador Soft-Core (PicoRV32): Implementado através do arquivo picorv32.v, o processador atua como a unidade central de processamento (CPU) do SoC. Foi selecionado por sua otimização para área, ocupando apenas 11% das LUTs da FPGA ECP5, conforme validado nos logs de síntese

IP Core Original (SPI Slave): Localizado em spi_slave.sv, este módulo é o responsável pela interface física com o RP2040. O código incorpora uma Máquina de Estados Finitos (FSM) de três estados e lógica de sincronização para mitigar problemas de metastabilidade em cruzamentos de domínios de clock (CDC).

Memória do Sistema e Bootloader: A infraestrutura de memória é composta pelo módulo soc_memory.v, que carrega o firmware.hex durante a síntese. O ciclo de inicialização é garantido pelo bootloader.s, responsável por configurar o Stack Pointer antes do desvio para o código em C (firmware.c).

Top Modules Especializados: A estrutura permite a validação de diferentes cenários de integridade através dos módulos top_oversampling.sv, top_filtro.sv e top_integridade.sv, garantindo a modularidade do design. Estes módulos foram implementados para assegurar testes de debug para o sincronismo, conforme destacado anteriormente.

5.2. Mapa de Memória (Memory Map)

A integração Hardware-Software é estabelecida através de um contrato de endereçamento fixo, definido no Linker Script (sections.lds) e manipulado pelo firmware através de ponteiros voláteis. Esta abordagem permite que o processador gerencie o hardware como extensões de memória.

Periférico	Endereço Base	Arquivo de Implementação	Função Técnica
Memória RAM	0x0000_0000	soc_memory.v	Armazenamento de código e dados do sistema.
IP Core SPI Slave	0x0001_0000	spi_slave.sv	Interface de comunicação com o microcontrolador host.
Registrador LEDs	0x0002_0000	top.sv	Saída de telemetria e depuração visual.

5.3. Fluxo de Compilação e Implementação

A robustez da arquitetura é suportada por um fluxo de desenvolvimento moderno (C++ / CMake / Python), permitindo a geração automática de binários para ambas as arquiteturas:

- **Host (RP2040):** Utiliza o Pico SDK e FreeRTOS para gerenciar as tarefas de comunicação e telemetria via USB-Serial (usb_task.c).
- **SoC (RISC-V):** Utiliza o RISC-V GNU Toolchain para compilar o firmware.c e gerar o arquivo de imagem de memória firmware.hex.
- **Hardware (FPGA):** O fluxo de síntese (Yosys) e roteamento (nextpnr-ecp5) consolida o bitstream **final (hardware.bit).**

6. DESENVOLVIMENTO DO SISTEMA

O desenvolvimento do sistema foi pautado pela modularidade e pelo isolamento de falhas, dividindo-se entre a lógica de hardware em SystemVerilog e o firmware em linguagem C. A estrutura do repositório reflete essa organização, com arquivos de topo especializados para cada cenário de teste de integridade. Outro detalhe para estruturar esta modularização, próximas etapas para evolução de cada ambiente poderá ser realizado independentemente, assegurando flexibilidade no desenvolvimento.

6.1. Implementação do IP Core Original (SPI Slave)

O módulo spi_slave.sv constitui o núcleo da interface de comunicação. Sua implementação foca na mitigação de problemas temporais através de:

- **Detecção de Borda Sincronizada:** O sinal de clock externo (SCK) é processado por um detector de borda (sck_rise) que utiliza dois flip-flops em cascata para evitar a metastabilidade;
- **Máquina de Estados (FSM):** O controle de fluxo é gerido por três estados (IDLE, RECEIVE, DONE), garantindo que a sinalização de dado válido (data_valid) ocorra apenas após a reconstrução completa do byte no registrador de deslocamento;
- **Filtro de Glitch e Oversampling:** Através do módulo top_oversampling.sv, implementou-se uma lógica de amostragem redundante para filtrar ruídos de alta frequência capturados na protoboard, garantindo que apenas transições válidas de sinal disparem a FSM.

6.2. Integração ao SoC RISC-V

A integração entre o hardware e o processador PicoRV32 ocorre no módulo top_soc.sv, onde o barramento de dados é estruturado via Memory-Mapped I/O.

- **Decodificação de Endereço:** Utiliza-se um multiplexador combinacional que monitora o prefixo do endereço de memória (mem_addr). Quando o acesso ocorre em 0x0001_XXXX, os dados do IP Core SPI são roteados para o barramento de leitura da CPU.
- **Sincronismo de Barramento:** O processador interage com o periférico através de polling no registrador de estado, onde o bit de validade sinaliza ao firmware a disponibilidade de novas informações para processamento. LEDs são acionados na protoboard.

6.3. Firmware e Lógica de Aplicação (C)

O código localizado em firmware.c gerencia o ciclo de vida da aplicação após a inicialização pelo bootloader.s.

- **Abstração de Hardware (HAL):** Foram implementados ponteiros voláteis para garantir acessos diretos aos registradores de hardware, evitando otimizações indesejadas do compilador;
- **Algoritmo de Telemetria:** O firmware executa um laço infinito que alterna entre o processamento de dados recebidos do RP2040 e a atualização de diagnósticos visuais nos LEDs.
- **Tratamento de Erros:** Foram desenvolvidas versões específicas do firmware (main_integridade.c, main_glitch.c) para validar a resiliência do sistema em diferentes níveis de interferência eletromagnética observada nos testes de bancada. Porém ao realizar o teste o main.c era chamado.

6.4. Ambiente do Microcontrolador Host (RP2040)

O código no diretório 01.RP2040 utiliza o FreeRTOS para coordenar as tarefas de sistema. A lógica implementada em src/main_integridade.c garante que o envio do bitstream e dos dados de telemetria respeite os tempos de estabilização exigidos pela FPGA, atuando como o mestre do barramento e garantindo o sincronismo da Prova de Conceito.

7. EVIDÊNCIAS DE FUNCIONAMENTO

A validação da Prova de Conceito foi estruturada em duas frentes complementares: a simulação lógica em nível de transferência de registros (RTL) e a verificação física em bancada. Este processo permitiu a identificação e mitigação das falhas de integridade de sinal discutidas anteriormente (BARBOSA, 2026).

7.1. Simulação Funcional (RTL)

Previamente à implementação física, o IP Core SPI Slave e a integração do SoC foram validados através de testbenchs desenvolvidos em SystemVerilog (sim/tb_spi_slave.sv).

Análise de Formas de Onda: Utilizando o simulador Icarus Verilog e o visualizador GTKWave (waveform.vcd), confirmou-se a eficácia do detector de borda sck_rise na conversão do clock assíncrono em pulsos síncronos internos.

Validação de Protocolo: As simulações demonstraram a reconstrução correta de bytes de teste (ex: 0xA5), com a sinalização precisa do bit data_valid após o oitavo ciclo de clock, comprovando a robustez da Máquina de Estados (FSM).

Cenários de Estresse: Foram realizados testes específicos de oversampling (tb_oversampling.sv) para garantir que o sistema ignorasse glitches de alta frequência, simulando o ruído observado em ambiente real.

7.2. Validação em Hardware (FPGA)

A implementação física foi realizada através da interconexão entre as plataformas BitDogLab (RP2040) e Colorlight i9 (ECP5), utilizando uma matriz de contatos para a interligação dos barramentos.

Logs de Síntese e Implementação: O fluxo de ferramentas Open Source confirmou a eficiência da arquitetura. O arquivo log_yosy, reporta uma ocupação de 11% (2734 LUTs), e o log_nextpn atesta uma frequência máxima de operação de 72.18 MHz, superando amplamente os 25 MHz nominais do sistema.

Telemetria em Tempo Real: Conforme demonstrado nos registros de execução a FPGA utilizou contadores internos para monitorar a integridade do pacote recebido. O terminal serial confirmou a recepção estável de dados, validando as técnicas de Delayed Sampling aplicadas para compensar a capacitância da protoboard.

Interação com Periféricos: A validação final incluiu o controle de LEDs via barramento do SoC e a leitura de sensores, comprovando que o firmware em C (firmware.c) opera em harmonia com os registradores de hardware mapeados.

7.3. Validação dos Requisitos do Programa

A seguir resume a conformidade do projeto com as exigências técnicas do programa EmbarcaTech:

- Soft-Core RISC-V - Integração do picorv32.v no módulo top_soc.sv.
- IP Core Original - Desenvolvimento do spi_slave.sv com tratamento de CDC.
- Uso de Sensores - Integração de telemetria de hardware e sensores via barramento.
- Programação em C - Firmware firmware.c funcional no RISC-V.
- Uso da BitDogLab - RP2040 atuando como Host de coordenação e OTA.

7.4. Demonstração Prática (Vídeo)

Como parte integrante da validação, o vídeo de demonstração (https://youtu.be/1LC8j3mSPU8) apresenta o sistema em operação, destacando a estabilização dos dados no console de depuração e a resposta imediata do hardware aos comandos enviados pelo microcontrolador host. O vídeo apresenta o protótipo funcional em operação real na bancada de testes.

8. CONCLUSÕES E PERSPECTIVAS FUTURAS

A execução desta Prova de Conceito permitiu validar a viabilidade técnica de um sistema embarcado heterogêneo operando sob condições de prototipagem reais. O projeto demonstrou que a integração entre o microcontrolador RP2040 e a FPGA Lattice ECP5 é possível, contudo exigências de uma análise rigorosa da integridade de sinal e do sincronismo de hardware.

8.1. Considerações Finais sobre o Desenvolvimento

A implementação do processador soft-core PicoRV32 mostrou-se uma escolha eficiente, ocupando apenas 11% dos recursos lógicos da FPGA, o que comprova a viabilidade de SoCs baseados em arquitetura aberta (RISC-V) para funções de gestão e telemetria. A principal contribuição técnica deste trabalho reside no desenvolvimento do IP Core SPI Slave original (BARBOSA, 2026).

A aplicação das metodologias de Delayed Sampling e filtros de Oversampling foi determinante para superar as capacitâncias e os ruídos de barramento identificados na etapa de problemática. Os resultados documentados, tanto em simulação (BARBOSA, 2026) quanto em hardware real (Vídeo), comprovam que a adoção de estratégias de resiliência fundamentadas na literatura clássica (HARRIS & HARRIS, 2012; SUTHERLAND, 2017) é essencial para garantir o determinismo em sistemas digitais complexos.

8.2. Limitações e Desafios Superados

Identificou-se que o ambiente de matriz de contatos (protoboard) atua como um limitador para a frequência de operação do barramento SPI. No entanto, o sistema de telemetria interna (contagem de bits) permitiu ao firmware em linguagem C monitorar e reportar o estado de integridade do sistema em tempo real, atendendo aos requisitos de observabilidade sugeridos por Koopman (2010). O ajuste fino do Stack Pointer e a gestão de memória RAM (1KB) garantiram a estabilidade do sistema bare-metal, prevenindo falhas de Stack Overflow.

8.3. Perspectivas Futuras

Como evolução natural deste projeto, vislumbram-se os seguintes desdobramentos:

- **Arquitetura física e análise de mitigação de falhas:** O desenvolvimento deve seguir com análise de outros parceiros do projeto, para debater e buscar formas de mitigar a ligação física e análise futura permitiria elevar a frequência do barramento SPI e reduzir a necessidade de filtros de oversampling, ou outras técnicas para assegurar funcionalidade;
- **Implementação do Ciclo OTA Persistente:** Concluir a integração para que o PicoRV32 gerencie a escrita do bitstream diretamente na memória Flash externa da FPGA (Cenário 1 do Pré-Projeto), permitindo atualizações de hardware 100% autônomas;
- Expansão da modularização da FPGA: Integrar funcionalidades para armazenamento de dados em memória e processamento, oferecer respostas pontuais de dados formatados para a RP2040, seguindo etapas de armazenamento, processamento e resposta.
- **Expansão da Telemetria:** Integrar novos sensores e protocolos na camada de transporte a FPGA.

Em suma, o projeto buscou cumpriu integralmente os requisitos do programa EmbarcaTech, consolidando conhecimentos em HDL, arquitetura de computadores e software embarcado, servindo como base sólida para futuras implementações e evolução desta solução.

9. REFERÊNCIAS

1. BARBOSA, Marcio. **Prova de Conceito de um Sistema Híbrido para Atualização Remota da Configuração (Bitstream) em Plataformas Embarcadas Baseadas em FPGA: Pré-Projeto**. Piracicaba: Programa EmbarcaTech (IFCE), 2025.

2. BARBOSA, Marcio. **Prova de Conceito de um Sistema Híbrido para Atualização Remota da Configuração (Bitstream) em Plataformas Embarcadas Baseadas em FPGA: Relatório de Evidência de Desenvolvimento (RED)**. Piracicaba: Programa EmbarcaTech (IFCE), 2026.

3. BROWN, Benjamin Bucklin. **Over-the-Air (OTA) Updates in Embedded Microcontroller Applications: Design Trade-Offs and Lessons Learned**. Analog Dialogue, v. 52, n. 11, p. 1-7, nov. 2018.

4. DUBEY, Rahul. **Introduction to Embedded System Design Using Field Programmable Gate Arrays**. London: Springer-Verlag, 2009. ISBN 978-1-84882-015-9.

5. EL JAOUHARI, Saad; BOUVET, Eric. **Toward a generic and secure bootloader for IoT device firmware OTA update**. In: INTERNATIONAL CONFERENCE ON INFORMATION NETWORKING (ICOIN), 36., 2022, Jeju. Anais [...]. [S.l.]: IEEE, 2022. p. 90-95.

6. FOOSN. **Fomu: An FPGA in your USB Port**. Documentação Técnica. Disponível em: https://foosn.com/products/fomu/. Acesso em: dez. 2025.

7. HARRIS, David Money; HARRIS, Sarah L. **Digital Design and Computer Architecture**. 2. ed. Waltham: Morgan Kaufmann, 2012.

8. KOOPMAN, Philip. **Better Embedded System Software**. Pittsburgh: Drumnadrochit Press, 2010.

9. LATTICE SEMICONDUCTOR. **ECP5 and ECP5-5G Family Data Sheet**. Disponível em: http://www.latticesemi.com. Acesso em: jan. 2026.

10. RASPBERRY PI FOUNDATION. **RP2040 Datasheet: A microcontroller by Raspberry Pi**. Disponível em: https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf. Acesso em: jan. 2026.

11. SHAH, D. et al. **Yosys+nextpnr: An Open Source Framework from Verilog to Bitstream for Commercial FPGAs**. In: IEEE ANNUAL INTERNATIONAL SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES (FCCM), 27., 2019, San Diego. Proceedings [...]. San Diego: IEEE, 2019.

12. SKYWATER TECHNOLOGY. **First Google-Sponsored MPW Shuttle Launched at SkyWater with 40 Open Source Community Submitted Designs**. 2021. Disponível em: https://www.skywatertechnology.com. Acesso em: dez. 2025.

13. SUTHERLAND, Stuart. **RTL Modeling with SystemVerilog for Simulation and Synthesis**. Tualatin: Sutherland HDL, Inc., 2017.

14. VIEL, F.; ZEFERINO, C. A. **A Module for Remote Reconfiguration of FPGAs in Satellites**. In: IBERCHIP WORKSHOP, 23., 2017, Mar del Plata. Anais [...]. [S.l.]: IEEE, 2017.

15. WOLF, Clifford. **PicoRV32 - A Size-Optimized RISC-V CPU**. Disponível em: https://github.com/YosysHQ/picorv32. Acesso em: dez. 2025.

10. APÊNDICES

Esta seção organiza os artefatos técnicos por modularização e competência, garantindo a o planejamento e a execução física descrita neste relatório.

10.1. Módulo 01: Host de Coordenação e Gateway (RP2040)

Arquivos desenvolvidos para gerenciar a comunicação SPI e as tarefas de diagnóstico.

- **Apêndice A.1:** 01.RP2040/src/main_integridade.c (Rotina de validação de pacotes e CRC).

```c
#include <stdio.h>

#include "pico/stdlib.h"

#include "hardware/spi.h"

#include "FreeRTOS.h"

#include "task.h"

#include "hardware/i2c.h"



// --- CONFIGURAÇÃO DE PINAGEM (BitDogLab) ---

#define SPI_PORT spi0

#define PIN_MISO 16

#define PIN_CS   17

#define PIN_SCK  18

#define PIN_MOSI 19

#define LED_PIN_RP2040 11



// Função de leitura simulada (Padrão 25°C - 35°C)

uint8_t read_aht10_temp() {
```

```
    static uint8_t t = 25;
    t++;
    if(t > 35) t = 25;
    return t;
}

void hardware_spi_init() {
    spi_init(SPI_PORT, 10 * 1000); // 10kHz para máxima estabilidade inicial
    gpio_set_function(PIN_MISO, GPIO_FUNC_SPI);
    gpio_set_function(PIN_SCK,  GPIO_FUNC_SPI);
    gpio_set_function(PIN_MOSI, GPIO_FUNC_SPI);
    gpio_init(PIN_CS);
    gpio_set_dir(PIN_CS, GPIO_OUT);
    gpio_put(PIN_CS, 1);
}

// --- TAREFA: AUDITORIA COM VERIFICAÇÃO DE RETORNO ---
void vSensorFpgaTask(void *pvParameters) {
    hardware_spi_init();
    uint8_t dado_recebido = 0;

    printf("\n--- TESTE 1: AUDITORIA DE INTEGRIDADE + LOOPBACK ---\n");

    for(;;) {
        uint8_t temp_enviada = read_aht10_temp();

        // Início da Transação
        gpio_put(PIN_CS, 0);
        vTaskDelay(pdMS_TO_TICKS(50)); // Setup time para estabilidade

        // Envia a temperatura e lê o retorno (Loopback físico na FPGA)
```

```
        spi_write_read_blocking(SPI_PORT, &temp_enviada, &dado_recebido, 1);

        vTaskDelay(pdMS_TO_TICKS(50));
        gpio_put(PIN_CS, 1); // Fim da transação (Gatilho da Auditoria na FPGA)

        // --- RELATÓRIO DE DIAGNÓSTICO NO TERMINAL ---
        printf("[RP2040] Enviado: %d C | Recebido: %d ", temp_enviada, dado_recebido);

        if (temp_enviada == dado_recebido) {
            printf("[LINK OK]\n");
        } else {
            // Se os dados forem diferentes, há ruído alterando os bits no percurso
            printf("[ERRO DE BIT]\n");
        }

        printf("-> Observe na FPGA: LED Verde (Integridade) e Vermelho (Limite >30)\n\n");

        vTaskDelay(pdMS_TO_TICKS(2000));
    }
}

// --- TAREFA: STATUS LOCAL (BLINK) ---
void vBlinkLocalTask(void *pvParameters) {
    gpio_init(LED_PIN_RP2040);
    gpio_set_dir(LED_PIN_RP2040, GPIO_OUT);
    for(;;) {
        gpio_put(LED_PIN_RP2040, 1);
        vTaskDelay(pdMS_TO_TICKS(100));
        gpio_put(LED_PIN_RP2040, 0);
        vTaskDelay(pdMS_TO_TICKS(900));
    }
```

```
}


int main() {
    stdio_init_all();
    sleep_ms(3000);

    printf("========================================\n");
    printf("   ESTRATEGIA 1: AUDITORIA DE BARRAMENTO \n");
    printf("========================================\n");

    xTaskCreate(vBlinkLocalTask, "LocalBlink", 128, NULL, 1, NULL);
    xTaskCreate(vSensorFpgaTask, "SensorFPGA", 256, NULL, 2, NULL);

    vTaskStartScheduler();
    while (1);
}
```

- **Apêndice A.2:** 01.RP2040/src/main_oversampling.c (Algoritmo de envio redundante para teste de ruído).

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/spi.h"
#include "FreeRTOS.h"
#include "task.h"
#include "hardware/i2c.h"

// --- CONFIGURAÇÃO DE PINAGEM (BitDogLab) ---
#define SPI_PORT spi0
#define PIN_MISO 16
#define PIN_CS   17
#define PIN_SCK  18
```

```c
#define PIN_MOSI 19
#define LED_PIN_RP2040 11

// Função de leitura mantendo o padrão (25°C - 35°C)
uint8_t read_aht10_temp() {
    static uint8_t t = 25;
    t++;
    if(t > 35) t = 25;
    return t;
}

void hardware_spi_init() {
    // Mantemos 10kHz para garantir que o "meio do bit" seja bem largo para a FPGA amostrar
    spi_init(SPI_PORT, 10 * 1000);
    gpio_set_function(PIN_MISO, GPIO_FUNC_SPI);
    gpio_set_function(PIN_SCK,  GPIO_FUNC_SPI);
    gpio_set_function(PIN_MOSI, GPIO_FUNC_SPI);
    gpio_init(PIN_CS);
    gpio_set_dir(PIN_CS, GPIO_OUT);
    gpio_put(PIN_CS, 1);
}

// --- TAREFA: VALIDAÇÃO DE PRECISÃO DE DADOS (OVERSAMPLING) ---
void vSensorFpgaTask(void *pvParameters) {
    hardware_spi_init();
    uint8_t dado_recebido = 0;

    printf("\n--- TESTE 3: PRECISÃO VIA AMOSTRAGEM NO CENTRO DO BIT ---\n");
    printf("Objetivo: Garantir que o LED Vermelho siga a regra > 30 com perfeição.\n");

    for(;;) {
```

```
    uint8_t temp_enviada = read_aht10_temp();

    gpio_put(PIN_CS, 0);
    // Delay proposital para garantir estabilidade elétrica antes do primeiro bit
    vTaskDelay(pdMS_TO_TICKS(50));

    // Transmissão Full-Duplex
    spi_write_read_blocking(SPI_PORT, &temp_enviada, &dado_recebido, 1);

    vTaskDelay(pdMS_TO_TICKS(50));
    gpio_put(PIN_CS, 1);

    // --- RELATÓRIO TÉCNICO NO TERMINAL ---
    printf("[RP2040] Temp Enviada: %d C | Retorno FPGA: %d ", temp_enviada, dado_recebido);

    if (temp_enviada == dado_recebido) {
        printf("[DADO ÍNTEGRO]\n");
    } else {
        // Se falhar aqui no Teste 3, indica que o deslocamento (skew) entre Clock e Dado
        // ainda é maior que a janela de compensação do oversampling.
        printf("[ERRO DE SINCRONISMO]\n");
    }

    // Dica visual para o usuário
    if (temp_enviada > 30) {
        printf(">> Esperado na FPGA: LED Vermelho ACESO (Alerta)\n");
    } else {
        printf(">> Esperado na FPGA: LED Vermelho APAGADO\n");
    }
    printf("-------------------------------------------------\n");
```

```
        vTaskDelay(pdMS_TO_TICKS(2000));
    }
}

// --- TAREFA: STATUS (BLINK) ---
void vBlinkLocalTask(void *pvParameters) {
    gpio_init(LED_PIN_RP2040);
    gpio_set_dir(LED_PIN_RP2040, GPIO_OUT);
    for(;;) {
        gpio_put(LED_PIN_RP2040, 1);
        vTaskDelay(pdMS_TO_TICKS(100));
        gpio_put(LED_PIN_RP2040, 0);
        vTaskDelay(pdMS_TO_TICKS(900));
    }
}

int main() {
    stdio_init_all();
    sleep_ms(3000);

    printf("========================================\n");
    printf("   ESTRATEGIA 3: OVERSAMPLING & SKEW     \n");
    printf("========================================\n");

    xTaskCreate(vBlinkLocalTask, "LocalBlink", 128, NULL, 1, NULL);
    xTaskCreate(vSensorFpgaTask, "SensorFPGA", 256, NULL, 2, NULL);

    vTaskStartScheduler();
    while (1);
}
```

- **Apêndice A.3:** 01.RP2040/src/main_glitch.c (Teste de resiliência a transientes no barramento).

```c
#include <stdio.h>

#include "pico/stdlib.h"

#include "hardware/spi.h"

#include "FreeRTOS.h"

#include "task.h"

#include "hardware/i2c.h"



// --- CONFIGURAÇÃO DE PINAGEM (BitDogLab) ---

#define SPI_PORT spi0

#define PIN_MISO 16

#define PIN_CS   17

#define PIN_SCK  18

#define PIN_MOSI 19

#define LED_PIN_RP2040 11



uint8_t read_aht10_temp() {

   static uint8_t t = 25;

   t++;

   if(t > 35) t = 25;

   return t;

}



void hardware_spi_init() {

   spi_init(SPI_PORT, 10 * 1000); // 10kHz

   gpio_set_function(PIN_MISO, GPIO_FUNC_SPI);

   gpio_set_function(PIN_SCK,  GPIO_FUNC_SPI);

   gpio_set_function(PIN_MOSI, GPIO_FUNC_SPI);

   gpio_init(PIN_CS);

   gpio_set_dir(PIN_CS, GPIO_OUT);

   gpio_put(PIN_CS, 1);
```

```c
}


// --- TAREFA: VALIDAÇÃO DE FILTRO VIA LOOPBACK ---
void vSensorFpgaTask(void *pvParameters) {
    hardware_spi_init();
    uint8_t dado_recebido = 0;

    printf("\n--- TESTE 2: ESTABILIZAÇÃO VIA FILTRO DE GLITCH ---\n");

    for(;;) {
        uint8_t temp_enviada = read_aht10_temp();

        gpio_put(PIN_CS, 0);
        vTaskDelay(pdMS_TO_TICKS(50)); // Tempo para estabilização do filtro na FPGA

        // spi_write_read_blocking envia e recebe ao mesmo tempo (Full-Duplex)
        spi_write_read_blocking(SPI_PORT, &temp_enviada, &dado_recebido, 1);

        vTaskDelay(pdMS_TO_TICKS(50));
        gpio_put(PIN_CS, 1);

        // --- PRINT DE TESTE ESPECÍFICO ---
        printf("[RP2040] TX: %d C | RX (Loopback): %d ", temp_enviada, dado_recebido);

        if (temp_enviada == dado_recebido) {
            printf("[OK - Sinal Limpo]\n");
        } else {
            // Se falhar aqui, o filtro na FPGA ainda não está vencendo o ruído físico
            printf("[ERRO - Ruído Detectado]\n");
        }
```

```
        vTaskDelay(pdMS_TO_TICKS(2000));
    }
}

// --- TAREFA: BLINK LOCAL ---
void vBlinkLocalTask(void *pvParameters) {
    gpio_init(LED_PIN_RP2040);
    gpio_set_dir(LED_PIN_RP2040, GPIO_OUT);
    for(;;) {
        gpio_put(LED_PIN_RP2040, 1);
        vTaskDelay(pdMS_TO_TICKS(100));
        gpio_put(LED_PIN_RP2040, 0);
        vTaskDelay(pdMS_TO_TICKS(900));
    }
}

int main() {
    stdio_init_all();
    sleep_ms(3000);

    printf("=======================================\n");
    printf("   ESTRATEGIA 2: FILTRO DE GLITCH (VOTACAO) \n");
    printf("=======================================\n");

    xTaskCreate(vBlinkLocalTask, "LocalBlink", 128, NULL, 1, NULL);
    xTaskCreate(vSensorFpgaTask, "SensorFPGA", 256, NULL, 2, NULL);

    vTaskStartScheduler();
    while (1);
}
```

- **Apêndice A.4:** 01.RP2040/src/usb_task.c (Interface serial para monitoramento da telemetria).

```c
#include <stdio.h>

#include "pico/stdlib.h"

#include "FreeRTOS.h"

#include "task.h"

#include "usb_task.h"

void vUsbTask(void *pvParameters) {

   int ch;

   for (;;){

       // Tenta ler um caractrere da USB com timeout de 0 (não bloqueia)

       // Se não tiver nada, ele retorna PICO_ERROR_TIMEOUT

       ch = getchar_timeout_us(0);



       // Se recebeu algo válida (diferente de erro)

       if(ch != PICO_ERROR_TIMEOUT){

           printf("Recebi na USB: %c (ASCII: %d)\n", ch, ch);



           //será adicionado os dados do bitstream no buffer

       }



       //Ele fica verificando constantemente a CPU

       //Parar 10ms para não sobrecarregar a CPU

       vTaskDelay(pdMS_TO_TICKS(10));

   }

}
```

- **Apêndice A.5:** 01.RP2040/freertos_config/FreeRTOSConfig.h (Parâmetros do Kernel para tempo real).

```c
#ifndef FREERTOS_CONFIG_H

#define FREERTOS_CONFIG_H



/* ========================================================== */
```

```
/* CONFIGURAÇÕES DO HARDWARE RP2040 (DUAL CORE)             */
/* Estas definições PRECISAM ficar no topo do arquivo.      */
/* ======================================================== */
#define configNUM_CORES                       2
#define configTICK_CORE                       0
#define configUSE_CORE_AFFINITY               0

/* ======================================================== */
/* CONFIGURAÇÕES GERAIS DO KERNEL FREERTOS                  */
/* ======================================================== */
#define configUSE_PREEMPTION                  1
#define configUSE_PORT_OPTIMISED_TASK_SELECTION 0
#define configUSE_TICKLESS_IDLE               0
#define configCPU_CLOCK_HZ                    125000000
#define configTICK_RATE_HZ                    1000
#define configMAX_PRIORITIES                  5
#define configMINIMAL_STACK_SIZE              128
#define configMAX_TASK_NAME_LEN               16
#define configUSE_16_BIT_TICKS                0
#define configIDLE_SHOULD_YIELD               1
#define configUSE_TASK_NOTIFICATIONS          1
#define configTASK_NOTIFICATION_ARRAY_ENTRIES 3
#define configUSE_MUTEXES                     1
#define configUSE_RECURSIVE_MUTEXES           1
#define configUSE_COUNTING_SEMAPHORES         1
#define configQUEUE_REGISTRY_SIZE             10
#define configUSE_QUEUE_SETS                  0
#define configUSE_TIME_SLICING                1
#define configUSE_NEWLIB_REENTRANT            0
#define configENABLE_BACKWARD_COMPATIBILITY   0
#define configNUM_THREAD_LOCAL_STORAGE_POINTERS 5
```

```
#define configSTACK_DEPTH_TYPE                  uint32_t
#define configMESSAGE_BUFFER_LENGTH_TYPE        size_t

/* Gerenciamento de Memória */
#define configSUPPORT_STATIC_ALLOCATION         0
#define configSUPPORT_DYNAMIC_ALLOCATION        1
#define configTOTAL_HEAP_SIZE                   (128*1024)
#define configAPPLICATION_ALLOCATED_HEAP        0

/* Hooks (Funções de Gancho) */
#define configUSE_IDLE_HOOK                     0
#define configUSE_TICK_HOOK                     0
#define configCHECK_FOR_STACK_OVERFLOW          0
#define configUSE_MALLOC_FAILED_HOOK            0
#define configUSE_DAEMON_TASK_STARTUP_HOOK      0

/* Estatísticas */
#define configGENERATE_RUN_TIME_STATS           0
#define configUSE_TRACE_FACILITY                1
#define configUSE_STATS_FORMATTING_FUNCTIONS    0

/* Co-rotinas e Software Timers */
#define configUSE_CO_ROUTINES                   0
#define configMAX_CO_ROUTINE_PRIORITIES         1
#define configUSE_TIMERS                        1
#define configTIMER_TASK_PRIORITY               (configMAX_PRIORITIES - 1)
#define configTIMER_QUEUE_LENGTH                10
#define configTIMER_TASK_STACK_DEPTH            configMINIMAL_STACK_SIZE

/* Debug: Travar se der erro (assert) */
```

```
/*#define configASSERT(x)                         if((x) == 0) { taskDISABLE_INTERRUPTS(); for(;;); }*/

/* Usa portDISABLE_INTERRUPTS que é mais baixo nível */

#define configASSERT( x )    if( ( x ) == 0 ) { portDISABLE_INTERRUPTS(); for( ;; ); }



/* Funções Opcionais da API */

#define INCLUDE_vTaskPrioritySet               1

#define INCLUDE_uxTaskPriorityGet              1

#define INCLUDE_vTaskDelete                    1

#define INCLUDE_vTaskSuspend                   1

#define INCLUDE_xResumeFromISR                 1

#define INCLUDE_vTaskDelayUntil                1

#define INCLUDE_vTaskDelay                     1

#define INCLUDE_xTaskGetSchedulerState         1

#define INCLUDE_xTaskGetCurrentTaskHandle      1

#define INCLUDE_uxTaskGetStackHighWaterMark    1

#define INCLUDE_xTaskGetIdleTaskHandle         1

#define INCLUDE_eTaskGetState                  1

#define INCLUDE_xEventGroupSetBitFromISR       1

#define INCLUDE_xTimerPendFunctionCall         1

#define INCLUDE_xTaskAbortDelay                1

#define INCLUDE_xTaskGetHandle                 1

#define INCLUDE_xTaskResumeFromISR             1



/* Definições específicas do port SMP (Multicore) */

#define configSMP_CORE_0_META_EXCEPTION_HANDLER 1

#define configSMP_CORE_1_META_EXCEPTION_HANDLER 1

#define configUSE_PASSIVE_IDLE_HOOK            0



#endif /* FREERTOS_CONFIG_H */
```

10.2. Módulo 02: Descrição de Hardware e IP Cores (FPGA)

Contém a descrição RTL e os módulos de topo que implementam as soluções de integridade de sinal.

- **Apêndice B.1:** 02.FPGA/hdl/spi_slave.sv (**IP Core Original**: FSM de recepção SPI).

```
module spi_slave(

    //Interface controle

    input logic clk,     //clock interno FPGA (25MHz)

    input logic rst,     //reset nivel baixo



    //Interface SPI (Recebe da RP2040)

    input logic sck,     //SCK pulso externo - clock SPI mestre

    input logic mosi,    //dados (master out slave in)

    input logic cs,      //chip select (ativo em 0)



    //Interface RISC-V

    output logic [7:0] data_out,     //byte de dados RP2040

    output logic       data_valid,   //aviso de dados

    output logic sck_rise   //pulso interno - borda de subida do SCK



);

    logic sck_atual;   //valor estado atual - FF A

    logic sck_antigo;   //valor estado antigo - FF B





    //logic sck_register;     //shift_register

    //localparam MAX = 25000000;  //25 MHz



    /* Construir a sequencia para receber os dados da RP2040

     ****     Iniciar com a Detecção da Borda    ****

     * Armazenar os sinais de Clock (25MHz) - sinais do sck - SPI

     * 1. Guardar o valor no estado agora

     * 2. Guardar o valor no estado anterior
```

```
 * 3. Comparar -- Se no estado anterior o sck era 0 e no agora é 1 - verificar a borda de subida
 * Objetivo: Sincronização transforma um sinal externo em um pulso interno - alinhamento com o chip
 */



//sincronizar o SCK externo - RP2040
always_ff @(posedge clk or negedge rst) begin
    if(!rst) begin
        sck_atual <= 1'b0;
        sck_antigo <= 1'b0;
    end else begin
        sck_atual <= sck;  //captura o externo
        sck_antigo <= sck_atual;   //guarda o passado
    end
end

assign sck_rise = sck_atual & ~sck_antigo; //detectar a borda de subida | inverte sck_antigo

//Ler sck_rise para obter o byte - FSM
//1. Defir os estados serão 3
typedef enum logic [1:0] {
    IDLE,       //cs = 1 -> esperando
    RECEIVE,    //cs = 0 -> recebendo
    DONE        // final byte completo
} state_t;

state_t estado_atual;

//2. Logica para armazenar os dados
logic [2:0] bit_count;   //contador
logic [7:0] shift_reg;   //byte transmitido
```

```
//3. Logica sequencial
always_ff @( posedge clk or negedge rst ) begin
    if(!rst) begin
        //IDLE -> reset
        estado_atual <= IDLE;
        bit_count <= 0;
        shift_reg <= 0;
        //saidas
        data_out <= 0;
        data_valid <= 0;
    end else begin
        case (estado_atual)
            IDLE: begin
                bit_count <= 0;       // Prepara contador
                data_valid <= 0;      // Desliga aviso anterior

                // Se o CS baixar (0)
                if (cs == 1'b0) begin
                    estado_atual <= RECEIVE;
                end
            end

            RECEIVE: begin
                // Segurança: Se o CS subir (1), cancela tudo
                if (cs == 1'b1) begin
                    estado_atual <= IDLE; // Reset se CS subir
                end else begin
                    // Só fazemos algo se o Detector de Borda avisar (sck_rise)
                    if (sck_rise) begin
                        // 1. Pega o bit do fio MOSI e empurra na fila
```

```
                        shift_reg <= {shift_reg[6:0], mosi};

                        // 2. Incrementa o contador de bits
                        bit_count <= bit_count + 1;

                        // 3. Verifica se completou 8 bits (0 ou 7)
                        // Como bit_count tem 3 bits, ao somar 1 em 7 (111), ele vira 0 (000).
                        // Então se bit_count chegou em 7, este é o último bit
                        if (bit_count == 3'd7) begin
                            estado_atual <= DONE;
                        end
                    end
                end
            end

            DONE: begin
                data_out <= shift_reg; // Publica o byte
                data_valid <= 1;       // valida para o RISC-V

                // Volta a dormir e esperar o próximo pacote
                estado_atual <= IDLE;
            end
        endcase
    end
end

endmodule
```

- **Apêndice B.2:** 02.FPGA/hdl/top_integridade.sv (Módulo de topo para validação de dados íntegros).

```
`timescale 1ns/1ps
```

```
module tb_integridade;

    // Sinais de interface
    logic clk_in, spi_cs, spi_sck, spi_mosi, rst;
    logic spi_miso;
    logic [1:0] led;


    // Instanciação do Device Under Test (DUT)
    // Usando o nome que você definiu para o arquivo
    //top_integridade
    top dut (
        .clk_in(clk_in),
        .spi_cs(spi_cs),
        .spi_sck(spi_sck),
        .spi_mosi(spi_mosi),
        .rst(rst),
        .spi_miso(spi_miso),
        .led(led)
    );


    // Gerador de Clock Principal (25MHz -> Período de 40ns)
    initial clk_in = 0;
    always #20 clk_in = ~clk_in;


    // Tarefa para simular o comportamento do mestre (RP2040)
    task enviar_byte(input [7:0] dado, input integer bits_para_enviar);
        integer i;
        $display("[SIM] Iniciando envio: 0x%h (%0d bits)", dado, bits_para_enviar);


        spi_cs = 0; // Ativa o Chip Select
        #200;       // Tempo de setup
```

```
    for (i = 7; i > 7 - bits_para_enviar; i = i - 1) begin
        spi_mosi = dado[i];
        #400; spi_sck = 1; // Borda de subida (FPGA amostra aqui)
        #400; spi_sck = 0; // Borda de descida
        #400;
    end

    #200;
    spi_cs = 1; // Desativa CS (Gatilho da Auditoria de Integridade)
    #1000;      // Espaço entre mensagens
endtask

// Procedimento de Teste
initial begin
    // Configuração para visualização no VS Code (WaveTrace/GTKWave)
    $dumpfile("integridade.vcd");
    $dumpvars(0, tb_integridade);

    // 1. Reset do Sistema
    rst = 1; spi_cs = 1; spi_sck = 0; spi_mosi = 0;
    #100 rst = 0;
    #200;

    // CENÁRIO A: Transação Válida (8 bits, Valor = 35)
    // O LED Vermelho deve acender (led[0]=0) pois 35 > 30
    // O LED Verde deve acender (led[1]=0) pois integridade é OK
    $display("Simulando: Envio válido de 8 bits (35C)");
    enviar_byte(8'd35, 8);

    // CENÁRIO B: Transação Válida (8 bits, Valor = 25)
```

```
        // O LED Vermelho deve apagar (led[0]=1) pois 25 < 30
        // O LED Verde deve continuar aceso (led[1]=0)
        $display("Simulando: Envio válido de 8 bits (25C)");
        enviar_byte(8'd25, 8);

        // CENÁRIO C: Falha de Integridade (Simulando ruído que "comeu" 1 bit)
        // Enviamos apenas 7 bits. O LED Verde DEVE APAGAR (led[1]=1)
        // O LED Vermelho não deve mudar, pois a transação foi descartada.
        $display("Simulando: ERRO - Apenas 7 bits detectados");
        enviar_byte(8'hFF, 7);

        #2000;
        $display("[SIM] Simulação finalizada. Analise o arquivo .vcd");
        $finish;
    end
endmodule
```

- **Apêndice B.3:** 02.FPGA/hdl/top_oversampling.sv (Implementação de amostragem múltipla).

```
`timescale 1ns/1ps

module tb_oversampling;
    // Declaração de sinais
    logic clk_in, spi_cs, spi_sck, spi_mosi, rst, spi_miso;
    logic [1:0] led;

    // Instanciação do módulo com Amostragem Retardada
    top dut (.*);

    // Gerador de Clock Principal (25MHz -> 40ns)
```

```
    initial clk_in = 0;
    always #20 clk_in = ~clk_in;

    initial begin
        // Configuração dos arquivos de onda para VS Code
        $dumpfile("oversampling.vcd");
        $dumpvars(0, tb_oversampling);

        // --- ESTADO INICIAL ---
        $display("\n[SIM] --- INICIANDO TESTE 3: OVERSAMPLING (AMOSTRAGEM NO CENTRO DO BIT) ---");
        rst = 1; spi_cs = 1; spi_sck = 0; spi_mosi = 0;
        #100 rst = 0;
        #200;

        // --- CENÁRIO: SIMULAÇÃO DE SKEW (ATRASO CRÍTICO) ---
        $display("[SIM] Passo 1: Iniciando transação com atraso proposital de sinal (Skew)");
        spi_cs = 0;
        #200;

        // Simulando um problema físico real: O dado MOSI demora a subir
        // Em um sistema comum, isso causaria erro. No Teste 3, a FPGA deve esperar.
        $display("[SIM] Passo 2: Subindo o Clock (SCK), mas mantendo o Dado (MOSI) instável por 20ns");
        spi_sck = 1;
        #20;
        spi_mosi = 1; // O dado só fica correto 20ns APÓS o clock subir
        #380;

        $display("[SIM] Passo 3: Clock desce. A FPGA deve ter capturado o dado apenas no 4o ciclo de
clk_in.");
        spi_sck = 0;

        // --- FINALIZAÇÃO E ANÁLISE ---
```

```
        #500;
        spi_cs = 1; // Gatilho de auditoria

        $display("[SIM] Passo 4: Verificando integridade final.");
        #100;

        $display("[SIM] Analise no Waveform:");
        $display("      - O contador 'sample_delay_cnt' deve subir após a borda do SCK.");
        $display("      - A captura no 'shift_reg' deve ocorrer exatamente quando o contador atingir 4.");
        $display("      - Isso prova que ignoramos a instabilidade dos primeiros 20ns.");

        #1000;
        $display("[SIM] --- FIM DO TESTE 3 --- \n");
        $finish;
    end
endmodule
```

- **Apêndice B.4:** 02.FPGA/hdl/top_filtro.sv (Implementação de filtros digitais contra *glitches*).

```
module top_filtro (
    input  logic clk_in, spi_cs, spi_sck, spi_mosi, rst,
    output logic spi_miso,
    output logic [1:0] led
);
    // 1. Registadores de Histórico (Amostragem de 4 estágios)
    logic [3:0] sck_hist, cs_hist, mosi_hist;
    logic sck_f, cs_f, mosi_f;

    always_ff @(posedge clk_in) begin
        sck_hist  <= {sck_hist[2:0],  spi_sck};
        cs_hist   <= {cs_hist[2:0],   spi_cs};
```

```
    mosi_hist <= {mosi_hist[2:0], spi_mosi};

    // Lógica de Votação (Majority Vote):
    // Só altera o sinal filtrado (_f) se os 4 últimos ciclos forem idênticos.
    if (sck_hist == 4'b1111) sck_f <= 1'b1; else if (sck_hist == 4'b0000) sck_f <= 1'b0;
    if (cs_hist  == 4'b1111) cs_f  <= 1'b1; else if (cs_hist  == 4'b0000) cs_f  <= 1'b0;
    if (mosi_hist == 4'b1111) mosi_f <= 1'b1; else if (mosi_hist == 4'b0000) mosi_f <= 1'b0;
end

// 2. Detecção de Borda nos Sinais Filtrados
logic sck_f_prev, cs_f_prev;
always_ff @(posedge clk_in) begin
    sck_f_prev <= sck_f;
    cs_f_prev  <= cs_f;
end
wire sck_posedge = (sck_f && !sck_f_prev);
wire cs_end      = (cs_f && !cs_f_prev); // Borda de subida do CS filtrado

// 3. Auditoria de Integridade (Herdada do Teste 1)
logic [3:0] bit_cnt;
logic [7:0] shift_reg, data_latch;
logic integrity_ok;

always_ff @(posedge clk_in) begin
    if (cs_f) begin
        bit_cnt <= 0;
    end else if (sck_posedge) begin
        bit_cnt   <= bit_cnt + 1;
        shift_reg <= {shift_reg[6:0], mosi_f};
    end
```

```
        if (cs_end) begin
            integrity_ok <= (bit_cnt == 4'd8);
            if (bit_cnt == 4'd8) data_latch <= shift_reg;
        end
    end

    assign led[1] = !integrity_ok;
    assign led[0] = (data_latch > 8'd30) ? 1'b0 : 1'b1;
    assign spi_miso = spi_mosi;
endmodule
```

- **Apêndice B.5:** 02.FPGA/hdl/top_soc.sv (Integração final: CPU + Memória + IPs).

```
/* * Arquivo: top_soc.sv
* Objetivo: Wrapper principal do SoC integrando PicoRV32, RAM e Periféricos (SPI/LEDs)
* Autor: Márcio Barbosa (Mentoria Técnica)
*/

module top_soc (
    input  logic clk_in,    // Clock de 25MHz (BitDogLab)
    input  logic rst_in,    // Reset Ativo Baixo (Vem do tb_bootloader)
    input  logic spi_cs,
    input  logic spi_sck,
    input  logic spi_mosi,
    output logic spi_miso,
    output logic [7:0] led // LEDs de diagnóstico (8 bits)
);

    // --- 1. SUBSISTEMA SPI (Lógica de Robustez do Teste 3) ---
    logic [3:0] sck_hist, cs_hist, mosi_hist;
    logic sck_f, cs_f, mosi_f;
```

```
logic sck_f_prev;
logic [2:0] sample_delay_cnt;
logic [3:0] bit_cnt;
logic [7:0] shift_reg;
logic [7:0] data_latch_spi;

// Filtro de Maioria (Majority Filter)
always_ff @(posedge clk_in) begin
    sck_hist  <= {sck_hist[2:0], spi_sck};
    cs_hist   <= {cs_hist[2:0],  spi_cs};
    mosi_hist <= {mosi_hist[2:0], spi_mosi};

    if (sck_hist == 4'b1111) sck_f <= 1'b1; else if (sck_hist == 4'b0000) sck_f <= 1'b0;
    if (cs_hist  == 4'b1111) cs_f  <= 1'b1; else if (cs_hist  == 4'b0000) cs_f  <= 1'b0;
    if (mosi_hist == 4'b1111) mosi_f <= 1'b1; else if (mosi_hist == 4'b0000) mosi_f <= 1'b0;
end

// Detecção de Borda e Captura com Oversampling
always_ff @(posedge clk_in) sck_f_prev <= sck_f;
wire sck_posedge = (sck_f && !sck_f_prev);
wire cs_active   = !cs_f;

always_ff @(posedge clk_in) begin
    if (!cs_active) begin
        bit_cnt <= 0;
        sample_delay_cnt <= 0;
    end else begin
        if (sck_posedge) sample_delay_cnt <= 3'd1;
        else if (sample_delay_cnt > 0 && sample_delay_cnt < 3'd5)
            sample_delay_cnt <= sample_delay_cnt + 3'd1;
```

```
        if (sample_delay_cnt == 3'd4) begin
            bit_cnt   <= bit_cnt + 1;
            shift_reg <= {shift_reg[6:0], mosi_f};
            sample_delay_cnt <= 0;
        end
    end
    // Atualiza o registrador quando o CS sobe (Fim da transação)
    if (cs_f && !cs_hist[1]) begin
        if (bit_cnt == 4'd8) data_latch_spi <= shift_reg;
    end
end
assign spi_miso = spi_mosi; // Loopback de diagnóstico

// --- 2. BARRAMENTO DO PROCESSADOR RISC-V ---
logic        mem_valid;
logic        mem_ready;
logic [31:0] mem_addr;
logic [31:0] mem_wdata;
logic [ 3:0] mem_wstrb;
logic [31:0] mem_rdata;

// --- 3. INSTANCIAÇÃO DO PICORV32 ---
picorv32 #(
    .ENABLE_COUNTERS(1),
    .ENABLE_REGS_16_31(1),
    .PROGADDR_RESET(32'h 0000_0000), // Início da RAM
    .STACKADDR(32'h 0000_0F00)       // Fim da RAM (4KB)
) cpu (
    .clk         (clk_in),
    .resetn      (rst_in),
    .mem_valid   (mem_valid),
```

```
    .mem_ready    (mem_ready),
    .mem_addr     (mem_addr),
    .mem_wdata    (mem_wdata),
    .mem_wstrb    (mem_wstrb),
    .mem_rdata    (mem_rdata)
);

// --- 4. DECODIFICADOR DE ENDEREÇOS (Memory Map) ---
logic [7:0] led_reg;
logic ram_ready;
logic [31:0] ram_rdata;
logic [31:0] keyboard_dummy = 32'hCAFE_BABE; // Placeholder Teclado

// Sinais de Seleção baseados no firmware.c
wire sel_ram = (mem_addr[31:16] == 16'h0000); // 0x00000000
wire sel_spi = (mem_addr == 32'h0001_0000);   // REG_SPI
wire sel_led = (mem_addr == 32'h0002_0000);   // REG_LEDS
wire sel_kbd = (mem_addr == 32'h0003_0000);   // REG_KEYBOARD

always_comb begin
    mem_ready = 1'b0;
    mem_rdata = 32'h0;
    if (sel_ram) begin
        mem_ready = ram_ready;
        mem_rdata = ram_rdata;
    end else if (sel_spi) begin
        mem_ready = 1'b1;
        mem_rdata = {24'd0, data_latch_spi};
    end else if (sel_led) begin
        mem_ready = 1'b1;
        mem_rdata = {24'd0, led_reg};
```

```
        end else if (sel_kbd) begin
            mem_ready = 1'b1;
            mem_rdata = keyboard_dummy;
        end
    end

    // Registro de Saída (Escrita nos LEDs)
    always_ff @(posedge clk_in) begin
        if (!rst_in) led_reg <= 8'h00;
        else if (mem_valid && sel_led && mem_wstrb[0])
            led_reg <= mem_wdata[7:0];
    end

    assign led = led_reg;

    // --- 5. INSTANCIAÇÃO DA MEMÓRIA (IP CORE INFERIDO) ---
    soc_memory ram_inst (
        .clk(clk_in),
        .valid(mem_valid && sel_ram),
        .ready(ram_ready),
        .addr(mem_addr[11:2]), // Endereço de palavra (4KB = 1024 words)
        .wdata(mem_wdata),
        .wstrb(mem_wstrb),
        .rdata(ram_rdata)
    );

endmodule
```

- **Apêndice B.6:** 02.FPGA/hdl/picorv32.v (Processador RISC-V utilizado como *soft-core*).
- **Apêndice B.7:** 02.FPGA/hdl/soc_memory.v (Instanciação de BRAM e interface de memória).

```
module soc_memory (
```

```
    input clk,
    input valid,
    output reg ready,
    input [9:0] addr,
    input [31:0] wdata,
    input [3:0] wstrb,
    output reg [31:0] rdata
);
    reg [31:0] mem [0:1023]; // RAM de 4KB

    // Carrega o firmware compilado do seu arquivo .hex
    initial $readmemh("hdl/firmware.hex", mem);

    always @(posedge clk) begin
        ready <= 1'b0;
        if (valid) begin
            ready <= 1'b1;
            rdata <= mem[addr];
            if (wstrb[0]) mem[addr][7:0]   <= wdata[7:0];
            if (wstrb[1]) mem[addr][15:8]  <= wdata[15:8];
            if (wstrb[2]) mem[addr][23:16] <= wdata[23:16];
            if (wstrb[3]) mem[addr][31:24] <= wdata[31:24];
        end
    end
endmodule
```

- **Apêndice B.8:** 02.FPGA/hdl/pins.lpf (Mapeamento físico de IOs da Lattice ECP5).

```
# Arquivo: pins.lpf

# Mapeamento para Colorlight i9 (Lattice ECP5)
```

```
# 1. Clock Principal (25MHz)
# Na Colorlight i9, o oscilador de 25MHz está sempre no pino P3
# LOCATE COMP "clk_in" SITE "P3";
# IOBUF PORT "clk_in" IO_TYPE=LVCMOS33;

# 2. Reset (Botão ou Pull-up)
# Usar um pino que geralmente tem botão ou deixamos flutuando com Pull-up interno
# K18 é comum nessas placas, mas se não tiver botão, o PULLMODE=UP segura o reset solto
# LOCATE COMP "rst_in" SITE "K18";
# IOBUF PORT "rst_in" IO_TYPE=LVCMOS33 PULLMODE=UP;

# 3. LED de Teste (Led Vermelho da Placa)
# O LED da Colorlight i9 fica no pino L16 e acende com nível BAIXO (invertido)
# LOCATE COMP "led[0]" SITE "L16";
# IOBUF PORT "led[0]" IO_TYPE=LVCMOS33;

# (Os outros LEDs do vetor [7:1] não precisamos mapear agora, o Nextpnr vai ignorar)

# Testar SPI acender um LED

# --- Mapeamento para Colorlight i9 (ECP5-45) ---

# --- Esquemático i5-i9-extboard ---

# Clock de 25MHz (P3 é padrão da Colorlight i9)
LOCATE COMP "clk_in" SITE "P3";
IOBUF PORT "clk_in" IO_TYPE=LVCMOS33;
```

```
# Botão (Reset) - PR20D (branco) -> Site K18 -- Botão Active High, Pull-Down ativado
LOCATE COMP "rst" SITE "K18";
IOBUF PORT "rst" IO_TYPE=LVCMOS33 PULLMODE=DOWN;

#LEDs (Active Low: 0=Aceso, 1=Apagado)
# LED 1 (Vermelho) - PL38C (azul) -> Site N4
LOCATE COMP "led[0]" SITE "N4";
IOBUF PORT "led[0]" IO_TYPE=LVCMOS33;

# LED 2 (Verde) - PR35B (marrom) -> Site P16
LOCATE COMP "led[1]" SITE "P16";
IOBUF PORT "led[1]" IO_TYPE=LVCMOS33;

# --- Interface SPI (Mapeamento) ---
# CS (PR5D (branco) -> F18) ==> GP17 (vermelho)
LOCATE COMP "spi_cs" SITE "F18";
IOBUF PORT "spi_cs" IO_TYPE=LVCMOS33;

# SCK (PR8D (verde) -> H16) ==> GP18 (laranja)
LOCATE COMP "spi_sck" SITE "H16";
IOBUF PORT "spi_sck" IO_TYPE=LVCMOS33;

# MOSI (PR11B (amarelo) -> H17) ==> GP19 (preto)
LOCATE COMP "spi_mosi" SITE "H17";
IOBUF PORT "spi_mosi" IO_TYPE=LVCMOS33;
```

```
# MISO (PR8B (azul) -> G18) ==> GP16 (marrom)
LOCATE COMP "spi_miso" SITE "G18";
IOBUF PORT "spi_miso" IO_TYPE=LVCMOS33;

# TECLADO MATRICIAL (Conforme Tabela)
# Linhas (Rows) - Saídas da FPGA (Scan)
LOCATE COMP "row[0]" SITE "D1";  # R1 (Cinza)
LOCATE COMP "row[1]" SITE "C2";  # R2 (Branco)
LOCATE COMP "row[2]" SITE "B4";  # R3 (Preto)
LOCATE COMP "row[3]" SITE "D3";  # R4 (Marrom)

IOBUF PORT "row[0]" IO_TYPE=LVCMOS33;
IOBUF PORT "row[1]" IO_TYPE=LVCMOS33;
IOBUF PORT "row[2]" IO_TYPE=LVCMOS33;
IOBUF PORT "row[3]" IO_TYPE=LVCMOS33;

# Colunas (Cols) - Entradas da FPGA (Leitura)
LOCATE COMP "col[0]" SITE "C17"; # C1 (Vermelho)
LOCATE COMP "col[1]" SITE "B18"; # C2 (Laranja)
LOCATE COMP "col[2]" SITE "B20"; # C3 (Amarelo)
LOCATE COMP "col[3]" SITE "F20"; # C4 (Verde)

#IOBUF PORT "col[0]" IO_TYPE=LVCMOS33 PULLMODE=DOWN;
#IOBUF PORT "col[1]" IO_TYPE=LVCMOS33 PULLMODE=DOWN;
#IOBUF PORT "col[2]" IO_TYPE=LVCMOS33 PULLMODE=DOWN;
#IOBUF PORT "col[3]" IO_TYPE=LVCMOS33 PULLMODE=DOWN;
```

IOBUF PORT "col[0]" IO_TYPE=LVCMOS33 PULLMODE=UP;

IOBUF PORT "col[1]" IO_TYPE=LVCMOS33 PULLMODE=UP;

IOBUF PORT "col[2]" IO_TYPE=LVCMOS33 PULLMODE=UP;

IOBUF PORT "col[3]" IO_TYPE=LVCMOS33 PULLMODE=UP;

10.3. Módulo 03: Firmware do SoC e Software Embarcado (C)

Códigos compilados para execução no processador interno da FPGA.

- **Apêndice C.1:** 03.Firmware/firmware.c (Lógica de aplicação e leitura de registradores SPI).

```c
/* Arquivo: firmware.c */

/* Objetivo: Lógica principal de controle do SoC (System on Chip) */



#include <stdint.h>



/* --- MAPEAMENTO DE HARDWARE (MMIO) --- */

/* Estes endereços DEVEM bater exatamente com o top.sv */



/* Endereço do Registrador SPI (Apenas Leitura no nosso IP) */

#define REG_SPI    (*((volatile uint32_t *)0x00010000))



/* Endereço do Registrador de LEDs (Apenas Escrita) */

#define REG_LEDS   (*((volatile uint32_t *)0x00020000))



/* Endereço de Registrador Teclado Matricial */

#define REG_KEYBOARD (*((volatile uint32_t *)0x00030000))



/* --- FUNÇÕES AUXILIARES --- */



/* Função de delay simples (Gastando ciclos de CPU) */

/* Não temos Timer Hardware, loop forçado */

void delay(uint32_t count) {
```

```
    while (count > 0) {
        /* 'volatile' aqui impede o compilador de remover este loop vazio */
        volatile uint32_t dummy = 0;
        count--;
    }
}

/* --- FUNÇÃO PRINCIPAL --- */
/* O bootloader.s pula para cá */
void main() {
  // uint32_t contador = 0;
  // uint32_t dados_spi = 0;
  // uint32_t dados_teclado = 0;

   /* Loop Infinito (Super Loop) */
  // while (1) {
        /* 1. Ler dados vindos do RP2040 via SPI */
        /* O IP Core SPI Slave atualiza esse endereço quando recebe algo */
  //     dados_spi = REG_SPI;
        // 2. Lemos o teclado e o SPI
 //      dados_teclado = REG_KEYBOARD;

        /* 2. Lógica de Visualização */
        /* Se recebermos zero do SPI, mostramos um contador automático (modo demo) */
        /* Se recebermos algo do SPI, mostramos o dado recebido nos LEDs */

        //if (dados_spi == 0) {
        //    REG_LEDS = contador; /* Escreve contador nos LEDs */
        //    contador++;          /* Incrementa contador interno */
        //} else {
        //    REG_LEDS = dados_spi; /* Mostra o byte recebido do RP2040 */
```

```
    //}


    // 2. Prioridade de Visualização no LED Verde:
    // Se houver tecla pressionada, mostra o código da tecla.
    // Se não, mostra o que veio do RP2040 via SPI.
//     if (dados_teclado != 0) {
//         REG_LEDS = dados_teclado;
//     } else {
//         REG_LEDS = dados_spi;
//     }


    /* 3. Atraso para o olho humano perceber a mudança */
    /* Clock 25MHz -> delay 500000 gera aprox 100-200ms */
    //delay(500000);
  //  delay(100000); // Delay menor para resposta mais rápida do teclado


  // Sinalização de boot: Acende o LED Verde (bit 0) com 0xAA (binario 10101010)
 // Como é Active Low, o LED verde ligado ao bit 0 acenderá.
REG_LEDS = 0xAA;


while (1) {
    uint32_t dados_teclado = REG_KEYBOARD;
    uint32_t dados_spi = REG_SPI;


    if (dados_teclado != 0) {
        REG_LEDS = dados_teclado; // Teclado assume o controle
    } else {
        REG_LEDS = dados_spi;     // Mostra dados do RP2040
    }
    delay(100000);
}
```

```
}

//}
```

- **Apêndice C.2:** 03.Firmware/bootloader.s (Rotina de inicialização em Assembly RISC-V).

```
/* Arquivo: bootloader.s */

/* Objetivo: Inicializar a Pilha (Stack) e pular para o C */



.section .init      /* insere no inicio */

.global _start      /* inicio do programa */



_start:

   /* 1. Configurar o Stack Pointer (Pilha) */

   /* Nossa RAM começa em 0x00000000 e tem 1024 bytes (0x400) */

   /* A pilha cresce de cima para baixo. Colocar no meio (512 ou 0x200) */

   /* deixar espaço para o código em cima e dados em baixo. */

   li sp, 0x00000200



   /* 2. Salta para a função main() (que estará no arquivo .c) */

   call main



   /* 3. Trava de Segurança (Loop Infinito) */

   /* Se o main() retornar (o que não deve acontecer), travamos aqui */

loop:

   j loop
```

- **Apêndice C.3:** 03.Firmware/sections.lds (Script de endereçamento do mapa de memória).

/* Arquivo: sections.lds */

/* Objetivo: Mapear o código para dentro da nossa pequena RAM de 1KB */

```
MEMORY {
    /* Definimos uma região chamada 'mem' */
    /* ORIGIN: Endereço inicial (0x00000000) */
    /* LENGTH: Tamanho total (0x400 = 1024 bytes = 1KB) */
    mem : ORIGIN = 0x00000000, LENGTH = 0x00000400
}

SECTIONS {
    /* A saída será gravada sequencialmente na região 'mem' */
    .memory : {
        . = 0x000000; /* O contador de posição começa no zero */

        /* 1. Primeiro: A seção .init (Vem do bootloader.s) */
        /* Isso garante que a primeira instrução da RAM seja o salto inicial */
        *(.init);

        /* 2. Segundo: A seção .text (O código do programa C) */
        *(.text);

        /* 3. Terceiro: As seções de dados (Variáveis globais e constantes) */
        *(.data);
        *(.rodata);
        *(.bss);

        /* O resto da memória fica livre para a Pilha (Stack) que cresce do fim para o começo */
    } > mem
}
```

10.4. Módulo 04: Testbenches e Documentação de Validação

Artefatos de simulação e comprovação de integridade.

- **Apêndice D.1:** 02.FPGA/sim/tb_spi_slave.sv (Testbench principal da FSM).

```
`timescale 1ns/1ps



module tb_spi_slave;



  // 1. Sinais declarados (DUT - Device Under Test)

  logic clk_in;

  logic rst_in;

  logic sck_in;

  logic mosi_in;

  logic cs_in;



  // Saídas que vamos observar

  logic [7:0] data_out_o;

  logic data_valid_o;

  logic sck_rise_o; // Debug



  // 2. Instancias SPI Slave - conectar os fios

  spi_slave DUT (

      .clk(clk_in),

      .rst(rst_in),

      .sck(sck_in),

      .mosi(mosi_in),

      .cs(cs_in),

      .data_out(data_out_o),

      .data_valid(data_valid_o),

      .sck_rise(sck_rise_o)
```

```
);

// 3. Gerar clock da FPGA (25MHz)
initial begin
    clk_in = 0;
    forever begin
         #20 clk_in = ~clk_in; //inverte a cada 20ns (total 40ns)
    end
end

// 4. Testar as conecções
initial begin
    $dumpfile("waveform.vcd");
    $dumpvars(0,tb_spi_slave);

    // A. Estado Inicial
    rst_in = 0; // Reset apertado (ativo baixo)
    cs_in = 1;  // Desligado (ativo baixo)
    sck_in = 0;
    mosi_in = 0;

    #100; // Espera o sistema estabilizar
    rst_in = 1; // Solta o reset
    #100;

    // B. Começar Transmissão: RP2040 ativa o chip
    $display("Iniciando transmissao SPI...");
    cs_in = 0;
    #200; // Pequena pausa

    // C. Enviar o byte 0xA5 (Binário: 10100101)
```

```
    // Vamos criar uma tarefa (função) para enviar bit a bit para não repetir código
    enviar_byte(8'hA5);

    // D. Finalizar
    #100;
    cs_in = 1; // Desativa o chip

    #200;
    $display("Teste finalizado. Verifique se data_out = A5 e data_valid pulsou.");
    $finish;
end

    // --- Tarefa Auxiliar: Simula o envio de 1 Byte via SPI ---
task enviar_byte(input logic [7:0] valor);
    integer i;
    begin
        // Loop para enviar 8 bits (do mais significativo 7 para o 0)
        for (i = 0; i <= 7; i = i + 1) begin
            // 1. Coloca o bit no fio MOSI
            // O protocolo SPI envia o Bit Mais Significativo (MSB) primeiro (Bit 7).
            // Quando i=0, queremos o bit 7. (7 - 0 = 7)
            // Quando i=1, queremos o bit 6. (7 - 1 = 6)
            mosi_in = valor[7 - i];
            #100; // aguarda a RP2040

            // 2. Sobe o Clock (SCK High) - A FPGA lê aqui!
            sck_in = 1; // subida - verifica o bit
            #100; //espero clock alto

            // 3. Desce o Clock (SCK Low)
            sck_in = 0; // descida - prepara o próximo
```

```
            #100; //espera para recomeçar o loop

        end

    end

endtask


endmodule
```

- **Apêndice D.2:** 02.FPGA/hdl/tb_integridade.sv (Simulação de fluxo de dados contínuo).

```
`timescale 1ns/1ps



module tb_integridade;

    // Sinais de interface

    logic clk_in, spi_cs, spi_sck, spi_mosi, rst;

    logic spi_miso;

    logic [1:0] led;



    // Instanciação do Device Under Test (DUT)

    // Usando o nome que você definiu para o arquivo

    //top_integridade

    top dut (

        .clk_in(clk_in),

        .spi_cs(spi_cs),

        .spi_sck(spi_sck),

        .spi_mosi(spi_mosi),

        .rst(rst),

        .spi_miso(spi_miso),

        .led(led)
```

```
);

// Gerador de Clock Principal (25MHz -> Período de 40ns)
initial clk_in = 0;
always #20 clk_in = ~clk_in;

// Tarefa para simular o comportamento do mestre (RP2040)
task enviar_byte(input [7:0] dado, input integer bits_para_enviar);
    integer i;
    $display("[SIM] Iniciando envio: 0x%h (%0d bits)", dado, bits_para_enviar);

    spi_cs = 0; // Ativa o Chip Select
    #200;       // Tempo de setup

    for (i = 7; i > 7 - bits_para_enviar; i = i - 1) begin
        spi_mosi = dado[i];
        #400; spi_sck = 1; // Borda de subida (FPGA amostra aqui)
        #400; spi_sck = 0; // Borda de descida
        #400;
    end

    #200;
    spi_cs = 1; // Desativa CS (Gatilho da Auditoria de Integridade)
    #1000;      // Espaço entre mensagens
endtask

// Procedimento de Teste
initial begin
    // Configuração para visualização no VS Code (WaveTrace/GTKWave)
    $dumpfile("integridade.vcd");
    $dumpvars(0, tb_integridade);
```

```
        // 1. Reset do Sistema
        rst = 1; spi_cs = 1; spi_sck = 0; spi_mosi = 0;
        #100 rst = 0;
        #200;

        // CENÁRIO A: Transação Válida (8 bits, Valor = 35)
        // O LED Vermelho deve acender (led[0]=0) pois 35 > 30
        // O LED Verde deve acender (led[1]=0) pois integridade é OK
        $display("Simulando: Envio válido de 8 bits (35C)");
        enviar_byte(8'd35, 8);

        // CENÁRIO B: Transação Válida (8 bits, Valor = 25)
        // O LED Vermelho deve apagar (led[0]=1) pois 25 < 30
        // O LED Verde deve continuar aceso (led[1]=0)
        $display("Simulando: Envio válido de 8 bits (25C)");
        enviar_byte(8'd25, 8);

        // CENÁRIO C: Falha de Integridade (Simulando ruído que "comeu" 1 bit)
        // Enviamos apenas 7 bits. O LED Verde DEVE APAGAR (led[1]=1)
        // O LED Vermelho não deve mudar, pois a transação foi descartada.
        $display("Simulando: ERRO - Apenas 7 bits detectados");
        enviar_byte(8'hFF, 7);

        #2000;
        $display("[SIM] Simulação finalizada. Analise o arquivo .vcd");
        $finish;
    end
endmodule
```

- **Apêndice D.3:** 02.FPGA/hdl/tb_filtro.sv (Simulação de rejeição de ruído em sinais de clock).

```
`timescale 1ns/1ps



module tb_filtro;

    // Declaração de sinais

    logic clk_in, spi_cs, spi_sck, spi_mosi, rst, spi_miso;

    logic [1:0] led;



    // Instanciação do Módulo com Filtro Majoritário

    top dut (.*);



    // Gerador de Clock Principal (25MHz -> 40ns)

    initial clk_in = 0;

    always #20 clk_in = ~clk_in;



    initial begin

        // Configuração dos arquivos de onda para VS Code/GTKWave

        $dumpfile("filtro_glitch.vcd");

        $dumpvars(0, tb_filtro);



        // --- ESTADO INICIAL ---

        $display("\n[SIM] --- INICIANDO TESTE 2: FILTRO DE GLITCH (VOTACAO MAJORITARIA) ---");

        rst = 1; spi_cs = 1; spi_sck = 0; spi_mosi = 0;

        #100 rst = 0;

        #200;



        // --- CENARIO A: ENVIO DE BIT VALIDO ---

        $display("[SIM] Passo 1: Enviando 1 bit normal (Pulso de clock estavel)");

        spi_cs = 0;    // Ativa comunicacao

        #200;

        spi_mosi = 1; // Dado
```

```
        #100;
        spi_sck = 1;  // Subida de clock valida (400ns)
        #400;
        spi_sck = 0;
        #100;

        // --- CENARIO B: INJECAO DE RUIDO (GLITCH) ---
        $display("[SIM] Passo 2: Injetando Glitch de 20ns no SCK (Ruido impulsivo)");
        // Este pulso dura apenas 20ns. Como o filtro exige 4 ciclos de 40ns (160ns),
        // a FPGA DEVE ignorar este pulso.
        spi_sck = 1;
        #20;           // Ruido muito rapido!
        spi_sck = 0;
        #380;          // Espera para estabilizacao

        // --- VALIDACAO ---
        $display("[SIM] Passo 3: Finalizando transacao e verificando Auditoria");
        #200;
        spi_cs = 1;    // Sobe CS para disparar a contagem final

        #100;
        // Se o filtro funcionou, o contador interno da FPGA deve ser 1, não 2.
        // O LED Verde (led[1]) estara APAGADO (1) porque 1 bit != 8 bits (Erro de Integridade esperado)
        // Mas o importante aqui e ver se o bit_cnt no Waveform ignorou o ruido.
        $display("[SIM] Verificacao: O bit_counter no GTKWave deve marcar exatamente 1 bit.");
        $display("[SIM] Se marcar 2, o Filtro de Glitch falhou.");

        #1000;
        $display("[SIM] --- FIM DO TESTE 2 --- \n");
        $finish;
    end
```

```
endmodule
```

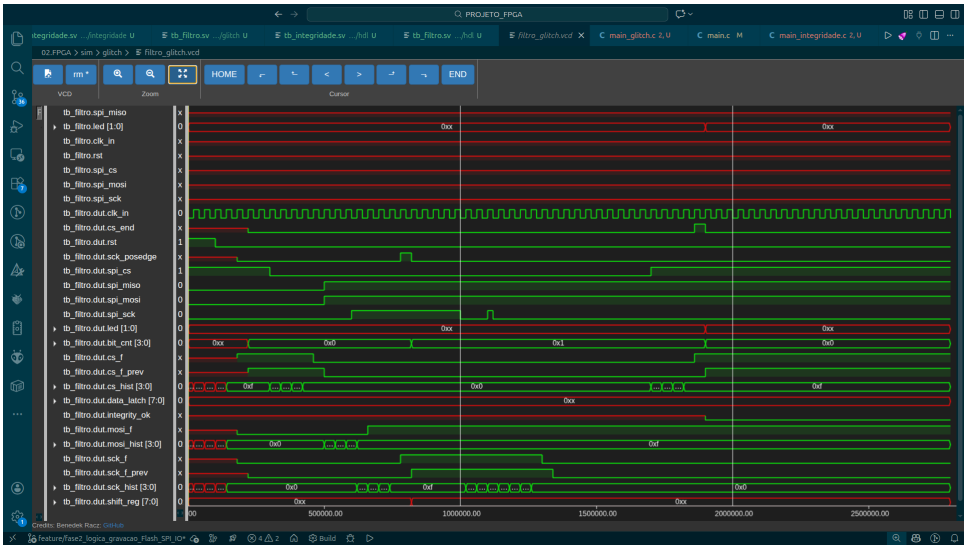
- **Apêndice D4:** 02.FPGA/hdl/tb_oversampling.sv (amostragem retardada).

```
`timescale 1ns/1ps



module tb_oversampling;

    // Declaração de sinais

    logic clk_in, spi_cs, spi_sck, spi_mosi, rst, spi_miso;

    logic [1:0] led;



    // Instanciação do módulo com Amostragem Retardada

    top dut (.*);



    // Gerador de Clock Principal (25MHz -> 40ns)

    initial clk_in = 0;

    always #20 clk_in = ~clk_in;



    initial begin

        // Configuração dos arquivos de onda para VS Code

        $dumpfile("oversampling.vcd");

        $dumpvars(0, tb_oversampling);



        // --- ESTADO INICIAL ---

        $display("\n[SIM] --- INICIANDO TESTE 3: OVERSAMPLING (AMOSTRAGEM NO CENTRO DO BIT) ---");

        rst = 1; spi_cs = 1; spi_sck = 0; spi_mosi = 0;

        #100 rst = 0;

        #200;



        // --- CENÁRIO: SIMULAÇÃO DE SKEW (ATRASO CRÍTICO) ---

        $display("[SIM] Passo 1: Iniciando transação com atraso proposital de sinal (Skew)");
```

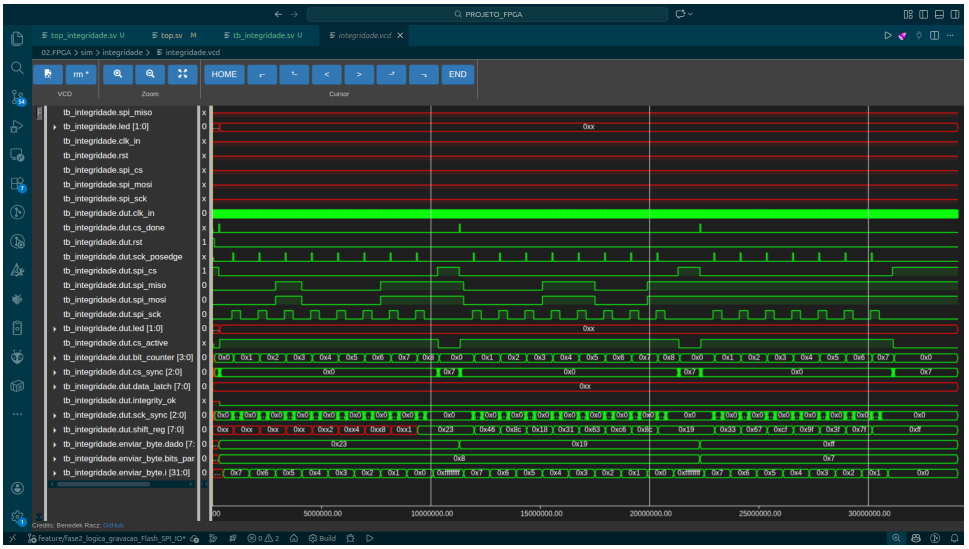
```
        spi_cs = 0;

        #200;



        // Simulando um problema físico real: O dado MOSI demora a subir

        // Em um sistema comum, isso causaria erro. No Teste 3, a FPGA deve esperar.

        $display("[SIM] Passo 2: Subindo o Clock (SCK), mas mantendo o Dado (MOSI) instável por 20ns");

        spi_sck = 1;

        #20;

        spi_mosi = 1; // O dado só fica correto 20ns APÓS o clock subir

        #380;



        $display("[SIM] Passo 3: Clock desce. A FPGA deve ter capturado o dado apenas no 4o ciclo de
clk_in.");

        spi_sck = 0;



        // --- FINALIZAÇÃO E ANÁLISE ---

        #500;

        spi_cs = 1; // Gatilho de auditoria



        $display("[SIM] Passo 4: Verificando integridade final.");

        #100;



        $display("[SIM] Analise no Waveform:");

        $display("      - O contador 'sample_delay_cnt' deve subir após a borda do SCK.");

        $display("      - A captura no 'shift_reg' deve ocorrer exatamente quando o contador atingir 4.");

        $display("      - Isso prova que ignoramos a instabilidade dos primeiros 20ns.");



        #1000;

        $display("[SIM] --- FIM DO TESTE 3 --- \n");

        $finish;

    end

endmodule
```

- **Apendice D5:** 02.FPGA/waveform.vcd (Dumping de sinais para análise no GTKWave).

```
========================================
   ESTRATEGIA 2: FILTRO DE GLITCH (VOTACAO)
========================================

--- TESTE 2: ESTABILIZAÇÃO VIA FILTRO DE GLITCH ---
[RP2040] TX: 26 C | RX (Loopback): 0 [ERRO - Ruído Detectado]
[RP2040] TX: 27 C | RX (Loopback): 0 [ERRO - Ruído Detectado]
[RP2040] TX: 28 C | RX (Loopback): 0 [ERRO - Ruído Detectado]
[RP2040] TX: 29 C | RX (Loopback): 0 [ERRO - Ruído Detectado]
[RP2040] TX: 30 C | RX (Loopback): 0 [ERRO - Ruído Detectado]
[RP2040] TX: 31 C | RX (Loopback): 0 [ERRO - Ruído Detectado]
[RP2040] TX: 32 C | RX (Loopback): 0 [ERRO - Ruído Detectado]
[RP2040] TX: 33 C | RX (Loopback): 0 [ERRO - Ruído Detectado]
[RP2040] TX: 34 C | RX (Loopback): 0 [ERRO - Ruído Detectado]
[RP2040] TX: 35 C | RX (Loopback): 0 [ERRO - Ruído Detectado]
```



```
========================================
   ESTRATEGIA 1: AUDITORIA DE BARRAMENTO
========================================

--- TESTE 1: AUDITORIA DE INTEGRIDADE + LOOPBACK ---
[RP2040] Enviado: 26 C | Recebido: 0 [ERRO DE BIT]
-> Observe na FPGA: LED Verde (Integridade) e Vermelho (Limite >30)

[RP2040] Enviado: 27 C | Recebido: 0 [ERRO DE BIT]
-> Observe na FPGA: LED Verde (Integridade) e Vermelho (Limite >30)

[RP2040] Enviado: 28 C | Recebido: 0 [ERRO DE BIT]
-> Observe na FPGA: LED Verde (Integridade) e Vermelho (Limite >30)

[RP2040] Enviado: 29 C | Recebido: 0 [ERRO DE BIT]
-> Observe na FPGA: LED Verde (Integridade) e Vermelho (Limite >30)

[RP2040] Enviado: 30 C | Recebido: 0 [ERRO DE BIT]
-> Observe na FPGA: LED Verde (Integridade) e Vermelho (Limite >30)

[RP2040] Enviado: 31 C | Recebido: 0 [ERRO DE BIT]
-> Observe na FPGA: LED Verde (Integridade) e Vermelho (Limite >30)

[RP2040] Enviado: 32 C | Recebido: 0 [ERRO DE BIT]
-> Observe na FPGA: LED Verde (Integridade) e Vermelho (Limite >30)

[RP2040] Enviado: 33 C | Recebido: 33 [LINK OK]
-> Observe na FPGA: LED Verde (Integridade) e Vermelho (Limite >30)

[RP2040] Enviado: 34 C | Recebido: 34 [LINK OK]
-> Observe na FPGA: LED Verde (Integridade) e Vermelho (Limite >30)

[RP2040] Enviado: 35 C | Recebido: 35 [LINK OK]
-> Observe na FPGA: LED Verde (Integridade) e Vermelho (Limite >30)

[RP2040] Enviado: 25 C | Recebido: 25 [LINK OK]
-> Observe na FPGA: LED Verde (Integridade) e Vermelho (Limite >30)

[RP2040] Enviado: 26 C | Recebido: 26 [LINK OK]
-> Observe na FPGA: LED Verde (Integridade) e Vermelho (Limite >30)
```



```
==========================================
   ESTRATEGIA 3: OVERSAMPLING & SKEW
==========================================

--- TESTE 3: PRECISÃO VIA AMOSTRAGEM NO CENTRO DO BIT ---
Objetivo: Garantir que o LED Vermelho siga a regra > 30 com perfeição.
[RP2040] Temp Enviada: 26 C | Retorno FPGA: 0 [ERRO DE SINCRONISMO]
>> Esperado na FPGA: LED Vermelho APAGADO
------------------------------------------------
[RP2040] Temp Enviada: 27 C | Retorno FPGA: 0 [ERRO DE SINCRONISMO]
>> Esperado na FPGA: LED Vermelho APAGADO
------------------------------------------------
[RP2040] Temp Enviada: 28 C | Retorno FPGA: 0 [ERRO DE SINCRONISMO]
>> Esperado na FPGA: LED Vermelho APAGADO
------------------------------------------------
[RP2040] Temp Enviada: 29 C | Retorno FPGA: 0 [ERRO DE SINCRONISMO]
>> Esperado na FPGA: LED Vermelho APAGADO
------------------------------------------------
[RP2040] Temp Enviada: 30 C | Retorno FPGA: 0 [ERRO DE SINCRONISMO]
>> Esperado na FPGA: LED Vermelho APAGADO
------------------------------------------------
[RP2040] Temp Enviada: 31 C | Retorno FPGA: 0 [ERRO DE SINCRONISMO]
>> Esperado na FPGA: LED Vermelho ACESO (Alerta)
------------------------------------------------
[RP2040] Temp Enviada: 32 C | Retorno FPGA: 0 [ERRO DE SINCRONISMO]
>> Esperado na FPGA: LED Vermelho ACESO (Alerta)
------------------------------------------------
```