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)

Autor: Marcio Barbosa
Programa: EmbarcaTech | Data: 12/01/2026
Orientador: Prof. Dr. David Ciarlini C. Freitas | Mentor(a): Luana Moura

1. RESUMO DA ETAPA

Este relatório tem o objetivo de mitigar problemas e riscos para o desenvolvimento de hardware e configuração remota (OTA) de sua configuração, validando a arquitetura base e soluções antes da implementação dos mecanismos de atualização.

O plano de projeto segue as competências técnicas de **Integração de Sistemas Heterogêneos**, compreensão profunda dos desafios de sincronização e comunicação entre domínios de *clock* diferentes (MCU e FPGA), **Design de IP Cores**, capacidade de traduzir especificações de protocolos (SPI) em *hardware* sintetizável (SystemVerilog) e **Fluxo de Design Moderno**, domínio de ferramentas de linha de comando (*CLI*) para síntese e verificação, superando a dependência de IDEs gráficas tradicionais.

Nesta etapa, foi projetado e integrado um *System-on-Chip (SoC)* composto pelo processador *Soft-Core PicoRV32*, memória RAM interna e um controlador de comunicação *SPI Slave*, elementos pré-requisitos para viabilizar a futura recepção de *bitstreams* via microcontrolador host.

O destaque técnico desta fase foi a validação bem-sucedida do fluxo de desenvolvimento baseado em ferramentas de código aberto (*Open Source Toolchain*) para criar uma métrica de avaliação dos resultados do fluxo sintetizado. A simulação e síntese física comprovaram que o SoC projetado é funcional, ocupando apenas 11% dos recursos lógicos do dispositivo e operando com margem segura de frequência (72 MHz). A proposta será a cada evolução do projeto apresentar resultados como este para cada etapa.

O projeto encontra-se estritamente dentro do cronograma planejado, com a plataforma de hardware pronta para a integração sistêmica na fase final.

2. ESTUDO E CORRELAÇÃO TÉCNICA

A arquitetura desenvolvida nesta Prova de Conceito (PoC) buscou o equilíbrio entre o processamento em hardware e o controle via software, alinhada às tendências em sistemas embarcados reconfiguráveis. Portanto, não seguiu a prática de uso de soluções Lattice, mas buscou outras soluções adotadas que permitisse o uso de linhas de comando e configurações de soluções *open source*. Segue, portanto, o resultado do estudo com evidências para a viabilidade técnica do projeto.

2.1. Arquiteturas Híbridas em Sistemas Críticos:

A estratégia de utilizar um microcontrolador externo (RP2040) para gerenciar a configuração de uma FPGA é amplamente validada em cenários que exigem alta disponibilidade, como o setor aeroespacial.

- **Referência:** Viel e Zeferino (2017) - *"A Module for Remote Reconfiguration of FPGAs in Satellites"*.
- **Correlação Técnica:** Os autores propõem o uso de um microcontrolador para realizar o *scrubbing* (reparação) e a reconfiguração remota de FPGAs em órbita via interfaces seriais.

De forma análoga, este projeto implementa um controle de configuração no MCU. Enquanto a aplicação aeroespacial foca em radiação, este projeto adapta a arquitetura para o contexto de IoT e atualização remota (OTA), demonstrando que a segregação entre controle (MCU) e processamento (FPGA) é a abordagem mais robusta para sistemas atualizáveis em campo.

2.2. Validação do Uso de *Soft-Core* para Gestão Auxiliar (*Housekeeping*)

A decisão de integrar o processador *Soft-Core* PicoRV32 na FPGA não visa substituir o processamento de alto desempenho, mas sim delegar a ele as tarefas de controle e gestão do sistema, gerenciar o protocolo SPI e a interação com o RP2040, uma estratégia validada em projetos de referência.

- **Referência:** Projeto *Open MPW / Caravel SoC* (Google/Efabless, 2021).
- **Correlação Técnica:** Nesta referência, o PicoRV32 é utilizado especificamente como "Núcleo de Gestão" (*Management Core*). Sua função é supervisionar a inicialização do *chip*, configurar interfaces de entrada/saída e gerenciar a comunicação, enquanto a área principal do silício fica reservada para a aplicação do usuário.

A proposta é atuar com uma "interface inteligente", garantindo que a lógica de atualização e comunicação seja processada via *software* interno, sem comprometer os recursos lógicos da FPGA (LUTs) que permanecem disponíveis para futuras implementações.

2.3. Ferramentas de "Hardware Ágil" (*Agile Hardware*)

A escolha das ferramentas de síntese, *toolchain* de código aberto (*Yosys/Nextpnr*), fundamenta-se na sua validação científica quanto à precisão de roteamento e análise temporal para FPGAs comerciais (Shah et al., 2019). Esta confiabilidade técnica permitiu transformar o processo de compilação em uma etapa de auditoria da qualidade para codificação e síntese.

- **Referência:** Shah et al. (2019) - *"Yosys+nextpnr: An Open Source Framework..."* (IEEE FCCM).
- **Correlação Técnica:** O estudo demonstra que ferramentas abertas atingem qualidade de roteamento comparável a softwares proprietários. A metodologia viabiliza a avaliação imediata da saúde do projeto através de três indicadores críticos (KPIs) extraídos dos logs de síntese:

- **Integridade do Fluxo:** Confirmação de que a descrição de *hardware* é sintetizável e livre de violações de regras de *design* (DRC).
- **Eficiência de Área:** A quantificação exata do consumo de recursos (**LUTs**) valida a viabilidade de integrar o processador sem saturar o dispositivo.
- **Estabilidade Temporal:** Análise Estática de Temporização (*STA*) certifica uma frequência máxima (**Fmax**), garantindo margem de segurança (*Slack*) para operação estável.

Dessa forma, a ferramenta atua como um instrumento de acompanhamento técnico, assegurando que cada iteração do projeto apresente padrões de qualidade que possam ser mensurados e analisados.

3. HARDWARE (HDL): DESENVOLVIMENTO, INTEGRAÇÃO E EVOLUÇÃO

A implementação do hardware na FPGA Lattice ECP5 foi realizada utilizando a linguagem **SystemVerilog**, focando na modularidade. Abaixo detalha-se o desenvolvimento dos núcleos, a integração do sistema e a evolução das métricas de design.

3.1. Desenvolvimento do IP Core Original (SPI Slave)

O IP Core de comunicação serial SPI Slave. Este módulo é o responsável pela interface física com o microcontrolador externo (RP2040), permitindo o recebimento de comandos e dados.

Este núcleo foi projetado para solucionar o desafio crítico de Cruzamento de Domínios de Clock (***CDC - Clock Domain Crossing***). Como o sinal de clock do SPI (SCK) provém de uma fonte externa assíncrona ao clock do sistema da FPGA (25 MHz), uma leitura direta causaria metaestabilidade e erros de dados. O mecanismo opera armazenando o estado histórico do sinal sck em dois flip-flops em cascata.

- **Detector de Borda (*Edge Detector*)** para sincronizar o sinal externo: A lógica é armazenar o valor no estado atual e anterior e comparar se no estado anterior o sck era 0 ou 1. A detecção da borda de subida ocorre pela comparação lógica entre o estado atual e o anterior (sck_atual & ~sck_antigo), gerando um pulso síncrono (sck_rise) com duração de exato um ciclo de clock da FPGA. Isso transforma a transição externa em um evento determinístico interno, permitindo a amostragem segura dos bits.
- **Máquina de Estados (FSM):** O controle de fluxo é gerido por uma FSM de 3 estados (IDLE, RECEIVE, DONE), assegurando que o processador seja notificado quando um byte completo (8 bits) for recebido e validado.

Trecho de Código (Sincronização e FSM):
1. Lógica de Sincronização e Captura (Arquivo: spi_slave.sv)

```
// Sincroniza o SCK externo (RP2040) com o Clock da FPGA
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 sinal externo
        sck_antigo <= sck_atual;  // Armazena o estado passado
    end
end
assign sck_rise = sck_atual & ~sck_antigo; // Detecta a borda de subida

// 2. Máquina de Estados (Arquivo: spi_slave.sv)
typedef enum logic [1:0] {
    IDLE,        // cs = 1 -> Esperando
    RECEIVE, // cs = 0 -> Recebendo bits
    DONE        // Byte completo
} state_t;

state_t estado_atual;
logic [2:0] bit_count;
logic [7:0] shift_reg;

always_ff @(posedge clk or negedge rst) begin
    if(!rst) begin
        estado_atual <= IDLE;
        bit_count <= 0;
        data_valid <= 0;
    end else begin
        case (estado_atual)
            IDLE: begin
                bit_count <= 0;
                data_valid <= 0;
                if (cs == 1'b0) begin
                     estado_atual <= RECEIVE;   // Se o CS baixar (0)
              end
            end

            RECEIVE: begin
                if (cs == 1'b1) begin
                    estado_atual <= IDLE;  // Reset se CS subir
                end else begin
                    // Amostragem segura usando o pulso sincronizado
                    if (sck_rise) begin  //Detector de borda muda
                        shift_reg <= {shift_reg[6:0], mosi}; // Deslocamento na fila
                        bit_count <= bit_count + 1;
                        // 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) estado_atual <= DONE;
                    end
                end
            end

            DONE: begin
                data_out <= shift_reg;   // Disponibiliza o byte
                data_valid <= 1;            // Sinaliza ao RISC-V
                estado_atual <= IDLE;
            end
        endcase
```

```
    end
end
```

3.2. Integração ao SoC (System-on-Chip)

A integração entre o processador e os periféricos foi estabelecida através de um Mapa de Memória. A Figura 1 demonstra o "contrato" de comunicação onde o Hardware expõe endereços físicos que são manipulados pelo Firmware através de ponteiros. O sistema utiliza o processador *Soft-Core* **PicoRV32**, configurado para otimização de área. A arquitetura do SoC baseia-se em um barramento compartilhado com a CPU sendo responsável pelos periféricos (Memória e SPI).

- **Mapa de Memória (Memory Map):** Foi desenvolvida uma lógica de decodificação de endereços no módulo de topo (top.sv). Utilizando um multiplexador combinacional, o sistema roteia as transações de leitura/escrita baseando-se no prefixo do endereço de 32 bits:
    - **0x0000_XXXX:** Acessa a Memória RAM interna (Firmware/Dados), 1KB de RAM (Endereço Base: 0x00000000).
    - **0x0001_XXXX:** Acessa o IP Core SPI Slave (Leitura de comandos externos). IP Core proprietário para recepção de dados
    - **0x0002_XXXX:** Acessa o registrador de LEDs (Saída de depuração). Registrador de saída para depuração visual.

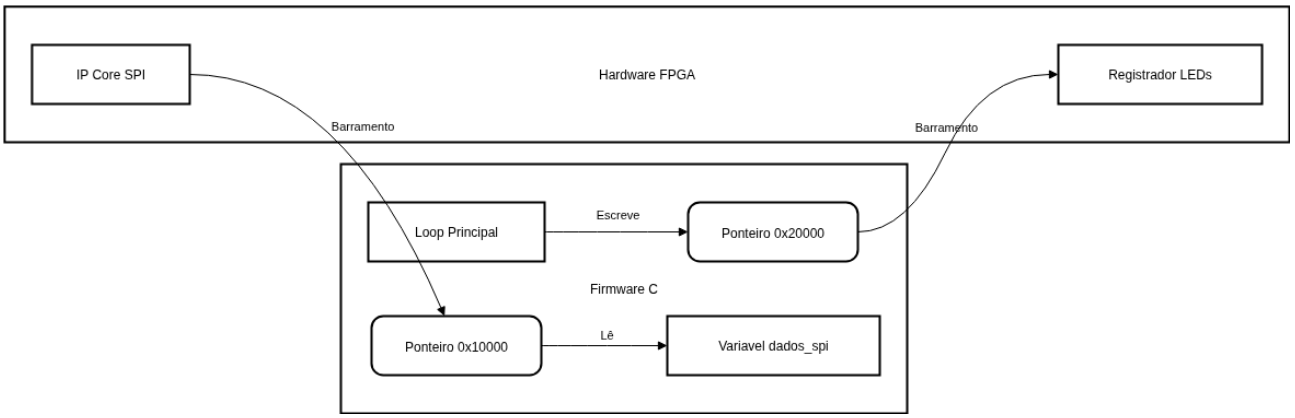

Figura 1: Diagrama de interação Hardware-Software via Memory-Mapped I/O.

Trecho de Código (Integração e Decodificação):

```
//Multiplexador Combinacional - Decodificador (Always Comb)
        // Mapa de memoria
        // 0x0000_xxxx -> memoria RAM
        // 0x0001_xxxx -> SPI Slave
// Lógica de Integração do Barramento (Arquivo: top.sv)
always_comb begin
   // Decodificador de Endereços
   //if(mem_addr_in[31:16] == 16'h0001) begin // não funciona erro no icarus | não descobri o porque
   if(addr_prefixo == 16'h0001) begin
```

```
      // Roteia para o IP Core SPI: Dados + Bit de Validade
      mem_rdata_in = {23'd0, spi_valid, spi_data_out};
   end else begin
      // Roteia para a Memória RAM
      mem_rdata_in = ram_rdata;
   end
end

// Instanciação do Processador RISC-V | PicoRV32 (Otimizado para área, sem FPU).
// 2. Instanciar o processador
picorv32 #(
   .PROGADDR_RESET(32'h0000_0000), //Memoria RAM
   .STACKADDR(32'h0000_0200)             //Stack Pointer
) cpu (
   .clk       (clk_in),
   .resetn    (rst_in), //ativo baixo
   .mem_valid  (mem_valid_in),
   .mem_ready  (mem_ready_in),
   .mem_addr   (mem_addr_in),
   .mem_wdata  (mem_wdata_in),
   .mem_rdata  (mem_rdata_in),
   .mem_wstrb  (mem_wstrb_in)
);
```

3.3. Evolução do Design e Otimizações

Em comparação à proposta inicial, o design evoluiu significativamente em termos de eficiência e ferramental. A migração para **Toolchain Open Source** pelo fluxo **Yosys + Nextpnr** permitiu maior controle sobre a síntese, facilitando a depuração através de logs textuais detalhados, divididos em dois domínios que convergem na gravação da FPGA (Figura 2):

1. **Fluxo de Software (*Cross-Compilation*):** Configurou-se o ***RISC-V GNU Toolchain*** para compilar código C/Assembly em uma máquina *host* (x86) gerando binários para o alvo embarcado (RV32I). O processo inclui a extração de binários puros (*Binary Stripping*) para gravação na memória.

2. **Fluxo de Hardware (*Synthesis & Implementation*):** Validou-se o uso do *Yosys* para a conversão de RTL em *netlist*, e do *Nextpnr-ecp5* para o mapeamento físico e geração do *bitstream*.

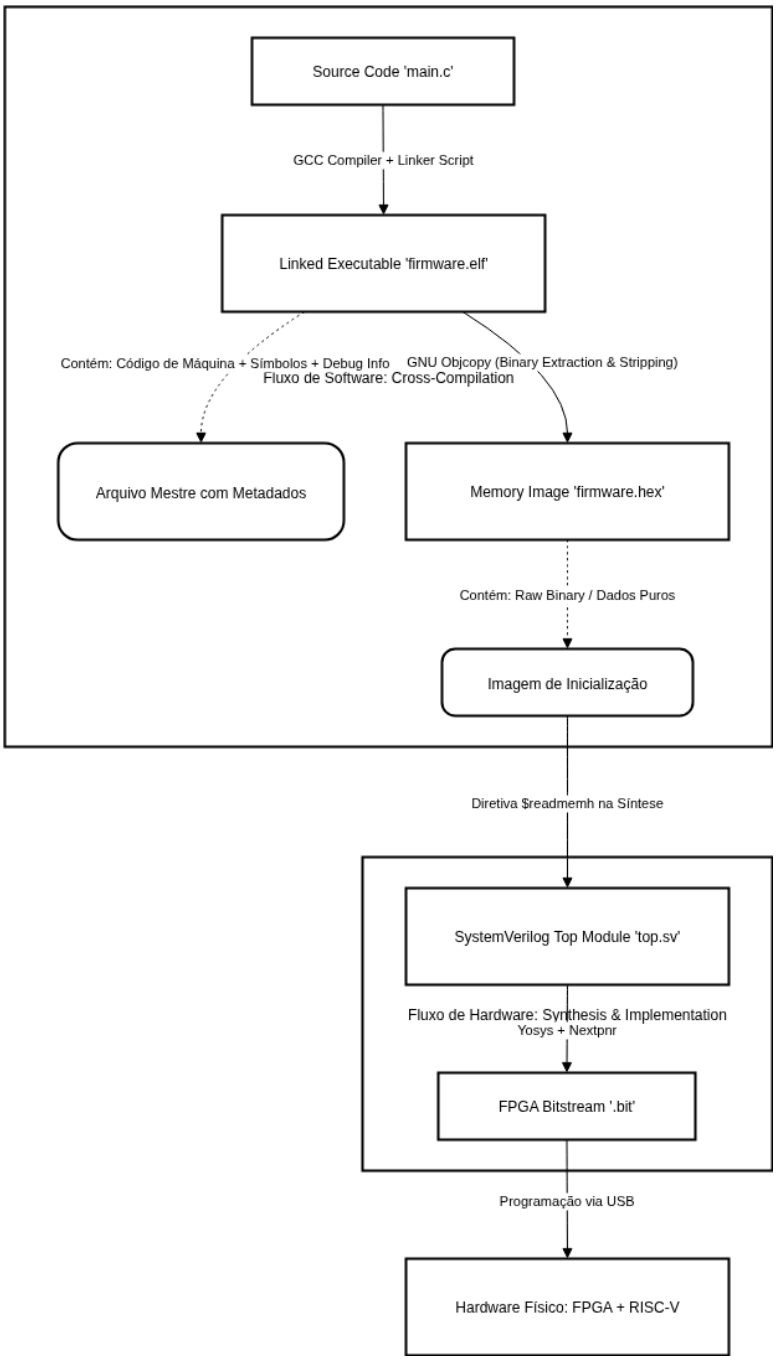

Figura 2: Fluxograma do processo de Compilação Cruzada e Síntese de Hardware implementado.

4. TESTBENCHS: SIMULAÇÕES E RESULTADOS OBTIDOS

A validação funcional do sistema foi realizada em ambiente de simulação RTL (*Register Transfer Level*) antes da síntese física. O objetivo principal foi garantir que o *IP Core SPI* fosse capaz de sincronizar sinais externos assíncronos e reconstruir os dados corretamente.

4.1. Metodologia de Verificação

Desenvolveu-se um Testbench dedicado (tb_spi_slave.sv) que atua como o "Mestre do Sistema", emulando o comportamento do microcontrolador RP2040. O Testbench injeta estímulos nos pinos SPI (sck, mosi, cs) da FPGA e monitora a reconstrução (loop de bits) do byte internamente.

Trecho de Código (Integração e Decodificação):

```
// Lógica de Integração do Testbench (Arquivo: tb_top.sv)
`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
   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 conexões
   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 le
        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
```

4.2. Resultados Visuais (Evidências)

A análise das formas de onda (Figura 3) comprovou a robustez da lógica de Clock Domain Crossing (CDC) e da Máquina de Estados:

- **Sincronização de Clock (CDC)**: Observando a linha DUT.sck_rise, acionada exatamente na borda de subida do sinal sck_in (linha vermelha). Isso comprova que o detector de borda interno está convertendo o sinal externo assíncrono em pulsos síncronos seguros para a FPGA.
- **Leitura dos bits (Shift Register)**: As linhas DUT.bit_count e DUT.shift_reg demonstram a construção sequencial do dado. O contador incrementa de 0 a 8 passo-a-passo, enquanto o registrador acumula os bits recebidos.
- **Resultado Final**: Na linha superior (data_out_o), o valor permanece 0x00 durante a transmissão e altera-se para 0xa5 imediatamente após o oitavo bit, validando a recepção correta do pacote enviado pelo testbench.

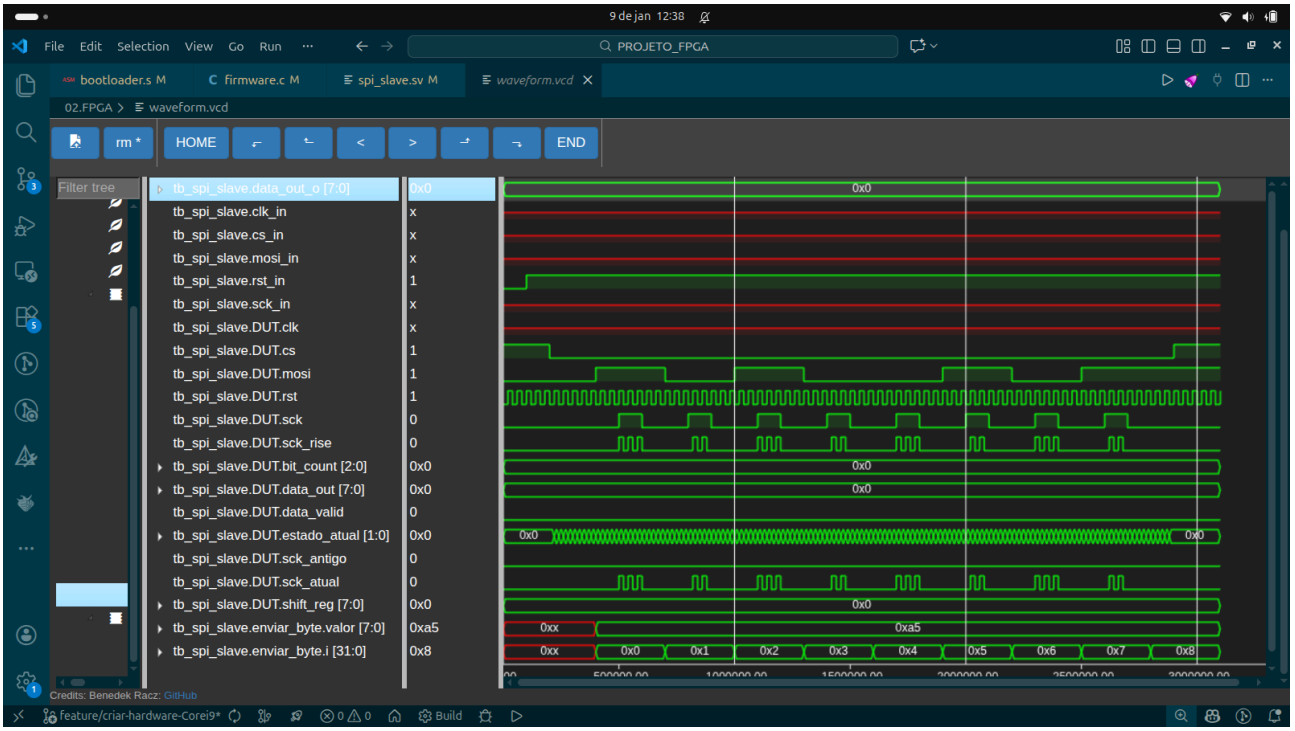

Figura 3: Sinais (Waveform) demonstrando a recepção e serialização do dado.

4.3. Métricas de Performance (KPIs)

A análise pós-roteamento (Place & Route) demonstrou que a arquitetura atual é altamente eficiente, deixando ampla margem para a implementação futura da lógica OTA:

1. **Ocupação de Área:** O SoC completo ocupa apenas **11% das LUTs** (Look-Up Tables) disponíveis na FPGA ECP5 (2.734 de 24.288), validando a escalabilidade do sistema.

2. **Timing:** A frequência máxima de operação atingida foi de **72.18 MHz**, superando em quase 3x o clock base do sistema (25 MHz), o que garante robustez contra falhas de temporização (*Positive Slack*).

Esta técnica será utilizada a cada etapa de evolução do SoC em etapas futuras. Segue algumas evidências de auditoria que correspondem a performance conforme comandos realizados.

Etapa 1: Auditoria de Memória (Evidência Crítica). Para validar se o código realmente iniciava no endereço zero (Reset Vector), utilizou-se a ferramenta objdump.

Comando Executado:

```
riscv64-unknown-elf-objdump -d firmware.elf | head -n 15
```

Evidência de Sucesso (Log): O resultado abaixo comprova que a seção .text inicia em 0x00000000, alinhada com o hardware do processador:

```
firmware.elf:     file format elf32-littleriscv
```

```
Disassembly of section .text:
00000000 <_start>:
   0:   00000293            li      t0,0
   4:   00000097            auipc   ra,0x0
```

Etapa 2: Execução do Fluxo de Place & Route

Comando Executado:

```
nextpnr-ecp5 --25k --package CSFBGA285 --speed 6 --json hardware.json --lpf pins.lpf --textcfg hardware.config
```

Etapa 3: Análise dos Indicadores de Qualidade (KPIs). Os logs finais desta etapa constituem a prova técnica da eficiência do projeto.

Evidência de Ocupação (Área):

```
Info: Device utilisation:
Info:          TRELLIS_COMB:  2734 / 24288 (11%)
Info:            TRELLIS_IO:    13 /   197 ( 6%)
Info:          MULT18X18D:     0 /    28 ( 0%)
```

Análise*:* O baixo uso de LUTs (11%) e zero uso de DSPs valida a escolha do núcleo PicoRV32 para aplicações de baixo custo.

Evidência de Temporização (Timing):

```
Info: Max frequency for clock 'clk': 72.18 MHz (PASS at 25.00 MHz)
Info: Clock 'clk' has 0.00 ns negative slack.
```

5. SOFTWARE (C/RISC-V): AVANÇOS DO CÓDIGO EMBARCADO

Devido à ausência de Sistema Operacional, desenvolveu-se uma camada de *software* de baixo nível. A validação desta etapa focou no ciclo de vida da inicialização do processador, conforme a Figura 4, a lógica de *boot* segue uma sequência crítica e portanto etapas deverão ser realizadas até a entrega final do projeto:

1. **Reset de Hardware:** O controlador força o *Program Counter* (PC) para 0x00000000.
2. **Inicialização (start.s):** O código Assembly configura o *Stack Pointer* (SP), preparando a memória para a execução de funções.
3. **Aplicação (main.c):** O fluxo é desviado para a função principal em C, que executa o *loop* de controle.

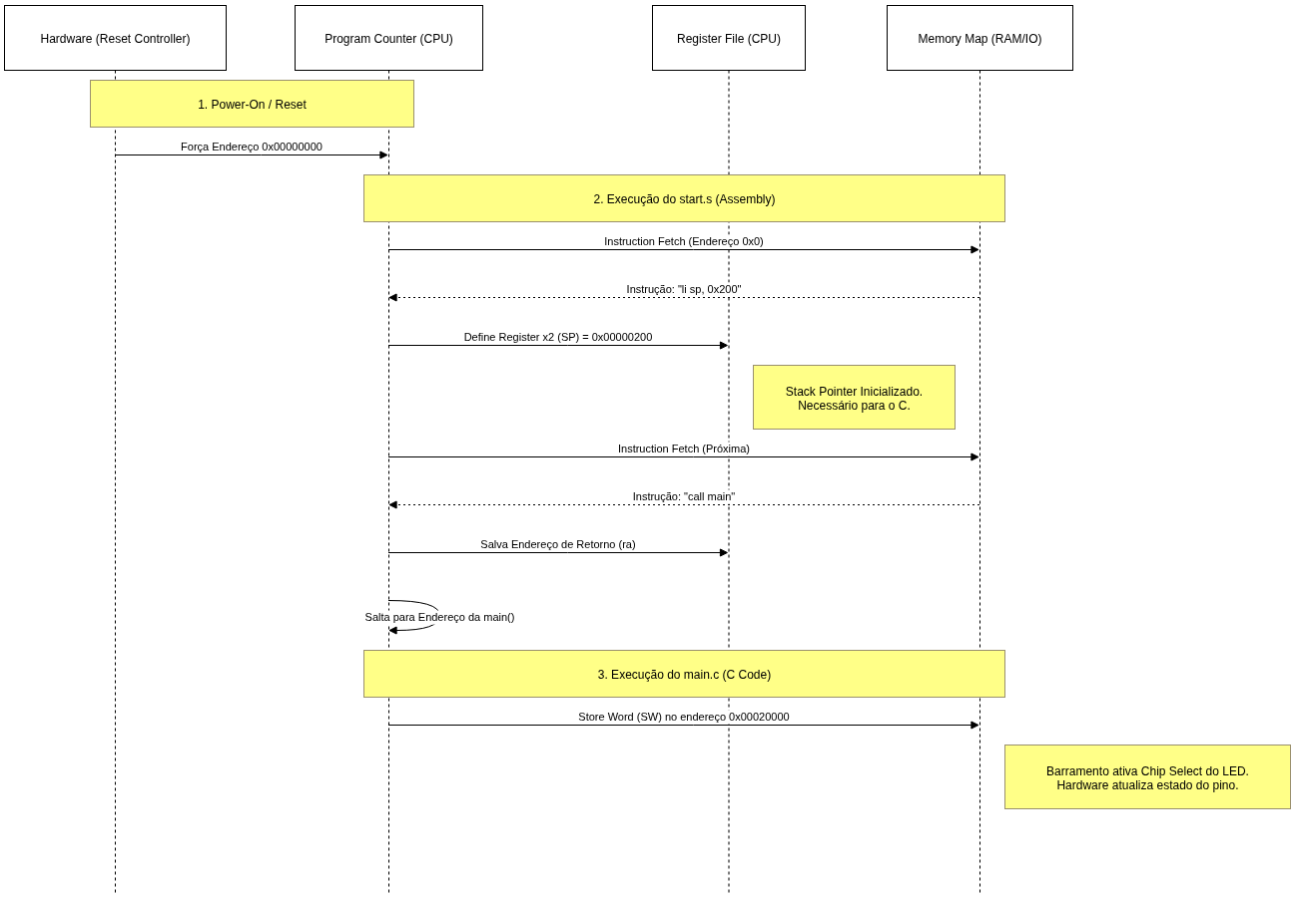

Figura 4: Diagrama de sequência do processo de Boot e execução Fetch-Decode-Execute.

O código foi compilado utilizando a toolchain **RISC-V GCC**, gerando um arquivo hexadecimal (firmware.hex) que é inicializado na memória RAM da FPGA durante a síntese.

5.1. Inicialização e Bootloader (Assembly)

Para garantir a correta execução do processador PicoRV32, foi implementado um bootloader mínimo em Assembly (bootloader.s). Este código é posicionado no endereço de reset (0x00000000) e tem como função crítica configurar o ponteiro de pilha (Stack Pointer - SP) antes de transferir o controle para a função principal em C.

A definição correta do Stack Pointer foi essencial para resolver os problemas de estouro de pilha (Stack Overflow) identificados durante os testes.

```
// Trecho do arquivo bootloader.s (Inicialização)
.section .init
.global _start
_start:
    // Configura o Stack Pointer para o topo da RAM (Endereço 0x00000200)
    // Isso garante 512 Bytes para variáveis locais e chamadas de função
    li sp, 0x00000200
    // Salta para a função main() do código C
    call main
    // Loop infinito de segurança caso o main retorne
loop:
```

```
j loop
```

5.2. Abstração de Hardware e Drivers (HAL)

A comunicação entre o software e os periféricos (IP Cores) é realizada através do mapeamento direto de memória (Memory-Mapped I/O). Definiram-se macros que convertem os endereços físicos em ponteiros desreferenciados, permitindo a leitura e escrita direta nos registradores de hardware.

- REG_LEDS (0x00020000): Registrador de escrita para controle dos LEDs.
- REG_SPI (0x00010000): Registrador de leitura para recebimento de dados seriais.

```
// Definição dos Ponteiros de Hardware firmware.c (Memory Map)
#define REG_LEDS  ((volatile uint32_t*)0x00020000)
#define REG_SPI  ((volatile uint32_t*)0x00010000)
```

5.3. Lógica da Aplicação Principal

A função main() implementa um laço infinito (super-loop) que gerencia o estado do sistema. A lógica de controle verifica continuamente o registrador SPI. Caso nenhum dado válido seja recebido (valor 0), o sistema entra em "Modo Demo", incrementando um contador binário nos LEDs para indicar atividade da CPU. Ao detectar dados válidos vindos do RP2040, o valor é imediatamente refletido na saída. Utilizou-se o qualificador volatile para garantir que o compilador não otimize os acessos ao barramento.

Esta lógica valida a cadeia completa de processamento: o sinal elétrico entra na FPGA, é serializado pelo IP Core, lido pelo processador via barramento e processado pelo software C.

```
// Lógica de Polling e Atuação (firmware.c)
/* O bootloader.s pula para cá */
void main() {
    uint32_t contador = 0;
    uint32_t dados_spi = 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. 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 */
        }
        /* 3. Atraso para o olho humano perceber a mudança */
        /* Clock 25MHz -> delay 500000 gera aprox 100-200ms */
        delay(500000);
    }
}

/* Função auxiliar de delay (Ciclos de CPU) */
void delay(uint32_t count) {
```

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

6. PROBLEMAS ENCONTRADOS E SOLUÇÕES ADOTADAS

O desenvolvimento enfrentou desafios técnicos quanto à integração de sistemas heterogêneos e restrições de hardware.

6.1. Incompatibilidade de Ambiente de Simulação (GLIBC)

- **Descrição do Desafio:** Durante a configuração da *toolchain*, o visualizador de ondas (GTKWave) falhava na inicialização, impedindo a validação visual do testbench.
- **Causa Técnica:** Conflito de versões entre a biblioteca padrão C (glibc) do sistema operacional host e as bibliotecas exigidas pelos binários pré-compilados das ferramentas de FPGA.
- **Solução Adotada:** Implementação de *scripts* de ambiente (shell scripts) para isolar a execução das ferramentas, configurando a variável LD_LIBRARY_PATH localmente. Isso garantiu que o simulador carregasse as dependências corretas sem comprometer o sistema operacional base.

6.2. Erros de Síntese por Sinais Flutuantes

- **Descrição do Desafio:** O processo de *Place & Route* (Nextpnr) abortava a geração do bitstream, reportando erros fatais em redes lógicas.
- **Causa Técnica:** O processador PicoRV32 expõe diversas interfaces de depuração e interrupção (ex: trace_data, eoi). Como estes sinais não foram utilizados na etapa atual, o sintetizador interpretou os fios desconectados como falhas de integridade do design.
- **Solução Adotada:** Aplicação de diretivas de síntese no código SystemVerilog para "aterrar" ou ignorar explicitamente as portas não utilizadas, além da revisão do arquivo de restrições (pins.lpf) para garantir que apenas os pinos físicos reais fossem mapeados.

```
# 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)
```

6.3. Estouro de Pilha (Stack Overflow) em Bare Metal

- **Descrição do Desafio:** Nas primeiras execuções em hardware, o processador travava imediatamente após o *boot*, apresentando comportamento errático nos LEDs.
- **Causa Técnica:** A restrição severa de memória RAM interna (configurada inicialmente para 1KB para economizar recursos da FPGA) resultou em um conflito de memória. O *Stack Pointer* (Ponteiro de Pilha) estava crescendo sobre a região de dados globais do firmware, corrompendo variáveis de execução.
- **Solução Adotada:**
    1. Refatoração do *Linker Script* para segregar rigidamente as seções .text (código) e .data (variáveis).
    2. Ajuste manual do endereço de topo da pilha no arquivo bootloaders.s (li sp, 0x00000200) para garantir margem de segurança.
    3. Otimização do código C para eliminar dependências de bibliotecas padrão pesadas, mantendo o binário enxuto.

7. PRÓXIMOS PASSOS (PLANEJAMENTO)

Com a arquitetura de hardware (SoC) e o fluxo de software (toolchain) validados em ambiente de simulação, o projeto encerra a Fase 2 e avança para a Fase 3: Integração Sistêmica e Entrega Final.

O foco para o próximo período (Janeiro e Fevereiro/2026) desloca-se da validação lógica para a validação física e implementação dos mecanismos de atualização remota.

7.1. Objetivos Prioritários para a Fase 3

O planejamento para a etapa final concentra-se em três pilares de execução:

1. **Integração Física (Camada de Hardware):** Conexão elétrica entre o microcontrolador RP2040 (Mestre) e a FPGA (Escrava) através do barramento SPI, validando os níveis lógicos (3.3V) e a integridade do sinal em alta frequência.
2. **Implementação de Periféricos (Requisito Obrigatório):** Desenvolvimento dos drivers finais para leitura do sensor de temperatura (AHT10 via I2C no RP2040) e implementação do IP Core de varredura para o teclado matricial na FPGA.
3. **Mecanismo OTA (Funcionalidade Central):** Desenvolvimento da lógica de *software* no RP2040 capaz de receber um novo *bitstream* e gravá-lo na memória Flash externa da FPGA, efetivando a atualização remota.

7.2. Cronograma de Execução Atualizado

Abaixo detalha-se o cronograma de atividades para a reta final do projeto, assegurando a entrega do protótipo funcional e dos materiais de divulgação dentro do prazo limite.

FASE 1: Entrega Parcial (Infraestrutura e Núcleo) - PRAZO: 12/01/26			
ID	**Atividade / Tarefa**	**Data Conclusão**	**Status**
1	Configuração Ambiente Software (SDK Pico + FreeRTOS)	01/12/2025	Concluído
2	Firmware RP2040: Task Básica (Blinky)	01/12/2025	Concluído
3	Firmware RP2040: Comunicação USB-Serial (CDC)	09/12/2025	Concluído
4	**Software PC: Script Python de Teste**	**Jan/2026**	***Planejado***
5	Configuração Toolchain FPGA (Yosys/Nextpnr)	10/12/2025	Concluído
6	Integração Core RISC-V (PicoRV32)	10/12/2025	Concluído
7	Desenvolvimento IP Core: SPI Slave	15/12/2025	Concluído
8	Integração SoC (Top Module: CPU + SPI + Mem)	15/12/2025	Concluído
9	Simulação Funcional do SoC (GTKWave)	16/12/2025	Concluído
10	Síntese Física e Geração de Bitstream	16/12/2025	Concluído
11	Configuração Toolchain Firmware (GCC RISC-V)	15/12/2025	Concluído
12	Codificação Firmware (C e Assembly)	16/12/2025	Concluído
13	Geração de Imagem de Memória (.hex)	16/12/2025	Concluído
14	**Integração Física em Bancada**	**Jan/2026**	***Planejado***
15	**Teste Funcional em Hardware (SPI Real)**	**Jan/2026**	***Planejado***
16	Documentação Técnica Hardware (HDL e Arquitetura)	21/12/2025	Concluído
17	Documentação Técnica Software (Firmware e Fluxo)	21/12/2025	Concluído
18	Documentação Técnica Evidências (Prints e Imagens)	21/12/2025	Concluído
19	Revisões e ajuste ao modelo Luana	21/12/2025	Concluído
20	Compilação do Relatório Final (RED)	21/12/2025	Concluído
FASE 2: Entrega Final (Produto e OTA) - PRAZO: 08/02/26			
ID	**Atividade / Tarefa**	**Previsão**	**Status**
21	**RP2040: Lógica de Gravação Flash (Mecanismo OTA) - Teste 1**	**Jan/2026**	***Planejado***
22	RP2040: Integração Sensor AHT10 (I2C)	Jan/2026	*Planejado*
23	FPGA: Desenvolvimento IP Teclado Matricial	Jan/2026	*Planejado*
24	**RP2040: Lógica de Gravação Flash (Mecanismo OTA) - Teste 2**	**Jan/2026**	***Planejado***
25	Integração Total do Sistema	Jan/2026	*Planejado*
26	Escrita: Fundamentação Teórica	Fev/2026	*Planejado*
27	Escrita: Detalhamento do Processo OTA	Fev/2026	*Planejado*
28	Gravação do Vídeo de Apresentação	Fev/2026	*Planejado*
29	Revisão Final e Formatação	Fev/2026	*Planejado*

7.3. Repositório de Código e Versionamento

Para permitir a auditoria do código-fonte desenvolvido (RTL, Firmware e Scripts de Build), todo o projeto foi versionado e disponibilizado publicamente.

Link de Acesso: https://github.com/maborbaa/PROJETO_FPGA

8. REFERÊNCIAS

1. BARBOSA, Marcio. **Prova de Conceito de um Sistema Híbrido para Atualização Remota da Configuração (Bitstream) em Plataformas Embarcadas Baseadas em FPGA**. 2025. Pré-Projeto (Residência Tecnológica em Sistemas Embarcados) – Instituto Federal de Educação, Ciência e Tecnologia do Ceará (IFCE), Programa EmbarcaTech, 2025.

2. 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.

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

4. 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.

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

6. LATTICE SEMICONDUCTOR. **ECP5 and ECP5-5G Family Data Sheet**. Disponível em: http://www.latticesemi.com. Acesso em: dez. 2025.

7. 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.

8. 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.

9. 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.

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